Analyzing the OODA Loop of an Edge-enabled Autonomous Drone System

Aditya Chanana

CMU-CS-24-158
December 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mahadev Satyanarayanan, Chair
Padmanabhan Pillai

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Copyright © 2024 Aditya Chanana

Keywords: edge computing, drones, mobile networks, latency, bandwidth, edge-native applications, computer vision, machine learning

Dedicated to my family and partner, for their belief in me.

Abstract

The ”Observe, Orient, Decide, Act” (OODA) loop can be applied to encapsulate the agility of cyber-physical or cyber-human systems that depend on continuous iterations of these steps. Systems with faster OODA loops react more quickly to changes in their environment. This work analyzes the OODA loop of the SteelEagle edge-enabled autonomous drone system, which transforms consumer aerial photography drones into fully autonomous UAVs by offloading computation to the edge using a drone payload with cellular connectivity.

We identify bottlenecks and opportunities for optimization, leading to a faster SteelEagle OODA loop and thus improved performance in active vision tasks such as obstacle avoidance and object tracking. This enables the drone to fly safely at higher speeds in crowded spaces, increasing the practicality of SteelEagle drones in applications such as search and rescue and infrastructure inspection. Our findings show that the hardware encoding of the H.264 video stream on the drone makes up about two-thirds of the drone-to-cloudlet latency.

Because of its dependence on offloading, SteelEagle is currently limited in its ability to operate in degraded network conditions. To mitigate these limitations, we analyze the use of onboard computation with SteelEagle by considering a new payload that can run float16-quantized DNNs. We discuss how onboard computational abilities can be combined with offloading to achieve an optimal system based on computation accuracy, energy efficiency, and latency.

Acknowledgments

Thank you to my research advisors, Dr. Mahadev Satyanarayanan and Dr. Padmanabhan Pillai, for their guidance and feedback on this project, and Thomas Eiszler Jr. and Mihir Bala, for their mentorship. A special thanks to Xiangliang Chen for his support and friendship.

I would like to express my gratitude to my parents, Dr. Rashmi Chanana and Sanjay Chanana, who have offered their unwavering love, encouragement, and support at every stage of my life. Thank you, Isabella, for believing in me and being a constant source of motivation. I couldn't have done this without you. I am grateful to my academic advisor, Dr. David A. Eckhardt, for being a reliable source of excellent advice and his encouragement for me to pursue research.

This material is based upon work supported by the U.S. Army Research Office and the U.S. Army Futures Command under Contract No. W519TC-23-C-0003, and by the National Science Foundation under grant number CNS-2106862. The content of the information does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

This work was done in the CMU Living Edge Lab, which is supported by Intel, Arm, Vodafone, Deutsche Telekom, CableLabs, Crown Castle, InterDigital, Seagate, Microsoft, the VMware University Research Fund, IAI, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the view of the above funding sources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Mobile devices, such as smartphones and smartwatches, have a key limitation—a limited battery life which allows them to operate for finite periods before requiring recharging. Resource-intensive applications, such as those involving augmented reality or deep learning inference, exacerbate the situation by accelerating battery depletion. Consequently, a mobile device cannot match the sustained performance and resource capacity of stationary computers, which are free from the requirements of mobility. Mobile devices will always lag behind static devices in computational ability because of their size and weight constraints [1]. Increased computational ability, at the same level of hardware efficiency, demands higher energy consumption. This requires larger batteries to preserve operating time, thereby increasing weight and size—which are undesirable for mobile devices. Table 1.1 shows how mobile devices have lagged behind servers in computational power over a period of 25 years. This limitation has been referred to as the "mobile penalty", the cost for reduction in performance required to adhere to mobility constraints [2].

Unmanned aerial vehicles (UAVs), or drones, which spend most of their energy on flight, are particularly affected by this limitation. Larger batteries required to sustain intensive on-drone computation increases drone weight. Increased drone weight leads to an upwards spiral in weight as larger rotors and more powerful motors are needed to achieve the same amount of lift. This requires even larger batteries which, in turn, may require a more reinforced aircraft structure, and so on. How, then, can we enable mobile devices to do more given the mobility constraints they are subject to? Advancements in hardware efficiency provide one possible way forward, but these advancements are slow and far between. It turns out that there is a way to ”cheat” that enables mobile devices to do more today, leveraging static infrastructure to overcome the mobility penalty, known as offloading.

A significant body of work in the field of mobile computing has utilized cloud or cloudlet offloading techniques to address the inherent resource poverty of mobile devices [5, 6]. Cloudlet offload allows mobile devices to remotely execute computationally expensive tasks on cloudlets that do not need to be light or small. This allows mobile devices to retain their low weight and small size, but possess superior computational abilities.

	Typical Server		Typical Mobile Device	
Year	Processor	Speed	Device	Speed
1997	Intel Pentium II	266 MHz	Palm Pilot	16 MHz
2002	Intel Itanium	1 GHz	Blackberry 5810	133 MHz
2007	Intel Core 2 Quad Q6600	2.4 GHz x 4	Apple iPhone	412 MHz
2012	Intel Xeon E5-2690	3.8 GHz x 8	Samsung Galaxy S3	1.4 GHz x 4
2017	Intel Xeon Platinum 8180	3.8 GHz x 28	Google Pixel 2	2.35 GHz x 4, 1.9 GHz x 4
2022	AMD EPYC 9654	3.7 GHz x 96	Samsung Galaxy S22 Ultra	3 GHz x 1, 2.5 GHz x 3, 1.8 Ghz x 4

*While processor efficiency depends on more factors than processor frequency, such as instructions per cycle, it provides a good first-order approximation.

Table 1.1: The Mobility Penalty Over a Period of 25 Years (Adapted from Chen [3], Flinn [4])

Drones are an excellent category of mobile devices to study because of their wide range of useful applications as well as their design requirements which take mobility constraints to the extreme. We benchmark an autonomous drone system that leverages offloading techniques to overcome the computational limitations of consumer-grade drones. We identify its limitations and evaluate its performance to understand the areas that future work should focus on to enable more capable drones that can have a wider variety of applications.

1.2 Overview

SteelEagle [7] is an autonomous drone system that leverages offloading. Its main tenet is the use of commercial-off-the-shelf (COTS) drones, which are readily available at low cost but typically have minimal to no on-board computational resources. Utilizing cloudlet offload, SteelEagle enables COTS drones to perform much more intelligent tasks than what they were designed for, and exhibit autonomous capabilities typically only found in larger, more expensive drones. SteelEagle focuses on active vision tasks which require the ability to perform live video analytics during flight to determine the next course of action. SteelEagle drones relay their camera video stream over a commercial cellular network to a nearby on-ground edge server, a cloudlet, which runs a computationally expensive pipeline often involving neural networks (Figure 1.1) to perform tasks such as monocular camera depth perception. This processing of the raw drone sensor stream allows obtaining higher level semantic information about the drone's physical environment. In the case of depth perception, we could infer obstacles that present the danger of a collision. The drone can then be sent piloting commands to react and actuate to avoid an impending collision.

An important consideration for SteelEagle is the agility of the resulting system. How quickly can a drone respond to a change in its environment? The end-to-end latency of the entire execution pipeline, including sensing, offloading, inferencing, decision making, and actuation, defines the

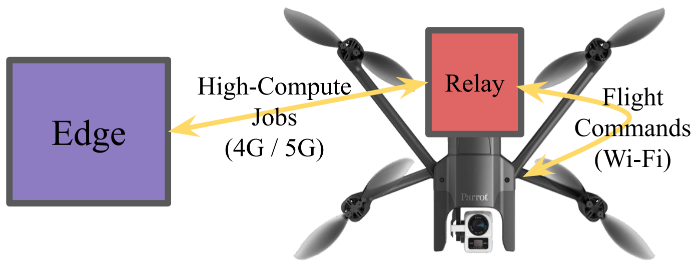


Figure 1.1: Cloudlet Offload in SteelEagle

agility. A high end-to-end latency can severely handicap the drone—it must fly at a higher altitude or at slower speeds to be safe if it takes a long time to identify obstacles and actuate to avoid them. Drone flight at a higher altitude precludes close observation, and slower speeds make missions take longer, limiting the capabilities of the drone. Search and rescue missions in forests, and missions to aid law enforcement operations in dense cities, for instance, must fly at low altitudes while avoiding obstacles in the environment that present a collision risk—trees branches, streetlight poles, utility wires, and buildings—without impacting mission speed. Unless we can achieve high agility, SteelEagle drones will struggle to perform these missions well. Since these missions are one of the most compelling use cases for autonomous drones, benchmarking the end-to-end pipeline to determine latency bottlenecks and identifying opportunities for latency optimization is a very worthy pursuit.

SteelEagle drones currently perform no onboard computation—they are controlled exclusively over the network. While this strategy allows treating consumer-grade non-programmable drones as black boxes, it presents severe limitations. First, SteelEagle drones struggle in areas with unreliable cell service and are completely inoperable in regions without cellular coverage. Second, cloudlet offload imposes an upper bound on drone agility as it adds the cost of a round-trip latency to a cloudlet. As originally envisioned, cloudlets are differentiated from clouds because of their network proximity, which allows application end-to-end response time to be just a few milliseconds. In practice, the usage of commercial cellular networks for offload to the cloudlet increases this latency to tens of milliseconds. This limits the agility of the drone, as the reaction time is at least the round-trip time to the cloudlet.

This thesis performs a comprehensive benchmarking of the SteelEagle system, revealing the current system bottlenecks and identifying opportunities for optimization. In recent years, domain-specific system-on-a-chip devices have become available that provide substantial energy-efficient on-board computational resources through the inclusion of hardware accelerators in the chip design. These chips can decode the video stream generated by the drone and perform analytics using TensorFlow Lite models, and often include 5G and Wi-Fi connectivity. Using such a chip as a payload on SteelEagle drones allows us to continue treating the drone as a black box, but employ new cloudlet offload tactics that result in increasing agility for use cases that can utilize the hardware accelerators, while retaining the generality offered by cloudlet offload. We explore the use of onboard computation to mitigate the limitations of SteelEagle, and explore the resulting impact on drone agility.

