Universidade do Minho
Escola de Engenharia

José Miguel Alves Pires

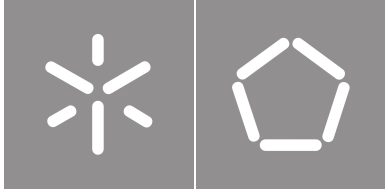
Trustworthy Open-Source Reference Software Stack for UAV applications

October, 2025

Trustworthy Open-Source Reference Software Stack for UAV applications

Jose Pires

UMinho | 2025



Universidade do Minho
Escola de Engenharia

José Miguel Alves Pires

Trustworthy Open-Source Reference Software Stack for UAV applications

Master Thesis
Master in Industrial Electronics and Computers Engineering
Specialization in Embedded Systems

Work developed under the supervision of:

Professor Doctor Sandro Pinto

October, 2025

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.2.5) [1].

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Sandro Pinto, for all the guidance, support, and technical expertise that was so valuable throughout this process.

I also would like to thank my lab colleagues Martins and David for all the help with the Bao hypervisor and for all the patience and wise advice, without which this work would not have been possible.

To my parents for always pushing me forward and for being the life example of character and sheer determination, a lamp lit on a stormy day. Thank you also for an upbringing full of debates and argumentation, which has undoubtedly contributed to mastering the challenges I took ahead, and for instilling in me the taste in good music which helped me throughout this journey, listening to the *Wish You Were Here*. Thank you for all the support, love, friendship, and for always believing in me, and sorry for my constant lack of time. To my sister, for always being so supportive and caring, helping out with your immense ingenuity and creativity, a mix between the engineer and the artist.

To my closest friends, Pedro and Chicão for taking me out to dinner, which is exactly what one needs sometimes. For all the support and the true friendship that survives even the end of times. To Pedro for giving me the opportunity of being a godfather to a wonderful girl, Matilde. I hope I made you proud too. To my beloved dog Pantufa for adopting me eleven years ago when I was going crazy with my first MSc thesis and never leaving my side.

And last, but most importantly, to my beautiful and amazing girlfriend, Rafaela, my one true love. We've faced so many challenges, the distance and the lack of time, but we strived, and, like a boat that emerges out of a storm, we came stronger and more vibrant, full of life. And now we can appreciate the petrichor from the summer rains together. Your unconditional love and support meant the world to me, from day one to today, and through the many more we'll face together. I hope you are proud of me, because this accomplishment is also yours. Love you so damn much!!!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Vizela, 3 de novembro de 2025

(Place) (Date)

(José Miguel Alves Pires)

”

"*Wir müssen wissen, Wir werden wissen.*
We must know, We shall know."

— David Hilbert, 1930
(mathematician)

To my parents and Rafaela
for all the love and support

Resumo

Pilha de software de referência para aplicações UAV

O rápido crescimento dos veículos aéreos não tripulados (*UAV*) impõe exigências de criticalidade mista: o controlo de voo (crítico) tem de coexistir com o *software* não-crítico da missão obedecendo aos limites de tamanho, peso, potência e custo (*SWaP-C*). As plataformas convencionais, *multi-board*, adicionam peso e latência nas comunicações e a separação por *hardware* é insuficiente, devido às interfaces partilhadas. Os esforços de consolidação em curso dependem de hipervisores proprietários ou assentam em implementações que dependem da platforma/aplicação ou que foram abandonados. Outras abordagens focam-se apenas na pilha de *software* da missão usando contentores ou componentes verificados formalmente. O contentor não garante o isolamento adequado, e ambos focam-se apenas na pilha da missão, deixando as comunicações e o piloto automático desprotegidos. Tais suposições são inseguras dada a extensa superfície de ataque dos *UAVs* e as vulnerabilidades reportadas nos pilotos automáticos de código aberto e sistemas operativos de tempo real comerciais. No presente trabalho concebemos e avaliamos uma consolidação de pilhas de software criticidade mista em *UAVs* recorrendo ao Bao, um hipervisor de particionamento estático de tipo I. Implementamos duas pilhas de software numa única plataforma (Raspberry Pi 4 + PilotPi): uma não supervisionada (*USPFS*), usada como base de comparação, e uma supervisionada (*SSPFS*), onde o PX4 (piloto automático) e uma aplicação de vídeo em tempo real (missão) executam como *guests* separados. Para partilhar periféricos com segurança, adicionamos ao Bao um mecanismo de supervisão por *mailbox* que medeia transações de *firmware* num barramento partilhado, prevenindo acessos indevidos entre domínios. As experiências demonstram que as falhas injetadas no *guest* não crítico provocam apenas a falha deste, enquanto o PX4 mantém o controlo e a *UAV* em voo. A avaliação realizada com o MiBench e nos testes de aplicação, indicam penalizações reduzidas no desempenho: 2% no escalonamento de tarefas do PX4 e na captura de vídeo. Os testes de voo demonstraram a robustez do controlo de posição e um aumento reduzido na carga do *CPU* (6%). O particionamento estático promove o isolamento entre domínios e o *cache coloring* mitiga a interferência entre eles. Em suma, um hipervisor de particionamento estático, como o Bao, permite consolidar as pilhas de *software* de voo e de missão numa única plataforma com impacto reduzido no desempenho, viabilizando sistemas confiáveis de criticidade mista em *UAV*s de código aberto.

Palavras-chave: UAV, Sistemas de criticidade mista, Hipervisor, Bao, PX4, Virtualização, SWaP-C

Abstract

Trustworthy Open-Source Reference Software Stack for UAV applications

The rapid growth of unmanned aerial vehicles (UAVs) brings mixed-criticality demands: safety-critical flight control must coexist with non-critical, resource-intensive mission software under strict size, weight, power, and cost (SWaP-C) limits. Conventional multi-board stacks add weight and link latency, and hardware separation alone is insufficient: compromises in non-critical nodes can cascade into critical functions via shared interfaces. Ongoing consolidation efforts either depend on closed-source hypervisors or rely on unmaintained, application-/platform-specific designs. Other approaches harden only the mission side using containerization or formally verified components, but shared-kernel containers cannot guarantee temporal, spatial, or fault isolation, and mission-only hardening leaves critical links and the autopilot unprotected. These assumptions are unsafe given UAVs' extensive attack surface, the bug density observed in open-source autopilots, and vulnerabilities reported in commercial real-time operating systems. This thesis designs and evaluates a trustworthy, open consolidation of mixed-criticality UAV workloads using Bao, a lightweight type-I hypervisor with static partitioning. We implement two stacks on Raspberry Pi 4 + PilotPi: an unsupervised single-platform flight stack (USPFS) – a baseline for comparison that co-locates mission and flight control without supervision – and a supervised single-platform flight stack (SSPFS) where PX4 (flight control) and a live video pipeline (mission) run as separate guests. To safely share peripherals, we add a mailbox-supervision mechanism to Bao that mediates firmware transactions over a shared bus, preventing unintended cross-domain access. Experiments show supervision-driven isolation: injected faults in the non-critical guest cause only that guest to fail, while the PX4 guest maintains control and keeps the UAV airborne. Offline benchmarking with the MiBench suite indicates small overheads, corroborated in application tests: PX4 task-scheduling ≈2% and camera frame-rate ≈2%. Real-flight tests demonstrate accurate position tracking and modest average CPU load (≈6%). Static partitioning bounds interference, and cache coloring further mitigates cross-domain effects. Overall, a small, open-source static-partitioning hypervisor, such as Bao, consolidates flight-control and mission workloads into a single platform with minimal overhead, offering a practical, open path to trustworthy mixed-criticality UAV systems, even in the face of large attack surfaces where component-level assurances alone are not sufficient.

Keywords: UAV, mixed-criticality systems, hypervisor, Bao, PX4, virtualization, SWaP-C

Contents

List of Figures

List of Listings

Acronyms

ABI Application Binary Interface *(p. 29)*
AC Alternating Current *(p. 17)*
AI Artificial Intelligence *(pp. 23–25)*
AIA Advanced Interrupt Architecture *(p. 11)*
AICS Automotive and Industrial Control Suite *(pp. 3, 5, 11, 63, 64, 70, 71, 85–87)*
API Application Programming Interface *(pp. 17, 28, 29, 37)*

BLDC Brushless Direct Current *(pp. 17, 19, 42)*
BOM Bill Of Materials *(p. 20)*
BSD Berkeley Software Distribution *(p. 21)*
BSP Board Support Package *(pp. 52, 59)*
BVLOS Beyond Visual Line-Of-Sight *(p. 13)*

CAD Computer-Aided Design *(p. 18)*
CAN Controller Area Network *(pp. 21, 25, 34)*
CMA Contiguous Memory Allocation *(p. 47)*
COTS Commercial Off-The-Shelf *(p. 20)*
CPU Central Processing Unit *(pp. 2–4, 9–11, 32, 41, 45–48, 55, 63, 64, 68, 71, 72, 74, 80, 85, 86)*
CSI Camera Serial Interface *(p. 43)*
CSP Context Security Policy *(p. 27)*

DC Direct Current *(pp. 17, 52)*
DDoS Distributed Denial Of Service *(p. 18)*
DMA Direct Memory Access *(pp. 2, 45, 47, 60, 75, 76, 87)*
DoS Denial of Service *(pp. 11, 18, 34)*
DTB Device Tree Blob *(pp. 52, 59)*

EEPROM Electrically Erasable Programmable Read-Only Memory *(p. 29)*

eMMC embedded Multi Media Card *(p. 25)*

ESC Electronic Speed Controller *(pp. 17, 42, 52, 56, 73)*

FAA Federal Aviation Administration *(p. 13)*

FCS Flight Control System *(pp. 17, 19)*

FIFO First In, First Out *(p. 73)*

FMU Flight Management Unit *(pp. 1, 19, 21, 24, 25, 29, 32, 34, 37–40, 42, 43, 55, 72, 74, 78, 81, 82, 87)*

FPGA Field-Programmable Gate Array *(pp. 23, 32, 33)*

FPS Frames Per Second *(pp. 48, 57, 64, 74, 76)*

FPV First-Person View *(pp. 18, 23, 28)*

GCS Ground Control Station *(pp. 17–19, 25, 27–29, 33, 34, 36–39, 42, 52, 55–58, 65, 70, 74, 78, 81, 82)*

GIC Generic Interrupt Controller *(pp. 11, 52, 59)*

GNSS Global Navigation Satellite System *(pp. 3, 17, 24)*

GPIO General Purpose Input/Output *(pp. 22, 45)*

GPL GNU Public License *(p. 21)*

GPOS General-Purpose Operating System *(pp. 18, 21, 28, 39, 87)*

GPS Global Positioning System *(pp. 17–19, 21–23, 25, 28, 29, 42, 44, 45, 52, 80)*

GPU Graphics Processing Unit *(pp. 43, 45, 46)*

GUI Graphical User Interface *(pp. 78, 81, 82)*

HAL Hardware Abstraction Layer *(pp. 28, 31)*

HAP High Altitude Platform *(pp. 14, 16)*

HD High-Definition *(p. 48)*

HDL Hardware Description Language *(p. 20)*

HFC Hydrogen Fuel Cell *(p. 14)*

HITL Hardware in the Loop *(pp. 17, 31)*

I/O Input/Output *(pp. 2, 11, 19, 21, 34, 40, 43, 68)*

I2C Inter-Integrated Circuit *(pp. 21, 24, 25, 50, 59, 60, 73)*

IDE Integrated Development Environment *(p. 21)*

IMU Inertial Measuring Unit *(pp. 17, 19, 21, 23, 24, 28, 45)*

IOMMU Input/Output Memory Management Unit *(pp. 2, 9)*

IPC Inter-Process Communication *(p. 10)*
ISA Instruction Set Architecture *(p. 10)*

LAN Local Area Network *(p. 38)*
LAP Low Altitude Platform *(pp. 14, 16)*
LCD Liquid Crystal Display *(p. 23)*
LiDAR Light Detection And Ranging *(pp. 17, 18, 25)*
LiPo Lithium Polymer *(pp. 14, 42, 52)*
LLC Last-Level Cache *(pp. 11, 70)*
LTE Long-Term Evolution *(pp. 2, 24)*

Mbps Megabits per second *(p. 49)*
MCS Mixed-Criticality System *(pp. 6, 7, 9, 10, 32)*
MCU Micro Controller Unit *(pp. 21, 23, 42)*
MMU Memory Management Unit *(p. 9)*

NASA National Aeronautics and Space Administration *(p. 13)*
NPU Neural Processing Unit *(p. 25)*

OOM Out Of Memory *(pp. 67, 68)*
OS Operating System *(pp. 8–10, 18, 27, 28, 32, 40, 41, 43, 49, 50, 53, 59, 66)*
OSH Open-Source Hardware *(pp. 20, 21, 28, 31)*
OSS Open-Source Software *(pp. 28, 29)*
OTA Over-The-Air *(p. 29)*

PCB Printed Circuit Board *(pp. 20, 21)*
PCIe Peripheral Component Interconnect Express *(pp. 3, 4, 36, 60, 64, 86)*
PID Proportional Integrative Derivative *(p. 19)*
PLIC Platform-Level Interrupt Controller *(p. 11)*
PMU Performance Monitor Unit *(p. 70)*
PV Photovoltaic *(p. 15)*
PWM Pulse-Width Modulation *(pp. 21, 24, 25, 42, 44, 56, 59)*

QoS Quality of Service *(p. 9)*

RAM Random Access Memory *(pp. 4, 23, 25, 47, 52, 60, 63, 64, 80, 86)*
RC Radio Controller *(pp. 17, 19, 24, 28, 29, 37, 42, 44, 45, 55, 59)*

ROS Robotic Operating System *(pp. 28, 32)*

RT Real-Time *(p. 9)*

RTOS Real-Time Operating System *(pp. 1, 18, 21, 28, 29, 32–34, 43, 53)*

RTP Real-time Transport Protocol *(p. 57)*

RTPS Real-Time Publish Subscribe *(p. 27)*

SD Storage Disk *(pp. 25, 42, 43, 52)*

SDK Software Development Kit *(pp. 14, 28, 29, 32, 37)*

SITL Software in the Loop *(pp. 17, 31)*

SoC System on Chip *(pp. 2, 23, 32–34, 43, 87)*

SPI Serial Peripheral Interface *(pp. 21, 25, 47, 50, 59, 60, 65, 75)*

SRAM Static Random Access Memory *(pp. 21, 42)*

SSH Secure Shell *(pp. 64, 67, 78, 81, 82)*

SSPFS Supervised Single-Platform Flight Stack *(pp. 3–5, 36, 40, 44, 45, 48–50, 52, 59, 63–70, 72, 74–76, 78, 80, 81, 85, 86)*

SWaP-C Size, Weight, Power and Cost *(pp. 1–4, 6, 7, 12, 33, 34, 36)*

SWD Serial Wire Debug *(p. 42)*

TCB Trusted Computing Base *(pp. 2, 10, 11, 30, 33, 34)*

TLB Translation Lookaside Buffer *(pp. 11, 70)*

TOF Time-of-Flight *(p. 17)*

TOPS Trillion Operations Per Second *(p. 25)*

UART Universal Asynchronous Receiver-Transmitter *(pp. 21, 22, 24, 25, 32, 37, 42, 45, 52, 67, 69)*

UAS Unmanned Aerial System *(p. 12)*

UAV Unmanned Aerial Vehicle *(pp. 1–4, 6–9, 12–15, 17–19, 21, 25, 28, 29, 33–36, 38, 41, 42, 48, 50, 52, 55–57, 67, 69, 74, 77, 78, 81, 82, 85, 86, 111)*

UAVIC Unmanned Aerial Vehicle Integrated Controller *(pp. 4, 5, 36, 39–41, 43–45, 50, 52, 53, 58, 59, 63, 65, 70, 75, 87)*

UDP User Datagram Protocol *(pp. 39, 57)*

UGV Unmanned Ground Vehicle *(p. 12)*

UMPFS Unsupervised Multi-Platform Flight Stack *(pp. 4, 37)*

uORB Micro Object Request Broker *(pp. 27, 28, 77, 80, 87)*

USB Universal Serial Bus *(pp. 43–45, 47–49, 53, 59, 60, 65, 78)*

USPFS Unsupervised Single-Platform Flight Stack *(pp. 3–5, 36, 39, 44–46, 50, 52, 59, 63, 64, 66–70, 72, 74–76, 78, 80–82, 85)*

USV Unmanned Surface Vehicle *(p. 12)*

UTM UAS Traffic Management *(pp. 1, 13)*
UUV Unmanned Under water Vehicle *(p. 12)*
UV Unmanned Vehicle *(pp. 12, 28)*

vCPU Virtual CPU *(p. 70)*
VM Virtual Machine *(pp. 3–5, 8–12, 33, 34, 36, 40, 41, 43, 45, 47, 48, 50, 60, 63–69, 72, 74–76, 78, 81, 82, 85, 86)*
VMA Virtual Memory Area *(p. 68)*
VMM Virtual Machine Monitor *(pp. 1, 9, 10, 33)*
VPN Virtual Private Network *(p. 24)*
VTOL Vertical Take-Off and Landing *(pp. 14, 25, 41)*

WCET Worst-Case Execution Time *(p. 8)*

1 Introduction

"The best way to predict the future is to invent it."
– Alan Kay, computer scientist

Unmanned Aerial Vehicles (UAVs) are increasingly deployed across safety-critical domains – from inspection and mapping to search-and-rescue and defense [2–7] – where flight control must coexist with mission applications under tight Size, Weight, Power and Cost (SWaP-C) constraints. In these settings, the flight-control stack (safety-critical) runs alongside bandwidth- and compute-intensive mission software (non-critical) and must remain protected against timing interference, memory corruption, and fault propagation. Meanwhile, many countries now permit routine operations in populated areas (often below 150 m) [8], raising both safety stakes (e.g., crash hazards to people and property) and security/privacy concerns (UAVs can collect sensitive data). In practice, security is often not engineered from the outset [9]: common configurations expose open, unauthenticated, or unencrypted links that invite cyber-attacks [10, 11]; policy reactions have included bans on certain commercial platforms over cybersecurity concerns [12]; and media-reported incidents illustrate potential misuse and harm [13–15]. As adoption accelerates [16] and regulators move from permissive pilots to explicit operational rules (e.g., European Union categories; UAS Traffic Management (UTM) initiatives), assurance and traceability requirements become more concrete [17, 18].

Conventional multi-board designs – comprising a dedicated Flight Management Unit (FMU) plus a companion computer – provide isolation by hardware separation but add weight, power, and inter-board latency. Yet hardware separation alone is insufficient: compromises in non-critical nodes can still cascade into critical functions via shared interfaces. In this space, formally verified components have been used on the mission side (e.g., seL4 + CAmkES Virtual Machine Monitor (VMM)) [19], but the threat models typically exclude attacks on communication links and assume a correct, uncompromised flight stack. These assumptions are hard to justify given the broad attack surface [8], known autopilot bug densities [20], and vulnerabilities in commercial Real-Time Operating Systems (RTOSs) [21].

Commercial single-board solutions often deploy application-specific software in containers on the mission processor [22, 23], but containers share a kernel and cannot, by themselves, enforce temporal, spatial, or fault isolation between domains [24]. Other consolidation efforts rely on closed-source hypervisors (e.g., PikeOS, CLARE [25]) with application-specific flight-stack customizations. While these paths improve integration, they tend to lack analyzable real-time guarantees and a small, auditable Trusted Computing Base (TCB).

This thesis investigates how to safely consolidate mixed-criticality UAV workloads on shared hardware while preserving temporal, spatial, and fault-containment isolation. We adopt a static-partitioning hypervisor approach centered on Bao, an open-source, lightweight type-I hypervisor, and evaluate it in a representative video-surveillance use case. Our goal is a trustworthy reference stack that is practical on commodity platforms, transparent by design, and measurable end-to-end.

1.1 Motivation

Field operations increasingly require live video, perception, and connectivity (e.g., Long-Term Evolution (LTE)/5G), which compete with control loops for Central Processing Unit (CPU), memory, cache, and Input/Output (I/O) bandwidth. Offloading to a companion computer raises SWaP-C, complicates power/thermal budgets, and introduces link latency/jitter between control and mission domains.

Safety-critical flight control demands bounded worst-case latency and strong fault containment. Containerization shares a kernel and cannot, by itself, provide spatial or fault isolation. Formally verified components improve assurance for what they verify, but securing only the mission side – or assuming benign links and perfect autopilots – leaves critical paths exposed, given the documented attack surface and defects [8, 20, 21]. We therefore need system-level supervision that covers both domains and their interfaces.

Emerging heterogeneous System on Chips (SoCs) make single-board integration attractive, but flight-control support, predictable device assignment, Direct Memory Access (DMA) / Input/Output Memory Management Unit (IOMMU) integrity, interrupt delivery, and cache interference are practical hurdles. A small, analyzable, open approach is needed to both *demonstrate* and *measure* trustworthy consolidation on such platforms. Static partitioning minimizes hypervisor complexity and scheduling overheads, enabling clear timing models and a reduced attack surface. Bao provides this with a small codebase and active open-source development, making it a suitable foundation for a trustworthy UAV stack.

1.2 Goals

The main goal of this work is to design, implement, and evaluate a trustworthy, open-source reference stack that consolidates safety-critical flight control and non-critical mission workloads on shared hardware using the Bao hypervisor, while preserving temporal, spatial, and fault-containment isolation under realistic

video-surveillance missions. We target small/medium multirotors and focus on onboard consolidation (not ground-side networks). Our adversarial tests emphasize inter-domain isolation (faults, misbehaving guests), not radio-frequency jamming or Global Navigation Satellite System (GNSS) spoofing.

The main goal of this work is to design, implement, and evaluate a trustworthy, open-source reference stack that consolidates safety-critical flight control and non-critical mission workloads on shared hardware using the Bao hypervisor, while preserving temporal, spatial, and fault-containment isolation under realistic video-surveillance missions. We target small/medium multirotors and focus on onboard consolidation (not ground-side networks). Our adversarial tests emphasize inter-domain isolation (faults, misbehaving guests), not radio-frequency jamming or GNSS spoofing.

Concretely, we implement and assess the design on a Raspberry Pi 4 + PilotPi platform, aiming to narrow the gap between open-source and commercial solutions and to encourage widespread adoption of secure-by-design features in UAV software stacks. To this end, we define the following objectives:

1. Analyze the state of the art in mixed-criticality UAV systems and define trustworthiness criteria with emphasis on hypervisor-based supervision.

2. Select a representative mixed-criticality application (flight control with video surveillance) to drive concrete design choices.

3. Derive application requirements and choose a target hardware platform that offers security primitives essential for hypervisor-based isolation (e.g., virtual memory, memory protection) while meeting SWaP-C constraints.

4. Design and implement an unsupervised single-platform baseline (Unsupervised Single-Platform Flight Stack (USPFS)) that co-locates mission and flight control without supervision, and a supervised counterpart (Supervised Single-Platform Flight Stack (SSPFS)) atop Bao with static partitioning, using open-source components.

5. Design and implement a mailbox-supervision mechanism to safely share Peripheral Component Interconnect Express (PCIe)–attached peripherals across Virtual Machines (VMs).

6. Validate fault containment by injecting malicious behavior into each system (kernel-level faults, CPU/memory abuse, user-space resource starvation) and observing outcomes. We expect SSPFS to contain faults originating in the mission domain and maintain stable flight, while USPFS exhibits fault propagation that can lead to UAV loss.

7. Quantify supervision overhead and inter-guest interference using MiBench Automotive and Industrial Control Suite (AICS) workloads (offline benchmarking), and evaluate mitigations (e.g., cache coloring).

8. Benchmark each guest with application-specific metrics (PX4 task-scheduling overhead; camera frame rate) and compare the unsupervised and supervised solutions.

9. Evaluate both stacks in real-flight conditions, including behavior under induced component failures, and analyze resource usage and control fidelity from flight logs.

Furthermore, we identified the following research questions:

- **RQ1:** Can static partitioning via Bao maintain flight-control timing while co-locating a live video workload on the same hardware platform?
- **RQ2:** Does Bao-based supervision contain failures in the non-critical mission VM, preserving control authority (i.e., keeping the UAV airborne)?
- **RQ3:** What performance overheads does SSPFS introduce versus USPFS – both in offline MiBench benchmarks and in application-level metrics (PX4 scheduling overhead, camera frame rate)?
- **RQ4:** What performance overheads does SSPFS introduce versus USPFS in PX4's position tracking and the system's resource usage (Random Access Memory (RAM) usage, CPU load) during flight?
- **RQ5:** Does mailbox supervision enable safe, practical sharing of PCIe devices between domains without violating isolation?

1.3 Document Structure

This thesis is organized into six chapters that progressively motivate, design, implement, and evaluate a trustworthy UAV software stack:

- **Chapter 2 – Background and Related Work:** Introduces mixed-criticality systems and virtualization, with emphasis on hypervisors (notably Bao). Surveys reference UAV hardware/software (open-source and commercial), then reviews related work and identifies gaps in security, safety, and SWaP-C efficiency that motivate this thesis.
- **Chapter 3 – Design:** Presents the Supervised Single-Platform Flight Stack (SSPFS) architecture, using the Bao hypervisor to consolidate flight-control and companion functions on one platform. Contrasts supervised SSPFS with unsupervised baselines – multi-platform (Unsupervised Multi-Platform Flight Stack (UMPFS)) and single-platform (USPFS) – and justifies design choices for a video-surveillance use case. Details hardware/guest mapping under static partitioning and introduces a Bao mailbox-supervision mechanism to safely mediate shared PCIe device access.
- **Chapter 4 – Implementation:** Describes the realization of USPFS and SSPFS on the Unmanned Aerial Vehicle Integrated Controller (UAVIC) platform (Raspberry Pi 4 + PilotPi): hardware bring-up, PX4 and video pipeline integration, Bao configuration, VM separation, and the mailbox-supervision driver.

- **Chapter 5 – Evaluation:** Assesses functionality, performance, and isolation. Establishes the USPFS baseline, then quantifies SSPFS overheads (MiBench Automotive and Industrial Control Suite (AICS)), guest-to-guest interference (including cache and device effects), and the impact of mailbox supervision. Compares single- versus dual-VM setups using guest-specific metrics (e.g., PX4 scheduling overhead, camera frame rate), and validates both systems in real flight with an automated mission (tracking accuracy and resource usage), including repetition of functional tests in flight.
- **Chapter 6 – Conclusions and Future Work:** Summarizes how SSPFS consolidates mixed-criticality stacks on shared hardware with robust isolation and modest overhead. Outlines future directions: broader UAVIC support, deeper system-resource accounting, and extended testing (long-duration flights, adversarial scenarios).
- **Appendices:** Provide supplemental material (e.g., UAV taxonomy). All configurations, code, and demonstration videos are available online (see [26]).

2

Background and Related Work

"We stand on the shoulders of giants."
– Isaac Newton, mathematician, physicist, astronomer

This chapter establishes the technical context for the thesis. We begin with mixed-criticality systems (Mixed-Criticality Systems (MCSs)): why they matter, where they appear, and the challenges they pose for timing predictability, isolation, and assurance under practical SWaP-C constraints. We then outline the current approach to manage this complexity, highlighting the trade-offs they introduce. Next, we survey UAVs' reference hardware and software stacks, covering both open-source and commercial solutions, with emphasis on properties relevant to mixed-critical deployments (e.g., platform support, determinism, and isolation). Finally, we review related work, organized recent research along the axes most pertinent to this thesis (flight stacks, onboard platforms, and virtualization). This positions the contribution that follows and clarifies the specific gaps the proposed design addresses.

2.1 Mixed Criticality Systems

Embedded platforms increasingly host many functions on the same hardware to reduce size, weight, power, and cost (SWaP-C) [27]. However, not all functions are equally critical. In safety-regulated domains, strong guarantees are required to ensure that consolidation does not compromise safety [27, 28]. Standards such as ISO 26262 [29] and DO-178C [30] formalize these assurances, particularly in automotive and avionics. On the one hand, consolidation pressures favor efficient resource sharing to minimize SWaP-C; on the other hand, domain isolation is needed to guarantee non-interference. The fundamental question in MCSs is how to reconcile these conflicting goals – providing strong separation for assurance while enabling sharing for efficiency [27]. Addressing this requires system designs that integrate software and hardware on a single platform, together with modeling, verification, and, where applicable, certification [28, 31].

A MCS is, simply, a system that integrates software components of two or more criticality levels on the same platform [28]. Consider a UAV executing a video-surveillance mission (Fig. 1). Tasks are color-coded

by criticality according to DO-178C [30, 31]. Navigation and stability are life-critical: flight-control loops must meet their deadlines under all conditions, or catastrophic failure may occur. Accordingly, sensor acquisition, communication handling, processing, and actuation must complete within predefined time bounds. Here, *criticality* denotes the assurance level required to bound the probability of such failures [30, 31].

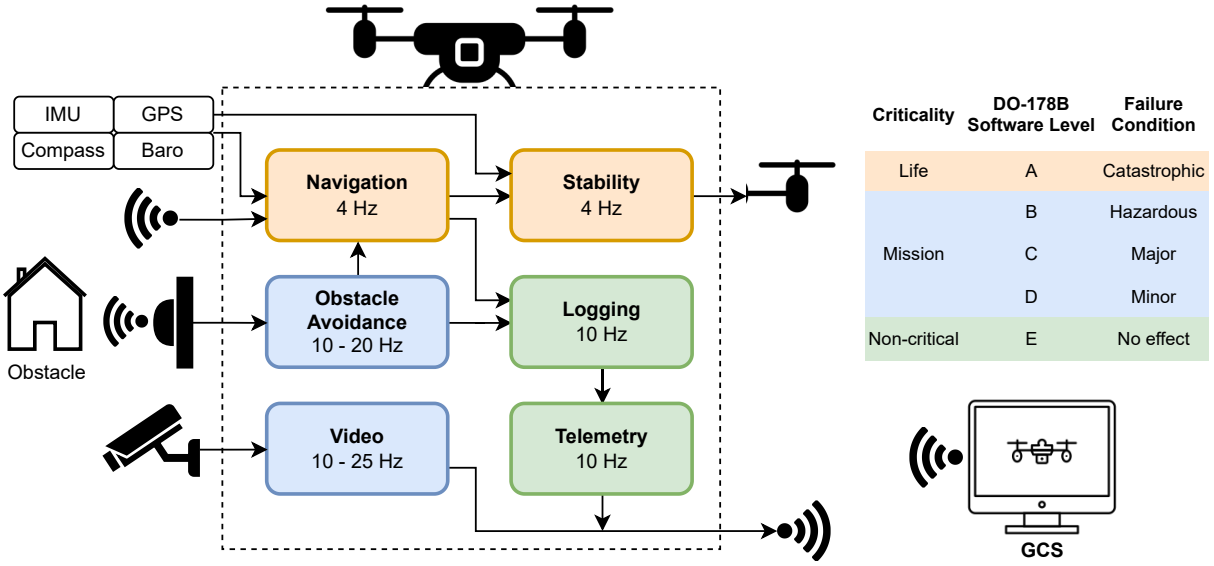


Figure 1: Mixed-criticality system example: UAV application (adapted from [32]).

By contrast, obstacle avoidance and video capture are mission-critical. They have soft real-time requirements, but their consequences differ: failing to detect an obstacle may endanger the aircraft, whereas dropping camera frames is generally acceptable. Lastly, auxiliary tasks such as logging and telemetry are non-critical, as missing soft deadlines does not cause safety harm.

One consequence of applying the DO-178C safety standard at a system level is that all UAV functions would need to be developed to the life-critical level of navigation and stability [28, 31]. In our case, that would make "missing a camera frame" as unlikely – and as costly to assure – as "missing an actuator output", resulting in substantial over-engineering of mission- and non-critical subsystems. The alternative is to demonstrate *temporal and spatial isolation*, i.e., that lower-assurance functions can neither delay nor corrupt higher-assurance functions, even under faults [28, 33].

While MCS principles apply broadly, UAVs face domain-specific hurdles. They must meet extreme SWaP-C limits: (1) every gram of weight counts, especially in lower weight classes [34], affecting the UAV's autonomy and flight dynamics; (2) they operate in uncontrolled, rapidly changing environments [35]; (3) and must maintain critical functions during faults, since not every error can be anticipated [16]. The shift to mainstream multicore architectures further complicates resource sharing [27]. Current approaches reveal a core trade-off: on one hand, multi-board designs provide strong isolation but poor SWaP-C metrics; on the other, single-board designs improve SWaP-C but weaken isolation.

Two major research branches stem from this dichotomy. A more *theoretical* line studies scheduling

with criticality-dependent Worst-Case Execution Times (WCETs) to schedule systems at each criticality level, but at the cost of higher processor utilization [36]. A more *practical* line focuses on safe partitioning with shared computational and communication resources, at the cost of increased design complexity [33]. Combining the two is hard as flexible scheduling typically requires at least dynamic partitioning, whereas certified systems favor complete separation or static partitioning [27]. This gap is acute in UAVs, where both efficiency and assurance are non-negotiable.

2.1.1 Virtualization

Mixed-criticality systems integrate software of disparate criticality on shared hardware. As noted above, the most promising practical approach is to partition the system safely while sharing compute resources. This leads to the concept of *virtualization*: a logical abstraction of hardware with isolation guarantees, realized via hypervisors, Operating System (OS)-level virtualization, or unikernels [33]. Fig. 51 maps the key concepts referenced in this section, and Fig. 2 illustrates representative approaches [33]. Starting on the left, a conventional software stack runs an OS (with runtime and libraries) directly on the hardware and hosts applications. In this model, failures in any component can propagate system-wide because no isolation layer intervenes.

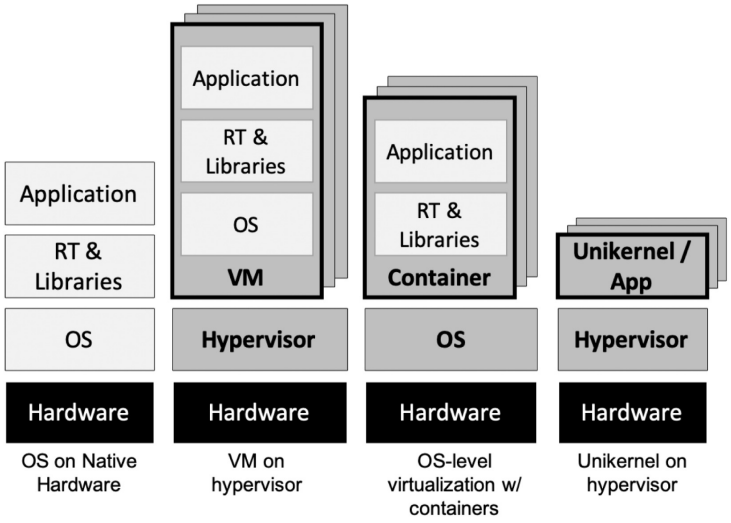


Figure 2: Examples of virtualization approaches [33][1]

Next is hypervisor-based virtualization, which inserts a software layer that abstracts hardware resources so that multiple isolated application environments – VMs or guests – can run on the same machine [33]. Hypervisors are detailed in the next section.

OS-level virtualization (containerization) does not emulate hardware. Popularized by Docker [37, 38], it is widely used in cloud settings to avoid replicating the full OS stack and to increase consolidation [33]. A container is a process-like domain atop the host OS, with isolated namespaces and managed resources

[1] Used with permission from Elsevier: license nr. 5457890117132

(CPU, memory, filesystem, network) [33]. Although it lacks hardware emulation, containerization is gaining traction in Real-Time (RT) contexts [39, 40] and in UAV systems to deploy user-space applications on mission computers [23]. Trade-offs include sources of non-determinism (e.g., dynamic resource assignment), large trusted code bases, and limited hardening, which complicate testing, assurance, and certification [33].

Finally, unikernel-based virtualization compiles the entire software stack (minimal OS components, libraries, language runtime, and application) into a single VM image – a unikernel or library OS – that typically runs atop a general-purpose hypervisor [33]. Unikernels offer fast boot, low memory footprint, high performance, and strong isolation (via the host hypervisor), but reduce flexibility: applications must target the unikernel environment. Further work is needed for MCSs, particularly regarding certification and dependability support [33].

Cinque et al. [33] relate hypervisor/guest-OS pairings to typical use cases. General-purpose (non-RT) hypervisors host non-RT guests for server consolidation, and can host RT guests for functional testing and prototyping (without hard guarantees). Real-time hypervisors host non-RT guests for Quality of Service (QoS)/performance studies and RT guests for safety-critical workloads. MCSs sit in this latter space: they require a real-time hypervisor capable of hosting both RT and non-RT guests with strong isolation and bounded interference. We therefore examine hypervisors next.

Before proceeding, we briefly clarify a related trend in the telecommunications domain often called *softwarization*, which occasionally appears under the umbrella of virtualization [41–43]. Softwarization implements network functions that historically resided in hardware appliances (routers, firewalls, etc.) as software running on general-purpose servers under VMs or containers [44]. These software instances are termed *virtual network functions* (e.g., virtual firewalls, virtual routers) [45]. Softwarization is also gaining traction in the UAV field [41, 46–48], supporting network-centric architectures, service deployment, and communication management, especially as systems evolve toward multi-UAV coordination [49, 50]. However, it targets non-critical networking stacks and thus falls outside the scope of our analysis.

2.1.2 Hypervisors

A hypervisor is a Virtual Machine Monitor (VMM): a software layer that abstracts hardware so multiple isolated application environments – VMs or guests – can run on the same machine (see Fig. 2). Each VM typically includes a guest OS and its applications. To be useful in mixed-criticality settings, the hypervisor must enforce temporal, spatial, and fault-containment isolation [33]. *Temporal isolation* means a task in one virtual domain must not cause harmful delays to tasks in another. *Spatial* (memory) isolation refers to separating code, data, and device access between domains, generally using hardware protection (e.g., Memory Management Unit (MMU), IOMMU). *Fault containment* ensures that errors do not propagate to block or crash the entire system.

Hypervisors are commonly classified by where they run, how guests interact with them, and whether they provide explicit real-time support. *Bare-metal* (type I) hypervisors run directly on hardware and control

resources themselves (e.g., Jailhouse [51], Bao [52], CLARE [53], Xen [54]).[2] *Hosted* (type II) hypervisors run atop a host OS and manage resources indirectly (e.g., KVM [55]). In *full virtualization* the guest runs unmodified while the hypervisor emulates or traps privileged operations (e.g., KVM, Hyper-V [56]); in *paravirtualization* the guest cooperates with the hypervisor via hypercalls (e.g., Xen [57]). *Real-time* hypervisors add mechanisms to allocate and police time budgets so VMs meet timing constraints [33]. These can be *dynamic* (resources assigned at run time; e.g., KVM [55], Xen [57]) or *static* (resources fixed at instantiation; e.g., Jailhouse [51], Bao [52]), with static designs often preferred for MCSs due to lower overhead, smaller code bases, and simpler testing/certification [33]. Hypervisors also vary by application scope, from embedded/mission-specific to general-purpose [58].

Solution families reflect these choices [33]. *Separation-kernel* hypervisors implement a small bare-metal partitioning layer that defines fixed VMs and information flows, delegating rich OS services and drivers to guests (e.g., PikeOS [59], Xtratum [60], Jailhouse [51], Bao [52]). *Microkernels* implement only minimal abstractions and operations in supervisor mode (e.g., memory and thread management, Inter-Process Communication (IPC)), placing drivers and higher-level services in user mode; some can serve as a hypervisor with a user-level VMM (e.g., seL4 with CAmkES VMM [61, 62]). These approaches differ in emphasis: microkernels structure an OS from small user-mode components atop a minimal kernel; separation kernels focus on static partitioning and isolation – often leveraging hardware virtualization – and rely on guests for OS-like functionality [33].

General-purpose hypervisors (e.g., KVM, Xen) can be extended with real-time features. Some approaches leverage security CPU extensions (e.g., Arm TrustZone, Intel SGX) to strengthen isolation, albeit with platform dependence (e.g., LTZVisor/RTZVisor [63, 64], VOSYSMonitor [65, 66]). Unikernel solutions package a single application with minimal OS services to run atop a hypervisor (e.g., ClickOS [67], HermitCore [68]), trading flexibility for fast boot, small footprint, and performance.

In practice, selection is guided by footprint, alignment with safety standards, licensing, fault tolerance, security, and hardware coverage [33]. Cinque et al. [33] provide a broad survey of such solutions in the context of MCSs and industrial requirements.

2.1.2.1 Bao

Bao (Mandarin “bǎohù”, “to protect”) is an open-source, lightweight, security- and safety-oriented bare-metal hypervisor for embedded real-time MCSs, developed by the Embedded Systems Research Group at the University of Minho [52, 69]. It follows the static, type-I partitioning model popularized by *Jailhouse* [51] but removes the Linux host dependency, thereby reducing the system TCB (currently ~ 8 kLOC of C plus ~ 500 LOC of assembly) and easing the path toward certification [52]. Bao supports Armv7/8 (-A and -R), and RISC-V (RV32/RV64), with ongoing work toward Infineon TriCore and Renesas RH850.

Bao is a clean-slate design (Fig. 3) comprising a thin, privileged layer that leverages Instruction Set Architecture (ISA) virtualization for static partitioning [52]. It has no scheduler and does not rely on

[2] Xen is typically classified as type I (bare metal) [33].

external libraries or a privileged VM (e.g., Linux), in contrast to Jailhouse [51]; instead, it depends only on standard platform-management firmware to initialize the system and perform platform-specific tasks [52]. Removing the Linux dependency improves safety/security posture and real-time behavior [52].

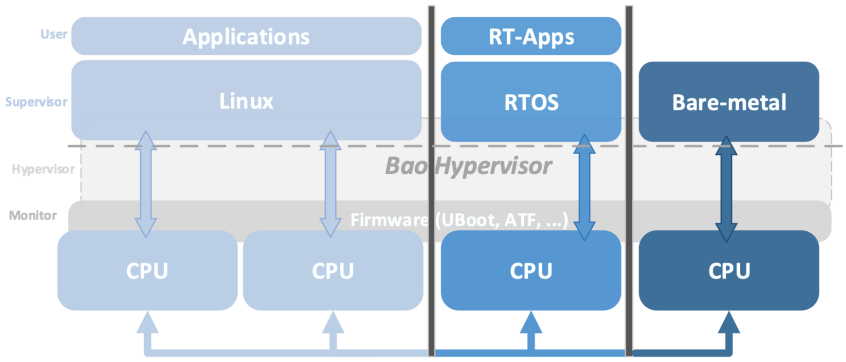


Figure 3: Bao hypervisor architecture [52][3]

Bao targets safety-critical systems by providing temporal, spatial, and fault-containment isolation [52]. *Temporal isolation* follows from the absence of a scheduler and the exclusive 1:1 mapping between virtual and physical CPUs at VM instantiation time, together with guest access to per-CPU architectural timers [52]. *Spatial isolation* is ensured by statically assigned memory enforced via two-stage translation, using superpages when possible to reduce translation and Translation Lookaside Buffer (TLB) pressure and to aid speculative fetch [52]. While these mechanisms provide strong isolation and predictable timing, shared hardware resources (e.g., Last-Level Caches (LLCs), interconnects, memory controllers) can still induce temporal interference and expose VMs to Denial of Service (DoS) and timing side channels [57, 70, 71]. To mitigate these effects, Bao supports cache coloring for LLC partitioning and memory-bandwidth regulation [52].

For devices, Bao favors direct assignment (pass-through). For memory-mapped I/O, the same two-stage translation used for memory protection applies; exclusivity is not enforced automatically, so deployments must avoid unintended sharing [52]. Paravirtualized I/O via *VirtIO* is available for selected devices [69, 72–75], enabling efficient sharing while preserving isolation and complementing pass-through where appropriate. Bao also offers inter-VM communication via shared memory [52, 76]. Interrupt virtualization currently targets Arm Generic Interrupt Controller (GIC)v2/v3, which route interrupts to the hypervisor for re-injection, adding latency and code complexity [52]. Arm GICv4 addresses this with direct guest delivery [77, 78], though adoption remains limited. On RISC-V, Bao supports both the legacy Platform-Level Interrupt Controller (PLIC) and the more recent Advanced Interrupt Architecture (AIA) [69, 79].

Relative to alternatives (e.g., Xen's Dom0-less mode), Bao offers a similar static-partitioning model with a smaller TCB and a clean security posture, with minimal virtualization overhead [52]. It has been extensively benchmarked with the MiBench AICS suite for real-time embedded domains, showing low

[3]Used under the terms of the Creative Commons BY 3.0 license.

overheads and demonstrating that cache coloring, although not optimal, can effectively mitigate inter-VM interference [80].

Overall, Bao is a strong foundation for a trustworthy, open-source reference software stack for UAV applications: it is open-source, enforces strict domain isolation with efficient resource sharing, and has undergone extensive benchmarking, enabling consolidation of mixed-criticality software on a single hardware platform while minimizing SWaP-C.Table 2 presents a concise summary of Bao's properties.

Table 2: Bao hypervisor summary [69]

<table>
<tr><th>Architecture</th><th>Supported platforms</th><th>Size</th><th>License</th><th>Version</th></tr>
<tr><td>Type I,
Static</td><td>Armv7/8-A, Armv7/8-R
RISC-V RV32 and RV64
Infineon Tricore 1.8 (WIP)
Renesas RH850 (WIP)</td><td>Small
~8 kLOC C +
~500 LOC ASM</td><td>Apache 2.0</td><td>1.0</td></tr>
<tr><th>Features</th><th colspan="4">Isolation</th></tr>
<tr><td rowspan="5">Static partioning
IO pass-through and VirtIO support
1:1 mapping of virtual to physical CPUs (no scheduler required)
Virtual interrupts directly mapped to physical ones
No external dependencies (except for standard platform management firmware)
Provides a basic mecanism for inter-VM communication
State-of the art partioning mechanisms to be implemented (e.g., memory throttling)</td><th>Spatial</th><th colspan="3">Temporal</th></tr>
<tr><td>Provided by a 2-stage translation
HW virtualization support

Translation overhead is minimized using superpages whenever possible, faciliting speculative fetch for potential guest performance improvement

It uses cache coloring to enable LLC cache partioning independently for each VM, but at the expenses of memory waste and fragmentation and increased boot time</td><td colspan="3">Exclusive CPU assignment discards the scheduler

Per-CPU architecture timers are available to and directly managed by the guests</td></tr>
<tr><th>IO</th><th>Interrupts</th><th colspan="3">Related Work</th></tr>
<tr><td>Directly assigned to guest in a pass-through only IO configuration

For memory-mapped IO it uses the existing memory mechanism and 2-stage translation provided by the virtualization support

Paravirtualized I/O via VirtIO is supported for selected devices, enabling efficient sharing with isolation</td><td>Supports Arm GICv2 and GICv3
Supports RISC-V PLIC and AIA

Interrupts are forwarded to the hypervisor which must re-inject them in the VM using a limited set of pending registers</td><td colspan="3">Jailhouse: pioneer in the static partioning adopted by Bao but requires the Linux Kernel to boot

Xen Dom0-less: eliminates the Linux dependency to boot and execute

Bao provides basically the same features from the previous ones, but has a smaller TCB and implements clean security features</td></tr>
</table>

2.2 Unmanned Aerial Vehicles

Unmanned Aerial Vehicle (UAV), Unmanned Aerial System (UAS) or, more commonly, *drones*, are unmanned robotic aircraft capable of executing missions and carrying payloads, guided either by remote control stations or autonomously [18, 34]. They belong to a broader class of Unmanned Vehicles (UVs), alongside Unmanned Ground Vehicles (UGVs), Unmanned Surface Vehicles (USVs) (e.g., boats), and Unmanned Under water Vehicles (UUVs) [18].

UAVs date back as far as the 18th century: in 1783, France publicly demonstrated the first uncrewed aircraft – a hot-air balloon [81]. In 1896, Alfred Nobel reportedly launched a camera-equipped rocket, and in 1935 the first modern radio-controlled aircraft was developed in the U.K. [81]. The term *drone* is often attributed to Lieutenant Commander Delmer Fahrney, who was in charge of the U.S. Navy's *Radio Controlled Aircraft* program [81]. A UAV designed for surveillance and scouting was developed in Israel in 1973, and the *Gulf War* marked the first conflict with significant UAV use [81]. Commercial momentum accelerated in the 2000s, with civilian permissions arriving in the U.S. in 2006; smartphone-controlled quadcopters appeared around 2010, and consumer camera drones reached the market by 2013 [81].