1.3 Contributions

The contributions of this thesis include

- A mapping of the SteelEagle pipeline to the OODA loop framework
- A discussion of efforts to optimize SteelEagle that yielded a two-fold improvement in drone-to-cloudlet latency
- An analysis of the SteelEagle pipeline's performance from a latency and bandwidth perspective
- An evaluation of the use of an onboard computation device to improve SteelEagle performance and alleviate its limitations

1.4 Organization

Chapter 2 describes the history of drones and their development over time. We discuss the applications that drones are used in today, as well as the capabilities of today's drones. We also discuss what autonomy means for drones. Chapter 3 provides a detailed background on edge computing and the SteelEagle autonomous drone system. In Chapter 4, we describe our experimental setup for measuring the latency of the SteelEagle system. We include measurements which provided the insight needed to optimize SteelEagle, allowing for a two-fold improvement in SteelEagle's drone-to-cloudlet latency. We also include details on how we performed granular measurements of individual stages in SteelEagle, which allowed us to identify the video encoding of the H.264 video stream on the drone as a bottleneck. In Chapter 5, we explore the use of an onboard computation device called the Modal AI VOXL to augment SteelEagle. We conclude with Chapter 6, presenting concluding remarks and providing a road map for future work.

Chapter 2

Primer on UAVs

This chapter presents background information on unmanned aerial vehicles (UAVs), or drones. Drones have had a long history of evolution, starting as huge gunnery targets in the 1910s to today's small consumer photography drones which are readily available in stores. Beginning with the history of their evolution in Section 2.1, we move on to describe the capabilities found in present-day drones in Section 2.2. We end in Section 2.3 with a description of a framework for characterizing autonomous capabilities in drones.

2.1 History of Drones

Figure 2.1: de Havilland DH82B Queen Bee [8], the first radio-controlled drone

Drones, are aircraft that can be operated remotely without the need for a human pilot on board. While there were many early pilotless aircraft, the first remote controlled aircraft appeared during the First World War, developed by Britain and the US in 1917 [9]. Many of these early drones were used as anti-aircraft gunnery training targets. In the 1930s, the term ”drone” arose inspired by the de Havilland DH82B Queen Bee (Figure 2.1), designed as a low-cost radio-controlled target aircraft, which saw over four hundred units built by 1943. The Queen Bee was the first drone designed with the ability to return to ground safely and be reused [10, 11].

These early drones were fixed-wing aircraft that were used primarily for combat or training. From the 1970s, drones such as the Ryan Model 147 Lightning Bugdrones were developed for use in surveillance missions. These drones carried a camera and could fly for hours at high altitude [12]. The Pioneer UAV, introduced in 1986, saw extensive use in the Gulf War. Today, the General Atomics MQ-1 Predator can fly for over 14 hours performing surveillance missions using an array of sensors, including infrared cameras.

Figure 2.2: DJI Mini 4K

The turn of the century saw the use of small drones in civilian settings, with technology advances making them cheaper to produce. Drones, particularly quadcopters like the DJI Mini 4K (Figure 2.2), started being used for mapping, aerial photography, industrial inspections and security, and precision agriculture [13]. Compared to previous fixed-wing military drones, quadcopters offer superior maneuverability and the ability to hover. They utilize four motor-rotors, with two spinning clockwise and the other two counterclockwise. Variations in motor speed allow for precise hovering and maneuverability. This makes quadcopter drones well-suited for both indoor and outdoor use. Since the 2010s, drones have evolved significantly, with a range of consumer photography and racing drones easily available to consumers, at a wide range of price points. Drones are commonly used for filming and recreation, and have also seen use in logistics, to deliver packages. They have also been put to use in search and rescue [14, 15, 16] and wildlife conservation [17, 18].

The scale of their adoption is immense—The Federal Aviation Administration (FAA) has received registrations for almost 800,000 drones [19]. This figure excludes thousands of hobbyist drones weighing under 250 g that do not require FAA registration.

2.2 Capabilities of Present-day Drones

Today, the set of features available in consumer-grade drones is impressive. The DJI Mini 4K is priced at $299, weighs under 250g, records video at 4K 30 FPS, and has a maximum flight time of about 30 minutes [20]. Most drone's today have the ability to fly semi-autonomously in addition to the ability to be controlled remotely by a human. As shown in Figure 2.3, drones have advanced microprocessors that abstract away the lower-level mechanics of quadcopter drone flight. Components such as the Electronic Speed Controller (ESC) enable precise control of the drone's brushless electric motors. On-board positioning systems such as Global Navigation Satellite System (GNSS) receivers give the drone the ability to navigate between waypoints. Using inputs from on-board inertial measurement unit (IMU) sensors such as gyroscopes, accelerometers, and magnetometers, drones can determine their orientation, acceleration, and rotation, allowing them to adjust their motors in real-time to counter wind and hover in a fixed position. They can also typically takeoff and land autonomously. On-board flight control software such as PX4 or ArduPilot, also known as an ”autopilot,” makes these higher-level functions possible, taking inputs from the IMU and GNSS sensors and outputting control commands to the ESC.

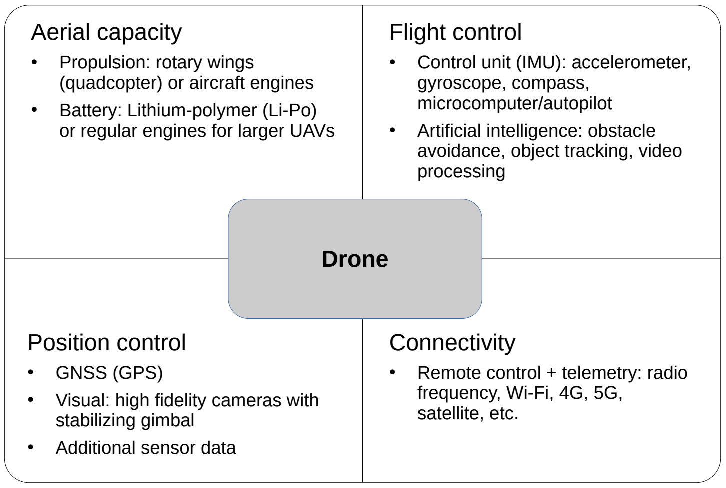

Aerial capacity
• Propulsion: rotary wings (quadcopter) or aircraft engines
• Battery: Lithium-polymer (Li-Po) or regular engines for larger UAVs
Flight control
• Control unit (IMU): accelerometer, gyroscope, compass, microcomputer/autopilot
• Artificial intelligence: obstacle avoidance, object tracking, video processing
Drone
Position control
• GNSS (GPS)
• Visual: high fidelity cameras with stabilizing gimbal
• Additional sensor data
Connectivity
• Remote control + telemetry: radio frequency, Wi-Fi, 4G, 5G, satellite, etc.

Figure 2.3: Components of a drone [13]

Drones are equipped with a variety of cameras, such as monocular and stereo. Monocular cameras capture images and videos from a single point of view, and are common in consumer aerial photography drones. Monocular cameras are often mounted on a gimbal, which stabilizes the drone’s camera during motion and also allows the ability to capture different viewing angles through gimbal motion. Stereoscopic cameras, on the other hand, offer multiple points of view. Having more than one camera allows the use of epipolar geometry and triangulation to determine distance from objects efficiently, and perform obstacle avoidance. The DJI Mini 4 Pro, for instance, has multiple binocular cameras, facing forward, backward, sideways, upwards, and downwards. This allows the Mini 4 Pro to perform omnidirectional obstacle sensing. During manual pilot flight, the drone offers pilot assistance features that stop the drone if it is headed for an imminent collision, and also provide the option to circumvent obstacles automatically altogether. Drones equipped with monocular cameras typically lack these advanced obstacle avoidance systems, but are cheaper and more common.

While consumer-grade drones offer many semi-autonomous features, complex fully-autonomous features are limited to more expensive commercial drones. For example, a consumer drone could be instructed to navigate to a given GPS coordinate or, in some advanced drones, even track an on-ground target. But complex missions, such as patrolling a set of waypoints while searching for an on-ground target, and transitioning to track the target once it is detected, remain out of the reach of consumer drones.

Consumer drones are often controlled over Wi-Fi using a controller or a smartphone app, and typically lack cellular connectivity, limiting their flying range. Most consumer drones using Wi-Fi control require the pilot to be no more than two-thirds of a mile away. Proprietary wireless control systems such as DJI's OcuSync 4 can increase range to over ten miles under ideal conditions.

2.3 Levels of Drone Autonomy

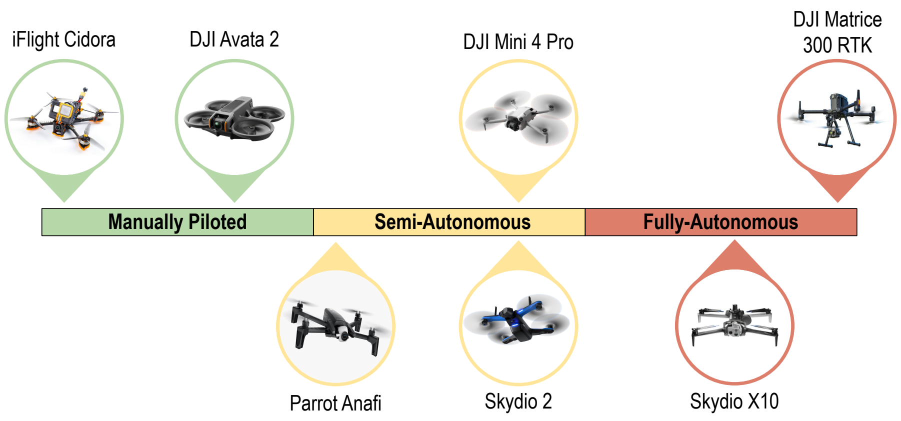


Figure 2.4: Drone Autonomy Levels

Drones exhibit varying levels of autonomy, as shown in Figure 2.4. As we move to the right, the reliance on human drone operators reduces, as the drones feature more autonomous capabilities. On the manual end of the spectrum are drones such as the DJI Avata 2, a first-person view (FPV) drone. FPV drone pilots receive a real-time video stream from the drone's onboard camera and manually control the drone. These drones are optimized for high-speed flight and maneuverability, giving them the ability to navigate through tight obstacles at speed. Manually piloting a drone requires skill and constant human pilot input, which reduces the usefulness of these drones.