Open-source autopilot projects amplified accessibility and capability: Paparazzi UAS (2003) [82], ArduPilot (2008) [83], and Dronecode/PX4 (2011) [84]. Prices dropped as ecosystems grew, fueling a commercial boom. In North America, UAV market revenue reached $737 millions in 2017, with forecasts of roughly $6.7 billions by 2026, driven by agriculture, security, and law enforcement [16]. UAV versatility spans high-value tasks often difficult or unsafe for humans [2, 3]: search and rescue [2], precision agriculture [4], pipeline inspection [5], delivery of goods and medical supplies [3], video capture and surveillance [6], and cartography [7], among others. Only recently have many countries explicitly enforced UAV-specific regulations, with routine operations over populated areas (often below 150 m) increasingly permitted under constraints [8]. For example, in 2019 the European Union established common rules allowing drones under 25 kg to fly without prior authorization under defined conditions; it also mandated operator registration and national no-fly zones [17]. However, global harmonization remains limited. In 2020, the Federal Aviation Administration (FAA), National Aeronautics and Space Administration (NASA), and partners released UTM 2.0, describing protocols to enable multiple Beyond Visual Line-Of-Sight (BVLOS) operations in shared airspace [18].

Broadly, five categories frame UAV regulation [85, 86]. First, *applicability* defines scope (vehicle type, weight, operational role). Second, *operational limitations* constrain where and how UAVs may fly (airspace class, altitude, proximity to people/infrastructure). Third, *administrative and legal requirements* establish oversight (registration, licensing, record-keeping). Fourth, *technology specifications and requirements* address mechanical integrity and command-and-control capabilities (geofencing, fail-safes). Finally, *moral and ethical* considerations concern privacy and the broader security of people and communities.

2.2.1 Classification

An UAV can be classified in multiple ways:

- **Flight principles**: UAVs may be *lighter-than-air* (e.g., balloons, blimps; see EBlimp Atom [87]) or *heavier-than-air* (e.g., gliders [88], rotorcraft, and ornithopters [89]).
- **Airframe**: The most visible external characteristic is the airframe, typically classified as fixed-wing, fixed-wing hybrid, single-rotor, and multirotor [16]. Fixed-wing platforms are fast, carry heavier payloads, and offer longer endurance, but generally require runway takeoff/landing and sustained

forward airspeed [16]. Hybrid variants address runway constraints via Vertical Take-Off and Landing (VTOL) like a rotorcraft while cruising as a fixed-wing; they improve operational flexibility at some cost in complexity and speed [16]. Rotorcraft – single-rotor and multirotor – can hover and perform VTOL. They offer precise positioning but tend to be more mechanically complex (vibration, maintenance) and expensive, with multirotors (tri-, quad-, hexa-, octocopters) being the most affordable and widely produced; their endurance is usually lower due to higher power draw [16].

- **Altitude**: Broadly, UAVs are divided into Low Altitude Platforms (LAPs) and High Altitude Platforms (HAPs). LAPs typically operate from ~ 3–$9\,\mathrm{km}$ and often support cellular communications; HAPs fly above $9\,\mathrm{km}$, enabling wider coverage (historically explored by major tech companies) [16].

- **Overall weight**: UAVs range from a few grams to hundreds of kilograms. For example, Australia classifies them as *micro* (below $100\,\mathrm{g}$), *very small* ($100\,\mathrm{g}$–$2\,\mathrm{kg}$), *small* (2–$25\,\mathrm{kg}$), *medium* (25–$150\,\mathrm{kg}$), and *large* (over $150\,\mathrm{kg}$) [34].

- **Power source**: UAVs may use electric power, liquid fuel, or hybrids. Most platforms use electric motors, powered by batteries (typically Lithium Polymers (LiPos); e.g., Parrot [90], DJI Mavic 3 [91]), hydrogen fuel cells (e.g., EnergyOr H2Quad 1000 [92]), or solar augmentation. Fuel-only systems (e.g., nitro or gasoline engines; see examples [93]) deliver high energy density and long range but add noise, emissions, and maintenance. Hybrids (fuel + battery or Hydrogen Fuel Cell (HFC) + battery; e.g., Flaperon MX8 [94]) trade between endurance and complexity. Batteries are simple and quiet but heavier per unit energy; HFCs sit in between with higher cost and more complex power management.

- **Target audience**: Platforms are typically designed for recreational/hobbyist [95], commercial/industrial [96], or defense applications [15, 97, 98].

- **User-modifiability**: In 2021, about 26% of commercial drones sold were open-source (e.g., Pixhawk4) [99]. Open-source adoption is rising due to transparency, extensibility, and the ability for users to build on an existing ecosystem (e.g., mature flight-control algorithms). At the other end are proprietary ecosystems (e.g., DJI) with dominant market share [99], where access is typically gated via vendor Software Development Kits (SDKs) [100].

Figure 4 illustrates several UAV types, focusing on airframe and power source. Core performance characteristics include speed, endurance (autonomy), payload, range, and ceiling. Speed depends on propulsion, aerodynamics, weather, and power usage: small UAVs may reach $50\,\mathrm{km/h}$, while larger platforms can exceed $300\,\mathrm{km/h}$ [16]. Table 3 summarizes the classification axes used in this survey, linking each axis to concrete categories with some examples.

Autonomy/Endurance is the maximum continuous flight time on a single energy source (battery, fuel, or hybrid), ranging from about 20–30 minutes for small UAVs to several hours for larger platforms. It is

(a) Firebird: Fixed-wing [101]

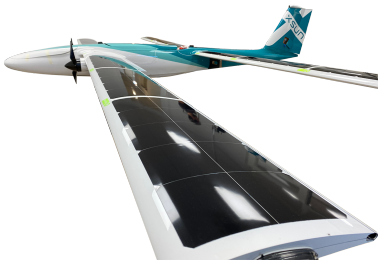
(b) Velos: single-rotor [102]

(c) Parrot: multirotor [90]

(d) DeltaQuad Pro: fixed-wing hybrid [103]

(e) Energyor Quad 1000: hybrid (HFC + batteries) [92]

(f) HAMR: hybrid (fuel+batteries) [104]



(g) Solar X One: solar powered [105]

Figure 4: UAV types examples

strongly influenced by size, weight, aerodynamics, and weather. Ongoing work explores in-flight recharging via renewable sources (Photovoltaic (PV) cells) or wireless power techniques (e.g., lasers emission) [16, 113]. *Payload* denotes the lifting capacity, spanning from a few grams (small UAVs) to hundreds of kilograms (large UAVs); common payloads include sensors and video cameras [16]. *Range* is the maximum command-and-control distance and depends on the communication technology, network conditions, link budget, and environment [16]. *Ceiling/Altitude* reflects the maximum operating height, limited by both technological performance (propulsion, energy, airframe) and regulatory constraints [16]. Figure 52 summarizes these concepts in a generic UAV overview.

2.2.2 System overview

Figure 5 presents an overview of the UAV system and its ecosystem with numbered callouts. At the core *(1)* are the flight-control hardware and software (flight computer + autopilot), whose main responsibilities include path planning, communication management, data acquisition, and mission execution [114].

Table 3: UAV classification axes with examples

Axis	Categories	Example platforms	Notes
Flight principles	Lighter-than-air; Heavier-than-air (glider, rotorcraft, ornithopter)	EBlimp Atom [87]; PX4 Glider [88]; Ornithopter [89]	Buoyancy vs. aerodynamic lift/propulsion.
Airframe	Fixed-wing; Fixed-wing hybrid (VTOL); Single-rotor; Multirotor (tri/quad/hexa/octo)	Firebird [101]; DeltaQuad Pro [103]; Velos [102]; Parrot [90]	Endurance vs. agility trade-offs; hybrids ease runway needs.
Altitude band	LAP (∼3–9 km); HAP (>9 km)	Parrot [90]; Pathfinder Plus [106]	LAP often for comms relays; HAP for wide-area coverage [16].
Overall weight	Micro (< 100g); Very small (100g–2kg); Small (2–25kg); Medium (25–150kg); Large (> 150kg)	BetaFPV Meteor65 Pro [107]; NXP HoverGames [108]; Hunter10 [109]; Shadow50 [110]; Reach-S [111]	Example scheme per AU regs [34]; impacts rules and mission profile.
Power source	Battery (LiPo); Fuel (gas/nitro); Hydrogen fuel cell; Hybrid (fuel+bat., HFC+bat.); Solar assist	Parrot [90], DJI Mavic 3 [91]; Gas-powered [93]; EnergyOr H2Quad 1000 [92]; Flaperon MX8 [94]	Energy density vs. mass/-complexity; hybrids extend endurance at added cost.
Target audience	Recreational/hobby; Commercial/industrial; Defense	KK2 flight board [95]; Skynode X [96]; Defense-use cases [15, 97, 98]	Drives cost, openness, certification, and ecosystem.
User modifiability	Open/user-modifiable; Proprietary (vendor SDK)	Pixhawk4-based (PX4) [112]; DJI (SDK) [100]	Transparency/extensibility vs. turnkey integration and market dominance [99].

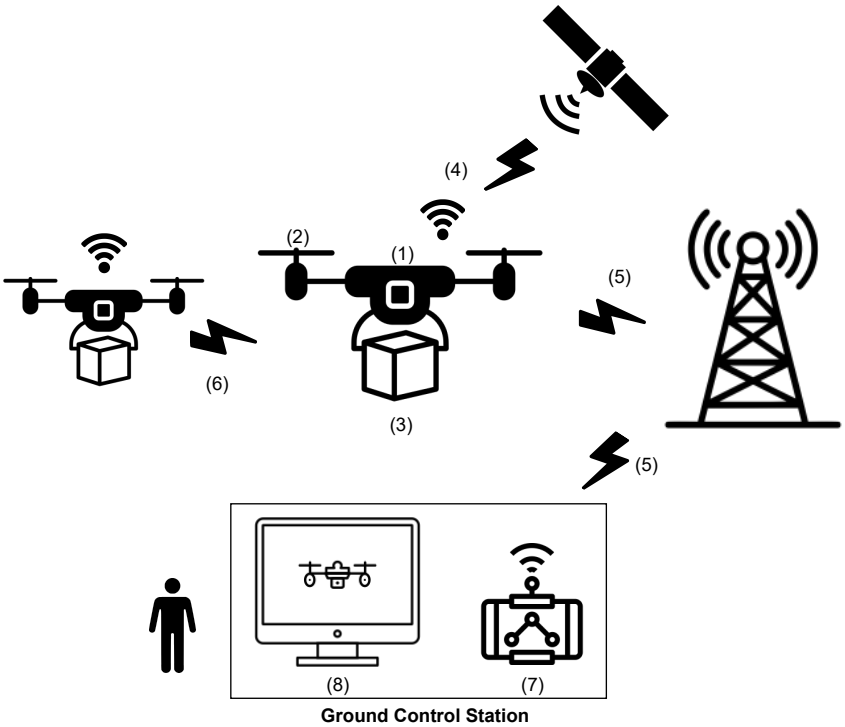


Figure 5: UAV system overview (adapted from [16, 114]).

Path planning in *(1)* works together with the ground control station *(5)* to assist navigation, find feasible/optimal trajectories, maintain environmental awareness (weather, obstacles), and regulate motion and speed for obstacle avoidance [114]. The on-board flight controller *(1)* fuses data from multiple sensors (e.g., Inertial Measuring Unit (IMU), barometer, Global Positioning System (GPS)) with commands from the Ground Control Station (GCS) *(5)* to perform attitude estimation and control (e.g., Kalman filtering) and to drive the propulsion system *(2)* (motors) [115].

The controller *(1)* typically manages three link classes [114]: (i) UAV–GCS link *(5)*, used for telemetry (including audio/video) and human remote control via the pilot interface *(8)* and handheld/console device *(7)*; (ii) UAV–satellite link *(4)*, which provides weather, climate, and GPS services for accurate navigation [116, 117]; and (iii) UAV–UAV inter-vehicle link *(6)*, supporting cooperative missions and shared situational awareness (e.g., hazard signaling) [49, 50]. The mission objective (e.g., video capture, topographic mapping [7]) directly determines the payload *(3)*, such as cameras or Light Detection And Ranging (LiDAR) [118].

Sensors on *(1)* are commonly grouped into *obstacle-avoidance*, *payload (3)*, and *navigation* classes [119]. Ultrasonic and infrared sensors are widely used for collision avoidance; Time-of-Flight (TOF) sensors like LiDAR offer better range/precision at higher cost and complexity. Navigation typically relies on an IMU (attitude/heading), a barometer (altitude with centimeter-level precision), and GNSS/GPS from *(4)* (global position). Because GPS alone can have errors on the order of meters, fusing IMU and barometer data is standard to improve accuracy [120].

Electronic propulsion dominates. In multirotors, Brushless Direct Currents (BLDCs) in the propulsion system *(2)* drive the rotors, controlled by Electronic Speed Controllers (ESCs) – high-frequency inverters that convert Direct Current (DC) to three-phase Alternating Current (AC) and modulate motor speed [121]. Fixed-wing hybrids additionally use servomotors for control surfaces (e.g., flaps) [122]. Figure 53 depicts a concept map of the main tasks and components. The UAV ecosystem can be outlined as a top-down view of five layers [18] (Figure 54):

(1) *Flight supervision* (flight management, safety/authorization, remote identification). Commands flow from the GCS to the Flight Control System (FCS); tools such as `QGroundControl` provide mission planning, mapping, and live telemetry. Fail-safes (geofencing, low battery, loss of Radio Controller (RC)/datalink) must be enforced by the pilot or autopilot. Airspace authorization can be obtained via `LAANC` or `Google Wing OpenSky`; since 2023, remote ID mandates broadcasting a unique identifier.

(2) *Command and control*. Ensures safe flight via human or autopilot-generated commands over proprietary links or open Application Programming Interfaces (APIs) (e.g., MAVLink SDK, Parrot SDK). Notable open-source autopilots include `PX4`, `ArduPilot`, and `Paparazzi UAS`.

(3) *System simulation*. Analyzes behavior under varied conditions via Software in the Loop (SITL) / Hardware in the Loop (HITL) (e.g., Gazebo [123] with PX4 [124] or ArduPilot [125], CoppeliaSim [126])

and full flight simulators (FlightGear [127]).

(4) *Operating systems.* Provide hardware abstraction: open-source RTOSs (`NuttX` [128], `ChibiOS` [129], `FreeRTOS` [130]) and General-Purpose Operating Systems (GPOSs) (`Linux` [131]), as well as proprietary options (`VxWorks` [132]). Some flight stacks also run natively without a full OS (e.g., `Paparazzi UAS` [133], `INAV` [134]).

(5) *Physical hardware.* Comprises sensors, compute, communications, and actuators (propulsion and payload), i.e., the electronic platform and effectors that realize the flight and mission functions.

2.2.3 Security and Safety

Security of the UAV ecosystem – and the associated safety of people and property – is often underemphasized at design time [9]. In practice, many platforms ship with onboard wireless links that are open, unencrypted, or unauthenticated, exposing them to a wide range of cyberattacks [10, 11]. The issue is not confined to consumer systems: in 2017 the U.S. Army restricted the use of DJI drones over cybersecurity concerns [12]. Compromise can yield unauthorized control or data exposure, and several media-reported incidents show weaponization risks [13–15]. With projected market growth [16] and regulations that increasingly permit operations over people [86], both likelihood and impact of misuse can rise.

Common attacks include DoS and Distributed Denial Of Service (DDoS), which degrade availability by draining batteries, overloading compute, or saturating communication links, leading to service interruptions [16]. More sophisticated threats span GPS spoofing (broadcasting counterfeit satellite signals), GPS jamming, instrument spoofing (e.g., gyroscope/compass), and even killing critical processes [8]. Ground control stations are also targeted; keyloggers, malware, and supply-chain compromises can provide indirect access to the UAV, enabling data exfiltration or malicious command injection [16].

Nassi et al. [8] provide an extensive survey, classifying threats by target: the UAV itself or people. *Attacks targeting UAVs* include tampering with onboard electronics (e.g., counterfeit or unsigned firmware, sensor spoofing, GPS jamming), the airframe/payload (e.g., malicious Computer-Aided Design (CAD) files, nets, projectiles), the GCS and First-Person View (FPV) links (e.g., deauthentication, video exfiltration, remote takeover), and cloud backends used for telemetry. Mitigations include signed/attested firmware, encrypted/authenticated links with frequency agility, runtime monitoring and fault containment [16], parachute systems, and strong access control (including multi-factor authentication). *Attacks targeting people* range from privacy violations to kinetic harm against individuals, organizations, or states. Counter-UAV measures follow a detect–assess–interdict pipeline using radar, optical/infrared, LiDAR, and acoustic sensing; each has gaps (e.g., small UAV radar cross-section, suppressible acoustic signatures) [135]. Upon confirmation, interdiction options include nets, jammers, birds of prey, and lasers – each with distinct effectiveness and safety trade-offs [8].

Safety failures pose risks to people and property and can be grouped into several categories [136]. First, *damage from calculation errors* arises when the UAV's flight proves more hazardous than anticipated

(e.g., entry into restricted zones). Second, *sensor failures* occur when onboard sensors miss delicate or transparent obstacles (wires, branches, glass), leading to collisions, sometimes compounded by pilot overconfidence. Third, *obstacle deviation* can cause crashes into buildings or vegetation due to sudden, unexpected changes in direction. Fourth, *direct attacks* use UAVs to harm individuals or steal payloads (e.g., medical supplies), with recreational or hobbyist platforms frequent targets. Finally, *accidental damage* results when a UAV goes out of control due to operator error, cyberattack, or software/hardware malfunction. Regardless of cause, a falling aircraft can inflict severe harm on people and goods.

Security and safety are fundamental to UAV operation, yet reducing attack vectors and overall attack surface requires deliberate design choices. Ferrão [136] argues that UAV design should address both dimensions simultaneously, as they strongly influence each other. Figure 55 illustrates the UAV security and safety taxonomy.

2.2.4 UAV Reference Hardware

In this section we discuss reference hardware for UAVs. We first present an overview of the hardware architecture, then analyze open-source and commercial solutions in more depth.

2.2.4.1 Overview and Architecture

Figure 6 illustrates a high-level abstraction of the UAV hardware architecture [9, 115]. The main computing platforms are shown in orange: (1) the flight controller or FMU (e.g., Pixhawk) and (2) an optional companion computer that offloads the FCS. The companion can assist with navigation (collision avoidance), support mission tasks (e.g., camera surveillance, topographic mapping), and process telemetry data.

The FCS typically comprises a main processor for UAV control and, optionally, a fail-safe coprocessor dedicated to handling contingencies (e.g., return-to-home on low battery). The main processor handles: (i) communications, processing commands from the manual RC controller and from the companion computer; (ii) aircraft control, typically using Proportional Integrative Derivative (PID) and estimation methods (e.g., Kalman filtering); and (iii) I/O, interfacing critical sensors (e.g., IMU, barometer, gyroscope, compass) and actuators (e.g., BLDCs, servomotors). The power system supplies energy to computation and I/O.

While the flight controller does not require a companion computer for line-of-sight manual flight, autonomous missions typically depend on non-critical sensors such as GPS and obstacle-avoidance sensors. In this mode, the companion computer transmits telemetry to the GCS – often overlaid on a map for navigation – and receives navigation and mission commands, processing them and dispatching lower-level commands to the flight controller or I/O subsystem.

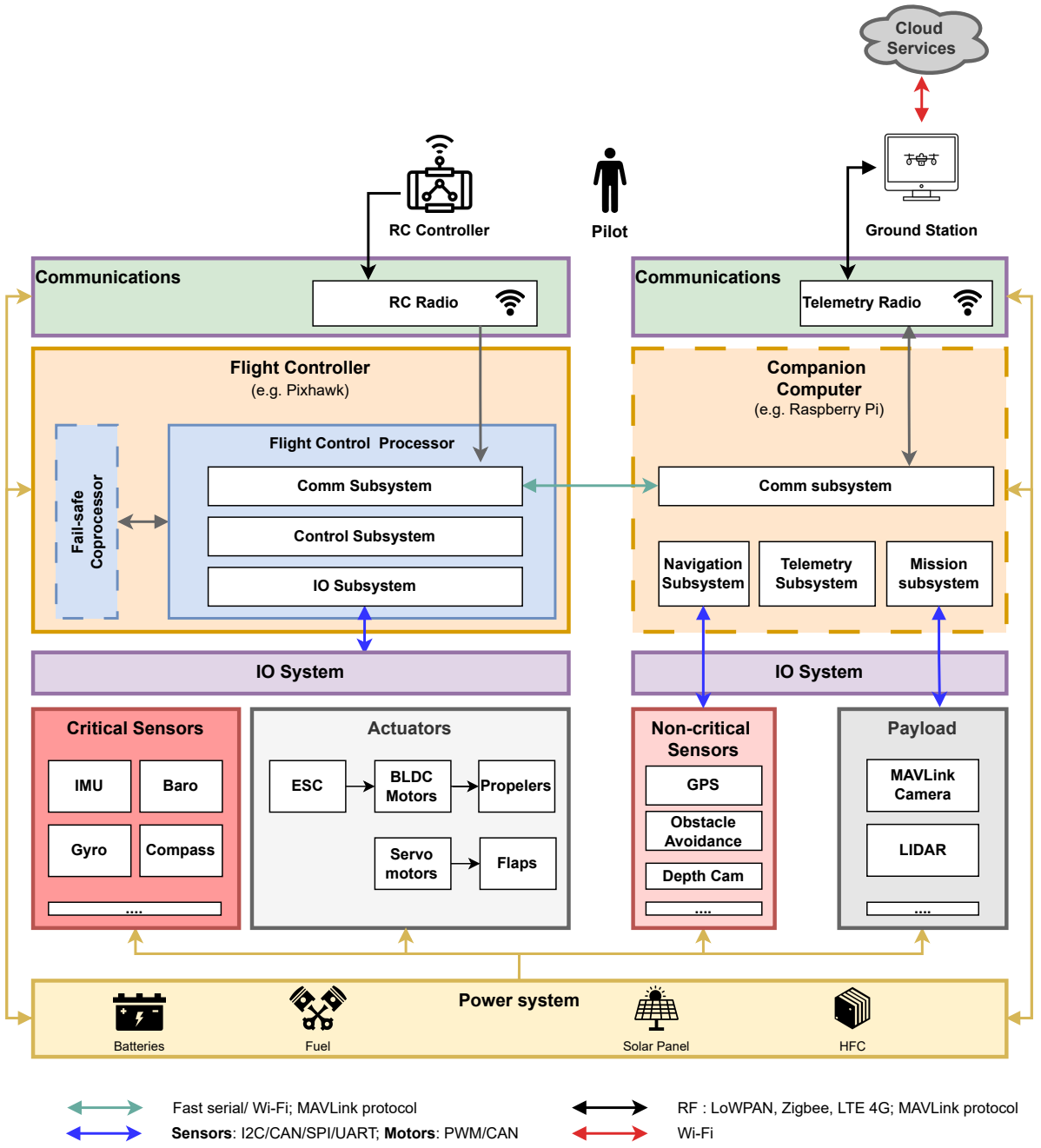


Figure 6: UAV hardware architecture: high-level abstraction.

2.2.4.2 Open-source solutions

Open-Source Hardware (OSH) platforms aim for transparency, extensibility, flexibility, and maintainability while remaining cost-effective. By Open-Source Hardware (OSH) we mean that the hardware design (mechanical drawings, schematics, Bill Of Materials (BOM), Printed Circuit Board (PCB) layout data, Hardware Description Language (HDL) sources) and the software that drives the hardware are released under free/libre terms [137]. Users can purchase or manufacture components individually or acquire Commercial Off-The-Shelf (COTS) kits.

Historically, OSH solutions fall into three broad families [120]: Atmel-based, Arm-based, and companion-computer-based platforms. The first two target the FMU; the last co-packages an FMU and a companion computer.

Atmel-based (legacy). Until recently, 8-bit Atmel platforms such as the ArduPilot Mega (APM) were still in production. APM is built around the ATmega2560 Micro Controller Unit (MCU), typically paired with an IMU and, optionally, a compass and GPS [138]. A key advantage was Arduino Integrated Development Environment (IDE) compatibility, which enabled hobbyists to customize ArduPilot features easily. However, the complexity of modern autopilots requires more capable 32-bit MCUs, leading to the deprecation of many 8-bit designs [139].

Arm-based flight controllers. The vast majority of current OSH projects use 32-bit Arm MCUs (`Pixhawk 4`, `CC3D`, `Paparazzi Chimera`, `CUAV v5 Plus`, etc.), supported by open hardware specifications such as the Pixhawk standards [140]. The `Pixhawk 4` (Figure 7) was developed by Holybro in collaboration with the PX4 team for academic and commercial use [141]. It is released under a Berkeley Software Distribution (BSD) license and integrates two 32-bit Arm processors: a Cortex-M7 STM32F765 as the FMU and a Cortex-M3 STM32F100 as the I/O coprocessor. Pixhawk 4 provides on-board sensors (accelerometers/gyroscopes, magnetometer, barometer) and a GPS module, and exposes standard interfaces (Pulse-Width Modulation (PWM), Controller Area Network (CAN), Inter-Integrated Circuit (I2C), Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI)) for sensors and actuators. UART and telemetry ports also support off-board computers [141] and remote controllers. The board typically runs PX4 atop the NuttX RTOS.

The `CUAV v5 Plus` is another open controller that uses a dual-Cortex-M7 architecture for FMU and I/O, released under a BSD license and commonly running ArduPilot [142]. In contrast, the `Paparazzi Chimera` integrates I/O on a single board around a Cortex-M7 STM32F767 MCU, is released under the GNU Public License (GPL), and runs the Paparazzi autopilot [133, 143]. These Arm-based boards typically feature up to ∼2 MiB of Flash and ∼512 KiB of Static Random Access Memory (SRAM).

Companion-computer-based stacks. These platforms expand I/O via a daughterboard and run the flight stack on an Arm Cortex-A companion under a Linux GPOS (often with PREEMPT or PREEMPT_RT), e.g., Erle-Brain 3 and PXFmini [120, 144]. PXFmini, designed for the Raspberry Pi Zero, weighs about 15 grams and adds an on-board barometer and IMU, expansion ports, and triple-redundant power [145]. Both Erle-Brain 3 and PXFmini have since been deprecated [146].

Raspberry Pi shields. The *PilotPi* shield targets hobbyists by enabling PX4 to run directly on Raspberry Pi with no proprietary drivers and fully open PCB designs and schematics [131]. Figure 8 shows the shield stacked on a Raspberry Pi 4. The intermediate board integrates typical UAV sensors (IMU,

[4]Used under the terms of the Creative Commons BY 4.0 license.

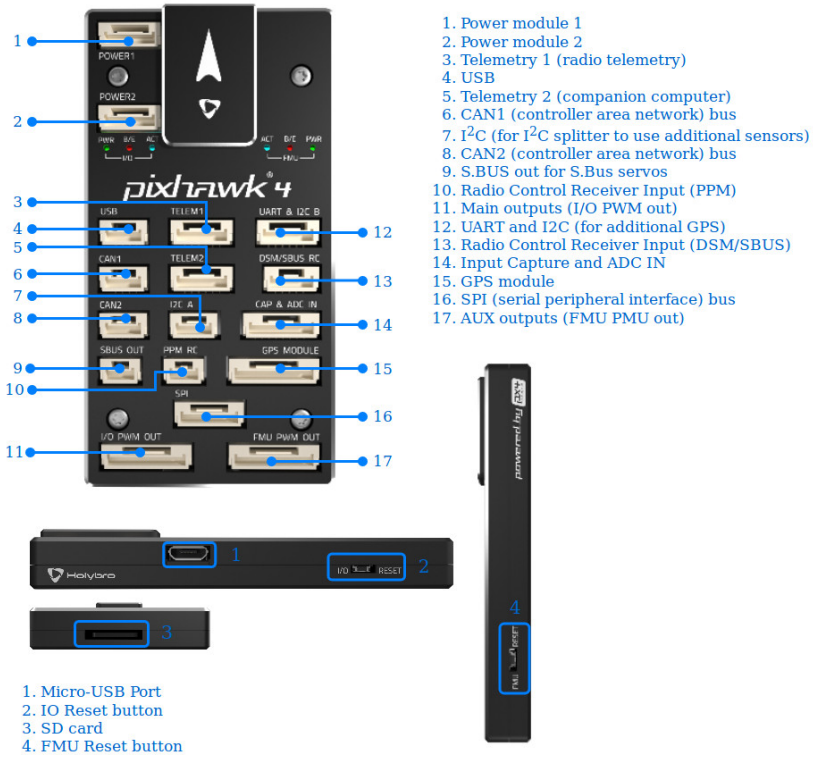


Figure 7: Pixhawk 4 flight controller [141][4]

compass, barometer) and exposes GPS and telemetry radios as UART devices. The top board handles power distribution/management, motor actuation, and peripherals via the General Purpose Input/Output (GPIO) header.

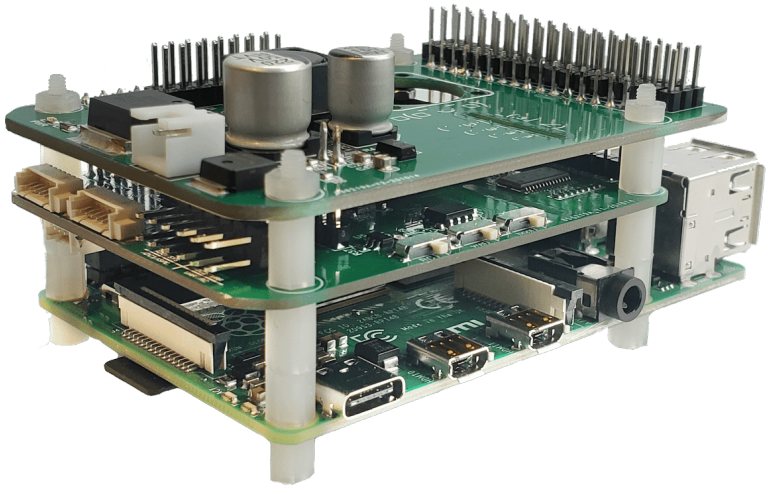


Figure 8: PilotPi shield [131][5]

[5]Used under the terms of the Creative Commons BY 4.0 license.

2.2.4.3 Commercial solutions

Commercial platforms are generally proprietary and thus closed source. This lack of transparency can raise concerns, as illustrated by the U.S. Army's restrictions on DJI drones over cybersecurity issues [12, 147]. Nevertheless, proprietary solutions account for a large share of the market, with DJI the dominant manufacturer [99].

Commercial offerings can be grouped into three categories: microcontroller-based, Field-Programmable Gate Array (FPGA)-based, and companion-computer-based platforms. Historically, 8-bit Atmel designs such as the HobbyKing KK2.1.5 are noteworthy. This board used an ATmega644PA 8-bit AVR (64 KB Flash) to run the KK2 firmware, an MPU-6050 IMU, and included an Liquid Crystal Display (LCD) with push buttons for user interaction [95].

Microcontroller-based. The `SPRacing H7 Extreme` is a performance-oriented flight controller built around an Arm STM32H750 MCU [148]. It includes external 128 MB Flash and common onboard sensors and interfaces. Targeted at the racing market, it supports FPV video via camera inputs [148] and typically runs `Betaflight` or `Cleanflight`, though it is also compatible with `ArduPilot` [149].

FPGA + SoC platforms. The `Aerotenna OcPoC-Zynq Mini` combines an Artix-7 FPGA with a dual-core Arm Cortex-A9 SoC and supports PX4 and ArduPilot [150] (Figure 9). Released in 2017 and now discontinued [151], it offered FPGA-based flexibility for I/O customization and acceleration (including Artificial Intelligence (AI) use cases), while Linux on the Cortex-A9 handled autopilot hosting, logging, and processing. It featured 512 MB RAM, 128 MB Flash, and supported triple redundancy for GPS, magnetometers, and IMUs in addition to conventional sensors [150].

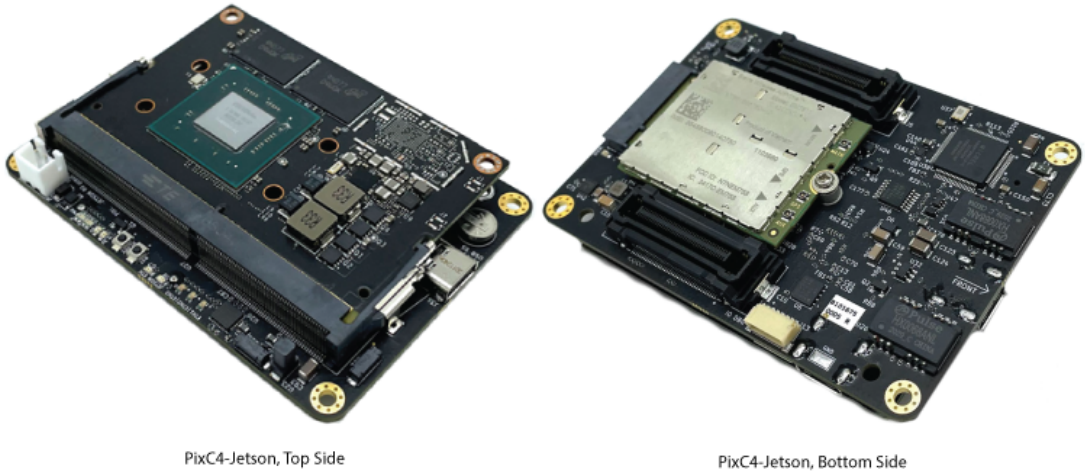
Figure 9: Aerotenna OcPoC-Zynq Mini flight controller [150][6]

Companion-computer-based. Representative examples include `Navio2`, `PixC4-Jetson`, and `Auterion Skynode X/S`. Like PilotPi, `Navio2` is a Raspberry Pi shield, but its hardware design is closed source. It runs a customized Raspberry Pi OS (Debian-based) [152] and supports both ArduPilot and PX4 [153,

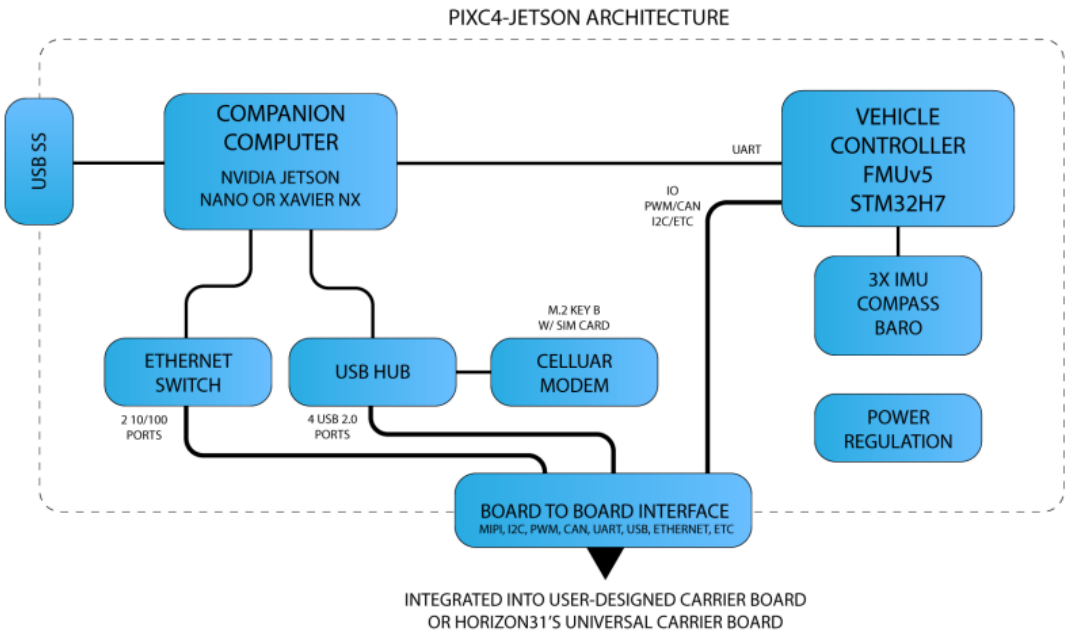
[6] Used under the terms of the Creative Commons BY 4.0 license.

154]. The shield integrates dual IMU, a barometer, a GNSS receiver, an RC I/O coprocessor, I2C and UART for sensors and radios, 14 PWM servo outputs, and triple-redundant power [153].

The `PixC4-Jetson` (Figure 10) is a professional controller that pairs dual-Arm FMU/I/O processors with an NVIDIA Jetson companion [155]. It supports PX4 and ArduPilot and can be configured remotely via terminal access [155], which necessitates additional security hardening. To this end, it provides LTE connection management with a Layer-2 peer-to-peer Virtual Private Network (VPN) and a secure cloud backend [155]. Live video is supported through multi-endpoint encoding pipelines and an optional low-latency WebRTC distribution service. The Jetson companion is well suited to computer vision, machine learning, and AI workloads [156].



(a) Board



(b) Hardware architecture

Figure 10: Horizon31 PixC4-Jetson flight controller [155][7]

Another notable option is `Auterion Skynode X`, which integrates a flight controller, a mission computer, and LTE connectivity in a compact form factor [96]. The FMU, based on Pixhawk FMUv6, uses an STM32H753 microcontroller and an STM32F103 I/O coprocessor, with triple-redundant inertial sensing to reduce failure risk [157]. It provides up to 8 UART, dual-redundant CAN, 100Base-TX Ethernet, 16 PWM outputs, 2 I2C, and 1 SPI interface. The FMU runs an enterprise-hardened PX4 variant (APX4) [157]. The mission computer features a quad-core Arm Cortex-A53 at 1.8 GHz, embedded GPUs (GCNanoUltra for 3D and GC320 for 2D), 4 GB RAM, and 16 GB embedded Multi Media Card (eMMC) + 128 GB internal Storage Disk (SD) storage [157]. Connectivity includes Wi-Fi, Bluetooth 5, and an LTE module; only USB 2.0 is exposed, as USB 3.0 may interfere with GPS [157]. Auterion OS (Linux based) communicates with the autopilot via the Auterion SDK [158]. Supported vehicle types include multicopter, VTOL airplane, and airplane, reportedly up to 500 kg takeoff weight; the price is around USD 1900 [159].

Announced in June 2024, `Auterion Skynode S` is a compact, lower-cost variant integrating FMU and mission computer in a 49 mm × 37 mm footprint [160]. It features an FMUv6x FMU and a mission computer with a dedicated 2.3 Trillion Operations Per Second (TOPS) Neural Processing Unit (NPU) for AI and computer-vision applications [160]. Its AI capabilities have enabled autonomous target tracking, including moving targets, reported to be resilient to GPS jamming or denial [161]. Operational reports claim a rise in mission success probability from 20% to 90% in the presence of electronic warfare [98]. Table 4 summarizes the UAV reference hardware.

2.2.5 UAV Reference Software

In this section we discuss reference software for UAVs. We first present an overview of the software architecture, then analyze open-source and commercial solutions in more depth.

2.2.5.1 Overview and Architecture

Figure 11 illustrates a high-level abstraction of common UAV software architectures [9, 115]. The main computing platforms are shown in orange: (1) the flight controller or FMU (e.g., Pixhawk); (2) the GCS (e.g., a desktop/laptop); and (3) an optional companion computer (e.g., a Raspberry Pi). The companion computer provides capabilities that are crucial for autonomy, such as collision avoidance, odometry, and payload control (camera, LiDAR, etc.). It can also assist navigation in GPS-denied and communications-denied environments using AI-based components (e.g., Auterion Skynode [161]). Physical hardware (sensors, actuators, payload) is depicted in purple, and the software layers are shown in green. Arrows indicate communication links and protocols.

The software stacks on the flight controller and on the companion computer are structurally similar and commonly organized into four layers:

[7] Used under the terms of the Creative Commons BY 4.0 license.

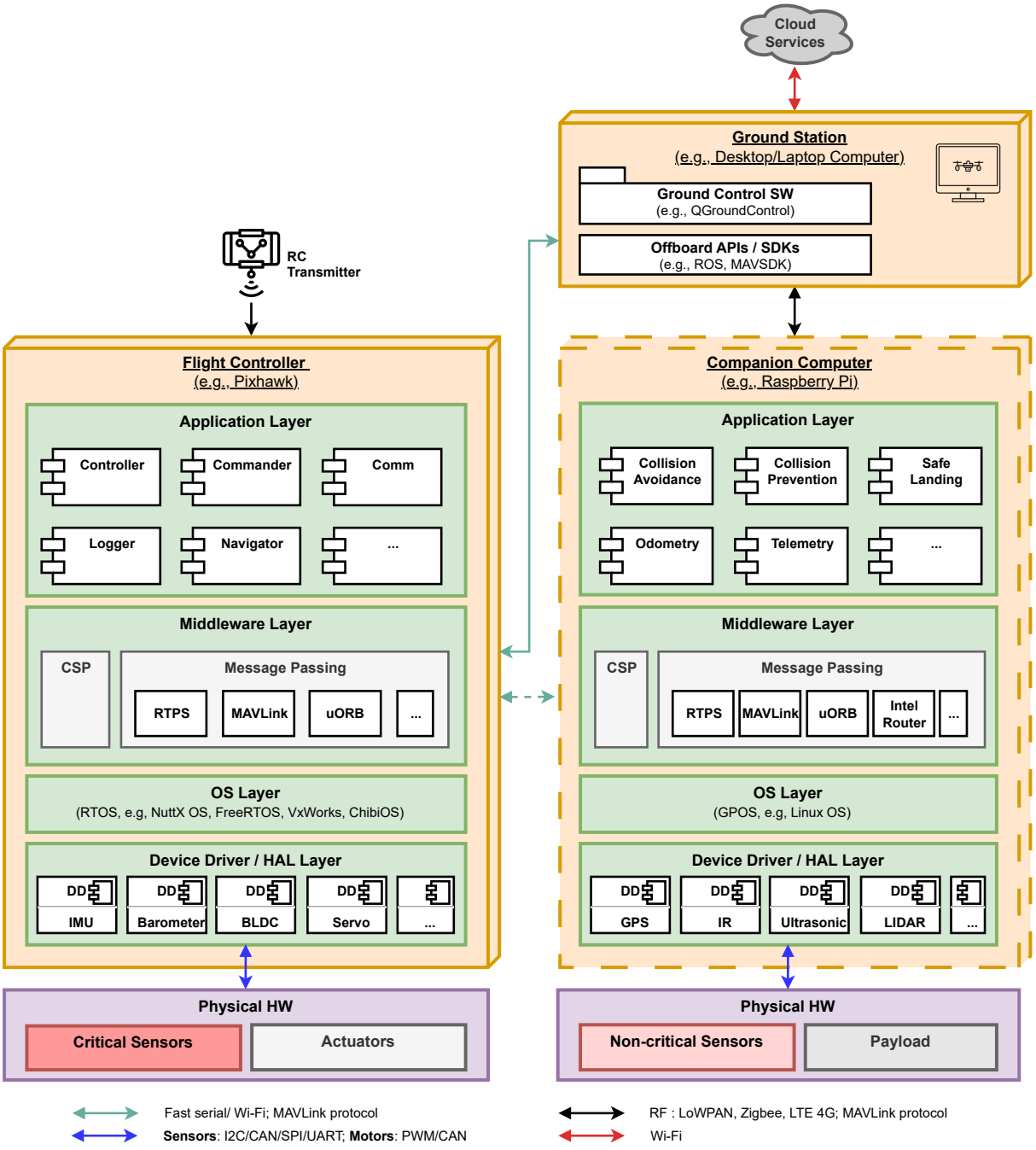


Figure 11: UAV software architecture: common structure

Table 4: UAV hardware reference summary

Platform	Type	Specifications	Software	Sensors	Interfaces	Notes
APM 2.8	OSH	FMU: ATMega 2560	ArduPilot	IMU; Magnetometer; Barometer	UART; I2C	D
Pixhawk 4	OSH	FMU: STM32F765; IO Processor: STM32F100	PX4	IMU; Magnetometer; Barometer; GPS	SPI; I2C; CAN; UART	
CC3D Revo	OSH	FMU: STM32F405RGT6	LibrePilot; ArduPilot	IMU; Magnetometer; Barometer	SPI; I2C; UART; CAN; USB; RF Modem	D
Paparazzi Chimera	OSH	FMU: STM32F767	Paparazzi UAS	IMU; Barometer	SPI; I2C; CAN; UART; XBEE; USB	
CUAV v5 Plus	OSH	FMU: STM32F765	PX4	IMU; Magnetometer; Barometer	SPI; I2C; CAN; UART	
Erle-Brain 2	OSH	FMU + CC: Raspberry Pi 3	Linux + ArduPilot; PX4	IMU; Magnetometer; Barometer; temperature	SPI; I2C; UART; USB; Ethernet; Bluetooth	D
PXFMini	OSH	FMU + CC: Raspberry Pi Zero	Linux + PX4	IMU; Magnetometer; Barometer	I2C; UART	D
PilotPi	OSH	FMU + CC: Raspberry Pi 2/3/4	Linux + PX4	IMU; Magnetometer; Barometer	SPI; I2C; UART; USB; Ethernet; Bluetooth; Wi-Fi; CSI	E
HobbyKing KK2	C	FMU: ATMega 644PA	KK2	IMU; Magnetometer; Barometer		D
SPRacing H7 Extreme	C	FMU: STM32H750	PX4; Betaflight; Cleanflight	IMU; Barometer	SPI; I2C; UART; IR; USB	
Aerotenna OcPoc-Zynq Mini	C	FMU + CC: Dual-Core Arm A9; IO: Artix-7 FPGA with 28K logic cells	Linux + PX4	IMU; Barometer	SPI; I2C; UART; USB; CSI; CAN	D
Navio2	C	FMU + CC: Raspberry Pi 3; RC I/O coprocessor	Linux + PX4; ArduPilot	IMU; Barometer	SPI; I2C; UART; USB; GNSS	
PixC4-Jetson	C	FMU: STM32H743; IO Processor: STM32F103; CC: Nvidia Jetson	ArduPilot; PX4	IMU; Magnetometer; Barometer	SPI; I2C; UART; CAN; USB; Ethernet	
Auterion Skynode X	C	FMU: STM32H753; IO Processor: STM32F103; CC: Quad-core Arm Cortex-A53	APX4	IMU; Magnetometer; Barometer	SPI; I2C; UART; CAN; Ethernet; Wi-Fi; Bluetooth; LTE module	
Auterion Skynode S	C	FMU: Arm Cortex-M7; CC: Quad-core Arm Cortex-A53; NPU: 2.3 TOPS	APX4	IMU; Magnetometer; Barometer	SPI; I2C; UART; CAN; CSI; USB	A

OSH: Open-Source Hardware; C: Commercial; D: Discontinued; E: Experimental; A: AI-capable

1. **Application layer**: high-level functions of the system. On the flight controller, these are safety-critical tasks, such as *Controller*, *Commander*, *Navigator*, *Logger*, and *Communications*. On the companion computer, secondary tasks provide mission and navigation features, such as *Collision Avoidance*, *Collision Prevention*, *Safe Landing*, *Odometry*, and *Telemetry*.

2. **Middleware layer**: abstracts the OS interface and communication via message passing using real-time or asynchronous protocols (e.g., Real-Time Publish Subscribe (RTPS), MAVLink, Micro Object Request Broker (uORB)). It can also enforce Context Security Policy (CSP) mechanisms for additional protection. The flight controller typically communicates with the GCS over Wi-Fi or other data links and, optionally, with the companion computer via high-throughput serial or Ethernet.

3. **OS layer**: provides basic services and a development environment. The flight controller runs an

RTOS (e.g., NuttX, FreeRTOS, VxWorks, ChibiOS) to meet tight control deadlines. The companion computer runs a GPOS (typically Linux) because its applications have soft real-time constraints and benefit from the richer software ecosystem (e.g., Robotic Operating System (ROS)-based avoidance libraries for Linux [115]).

4. **Device-driver (Hardware Abstraction Layer (HAL)) layer**: abstracts the underlying hardware and exposes a stable interface to the OS for sensor data acquisition and actuator control.

The GCS software (e.g., *QGroundControl*) typically runs on a desktop/laptop or mobile device, providing real-time monitoring (often map-overlaid) and flight control. Communication can be direct to the flight controller or routed via the companion computer. Interfaces between onboard components and the GCS commonly use off-board APIs/SDKs, such as ROS or MAVSDK [115].

2.2.5.2 Open-source solutions

Open-Source Software (OSS) platforms, like their hardware counterparts, aim for transparency, flexibility, and maintainability while remaining cost-effective. By Open-Source Software (OSS) we mean that the software source code is released under free/libre terms [137]. The most prominent open autopilots today are PX4 [162], ArduPilot [163], Paparazzi UAS [133], and INAV. Other flight-control firmware – such as Betaflight [164], Cleanflight [165], Crazyflie [166], and LibrePilot [130] – targets the FPV racing domain and assumes expert manual piloting, so full autopilot features are typically unnecessary and not supported. LibrePilot runs on FreeRTOS but is not actively maintained [167], whereas Betaflight , Cleanflight, and Crazyflie run directly on the board.