Drones such as the Parrot Anafi include semi-autonomous features such as the ability to hover and navigate between waypoints. These drones often include a return-to-home (RTH) function which automatically returns the drone to its takeoff location if the battery is low, or if the drone loses its connection to the pilot's controller. Advanced semi-autonomous drones such as the DJI Mini 4 Pro can also follow a human or vehicle target. These semi-autonomous features reduce the level of skill required for piloting and the reliance on the human pilot, but still require the pilot to specify the next course of action to fulfill higher-level mission objectives.

On the right end of the spectrum are fully-autonomous drones, which execute a pre-programmed mission without a human pilot while adapting to operational and environmental conditions during flight. The DJI Matrice 300 RTK, for instance, can inspect pre-specified objects of interest, such as power transmission towers, without any human input.

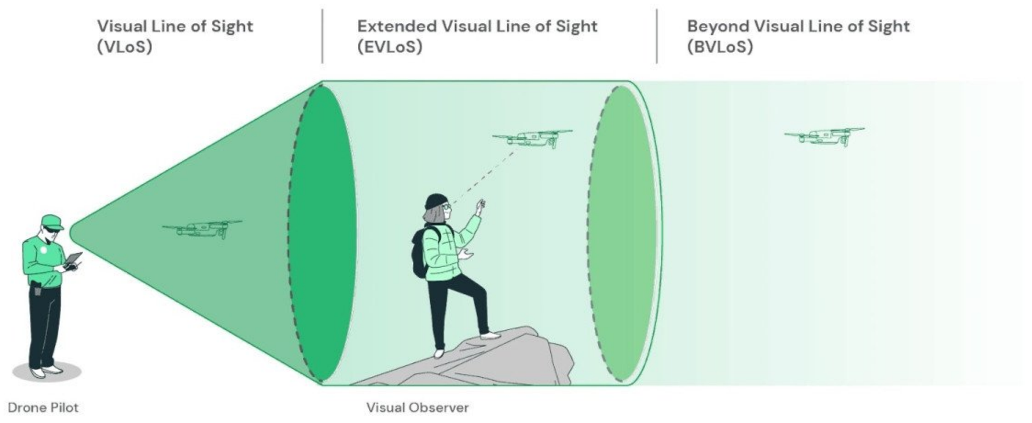


Figure 2.5: Drone Operation Line-of-Sight Requirements [21]

Regulatory hurdles, however, impact the versatility of fully-autonomous drones. FAA regulations require that drone operators keep drones within sight during flight [22]. For FPV drones, it is sufficient to have a visual observer that always has the drone in sight. These two modes of operation correspond to “Visual Line of Sight” (VLoS) and “Extended Visual Line of Sight” (EVLoS) in Figure 2.5. While a number of use cases can be covered under these modes of operation, the holy grail for drones lies in “Beyond Visual Line of Sight” (BVLoS) operation. BVLoS operation allows drones to fly a much wider range of missions. Missions involve package delivery requires drones to fly to a far-off destination, improving speed and reducing the logistical cost compared to human delivery via road. Requiring a human observer to maintain sight of the drone at all times defeats the purpose of this application.

Chapter 3

Background and Related Work

This chapter presents background information on the field of edge computing (Section 3.1), the SteelEagle autonomous drone system (Section 3.2), and the OODA loop framework (Section 3.3).

3.1 Edge Computing

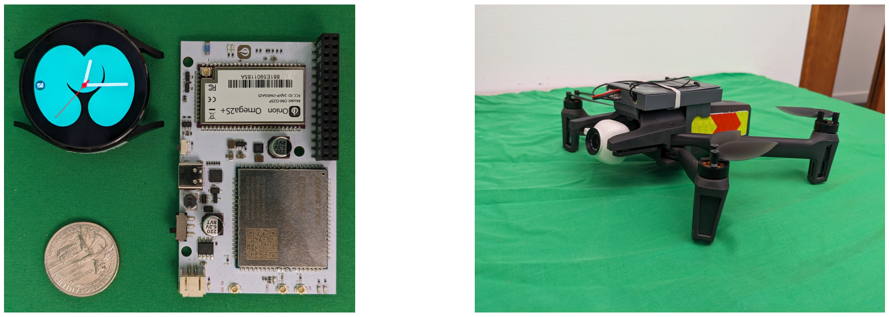
There has been a huge increase in the number of mobile and Internet of Things (IoT) devices in recent years, driven by advancements in sensor, networking, and processing technologies. These innovations have made devices smaller, more affordable, and more versatile, enabling their integration into nearly every aspect of modern life. As explained in Section 1.2, despite the innovations, these mobile devices remain resource-poor relative to static resources. A wearable device like a smartwatch, for example, has limited battery life, which constrains its ability to sustain prolonged conversation with an onboard AI virtual assistant. In 1997, Noble et al extended an adaptive application-aware framework called Odyssey, which provides remote data access to mobile clients, to perform speech recognition on a resource-constrained mobile device by offloading compute to a remote server [23]. This offloading technique allows a mobile device to circumvent its resource limitations.

But a key question remains—where to offload? The cloud is an appealing choice today because of its on-demand and scalable nature. However, cloud computing resources are consolidated into datacenters at a small number of geographic locations to leverage economies of scale, increasing the distance between end users and the cloud, thus increasing network latency. This can be a deal breaker for latency-sensitive interactive applications such as augmented reality. An empirical study conducted with 2,504 Amazon EC2 clients found that more than 60% of the clients experienced latencies higher than 40 ms [24]. However, immersive augmented reality on a wearable device has a latency bound of 16 milliseconds [25]. To attain crisp interactive application response, Satyanarayanan et al introduced the novel paradigm of edge computing [6].

Edge computing is a paradigm in which substantial computing and storage resources—referred to as ”cloudlets”—are situated at the edge of the Internet in close proximity to mobile devices, sensors, end users, and Internet of Things (IoT) devices. This proximity offers several key advantages [26]:

- It allows offloading to truly shine, enabled by the low latency, high bandwidth, and low jitter links to the cloudlets
- It reduces the bandwidth demand that IoT devices like video cameras place on the cloud, thus increasing scalability [27]
- It can enforce a user's privacy policies specified for their IoT sensors before the data is sent to the cloud
- Cloudlets help mitigate cloud outages by serving as a fallback service

3.2 SteelEagle Autonomous Drone System



(a) The Samsung Galaxy Watch and the Onion Omega 2 LTE

(b) Onion Omega 2 LTE mounted on Parrot Anafi

Figure 3.1: The Hardware Used in SteelEagle (from Bala et al [7])

Autonomous drones perform tasks such as navigating between waypoints and tracking moving objects without the need for a human pilot. However, autonomous drones today are large and expensive. Lightweight drones are more appealing, as they present a smaller public safety hazard and thus face fewer regulations. The FAA, for instance, has pre-authorized flights over people and vehicles by drones weighing less than 250 g. Cheaper autonomous drones will help accelerate their uptake in scenarios that stand to benefit the most from their abilities. Search and rescue operations [16], as well as wildlife conservation efforts that involve monitoring wildlife populations, would benefit immensely from the ability of drones to cover large areas quickly. The potential for good increases exponentially with a swarm of drones working cooperatively [14]. In rural areas with limited law enforcement resources, for instance, a swarm of drones could quickly scan large swathes of land looking for an abducted child.

Over time, autonomous drones will become cheaper and lighter as new hardware designs are developed that are more energy efficient and mass production lowers costs. Until then, leveraging edge computing to add autonomous features to lighter consumer-grade drones at a much lower price tag is a very appealing proposition. This is the software-hardware co-evolution path that Satyanarayanan et al outlined, presenting offloading as a way to ”cheat” until ASIC designs are available [28].

Bala et al [7] have demonstrated that even drones with a monocular camera can perform tasks such as object tracking and depth inference reasonably well. In their setup, Bala et al initially used a Samsung Galaxy smartwatch as a communications relay, mounted on top of a Parrot Anafi drone, for a total takeoff weight of about 360 g. The Samsung Galaxy smartwatch is appealing because it is an enclosed system, including a battery and an enclosure protected from the elements. It also has the ability to run Android applications onboard. However, the constant LTE transmission on the watch caused it to hit its thermal limits, set so that the watch can be safely worn on the human wrist, and shut down. Figure 3.2 shows the increase in temperature of the watch for a frame rate of 0.7 and 2 FPS.

As a result, the Samsung Galaxy smartwatch could sustain a very low frame rate.

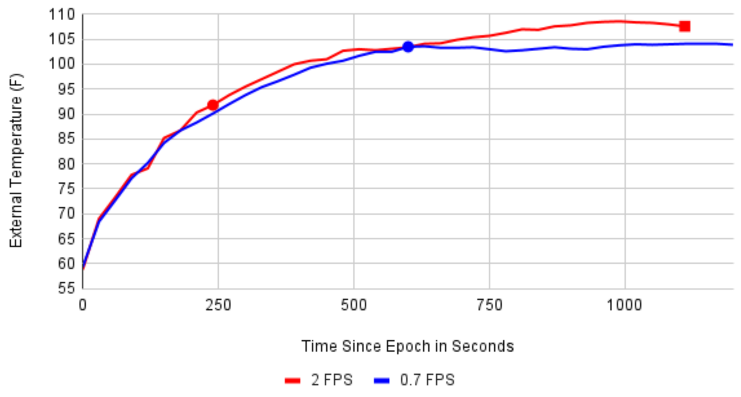


Figure 3.2: The Samsung Galaxy Payload Thermal Curve (from Bala et al [7])

As an alternative, Bala et al used the Onion Omega 2 system-on-a-chip device [29]. While the Omega 2 does not have the thermal limitations present in the Galaxy smartwatch, it has very weak computational capabilities. Intended for use as an IoT module, the Omega 2 runs the 580MHz MIPS 24KEc CPU and has only 16 MB of flash storage. In the SteelEagle setup, a VPN tunnel is established between the cloudlet and the Omega 2, and the drone is connected to the Omega 2 over Wi-Fi. This allows communication with the drone that is connected to the Omega 2 over 4G LTE. The Omega 2 routes packets between its Wi-Fi and LTE network interfaces, acting as a communications relay.

3.3 OODA Loop Framework

The ”Observe, Orient, Decide, Act” (OODA) loop framework [30] devised by military strategist John R. Boyd provides a framework to structure our investigation into the end-to-end latency of SteelEagle. It consists of four stages:

- **Observe.** Involves obtaining new information about the environment
- **Orient.** Analysis and interpretation of the obtained information
- **Decide.** Choosing a course of action

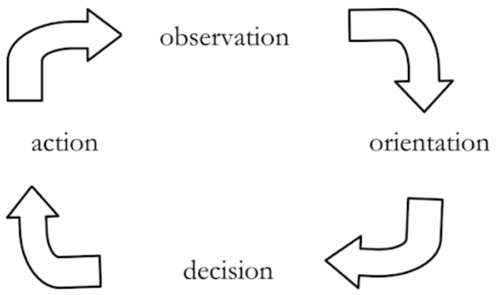


Figure 3.3: The OODA Loop

- **Act.** Execution of the chosen action

According to Boyd, decision-making happens in a continuous iteration of these steps. Boyd attributed the faster OODA loop of U.S. pilots flying F-86s to the bubble-shaped canopy offering better visibility and hydraulic controls that allowed for easier switching between maneuvers, as the reason the slower F-86s fared better than the North Korean MiG-15s during the Korean War [31]. In the context of autonomous drones, a drone with a tighter OODA loop corresponds to a more agile drone. It means that the drone is able to react faster to changes in its environment.

Instead of measuring just the overall system latency, performing a break down of the latency across the OODA steps provides more insight into system latency bottlenecks.

Chapter 4

Optimizing SteelEagle Performance

As explained in Section 1.2, the performance of the SteelEagle system determines its versatility. A drone that is slow to react will be unable to effectively navigate obstacle-dense environments. The chances of the drone losing track of a fast on-ground target that is moving erratically increases substantially if the drone is slow in keeping the target centered in its view. Given the importance of performance, this chapter details how performance is defined for SteelEagle, how it is measured, and work done to identify performance bottlenecks and optimize the system.

Section 4.1 describes the factors that determine the performance of SteelEagle. Section 4.2 includes details about the technique used to measure end-to-end latency. Section 4.3 describes initial efforts to optimize the system that yielded a more than two-fold reduction in drone-to-cloudlet latency, discovering negative scale-out attributes of FFmpeg, a video decoding library. Section 4.4 describes subsequent efforts to establish a more systematic approach to profiling SteelEagle, which found the encoding of the RTSP video stream generated by the drone to be the biggest contributor to latency.

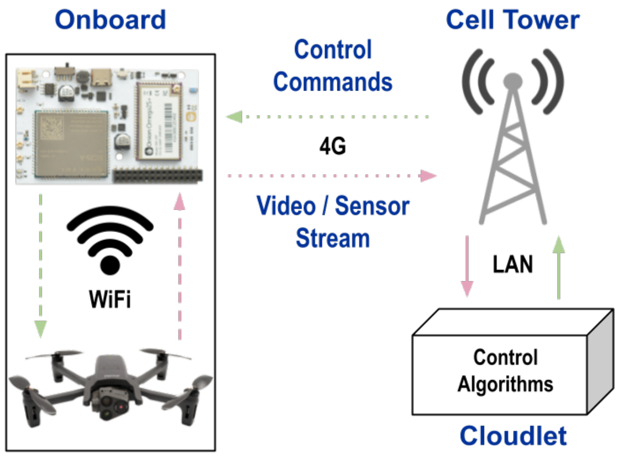


Figure 4.1: SteelEagle Edge Offload Pipeline

4.1 How is Performance Defined in SteelEagle?

The performance of SteelEagle is determined by the end-to-end latency and throughput of the system. Currently, SteelEagle drones function as thin clients, with the use of a communications relay to make up for the lack of cellular connectivity on commercial-off-the-shelf (COTS) drones. The sensor stream from the drone is forwarded to the communications relay over Wi-Fi, which in turn forwards it to the cloudlet over 4G LTE. After performing inference on the received data, the cloudlet sends back piloting commands to the drone, via a hop through the communications relay (Figure 4.1). As a result, the end-to-end performance is determined by several components of the pipeline:

(a) **On-drone sensing.** Capture of images by the drone's camera

(b) **On-drone pre-processing.** Pre-processing of sensor data. For example, generation of video stream from raw camera images

(c) **Transmission to cloudlet.** Offloading to cloudlet to perform resource-intensive computation

(d) **Cloudlet processing.** Inferencing, planning, and decision making based on sensor data

(e) **Transmission to drone.** Transmission of actuation command back to the drone

(f) **Drone post-processing.** Drone processing to interpret command

(g) **Drone actuation.** Electromechanical actuation of drone's motors

Each of these components has a latency associated with it, and a throughput that it is capable of. The performance of components (a), (b), (f), and (g) is fixed for a given drone. Factors such as the drone camera's sensor readout time and shutter speed affect the frame capture latency, and thus determine the performance of component (a). Component (b) consists of any processing of the sensor data before it is transmitted from the drone. The generation of an H.264 stream, for instance, adds latency since the compression process is computationally intensive, involving analysis of deltas between successive frames.

For components (c) and (e), performance is determined by the cellular network used for communication between the relay and the cloudlet. While there is variance associated with the performance of these components, factors such as the number of network hops, distance between the relay to the cloudlet, and the network signal strength available to the relay largely determine the 99th percentile value of their latency and throughput over a longer period of time. The performance of component (d) can vary largely based on whether decoding of the drone stream data is required and type of inferencing that is performed. A DNN with a complex architecture, for instance, can take much longer for inference than a traditional computer vision approach. The same inference can also take less time on a more powerful cloudlet.

Figure 4.2 shows how these components can be mapped to the various stages of the OODA loop framework, defined in Section 3.3. The "Observe" phase includes components (a), (b), and (c). Components (d) maps to the "Orient" and "Decide" phases. Finally, components (e), (f), and (g) correspond to the "Act" phase.

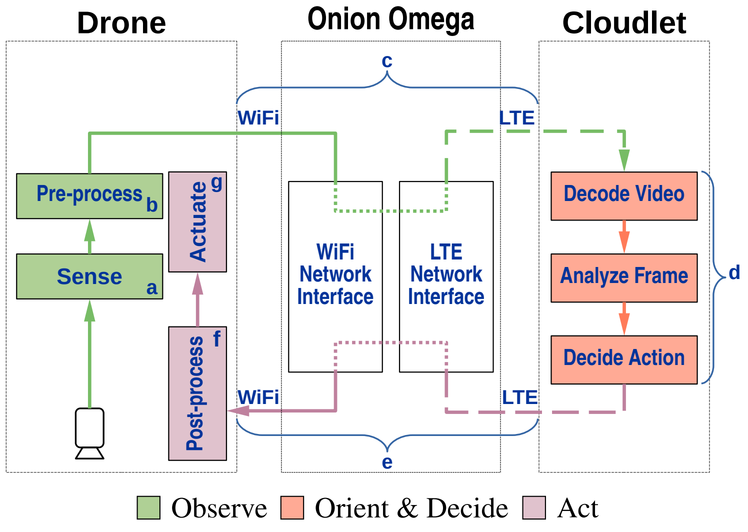


Figure 4.2: Mapping the SteelEagle pipeline to the OODA loop

4.2 Measuring the Performance of the SteelEagle Pipeline

Benchmarking the performance of SteelEagle is challenging since it uses COTS drones that restrict modification of its onboard software. The inability to add software instrumentation on the drone makes it difficult to determine when a given frame was transmitted from the drone. To circumvent this restriction, Bala et al adapted the technique George et al used for measuring motion-to-photon latency in augmented reality [32].

4.2.1 Setup

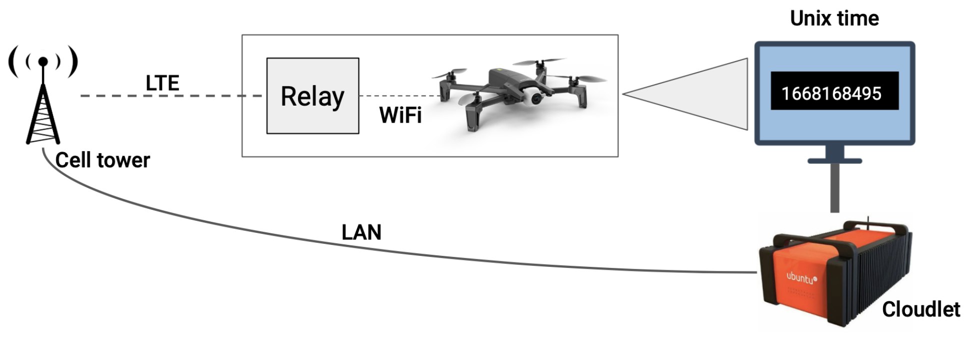


Figure 4.3: Technique for measuring end-to-end latency

As shown in Figure 4.3, the drone is kept stationary in a lab setting with its camera pointed at a display connected to the cloudlet showing the current Unix timestamp at millisecond granularity. The drone captures images containing the timestamp displayed and transmits them to the cloudlet through the SteelEagle pipeline. The cloudlet records the timestamp at which it receives each frame, storing the frame along with this timestamp. These saved frames are then post-processed to compute the difference between the timestamps shown in each frame and the timestamps at which they were received at the cloudlet. This difference corresponds to the latency of components (a) through (d).

4.2.2 Sources of Error

While the display connected to the cloudlet shows the current timestamp at millisecond granularity, the measurements obtained have a much higher experimental error. We identify the following sources of error:

Timestamp granularity. Since the timestamp is displayed at a millisecond granularity, and is updated at most once every millisecond, an error of about a millisecond is inevitable.

Printing to the console. The technique described in Section 4.2 utilizes a script that continuously prints the Unix timestamp to the console. We determine this to contribute to about 0.2 ms of latency by profiling code that prints to the console.

Refresh rate of the display. The refresh rate of the display determines the rate at which its contents are updated. Most displays available today have a refresh rate of at least 60 Hz. This means that the timestamp displayed can be stale by almost 17 ms for a 60 Hz monitor.