PX4. PX4 is an autopilot stack for UVs, with a strong focus on UAVs. Started in 2012 and released under a permissive BSD-2-Clause license [162], it is widely adopted in industrial and commercial settings [158, 168]. Like ArduPilot, it supports a broad range of UAV airframes and other vehicles. PX4 is flashed onto flight-controller hardware using *QGroundControl* and typically runs atop NuttX; ROS-based workflows are also available [169]. *QGroundControl* configures PX4 and provides the operator interface; an RC unit can be used for manual control. PX4 uses MAVLink for offboard communication and uORB for intra-stack messaging [115, 169]. Manual modes require an IMU, magnetometer, and barometer; autonomous modes also require GPS. Logs are produced on takeoff and can be analyzed with *Flight Review* or *Px4Tools* [18].

ArduPilot. ArduPilot is another widely used stack for UVs, with strong emphasis on UAVs. Started in 2009 under the GPLv3 license [83], it supports Linux and ChibiOS [163, 169] and runs on many OSH platforms (e.g., Pixhawk). Features – similar in scope to PX4 [170] – include multiple flight modes (manual, semi-autonomous, autonomous) with several stabilization options, programmable 3D waypoint missions with optional geofencing, broad sensor and bus support, fail-safes, and navigation in GPS-denied environments [129]. ArduPilot uses MAVLink to communicate with GCSs and companion computers [129].

Mission commands are stored in Electrically Erasable Programmable Read-Only Memory (EEPROM) and executed sequentially in *Auto* mode [171].

Paparazzi UAS. `Paparazzi UAS` is the oldest still-active open autopilot. Started in 2003 under GPLv2, it supports multiple UAV airframes [82]. It runs atop ChibiOS [172], uses *PPRZLINK* for GCS and companion communication [172], and employs the *AirBorne Ivy* (Application Binary Interface (ABI)) middleware for intra-stack messaging [169].

INAV. Derived from `Cleanflight`, `INAV` targets multirotors, fixed-wing aircraft, and rovers that need GPS-assisted flight rather than acro/racing [134]. It sits between racing-oriented `Betaflight` and full-autonomy stacks such as `ArduPilot`, providing features like position/altitude hold, return-to-home, and waypoint missions configured via the `INAV Configurator` [134]. Hardware support is limited compared with PX4/ArduPilot, focusing on STM32 F4/F7/H7 and AT32 families [134].

2.2.5.3 Commercial solutions

Commercial proprietary platforms typically disclose limited technical detail. Extensibility is offered via vendor SDKs (e.g., DJI [100], Parrot [173]), which enable access to additional features but require specialized knowledge. Some vendors also leverage permissive OSS licenses (e.g., PX4's BSD) to add features and ship closed, enterprise-hardened variants. A representative example is Auterion PX4 (APX4), a hardened PX4 distribution integrated with Auterion FMUs and their mission-computer stack. Figure 12 sketches the Auterion software architecture.

The `Auterion Skynode` co-packages a Pixhawk FMUv6x-based FMU and a companion/mission computer on a single board [174]. The flight controller runs APX4 (PX4-based) atop NuttX; NuttX is Apache 2.0 licensed, which permits proprietary reuse. The mission computer runs Auterion OS (AOS), a customized embedded Linux distribution that hosts autonomy and payload services (e.g., collision avoidance, odometry, camera control) [174]. These services follow a microservices model and are containerized using Docker to improve isolation and streamline updates. To reduce storage and update overhead, Auterion recommends: (i) packaging related services into a single deployable application (e.g., a *Companion App* distributed as a `.auterionos` image), and (ii) using a shared base `Dockerfile` across services [23]. Applications interact with the autopilot through the `Auterion SDK` APIs. For control and fleet operations, *Auterion Mission Control* provides a GCS tailored to Auterion vehicles [175], while *Auterion Suite* offers fleet management, real-time health monitoring, analytics, and Over-The-Air (OTA) updates [176]. Skynode systems may also be operated with a conventional RC transmitter or third-party GCS software.

If we broaden the view to include any proprietary software component in the stack, notably commercial RTOSs such as `VxWorks`, the landscape expands considerably. Examples include Northrop Grumman's X-47B [97], Airbus Helionix [132], and Airbus Atlante [177]. Table 5 summarizes the UAV reference software.

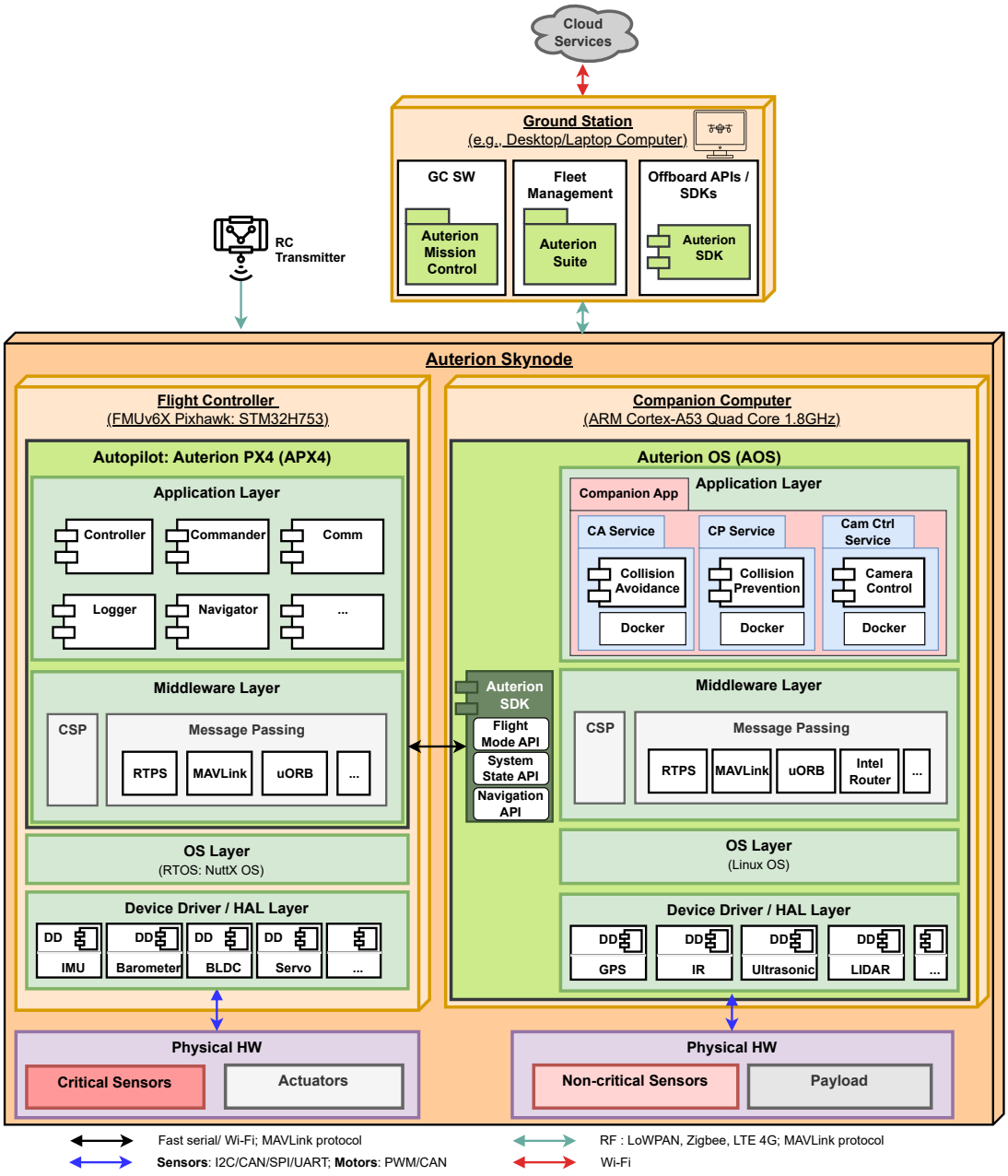


Figure 12: UAV software architecture: Auterion software stack.

2.3 Related Work

We review prior work on UAV flight stacks, onboard platforms, and virtualization approaches to understand what is required to safely consolidate mixed-critical workloads on a single platform. We focus on deployments where flight-control (safety-critical) and mission/auxiliary functions (non-critical) must coexist with bounded interference and predictable timing. We define *trustworthiness* as the conjunction of: (i) temporal, spatial, and fault-containment isolation with verifiable real-time behavior; (ii) deterministic execution and analyzable latency budgets; (iii) a small, auditable TCB and a maintainable codebase; (iv) platform fit under SWaP-C constraints; and (v) openness to enable scrutiny and broad adoption. We use these criteria

Table 5: UAV reference software summary

FCS	License	OS	Autopilot	Notes
ArduPilot	GPL 3	ChibiOS; Linux	Yes	Autopilot
PX4	BSD 3	NuttX; Linux	Yes	Autopilot
Paparazzi UAS	GPL 2	ChibiOS	Yes	Autopilot
INAV	GPL 3	Baremetal	Yes	Autopilot
Crazyflie	GPL 3	Baremetal	No	Racing
Betaflight	GPL 3	Baremetal	No	Racing
CleanFlight	GPL 3	Baremetal	No	Racing
KK2	GPL 3	Baremetal	No	Not maintained
LibrePilot	GPL 3	FreeRTOS	No	Not maintained
Dji	Proprietary	ND	Yes	Provides SDK
Parrot	Proprietary	ND	Yes	Provides SDK
APX4	Proprietary	NuttX	Yes	Provides SDK

to assess prior work throughout this section.

2.3.1 Flight Control Software Used in Practice

The most widely used open-source autopilots are PX4, ArduPilot, Paparazzi UAS, and INAV. PX4, widely adopted in industrial and commercial settings, runs on NuttX (and Linux), offers broad platform coverage, and has a modular architecture. ArduPilot targets Linux and ChibiOS, with extensive support for OSH platforms (e.g., Pixhawk) and rich vehicle types. Paparazzi UAS, one of the earliest open stacks, runs atop ChibiOS with a strong fixed-wing heritage. INAV sits between racing firmware (e.g., Betaflight) and full autopilots, emphasizing GPS-assisted flight but supporting only a subset of boards (notably STM32 F4/F7/H7 and AT32).

Jargalsaikhan et al. [169] analyzed module-level portability across PX4, ArduPilot, Paparazzi, and a custom stack, evaluating structure, messaging, and HAL design against portability requirements (simplicity, modularity, centralized messaging, standard interfaces, HAL quality, documentation). They found that only *ArduPilot* and *PX4* provide HALs and that *ArduPilot* lacks message centralization. Moreover, all surveyed systems rely on non-standard messaging, which hinders portability despite being open source.

In 2021, Wang et al. [20] conducted an exploratory study of autopilot software bugs, analyzing 569 fixed issues from PX4 and ArduPilot and identifying 168 UAV-specific bugs. Root causes spanned eight classes (e.g., missing initialization, limit/parameter mishandling, hardware/software-dependent priority), highlighting recurring pain points. They reported that 39% of those bugs pertained to PX4 (with ≈609 kLOC) versus ArduPilot (> 1505 kLOC), and that PX4 had more resolved issues overall (≈5k vs. ≈3.5k), suggesting a highly engaged community. Such scrutiny is only possible with open-source stacks, improving transparency and, by extension, trustworthiness.

Much academic work with PX4 and ArduPilot targets flight control [178–181] and simulation-based verification [180, 182, 183], often to develop adaptive control and to broaden functional test coverage via SITL/HITL. D'Angelo et al. [181] explicitly cite PX4's BSD-3 license as enabling code modifications without

mandatory upstreaming, helping companies protect intellectual property.

In the commercial domain, flight-control software is typically closed-source, with extensibility exposed via SDKs [173]. Auterion, by contrast, maintains a proprietary PX4 fork for the flight controller and deploys mission applications via containers on the companion computer. Buquerin [21] evaluated the VxWorks 7 RTOS (commercial, avionics) and reported that basic attacks (e.g., buffer overflows, string vulnerabilities) caused noticeable performance degradation, with no built-in protection against malware or command injection. This underscores the need for supervising technology (e.g., a hypervisor) to contain faults across domains.

With open-source RTOSs, transparency increases but isolation is still limited. Zhang et al. [184] compared NuttX and ChibiOS: ChibiOS generally outperformed NuttX on timing, avoided priority inversion with mutexes, and exhibited lower context-switch overhead; NuttX lacks priority inheritance with mutexes and has a larger codebase. Thus, an RTOS alone does not provide the inter-domain isolation required for MCSs.

From a mixed-criticality perspective, PX4 and ArduPilot emerge as the most mature and widely supported open-source autopilots, but non-standard messaging and large codebases complicate portability and assurance. Paparazzi remains influential historically yet narrower in scope, while INAV targets GPS-assisted use cases on a limited set of boards. Openness consistently improves auditability and community response, but real-time determinism and isolation depend on the underlying OS (NuttX/ChibiOS/Linux) and on supervisory mechanisms (e.g., hypervisors). Because none of these stacks alone provide strong inter-domain isolation, we next examine the hardware platforms and hypervisor-based partitioning for mixed-critical deployments.

2.3.2 Hardware Platforms for Onboard Integration

Common architectures fall into three categories. First, a dedicated FMU (running the autopilot) paired with a companion computer for mission functions [141, 142]. Some products co-package both (e.g., PixC4–Jetson [156]) yet still communicate over external links (often UART). Second, single-board Linux systems with add-on shields merge flight control and mission functions on one SoC (e.g., Navio2 [154], PilotPi [131] on Raspberry Pi), trading strong isolation for availability and cost. To consolidate critical and non-critical stacks on a shared hardware platform, a resource-efficient and secure mechanism is needed – typically a hypervisor with disciplined device assignment.

Third, heterogeneous FPGA+SoC platforms integrate high-performance application processing with accelerated I/O or perception. The Aerotenna OcPoC-Zynq Mini, currently discontinued, [150, 151] combined dual-core Cortex-A9 Linux with an Artix-7 FPGA for extended sensors and I/O, running PX4 or ArduPilot. In academia, Kovari and Ebeid [185] used an Avnet Ultra96-V2 (Zynq UltraScale+ MPSoC) with ROS on the CPU (OpenCV) and YOLOv3 offloaded to the FPGA, while PX4 remained on an external Pixhawk, illustrating that the mixed-criticality stacks were not consolidated onto the same platform. Cittadini et al. [25] implemented a two-domain stack on a Zynq UltraScale+: Linux (three cores) runs a

YOLOv3 pipeline on an accelerator while a FreeRTOS domain handles low-level control atop the static-partitioning hypervisor CLARE [53]. However, CLARE is closed-source and the flight stack was tailored, limiting scalability.

A complementary approach is to use heterogeneous SoCs that host Linux on high-performance cores and an RTOS on real-time cores, promising cleaner separation with lower SWaP-C if toolchains and hypervisors are mature. Valente et al. [186] exemplify this with *Shaheen*, a RISC-V heterogeneous SoC (RV64 host + RV32 cluster) enabling RTOS+Linux co-existence under nano-UAV power budgets via the Bao hypervisor. To our knowledge, this is the only open-standards heterogeneous platform explicitly targeting UAV stack consolidation to date. This context motivates the hypervisor-based partitioning survey below: a small, analyzable TCB that can deliver strong spatial, temporal, and fault isolation while fitting SWaP-C budgets.

2.3.3 Virtualization for Onboard Mixed-Criticality

Wang et al. [24] analyzed container- and KVM-based virtualization on Arm SoCs (e.g., Jetson TX2). KVM (with vGIC/QEMU) provides VM-level isolation via trap-and-emulate and two-stage address translation, whereas containers rely on Linux namespaces and cgroups (shared kernel). Experiments show containers are closer to native for compute and network throughput, while KVM provides stronger kernel isolation and remains resilient under fork-bomb stress. Auterion deploys mission applications in containers on the companion computer [23]. However, because containers share a kernel, they cannot by themselves enforce strong separation for mixed-critical workloads.

Virtualization provides isolation properties that containerization cannot: temporal isolation (bounded interference), spatial isolation (memory and device protection), and fault containment across VMs. Containers share a host kernel and therefore cannot, by themselves, meet strong separation requirements for mixed-criticality workloads.

Fautrel et al. [187] advocate a type-1 hypervisor with two-level scheduling: VMs are scheduled first with fixed period/slot windows, and tasks in each VM use fixed task priority. They provide schedulability conditions, minimum and maximum VM-period bounds accounting for context-switch overhead, and an algorithm to compute feasible VM periods and slots to meet all tasks' deadlines. To validate the algorithm, the authors use an inspection drone system comprising three mixed-criticality VMs: (1) high – autopilot on bare metal; (2) medium – GCS-UAV communications on OpenWRT; and (3) low – video processing on Debian Linux. The authors advocated PikeOS Level 1 as a DO-178B/C-certifiable option for airborne systems, but PikeOS [59] is closed source, limiting adoption in low-cost UAVs. Cittadini et al. [25] similarly used the closed-source CLARE hypervisor to implement a two-domain UAV stack. With static allocation and FPGA pass-through, they reported near-zero overhead relative to a non-hypervisor (bare-metal) baseline and µs–ms shared-memory latencies.

Klein et al. [19] used the open-source seL4 microkernel combined with the CAmkES VMM to deploy

two VMs on the mission computer: a non-critical Linux guest for camera and Wi-Fi; and a critical bare-metal application handling CAN-over-Ethernet communications and cryptography. However, the threat model excludes radio-frequency jamming/DoS on the GCS link and assumes a correct, uncompromised autopilot. These are strong assumptions given known bug densities in open-source autopilots [20] and vulnerabilities in commercial RTOSs [21].

Farrukh and West [188] presented FlyOS, an architecture that consolidates critical and non-critical stacks on a heterogeneous multicore platform: the non-critical stack runs in a Linux VM, while the critical stack runs in the Quest RTOS [189–191]. Both VMs execute atop the Quest-V separation-kernel hypervisor [192]. The authors tailored Cleanflight to the application's needs and used shared memory to send mission commands from the Linux VM to the critical VM. They added fault tolerance via VM-level redundancy (heartbeat monitoring and re-instantiation). The prototype targets the Intel Aero Compute Board (now deprecated) [193]. While compelling, the design has limitations: (1) customizing the flight-control firmware reduces portability; (2) because Cleanflight is not a full autopilot, the Linux guest must continuously compute and emit setpoints; (3) and Quest OS currently supports only x86 [190] and, despite being open-source, is unmaintained [191], which weakens trustworthiness.

Valente et al. [186] followed a complementary approach: when designing a heterogeneous RISC-V SoC for secure nano-UAVs, they advocated the Bao hypervisor as the key element to isolate mixed-critical stacks. By design, this attests to the centrality of hypervisor-based isolation in UAV consolidation.

2.3.4 Synthesis and Gap Analysis

Table 6 compares representative systems against consolidation-relevant criteria for UAVs.

The conventional *FMU + companion* split provides strong fault isolation by hardware separation, but incurs SWaP-C costs and introduces latency/bandwidth limits across board boundaries. Containerization helps package mission software but shares a kernel and thus cannot, by itself, enforce strong temporal/spatial isolation. Hypervisor-based partitioning emerges as the practical path to consolidate mixed-critical workloads on one platform; within this space, static-partitioning hypervisors (e.g., Bao) are attractive due to small TCBs, 1:1 vCPU:pCPU mapping, pass-through I/O, and reliance on two-stage translation for memory/device isolation.

Two cross-cutting gaps remain: *trustworthiness* and *openness*. Industrial-grade options like PikeOS and CLARE are not open-source, limiting transparency and community audit. seL4 + CAmkES VMM demonstrations typically protect the mission computer while leaving the flight controller off-board, i.e., without full consolidation on a single SoC. FlyOS/Quest-V shows an integrated design but depends on x86-only, unmaintained components and a customized flight stack, which constrains portability. These gaps motivate a trustworthy, open solution with a small auditable TCB and static partitioning on accessible hardware. In this context, Bao provides a suitable foundation, following the footsteps of the Shaheen heterogeneous SoC[186] premises for nano-UAV applications: static partitioning plus carefully selected device assignment can meet SWaP-C budgets while preserving isolation. Given the SWaP-C, openness,

Table 6: Prior systems vs. consolidation-relevant criteria (UAV focus)

System/Work	Open?	Isolation model	RT evidence	TCB	Platform	Notes
FMU + Companion (no HV)	—	*Board separation* (no virtualization); autopilot on dedicated FCU; missions on companion	FCU RT via PX4/ArduPilot/RTOS; isolation by hardware boundary	Split across boards (small FCU RTOS; larger companion OS)	Pixhawk + Jetson/RPi (typical)	Widely deployed; e.g., Pixhawk 4, PixC4–Jetson [141, 156]
Docker vs. KVM on TX2 [24]	—	Containers (shared kernel) vs. KVM VMs	Throughput + stress; containers ≈ native; KVM resilient to fork-bomb	KVM larger; containers share kernel	Jetson TX2 (Arm)	Mission-computer study
Two-level sched. [187]	—	Type-1 static; VM time slots + fixed-priority tasks	Schedulability bounds incl. context-switch cost	HV-dependent	Generic (Arm)	Algorithms validated on UAV inspection prototype
PikeOS [59]	No	ARINC-653 time/space partitions (table-based)	Certified RT services; partition slots	Mid/High	Arm (var.)	Industrial/avionics deployments
seL4 + CAmkES VMM [19]	Yes	Microkernel + VMM; native components + Linux VM	Kernel proofs; containment demo on UAV	Very small (verified kernel)	Arm	Virtualization on mission computer only; unconsolidated HW
FlyOS + Quest-V [188]	Partial	Separation-kernel VMs (Linux + Quest RT)	Fault tolerance (VM heartbeat/restore)	Mid (kernel+guests)	Intel Aero (x86)	Custom Cleanflight-based firmware; maintenance status unclear
CLARE on ZCU104 [25]	Partial (HV)	Type-1 static; Linux (AI) + FreeRTOS (control)	Near-zero HV overhead; μs–ms cross-domain; ∼10 ms fail-safe	Mid (HV+OSes)	Zynq US+ MPSoC	UAV tracking demo
Shaheen SoC [186]	Yes	HW virt (RISC-V H-ext) + Bao-style static partitions	Power/throughput + timing-channel controls	Small HV + SoC IPs	RV64 host + RV32 cluster	Nano-UAV silicon platform
Bao (static partitioner)	Yes	Type-1, static; 1:1 vCPU:pCPU; static memory; cache coloring	Embedded benchmarks; low HV overhead	Small (∼8k C + ∼0.5k asm)	Armv7/8-A/-R; RISC-V	Basis for mixed-critical stacks [52, 69, 80]

and isolation requirements, we adopt Bao for enabling the consolidation of mixed-criticality software stacks in UAV applications.

3

Design

"*Simplicity is the ultimate sophistication.*"
– Leonardo Da Vinci, polymath

The previous survey motivates using a small, open-source, statically partitioned hypervisor as the consolidation mechanism. We now translate those findings into a concrete architecture for video-surveillance UAVs under tight SWaP-C: the SSPFS, a flight stack that leverages the Bao hypervisor so that resource-intensive, non-critical streaming can coexist with safety-critical flight control on consolidated hardware. We begin by stating requirements and constraints for the target application, then analyze the conventional split architecture and its drawbacks. We implement an unsupervised single-platform baseline (USPFS) to quantify consolidation overheads, and subsequently deploy the supervised counterpart atop Bao (SSPFS). Next, we select the hardware (the UAV and the UAVIC) and map these components into the USPFS and SSPFS designs. To support shared PCIe devices across VMs, we introduce a supervised mailbox mechanism. We conclude by adapting the hardware mapping to constraints imposed by the UAVIC platform and the Bao hypervisor.

3.1 Requirements and Constraints

Video-surveillance missions require two capabilities: (i) geolocation control to survey a designated area and (ii) image acquisition to gather pertinent information about that region. Both can be achieved via *offline* or *online* command methods, or a combination of the two [194]. In the *offline* approach, the target area and the information to be captured are well defined and can be specified *a priori* to the UAV [194]. In cartographic applications, for example, the UAV systematically scans the area to collect topographic data [7]; the vehicle can then operate fully autonomously and store data locally [195]. The *online* approach suits dynamic and unpredictable environments where the area of interest and relevant cues are not fully predefined [194]. In rescue missions, target identification is critical and requires active supervision by the GCS [16]; the GCS must be able to remotely command the UAV and receive real-time feedback, including

Table 7: System requirements and constraints for the UAV flight stack

Categorization	Functional	Technical
Requirements	• Remote UAV command and geolocation in (soft) real time • Real-time image acquisition • Onboard flight-stabilization mechanisms • Battery-powered flight autonomy	• Fault tolerance: video compromise must not affect flight control • Minimal added latency from security mechanisms • Consolidation of flight and companion functions on a single hardware platform
Constraints	• Minimal weight to preserve flight autonomy • Bandwidth-optimized video transmission	• Open-source flight stack (e.g., PX4) • Wireless control and image transmission • Use of the Bao hypervisor for security and fault tolerance

telemetry and live video. Given these demands, this work focuses on the online command mode, which imposes stricter operational requirements on both the platform and the software stack.

Table 7 summarizes the requirements and constraints considered in this thesis. Functional requirements include real-time telemetry and command, video surveillance, autonomous flight control, and battery operation. Technical requirements mandate fault-tolerant isolation between components, minimal security overhead, and consolidation of flight-control and companion functions on shared hardware. Functional constraints prioritize weight minimization and bandwidth-efficient video transmission, while technical constraints specify an open-source stack, wireless communications, and the use of the Bao hypervisor for enforcement of isolation.

3.2 System Architecture

Figure 13 illustrates the conventional solution (Unsupervised Multi-Platform Flight Stack (UMPFS)) for video surveillance, which separates concerns across two nodes. The flight-controller hardware node manages flight-critical functions, while the companion-computer node handles secondary and compute-intensive tasks (e.g., collision avoidance, odometry, and video streaming to the GCS).

We adopt the PX4 flight stack for its open model, broad platform support, modular architecture, and industrial adoption. PX4 runs on the flight controller atop NuttX. In a typical setup, the Companion Computer routes communication between the GCS and the FMU: a fast serial or Ethernet link (UART or Ethernet) is established between the FMU and the Companion Computer using MAVLink; the Companion Computer runs auxiliary routing software (e.g., MAVLink Router [196]). The `User` interacts with *QGroundControl* for configuration and supervision and may also use offboard APIs/SDKs (e.g., `MAVSDK`) to interact with the Companion Computer. The RC link is omitted here because manual flight is not central to the

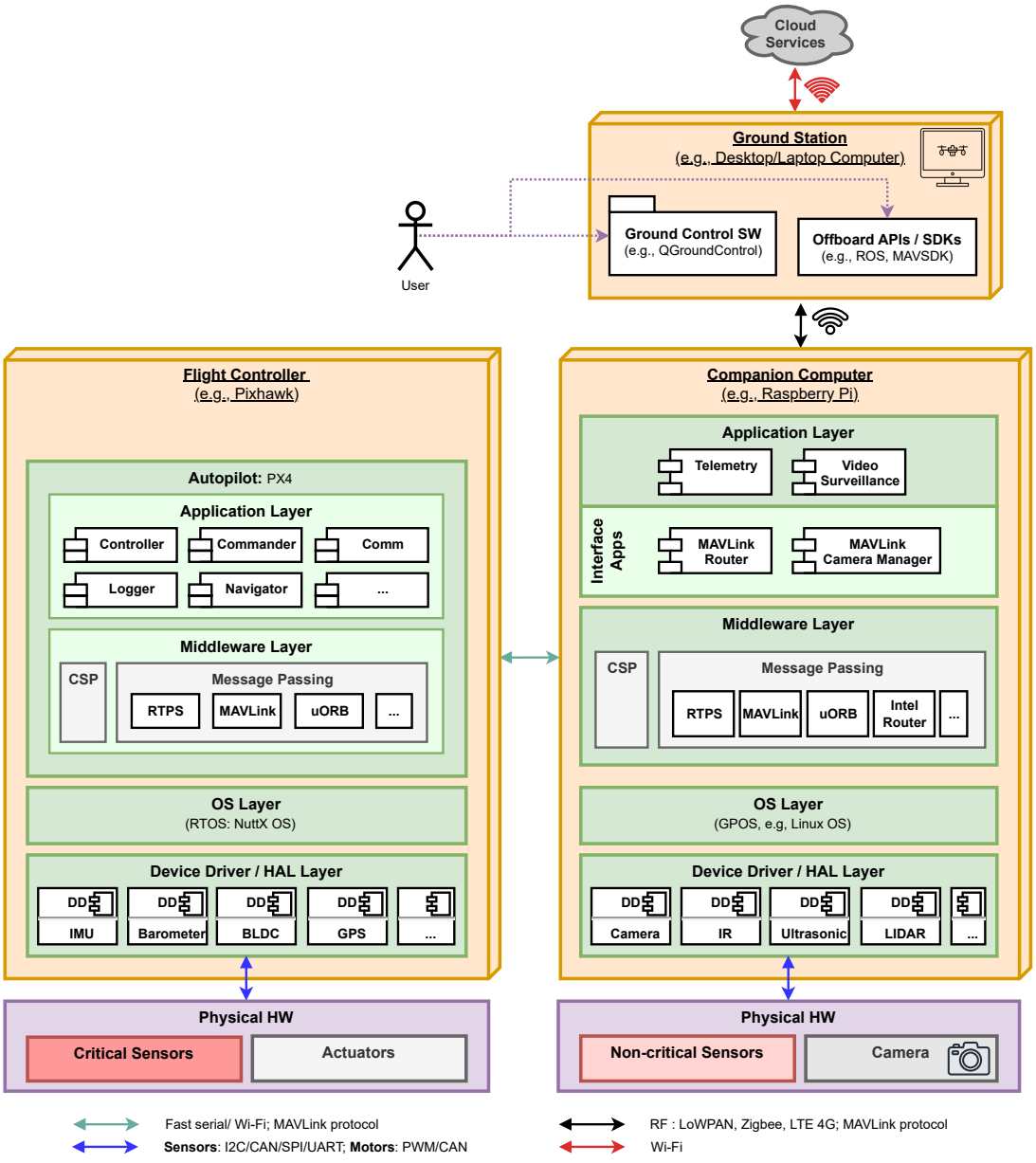


Figure 13: UAV design: conventional solution (full).

video-surveillance use case.

Camera control usually requires an additional service on the Companion Computer, such as the MAVLink Camera Manager [197], which bridges PX4's MAVLink Camera Protocol v2 to the camera's native protocol. This indirection adds complexity and latency. More importantly, it increases risk: if the Companion Computer is compromised, the FMU can be affected via corrupted data or link disruption.

Figure 14 presents a simplified conventional variant specialized for video surveillance, with tighter decoupling between telemetry and payload video. Dedicated links are used for GCS ↔ FMU telemetry (e.g., radio) and for camera streaming (e.g., Wi-Fi). To simplify networking and avoid extra hardware, the GCS and the UAV share the same Local Area Network (LAN). The video subsystem is split into a client on the GCS and a server on the Companion Computer, each exposing a control port (e.g., 5000). The client

sends commands to the server, which configures a capture/encode pipeline and streams frames back; the receiver on the GCS sets up a matching pipeline for decode and display. Occasional frame loss is acceptable for situational awareness, so a connectionless transport such as User Datagram Protocol (UDP) is appropriate (lower overhead and reduced blocking compared with reliable streams). This simplified conventional design serves as the baseline for our subsequent platform unification.

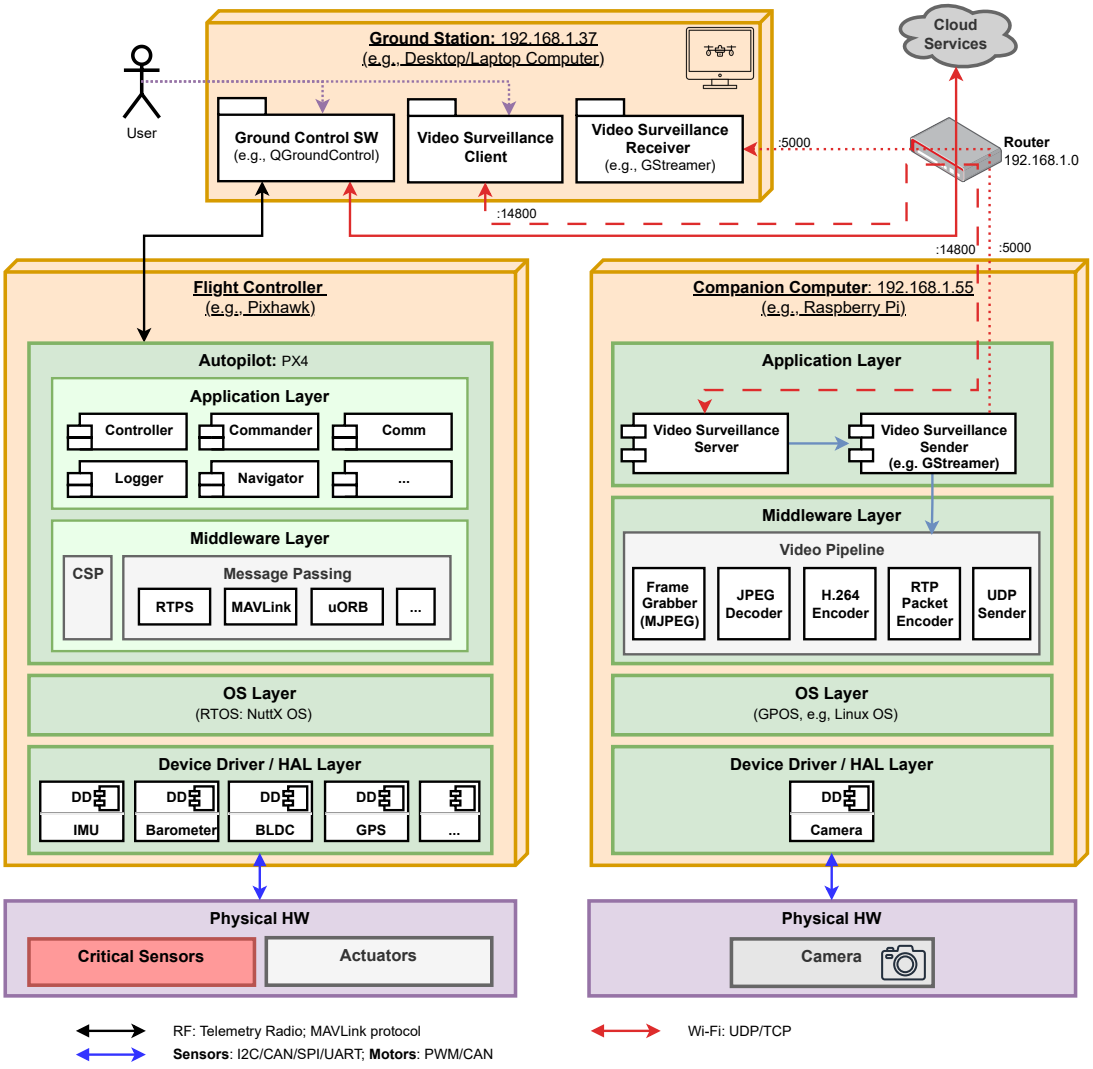


Figure 14: UAV design: conventional solution (simplified).

3.2.1 Unsupervised Single-Platform Flight Stack

Integrating the FMU and the companion-computer functionality on a single platform requires encapsulating both as standalone components. In the Unsupervised Single-Platform Flight Stack (USPFS), these components are realized as processes on a GPOS.

Figure 15 shows the USPFS architecture. Software processes for the GCS and the UAVIC are shown in blue. The UAVIC consolidates FMU and companion roles on one Linux platform: PX4 runs on core 0,

while the remaining cores are allocated to the video surveillance application.

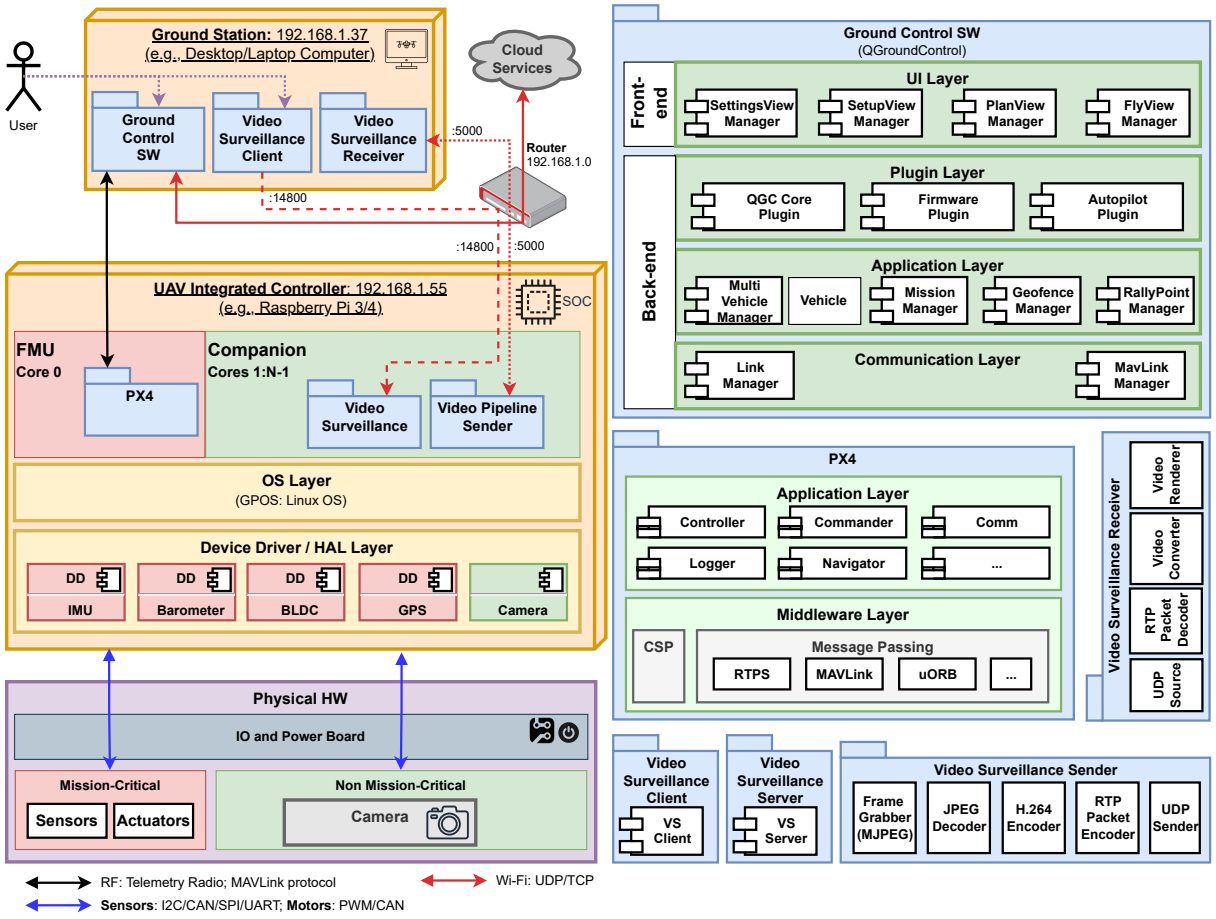


Figure 15: UAV design: unsupervised single-platform flight stack.

Because PX4 no longer runs on NuttX, meeting soft real-time requirements becomes more challenging. To mitigate this on Linux, we (i) dedicate a CPU to PX4, (ii) use a real-time kernel configuration (e.g., `PREEMPT_RT`) with suitable I/O scheduling, and (iii) set IRQ and thread affinities with priority policies appropriate for the control loop.

Despite these mitigations, this architecture provides no strong isolation: faults or overloads in non-critical processes can impact the PX4 process and its resources. This propagation risk makes the unsupervised design insufficient as a final solution; supervision is required to ensure reliable consolidation and safe integration.

3.2.2 Supervised Single-Platform Flight Stack

Figure 16 shows the Supervised Single-Platform Flight Stack (SSPFS). The FMU and companion roles are realized as guest VMs running atop the Bao hypervisor on the UAVIC. This design provides strong isolation between mixed-criticality domains so that faults in the non-critical stack do not impact the FMU.

Each VM runs a Linux-based OS, enabling tailored configurations. The FMU VM can use a real-time kernel and be pinned to a dedicated CPU, while the companion VM can run a standard kernel. Bao's

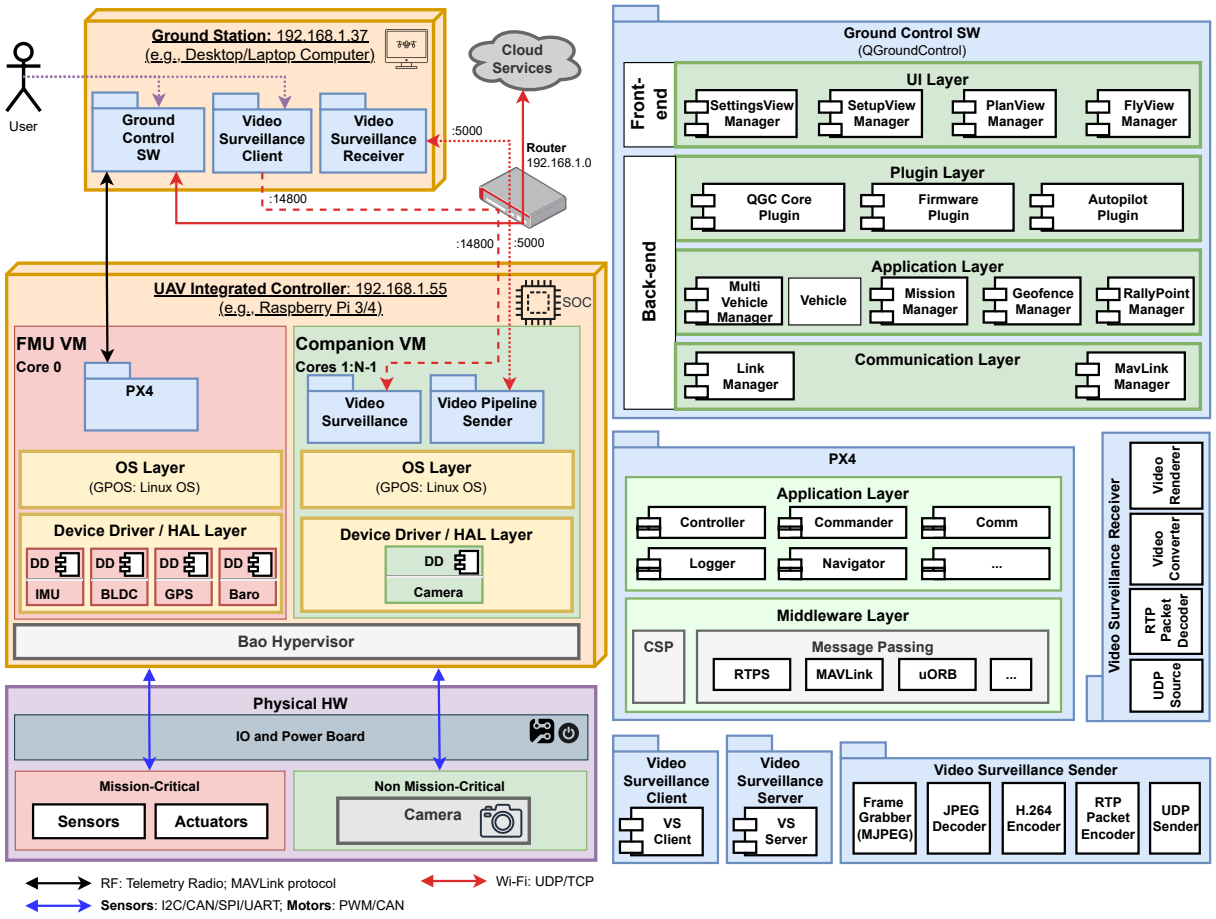


Figure 16: UAV design: supervised single-platform flight stack.

static partitioning assigns CPUs, memory, and devices to each VM exclusively, and device drivers reside within the respective guest, limiting the effects of software bugs.

The main costs are footprint and configuration effort: every VM carries a full OS, increasing binary size and memory use, and the `User` must select and assign hardware resources carefully to avoid contention while meeting performance targets for CPU, memory, and I/O. In practice, the mapping may need small adjustments to the chosen hardware platform.

3.3 Hardware Selection

This section selects hardware for the UAV, the UAVIC, and required add-ons. Choices are mapped to PX4 requirements and to constraints imposed by Bao on the UAVIC.

3.3.1 UAV

We target a multirotor airframe due to its availability, compact form factor, and VTOL capability that reduces takeoff area. Operations are assumed at low altitude for regulatory and operational reasons. We also

favor a high power-to-weight ratio to extend flight endurance. Figure 17 shows the selected platform: `KIT-HGDRONEK66`, commonly known as the NXP HoverGames UAV kit [108]. This professional DIY kit, priced under USD 500, includes the RDDRONE-FMUK66 as the FMU *(1)*.

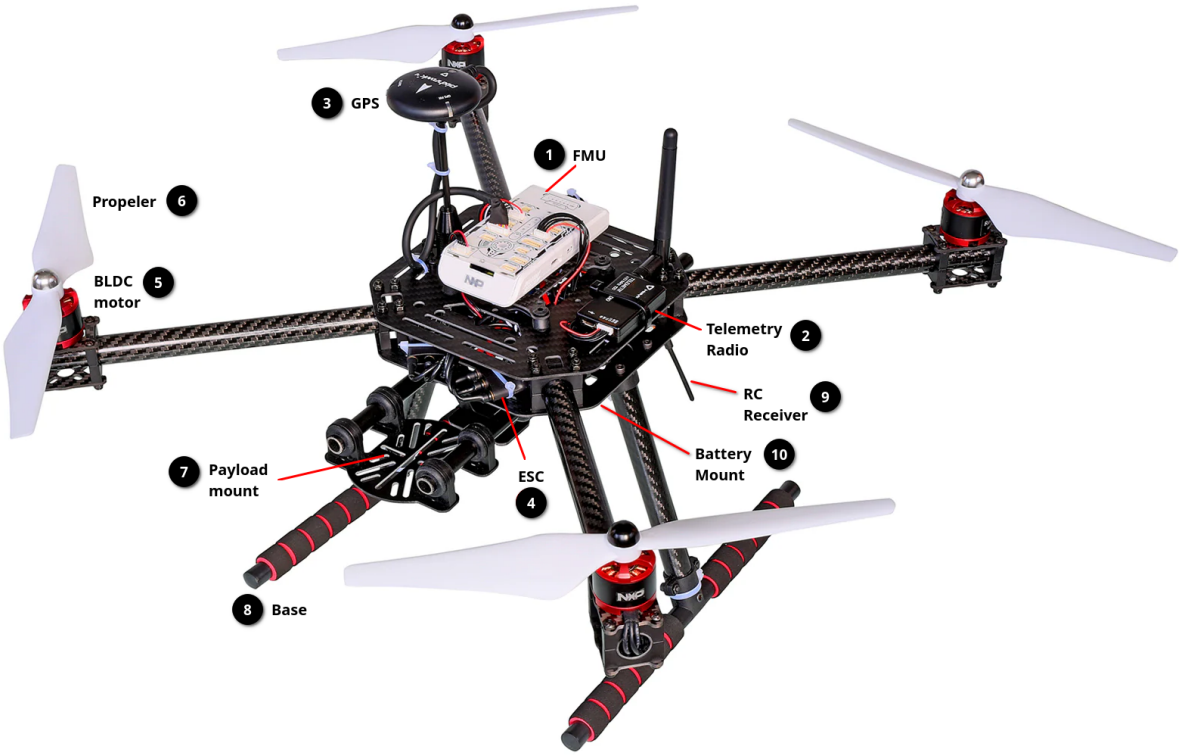


Figure 17: NXP HoverGames UAV kit (adapted from [108])[1]

The kit uses an S500 carbon-fiber quadcopter frame (500 mm wheelbase) with BLDC motors *(5)* driving the propellers *(4)*, each controlled by an ESC *(6)*. For autonomous flight it includes a GPS module *(3)* and a payload mount *(7)* for accessories such as cameras. Power is supplied by a 3S LiPo battery (purchased separately) in the 3500–5000 mAh range. A telemetry radio *(2)* (purchased separately) can be connected to the FMU; in Europe a 433 MHz variant is typically used. The kit also includes an RC transmitter (GS-i6S) and a receiver.

Figure 18 depicts the block diagram. The RDDRONE-FMUK66 FMU integrates an NXP Kinetis® K66 MCU (Arm® Cortex®-M4 at 180 MHz, 2 MB flash, 256 KB SRAM) [198] running the PX4 autopilot. A power-distribution board provides current and voltage sensing for battery-state estimation. The FMU interfaces typical sensors (accelerometer, gyroscope, magnetometer, barometer) and actuators (BLDC via PWM, servos where applicable). Communication with a companion computer is via UART; links to the GCS are provided by a telemetry radio or an RC link. An optional SD card enables flight logging for offline analysis and simulation replay. A SEGGER J-Link EDU Mini Serial Wire Debug (SWD) adapter supports on-target debugging and firmware upload to the FMU.

[1] Copyright © NXP Semiconductors: Used with permission for non-commercial purposes only
[2] Copyright © NXP Semiconductors: Used with permission for non-commercial purposes only

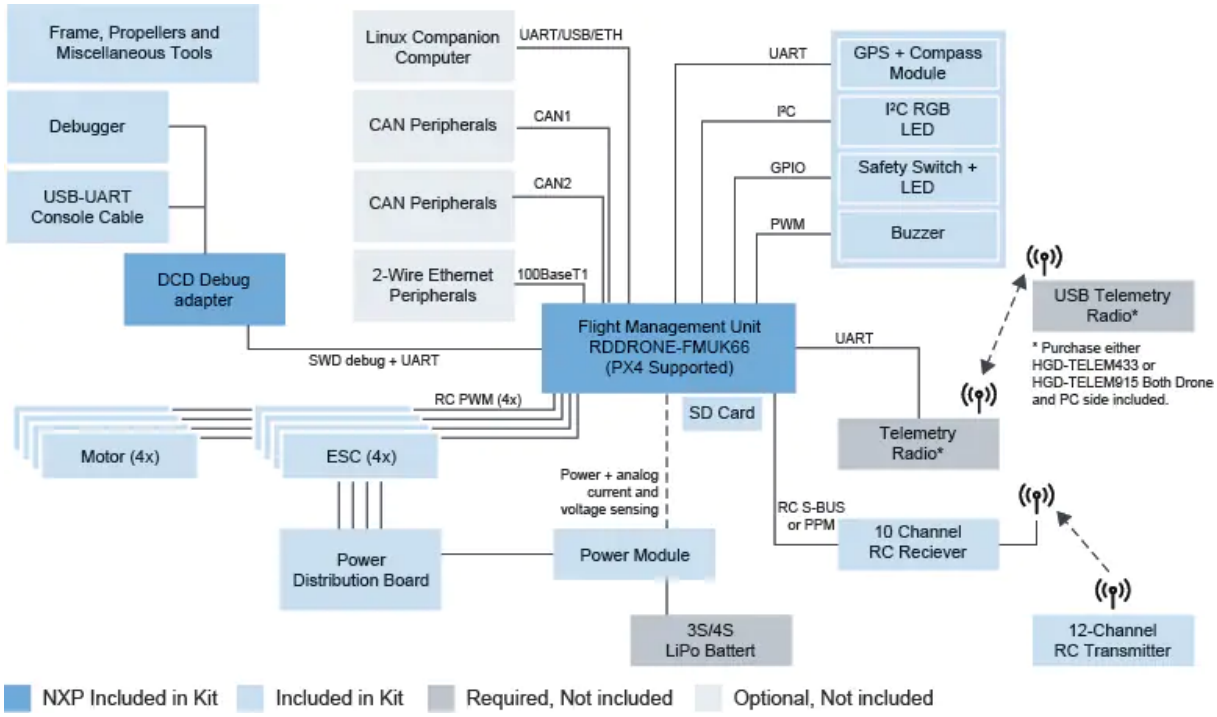


Figure 18: NXP HoverGames block diagram (adapted from [108])[2]

3.3.2 UAV Integrated Controller

The Unmanned Aerial Vehicle Integrated Controller (UAVIC) merges FMU and companion-computer functionality on a single platform. For mixed criticality, the ideal design separates computing domains, e.g., a real-time processor for the FMU and a general-purpose processor for the companion stack. This points to a heterogeneous SoC, such as the NXP i.MX 8M Nano [199], which integrates an Arm® Cortex®-M7 (well suited to NuttX) and a quad-core Cortex®-A53 (well suited to a Linux OS). However, the NuttX RTOS does not currently support this board [200].

Our initial plan was to port NuttX to the Cortex-M7 and then enable PX4 on that target, including the required device drivers. In parallel, Bao support for this board would be needed also. The combined effort (new NuttX port, PX4 enablement, and Bao enablement) proved impractical for this work's scope, so we adopted a more direct path. We next considered running PX4 directly on Linux with a real-time kernel and appropriate I/O scheduling to limit latency. PX4's Linux support, however, is restricted to a small set of platforms (e.g., BeagleBone Blue and Raspberry Pi 2/3/4 with specific shields) [201], which constrained options. Consequently, we consolidated each software stack into its own Linux OS VM, rather than splitting across heterogeneous cores.

As the host platform, we selected Raspberry Pi 4 paired with the PilotPi shield, balancing availability, documentation quality, and cost. Fig. 19 shows the resulting UAVIC. The Raspberry Pi 4 Model B (1) runs a Linux-based OS from the SD card and exposes Camera Serial Interface (CSI) camera and Universal Serial Bus (USB) interfaces. It features the Broadcom BCM2711 SoC with a 64-bit quad-core Arm® Cortex®-A72 at 1.8 GHz and a VideoCore VI Graphics Processing Unit (GPU) at 500 MHz [202, 203], plus 8 GB

LPDDR4-3200 SDRAM. Connectivity includes dual-band 802.11ac, Bluetooth 5.0, Gigabit Ethernet, two USB 3.0 ports, and two USB 2.0 ports.

The sensor board (2) mounts atop the Raspberry Pi, providing GPS, telemetry, and RC external interfaces mapped to `/dev/ttySC0`, `/dev/ttySC1`, and `/dev/ttyAMA0`, respectively. Onboard sensors include an accelerometer/gyroscope (`ICM42688P`), a magnetometer (`IST8310`), and a barometer (`MS5611`) required by PX4. The topmost layer contains the power board (3), which handles power supply (8), monitoring (11), and motor actuation via PWM (10) using the Linux `PCA9685` driver. A header exposes unused pins, enabling additional connections such as a remote serial interface (UART5).

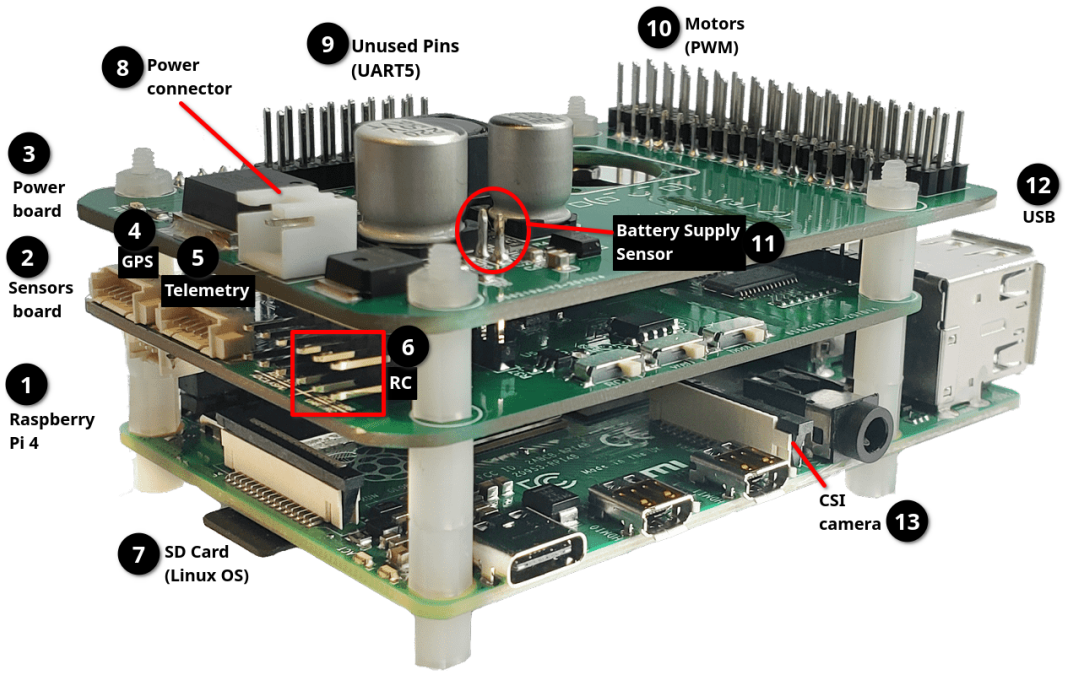


Figure 19: UAVIC: Raspberry Pi 4 + PilotPi shield (adapted from [131])[3]

3.3.3 Hardware mapping

The UAVIC platform must satisfy PX4 requirements for sensors and actuators. Because the Bao hypervisor enforces static partitioning, we must also determine whether the UAVIC hardware can be made fully available to each guest in the SSPFS solution or if alternatives are needed. Accordingly, we first map the PX4-required hardware to the Linux device tree for the USPFS solution and then adapt that mapping to meet Bao's constraints in the SSPFS implementation.

[3]Used under the terms of the Creative Commons BY 4.0 license.

3.3.3.1 USPFS

Fig. 20 depicts the full device tree for the UAVIC system, representing the USPFS solution (base scenario). Solid lines denote aggregation (e.g., the `root` node includes the `memory` node); dashed lines denote dependency (e.g., the `power` and `firmware` nodes depend on `mailbox`). The coloring highlights functional groupings:

- Generic nodes: essential for all Raspberry Pi 4 configurations (e.g., `memory`, `cpus`, `power-regulator`).
- PX4-required nodes: `i2c1` (motor actuation), `spi0` (IMU, barometer, magnetometer), `spi1` (GPS, telemetry radio), and UARTs 0/5 (RC link, debug console).
- Companion VM nodes: `i2c0` and `csi1` (camera), `mmcnr` (Wi-Fi support).
- Firmware nodes: enable GPU–CPU communication via `mailbox` [204].

Device-tree analysis reveals several shared dependencies between PX4 and the Companion VM: the clock manager (`cprman`), the GPIO controller, and the DMA controller. Because Bao's isolation model prohibits device sharing, an alternative architecture is required.

We retain PX4's native device set and migrate Companion VM devices to the USB interface. As indicated by the dependency path (red line), the `usb` device (under `pcie`) shares only the `firmware` node with PX4, which itself depends on `mailbox`. This simplifies the SSPFS design, but shared `mailbox` access must be reconciled with Bao. Removing `firmware` from either VM would break the system, so this is infeasible. Accordingly, we combine the Companion VM migration to USB with supervised mailbox access in Bao. This preserves hardware isolation while enabling secure use of the required shared resource.

3.3.3.2 Supervised mailbox access

Fig. 21 illustrates the mailbox access for the conventional case (left) and the supervised one (right). The solid lines denote synchronous events, while the dashed lines denote asynchronous ones. Red arrows indicate the outgoing path (from the mailbox driver), and blue arrows indicate the incoming path. Mailbox transactions are shown as white (conventional), violet (VM1), and grey (VM2) envelopes. The synchronization mechanisms (locks) appear in brown for the mailbox and in blue for Bao. Hypercalls are shown as phones: green for `Start_TX` and red for `End_TX`. Numbered labels mark the event sequence in each case.

In the conventional case, the mailbox device driver can initiate a transaction request if no other is pending *(1)*. The transaction must be completed *(8)* before a timeout occurs, freeing the mailbox for further requests. An interrupt is triggered *(2)* and the CPU forwards the request to the mailbox *(3)* which requests data from the GPU *(4)*. The GPU firmware handles the transaction and returns a result to the mailbox *(5)*. The mailbox forwards the response to the mailbox's driver *(6), (7)*, completing the request and freeing the mailbox [205].

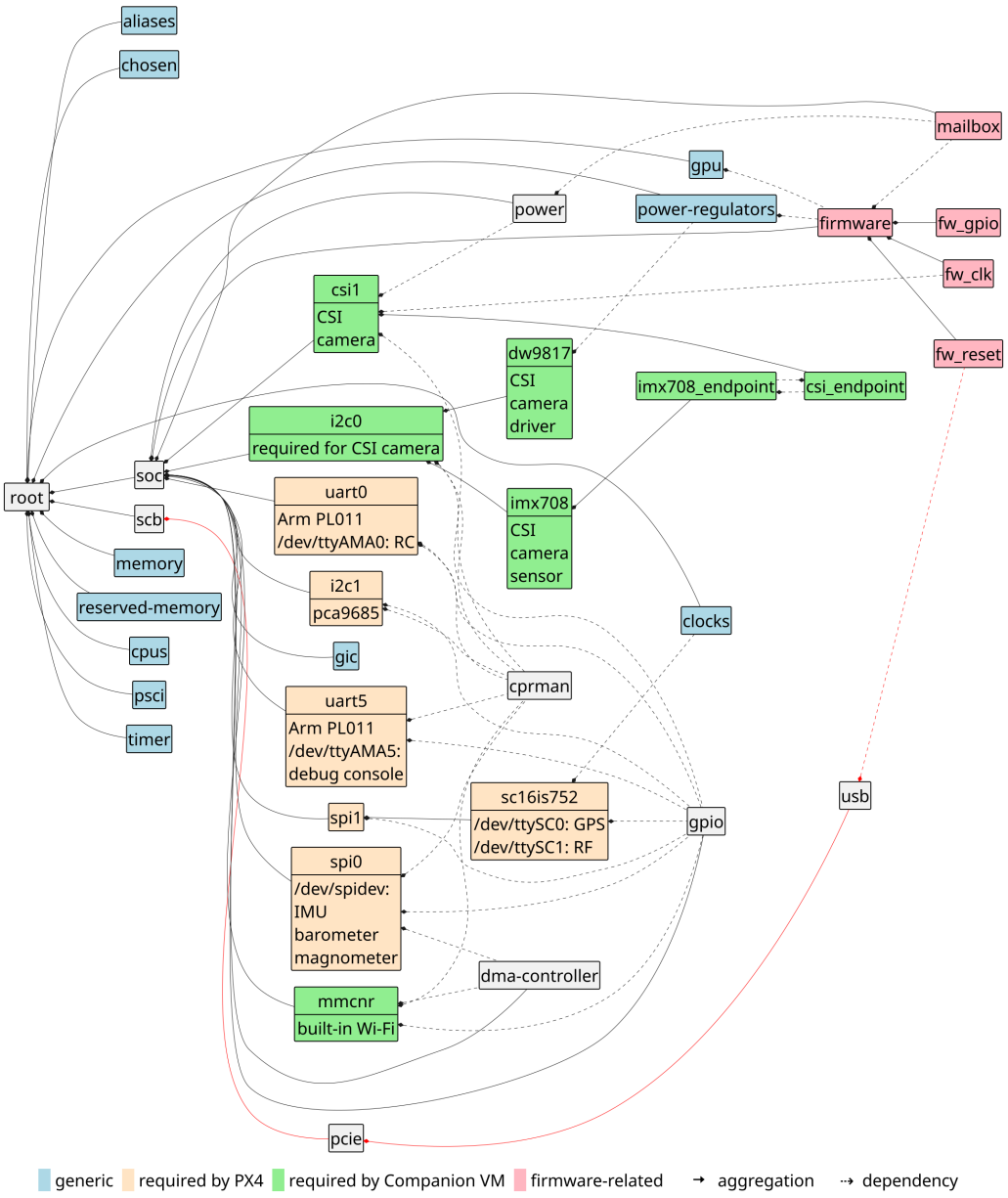


Figure 20: Hardware mapping: USPFS device tree

In the supervised case, VM1's mailbox driver attempts to start a transaction when none is pending *(1)*. Now, before transmission, the mailbox driver must signal to Bao it wants to start a transaction by performing a `Start_TX` hypercall *(3)*. Bao acknowledges this request if no transaction is ongoing, locking the mailbox manager *(5)*. VM2 performs the same attempt *(2), (4)* but must wait because VM1 holds the lock. The mailbox manager handles the hypercall, associates the guest, target device (mailbox), and interrupt ID, and injects the interrupt into the appropriate CPU *(7)*. VM1's driver issues the transaction to the mailbox *(8), (9)* which forwards it to the GPU *(10)*. After the GPU processes the request, the response returns to VM1's driver *(11), (12), (13)*. On completion, VM1 issues `End_TX`, signaling Bao to release the mailbox lock. Bao releases the manager's lock *(15)*, which allows VM2's pending request *(6)* to resume. Bao then injects the corresponding interrupt *(16)* into the appropriate CPU, and VM2's driver proceeds

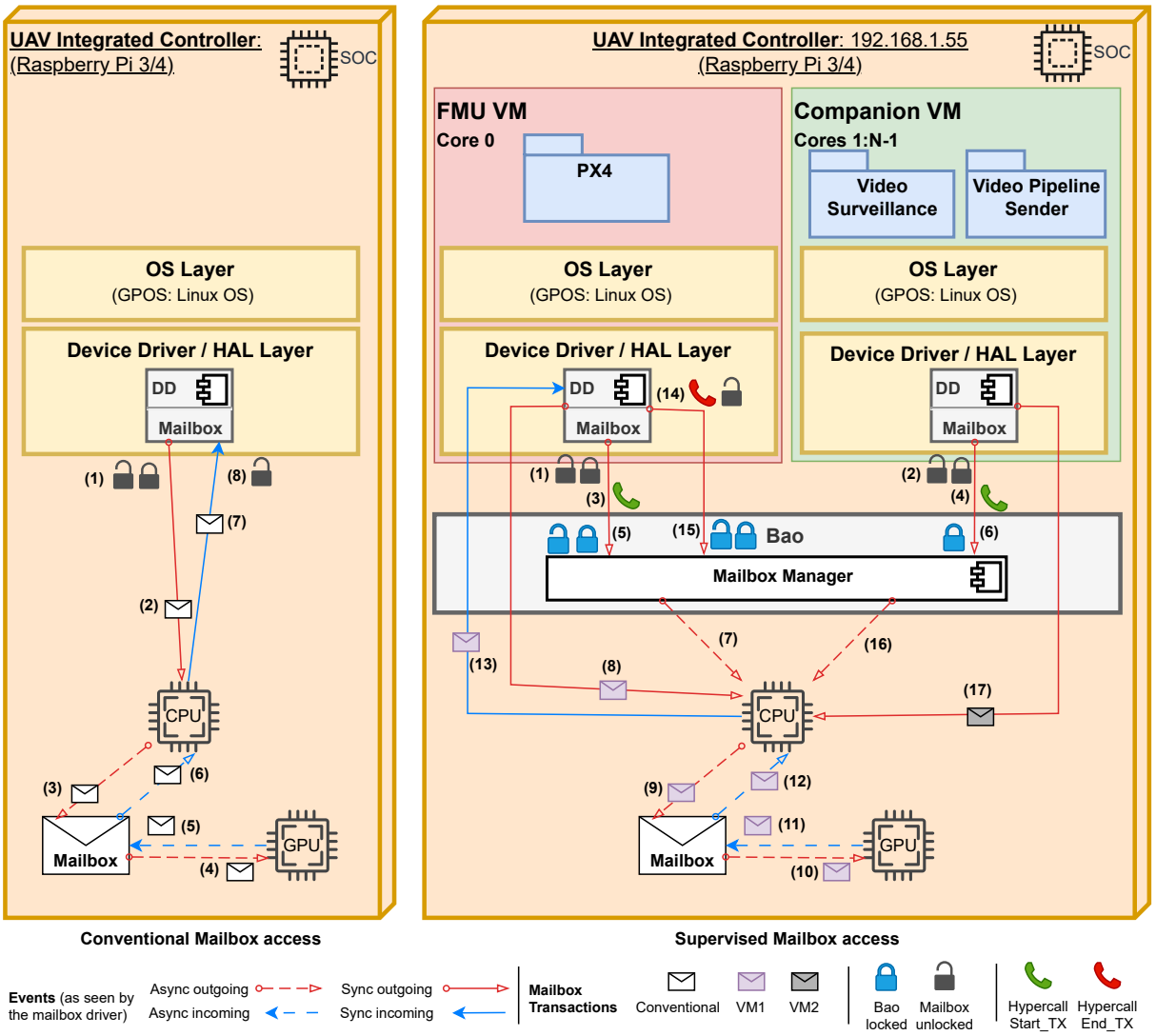


Figure 21: Mailbox access: conventional (left); supervised (right)

with its firmware transaction *(17)*, after which the process repeats itself.

3.3.3.3 SSPFS

After addressing device sharing between VMs via supervised mailbox management, we assign hardware resources to each VM. Fig. 22 and Fig. 23 show the device trees for the PX4 VM and the Companion VM, respectively.

The PX4 VM includes only essential onboard sensors and actuators plus an optional debug console. Memory is split into two RAM regions of 144 MB and 3 GB. Within the first region, a 32 MB Contiguous Memory Allocation (CMA) area is reserved to support DMA transactions for SPI devices, which must operate within the first 1 GB of memory [206]. Processing resources comprise a single Arm A72 CPU (core 0). The Companion VM exposes the USB host for a USB camera and a Wi-Fi dongle. Memory allocation features two RAM regions of 624 MB and 2 GB. Within the first region, a 384 MB CMA area

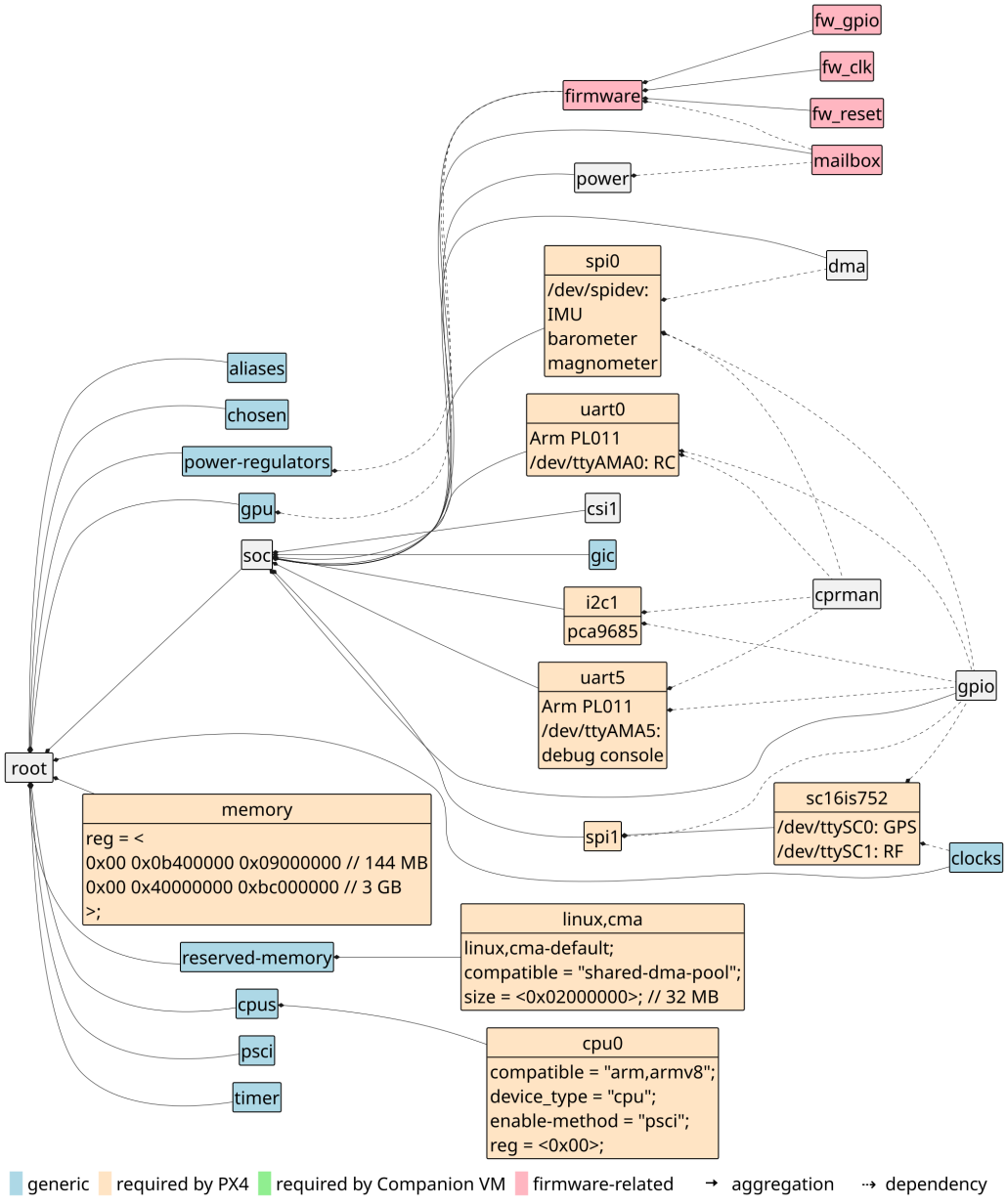


Figure 22: Hardware mapping: SSPFS device tree — PX4

supports the video pipeline, which is also restricted to the first 1 GB of memory [206]. Processing resources comprise three Arm A72 CPUs (cores 1–3).

3.3.4 Addons

The Companion VM requires two USB devices: a camera and a Wi-Fi dongle. Fig. 24a shows the selected camera, the Creative Live! Cam Sync 1080p V2 [207]. It is an affordable USB device supporting full High-Definition (HD) capture (1920×1080 @ 30 Frames Per Second (FPS)) with dual built-in microphones. It is plug-and-play on Linux with in-kernel driver support, and its long cable allows flexible placement on the UAV.

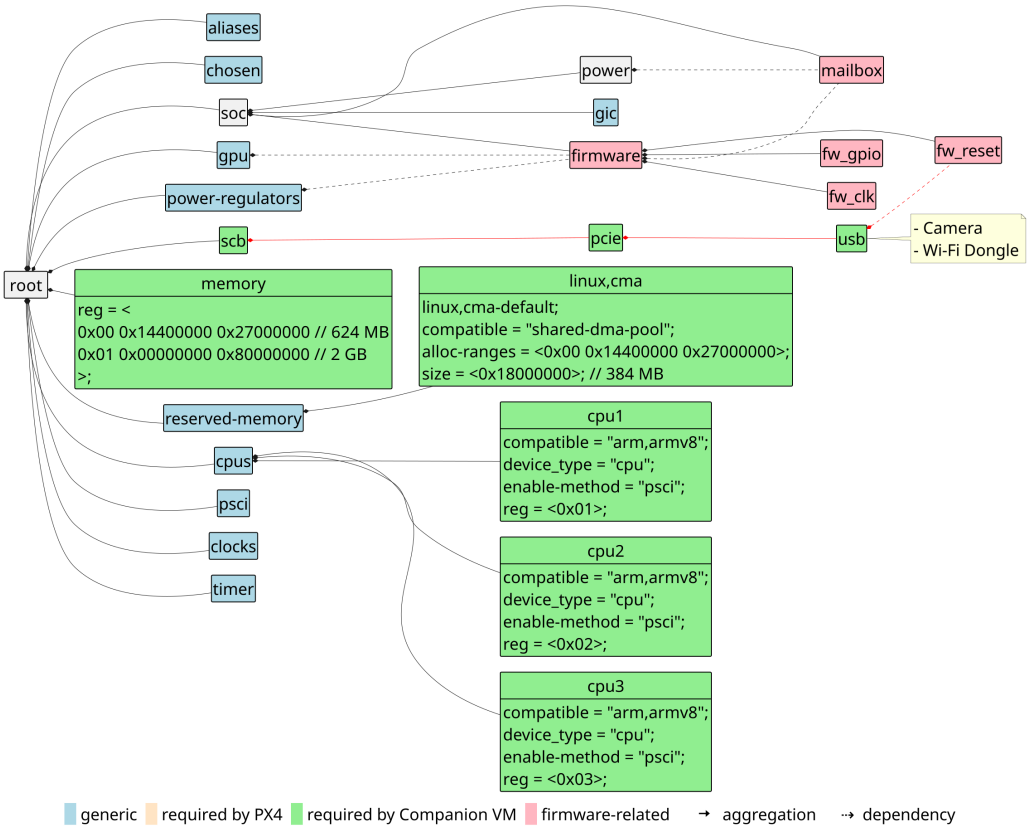


Figure 23: Hardware mapping: SSPFS device tree — Companion VM

Fig. 24b shows the selected USB Wi-Fi dongle, the EDUP AX3000. This dual high-gain model offers tri-band support (2.4, 5, and 6 GHz) and data rates up to 3000 Megabits per second (Mbps) over a USB 3.0 interface [208]. It uses the Mediatek `mt7921au` chipset, has broad OS compatibility, and has been supported in-kernel since Linux 5.18 [209].

(a) USB Camera – Creative Live!

(b) USB Wi-Fi dongle – EDUP AX3000

Figure 24: Addons

4

Implementation

"Talk is cheap. Show me the code."
– Linus Torvalds, software engineer

This chapter details the implementation of the USPFS and SSPFS architectures. We begin with a brief overview of the implementation workflow. Next, we present the base system – hardware and software components shared by both architectures – along with initial validation of the UAV assembly and its configuration. We then describe the USPFS implementation, deploying the flight-control and companion stacks on a custom, embedded Linux–based OS. Finally, we detail the SSPFS implementation: each stack is deployed in a separate VM under the Bao hypervisor, and we implement the mailbox supervisor to coordinate safe access to shared firmware services. The complete source code is documented in the associated repository [26].

4.1 Workflow

Fig. 25 illustrates the overall implementation workflow. The USPFS path comprises the *Guests*, *Firmware*, and *Deployment* stages, while the SSPFS path adds a *Hypervisor* stage. The workflow has four primary stages: (1) build guests; (2) build hypervisor and VMs (for SSPFS only); (3) build firmware; and (4) deployment. Here, "guest" means either a true virtual machine under the Bao hypervisor (SSPFS) or a native binary on the UAVIC platform (USPFS).

We first build the guests and their components. In USPFS, a single binary wraps the PX4 and video-surveillance functionality. In SSPFS, PX4 and the video pipeline execute concurrently, each in its own isolated guest. The PX4 build targets the `PilotPi` board for `aarch64`, producing PX4 binaries and runtime configuration files. These, along with network settings, are staged into Buildroot's root filesystem for inclusion in the guest image. The Linux kernel is configured with essential drivers (SPI, I2C, Wi-Fi, video, etc.). Userspace includes network, login, console, and packages (e.g., `gstreamer` for video, `iwd` for wireless). Buildroot then produces a Linux kernel image `Image_X` (where *X* denotes the guest index). A guest-specific device-tree source `linux_X.dts` is compiled to `linux_X.dtb`. The `b_loader` tool

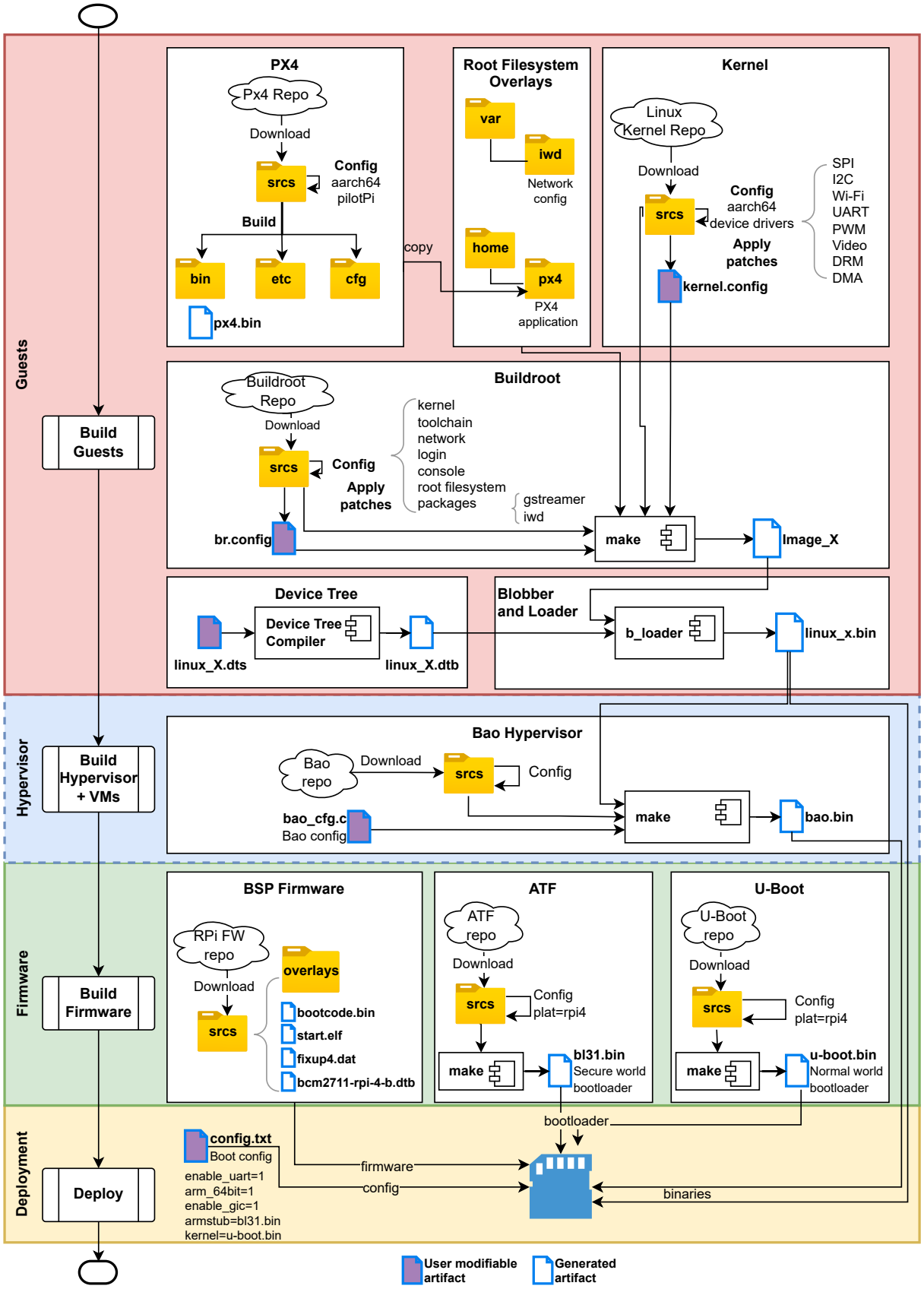


Figure 25: Implementation workflow

combines the kernel and DTB into a self-contained executable with minimal runtime dependencies. For USPFS, this yields a single target binary. For SSPFS, the process is repeated per guest, producing two binaries later merged by Bao.

The `bao_cfg.c` file specifies guest paths, entry points, CPU allocation, memory regions, and device memory/interrupt mappings. Building Bao produces a consolidated image `bao.bin` that encapsulates the hypervisor plus both guests. We next build the platform firmware. After fetching the Raspberry Pi 4 Board Support Package (BSP), we configure and build Arm Trusted Firmware (`bl31.bin`), required by Bao, followed by the normal-world bootloader (`u-boot.bin`). U-Boot is responsible for loading the target: `linux_X.bin` for USPFS or `bao.bin` for SSPFS. Deployment places the boot artifacts onto the SD card: Raspberry Pi firmware, secondary bootloaders, the target binary (`linux_X.bin` or `bao.bin`), and `config.txt`.

To ensure a correct boot, we follow the UAVIC boot flow in Fig. 26. The first-stage bootloader (`bootcode.bin`) initializes hardware, loads firmware from the SD card into RAM, and parses `config.txt`. The GPU firmware `start4.elf` processes `config.txt`, enables UART and the GIC, and transfers control to `bl31.bin` (secure services). Then `u-boot.bin` initializes peripherals (including the console) using the firmware's Device Tree Blob (DTB) and its environment, and loads the target binary.

We focus on the more generic SSPFS case. When `bao.bin` executes, it: (1) initializes CPUs, memory, and the system console; (2) configures the interrupt controller; (3) initializes the mailbox supervisor; and (4) starts the VM manager. Each guest then: (1) boots its Linux kernel using the embedded DTB; (2) initializes guest-specific hardware; (3) mounts the root filesystem; and (4) launches `init`. The result is isolated guest execution under Bao's supervision.

4.2 Base system

The base system is the common hardware/software foundation for both USPFS and SSPFS. It covers UAV assembly and configuration, plus stand-alone testing of PX4 and the video-surveillance stack.

4.2.1 UAV assembly

The first step is assembling the airframe and configuring it via the GCS. Fig. 27 shows the HoverGames-based build alongside the QGroundControl interface. We began with the S500 frame *(3)* as the structural base, then mounted the landing gear, arms, and electronics plates. The NEO-M8N GPS module *(1)* was chosen for cost/performance in urban settings [210]. Four brushless DC motors *(2)* were installed at the arm tips and driven by opto-coupled 40 A ESCs.

Power is provided by a 3S LiPo battery *(12)* (5000 mAh) with an XT-60 connector *(4)* [211], yielding roughly 30 minutes at full throttle (indicative). The UAVIC platform *(7)* combines a Raspberry Pi 4 with the PilotPi shield. Telemetry radios *(5) (6)* link the UAV to QGroundControl on the GCS *(11)*. For early bring-up we enabled a debug UART port *(8)* (omitted in the final system). The Wi-Fi dongle *(9)* and

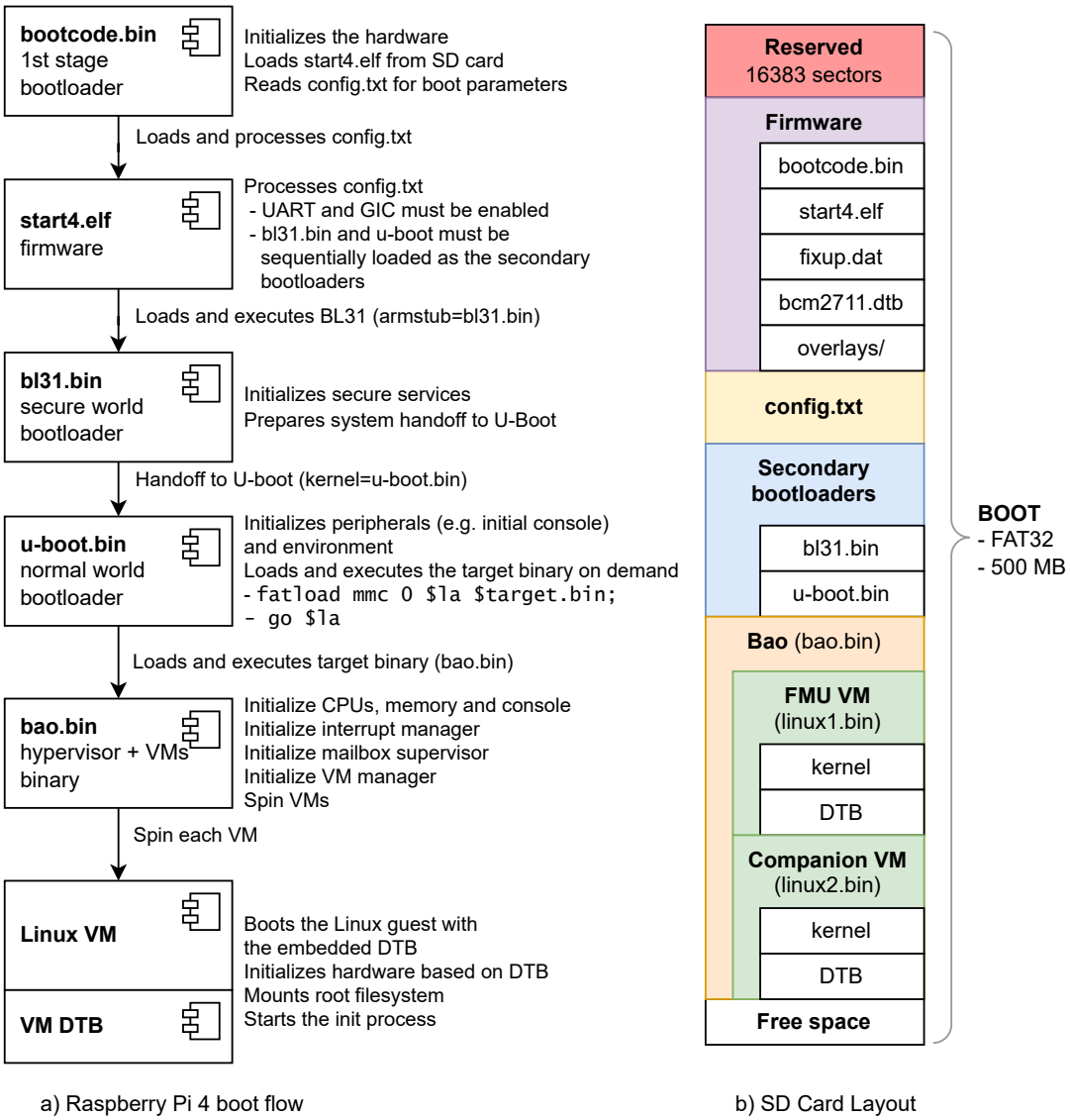


Figure 26: UAVIC boot: (a) platform boot flow; (b) SD card layout

camera *(10)* required for video surveillance connect via USB to the UAVIC. Before parameter configuration in QGroundControl, PX4 must be deployed to the UAVIC. Once PX4 is running, the airframe can be configured over telemetry radio or Wi-Fi.

4.2.2 PX4

After validating the assembly, we deployed PX4 on a general-purpose Linux OS on the UAVIC platform. The build procedure is scripted in `src/buildPilot.sh` (see [26]) and uses a Python virtual environment for dependencies. We fetched the PX4 source (including NuttX applications via submodules), configured the PilotPi target, compiled the autopilot, and finally deployed the resulting artifacts to the UAVIC over Wi-Fi. The NuttX RTOS is configured via the `kconfig` system, analogous to Linux. Listing 4.1 shows an excerpt of the configuration (`configs/px4/px4.config` in the repo [26]) that selects the platform/architecture,

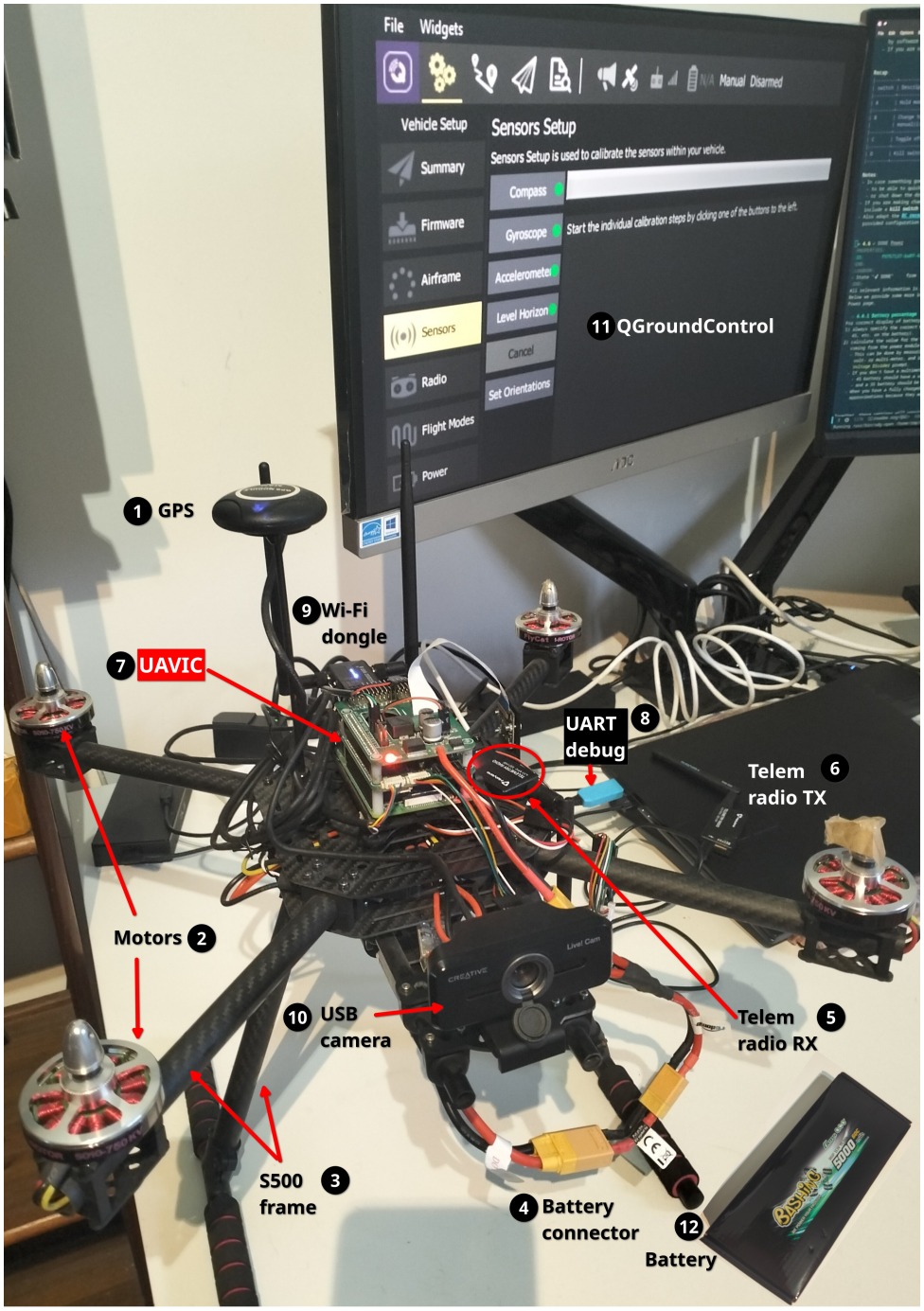


Figure 27: UAV assembly and configuration: UAV (left); Ground Station running QGroundControl (right)

toolchain, and the modules/drivers required for this work.

```
CONFIG_BOARD_TOOLCHAIN="aarch64-linux-gnu"
CONFIG_PLATFORM_POSIX=y
CONFIG_BOARD_LINUX_TARGET=y
CONFIG_BOARD_TOOLCHAIN="arm-linux-gnueabihf"
CONFIG_BOARD_ARCHITECTURE="cortex-a53"
CONFIG_BOARD_TESTING=y
CONFIG_DRIVERS_GPS=y
CONFIG_DRIVERS_IMU_INVENSENSE_ICM42605=y
CONFIG_DRIVERS_MAGNETOMETER_HMC5883=y
```

```
CONFIG_DRIVERS_PCA9685_PWM_OUT=y
CONFIG_COMMON_RC=y
CONFIG_MODULES_BATTERY_STATUS=y
CONFIG_MODULES_CAMERA_FEEDBACK=y
CONFIG_MODULES_COMMANDER=y
CONFIG_MODULES_CONTROL_ALLOCATOR=y
CONFIG_MODULES_DATAMAN=y
```

Listing 4.1: PX4 configuration file (excerpt)

At startup, PX4 runs `pilotpi_mc.config` (`configs/px4/pilotpi_mc.config` in the repo [26]) to configure the vehicle and enable services:

1. Import saved UAV parameters
2. Select vehicle type and airframe
3. Configure camera triggering via MAVLink
4. Initialize geofence, CPU monitoring, and battery status
5. Initialize sensors and actuators
6. Launch the RC communication manager
7. Start the main FMU state machine
8. Initialize multicopter tasks: navigation, position/attitude control, logging, etc.
9. Enable MAVLink telemetry over radio and Wi-Fi
10. Notify the GCS that boot is complete (further configuration by the user)

Fig. 28 shows a successful PX4 initialization: sensors and actuators are online, and the Wi-Fi link is active (`partner IP: 192.168.1.37`) for MAVLink communication with the GCS.