4.3 Optimizing SteelEagle Video Decoding

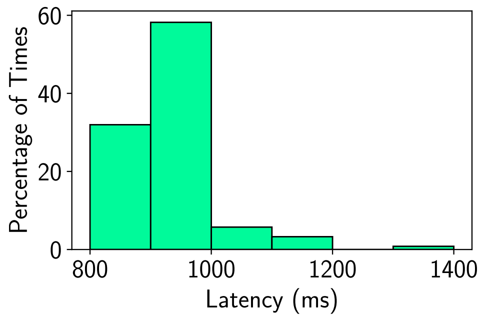


Figure 4.4: Original SteelEagle Latency [7]

The starting point of our investigation into the performance of the SteelEagle system is the mean drone-to-cloudlet latency of 933 ms reported by Bala et al in their benchmarking of the SteelEagle system (Figure 4.4). This includes the “Observe” and “Orient & Decide” components of the OODA pipeline, components (a) through (d) in Figure 4.2. The latency was obtained using the Parrot Anafi drone (Figure 4.5a), which transmits a 720p H.264 RTSP stream at 30 FPS over UDP from its monocular camera. The Anafi uses a slice encoding and intra-refresh scheme that disperses keyframe slices across multiple network packets [33]. To reduce network bandwidth requirements, it generates an H.264 compressed video stream using onboard hardware from the raw frames that it obtains from its camera. Consequently, decoding of this H.264 stream is required to obtain individual video frames. The setup uses the Onion Omega 2 LTE (Figure 4.5b) as a network gateway to allow the Anafi to reach the cloudlet over LTE, since the Parrot Anafi lacks cellular connectivity.

The technique to measure latency described in Section 4.2 does not provide a granular latency breakup. As a result, it is not clear from the measurements where the bottleneck resides. As an initial approach, we measure the impact on latency when the cloudlet location and the video decoding library used are modified. If the latency is significantly reduced by changing a factor, we can conclude that the bottleneck resides in the corresponding component.

4.3.1 Experimentation Parameters

Our first experiment measures the impact on latency when the location of the backend and the video decoding library used is changed. Table 4.1 summarizes the different parameters tested along these dimensions.

Table 4.1: Latency Pipeline Experimentation Parameters

Dimension	**Parameters**
Backend Location	Cloudlet, AWS Small, AWS Big
Video Decoding Library	FFmpeg, PDrAW

We now describe the details of these parameters:

- **Location of backend.** Three backend locations with different specifications, each offering varying levels of performance, are considered to evaluate how hardware scale-up affects system latency. The SteelEagle backend is either set up on a bare-metal server on CMU's campus, labeled "Cloudlet", or on an EC2 VM in AWS East (Virginia). Two types of EC2 instances are used, "AWS Small" and "AWS Big". The CMU Cloudlet has two Intel® Xeon® E5-2699 v3 CPUs clocked at 2.30 GHz for a total of 72 vCPUs, 128GB main memory, and an NVIDIA® GeForce® GTX 1080 Ti GPU. AWS denotes a `g4dn.xlarge` EC2 instance which has 4 vCPUs, 16GB of memory, and an NVIDIA T4 GPU. AWS Big is a

(a) Parrot Anafi

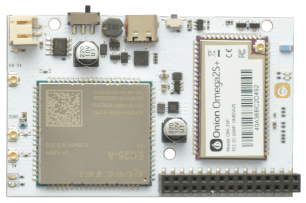

(b) Onion Omega

Figure 4.5: The commercial-off-the-shelf components used in the SteelEagle system

Table 4.2: Specifications of Backends Used in Experiment

Backend	**vCPUs**	**Memory**	**GPU**
AWS Small	4	16 GB	NVIDIA® T4
AWS Big	64	256 GB	NVIDIA® T4
Cloudlet	72	128 GB	NVIDIA® GeForce® GTX 1080 Ti

`g4dn.16xlarge` EC2 instance which has 64 vCPUs, 256GB main memory, and also an NVIDIA T4 GPU. Table 4.2 summarizes these specifications.

- **Video decoding library.** Two video decoding libraries are considered: FFmpeg and PDrAW (pronounced "pedro"). FFmpeg [34] is an open-source project that offers libraries for video encoding/decoding and multiplexing/demultiplexing. FFmpeg is known for forming a core part of the VLC media player. We interact with FFmpeg through OpenCV [35], which offers the ability to use FFmpeg as a backend for its video capture APIs. PDrAW [36] is a part of Parrot's Ground SDK software. Similar to FFmpeg, it includes multiplexing/demultiplexing abilities and can read from the RTSP stream generated by the Parrot Anafi drone. Unlike FFmpeg, however, PDrAW is intended as a video player for RTSP and MP4 videos and lacks general-purpose encoding and decoding abilities.

4.3.2 Experimental Results

Table 4.3 summarizes the results obtained. For each backend location, the latency is measured using the two different video decoding libraries. The mean latency of 933 ms obtained by Bala et al used FFmpeg with the "Cloudlet" backend. We measured a lower mean of 888 ms using these parameters because our experiments do not include inference time on the cloudlet. Inference time can vary across different machine learning models, depending on the kind of pre-preprocessing performed and architectural details such as the number of hidden layers used in deep neural networks. Not including inference time, then, allows us to focus on the contribution of other components.

When using FFmpeg, moving the backend from the Cloudlet to AWS Small leads to an extreme drop in latency, with a reduction in p95 latency from 917 ms to 600 ms (Table 4.3). This amounts to a 1.5$\times$ speedup, with an improvement of over 300 ms. Figure 4.6 shows an interesting trend. The latency for AWS Small, which has the weakest computation power, is the lowest when using FFmpeg and the highest when using PDrAW. This discrepancy is unexpected, as we anticipate AWS Small to perform consistently across both setups.

This led to the hypothesis that FFmpeg's performance degrades with increased parallelism, as it struggles to scale effectively with higher CPU counts. To test this hypothesis, measurements were obtained by varying the number of threads used for FFmpeg from one to six (see Figure 4.7). The OpenCV option `CAP_PROP_N_THREADS` was used to set the number of threads used for the FFmpeg backend. The results show that the latency increases as we increase the number of threads, suggesting that FFmpeg suffers from negative-scale out attributes. Adding more threads

Table 4.3: Drone-to-Cloudlet SteelEagle Latency (in ms)

Configuration	Average	Median	p95	Min	Max
Cloudlet					
FFmpeg	888 ± 30	887	917	838	1,030
PDrAW	380 ± 15	379	402	344	420
AWS Small					
FFmpeg	536 ± 75	521	600	473	936
PDrAW	429 ± 21	427	467	387	475
AWS Big					
FFmpeg	870 ± 20	868	900	837	925
PDrAW	367 ± 20	365	392	327	416

50 samples obtained for each configuration.

to FFmpeg hurts latency.

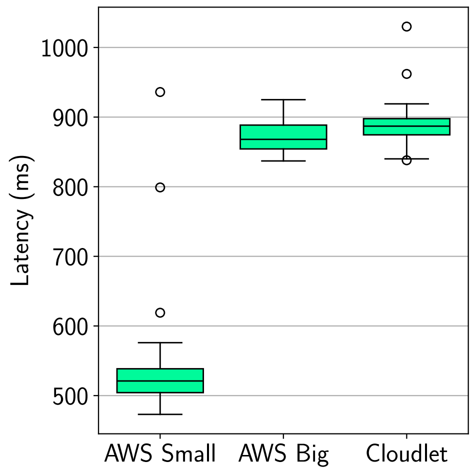
Across all backend locations, we achieve the lowest latency by replacing FFmpeg with PDrAW for video decoding. For CMU Cloudlet, latency reduced from 917 ms to 402 ms by switching to PDrAW, a speedup of almost $2.3\times$.

Figure 4.8 shows the distribution of latency obtained using our optimized configuration with PDrAW and the CMU cloudlet, obtaining a mean of 380 ms and a 95th percentile latency of 402 ms.

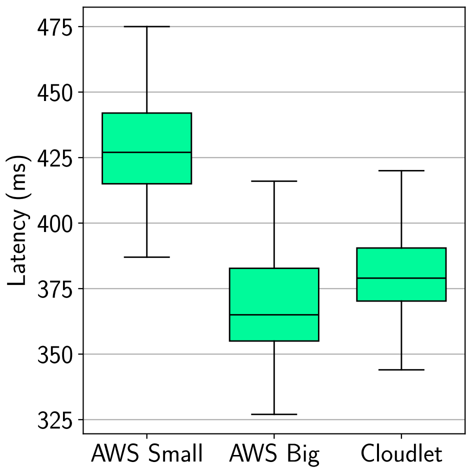
4.4 Structured Benchmarking Using the OODA Loop Framework

Section 4.3 covered an initial approach to benchmarking and optimizing the SteelEagle pipeline, which only considered end-to-end drone-to-cloudlet latency. It did not provide insight into the latency breakup of the pipeline and where the new performance bottleneck resides. Several aspects of the pipeline hinder obtaining this breakup.

The drone sensing and pre-processing, components (a) and (b) from Figure 4.2, cannot be measured individually because the drone is a COTS product that runs closed source software which does not allow the ability to insert software instrumentation. Similarly, drone post-processing and actuation, components (f) and (g), must be measured together. Even measuring components (a) and (b) in aggregate is challenging because of the way the Onion Omega is used. Acting as a network gateway for the drone, the Onion Omega routes network packets between the drone and the cloudlet. This is done by establishing a VPN tunnel between the cloudlet and the Onion Omega using WireGuard. The Onion Omega, in turn, connects to the drone's Wi-Fi hotspot, allowing the cloudlet to reach the drone. There is no userspace application on the Onion Omega where instrumentation can be inserted, the packet routing is done by the kernel.



(a) FFmpeg



(b) PDrAW

Each box extends from the first quartile (Q_1) to the third quartile (Q_3), with a line at the median. Whiskers extend from the box to the farthest data point lying within 1.5x the inter-quartile range ($IQR = Q_3 - Q_1$) from the box. Circles represent outliers. 50 samples obtained for each configuration.