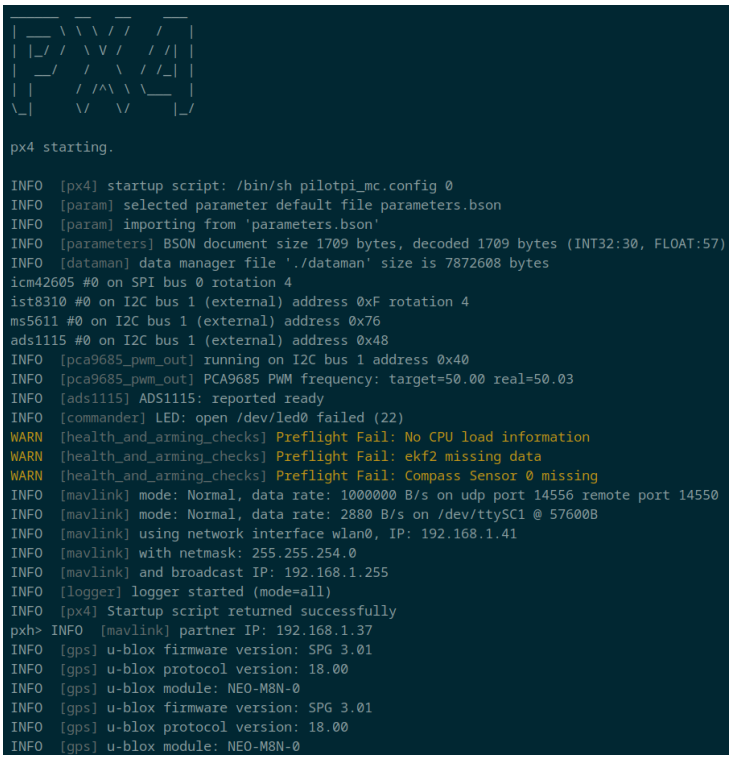


Figure 28: UAV configuration: PX4 boot

4.2.3 UAV configuration

After pairing the UAV with the GCS, we configured the vehicle in *QGroundControl*. First, we selected the airframe: the quadrotor NXP HoverGames variant (Fig. 29). Next, we calibrated the compass, gyroscope, and accelerometer by following the GCS prompts and moving the airframe through the prescribed orientations (Fig. 30). We then configured the power source (battery chemistry/cell count) to obtain correct power/voltage readings and calibrated ESC PWM minimum and maximum values to avoid output saturation. Next, we configured the motors. We specified motor count and geometry (position/direction), mapped motors to output channels, and verified spin direction and speed using the motor test utility. Finally, we reviewed and saved the parameter set; the resulting configuration can be replicated across PX4 instances to bootstrap identical UAVs. Fig. 31 summarizes the final setup.

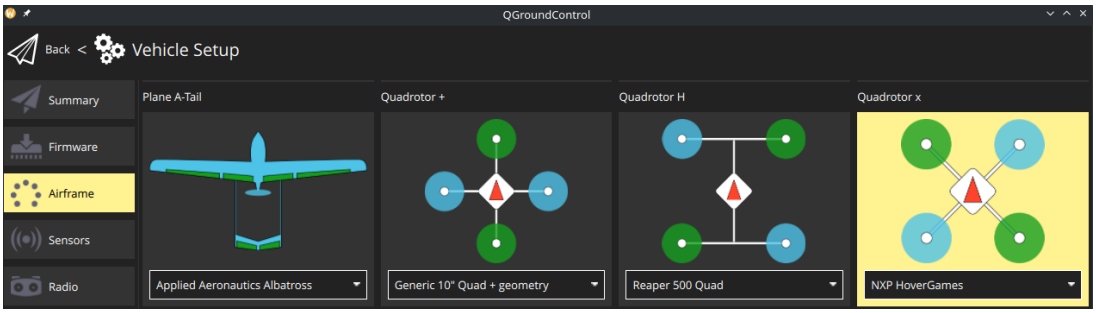


Figure 29: UAV configuration: airframe

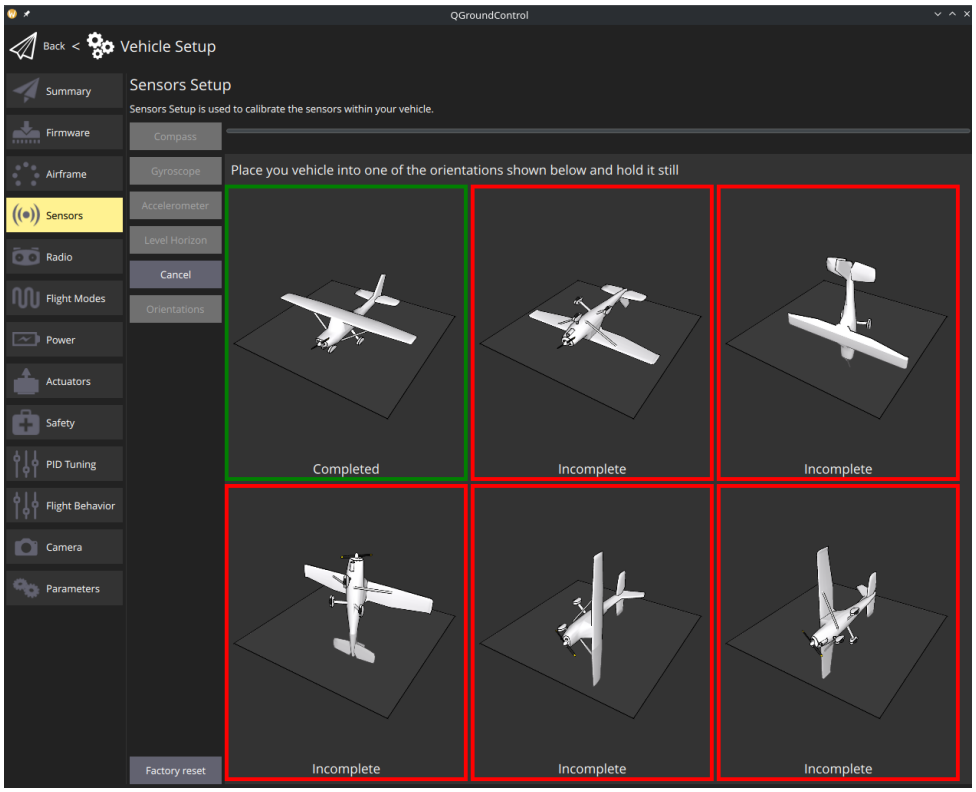


Figure 30: UAV configuration: sensor calibration

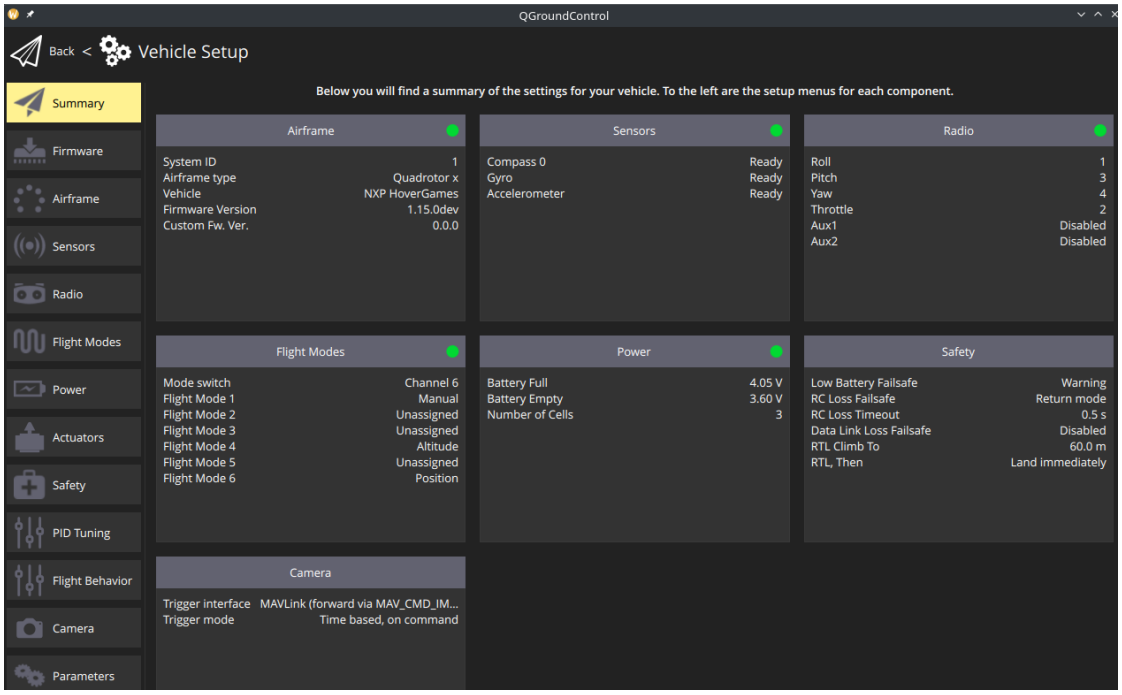


Figure 31: UAV configuration: summary

4.2.4 Video surveillance

The video-surveillance pipeline (see Fig. 15) was implemented with `GStreamer`. Its modular design, broad codec support, and built-in networking made it easy to iterate and test. In the sender pipeline, the UAV captures frames from a USB camera (MJPEG at 640×480, 30 FPS), decodes them, re-encodes as H.264, and packets the stream as Real-time Transport Protocol (RTP) over UDP for low-latency delivery. Listing 4.2 shows the test script, including device selection and the target host/port so only the designated GCS receives the stream.

```
1 #!/bin/sh
2 gst-launch-1.0 -v v4l2src device=/dev/video2 ! image/jpeg,width=640,height=480,framerate=30/1 !
  ↪ jpegdec ! videoconvert ! x264enc bitrate=10000 tune=zerolatency ! rtph264pay
  ↪ config-interval=10 pt=96 ! udpsink host=192.168.1.37 port=5000
```

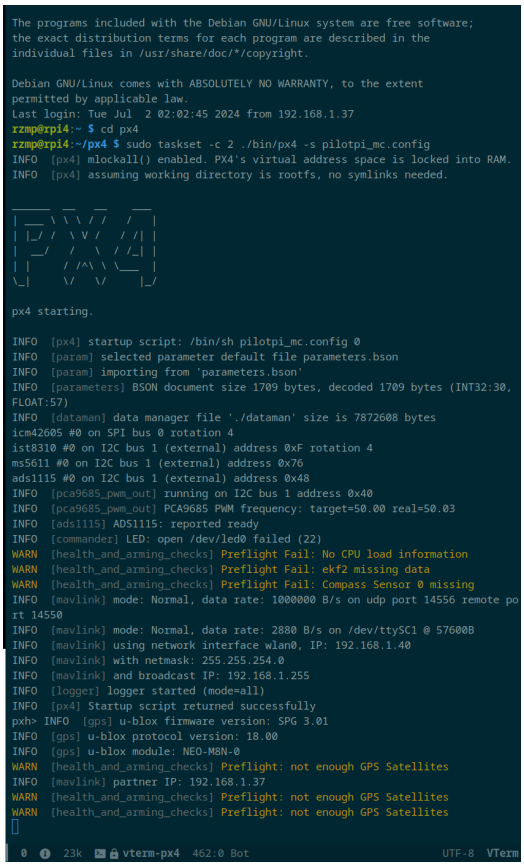
Listing 4.2: Video surveillance sender script

In the receiver end, the ground station listens on the chosen UDP port, unpacks the payload and decodes H.264, converts to a display format, and renders the video. Listing 4.3 shows the matching receiver.

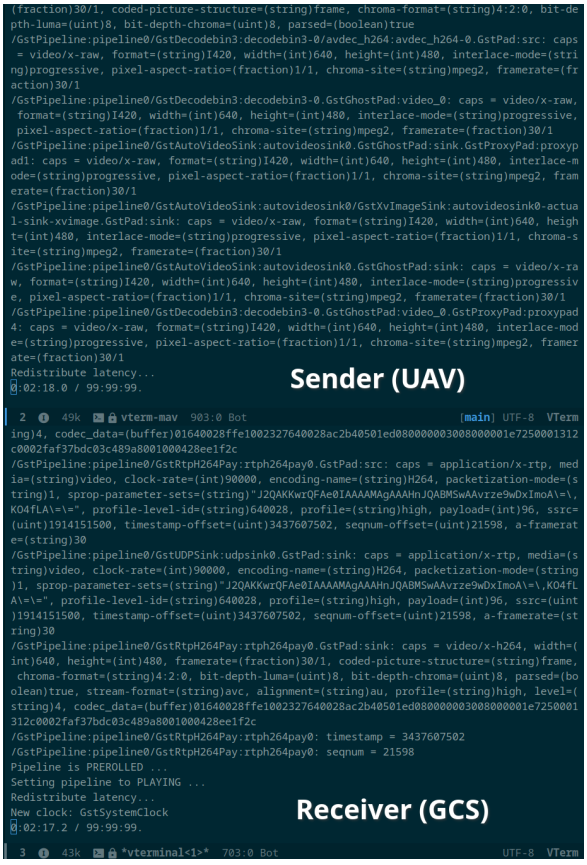
```
1 #!/bin/sh
2 gst-launch-1.0 -v udpsrc port=5000
  ↪ caps="application/x-rtp,media=video,encoding-name=H264,payload=96" ! queue ! rtph264depay !
  ↪ queue ! avdec_h264 ! queue ! videoconvert ! queue ! autovideosink sync=false
```

Listing 4.3: Video surveillance receiver script

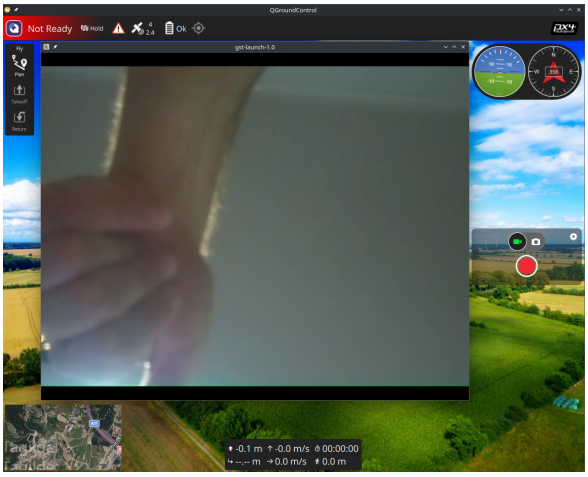
We validated the end-to-end path by running (1) PX4 (Fig. 32a) and the sender on the UAVIC (Fig. 32b, top), and (2) the receiver plus QGroundControl on the GCS (Fig. 32b, bottom). As shown in Fig. 32c, telemetry appears in QGroundControl alongside the live video, validating the pipeline.



(a) PX4 boot



(b) Video pipeline: sender (UAV); receiver (GCS)



(c) QGroundControl + video receiver

Figure 32: Video-surveillance pipeline validation

4.3 USPFS

The USPFS solution consolidates PX4 and the video-surveillance pipeline on a custom embedded Linux OS running on the UAVIC. We first configured a stable Raspberry Pi kernel (6.6) with the drivers required by both stacks. The full configuration is in `configs/uspfs/kernel-uspfs.config` [26]. PX4 needs SPI/I2C (sensors/actuators), PWM (motors), and PL011 UART (RC/console). The video pipeline needs V4L2/video and the Mediatek `mt7921au` module for the AX3000 USB Wi-Fi dongle.

We then configured Buildroot (`configs/uspfs/br-uspfs.config`) to: (i) integrate the custom kernel; (ii) overlay the PX4 binaries/configuration into the rootfs; (iii) set the boot console to UART5 (`ttyAMA5`); (iv) include `GStreamer` for streaming; and (v) package the Mediatek firmware for the Wi-Fi dongle. This produced the Linux image `Image_1`. Next, we customized the device tree to include the required hardware (see Fig. 20) and combined it with the kernel to form a self-contained executable blob, `linux_1.bin`.

For the firmware, we fetched the Raspberry Pi 4 BSP and patched the Arm Trusted Firmware (`bl31.bin`) to redirect the debug console from UART0 to UART5 (`ttyAMA5`). The make fragment `configs/atf/atf-patch.mk` [26] shows the inline `serial0`→`serial5` change and the UART offset adjustment. We also customized U-Boot (`configs/uboot/uboot.mk` and `configs/uboot/uboot.env` [26]) to autoload `linux_1.bin` at the chosen address after `bl31.bin` hands off.

Finally, we wrote the boot artifacts to the SD card: `linux_1.bin`, firmware files, secondary bootloaders, and `config.txt`. The latter (`configs/uspfs/config.txt` [26]) enables early GIC interrupts, selects the secondary bootloader and kernel, and sets UART5 as the system console. We validated boot by monitoring `ttyUSB0` at 115200 bps (`screen /dev/ttyUSB0 115200`); the captured log (`logs/boot-uspfs.txt` [26]) confirms kernel/Buildroot versions and UART5 console bring-up. We then repeated the procedures in Sections 4.2.2 and 4.2.4, verifying that PX4 and the streaming pipeline run correctly in the USPFS environment.

4.4 SSPFS

For the SSPFS system we build two isolated Linux guests atop the Bao hypervisor: one for PX4 and another for the video pipeline. Guest construction mirrors the USPFS flow. For each guest we: (1) select the kernel, enable the required drivers/packages, and compile a Linux OS image (`Image_X`); (2) customize the guest-specific device tree; (3) pack the image and DTB into a self-contained executable blob (`linux_1.bin` for PX4 and `linux_2.bin` for video).

We forked Bao [212] to meet SSPFS-specific platform needs on Raspberry Pi 4, namely redirecting the console from UART0 to UART5 and fixing its clock/offset. In the platform description (`src/rpi4_desc.c` [26]) we set the console base to `0xfe201000` (4 KB-aligned). In the platform header (`src/platform.h` [26]) we configured UART5 with a 48 MHz clock and `0xa00` offset, yielding the final UART5 address `0xfe201a00`.

The VM layout is specified in `configs/sspfs/bao_sspfs_cfg.c` [26]. The PX4 guest receives: (1) one Arm A72 core; (2) two RAM regions (144 MB and 3 GB) – the split reserves the first 1 GB for DMA-reachable buffers (CMA for SPI/I2C devices); (3) the essential device set (Fig. 22). The Companion guest receives: (1) three Arm A72 cores; (2) two RAM regions (624 MB and 2 GB), with a large CMA window in the first 1 GB to sustain the video pipeline; (3) the USB path for camera/Wi-Fi (Fig. 23).

Shared architectural resources are handled as follows. The architectural timer is virtualized by Bao and requires no additional work. The firmware mailbox, needed by devices behind PCIe (e.g., USB), is mediated by the mailbox supervisor described in Sec. 3.3.3.2: guests bracket transactions with `Start_TX` /`End_TX` hypercalls, and Bao serializes access and routes completions to the correct VM.

Finally, we build Bao with the above configuration to produce `bao.bin`, which encapsulates the hypervisor plus both guest blobs. Boot artifacts (firmware, `bl31.bin`, `u-boot.bin`, `bao.bin`, and `config.txt`) are then deployed as in Sec. 4.1, yielding two isolated Linux guests executing under Bao with the device maps of Fig. 22 and Fig. 23.

4.4.1 Mailbox Supervisor

The supervision mechanism in Fig. 21 is realized with two pieces: (1) a mailbox manager inside Bao; and (2) a minimal patch to the Raspberry Pi mailbox firmware driver in Linux. Listing 4.4 shows the Bao-side manager. We reserve a dedicated hypercall identifier and two arguments to mark the *start* and *end* of a mailbox transaction. During platform bring-up, `plat_rpi_init` (invoked before guest boot; see `src/init.c` [26]) binds the mailbox's interrupt to `rpi_mailbox_irq_handler`. When a mailbox event arrives, the handler injects the configured interrupt ID into the current virtual CPU and suppresses Bao's default handling path, ensuring completions are delivered to the correct guest. Guests bracket each transaction with a hypercall (Listing 4.5), which routes into `rpi_mailbox_hypercall`. On *start*, Bao grants the calling `vCPU` exclusive access to the mailbox (locks and enables the relevant interrupts); on *end*, it releases the lock and disables the guest's mailbox interrupts, allowing the next waiting guest to proceed.

```
#define RPI_MAILBOX_IRQ_ID 65
#define RPI_HYP_ARG_START  1
#define RPI_HYP_ARG_END    2
spinlock_t rpi_firmware_lock = SPINLOCK_INITVAL;

static void rpi_mailbox_irq_handler(irqid_t irq_id) {
  vcpu_inject_irq(cpu()->vcpu, irq_id);
  interrupts_cpu_enable(irq_id, false);
}

void plat_rpi_init(void) {
  interrupts_reserve(RPI_MAILBOX_IRQ_ID, rpi_mailbox_irq_handler);
}

long rpi_mailbox_hypercall(unsigned long arg0, unsigned long arg1, unsigned long arg2)
{
  switch (arg0) {
  case RPI_HYP_ARG_START:
    spin_lock(&rpi_firmware_lock);
```

```
20      interrupts_cpu_enable(RPI_MAILBOX_IRQ_ID, true);
21      break;
22    case RPI_HYP_ARG_END:
23      interrupts_cpu_enable(RPI_MAILBOX_IRQ_ID, false);
24      spin_unlock(&rpi_firmware_lock);
25      break;
26    default:
27      ERROR("func %s, unknown arg0 = %lu", __func__, arg0);
28    }
29    return 0;
30  }
```

Listing 4.4: SSPFS: Mailbox manager added to Bao (see src/rpi_firmware.c [26])

```
1  enum { HC_INVAL = 0, HC_IPC = 1, HC_RPI_FIRMWARE = 2 };
2
3  long int hypercall(unsigned long id)
4  {
5    long int ret = -HC_E_INVAL_ID;
6
7    unsigned long ipc_id = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(0));
8    unsigned long arg1 = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(1));
9    unsigned long arg2 = vcpu_readreg(cpu()->vcpu, HYPCALL_ARG_REG(2));
10
11   switch (id) {
12   case HC_IPC:
13     ret = ipc_hypercall(ipc_id, arg1, arg2);
14     break;
15   case HC_RPI_FIRMWARE:
16     ret = rpi_mailbox_hypercall(ipc_id, arg1, arg2);
17     break;
18   default:
19     WARNING("Unknown hypercall id %lu", id);
20   }
21
22   return ret;
23 }
```

Listing 4.5: SSPFS: Bao hypercall manager (see src/hypercall.c [26])

On the Linux side (Listing 4.6), the mailbox driver uses the same hypercall identifier and *start*/*end* arguments. After taking its driver-level lock, it issues an `hvc` with the *start* argument *before* touching the mailbox registers, performs the firmware transaction, then issues the *end* hypercall and releases the lock. This minimal change lets Bao serialize mailbox access across guests without altering the driver's core logic or firmware protocol.

```
1  #define RPI_HYP_ARG_START  1
2  #define RPI_HYP_ARG_END 2
3  enum {HC_INVAL = 0, HC_IPC = 1, HC_RPI_FIRMWARE = 2};
4
5  static int
6  rpi_firmware_transaction(struct rpi_firmware *fw, u32 chan, u32 data)
7  {
8      u32 message = MBOX_MSG(chan, data);
9      int ret;
10
11     WARN_ON(data & 0xf);
12     mutex_lock(&transaction_lock);
13
14     /* HYP call to lock mailbox access */
15     register uint64_t x0 asm("x0") =
```

```
16          ARM_SMCCC_CALL_VAL(ARM_SMCCC_FAST_CALL, ARM_SMCCC_SMC_64, ARM_SMCCC_OWNER_VENDOR_HYP,
            ↪  HC_RPI_FIRMWARE);
17      register uint64_t aux = x0;
18      register uint64_t x1 asm("x1") = RPI_HYP_ARG_START;
19      register uint64_t x2 asm("x2") = 0; /* DO NOT CARE */
20      asm volatile("hvc 0" : "=r"(x0) : "r"(x0), "r"(x1), "r"(x2) : "memory", "cc");
21      /* Check for errors */
22      if ((int64_t)x0 < 0) {
23        dev_err(fw->cl.dev, "HYPCALL START failed: ret = 0x%llx\n", x0);
24        return -EINVAL; /* Abort */
25      }
26
27      reinit_completion(&fw->c);
28      ret = mbox_send_message(fw->chan, &message);
29      if (ret >= 0) {
30          if (wait_for_completion_timeout(&fw->c, HZ)) {
31              ret = 0;
32          } else {
33              ret = -ETIMEDOUT;
34              WARN_ONCE(1, "Firmware transaction timeout");
35          }
36      } else {
37          dev_err(fw->cl.dev, "mbox_send_message failed for chan: %u, data: 0x%08x, ret: %d\n",
            ↪  chan, data, ret);
38      }
39
40      /* HYP call to unlock mailbox access */
41      x0 = ARM_SMCCC_CALL_VAL(ARM_SMCCC_FAST_CALL, ARM_SMCCC_SMC_64,
42                  ARM_SMCCC_OWNER_VENDOR_HYP, HC_RPI_FIRMWARE);
43      aux = x0;
44      x1 = RPI_HYP_ARG_END;
45      x2 = 0; /* DO NOT CARE */
46      asm volatile("hvc 0" : "=r"(x0) : "r"(x0), "r"(x1), "r"(x2) : "memory", "cc");
47      /* Check for errors */
48      if ((int64_t)x0 < 0) {
49        dev_err(fw->cl.dev, "HYPCALL END failed: ret = 0x%llx\n", x0);
50        return -EINVAL; /* Abort */
51      }
52      mutex_unlock(&transaction_lock);
53
54      return ret;
55  }
```

Listing 4.6: SSPFS: Linux's Raspberry Pi mailbox driver — patch (see `src/linux-rpi-fw.c` [26])

5 Evaluation

"*Without data, you're just another person with an opinion.*"
– W. Edwards Deming, statistician

This chapter evaluates the unsupervised (USPFS) and supervised (SSPFS) systems end-to-end. We first outline the evaluation design and indicate where each research question (RQ) is answered. We then (i) validate the mailbox supervisor and overall SSPFS operation, (ii) assess resilience under adversarial conditions, (iii) quantify consolidation overheads with and without the mailbox patch using MiBench AICS (including interference mitigation via cache coloring), (iv) benchmark each guest with application-level metrics (PX4 scheduling overhead, camera frame rate) to compare USPFS versus SSPFS, and (v) run real-flight experiments to measure position-tracking accuracy and in-flight system resource usage (CPU, RAM) and analyze system behavior with and without compromise.

5.1 Evaluation Design

Table 8 maps each research question to its methods, metrics, and the section where it is addressed. We evaluate *(RQ1)* timing preservation under consolidation via *offline* measurements (Linux `perf` events) and MiBench AICS microbenchmarks; *(RQ2)* fault containment via controlled fault injection in the mission VM (panic attack, memory exhaustion and corruption, kernel module faults, user-space resource starvation) and observation of control authority (aircraft remains airborne); *(RQ3)* consolidation overhead versus USPFS using offline MiBench AICS plus application-level metrics (PX4 scheduling overhead, camera frame rate); *(RQ4)* in-flight behavior of SSPFS versus USPFS, focusing on position-tracking accuracy (trajectory versus setpoints) and system resources (CPU load, RAM footprint); and *(RQ5)* supervised mailbox sharing correctness and performance via functional tests (no cross-domain leakage) and throughput/latency measurements for firmware-mediated transactions.

We use the UAVIC platform (Raspberry Pi 4 + PilotPi), two guests (PX4 flight VM and Companion VM), and a repeatable QGroundControl mission profile. Metrics comprise: (i) offline MiBench AICS microbenchmarks and Linux `perf`-based timing; (ii) application-level telemetry (PX4 scheduling overhead,

Table 8: Research question section mapping and associated metrics/artifacts

RQ	Focus	Primary metrics / artifacts	Section
RQ1	Timing under consolidation (offline)	MiBench AICS microbenchmarks; Linux `perf` timing events	5.4
RQ2	Fault containment / control authority	Mission-VM faults (CPU/mem abuse, kmod faults, user-space starvation); flight outcome (aircraft remains airborne)	5.3
RQ3	Overheads vs. USPFS (offline & app-level)	MiBench slowdowns; PX4 scheduling overhead; camera FPS	5.4, 5.5
RQ4	In-flight behavior (accuracy & resources)	Position tracking (trajectory vs. setpoints); CPU load; RAM footprint (onboard logs)	5.5
RQ5	Mailbox sharing correctness/performance	Pass/fail isolation tests; firmware transaction latency/throughput	5.2

camera FPS); (iii) in-flight behavior via PX4 logs (position-tracking accuracy: trajectory versus setpoints; system counters: CPU/RAM); and (iv) isolation outcomes under fault injection, plus mailbox-supervision correctness/latency for firmware-mediated services. Statistical comparisons use paired runs with identical configurations; we report means and worst cases and flag non-significant changes where applicable.

5.2 Functional validation

This section validates the mailbox supervisor and the SSPFS system end to end. For the mailbox path, we added lightweight log prints around each transaction so both guests emit traces during boot. We captured logs via the serial debug console (PX4 VM) and an Secure Shell (SSH) session over Wi-Fi (Companion VM).

Listings 5.1 and 5.2 show boot excerpts for the PX4 and Companion VMs, respectively. In both cases the *start*/*end* hypercalls are issued and trapped correctly, and the transaction completes, confirming that both guests can access the firmware mailbox and, in turn, initialize required PCIe devices.

```
                                                zmp : screen
[    2.470095] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 =
↪  0x0, ret = 0x0
[    2.599543] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 =
↪  0x0, ret = 0x0
[    2.522995] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
[    2.538636] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully:
↪  0xd0000008
[    2.558100] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 =
↪  0x0, ret = 0x0
[    2.586126] fe201a00.serial: ttyAMA5 at MMIO 0xfe201a00 (irq = 24, base_baud = 0) is a PL011
↪  rev2
[    2.602144] printk: console [ttyAMA5] enabled
[    2.602144] printk: console [ttyAMA5] enabled
[    2.622088] printk: bootconsole [p111] disabled
[    2.622088] printk: bootconsole [p111] disabled
[    2.642680] uart-p1011 fe201000.serial: there is not valid maps for state default
[    2.658360] uart-p1011 fe201000.serial: cts_event_workaround enabled
[    2.674445] fe201000.serial: ttyAMA0 at MMIO 0xfe201000 (irq = 24, base_baud = 0) is a PL011
↪  rev2
```

```
15 [    2.693306] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2 =
   ↪  0x0, ret = 0x0
16 [    2.718124] raspberrypi-firmware soc:firmware: Firmware message: 0xd0000008
17 [    2.733485] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully:
   ↪  0xd0000008
18 [    2.759552] spi1.0: ttySC0) at I/O 0x0 (irq = 27, base_baud = 921600) is a SC16IS752
19 [    2.762197] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002,
20 x1 = 0x2, x2 = 0x0, ret = 0x0
21 [    2.778563] spi1.0: ttySC1 at I/O 0x1 (irq = 27, base_baud = 921600) is a SC16IS752
```

Listing 5.1: SSPFS: mailbox supervisor validation – PX4 VM boot log excerpt (see `logs/rpi-fw-validation-1.txt` [26])

```
1                                                  (root) 192.168.1.59
2 [    16.618543] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2
   ↪  = 0x0, ret = 0x0
3 [    16.618580] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
4 [    16.622791] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully:
   ↪  0xe3480008
5 [    16.622808] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 =
   ↪  0x0, ret = 0x0
6 [    16.622827] raspberrypi-firmware soc:firmware: HYPCALL START: x0 = 0xc6000002, x1 = 0x1, x2
   ↪  = 0x0, ret = 0x0
7 [    16.622838] raspberrypi-firmware soc:firmware: Firmware message: 0xe3480008
8 [    16.622873] raspberrypi-firmware soc:firmware: Firmware transaction completed successfully:
   ↪  0xe3480008
9 [    16.622886] raspberrypi-firmware soc:firmware: HYPCALL END: x0 = 0xc6000002, x1 = 0x2, x2 =
   ↪  0x0, ret = 0x0
```

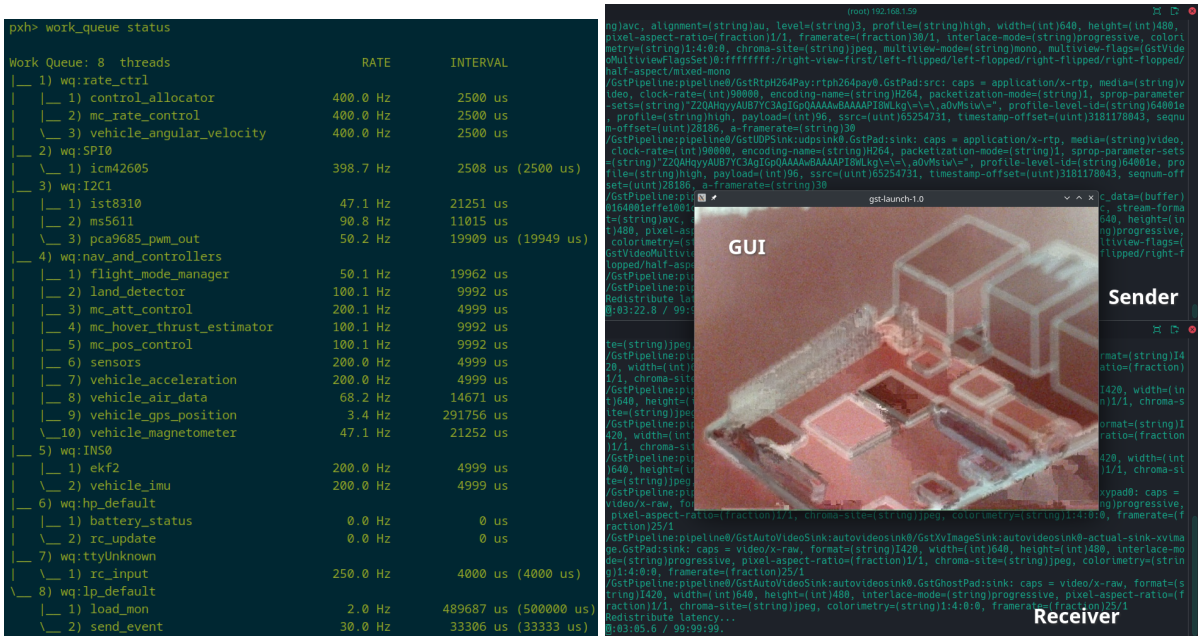
Listing 5.2: SSPFS: mailbox supervisor validation – Companion VM boot log excerpt (see `logs/rpi-fw-validation-2.txt` [26])

To validate SSPFS as a whole, we loaded `bao.bin` from the U-Boot prompt on the UAVIC and inspected both boot logs (`logs/sspfs-boot-px4.txt` and `logs/sspfs-boot-companion.txt` [26]). The PX4 VM log shows: (1) Bao initialization on the shared console; (2) expected memory map; (3) UART5 console setup; (4) successful firmware mailbox access; (5) SPI device bring-up; (6) Buildroot startup. The Companion VM log shows: (1) expected memory map; (2) all three secondary cores online; (3) firmware mailbox access; (4) USB enumeration for camera and Wi-Fi; (5) automatic Wi-Fi association; (6) network configuration. We conclude that PX4 and the video stack execute in isolation as required, while sharing the firmware mailbox under supervision.

We then validated application behavior within each guest. For PX4, we launched the stack and connected QGroundControl over the telemetry radio. Running `work_queue status` (Fig. 33a) shows the expected set of active threads (navigation, communications, sensor pipelines), confirming telemetry and PX4 integrity. For video, we started the receiver on the GCS and the sender on the Companion VM; the live stream appears alongside QGroundControl (Fig. 33b), validating the end-to-end pipeline.

5.3 Functional tests

Fig. 34 outlines the functional test methodology, which assesses resilience under adversarial faults. We emulate compromises by deploying a malicious user-space program (`crash_app`) and a malicious kernel



(a) PX4 *(b)* Video surveillance

Figure 33: SSPFS: validation of application software

module (`crash_mod.ko`) against both USPFS and SSPFS. In the *unsupervised* case, malicious components execute directly on the host Linux OS; a process crash or kernel fault is expected to propagate system-wide, terminating both the non-critical (video streaming) and safety-critical (PX4) stacks, leading to loss of flight control. In the *supervised* case, we deploy malicious components solely to the Companion VM, reflecting a likely compromise vector via Wi-Fi. Here, we expect the hypervisor to contain the failure to the Companion VM: a fatal user-space abort or kernel fault triggers a guest crash while the PX4 VM continues operating, preserving flight capability. This comparative setup demonstrates fault containment with SSPFS: critical functions remain available despite failures in the non-critical domain.

5.3.1 Privileged mode

Malicious actors with privileged access – via superuser escalation or corrupted kernel modules – pose severe risks. This section evaluates vulnerability to privileged-mode attacks by deploying malicious kernel modules. A basic kernel attack vector is an explicit `panic` invocation, which signals an unrecoverable state. Listing 5.3 (line 5) implements this and shows the skeleton of a kernel module. Additional modules are in `eval/kmod/` [26]. When executed, we expect: (1) local interrupts and preemption disabled; (2) secondary cores halted; (3) a `Forced kernel panic!` message; (4) system reboot after a timeout or a hang in an infinite loop.

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3
4 static int __init panic_init(void) {
5     panic("Forced kernel panic!");
```

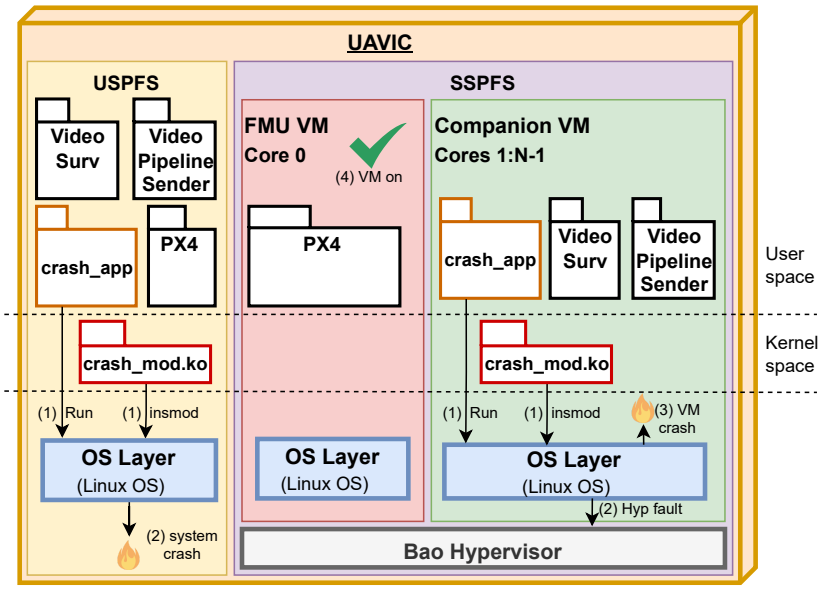


Figure 34: Functional tests' overview

```
6     return 0;
7 }
8
9 static void __exit panic_exit(void){
10   return;
11 }
12
13 module_init(panic_init);
14 module_exit(panic_exit);
```

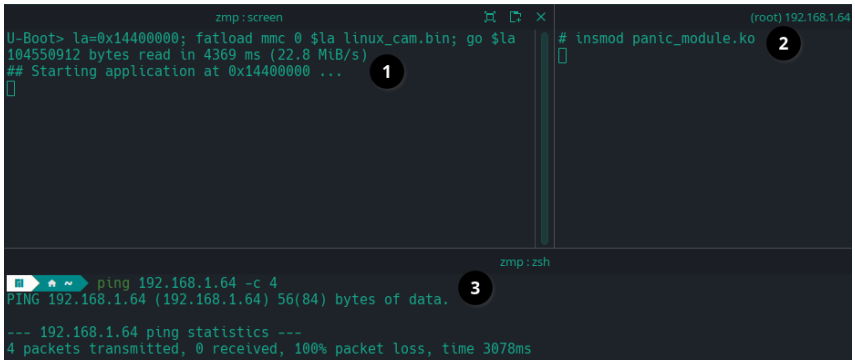
Listing 5.3: Functional tests: implementation of the Panic kernel module (see `panic_module.c` [26])

Fig. 35a shows testing on USPFS. We boot via UART (1), then insert the malicious module over SSH on Wi-Fi (2). The SSH session froze, and the host became unreachable (3), confirming a system-wide crash (catastrophic UAV failure).

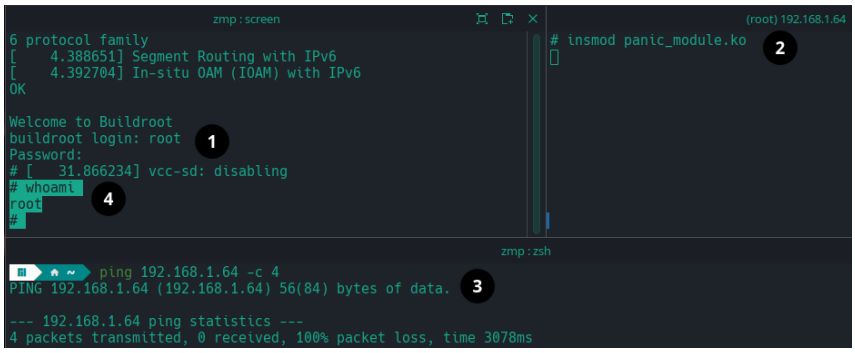
We then repeated the test on SSPFS (Fig. 35b). The system was started from the U-Boot UART console, later used by Bao and the PX4 VM, allowing login (1). After inserting the malicious module via SSH into the Companion VM (2), the SSH session froze and the Companion VM became unreachable (3). The PX4 VM continued running (4), preserving UAV control. These results show that Bao maintains domain isolation under privileged-mode faults.

A memory corruption attack writes to an unmapped physical address using `ioremap` of a valid but non-existent region (source in `memcorrupt_module.c` [26]). The write (line 36) is expected to fault due to missing physical backing [213]. Results matched the panic case: USPFS suffered a system-wide failure, whereas in SSPFS only the Companion VM crashed and the PX4 VM remained operational.

Memory exhaustion is induced by lowering the Out Of Memory (OOM) score of the kernel thread (line 11) to minimize kill likelihood, then allocating and pinning pages in a loop (lines 15, 20) to prevent reclamation (`memhog_module.c` [26]). Again, USPFS collapsed under pressure, while SSPFS contained the failure to the Companion VM and kept PX4 running.



(a) USPFS



(b) SSPFS

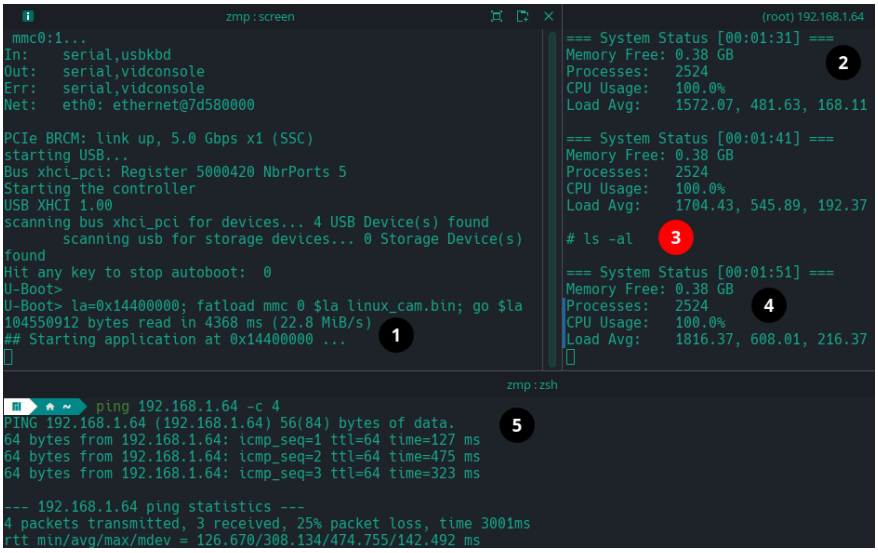
Figure 35: Functional tests: panic kernel module testing

Finally, terminating `init` compromises the system. The module (`initkiller_module.c` [26]) references `init` (line 16), walks its Virtual Memory Areas (VMAs) (line 28), pins pages (line 36), and corrupts the memory map (line 42), provoking failure. As before, USPFS crashed globally, whereas SSPFS preserved the PX4 VM and flight capability.

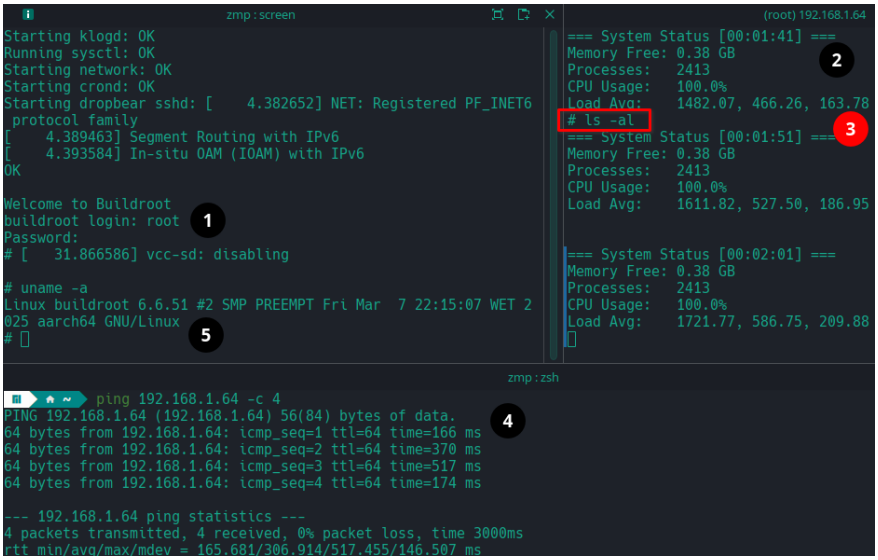
5.3.2 Unprivileged mode

User-space attacks expose a smaller surface because access to critical resources is limited. A common tactic is to induce extreme contention that degrades performance until the critical system becomes unresponsive, which can still lead to a hazardous outcome in flight. The user-space resource-exhaustion application is available at `eval/userspace/memhog_module.c` [26]. It employs three concurrent vectors: (1) memory exhaustion by unbounded allocation while attempting to evade the OOM killer (`memory_attack()`); (2) process exhaustion via a fork storm that spawns processes which touch shared memory and then hang (`process_attack()`); (3) CPU and I/O starvation (`cpu_io_attack()`) by raising the process to the highest scheduling priority (line 132), issuing continuous random page reads (line 140), and forcing frequent rescheduling (line 143). A monitoring thread (`status_thread()`) records (1) elapsed time, (2) free memory via `sysinfo()`, (3) process count `sysinfo.procs`, (4) CPU usage from `/proc/stat`, and (5) load average from `sysinfo.loads`. The `main()` routine removes resource limits, launches the monitor, starts each attack in a child process, and then stays alive in a non-optimizable loop.

Fig. 36a shows testing on USPFS. After boot over UART (1) and SSH login over Wi-Fi (2), the system quickly became unresponsive (3). Basic commands such as `ls` failed and resources were permanently pinned by the attack (4). Although the host still answered at the network level, it could no longer service PX4 or the video pipeline, severely impacting flight functionality. We repeated the procedure on SSPFS (Fig. 36b). The platform was booted over UART, bringing up the PX4 and Companion VMs. After logging into the PX4 VM (1), we connected to the Companion VM via SSH (2) and started the attack there. The Companion VM became unresponsive (3) and dedicated its resources to the attacker (4), but the PX4 VM remained responsive and controllable. Hence only the non-critical domain failed, and the UAV retained flight capability. These results confirm that Bao preserves critical-domain availability under severe user-space contention in the non-critical domain, meeting the consolidation and isolation goals.



(a) USPFS



(b) SSPFS

Figure 36: Functional tests: user-space resource-exhaustion application

Table 9 summarizes functional validation results by privilege level and failure mode for each system. *P*, *C*, *M*, *L*, and *B* denote PX4, Companion, Malicious, Linux, and Bao, respectively. Cell colors indicate the execution status: **running**, **heavy impact**, and **failure**. Results show that, under supervision (SSPFS), faults are contained within the non-critical stack, whereas the unsupervised baseline (USPFS) exhibits propagation to critical components.

Table 9: Functional validation by privilege level and failure mode for each system

	Privileged												Unprivileged		
System	**Panic**			**Mem corruption**			**Mem exhaust**			**Kill `init`**			**Mem+Proc+CPU/IO**		
USPFS	P	C	M	P	C	M	P	C	M	P	C	M	P	C	M
		L			L			L			L			L	
SSPFS	P	C	M	P	C	M	P	C	M	P	C	M	P	C	M
	L		L	L		L	L		L	L		L	L		L
		B			B			B			B			B	

Running | Heavy impact | Fail | P: PX4 | C: Companion | M: Malicious | L: Linux | B: Bao

5.4 Bao benchmarks

Prior work has benchmarked the Bao hypervisor in mixed-criticality settings [80, 214]. We follow a similar methodology but focus strictly on performance. First, we establish a baseline on the USPFS system (native execution). Then, we quantify the overhead introduced by Bao on the SSPFS system. Performance degradation (PD) is computed as the relative increase in execution time (ET):

$$PD\ (\%) = \frac{ET_{\text{SSPFS}} - ET_{\text{USPFS}}}{ET_{\text{USPFS}}} \cdot 100 \tag{5.1}$$

The test suite is MiBench AICS [215], with both *small* (resource-constrained) and *large* (realistic load) variants. On Linux guests we use `perf` [216] to time executions and collect microarchitectural counters; bare-metal guests use the Arm Generic Timer (10 ns resolution) with a custom Performance Monitor Unit (PMU) driver. The monitored events include: (1) instruction count; (2) TLB accesses/refills; (3) cache accesses/refills; (4) exceptions; and (5) interrupts. To study inter-VM interference, we deploy a custom bare-metal guest with three Virtual CPUs (vCPUs) that continuously stream writes into a buffer sized to the LLC, creating a sustained memory subsystem load (not necessarily worst-case, but intentionally heavy).

Fig. 37 summarizes the workflow: build guests (Linux+`perf` for MiBench and the interference guest), run experiments, capture logs, and plot results [217]. We evaluate the USPFS baseline and four SSPFS configurations: `mibench` (Linux guest on Bao), `mibench+col` (+ page coloring), `mibench+interf` (+ interference guest), and `mibench+interf+col` (+ interference and coloring). For each run we place the executable `BIN` (`linux.bin` or `bao.bin`) on the SD card (1), open a persistent capture from GCS to UAVIC (2), power on (3), start the binary from U-Boot (4), and append results to a log file (8). Some test runs require a second binary `BIN2` (e.g., iterating the full MiBench set) whose output is redirected into the

same log (7). We then close the log and plot relevand data with a Python script. All MiBench configuration files, logs, and plotting scripts are in `mibench/` [26].

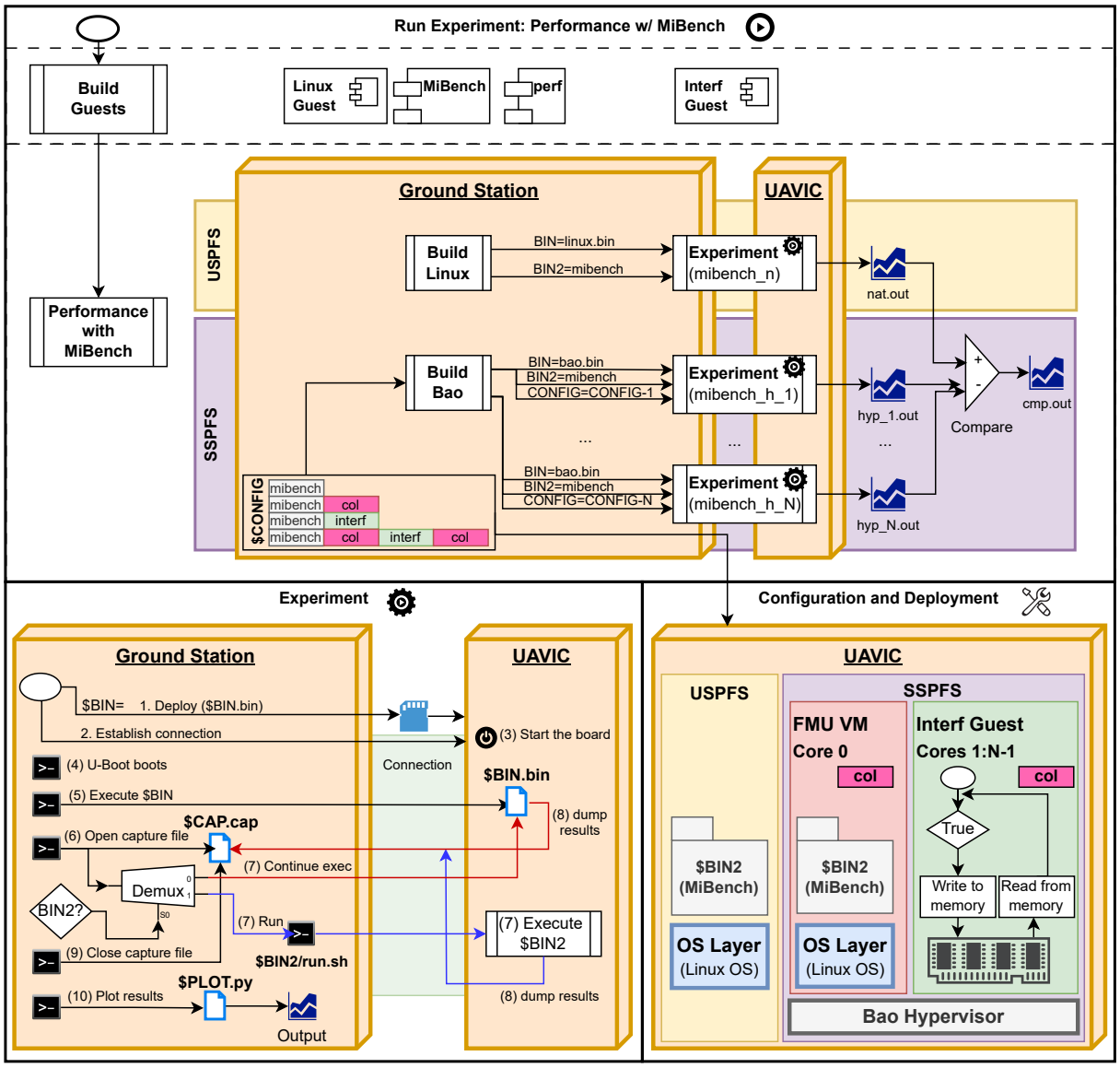


Figure 37: Bao performance benchmarking workflow

Fig. 38 reports relative PD for MiBench AICS across three configurations: (1) **baremetal-noMail** — native execution without the mailbox driver patch; (2) **bao-noMail** — Linux guest on Bao without the mailbox driver patch; and (3) **bao** — Linux guest on Bao with the mailbox driver patch. Average baseline execution times are printed below each benchmark. PD is small overall, especially for longer-running benchmarks (suffix `large`, e.g., `basicmath large`, `qsort large`). The mailbox patch adds negligible overhead and Bao's contribution to PD is also modest.

Fig. 39 shows PD under cache partitioning and interference for: (1) **bao+col** — MiBench on VM1 with two cache colors; (2) **bao+interf** — MiBench on VM1 with the interference guest on VM2 using three CPUs; and (3) **bao+interf+col** — MiBench on VM1 with two colors, plus the interference guest

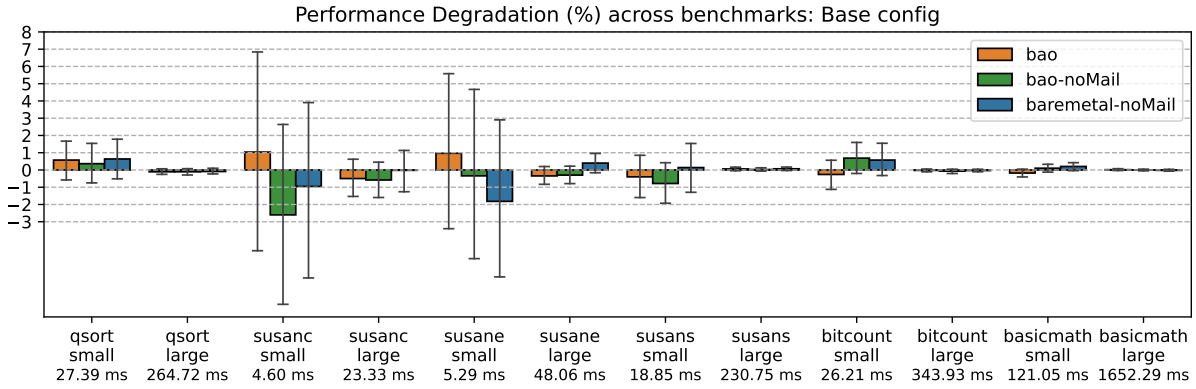


Figure 38: Relative performance degradation (%) for MiBench AICS

on VM2 using three CPUs and two colors. Colors are evenly split between VMs; average baseline times appear below each benchmark. Interference significantly raises PD, especially for short runs (suffix `small` , e.g., `susanc small`, `susane small`), while longer runs are less sensitive. Page coloring consistently mitigates interference for VM1, reducing PD across the set.

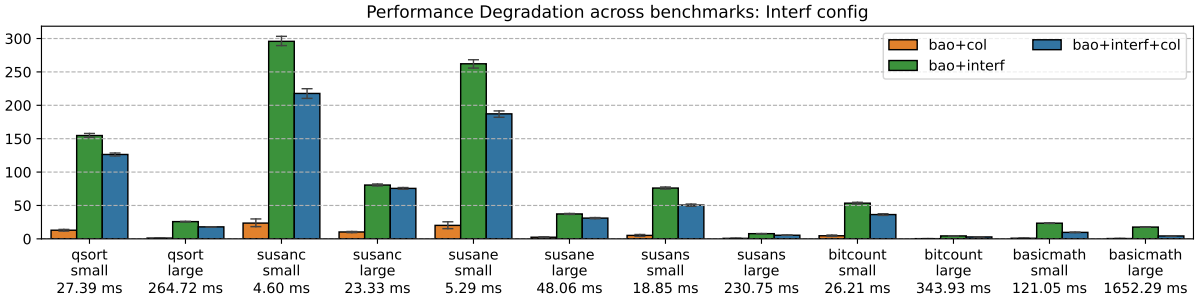


Figure 39: Relative performance degradation (%) for MiBench AICS under cache partitioning and interference

5.4.1 Guests benchmarking

We benchmark each VM separately with application-specific metrics, then compare USPFS and SSPFS using those metrics.

5.4.1.1 FMU VM

The FMU VM runs PX4 atop Linux under the Bao hypervisor. We adapted the MiBench-based methodology to profile PX4 with `perf`. Unlike MiBench (finite runs), PX4 is long-running, so we use `perf --timeout` to fork PX4, collect counters for a fixed interval, and terminate the child.

Listing 5.4 shows the procedure: five warm-up runs and twenty 30 seconds test runs. The `px4_bench()` helper invokes `perf` with the desired events/timeout and the PX4 launch command (lines 6–17). We then set the events (line 18) and call `px4_bench` for warm-up and measured runs.

```
#!/bin/sh
WARM_UP=5
REPS=20
TIMEOUT=30000 # ms

# Seed random generator
cat /dev/random | head > /dev/null

px4_bench(){
    events="$1"
    reps="$2"
    timeout="$3"
    printf "\n--- EVENTS: ${events}: RUNS=$reps; timeout=$timeout ---\n"
    perf stat --table -n -r $reps "$events" --timeout=$timeout ./bin/px4 -s pilotpi_mc.config
}

# Run benchmarks for each event set
EVENTS="-e r08:uk,r08:h,r09:uk,r09:h,r17:uk,r17:h"
px4_bench "$EVENTS" $WARM_UP $TIMEOUT
px4_bench "$EVENTS" $REPS $TIMEOUT

EVENTS="-e r02:uk,r02:h,r05:uk,r05:h"
px4_bench "$EVENTS" $WARM_UP $TIMEOUT
px4_bench "$EVENTS" $REPS $TIMEOUT
```

Listing 5.4: PX4 benchmarking using `perf`

Wall-clock time is unsuitable here (forced termination dominates): native averaged 30.044213 ± 0.000613 s and Bao 30.04608 ± 0.00199 s (max PD $\approx 0.015\%$). Instead, we compute PD from the instruction count (`r08:uk`). Native recorded 3,077,590,067 events versus Bao's 3,016,875,258, a small degradation of 1.97%.

For critical systems, deadline compliance matters most. PX4 supports (i) traditional tasks (dedicated stacks/priorities) and (ii) work-queue tasks (shared stacks/priorities), the latter preferred for efficiency [218]. We therefore profile work-queue update rates for six critical operations [219–221]: (1) `control_allocator` (torque/thrust to actuator setpoints); (2) `mc_rate_control` (multicopter rate control); (3) `pca9685_pwm_out` (I2C PWM to ESC motors); (4) `flight_mode_manager` (setpoint generation per flight mode); (5) `mc_pos_control` (position/velocity control); (6) `sensors` (acquisition/publish).

To query PX4 while running, we inject `work_queue status` via a shared First In, First Out (FIFO). The script `px4_wq_run.sh` (see [26]) creates the FIFO (line 12), launches PX4 and captures its PID (lines 13–15), writes periodic commands (lines 28–32), then shuts down and cleans up. We run five warm-ups followed by twenty measured runs under both native and Bao. Each run issues `work_queue status` every 10 s for 3 min (18 samples/run).

Fig. 40 reports results. The x-axis lists the tasks and the native mean update interval (µs) as the baseline. Bars show Bao's mean scheduling overhead and standard deviation, defined as:

$$\mu_{ov}(\%) = \frac{\mu_{bao} - \mu_{bm}}{\mu_{bm}} \cdot 100 \tag{5.2}$$

$$\sigma_{ov}(\%) = \sqrt{\left(\frac{\sigma_{bao}}{\mu_{bao}}\right)^2 + \left(\frac{\sigma_{bm}}{\mu_{bm}}\right)^2} \cdot 100 \tag{5.3}$$

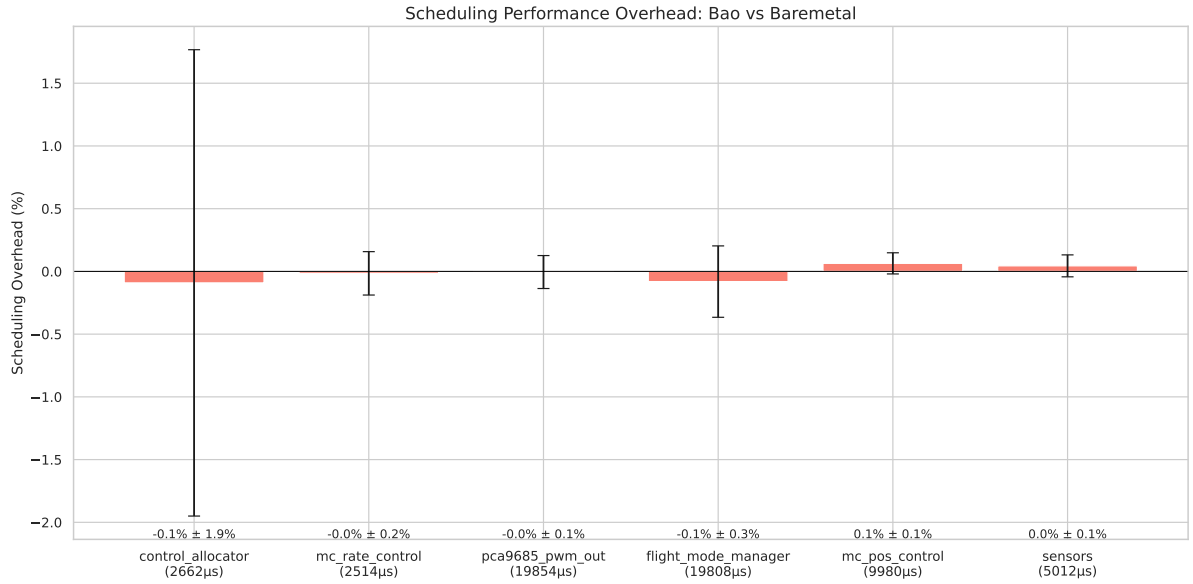


Figure 40: PX4 tasks scheduling overhead

Overheads are very low across tasks. Notably, `pca9685_pwm_out` (motor actuation) shows negligible overhead, while `control_allocator` (control-loop core) remains $\leq 2\%$. Overall, consolidating PX4 with Bao imposes minimal scheduling overhead on the critical FMU.

5.4.1.2 Companion VM

The Companion VM runs the `gstreamer` pipeline for video capture and streaming to the GCS. User experience is driven primarily by two metrics: sender-side FPS and end-to-end frame latency (capture-to-reception delay). These can be distorted by external factors (e.g., GCS CPU load or network jitter). To isolate consolidation overheads from environmental noise, we measure frame rate on the UAV side. The script `cam_fps.sh` (see [26]) runs the pipeline locally and reports FPS. We execute five warm-up runs and twenty 2-minute test runs, then terminate the pipeline. Fig. 41 reports the relative FPS performance degradation (Bao versus native) across runs, computed with Equation 5.2 and Equation 5.3. Most runs are near 0% degradation; the worst case stays below 2%. Confidence intervals are narrow and include 0%, indicating no statistically significant FPS penalty from Bao.

5.4.2 USPFS vs SSPFS

After benchmarking guests individually, we compare USPFS and SSPFS by running PX4 scheduling measurements and camera FPS tests concurrently on each system, logging data per-architecture. Fig. 42 contrasts scheduling overheads and shows the effect of cache coloring. The baseline is USPFS; orange and blue bars denote SSPFS with and without coloring, respectively. Positive values indicate degradation and negative values indicate improvement. SSPFS shows only marginal degradation (1%) for

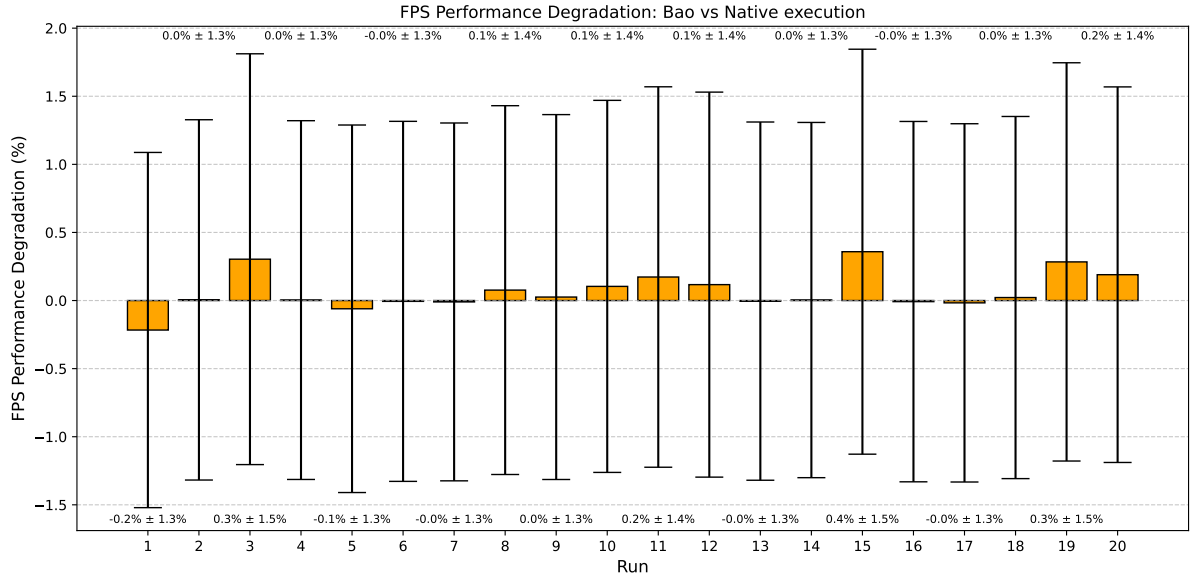


Figure 41: Relative FPS performance degradation: Bao versus native execution

`flight_mode_manager` and `pca9685_pwm_out`, while outperforming USPFS elsewhere – most notably `control_allocator` (6-15% improvement).

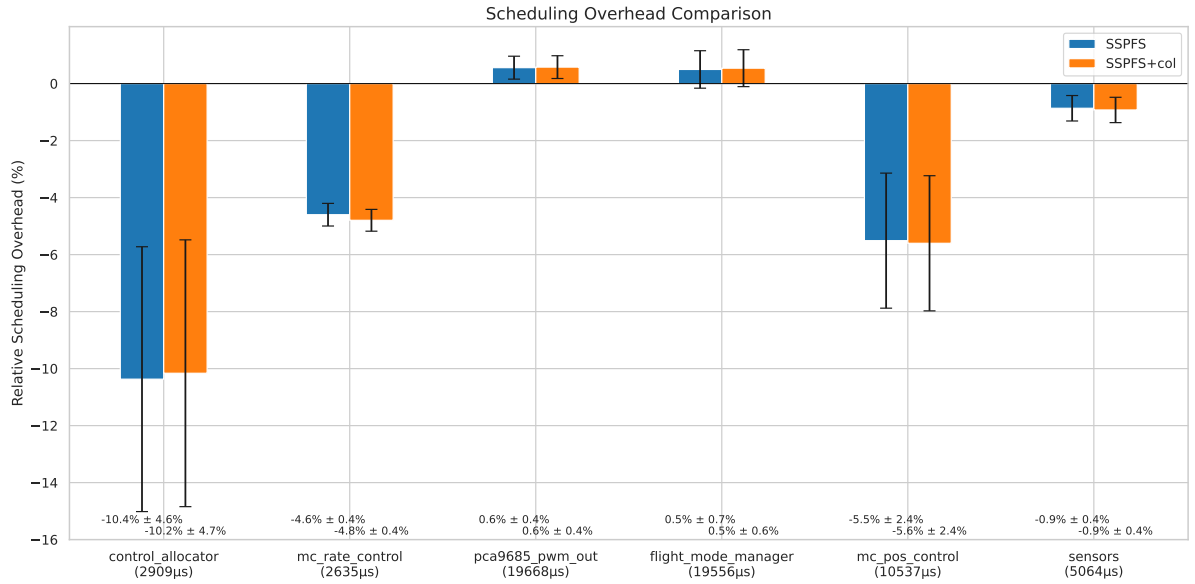


Figure 42: PX4 task scheduling overhead: USPFS versus SSPFS

We attribute this divergence to Bao's isolation: USPFS processes contend for shared resources, whereas SSPFS VMs enjoy statically partitioned resources. Cache coloring yields only marginal gains on the UAVIC due to DMA constraints from SPI sensors; required DMA mappings in the first 1 GiB constrain virtual-coloring effectiveness. Listing 5.5 shows the enforced physical placement in the SSPFS configuration via `place_phys = true`.