Figure 4.6: Latency Across Backend Locations Using Different Video Decoding Libraries

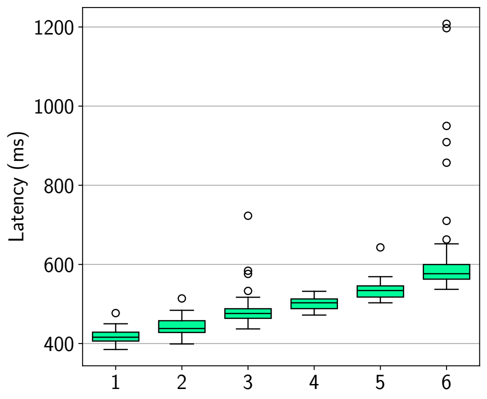


Figure 4.7: Latency vs. Number of FFmpeg Threads

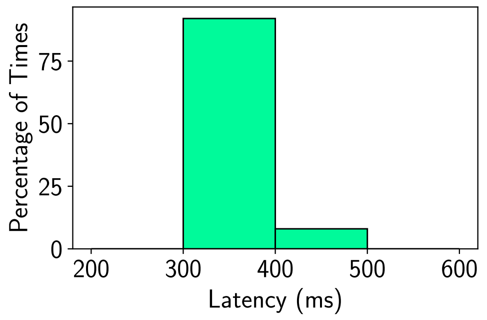
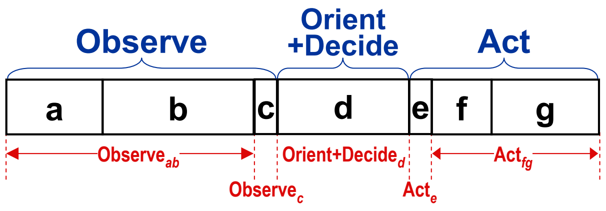


Figure 4.8: Optimized SteelEagle Latency using PDrAW for Video Decoding

Another challenge involves measuring the video decoding time on the cloudlet, part of component (d). To measure decoding time, we need to measure the time taken to receive the full decoded frame after the first network packet corresponding to the frame arrived. A mapping from network packets to decoded frames must be obtained to measure this. Figure 4.9 shows the parts of the pipeline that can be measured in red: Oberve$_{ab}$, Observe$_c$, Orient+Decide$_d$, Act$_e$, and Act$_{fg}$.



Only items in red above can be measured.

a = on-drone sensing	e = transmission to drone
b = on-drone pre-processing	f = on-drone post-processing
c = transmission to cloudlet	g = drone actuation
d = processing on cloudlet	

Figure 4.9: Measurable Components of the OODA Loop

Our approach involves using the `tcpdump` program to capture network packets on the Onion Omega and the cloudlet. As shown in Figure 4.10, we record four timestamps, t_1 through t_4. t_1 corresponds to the Unix timestamp contained in the frame that the drone's camera sees. t_2 is the timestamp of the first network packet transmitted by the Onion Omega corresponding to this frame. $t_2 - t_1$, then, corresponds to Observe$_{ab}$. t_3 is the time the first network packet corresponding to this frame arrives at the cloudlet. $t_3 - t_2$ is the network latency Observe$_c$. Finally, t_4 is the timestamp recorded when this frame is decoded on the cloudlet. $t_4 - t_3$ corresponds to the decoding time for this frame, which is a part of Orient+Decide$_d$.

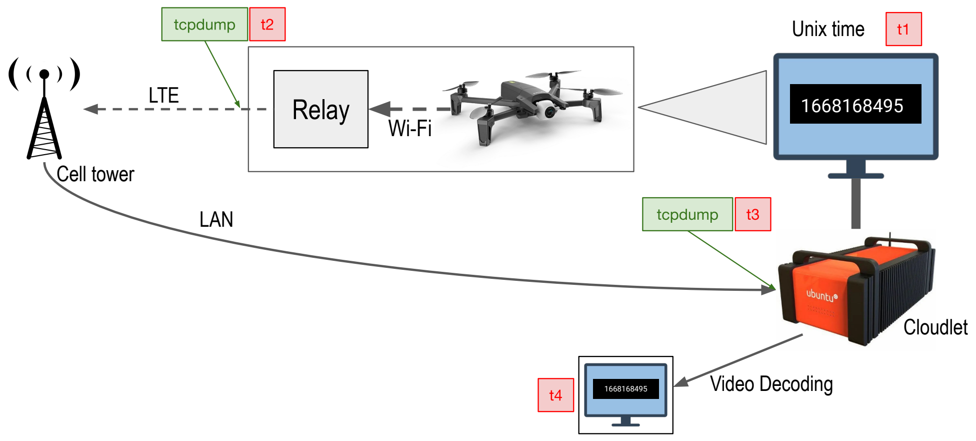


Figure 4.10: Obtaining Observe and Orient Latency using `tcpdump`

4.4.1 Obtaining a Correspondence between Network Packets and Frames

To establish a correspondence between network packets and frames that is required in this approach, we exploit the structure of the slice-encoded H.264 video stream transmitted by the drone[1]. Video encoding formats typically includes two kinds of frames: intra-coded and predicted. Intra-coded frames, also known as I-frames, can be decoded independently since they are self-contained. Predicted frames, or P-frames, are decoded relative to a previous I-frame since they contain deltas calculated using motion compensation techniques. In H.264 video encoding I-frames are sent at a fixed periodicity, with a single I-frame sent in a group of pictures (GOP), a unit defined as a fixed number of successive frames. In the presence of network degradation, packet loss can cause a lost or incomplete I-frame to affect the decoding of subsequent frames for the entire GOP.

To mitigate this, the Anafi drone performs video encoding at the slice level, such that each 720p (1280x720) frame is divided into 45 slices, where each slice is a 1280 pixels wide and 16 pixels high row of the frame. I-frames and P-frames become I-slice and P-slices in this slice-level encoding. The video encoding contains periodic I-slices to prevent error propagation, sending five I-slices every three frames so that all slices are refreshed with an I-slice every 27 frames (Figure 4.11).

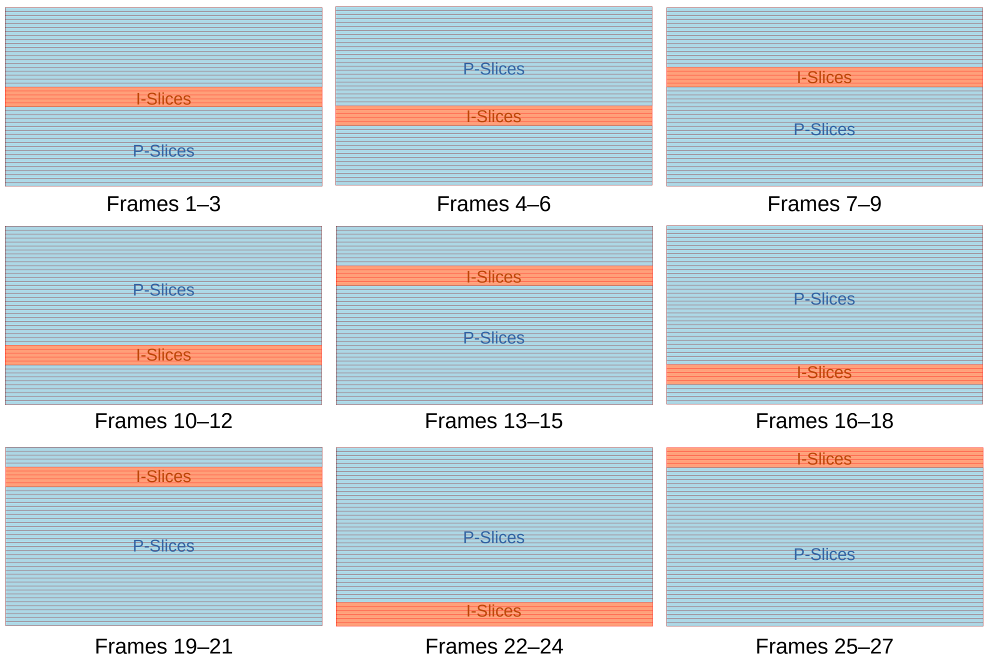


Figure 4.11: The Anafi's Slice-Level Video Encoding Refresh Wave

Our approach temporarily filters network packets on the cloudlet originating from the drone using the Linux kernel's `netfilter` framework. During this time, the video decoder will miss the I-slices that it needs to decode P-slices. After resuming packet flow, the video decoder will be temporarily unable to correctly decode P-slices for which it missed an I-slice, leading to degraded

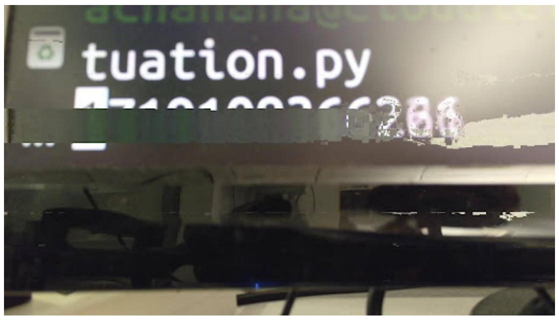
[1]Credit to Qifei Dong, Ph.D. student at Carnegie Mellon University, for devising this approach.

regions in the decoded frame. This degradation stops as new I-slices are received during the refresh cycle, as the decoder only needs the latest I-slices.

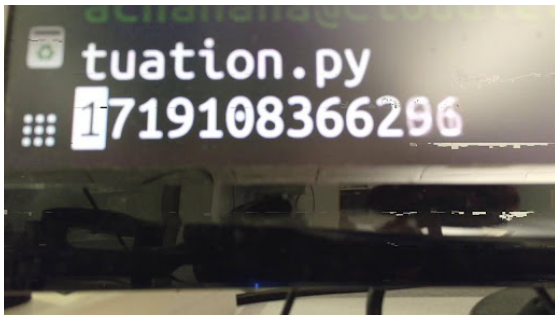
When packets are unfiltered, we can look for the first network packet that contains an I-slice for a slice of our choosing. H.264 I-slices and P-slices are encapsulated into Network Abstraction Layer Units (NALUs) when transmitted over a network, which contains metadata that allows us to infer information about the type of slice and the row that it corresponds to.

In practice, it is easier to filter network packets containing NALUs with sequence parameter set (SPS) and picture parameter set (PPS) information, which has general metadata related to the video stream, such as the video format. This metadata is always sent along with the first frame in a GOP. Since the first frame of the GOP contains the I-slice for rows 21 through 25 of the frame, we need to identify the first frame that resolves degradations for these rows.

In Figure 4.12a, we see that the slice corresponding to the Unix timestamp shown on the screen is degraded in the first few frames after we unfilter UDP packets. Once we find the first network packet containing an I-slice corresponding to this row, we can map it to the first frame output by the decoder that resolves the degradation for that row (Figure 4.12b).



(a) Degraded timestamp because of missed I-slice



(b) I-slice received for slice corresponding to timestamp

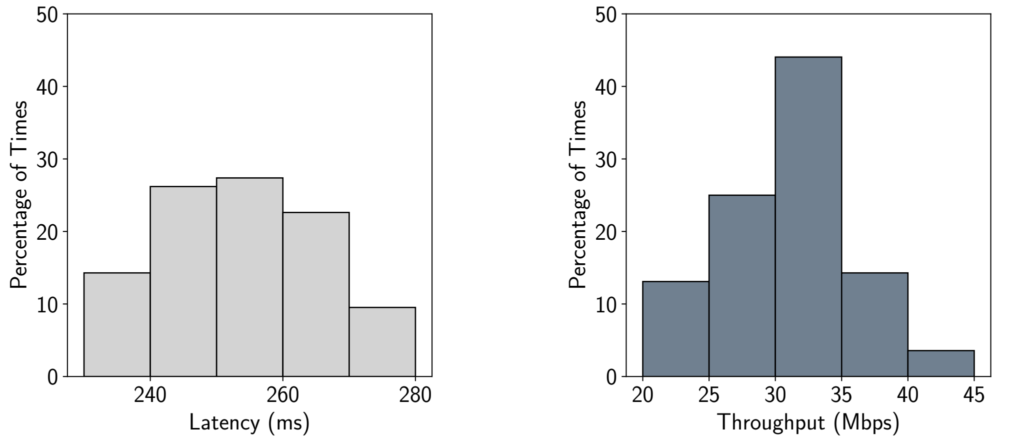
Figure 4.12: Degraded Regions in Decoded Frame After Unfiltering UDP Packets

The kernel timestamps network packets, so we obtain t_3, the time the first packet for frame Figure 4.12b arrived. Once a correspondence has been establish between a network packet and a frame, we can look at the logical timestamp that is associated with each frame during encoding and included in each packet's metadata to obtain t_3 for subsequent frames—whenever the logical timestamp increments, we know that the packet corresponds to the next frame.

4.4.2 Experimental Results

In this section we share our results obtained from benchmarking each OODA loop component. For latency we share the 99th percentile measurements and for bandwidth we share the first percentile measurements, in addition to the mean and standard deviation. "p99" refers to the 99th percentile measurement and "p1" refers to the first percentile.

4.4.2.1 Observe$_{ab}$



Mean: 253±12 ms p99: 277 ms

(a) Observe$_{ab}$ Latency

Mean: 30.83±4.95 Mbps p1: 22.1 Mbps

(b) Observe$_{ab}$ Throughput

Figure 4.13: Observe$_{ab}$ Measurements

Figure 4.13 shows that onboard drone sensing and pre-processing, Observe$_{ab}$, has a mean latency of 253 ms, with a standard deviation of 12 ms, and a p99 of 277 ms. Instantaneous throughput is also measured in frames per second (fps), using the time taken for a new frame to be transmitted by the drone. It has a mean of 31 fps, with a standard deviation of 5 fps and a p1 of 22 fps.

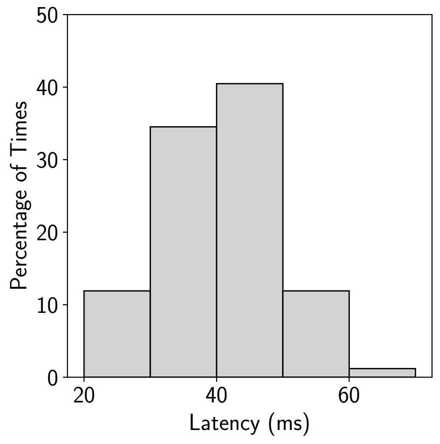
4.4.2.2 Observe$_c$

Figure 4.14 shows the measurements obtained for Observe$_c$, which involves transmission from the drone to the cloudlet. This component includes a short Wi-Fi segment from the drone to the Onion Omega, carried onboard the drone, and then a longer 4G LTE segment to the cloudlet. Observe$_c$ has a mean latency of 39 ms, with a standard deviation of 8 ms, and a p99 of 59 ms. The throughput measured on the link using `iperf` has a mean of 16.25 Mbps, with a standard deviation of 1 Mbps and a p1 of 14.36 Mbps.

4.4.2.3 Orient+Decide$_d$

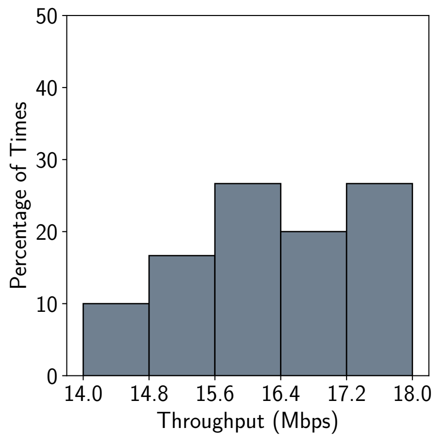
Figure 4.15 presents the latency and throughput measurements for Orient+Decide$_d$. This component includes three stages:

- **Stage 1.** Decoding of the drone's RTSP H.264 video stream to produce individual frames
- **Stage 2.** Inferencing on each frame. This could involve using a deep neural network (DNN) to perform tasks such as depth estimation or object detection.



Mean: 39±8 ms p99: 60 ms

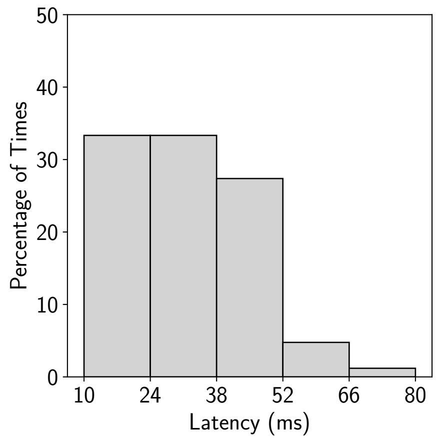
(a) Observe$_c$ Latency



Mean: 16.25±1.01 Mbps p1: 14.36 Mbps

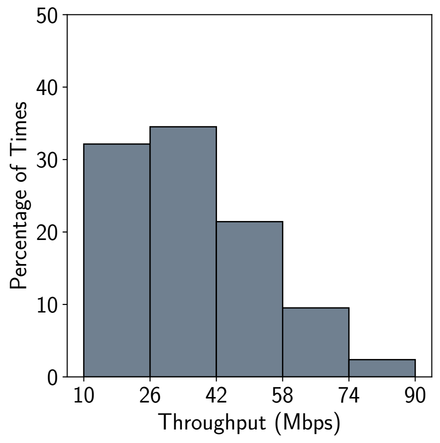
(b) Observe$_c$ Throughput

Figure 4.14: Observe$_c$ Measurements



Mean: 32±13 ms p99: 59 ms

(a) Orient+Decide$_d$ Latency



Mean: 36.57±15.88 fps p1: 17.24 fps

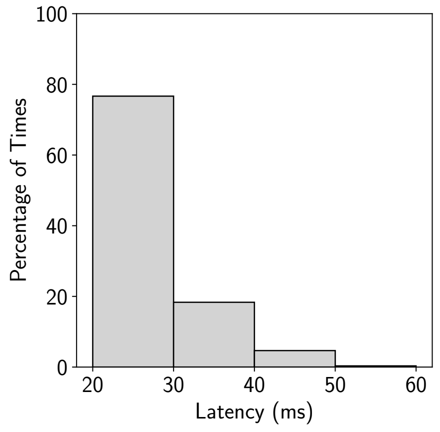
(b) Orient+Decide$_d$ Throughput

Figure 4.15: Orient+Decide$_d$ Measurements

- **Stage 3.** Determining drone actuation needed based on findings from Stage 2 and generating the corresponding command.

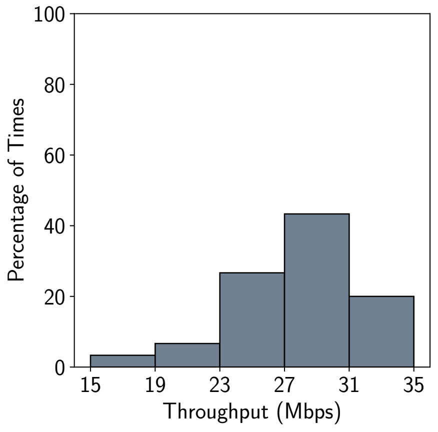
Stage 2 and Stage 3's performance can vary depending on the application. Different DNNs can have different inference times, and the decision logic can vary in complexity. Stage 1, however, remains the same and so we focus on measuring Stage 1 for Observe+Decide$_d$. We see a mean latency of 32 ms for Stage 1, with a standard deviation of 13 ms, and a p99 of 59 ms. The decoding software's throughput was measured as the inverse of its latency, giving an indication of the frame rate that the decoding software can sustain on our cloudlet hardware. The throughput has a mean of 37 fps with a standard deviation of 17 fps and a p1 of 17 fps.

4.4.2.4 Act$_e$



Mean: 30±4 ms p99: 49 ms

(a) Act$_e$ Latency



Mean: 28±3.66 Mbps p1: 19 Mbps

(b) Act$_e$ Throughput

Figure 4.16: Act$_e$ Measurements

Figure 4.16 shows the measurements for Act$_e$, which involves the transmission of drone actuation commands back to the drone via the Onion Omega. This is the return path of Observe$_c$, and involves a 4G LTE and Wi-Fi segment as before. Since drone commands have a small data size, latency is a more crucial aspect of this component. Nevertheless, bandwidth for Act$_e$ is much higher than Observe$_c$ because for the Onion Omega it corresponds to 4G LTE downlink, which offers a higher bandwidth than uplink because mobile networks are designed to optimize downlink performance.

The latency has a mean of 30 ms, with a standard deviation of 4 ms and a p99 of 49 ms. The throughput has a mean of 28 Mbps, with a standard deviation of 3.66 Mbps and a p1 of 19 Mbps.

4.4.2.5 Act$_{fg}$

Act_{fg} involves processing of an actuation command by the drone and the initiation of actuation. We measure the latency as the time difference between the receipt of an actuation command by the drone and the start of actuation. We measure actuation performance using gimbal actuations because they are commonly used for object tracking and are easy to measure without needing to fly the drone. To measure it, we position a stationary drone in front of a display connected to the cloudlet showing the current timestamp at millisecond granularity. An actuation command is sent to the drone to move its gimbal while the display and gimbal are video-recorded using a slow-motion camera.