```
.vmlist = {{.colors = 0b00000011,
          .platform = {.region_num = 2,
               .regions = (struct vm_mem_region[]){
                 {.base = VM1_MEM1_BASE,
                  .size = VM1_MEM1_SIZE,
                  .place_phys = true,
                  .phys = VM1_MEM1_BASE},
                 {.base = VM1_MEM2_BASE,
                  .size = VM1_MEM2_SIZE,
                  .place_phys = true,
                  .phys = VM1_MEM2_BASE}},
          }}
```

Listing 5.5: Physical memory mapping in SSPFS: `place_phys = true`

Fig. 43 compares relative FPS degradation across systems and evaluates cache coloring. Most runs show no statistically significant difference (whiskers include 0). Runs 10 and 20 exhibit notable outliers with over 30% improvement, while cache coloring again has limited impact given DMA constraints. These results align with the Companion VM microbenchmarks, indicating minimal impact on video performance. Overall, beyond isolation benefits, SSPFS can deliver performance advantages in mixed-criticality consolidation. Static resource partitioning reduces contention seen in USPFS, making SSPFS a strong choice for integrating flight control and video surveillance.

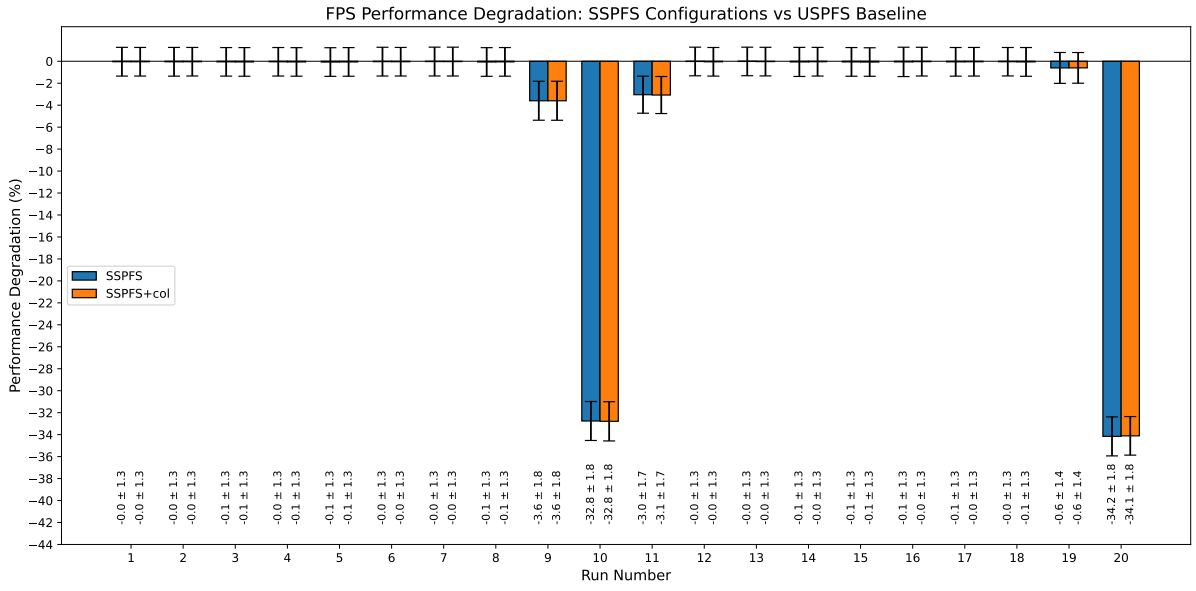


Figure 43: Relative FPS performance degradation: USPFS versus SSPFS

5.5 UAV benchmarks

Finally, we evaluated both flight stacks under realistic conditions. To this end, we configured an automated mission in QGroundControl (Fig. 44), ensuring a repeatable flight profile that can be used to benchmark the USPFS and SSPFS systems in real-flight scenarios. The mission design incorporates two critical geospatial

elements. An external polygon defines the geofence perimeter, which triggers a return-to-home failsafe if violated. The internal path is the flight path from takeoff `T` to landing `L`. At takeoff, the UAV ascends to an altitude of two meters and flies to waypoint `3`, then reverses direction and proceeds to `4`, concluding at `L` while maintaining a constant altitude. During landing, the UAV reduces speed to enable a safe, smooth touchdown.

Figure 44: Automated mission configuration in QGroundControl

We performed a total of 33 flights per system under consistent weather conditions. Each system was tested in a batch, as switching between systems requires significant battery and time resources, preventing randomized sequencing. The flight logs were preserved for subsequent analysis of two critical metrics: (1) position-tracking accuracy and (2) system resource utilization patterns. The sample size was chosen to achieve statistical power sufficient to detect meaningful differences in these operational parameters. PX4 flight logs are stored as ULog files (`.ulg`), containing uORB topic messages subscribed to or published by various modules. Specifically, we analyzed the `vehicle_local` and `vehicle_local_setpoint` topics for position-tracking assessment, and the `cpuload` topic to evaluate system resource utilization. To account for temporal variations between flights, the time domain was normalized to a mission-progress scale $[0, 100]$ using linear interpolation:

$$t_{\text{norm}} = \frac{t - t_{\min}}{t_{\max} - t_{\min}} \cdot 100 \tag{5.4}$$

where t is the original timestamp, and $t_{\min}$ and $t_{\max}$ represent the start and end times of each flight, respectively. At each normalized time point τ, the 95% confidence intervals for group means were calculated using the Student's t-distribution [222]:

$$\text{CI}(\tau) = \bar{x}(\tau) \pm t_{\alpha/2,df} \cdot \frac{s(\tau)}{\sqrt{n(\tau)}} \tag{5.5}$$

where:

- $\bar{x}(\tau)$ is the sample mean at progress τ,
- $s(\tau)$ is the sample standard deviation,
- $n(\tau)$ is the number of valid observations,
- $df = n(\tau) - 1$ degrees of freedom,
- $t_{\alpha/2,df}$ is the critical t-value ($\alpha = 0.05$).

Statistical significance between systems was evaluated using the two-sided Welch unequal-variance t-test at each mission-progress point. This method is preferred over the pooled-variance t-test, as the samples are independent and variances may differ [223]:

$$t(\tau) = \frac{\bar{x}_1(\tau) - \bar{x}_2(\tau)}{\sqrt{\frac{s_1^2(\tau)}{n_1(\tau)} + \frac{s_2^2(\tau)}{n_2(\tau)}}} \tag{5.6}$$

The degrees of freedom were computed via:

$$df(\tau) = \frac{\left(\frac{s_1^2(\tau)}{n_1(\tau)} + \frac{s_2^2(\tau)}{n_2(\tau)}\right)^2}{\frac{s_1^4(\tau)}{n_1^2(\tau)\,(n_1(\tau)-1)} + \frac{s_2^4(\tau)}{n_2^2(\tau)\,(n_2(\tau)-1)}} \tag{5.7}$$

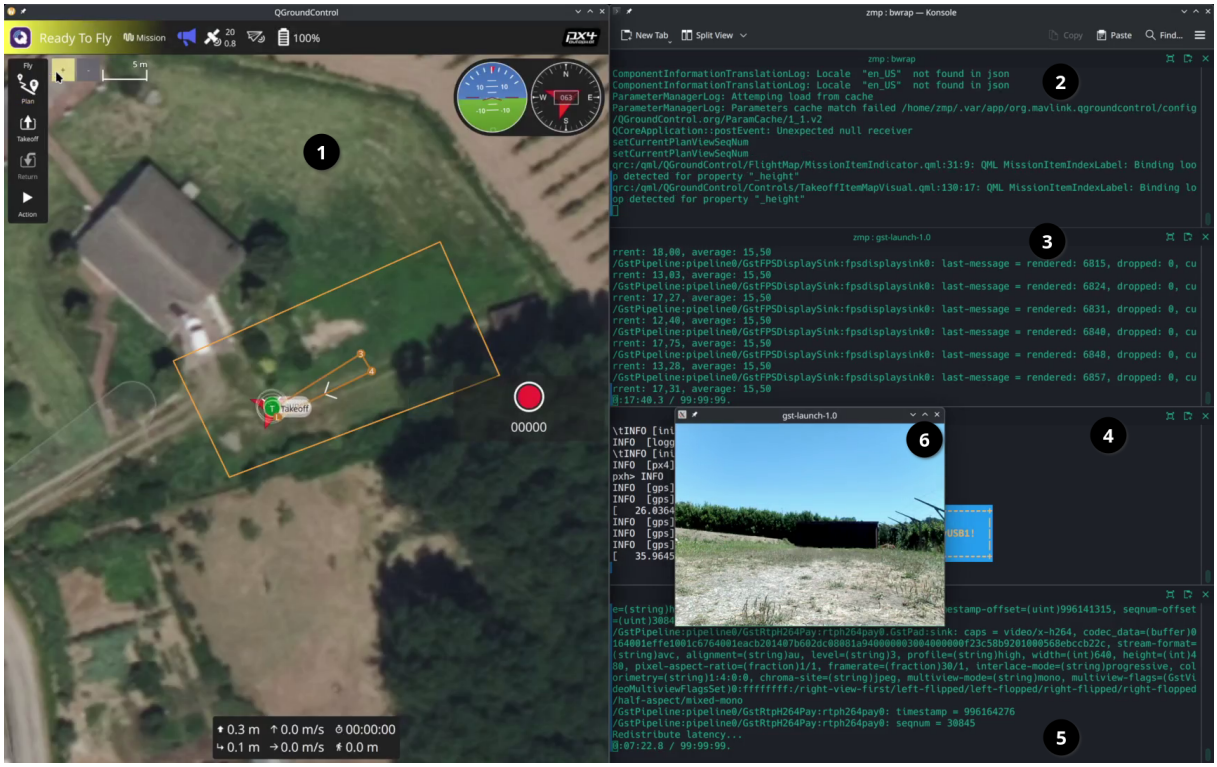
The null hypothesis (H_0) states no statistical difference between systems:

$$H_0 : \mu_{\text{USPFS}}(\tau) = \mu_{\text{SSPFS}}(\tau) \tag{5.8}$$

H_0 was rejected at the $\alpha = 0.05$ significance level when $p(\tau) < 0.05$. The `cmpLogs.py` Python script (see [26]) contains implementation details for the log analysis.

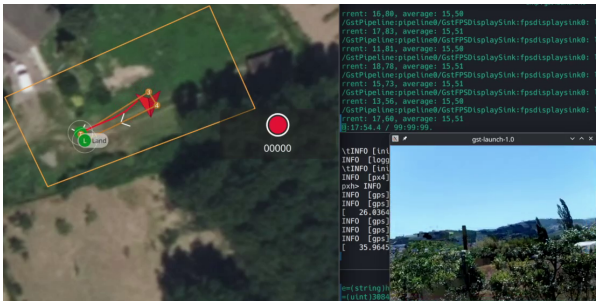
Fig. 45 illustrates key phases of a mission executing on the SSPFS system, while Fig. 46 shows the actual flight path. Fig. 45a depicts the takeoff stage alongside the software components used, including: (1) QGroundControl for mission flight management and logging on the host system (2); (3) the `gstreamer` receiver pipeline with its Graphical User Interface (GUI) executing on the host (6); the U-Boot loader and the FMU VM console responsible for PX4 execution (4); and the `gstreamer` sender pipeline running on the Companion VM, accessed via a remote SSH connection over Wi-Fi.

Fig. 45b and Fig. 45c show the flight path progression through the intermediate waypoints and the corresponding video streaming. Fig. 45d demonstrates the landing stage, showing the UAV operator, and shortly after the mission is completed successfully (Fig. 45e). These results confirm that the SSPFS system functions as designed: the autopilot operates within the FMU VM, exchanging data with the GCS through the telemetry radio link, while video streaming executes on the `Companion` VM, capturing live feed from the USB camera and transmitting it over Wi-Fi.



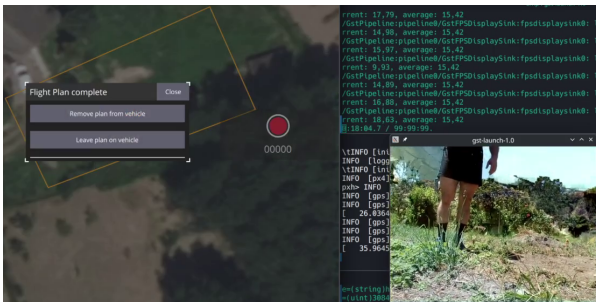
(a) Takeoff

(b) Point 3



(c) Point 4

(d) Landing



(e) Complete

Figure 45: Mission execution — SSPFS case

To interpret the test results we analyze the actual mission flight path, illustrated as a red line in Fig. 46. Mission paths exhibit slight variations due to differing sensor estimates and weather conditions. As anticipated, the autopilot dynamically adjusts trajectories according to vehicle flight dynamics and parameters rather than strictly following predefined paths. This adaptation is particularly evident during

the sharp turn between points 3 and 4. Consequently, although PX4 tries to adapt the trajectory to follow smooth curves (defined by the acceptance radius parameter `NAV_ACC_RAD` [224]), the velocity modulation during approach and departure phases may limit this, based on jerk-limiting tuning parameters.

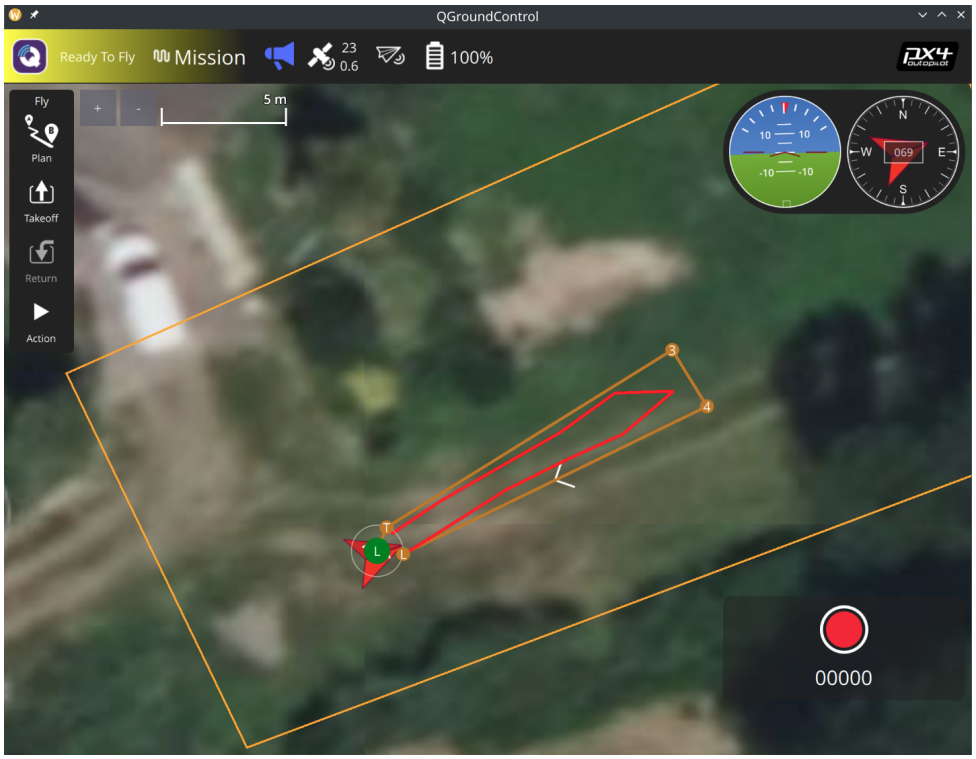


Figure 46: Actual flight path of a mission in QGroundControl

Fig. 47 presents the comparison in position tracking performance between the USPFS and SSPFS systems across X, Y, and Z axes. The solid lines indicate the actual trajectories, while the dotted ones represent the setpoint trajectories. The blue and orange traces correspond to the USPFS and SSPFS systems, respectively, with the shaded bands denoting the 95% confidence intervals for the mean. The controller demonstrates accurate position tracking, with the observed trajectories closely following setpoints except during takeoff and landing phases due to flight dynamics. Notably, setpoint variations occur between systems despite identical missions. This can be attributed to differing flight dynamics influenced by environmental factors, particularly air pressure variations affecting Z-axis estimates [225, 226], and sensor estimation discrepancies, such as noise in the GPS measurements. The significant overlap between the actual and setpoint bands indicates no statistical difference in X and Z-axis tracking between systems. However, a consistent Y-axis offset of approximately 2 meters is observed. Further data collection is needed to investigate the underlying causes of this deviation. Nonetheless, adding supervision to the autopilot shows no adverse impact on overall position tracking performance.

We compared resource usage between the USPFS and SSPFS using PX4 uORB-derived proxies rather than direct host-side measurements (Fig. 48). The supervised system shows an $\approx 6\%$ increase in CPU load and about a ninefold rise in PX4-reported RAM use. The CPU delta is consistent with prior Bao results (Section 5.4). The RAM increase is best explained as the combination of *SSPFS baseline costs*

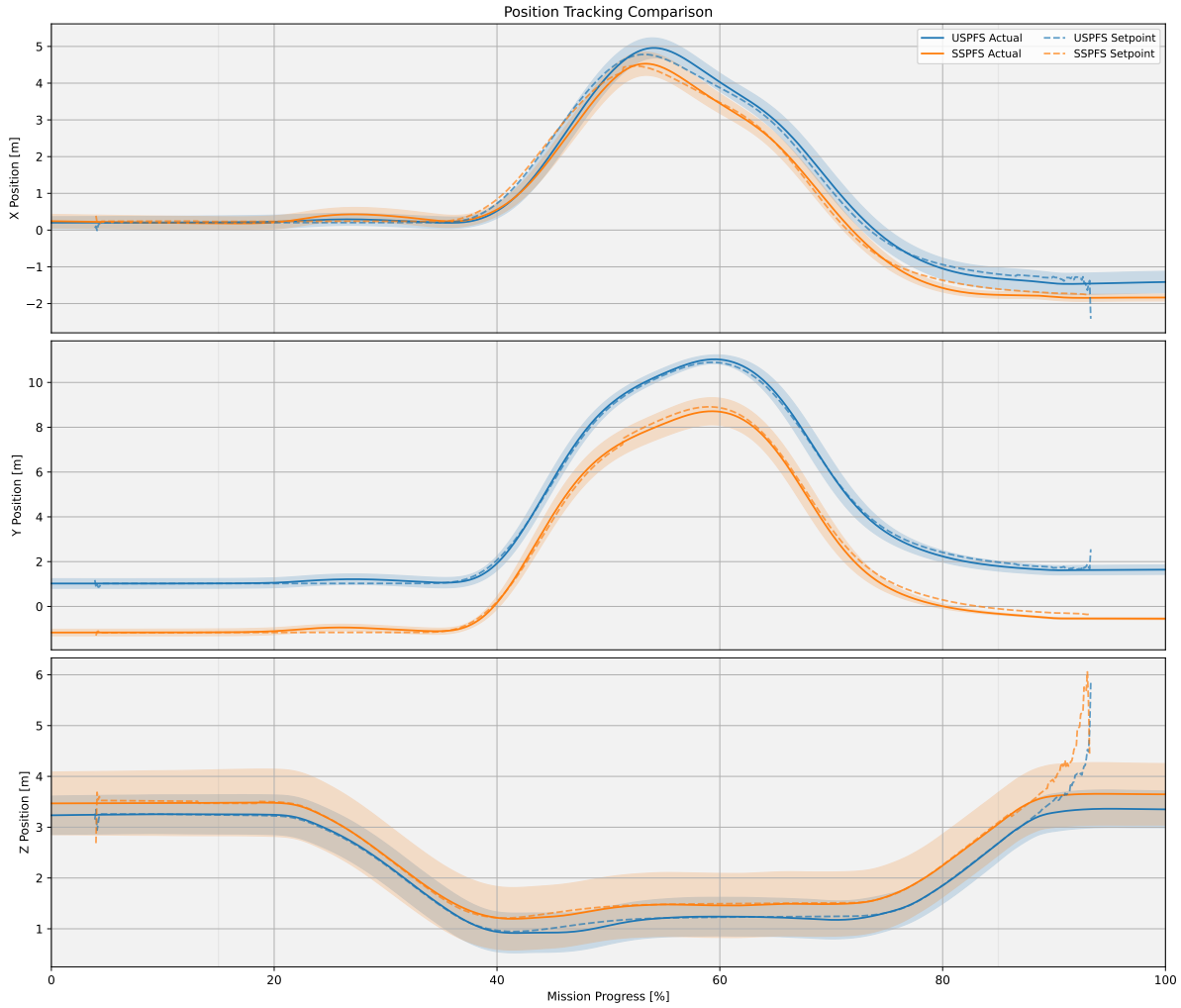


Figure 47: Position tracking comparison between USPFS and SSPFS systems

(per-guest stage-2 page tables and small Bao metadata/state) and *mailbox-related driver allocations* in Linux (transient request/response buffers and slab caching), rather than the mailbox supervisor alone. No hypervisor-managed shared-memory window is introduced by our design. In absolute terms, the increase is small relative to the FMU VM's 144 MB budget (about 90 kB in SSPFS versus 10 kB in USPFS). Because these metrics are indirect, host-side accounting would be required to analyze overhead more precisely.

The functional tests were replicated in real-flight scenarios. They are fully detailed in videos available in the online repository [26]. Fig. 49 demonstrates the SSPFS system behavior during a system compromise. Fig. 49a shows the software tools used and the UAV's flight view: (1) QGroundControl mission planning and tracking; (2) `gstreamer` receiver pipeline running at the GCS and its GUI (6); telemetry data collected by PX4 and accessible through QGroundControl (3); (4) U-Boot loader and FMU VM executing PX4; (5) `Companion` VM running the `gstreamer` sender pipeline and accessible via SSH over Wi-Fi; (7) and a remote SSH shell to compromise the Companion VM. We observe that the video streaming is operating normally during takeoff and that the UAV starts to fly (Fig. 49b). We then executed the malicious kernel module within the `Companion` VM. Video streaming immediately froze, indicating compromised

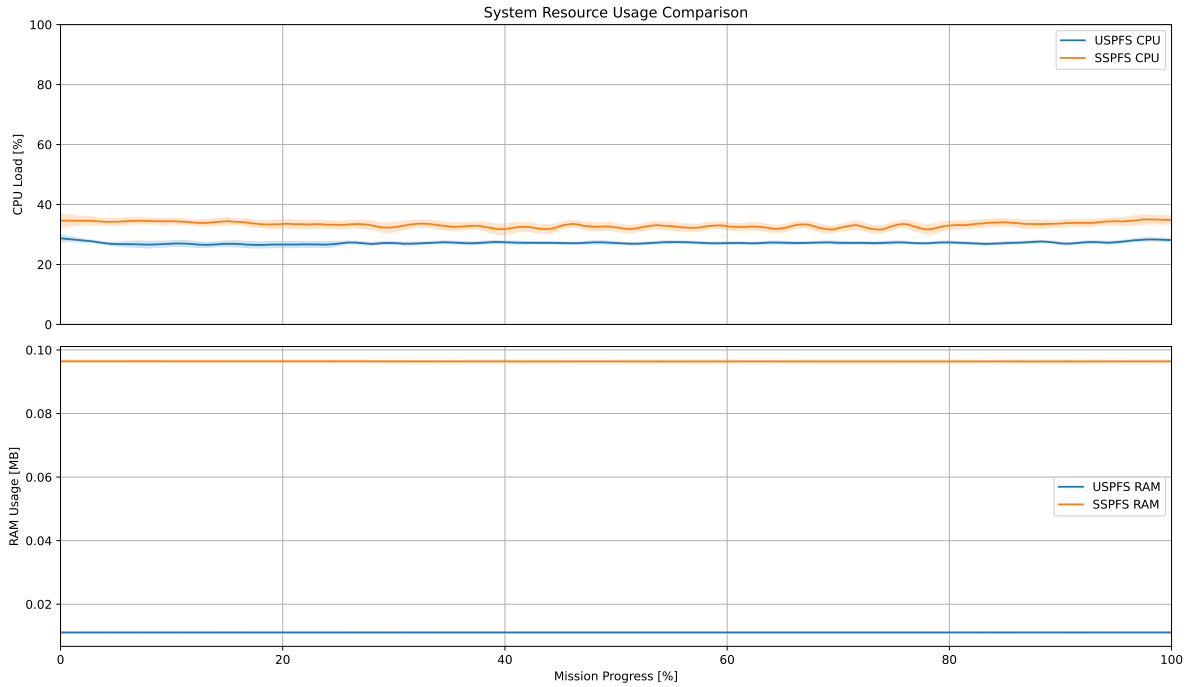
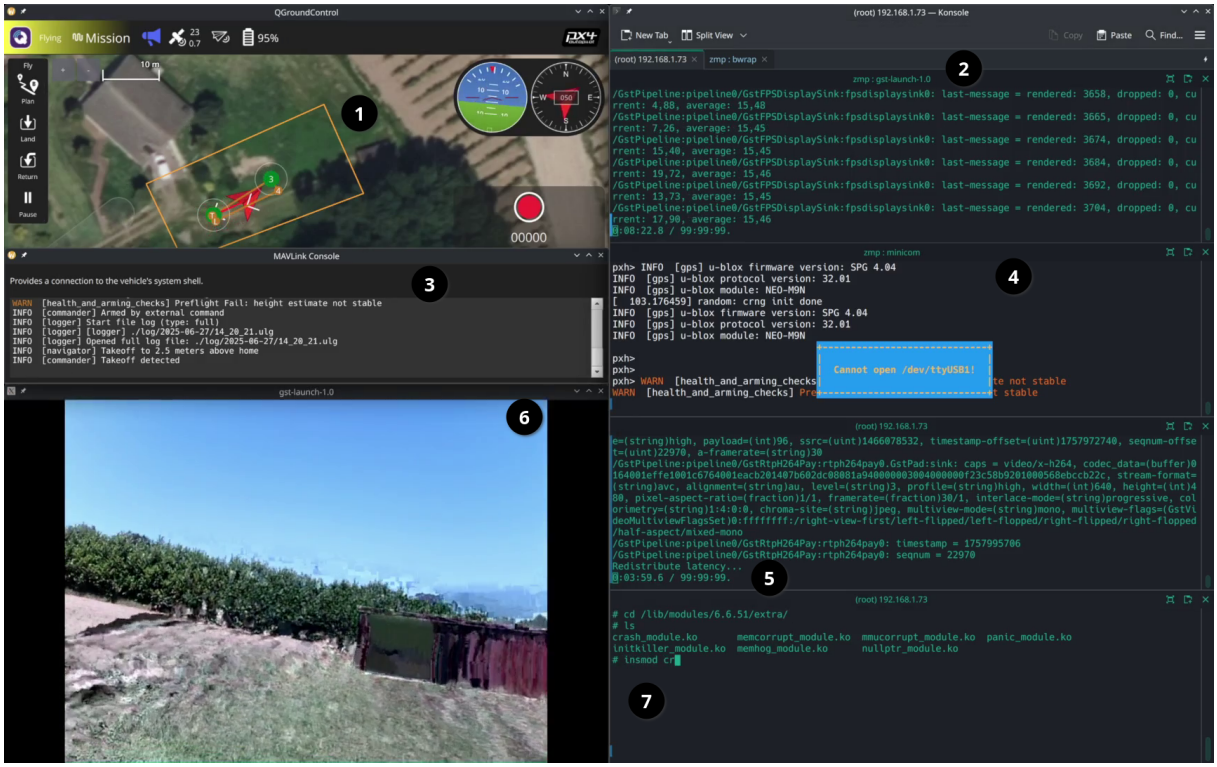


Figure 48: System resource usage comparison between USPFS and SSPFS systems

functionality in this VM. Despite this failure, the UAV lands (Fig. 49c), indicating that PX4 remained fully operational and, thus, the mission completed successfully. These results demonstrate Bao's hypervisor capability to effectively isolate the critical flight stack (FMU VM) from non-critical components (`Companion` VM). Consequently, crashes in the `Companion` VM remain contained without propagating to the autopilot, preventing potential catastrophic UAV failure.

The same functional test procedure was subsequently applied to the USPFS system. Fig. 50 demonstrates the USPFS system behavior during a system compromise. Fig. 50a shows the software tools used and the UAV's flight view: (1) QGroundControl mission planning and tracking, and logs (2); (3) `gstreamer` receiver pipeline running at the GCS and its GUI (4); telemetry data collected by PX4 and accessible through QGroundControl (3); (6) PX4 application running in the USPFS; (5) `gstreamer` sender pipeline running on the USPFS and accessible via SSH over Wi-Fi; (7) a remote SSH shell to compromise the USPFS system. We observe that video streaming operates normally during takeoff and that the UAV starts to fly (Fig. 50b). We then executed the malicious kernel module within the USPFS system. Video streaming immediately froze, indicating system failure, and the UAV began erratic maneuvers until it crashed (Fig. 50c). This outcome demonstrates that in unsupervised architectures, non-critical component failures propagate system-wide, resulting in catastrophic UAV loss.

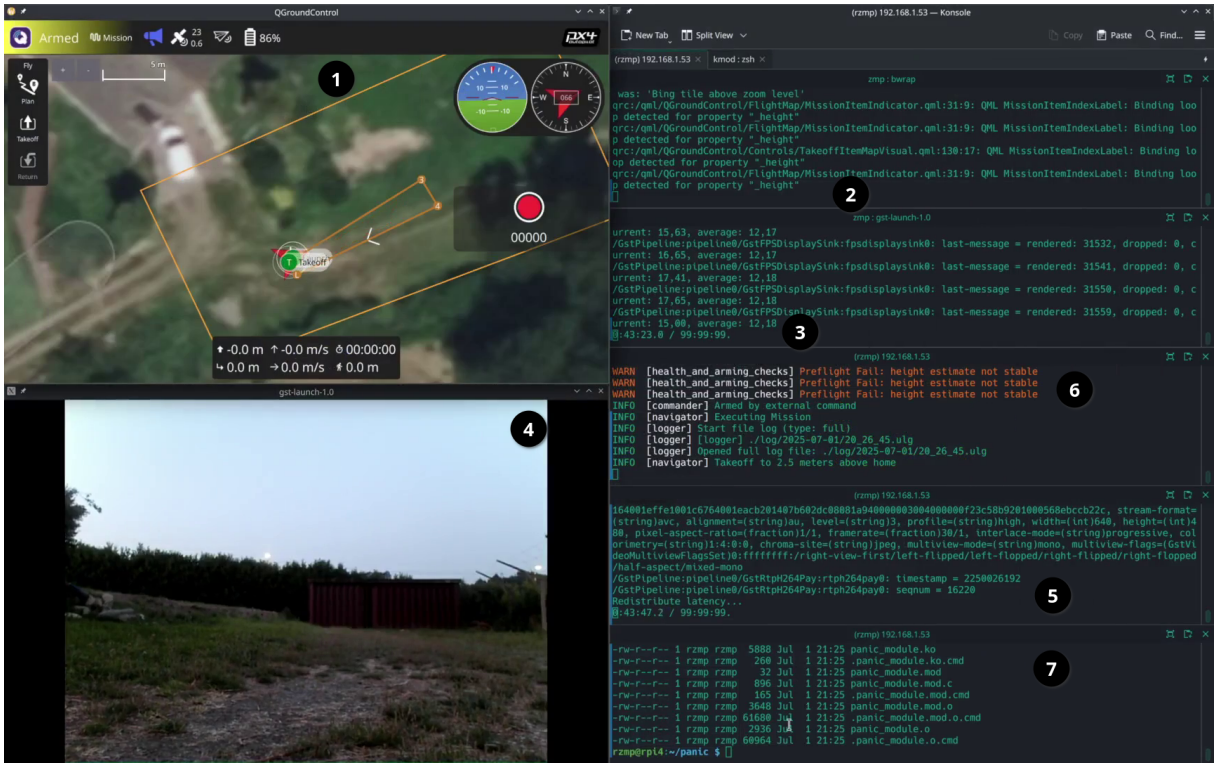


(a) UAV's view: takeoff

(b) Operator's view: takeoff

(c) Operator's view: successful landing

Figure 49: Mission execution: functional test (SSPFS)



(a) UAV's view: takeoff

(b) Operator's view: takeoff

(c) Operator's view: UAV crash

Figure 50: Mission execution: functional test (USPFS)

6

Conclusions and Future Work

"The only way of discovering the limits of the possible is to venture a little way past them into the impossible."
– Arthur C. Clarke, science fiction writer and inventor

This chapter presents the findings with respect to the research questions, the main conclusions, the resulting contributions, and prospects for future work.

6.1 Findings w.r.t. Research Questions

We summarize the main findings for each research question (RQ), based on offline benchmarking and real-flight experiments.

- **RQ1 (Timing under consolidation).** Offline timing measurements (perf + MiBench AICS) show small performance overheads attributable to supervision and that interference effects can be mitigated via cache coloring. These findings are consistent with prior Bao evaluations [80].
- **RQ2 (Fault containment / control authority).** When faults were confined to the mission VM (CPU/memory abuse and user-space resource-starvation patterns) SSPFS contained failures to that domain: the PX4 VM preserved control authority and the UAV remained airborne. The unsupervised USPFS baseline exhibited fault propagation and system-level instability under comparable conditions, causing a catastrophic UAV failure.
- **RQ3 (Overheads in offline and application-level metrics).** Relative to USPFS, SSPFS introduced small overheads: PX4 scheduling overhead $\approx 2\%$ (worst case) and camera frame-rate overhead $\approx 2\%$ (not statistically significant across runs). Offline MiBench AICS benchmarks showed modest slowdowns consistent with consolidation costs.
- **RQ4 (Position tracking and system resources during flight).** Real-flight logs show accurate position tracking, i.e., trajectories closely follow setpoints outside take-off/landing transients.

Average CPU overhead was $\approx 6\%$ and additional RAM usage for the flight VM was small (tens of kilobytes), indicating supervision costs remain modest at system scale.

- **RQ5 (Mailbox supervision for shared devices).** The mailbox-supervision mechanism enabled safe, practical use of firmware-mediated services (e.g., PCIe-related): legitimate transactions completed with negligible extra latency, and no isolation violations were observed under stress scenarios.

6.2 Conclusions

This work designed, implemented, and evaluated a trustworthy open-source software stack for UAV applications with an emphasis on security and safety, aiming to narrow the gap between open-source and commercial offerings. Conventional multi-board stacks provide physical separation but increase weight and inter-board latency. Pure containerization on single boards improves integration yet lacks kernel-level separation. Prior single-board consolidations often depend on closed components or protect only mission software, leaving critical paths exposed.

To overcome these limitations, we designed, implemented, and evaluated the Supervised Single-Platform Flight Stack (SSPFS), which consolidates flight-control and companion workloads on a single platform under the Bao hypervisor, demonstrating: (i) robust isolation under realistic fault conditions, (ii) low performance overhead on commodity hardware, and (iii) practical mitigation of shared-resource interference (cache coloring) and shared-firmware paths (mailbox supervision).

In summary, SSPFS leveraging Bao was shown to be an effective solution for consolidating mixed-criticality stacks in UAVs, offering strong isolation with low overhead. We achieved the main objective of this thesis: the design, implementation, and validation of a trustworthy, open-source reference software stack for UAV applications.

6.3 Contributions

The main contributions of the present work are:

1. **A trustworthy consolidation architecture** (SSPFS) for co-locating flight control (PX4) and a live video pipeline on Raspberry Pi 4 + PilotPi under Bao.

2. **A supervised mailbox mechanism** for safe sharing of firmware-mediated peripherals across VMs on the selected platform.

3. **A comprehensive evaluation** combining offline MiBench AICS benchmarks and real-flight experiments, including targeted fault injection to validate isolation.

6.4 Future Work

Based on the insights gained from this work, several avenues for future research and development can be pursued:

- **Broader UAVIC support (toward RT flight on smaller footprints).** Our consolidation was demonstrated on Raspberry Pi 4 + PilotPi with a Linux GPOS flight guest. While effective, Linux does not provide hard real-time guarantees. A next step is to target *native* PX4 on NuttX (smaller footprint, tighter timing), on a platform that supports it well (e.g., NXP i.MX 8M Nano [199]). Concretely: (i) complete/maintain NuttX bring-up on the chosen UAVIC; (ii) map devices for pass-through under Bao; (iii) re-evaluate timing and footprint versus the current Linux-based FMU guest.

- **Deeper system-resource accounting and memory optimizations.** Our CPU/RAM results used PX4 uORB-derived proxies. A host-side accounting pass (hypervisor- and guest-level) would attribute overheads more precisely (e.g., stage-2 page tables, Bao metadata/state, driver buffers). With that visibility, we can pursue optimizations in the mailbox path and allocator behavior. In parallel, we will evaluate alternative sharing mechanisms (e.g., VirtIO for suitable devices), which Bao already supports [69, 72–75]. On platforms where PX4 runs on NuttX with direct device ownership, we may avoid inter-guest device sharing entirely.

- **Extended UAV benchmarking and adversarial testing.** The current flight evaluation analyzed a limited set of PX4 topics. We plan to expand telemetry coverage (additional control, estimator, and sensor topics) and to repeat the experiments on a NuttX-based UAVIC. Beyond the validated fault models, longer-duration and higher-intensity adversarial scenarios (e.g., coordinated CPU/memory pressure with I/O bursts) will stress isolation over time and under interference.

- **Advanced shared-resource management.** Cache coloring mitigated some interference but is constrained by hardware realities (e.g., DMA traffic). Next steps include memory-bandwidth throttling which is already in-line with Bao's roadmap [52] and can be evaluated using the same AICS plus flight stack setup.

- **Optimized interrupt virtualization.** Bao currently targets Arm GICv2/v3 [69], which requires hypervisor re-injection. Supporting GICv4 [77] would enable direct guest delivery, reducing interrupt latency and hypervisor complexity. This requires a different UAVIC (Raspberry Pi 4 is GICv2 [227, 228]; i.MX 8M Nano is GICv3 [229]) or future SoCs with GICv4.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[2] M. Silvagni et al. "Multipurpose UAV for search and rescue operations in mountain avalanche events". en. In: *Geomatics, Natural Hazards and Risk* 8.1 (2017-01), pp. 18–33. issn: 1947-5705, 1947-5713. doi: 10.1080/19475705.2016.1238852. url: https://www.tandfonline.com/doi/full/10.1080/19475705.2016.1238852 (visited on 2025-10-27) (cit. on pp. 1, 13).

[3] D. T. Lammers et al. "Airborne! UAV delivery of blood products and medical logistics for combat zones". en. In: *Transfusion* 63.S3 (2023-05). issn: 0041-1132, 1537-2995. doi: 10.1111/trf.17329. url: https://onlinelibrary.wiley.com/doi/10.1111/trf.17329 (visited on 2025-10-27) (cit. on pp. 1, 13).

[4] D. C. Tsouros, S. Bibi, and P. G. Sarigiannidis. "A Review on UAV-Based Applications for Precision Agriculture". en. In: *Information* 10.11 (2019-11), p. 349. issn: 2078-2489. doi: 10.3390/info10110349. url: https://www.mdpi.com/2078-2489/10/11/349 (visited on 2025-10-27) (cit. on pp. 1, 13).

[5] C. Yu et al. "UAV-based pipeline inspection system with Swin Transformer for the EAST". en. In: *Fusion Engineering and Design* 184 (2022-11), p. 113277. issn: 09203796. doi: 10.1016/j.fusengdes.2022.113277. url: https://linkinghub.elsevier.com/retrieve/pii/S0920379622002691 (visited on 2025-10-27) (cit. on pp. 1, 13).

[6] N. Dilshad et al. "Applications and Challenges in Video Surveillance via Drone: A Brief Survey". In: *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. Jeju, Korea (South): IEEE, 2020-10, pp. 728–732. isbn: 978-1-7281-6758-9. doi: 10.1109/ICTC49870.2020.9289536. url: https://ieeexplore.ieee.org/document/9289536/ (visited on 2025-10-27) (cit. on pp. 1, 13).

[7] G. Caroti, A. Piemonte, and R. Nespoli. “UAV-Borne photogrammetry: a low cost 3D surveying methodology for cartographic update”. In: *MATEC Web of Conferences* 120 (2017). Ed. by A. Shanableh et al., p. 09005. issn: 2261-236X. doi: 10.1051/matecconf/201712009005. url: http://www.matec-conferences.org/10.1051/matecconf/201712009005 (visited on 2025-10-25) (cit. on pp. 1, 13, 17, 36).

[8] B. Nassi et al. “SoK: Security and Privacy in the Age of Commercial Drones”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1434–1451. doi: 10.1109/SP40001.2021.00005 (cit. on pp. 1, 2, 13, 18).

[9] M. Leccadito et al. “A survey on securing UAS cyber physical systems”. In: *IEEE Aerospace and Electronic Systems Magazine* 33.10 (2018-10), pp. 22–32. issn: 0885-8985, 1557-959X. doi: 10.1109/MAES.2018.160145. url: https://ieeexplore.ieee.org/document/8538989/ (visited on 2025-10-26) (cit. on pp. 1, 18, 19, 25).

[10] C. G. L. Krishna and R. R. Murphy. “A review on cybersecurity vulnerabilities for unmanned aerial vehicles”. In: *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. 2017, pp. 194–199. doi: 10.1109/SSRR.2017.8088163 (cit. on pp. 1, 18).

[11] K. Mansfield et al. “Unmanned aerial vehicle smart device ground control station cyber security threat model”. In: *2013 IEEE International Conference on Technologies for Homeland Security (HST)*. 2013, pp. 722–728. doi: 10.1109/THS.2013.6699093 (cit. on pp. 1, 18).

[12] sUAS News. *US Army calls for units to discontinue use of DJI equipment*. 2017. url: https://www.suasnews.com/2017/08/us-army-calls-units-discontinue-use-dji-equipment/ (visited on 2024-12-05) (cit. on pp. 1, 18, 23).

[13] The Independent. *Man arrested for landing 'radioactive' drone on japanese prime minister's roof*. 2015. url: https://www.independent.co.uk/news/world/asia/man-arrested-for-landing-radioactive-drone-on-japanese-prime-ministers-roof-10203517.html (visited on 2024-12-05) (cit. on pp. 1, 18).

[14] New York Times. *Venezuelan President Target By Drone Attack, Officials say*. 2018. url: https://www.nytimes.com/2018/08/04/world/americas/venezuelan-president-targeted-in-attack-attempt-minister-says.html (visited on 2024-12-05) (cit. on pp. 1, 18).

[15] The Drive. *Russia Offers New Details About Syrian Mass Drone Attack, Now Implies Ukrainian Connection*. 2019. url: https://www.thedrive.com/the-war-zone/17595/russia-offers-new-details-about-syrian-mass-drone-attack-now-implies-ukrainian-connection (visited on 2024-12-05) (cit. on pp. 1, 14, 16, 18).

[16] S. A. H. Mohsan et al. “Towards the Unmanned Aerial Vehicles (UAVs): A Comprehensive Review”. en. In: *Drones* 6.6 (2022-06), p. 147. issn: 2504-446X. doi: 10.3390/drones6060147. url: https://www.mdpi.com/2504-446X/6/6/147 (visited on 2025-10-25) (cit. on pp. 1, 7, 13–16, 18, 36).

[17] Z. Ullah, F. Al-Turjman, and L. Mostarda. “Cognition in UAV-Aided 5G and Beyond Communications: A Survey”. In: *IEEE Transactions on Cognitive Communications and Networking* 6.3 (2020-09), pp. 872–891. issn: 2332-7731, 2372-2045. doi: 10.1109/TCCN.2020.2968311. url: https://ieeexplore.ieee.org/document/8964329/ (visited on 2025-10-26) (cit. on pp. 1, 13).

[18] J. Glossner, S. Murphy, and D. Iancu. *An Overview of the Drone Open-Source Ecosystem*. Version Number: 1. 2021. doi: 10.48550/ARXIV.2110.02260. url: https://arxiv.org/abs/2110.02260 (visited on 2025-10-26) (cit. on pp. 1, 12, 13, 17, 28).

[19] G. Klein et al. “Formally verified software in the real world”. en. In: *Communications of the ACM* 61.10 (2018-09), pp. 68–77. issn: 0001-0782, 1557-7317. doi: 10.1145/3230627. url: https://dl.acm.org/doi/10.1145/3230627 (visited on 2025-10-29) (cit. on pp. 1, 33, 35).

[20] D. Wang et al. “An exploratory study of autopilot software bugs in unmanned aerial vehicles”. en. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, 2021-08, pp. 20–31. isbn: 978-1-4503-8562-6. doi: 10.1145/3468264.3468559. url: https://dl.acm.org/doi/10.1145/3468264.3468559 (visited on 2025-10-29) (cit. on pp. 1, 2, 31, 34).

[21] K. K. G. Buquerin. “Security evaluation for the real-time operating system VxWorks 7 for avionic systems”. PhD Thesis. Technische Hochschule Ingolstadt, 2018. url: https://kmyr.de/thesis/Bachelorarbeit-Buquerin.pdf (visited on 2025-10-26) (cit. on pp. 1, 2, 32, 34).

[22] Auterion. *Skynode S*. 2024. url: https://auterion.com/product/skynode-s/ (visited on 2024-08-30) (cit. on p. 2).

[23] Auterion. *Structuring Applications with Multiple Services*. 2024. url: https://docs.auterion.com/app-development/app-framework/structuring-applications-with-multiple-services (visited on 2024-08-07) (cit. on pp. 2, 9, 29, 33).

[24] B. Wang et al. “Enabling High-Performance Onboard Computing with Virtualization for Unmanned Aerial Systems”. In: *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*. Dallas, TX: IEEE, 2018-06, pp. 202–211. isbn: 978-1-5386-1354-2. doi: 10.1109/ICUAS.2018.8453368. url: https://ieeexplore.ieee.org/document/8453368/ (visited on 2025-10-29) (cit. on pp. 2, 33, 35).

[25] E. Cittadini et al. “Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology”. en. In: *Real-Time Systems* 59.4 (2023-12), pp. 609–635. issn: 0922-6443, 1573-1383. doi: 10.1007/s11241-023-09402-4. url: https://link.springer.com/10.1007/s11241-023-09402-4 (visited on 2025-10-29) (cit. on pp. 2, 32, 33, 35).

[26] Jose Pires. *Trustworthy Open-Source Reference SW stack for UAV applications*. GitHub repository. 2025. url: https://github.com/ElectroQuanta/trustworthy-oss-uav-stack (visited on 2025-10-23) (cit. on pp. 5, 50, 53, 55, 59–62, 65–68, 71, 73, 74, 78, 81).

[27] A. Burns and R. I. Davis. *Mixed Criticality Systems - A Review:(13th Edition, February 2022)*. en. York, 2022-02. url: https://eprints.whiterose.ac.uk/id/eprint/183619/ (visited on 2025-10-26) (cit. on pp. 6–8).

[28] R. I. Davis, S. Altmeyer, and A. Burns. “Mixed Criticality Systems with Varying Context Switch Costs”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Porto: IEEE, 2018-04, pp. 140–151. isbn: 978-1-5386-5295-4. doi: 10.1109/RTAS.2018.00024. url: https://ieeexplore.ieee.org/document/8430078/ (visited on 2025-10-27) (cit. on pp. 6, 7).

[29] R. Debouk. “Overview of the Second Edition of ISO 26262: Functional Safety— Road Vehicles”. In: *Journal of System Safety* 55.1 (2019-03), pp. 13–21. issn: 0743-8826. doi: 10.56094/jss.v55i1.55. url: https://jsystemsafety.com/index.php/jss/article/view/55 (visited on 2025-10-26) (cit. on p. 6).

[30] R. (SC 167. *Software considerations in airborne systems and equipment certification*. RTCA, Incorporated, 1992 (cit. on pp. 6, 7).

[31] W. K. Youn et al. “Software certification of safety-critical avionic systems: DO-178C and its impacts”. In: *IEEE Aerospace and Electronic Systems Magazine* 30.4 (2015-04), pp. 4–13. issn: 0885-8985. doi: 10.1109/MAES.2014.140109. url: https://ieeexplore.ieee.org/document/7104300 (visited on 2025-10-27) (cit. on pp. 6, 7).

[32] E. Yip et al. “Relaxing the synchronous approach for mixed-criticality systems”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Berlin, Germany: IEEE, 2014-04, pp. 89–100. isbn: 978-1-4799-4829-1 978-1-4799-4691-4. doi: 10.1109/RTAS.2014.6925993. url: http://ieeexplore.ieee.org/document/6925993/ (visited on 2025-10-27) (cit. on p. 7).

[33] M. Cinque et al. “Virtualizing mixed-criticality systems: A survey on industrial trends and issues”. In: *Future Generation Computer Systems* 129 (2022), pp. 315–330. issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2021.12.002. url: https://www.sciencedirect.com/science/article/pii/S0167739X21004787 (cit. on pp. 7–10).

[34] T. Alladi et al. “Applications of blockchain in unmanned aerial vehicles: A review”. In: *Vehicular Communications* 23 (2020), p. 100249. issn: 2214-2096. doi: https://doi.org/10.1016/j.vehcom.2020.100249. url: https://www.sciencedirect.com/science/article/pii/S2214209620300206 (cit. on pp. 7, 12, 14, 16).

[35] B. S. Faiçal et al. “An adaptive approach for UAV-based pesticide spraying in dynamic environments”. en. In: *Computers and Electronics in Agriculture* 138 (2017-06), pp. 210–223. issn: 01681699. doi: 10.1016/j.compag.2017.04.011. url: https://linkinghub.elsevier.com/retrieve/pii/S0168169916304173 (visited on 2025-10-28) (cit. on p. 7).

[36] J. Lee et al. “Estimating Probabilistic Safe WCET Ranges of Real-Time Systems at Design Stages”. en. In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023-04), pp. 1–33. issn: 1049-331X, 1557-7392. doi: 10.1145/3546941. url: https://dl.acm.org/doi/10.1145/3546941 (visited on 2025-10-28) (cit. on p. 8).

[37] M. Sollfrank et al. “Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation”. In: *IEEE Transactions on Industrial Informatics* 17.5 (2021-05), pp. 3566–3576. issn: 1551-3203, 1941-0050. doi: 10.1109/TII.2020.3022843. url: https://ieeexplore.ieee.org/document/9187833/ (visited on 2025-10-28) (cit. on p. 8).

[38] K. Kumar and M. Kurhekar. “Economically Efficient Virtualization over Cloud Using Docker Containers”. In: *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. Bangalore, India: IEEE, 2016-10, pp. 95–100. isbn: 978-1-5090-4573-0. doi: 10.1109/CCEM.2016.025. url: http://ieeexplore.ieee.org/document/7819678/ (visited on 2025-10-28) (cit. on p. 8).

[39] Xilinx. *RunX*. GitHub repository. 2020. url: https://github.com/Xilinx/runx (visited on 2024-11-27) (cit. on p. 9).

[40] V. Struhár et al. “Real-Time Containers: A Survey”. en. In: *OASIcs, Volume 80, Fog-IoT 2020* 80 (2020). Ed. by A. Cervin and Y. Yang. Artwork Size: 9 pages, 380638 bytes ISBN: 9783959771443 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 7:1–7:9. issn: 2190-6807. doi: 10.4230/OASICS.FOG-IOT.2020.7. url: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Fog-IoT.2020.7 (visited on 2025-10-26) (cit. on p. 9).

[41] N. Pathak et al. “UAV Virtualization for Enabling Heterogeneous and Persistent UAV-as-a-Service”. In: *IEEE Transactions on Vehicular Technology* 69.6 (2020-06), pp. 6731–6738. issn: 0018-9545, 1939-9359. doi: 10.1109/TVT.2020.2985913. url: https://ieeexplore.ieee.org/document/9072443/ (visited on 2025-10-28) (cit. on p. 9).

[42] Y. Liu, J. Yan, and X. Zhao. “Deep Reinforcement Learning Based Latency Minimization for Mobile Edge Computing With Virtualization in Maritime UAV Communication Network”. In: *IEEE Transactions on Vehicular Technology* 71.4 (2022-04), pp. 4225–4236. issn: 0018-9545, 1939-9359. doi: 10.1109/TVT.2022.3141799. url: https://ieeexplore.ieee.org/document/9678008/ (visited on 2025-10-28) (cit. on p. 9).

[43] B. Nogales et al. “Adaptable and Automated Small UAV Deployments via Virtualization”. en. In: *Sensors* 18.12 (2018-11), p. 4116. issn: 1424-8220. doi: 10.3390/s18124116. url: https://www.mdpi.com/1424-8220/18/12/4116 (visited on 2025-10-28) (cit. on p. 9).

[44] A. Kumari et al. “A taxonomy of blockchain-enabled softwarization for secure UAV network”. en. In: *Computer Communications* 161 (2020-09), pp. 304–323. issn: 01403664. doi: 10.1016/j.comcom.2020.07.042. url: https://linkinghub.elsevier.com/retrieve/pii/S0140366420318545 (visited on 2025-10-28) (cit. on p. 9).

[45] D. Cotroneo, L. De Simone, and R. Natella. “NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems”. In: *IEEE Transactions on Network and Service Management* 14.4 (2017-12), pp. 934–948. issn: 1932-4537. doi: 10.1109/TNSM.2017.2733042. url: http://ieeexplore.ieee.org/document/7995070/ (visited on 2025-10-28) (cit. on p. 9).

[46] M. Sara, I. Jawhar, and M. Nader. “A Softwarization Architecture for UAVs and WSNs as Part of the Cloud Environment”. In: *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. Berlin, Germany: IEEE, 2016-04, pp. 13–18. isbn: 978-1-5090-3684-4. doi: 10.1109/IC2EW.2016.17. url: http://ieeexplore.ieee.org/document/7527808/ (visited on 2025-10-28) (cit. on p. 9).

[47] M. A. B. Siddiki Abir, M. Z. Chowdhury, and Y. M. Jang. “Software-Defined UAV Networks for 6G Systems: Requirements, Opportunities, Emerging Techniques, Challenges, and Research Directions”. In: *IEEE Open Journal of the Communications Society* 4 (2023), pp. 2487–2547. issn: 2644-125X. doi: 10.1109/OJCOMS.2023.3323200. url: https://ieeexplore.ieee.org/document/10274880/ (visited on 2025-10-28) (cit. on p. 9).

[48] O. Sami Oubbati et al. “Softwarization of UAV Networks: A Survey of Applications and Future Trends”. In: *IEEE Access* 8 (2020), pp. 98073–98125. doi: 10.1109/ACCESS.2020.2994494 (cit. on p. 9).

[49] Y. Li et al. “A distributed framework for multiple UAV cooperative target search under dynamic environment”. en. In: *Journal of the Franklin Institute* 361.8 (2024-05), p. 106810. issn: 00160032. doi: 10.1016/j.jfranklin.2024.106810. url: https://linkinghub.elsevier.com/retrieve/pii/S001600322400231X (visited on 2025-10-27) (cit. on pp. 9, 17).

[50] M. M. Azari et al. “UAV-to-UAV Communications in Cellular Networks”. In: *IEEE Transactions on Wireless Communications* 19.9 (2020-09), pp. 6130–6144. issn: 1536-1276, 1558-2248. doi: 10.1109/TWC.2020.3000303. url: https://ieeexplore.ieee.org/document/9115898/ (visited on 2025-10-27) (cit. on pp. 9, 17).

[51] Siemens A.G. *Jailhouse hypervisor source code*. GitHub repository. url: https://github.com/siemens/jailhouse (visited on 2024-11-30) (cit. on pp. 10, 11).

[52] J. Martins et al. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. Ed. by M. Bertogna and F. Terraneo. Vol. 77. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:14. isbn: 978-3-95977-136-8. doi: 10.4230/OASIcs.NG-RES.2020.3. url: https://drops.dagstuhl.de/opus/volltexte/2020/11779 (cit. on pp. 10, 11, 35, 87).

[53] Accelerat. *CLARE hypervisor*. 2022. url: https://accelerat.eu/clare/ (visited on 2024-12-08) (cit. on pp. 10, 33).

[54] Xen project. *Xen hypervisor*. 2022. url: https://xenproject.org/ (visited on 2024-12-08) (cit. on p. 10).

[55] A. Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux symposium*. Vol. 1. Issue: 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230. url: https://www.kernel.org/doc/mirror/ols2007v1.pdf#page=225 (visited on 2025-10-26) (cit. on p. 10).

[56] Microsoft Corporation. *Hyper-V*. 2020. url: https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview (visited on 2024-11-30) (cit. on p. 10).

[57] P. Barham et al. “Xen and the art of virtualization”. en. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003-12), pp. 164–177. issn: 0163-5980. doi: 10.1145/1165389.945462. url: https://dl.acm.org/doi/10.1145/1165389.945462 (visited on 2025-10-26) (cit. on pp. 10, 11).

[58] G. Heiser. “The role of virtualization in embedded systems”. en. In: *Proceedings of the 1st workshop on Isolation and integration in embedded systems*. Glasgow Scotland: ACM, 2008-04, pp. 11–16. isbn: 978-1-60558-126-2. doi: 10.1145/1435458.1435461. url: https://dl.acm.org/doi/10.1145/1435458.1435461 (visited on 2025-10-26) (cit. on p. 10).

[59] SysGO. *PikeOS product overview*. url: https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/redaktion/downloads/pdf/data-sheets/SYSGO-Product-Overview-PikeOS.pdf (visited on 2024-11-30) (cit. on pp. 10, 33, 35).

[60] M. Masmano et al. “Xtratum: a hypervisor for safety critical embedded systems”. In: *11th Real-Time Linux Workshop*. Vol. 9. Citeseer, 2009. url: https://www.academia.edu/download/96082900/proceedings_2009.pdf#page=273 (visited on 2025-10-26) (cit. on p. 10).

[61] G. Klein et al. “seL4: formal verification of an OS kernel”. en. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Big Sky Montana USA: ACM, 2009-10, pp. 207–220. isbn: 978-1-60558-752-3. doi: 10.1145/1629575.1629596. url: https://dl.acm.org/doi/10.1145/1629575.1629596 (visited on 2025-10-28) (cit. on p. 10).

[62] E. D. Matos and M. Ahvenjärvi. “seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM”. en. In: *Electronics* 11.24 (2022-12), p. 4201. issn: 2079-9292. doi: 10.3390/electronics11244201. url: https://www.mdpi.com/2079-9292/11/24/4201 (visited on 2025-10-28) (cit. on p. 10).

[63] S. Pinto et al. “Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems”. In: *IEEE Computer Architecture Letters* 16.2 (2017-07), pp. 158–161. issn: 1556-6056. doi: 10.1109/LCA.2016.2617308. url: http://ieeexplore.ieee.org/document/7590042/ (visited on 2025-10-26) (cit. on p. 10).

[64] J. Martins et al. “µTZVisor: A Secure and Safe Real-Time Hypervisor”. In: *Electronics* 6.4 (2017). issn: 2079-9292. doi: 10.3390/electronics6040093. url: https://www.mdpi.com/2079-9292/6/4/93 (cit. on p. 10).

[65] P. Lucas et al. “VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A”. en. In: *LIPIcs, Volume 76, ECRTS 2017* 76 (2017). Ed. by M. Bertogna. Artwork Size: 18 pages, 780702 bytes ISBN: 9783959770378 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 6:1–6:18. issn: 1868-8969. doi: 10.4230/LIPICS.ECRTS.2017.6. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2017.6 (visited on 2025-10-26) (cit. on p. 10).

[66] P. Lucas et al. “VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System”. In: *2018 23rd Conference of Open Innovations Association (FRUCT)*. Bologna: IEEE, 2018-11, pp. 231–238. isbn: 978-952-68653-6-2. doi: 10.23919/FRUCT.2018.8588018. url: https://ieeexplore.ieee.org/document/8588018/ (visited on 2025-10-26) (cit. on p. 10).

[67] J. Martins et al. “{ClickOS} and the Art of Network Function Virtualization”. en. In: 2014, pp. 459–473. isbn: 978-1-931971-09-6. url: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins (visited on 2025-10-26) (cit. on p. 10).

[68] S. Lankes, S. Pickartz, and J. Breitbart. “HermitCore: A Unikernel for Extreme Scale Computing”. en. In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. Kyoto Japan: ACM, 2016-06, pp. 1–8. isbn: 978-1-4503-4387-9. doi: 10.114

5/2931088.2931093. url: https://dl.acm.org/doi/10.1145/2931088.2931093 (visited on 2025-10-26) (cit. on p. 10).

[69] Embedded Systems Research Group v3. *Bao hypervisor*. GitHub repository. 2020. url: https://github.com/bao-project/bao-hypervisor (visited on 2024-11-27) (cit. on pp. 10–12, 35, 87).

[70] A. Bansal et al. "Evaluating memory subsystem of configurable heterogeneous MPSoC". In: *Proceedings of the Operating Systems Platforms for Embedded Real-Time applications* (2018). url: https://research.utwente.nl/files/276565013/proceedings_ospert2018.pdf#page=56 (visited on 2025-10-26) (cit. on p. 11).

[71] Q. Ge et al. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* 8.1 (2018), pp. 1–27. doi: 10.1007/s13389-016-0141-6. url: https://doi.org/10.1007/s13389-016-0141-6 (cit. on p. 11).

[72] J. R. L. Costa. "VirtIO infrastructure for a static partition hypervisor: VirtIO-Net". eng. MA thesis. 2023-01. url: https://hdl.handle.net/1822/88717 (visited on 2025-10-26) (cit. on pp. 11, 87).

[73] A. N. A. C. C. Ribeiro. "VirtIO infrastructure for a static partition hypervisor: virtio-blk and virtio-console". eng. MA thesis. 2023-04. url: https://hdl.handle.net/1822/91640 (visited on 2025-10-26) (cit. on pp. 11, 87).

[74] F. M. B. Rocha. "Mitigating platform-level memory interference on a static partitioning hypervisor". eng. MA thesis. 2023-01. url: https://hdl.handle.net/1822/88668 (visited on 2025-10-26) (cit. on pp. 11, 87).

[75] J. Peixoto et al. "BiRtIO: VirtIO for Real-Time Network Interface Sharing on the Bao Hypervisor". In: *IEEE Access* 12 (2024), pp. 185434–185447. doi: 10.1109/ACCESS.2024.3512777 (cit. on pp. 11, 87).

[76] J. Martins and S. Pinto. "Bao: A modern lightweight embedded hypervisor". In: *Proc. Embedded World Conf.* 2020, pp. 1–5. url: https://sandro2pinto.github.io/files/ew2020-bao.pdf (visited on 2025-10-26) (cit. on p. 11).

[77] ARM. *A brief history of the Arm CoreLink GIC*. 2025. url: https://developer.arm.com/documentation/198123/0302/What-is-a-Generic-Interrupt-Controller- (visited on 2025-06-15) (cit. on pp. 11, 87).

[78] C. Dall. *The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization*. Columbia University, 2018 (cit. on p. 11).

[79] F. Marques et al. ""Interrupting" the Status Quo: A First Glance at the RISC-V Advanced Interrupt Architecture (AIA)". In: *IEEE Access* 12 (2024), pp. 9822–9833. issn: 2169-3536. doi: 10.1109/ACCESS.2024.3352114. url: https://ieeexplore.ieee.org/document/10387321/ (visited on 2025-10-28) (cit. on p. 11).

[80] J. Martins and S. Pinto. "Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems". In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. San Antonio, TX, USA: IEEE, 2023-05, pp. 40–53. isbn: 979-8-3503-2176-0. doi: 10.1109/RTAS58335.2023.00011. url: https://ieeexplore.ieee.org/document/10155684/ (visited on 2025-10-26) (cit. on pp. 12, 35, 70, 85).

[81] ConsortIQ. *A not-so-short history of Unmanned Aerial Vehicles (UAV)*. 2022. url: https://consortiq.com/uas-resources/short-history-unmanned-aerial-vehicles-uavs (visited on 2024-11-30) (cit. on p. 13).

[82] Paparazzi. *Paparazzi home page*. 2022. url: https://wiki.paparazziuav.org/wiki/Main_Page (visited on 2024-12-08) (cit. on pp. 13, 29).

[83] ArduPilot. *History of ArduPilot*. 2022. url: https://ardupilot.org/copter/docs/common-history-of-ardupilot.html (visited on 2024-11-30) (cit. on pp. 13, 28).

[84] Auterion. *The story of PX4 and Pixhawk*. 2022. url: https://auterion.com/company/the-history-of-pixhawk/ (visited on 2024-11-30) (cit. on p. 13).

[85] A. Fotouhi et al. "Survey on UAV Cellular Communications: Practical Aspects, Standardization Advancements, Regulation, and Security Challenges". In: *IEEE Communications Surveys & Tutorials* 21.4 (2019-10), pp. 3417–3442. issn: 1553-877X. doi: 10.1109/COMST.2019.2906228 (cit. on p. 13).

[86] C. Stöcker et al. "Review of the Current State of UAV Regulations". In: *Remote Sensing* 9.5 (2017). issn: 2072-4292. doi: 10.3390/rs9050459. url: https://www.mdpi.com/2072-4292/9/5/459 (cit. on pp. 13, 18).

[87] PX4. *EBlimp Atom*. Projects. 2024. url: https://px4.io/project/eblimp-atom/ (visited on 2024-09-22) (cit. on pp. 13, 16).

[88] Hackster.io. *Using PX4 GPS Autopilot to Guide a Glider Home*. 2024. url: https://www.hackster.io/news/using-px4-gps-autopilot-to-guide-a-glider-home-e89aa4078d80 (visited on 2024-09-22) (cit. on pp. 13, 16).

[89] The Drone Bird Company. *Home page*. 2024. url: https://www.thedronebird.com/ (visited on 2024-09-22) (cit. on pp. 13, 16).

[90] Parrot. *Parrot drones*. 2022. url: https://www.parrot.com/us/drones (visited on 2024-11-30) (cit. on pp. 14–16).

[91] Dji. *Dji Mavic 3*. 2022. url: https://www.dji.com/pt/mavic-3 (visited on 2024-11-30) (cit. on pp. 14, 16).

[92] Energyor. *Energyor H2Quad 1000*. 2022. url: http://energyor.com/products/detail/h2quad-1000 (visited on 2024-11-30) (cit. on pp. 14–16).

[93] Flyability. *Gas powered drone: A guide*. 2022. url: https://www.flyability.com/gas-powered-drone (visited on 2024-11-30) (cit. on pp. 14, 16).

[94] Flaperon. *Flaperon MX8*. 2022. url: https://flaperon.com/ (visited on 2024-11-30) (cit. on pp. 14, 16).

[95] HobbyKing. *HobbyKing KK2.1.5 Multi-rotor LCD flight control board*. 2024. url: https://hobbyking.com/hobbyking-kk2-1-5-multi-rotor-lcd-flight-control-board-with-6050mpu-and-atmel-644pa.html (visited on 2024-12-12) (cit. on pp. 14, 16, 23).

[96] Auterion. *Skynode X*. 2024. url: https://auterion.com/product/skynode-x/ (visited on 2024-07-30) (cit. on pp. 14, 16, 25).

[97] WindRiver Blog. *Northrop Grumman X-47B UCAS-D Running on Wind River VxWorks Catapults from Aircraft Carrier*. 2013. url: https://blogs.windriver.com/wind_river_blog/2013/05/historic-milestone-for-northrops-x-47b-and-wind-river/ (visited on 2024-12-09) (cit. on pp. 14, 16, 29).

[98] Defense Express. *Ukraine Receives Skynode S Universal Machine Vision for Drones from American Company Auterion*. 2024. url: https://en.defence-ua.com/weapon_and_tech/ukraine_receives_skynode_s_universal_machine_vision_for_drones_from_american_company_auterion-11011.html (visited on 2024-08-05) (cit. on pp. 14, 16, 25).

[99] Drone Analyst. *The rise of open-source drones*. 2021. url: https://droneanalyst.com/2021/05/30/rise-of-open-source-drones (visited on 2024-11-30) (cit. on pp. 14, 16, 23).

[100] Dji. *Dji Software Development Kit*. 2022. url: https://developer.dji.com/ (visited on 2024-11-30) (cit. on pp. 14, 16, 29).

[101] The Verge. *Yuneec announces Typhoon H Plus alongise first fixed-wing and racing drones*. 2022. url: https://www.theverge.com/2018/1/9/16867090/yuneec-typhoon-h-plus-firebird-fpv-hd-racer-drones-ces-2018 (visited on 2024-11-30) (cit. on pp. 15, 16).

[102] Velos Rotors. *Velos UAV*. 2022. url: https://www.velosuav.com/velosv3/ (visited on 2024-11-30) (cit. on pp. 15, 16).

[103] DeltaQuad. *DeltaQuad Pro VTOL UAV*. 2022. url: https://www.deltaquad.com/ (visited on 2024-11-30) (cit. on pp. 15, 16).

[104] Advanced Aircraft Company. *Hybrid Advanced Multi-Rotor (HAMR)*. Product homepage. 2022. url: https://advancedaircraftcompany.com/hamr/ (visited on 2024-11-30) (cit. on p. 15).

[105] XSun. *Solar X One*. 2022. url: https://xsun.fr/autonomous-drone/ (visited on 2024-11-30) (cit. on p. 15).

[106] Avinc. *Pathfinder Plus*. 2024. url: https://www.avinc.com/innovative-solutions/hale-uas (visited on 2024-11-28) (cit. on p. 16).

[107] GetFPV. *BETAFPV Meteor65 Pro*. 2024. url: https://www.getfpv.com/ready-to-fly-quadcopters/micro-ready-to-fly/betafpv-meteor65-pro-brushless-whoop-hd-w-dji-o4-elrs-2-4ghz.html (visited on 2024-11-28) (cit. on p. 16).

[108] NXP Semiconductors. *NXP HoverGames drone kit including RDDRONE-FMUK66 and peripherals*. url: https://www.nxp.com/design/design-center/development-boards-and-designs/nxp-hovergames-drone-kit-including-rddrone-fmuk66-and-peripherals:KIT-HGDRONEK66 (visited on 2024-06-08) (cit. on pp. 16, 42, 43).

[109] Edge Group. *Hunter10*. 2024. url: https://edgegroupuae.com/solutions/hunter-10 (visited on 2024-11-28) (cit. on p. 16).

[110] Army Recognition. *Shadow 50 Loitering Munition Expands Deep-Strike Precision Against Fixed Targets*. 2025. url: https://www.armyrecognition.com/news/army-news/2025/idef-2025-shadow-50-loitering-munition-expands-deep-strike-precision-against-fixed-targets (visited on 2025-08-28) (cit. on p. 16).

[111] Edge Group. *Reach-S*. 2024. url: https://edgegroupuae.com/solutions/reach-s (visited on 2024-11-28) (cit. on p. 16).

[112] Holybro. *X650 Development Kit*. 2024. url: https://holybro.com/collections/multicopter-kit/products/x650-development-kit (visited on 2024-11-28) (cit. on p. 16).

[113] S. A. H. Mohsan et al. "A Comprehensive Review of Micro UAV Charging Techniques". en. In: *Micromachines* 13.6 (2022-06), p. 977. issn: 2072-666X. doi: 10.3390/mi13060977. url: https://www.mdpi.com/2072-666X/13/6/977 (visited on 2025-10-27) (cit. on p. 15).

[114] S. Aggarwal and N. Kumar. "Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges". In: *Computer Communications* 149 (2020), pp. 270–299. issn: 0140-3664. doi: https://doi.org/10.1016/j.comcom.2019.10.014. url: https://www.sciencedirect.com/science/article/pii/S0140366419308539 (cit. on pp. 15–17).

[115] PX4 Autopilot. *PX4 System Architecture*. 2022. url: https://docs.px4.io/main/en/concept/px4_systems_architecture.html (visited on 2024-12-08) (cit. on pp. 17, 19, 25, 28).

[116] T. Liang, T. Zhang, and Q. Zhang. "Toward Seamless Localization and Communication: A Satellite-UAV NTN Architecture". In: *IEEE Network* 38.4 (2024-07), pp. 103–110. issn: 0890-8044, 1558-156X. doi: 10.1109/MNET.2024.3384298. url: https://ieeexplore.ieee.org/document/10488448/ (visited on 2025-10-27) (cit. on p. 17).

[117] E. Alvarez-Vanhard, T. Corpetti, and T. Houet. "UAV & satellite synergies for optical remote sensing applications: A literature review". en. In: *Science of Remote Sensing* 3 (2021-06), p. 100019. issn: 26660172. doi: 10.1016/j.srs.2021.100019. url: https://linkinghub.elsevier.com/retrieve/pii/S2666017221000067 (visited on 2025-10-27) (cit. on p. 17).

[118] S. Cai et al. "Branch architecture quantification of large-scale coniferous forest plots using UAV-LiDAR data". en. In: *Remote Sensing of Environment* 306 (2024-05), p. 114121. issn: 00344257. doi: 10.1016/j.rse.2024.114121. url: https://linkinghub.elsevier.com/retrieve/pii/S0034425724001329 (visited on 2025-10-27) (cit. on p. 17).

[119] T. Vogeltanz. "A Survey of Free Software for the Design, Analysis, Modelling, and Simulation of an Unmanned Aerial Vehicle". In: *Archives of Computational Methods in Engineering* 23.3 (2016), pp. 449–514. doi: 10.1007/s11831-015-9147-y. url: https://doi.org/10.1007/s11831-015-9147-y (cit. on p. 17).

[120] E. Ebeid, M. Skriver, and J. Jin. "A Survey on Open-Source Flight Control Platforms of Unmanned Aerial Vehicle". In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, pp. 396–402. doi: 10.1109/DSD.2017.30 (cit. on pp. 17, 21).

[121] B. A. Malyshev et al. "Research of Electronic Speed Controllers Designs and Functional for Unmanned Aerial Vehicles". In: *2024 IEEE 25th International Conference of Young Professionals in Electron Devices and Materials (EDM)*. IEEE, 2024-06, pp. 1150–1155. isbn: 979-8-3503-8923-4. doi: 10.1109/EDM61683.2024.10615216. url: https://ieeexplore.ieee.org/document/10615216/ (visited on 2025-10-27) (cit. on p. 17).

[122] D. L. Gabriel, J. Meyer, and F. du Plessis. "Brushless DC motor characterisation and selection for a fixed wing UAV". In: *IEEE Africon '11*. 2011, pp. 1–6. doi: 10.1109/AFRCON.2011.6072087 (cit. on p. 17).

[123] J. García and J. M. Molina. "Simulation in real conditions of navigation and obstacle avoidance with PX4/Gazebo platform". en. In: *Personal and Ubiquitous Computing* 26.4 (2022-08), pp. 1171–1191. issn: 1617-4909, 1617-4917. doi: 10.1007/s00779-019-01356-4. url: https://link.springer.com/10.1007/s00779-019-01356-4 (visited on 2025-10-27) (cit. on p. 17).

[124] PX4. *Simulation*. PX4 Guide. 2024. url: https://docs.px4.io/main/en/simulation/ (visited on 2025-05-22) (cit. on p. 17).

[125] ArduPilot. *SITL Simulator (Software in the Loop)*. Documentation. 2024. url: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html (visited on 2025-05-22) (cit. on p. 17).

[126] Coppelia Robotics. *CoppeliaSim*. 2024. url: https://www.coppeliarobotics.com/ (visited on 2024-11-28) (cit. on p. 17).

[127] FlightGear. *FlightGear free open-source fligh simulator*. 2024. url: https://www.flightgear.org/ (visited on 2024-11-28) (cit. on p. 18).

[128] PX4 Autopilot. *Opens Source Autopilot for Drone Developers*. 2022. url: https://px4.io/ (visited on 2024-12-08) (cit. on p. 18).

[129] ArduPilot. *ArduPilot, Home*. 2022. url: https://ardupilot.org/ (visited on 2024-12-08) (cit. on pp. 18, 28).

[130] LibrePilot. *LibrePilot system architecture*. 2022. url: https://librepilot.atlassian.net/wiki/spaces/LPDOC/pages/100523730/LibrePilot+System+Architecture (visited on 2024-12-12) (cit. on pp. 18, 28).

[131] PX4. *RPi PilotPi Shield*. url: https://docs.px4.io/main/en/flight_controller/raspberry_pi_pilotpi.html (visited on 2024-06-13) (cit. on pp. 18, 21, 22, 32, 44).

[132] Avionics International. *Wind River VxWorks 653 Providing Power for Airbus Helionix*. 2016. url: https://www.aviationtoday.com/2016/05/11/wind-river-vxworks-653-providing-power-for-airbus-helionix/ (visited on 2024-12-09) (cit. on pp. 18, 29).

[133] Paparazzi UAS. *Paparazzi UAS Autopilot*. GitHub repository. 2022. url: https://github.com/paparazzi/paparazzi (visited on 2024-12-08) (cit. on pp. 18, 21, 28).

[134] iNAV. *iNAV*. Github repository. 2022. url: https://github.com/iNavFlight/inav (visited on 2024-12-12) (cit. on pp. 18, 29).

[135] D. Sathyamoorthy. "A review of security threats of unmanned aerial vehicles and mitigation steps". In: *J. Def. Secur* 6.1 (2015), pp. 81–97. url: https://www.researchgate.net/profile/Dinesh-Sathyamoorthy/publication/282443666_A_Review_of_Security_Threats_of_Unmanned_Aerial_Vehicles_and_Mitigation_Steps/links/56105c9f08ae48337519f39d/A-Review-of-Security-Threats-of-Unmanned-Aerial-Vehicles-and-Mitigation-Steps.pdf (visited on 2025-10-26) (cit. on p. 18).

[136] I. G. Ferrao et al. "STUART: ReSilient archiTecture to dynamically manage Unmanned aeriAl vehicle networks under atTack". In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. Rennes, France: IEEE, 2020-07, pp. 1–6. isbn: 978-1-7281-8086-1. doi: 10.1109/ISCC50000.2020.9219689. url: https://ieeexplore.ieee.org/document/9219689/ (visited on 2025-10-26) (cit. on pp. 18, 19).

[137] Free Software Foundation. *GNU Operating System*. 2022. url: https://gnu.org/philosophy/free-hardware-designs.en.html (visited on 2024-12-05) (cit. on pp. 20, 28).

[138] RoboKits India. *ArduPilot Mega APM 2.8 flight controller*. 2024. url: https://robokits.co.in/multirotor-spare-parts/flight-controller-and-frame/ardupilot-mega-apm-2.8-flight-controller-arduino-compatible-with-compass (visited on 2024-12-12) (cit. on p. 21).

[139] Discuss. *Ardupilot or Ardupilot Mega: active projects*. 2022. url: https://discuss.ardupilot.org/t/ardupilot-or-ardupilot-mega-active-projects/92917/3 (visited on 2024-12-18) (cit. on p. 21).

[140] Pixhawk. *Pixhawk: The reference standards*. 2023. url: https://pixhawk.org/ (visited on 2024-12-16) (cit. on p. 21).

[141] PX4 Autopilot. *Pixhawk 4*. 2022. url: https://docs.px4.io/main/en/flight_controller/pixhawk4.html (visited on 2024-12-07) (cit. on pp. 21, 22, 32, 35).

[142] ArduPilot. *CUAV v5 Plus overview wiki page*. 2022. url: https://ardupilot.org/copter/docs/common-cuav-v5plus-overview.html (visited on 2024-12-07) (cit. on pp. 21, 32).

[143] Paparazzi. *Paparazzi Chimera v1.00 Wiki*. 2022. url: https://wiki.paparazziuav.org/wiki/Chimera/v1.00 (visited on 2024-12-07) (cit. on p. 21).

[144] ROS Robots. *Erle Brain*. 2022. url: https://robots.ros.org/erle-brain/ (visited on 2024-12-19) (cit. on p. 21).

[145] Ardupilot. *PXFMini wiki*. 2022. url: https://ardupilot.org/copter/docs/common-pxfmini.html (visited on 2024-12-19) (cit. on p. 21).

[146] Dojo For Drones. *PXFMini shield for Pi Zero drones*. 2022. url: https://dojofordrones.com/pxfmini/ (visited on 2024-12-19) (cit. on p. 21).

[147] DroneDJ. *After product ban, the US DoD formally blacklists drone giant DJI (Update)*. 2022. url: https://dronedj.com/2022/10/07/dji-dod-drone/ (visited on 2024-12-08) (cit. on p. 23).

[148] SeriouslyPro. *SPRacing H7 extreme specifications*. 2022. url: http://seriouslypro.com/products/spracingh7extreme%5C#technical-specifications (visited on 2024-12-08) (cit. on p. 23).

[149] ArduPilot. *SPRacing H7 extreme*. 2022. url: https://ardupilot.org/copter/docs/common-spracingh7-extreme.html (visited on 2024-12-08) (cit. on p. 23).

[150] Dronecode. *Aerotenna OcPoC-Zynq Mini Flight Controller*. 2022. url: https://docs.px4.io/v1.11/en/flight_controller/ocpoc_zynq.html (visited on 2024-12-08) (cit. on pp. 23, 32).

[151] PX4. *Aerotenna OcPoC-Zynq Mini Flight Controller:Discontinued*. 2022. url: https://docs.px4.io/v1.13/en/flight_controller/ocpoc_zynq.html (visited on 2024-12-08) (cit. on pp. 23, 32).

[152] EMLID. *Preconfigured Raspberry Pi OS image for Navio2*. 2022. url: https://docs.emlid.com/navio2/configuring-raspberry-pi/ (visited on 2024-12-12) (cit. on p. 23).

[153] ArduPilot. *Navio2 Overview*. 2022. url: https://ardupilot.org/copter/docs/common-navio2-overview.html (visited on 2024-12-07) (cit. on pp. 23, 24).

[154] PX4. *Raspberry Pi 2/3/4 Navio2 Autopilot*. 2022. url: https://docs.px4.io/main/en/flight_controller/raspberry_pi_navio2 (visited on 2024-12-12) (cit. on pp. 23, 32).

[155] ArduPilot. *Horizon31 PixC4-Jetson*. 2022. url: https://ardupilot.org/copter/docs/common-horizon31-pixc4-jetson.html (visited on 2024-12-08) (cit. on p. 24).

[156] Horizon. *PixC4-Jetson documentation*. 2021. url: https://echomav.com/wp-content/uploads/2021/10/PixC4-Jetson-Documentation.pdf (visited on 2024-12-10) (cit. on pp. 24, 32, 35).

[157] Auterion. *Skynode X Datasheet*. 2024. url: https://3329189600-files.gitbook.io/~/files/v0/b/gitbook-x-prod.appspot.com/o/spaces%5C%2FFW1Ge1p1f6WHyiYCb146%5C%2Fuploads%5C%2Fg5ziQAabZrGnMfLuX7OC%5C%2FSkynode%5C%20X%5C%20Datasheet.pdf?alt=media&token=4c36ecb2-4299-4551-afa5-2a1d8c83841a (visited on 2024-07-30) (cit. on p. 25).

[158] PX4. *Auterion Skynode X*. 2022. url: https://docs.px4.io/main/en/companion_computer/auterion_skynode (visited on 2024-12-15) (cit. on pp. 25, 28).

[159] LucidBots Shop. *Auterion Flight Controller Sky Node - Enterprise Edition*. 2024. url: https://lucidbots.shop/products/auterion-flight-conroller-sky-node (visited on 2024-07-30) (cit. on p. 25).

[160] Auterion. *Auterion Announces New All-In-One solution for Small Unmanned Systems*. 2024. url: https://auterion.com/auterion-announces-new-all-in-one-solution-for-small-unmanned-systems/ (visited on 2024-07-30) (cit. on p. 25).

[161] Breaking Defense. *Auterion Announces New All-In-One solution for Small Unmanned Systems*. 2024. url: https://breakingdefense.com/2024/06/skynode-s-auterion-autonomy-kit-lets-attack-drones-fly-through-jamming/ (visited on 2024-08-05) (cit. on p. 25).

[162] PX4. *PX4 Autopilot*. GitHub repository. 2022. url: https://github.com/PX4/PX4-Autopilot (visited on 2024-12-08) (cit. on p. 28).

[163] ArduPilot. *ArduPilot Autopilot*. GitHub repository. 2022. url: https://github.com/ArduPilot/ardupilot (visited on 2024-12-08) (cit. on p. 28).

[164] Betaflight. *Betaflight flight controller software*. GitHub repository. 2022. url: https://github.com/betaflight/betaflight (visited on 2024-12-08) (cit. on p. 28).

[165] Cleanflight. *Cleanflight flight controller software*. GitHub repository. 2022. url: https://github.com/cleanflight/cleanflight (visited on 2024-12-08) (cit. on p. 28).

[166] Bitcraze. *Bitcraze repository overview*. 2022. url: https://www.bitcraze.io/documentation/repository/ (visited on 2024-12-18) (cit. on p. 28).

[167] LibrePilot. *LibrePilot*. BitBucket repository. 2022. url: https://bitbucket.org/librepilot/librepilot/src/next/ (visited on 2024-12-12) (cit. on p. 28).

[168] PX4. *SPRacing H7 Extreme (PX4 Edition)*. 2022. url: https://docs.px4.io/main/en/flight_controller/spracingh7extreme (visited on 2024-12-15) (cit. on p. 28).

[169] T. Jargalsaikhan et al. "Architectural Process for Flight Control Software of Unmanned Aerial Vehicle with Module-Level Portability". en. In: *Aerospace* 9.2 (2022-01), p. 62. issn: 2226-4310. doi: 10.3390/aerospace9020062. url: https://www.mdpi.com/2226-4310/9/2/62 (visited on 2025-10-26) (cit. on pp. 28, 29, 31).

[170] PX4 Autopilot. *Multicopter-specific features*. 2022. url: https://px4.io/ (visited on 2024-12-08) (cit. on p. 28).

[171] Ardupilot. *Mavlink commands: autonomous missions*. Wiki page. 2022. url: https://ardupilot.org/dev/docs/mavlink-commands.html (visited on 2024-12-08) (cit. on p. 29).

[172] Paparazzi. *Paparazzi UAS system overview*. 2022. url: https://paparazzi-uav.readthedocs.io/en/latest/developer_guide/system_overview.html (visited on 2024-12-09) (cit. on p. 29).

[173] Parrot. *Air SDK*. 2022. url: https://developer.parrot.com/docs/airsdk/general/overview.html (visited on 2024-12-17) (cit. on pp. 29, 32).

[174] PX4. *Auterion Skynode*. 2024. url: https://docs.px4.io/main/en/companion_computer/auterion_skynode.html (visited on 2024-08-06) (cit. on p. 29).

[175] Auterion. *Auterion Mission Control*. 2024. url: https://auterion.com/product/mission-control/ (visited on 2024-08-06) (cit. on p. 29).

[176] Auterion. *Auterion Suite*. 2024. url: https://auterion.com/product/suite/ (visited on 2024-08-06) (cit. on p. 29).

[177] sUAS News. *Wind River Technology Powers Airbus Group's Innovative Unmanned Aerial Vehicle ATLANTE*. 2014. url: https://www.suasnews.com/2014/09/wind-river-technology-powers-airbus-groups-innovative-unmanned-aerial-vehicle-atlante/ (visited on 2024-12-09) (cit. on p. 29).

[178] P. Li et al. "Embedding Adaptive Features in the ArduPilot Control Architecture for Unmanned Aerial Vehicles". In: *2022 IEEE 61st Conference on Decision and Control (CDC)*. Cancun, Mexico: IEEE, 2022-12, pp. 3773–3780. isbn: 978-1-6654-6761-2. doi: 10.1109/CDC51059.2022.9993292. url: https://ieeexplore.ieee.org/document/9993292/ (visited on 2025-10-29) (cit. on p. 31).

[179] P. Li, D. Liu, and S. Baldi. "Plug-and-play adaptation in autopilot architectures for unmanned aerial vehicles". In: *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*. Toronto, ON, Canada: IEEE, 2021-10, pp. 1–6. isbn: 978-1-6654-3554-3. doi: 10.1109/IECON48115.2021.9589106. url: https://ieeexplore.ieee.org/document/9589106/ (visited on 2025-10-29) (cit. on p. 31).

[180] S. Baldi et al. "ArduPilot-Based Adaptive Autopilot: Architecture and Software-in-the-Loop Experiments". In: *IEEE Transactions on Aerospace and Electronic Systems* 58.5 (2022-10), pp. 4473–4485. issn: 0018-9251, 1557-9603, 2371-9877. doi: 10.1109/TAES.2022.3162179. url: https://ieeexplore.ieee.org/document/9741360/ (visited on 2025-10-29) (cit. on p. 31).

[181] S. D'Angelo et al. "Efficient development of model-based controllers in PX4 firmware: a template-based customization approach". In: *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2024, pp. 1155–1162. url: https://www.researchgate.net/profile/Simone-Dangelo-4/publication/380604433_Efficient_Development_of_Model-Based_Controllers_in_PX4_Firmware_A_Template-Based_Customization_Approach/links/667530671dec0c3c6f985da1/Efficient-Development-of-Model-Based-Controllers-in-PX4-Firmware-A-Template-Based-Customization-Approach.pdf (visited on 2025-10-29) (cit. on p. 31).

[182] K. Dang Nguyen and T.-T. Nguyen. "Vision-Based Software-in-the-Loop-Simulation for Unmanned Aerial Vehicles Using Gazebo and PX4 Open Source". In: *2019 International Conference on System Science and Engineering (ICSSE)*. Dong Hoi, Vietnam: IEEE, 2019-07, pp. 429–432. isbn: 978-1-7281-0525-3. doi: 10.1109/ICSSE.2019.8823322. url: https://ieeexplore.ieee.org/document/8823322/ (visited on 2025-10-29) (cit. on p. 31).

[183] X. Dai et al. “RFlySim: Automatic test platform for UAV autopilot systems with FPGA-based hardware-in-the-loop simulations”. en. In: *Aerospace Science and Technology* 114 (2021-07), p. 106727. issn: 12709638. doi: 10.1016/j.ast.2021.106727. url: https://linkinghub.elsevier.com/retrieve/pii/S1270963821002376 (visited on 2025-10-29) (cit. on p. 31).

[184] M. Zhang et al. “Which Is the Best Real-Time Operating System for Drones? Evaluation of the Real-Time Characteristics of NuttX and ChibiOS”. In: *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. Athens, Greece: IEEE, 2021-06, pp. 582–590. isbn: 978-1-6654-1535-4. doi: 10.1109/ICUAS51884.2021.9476878. url: https://ieeexplore.ieee.org/document/9476878/ (visited on 2025-10-31) (cit. on p. 32).

[185] B. B. Kovari and E. Ebeid. “MPDrone: FPGA-based Platform for Intelligent Real-time Autonomous Drone Operations”. In: *2021 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. New York City, NY, USA: IEEE, 2021-10, pp. 71–76. isbn: 978-1-6654-1764-8. doi: 10.1109/SSRR53300.2021.9597857. url: https://ieeexplore.ieee.org/document/9597857/ (visited on 2025-10-29) (cit. on p. 32).

[186] L. Valente et al. “A Heterogeneous RISC-V Based SoC for Secure Nano-UAV Navigation”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 71.5 (2024-05), pp. 2266–2279. issn: 1549-8328, 1558-0806. doi: 10.1109/TCSI.2024.3359044. url: https://ieeexplore.ieee.org/document/10423921/ (visited on 2025-10-29) (cit. on pp. 33–35).

[187] T. Fautrel et al. “An hypervisor approach for mixed critical real-time UAV applications”. In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. Kyoto, Japan: IEEE, 2019-03, pp. 985–991. isbn: 978-1-5386-9151-9. doi: 10.1109/PERCOMW.2019.8730705. url: https://ieeexplore.ieee.org/document/8730705/ (visited on 2025-10-29) (cit. on pp. 33, 35).

[188] A. Farrukh and R. West. “FlyOS: rethinking integrated modular avionics for autonomous multicopters”. en. In: *Real-Time Systems* 59.2 (2023-06), pp. 256–301. issn: 0922-6443, 1573-1383. doi: 10.1007/s11241-023-09399-w. url: https://link.springer.com/10.1007/s11241-023-09399-w (visited on 2025-10-29) (cit. on pp. 34, 35).

[189] M. Danish, Y. Li, and R. West. “Virtual-CPU Scheduling in the Quest Operating System”. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Chicago, IL, USA: IEEE, 2011-04, pp. 169–179. isbn: 978-1-61284-326-1. doi: 10.1109/RTAS.2011.24. url: http://ieeexplore.ieee.org/document/5767149/ (visited on 2025-10-29) (cit. on p. 34).

[190] Quest. *Quest: a lightweight, predictable and dependable kernel for multicore processors*. Home page. 2025. url: http://www.questos.org/ (visited on 2025-04-27) (cit. on p. 34).

[191] Quest. *Quest OS*. GitHub repository. 2025. url: https://github.com/QuestOS/quest (visited on 2025-04-27) (cit. on p. 34).

[192] Y. Li, R. West, and E. Missimer. *The Quest-V Separation Kernel for Mixed Criticality Systems*. Version Number: 1. 2013. doi: 10.48550/ARXIV.1310.6298. url: https://arxiv.org/abs/1310.6298 (visited on 2025-10-29) (cit. on p. 34).

[193] Intel. *Intel Aero Compute board*. Specifications. 2025. url: https://www.intel.com/content/www/us/en/products/sku/97178/intel-aero-compute-board/specifications.html (visited on 2025-04-27) (cit. on p. 34).

[194] G. Gugan and A. Haque. "Path Planning for Autonomous Drones: Challenges and Future Directions". In: *Drones* 7.3 (2023). issn: 2504-446X. doi: 10.3390/drones7030169. url: https://www.mdpi.com/2504-446X/7/3/169 (cit. on p. 36).

[195] QGroundControl. *Survey (Plan Pattern)*. QGC Guide. 2024. url: https://docs.qgroundcontrol.com/master/en/qgc-user-guide/plan_view/pattern_survey.html (visited on 2024-09-22) (cit. on p. 36).

[196] PX4 Autopilot. *PX4 Companion Computer: Routers*. 2022. url: https://docs.px4.io/main/en/companion_computer/%5C#routers (visited on 2024-10-08) (cit. on p. 37).

[197] PX4 Autopilot. *PX4 MAVLink Cameras: Camera managers*. 2022. url: https://docs.px4.io/main/en/camera/mavlink_v2_camera.html%5C#camera-managers (visited on 2024-10-08) (cit. on p. 38).

[198] NXP Semiconductors. *PX4 Robotic Drone Vehicle/Flight Management Unit (VMU/FMU) - RDDRONE-FMUK66*. url: https://www.nxp.com/design/design-center/development-boards-and-designs/px4-robotic-drone-vehicle-flight-management-unit-vmu-fmu-rddrone-fmuk66:RDDRONE-FMUK66 (visited on 2024-06-08) (cit. on p. 42).

[199] NXP. *i.MX 8M Nano Family - Arm Cortex-A53, Cortex-M7*. 2024. url: https://www.nxp.com/products/i.MX8MNANO (visited on 2025-06-25) (cit. on pp. 43, 87).

[200] NuttX. *Supported platforms*. Documentation. 2024. url: https://nuttx.apache.org/docs/10.0.1/introduction/supported_platforms.html (visited on 2024-09-27) (cit. on p. 43).

[201] PX4. *Experimental Autopilots*. url: https://docs.px4.io/main/en/flight_controller/autopilot_experimental.html (visited on 2024-06-12) (cit. on p. 43).

[202] Raspberry Pi. *Raspberry Pi 4 Model B specifications*. url: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/ (visited on 2024-06-14) (cit. on p. 43).

[203] R. Pi. *Processors*. url: https://www.raspberrypi.com/documentation/computers/processors.html (visited on 2024-06-14) (cit. on p. 43).

[204] Raspberry Pi. *Raspberry Pi Firmware*. GitHub repository. url: https://github.com/raspberrypi/firmware/wiki (visited on 2024-06-27) (cit. on p. 45).

[205] Raspberry Pi Linux kernel Github. *Mailbox device driver*. GitHub repository. url: https://github.com/raspberrypi/linux/blob/rpi-6.6.y/drivers/firmware/raspberrypi.c (visited on 2024-07-30) (cit. on p. 45).

[206] R. P. Ltd. *BCM2711 ARM Peripherals*. Raspberry Pi Ltd, 2022, p. 6 (cit. on pp. 47, 48).

[207] Creative. *Creative Live! Cam Sync 1080p V2*. url: https://en.creative.com/p/webcams/creative-live-cam-sync-1080p-v2 (visited on 2024-08-10) (cit. on p. 48).

[208] Amazon. *EDUP AX3000M USB 6E WiFi Adapter for PC*. url: https://www.amazon.es/-/en/dp/B0DFYBZSR6 (visited on 2024-08-10) (cit. on p. 49).

[209] Morrownr. *USB WiFi Adapters that are supported with Linux in-kernel drivers*. GitHub repository. url: https://github.com/morrownr/USB-WiFi/blob/main/home/USB_WiFi_Adapters_that_are_supported_with_Linux_in-kernel_drivers.md%5C#axe3000---usb30---24-ghz-5-ghz-and-6-ghz-wifi-6e (visited on 2024-08-10) (cit. on p. 49).

[210] u-blox. *NEO-M8 series*. url: https://www.u-blox.com/sites/default/files/products/documents/NEO-M8_ProductSummary_UBX-16000345.pdf (visited on 2024-07-10) (cit. on p. 52).

[211] RTR Modelismo. *LiPo Stick Pack 11.1V-50C 5000 Hardcase (XT60)*. url: https://rtr-modelismo.com/pt/electronica-baterias-li-po/107050-lipo-stick-pack-11-1v-50c-5000-hardcase-4250650938475.html?gad_source=1 (visited on 2024-07-10) (cit. on p. 52).

[212] Jose Pires. *Bao hypervisor fork*. GitHub repository. 2025. url: https://github.com/ElectroQuanta/bao-hypervisor-porting (visited on 2025-05-27) (cit. on p. 59).

[213] Bootlin Elixir Cross Referencer. *iomap.c*. 2024. url: https://elixir.bootlin.com/linux/v6.6.53/source/lib/iomap.c%5C#L221 (visited on 2025-01-28) (cit. on p. 67).

[214] D. Costa et al. "IRQ Coloring and the Subtle Art of Mitigating Interrupt-Generated Interference". In: *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2023, pp. 47–56. doi: 10.1109/RTCSA58653.2023.00015 (cit. on p. 70).

[215] M. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. Austin, TX, USA: IEEE, 2001, pp. 3–14. isbn: 978-0-7803-7315-0. doi: 10.1109/WWC.2001.990739. url: http://ieeexplore.ieee.org/document/990739/ (visited on 2025-10-26) (cit. on p. 70).

[216] Perf wiki. *perf: Linux profiling with performance counters*. 2024. url: https://perfwiki.github.io/main/ (visited on 2025-01-28) (cit. on p. 70).

[217] J. Martins and S. Pinto. *Shedding Light repository*. GitHub repository. 2022. url: https://github.com/ESRGv3/shedding-light-static-partitioning-hypervisors (visited on 2025-01-27) (cit. on p. 70).

[218] PX4. *Module Template for Full applications – Workqueue*. 2024. url: https://docs.px4.io/main/en/modules/module_template.html (visited on 2025-01-30) (cit. on p. 73).

[219] PX4. *Modules Reference – Controller*. 2024. url: https://docs.px4.io/main/en/modules/modules_controller.html (visited on 2025-01-30) (cit. on p. 73).

[220] PX4. *Modules Reference – Drivers*. 2024. url: https://docs.px4.io/main/en/modules/modules_driver.html (visited on 2025-01-30) (cit. on p. 73).

[221] PX4. *Modules Reference – System*. 2024. url: https://docs.px4.io/main/en/modules/modules_system.html (visited on 2025-01-30) (cit. on p. 73).

[222] The Comprehensive R Archive Network. *T confidence interval for a mean*. 2024. url: https://cran.r-project.org/web/packages/distributions3/vignettes/one-sample-t-confidence-interval.html (visited on 2025-06-20) (cit. on p. 77).

[223] Danielle Navarro. *The Independent Samples t-test (Welch Test)*. LibreTexts Statistics. 2024. url: https://stats.libretexts.org/Bookshelves/Applied_Statistics/Learning_Statistics_with_R_-_A_tutorial_for_Psychology_Students_and_other_Beginners_(Navarro)/13%5C%3A_Comparing_Two_Means/13.04%5C%3A_The_Independent_Samples_t-test_(Welch_Test) (visited on 2025-06-20) (cit. on p. 78).

[224] PX4. *Mission mode (Multicopter) – Rounded turns: Inter-Waypoint Trajectory*. 2024. url: https://docs.px4.io/main/en/flight_modes_mc/mission.html%5C#rounded-turns-inter-waypoint-trajectory (visited on 2025-06-28) (cit. on p. 80).

[225] PX4. *Static pressure buildup*. PX4 Guide. 2024. url: https://docs.px4.io/main/en/advanced_config/static_pressure_buildup (visited on 2025-06-22) (cit. on p. 80).

[226] PX4. *Correction for the static pressure position error*. PX4 Guide. 2024. url: https://docs.px4.io/main/en/advanced_config/tuning_the_ecl_ekf%5C#correction-for-static-pressure-position-error (visited on 2025-06-22) (cit. on p. 80).

[227] Raspberry Pi Ltd. *BCM2711 ARM Peripherals*. 2022. url: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf (visited on 2025-06-15) (cit. on p. 87).

[228] ARM. *CoreLink GIC-400 Generic Interrupt Controller Technical Reference Manual*. 2022. url: https://developer.arm.com/documentation/ddi0471/b/introduction/compliance (visited on 2025-06-15) (cit. on p. 87).

[229] NXP. *i.MX 8M Nano Applications Processor Reference Manual*. 2022. url: https://www.nxp.com/webapp/Download?colCode=IMX8MNRM (visited on 2025-06-15) (cit. on p. 87).

A

Taxonomy

The following appendix contains the taxonomies for the virtualization and UAV concepts.

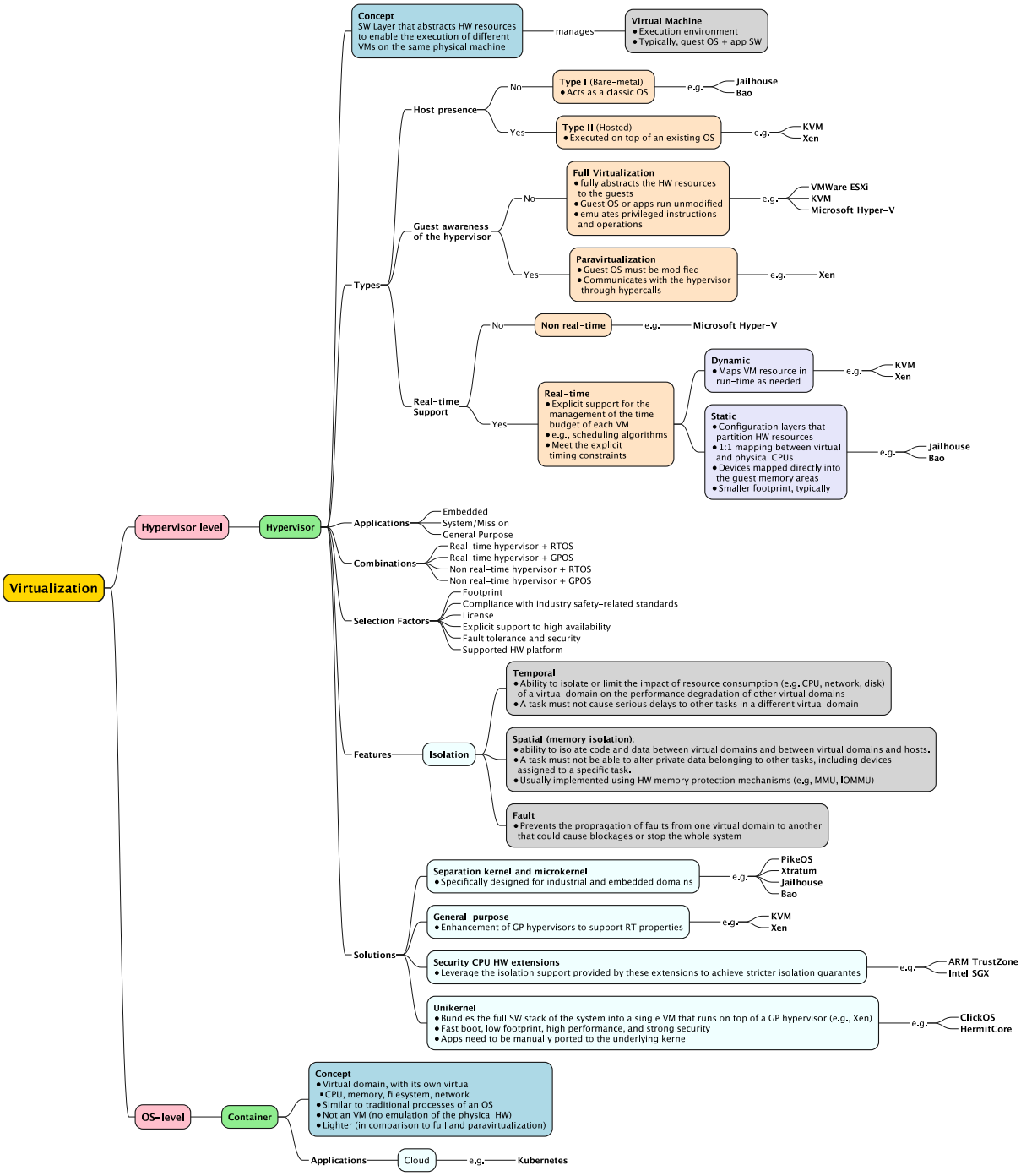


Figure 51: Virtualization mind map

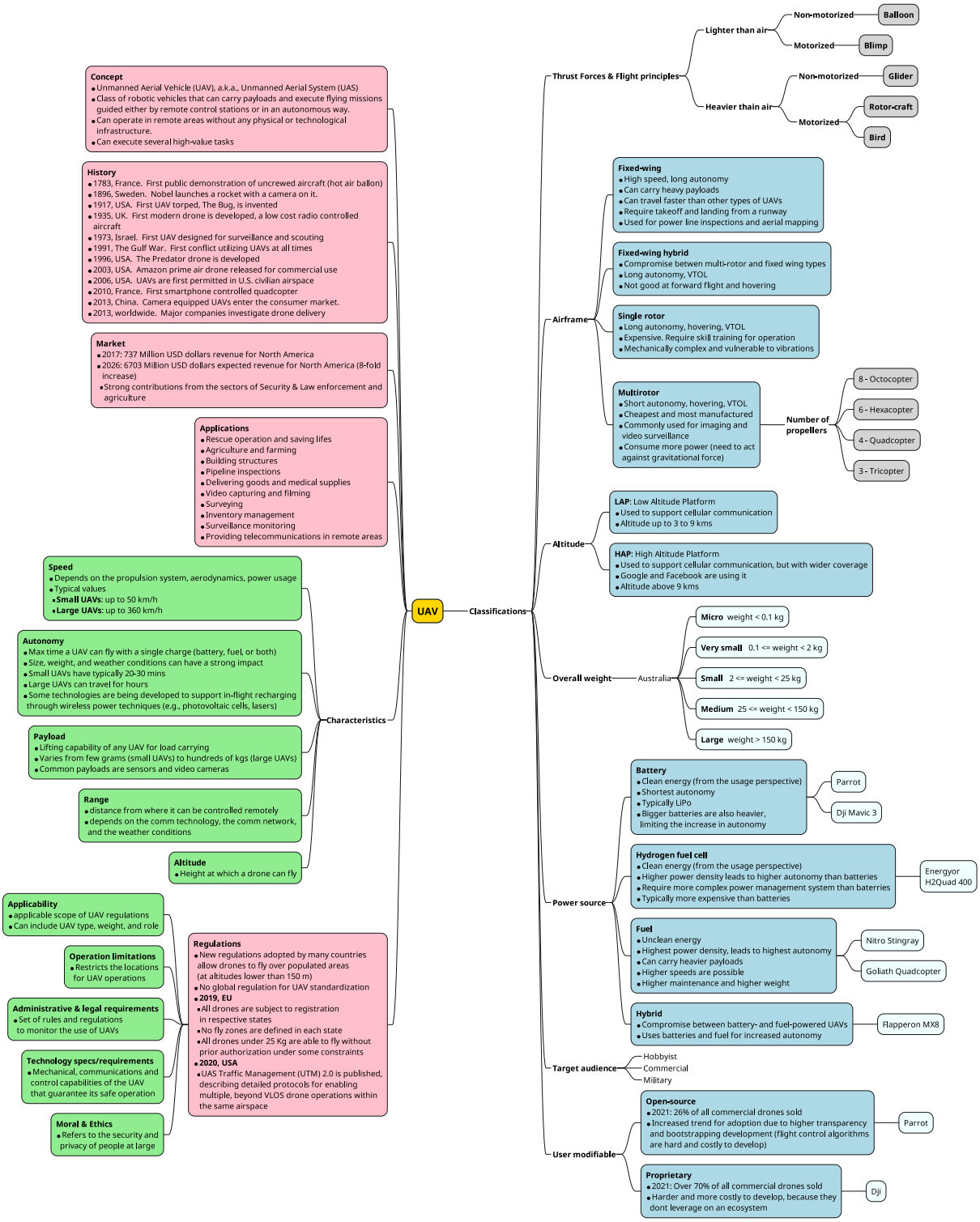


Figure 52: UAV mind map: generic overview

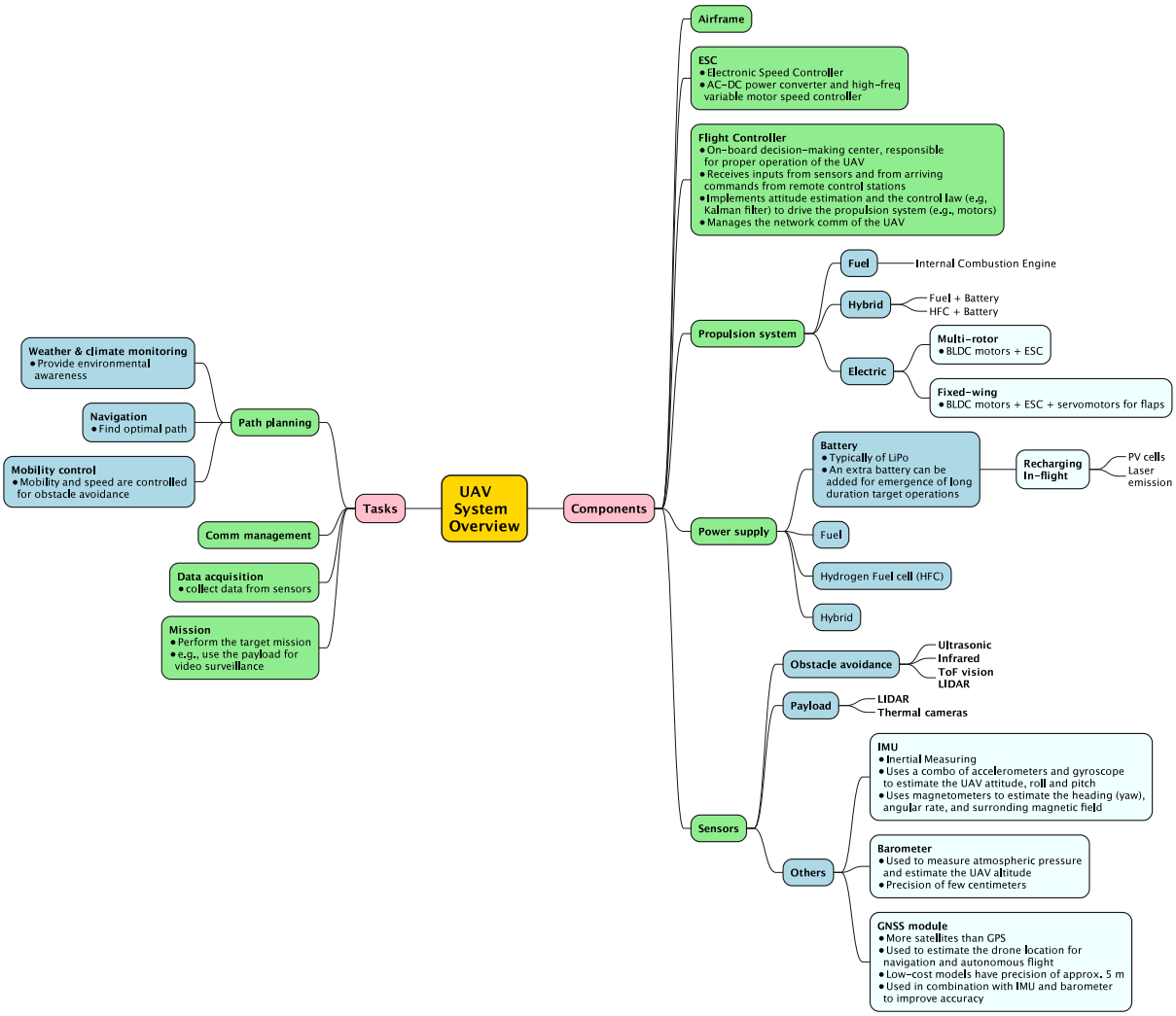


Figure 53: UAV mind map: System overview — Tasks and components

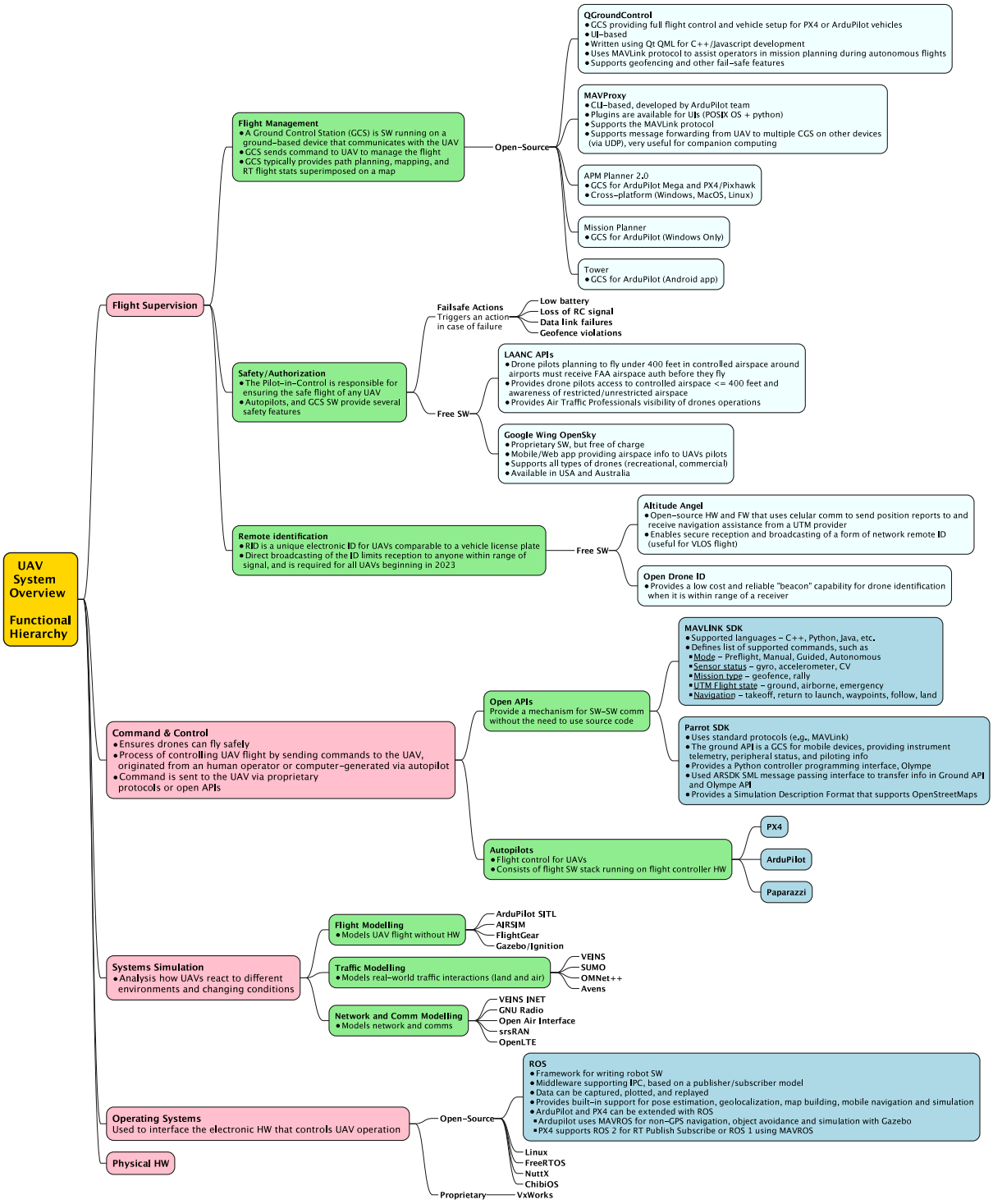


Figure 54: UAV mind map: System overview — Functional Hierarchy

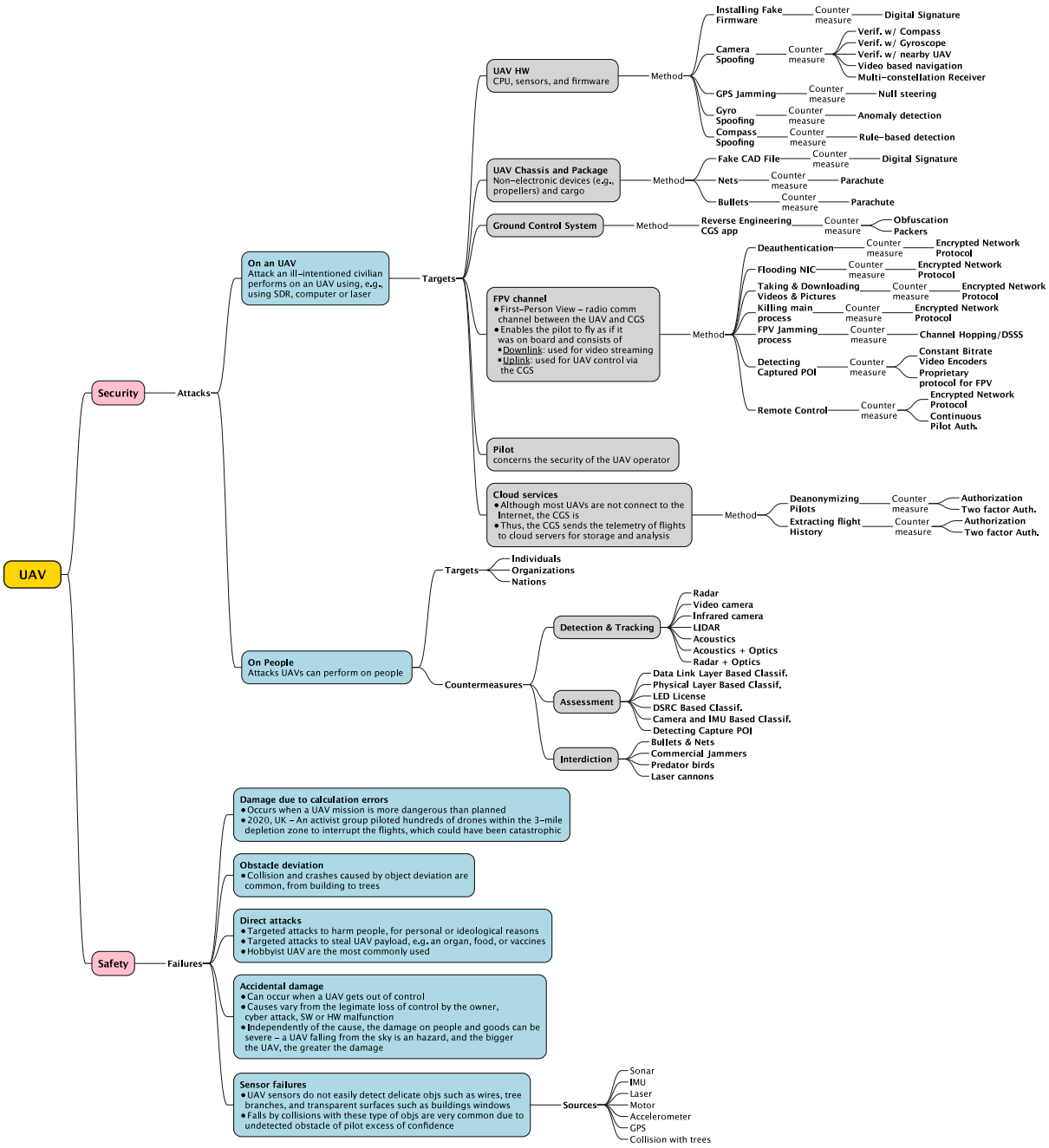


Figure 55: UAV mind map: security and safety

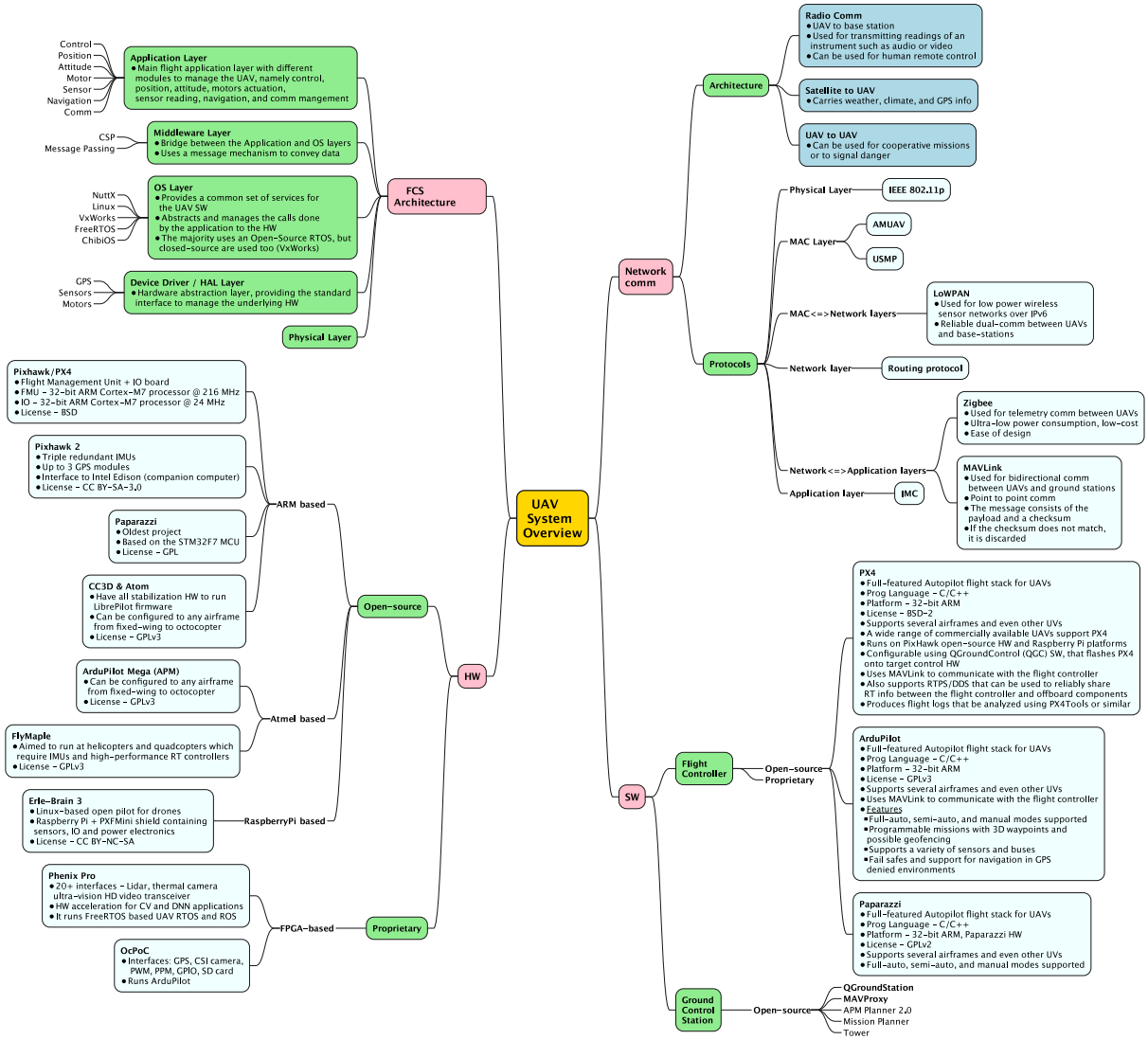


Figure 56: UAV mind map: system overview — Architecture, HW, SW and communications