Run	Latency (ms)
1	188
2	170
3	162
4	189
5	155
Mean	173 ±15

Figure 4.17: Act_{fg} Latency

We output the timestamp at which the actuation command is sent and identify the first video frame showing gimbal movement. Then, we calculate the difference between the timestamp shown on the display in this frame and the timestamp at which the drone actuation command was sent. Act_{fg} latency is calculated by subtracting the mean Act_e latency from this figure. Figure 4.17 shows that Act_{fg} has a mean latency of 173 ms, with a standard deviation of 15 ms. We use a slow-motion camera which operates at 240 fps, resulting in a frame interval of approximately 4 ms. The measurements have an error margin of about five frames, resulting in an experimental error of approximately 20 ms.

Act_{fg} involves electromechanical actuation, which does not have a concept of streaming. As a result, throughput can be interpreted as the reciprocal of latency.

4.4.3 Overview of OODA Loop

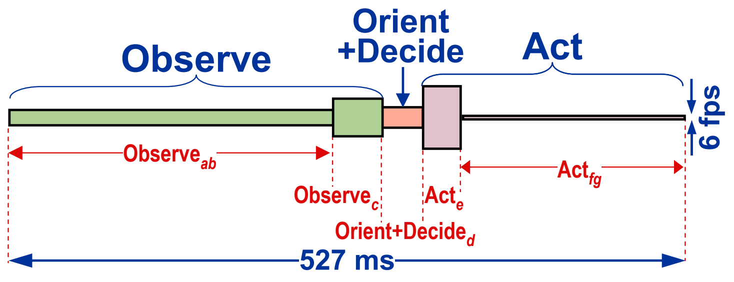


Figure 4.18: SteelEagle OODA Loop Throughput and Latency Overview

Figure 4.18 presents a summary of the measurements from Section 4.4.2, with the height and width of each component representing the bandwidth and latency respectively. This depiction shows that in the best case, assuming no time is spent on Stage 2 and Stage 3 in the Orient+Decide_d component, the drone takes 527 ms to react to a change in its environment. In practice, Stage 2 and Stage 3 can involve complex computationally-intensive inferencing and decision-making, such as the use of DNNs for depth estimation or object tracking, and route planning algorithms. The height and width of the Orient+Decide_d component in Figure 4.18 would need to be scaled appropriately to account for this.

Observe$_{ab}$, which involves sensing and pre-processing on the drone, contributes to almost half of the total OODA loop latency, or more than two-thirds of the drone-to-cloudlet latency, which consists of Observe$_{ab}$, Observe$_{c}$, and Orient+Decide$_{d}$. This is a property of the Parrot Anafi drone that we used in our evaluation. Other drones could have different values of Observe$_{ab}$.

In our evaluation, we measured Act$_{fg}$ as the time taken for the drone to commence moving its gimbal. Gimbal movement is inherently slower than, say, turning on an LED on the drone because it involves electromechanical actuation. While we focused on gimbal actuations, other kinds of actuation such as changing rotor speed may be be faster than gimbal actuations depending on the inertia of the moving component and the available torque in the motor.

Gimbal actuation performance is critical in cases where rapid actuations are needed to accomplish a task. If the drone is, say, following a target traveling at a fixed speed, it does not need to actuate the gimbal and thus Act$_{e}$ and Act$_{fg}$ are not involved. Only when the target moves abruptly does the drone need to rapidly actuate to continue following the target. Without abrupt target movement we see that Observe$_{ab}$ dominates the OODA loop.

While we analyzed the OODA loop assuming no time is spent on inferencing or planning, the eventual goal is to optimize other components so that inferencing starts dominating the OODA loop latency. Achieving this will require lower latency cellular connectivity and drones with better Observe$_{ab}$ and Act$_{fg}$ performance.

Chapter 5

Utilizing Onboard Compute In SteelEagle

This chapter discusses work to replace the communications relay used in SteelEagle to the Modal AI VOXL 2. The VOXL 2 offers significant computational capability, allowing the execution of machine learning models. We start by characterizing this shift in offloading strategy in more general terms.

5.1 The Computation Offload Spectrum

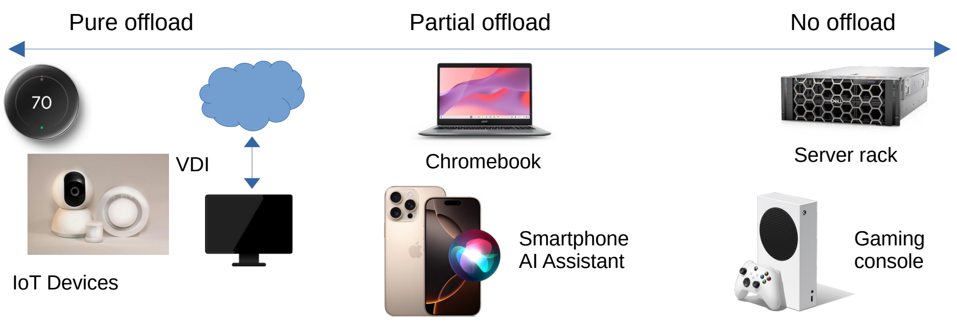


Figure 5.1: Devices Placed in the Computation Offloading Spectrum

Devices today leverage offloading to various degrees. Figure 5.1 shows how devices can be placed in an offloading spectrum, with thin clients found towards the left of the spectrum and thick clients to the right. A thin client, as opposed to a thick client, has minimal compute and can be made smaller, lighter, and cheaper. Virtual desktop infrastructure (VDI), for example, leverages pure offload by providing remote access to desktops hosted on centralized servers. This reduces the computational demands on the client device, enabling the use of computationally-intensive applications on weaker and older "thin" hardware, reducing costs. Similarly, internet-of-things (IoT) sensors, such as video cameras, circumvent their limited storage and computational ability by leveraging pure offload, transmitting their sensor streams to the cloud for storage and analysis.

On the other end of the spectrum, we find devices such as gaming consoles, desktop computers, and autonomous cars. Gaming consoles and desktop computers are expected to have crisp interaction, which is not achievable by offloading to cloud. Offloading to cloudlets, which can provide low latency, is not a feasible option for these devices currently because of the absence of widespread cloudlet infrastructure. Gaming consoles and desktop computers can provide high-levels of onboard computational power because they are not subject to mobility constraints. Although autonomous cars are mobile, they typically have sufficient energy available to perform intensive onboard computations since the majority of energy is needed to propel the car forward. They must also perform real-time decision making reliably even in the absence of network connectivity, which makes offloading less appealing.

In the middle of the spectrum, we find devices that perform partial offloading. These devices have sufficient computational power to function when offloading is unavailable or not advantageous, but they also offload computation in other cases. Smartphone AI voice assistants such as Apple's Siri, for example, originally offloaded all computation to the cloud. Over time, it evolved to utilize on-device capabilities to perform simpler tasks, allowing its use in the absence of an internet connection. Chromebooks were designed with a focus on the use of cloud-based services, allowing them to be fitted with lower processing power, memory, and storage capacity compared to traditional laptops.

5.1.1 Deciding the Offloading Strategy

Choosing an offload strategy for a given application can be done based on some key attributes that are considered important for the application:

- **Mobility requirements.** Is the application subject to mobility constraints?
- **Disconnected operation.** Does the application need to work reliably in the face of network disconnection or degradation? Is degraded operation acceptable during disconnected operation?
- **Latency requirements.** Is the application latency-sensitive?
- **Bandwidth constraints.** How much network bandwidth is available?
- **Cost requirements.** Is low cost for devices a priority in this application?

These attributes encapsulate the benefits and limitations of offloading. Offloading is most beneficial for mobile devices. A mobile device that is dependent entirely on offloading will be unable to function adequately when offloading is impossible altogether because of network disconnection, or impacted because of degraded network conditions, such as high latency and low bandwidth availability. Applications that are mission-critical would be wise to equip devices with sufficient computational resources to support an acceptable level of function when adverse network conditions impact offloading. Applications that are not mission-critical can rely on pure offload to benefit from the cost savings resulting from using devices that are not required to have significant computational resources.

Offloading allows mobile devices to achieve a much higher level of functionality than their onboard hardware allows for. Equipping mobile devices with the hardware needed to maintain the same level of functionality during offload degradations reduces the benefit of offloading—the mobile device is weighed down with the additional size and weight of more complex hardware that remains unused when offloading is possible.

A more reasonable approach is to include inferior on-device computational resources on the device, especially if offload degradations are infrequent. These resources provide results that are inferior according to an output-specific metric, compared to those obtained using offloading. For predictions from a machine learning model, the metric could be the accuracy of the prediction. In previous work, the concept of *fidelity* has been used in this context, defined as the degree to which the results differ according to the metric [23].

5.1.2 The Dynamic Nature of Partial Offload

Network conditions affect the performance of offloading, but they are dynamic in nature. Mobile devices utilizing partial offload, then, must decide an offloading strategy at runtime. But the decision is not binary—exclusively using onboard compute or remote compute resources. As Figure 5.1 shows, there is a wide range between the two ends of the spectrum. Depending on the current network conditions, partial-offload mobile devices can utilize a combination of onboard and remote computational resources that results in optimal performance.

Mobile devices can achieve this through an optimal partitioning of applications into local and remote components. Flinn et al tackled the issue of developing applications that can support devices with different capabilities and dynamic execution conditions with a self-tuning remote execution system. The system, called *Spectra*, continuously monitors the application's resource supply and demand and suggests which components of an application should be executed remotely, taking into account factors such as performance, energy usage, and fidelity [37]. This work requires the application developer to partition the application, specifying application components that could benefit from remote execution.

More recently, projects such as MAUI [38] and CloneCloud [39] have attemped to reduce the reliance on developers to partition applications, and require minimum modification of existing software. MAUI requires the developer to explicitly mark methods that are safe to execute remotely, deciding at runtime whether to offload methods or execute them locally. Its partitioner includes a profiler and solver, determining execution costs and making offloading decisions through reduction to an optimization problem with the costs as input. CloneCloud performs offloading at the thread level using thread migration, completely eliminating the need for software application modification and making offloading decisions transparently.

5.1.3 Offloading Shaping

Hu et al [40] demonstrated the value in doing additional onboard computation when offloading, which was not originally part of the application. The approach, called *offload shaping*, attempts to conserve wireless bandwidth and energy through the process of *early discard*. The process involves selectively sending inputs for offload computation based on an application-specific metric of value assigned to each input. In the case of video analytics, for instance, onboard computation can detect blurry frames and not transmit them to the cloudlet. Hu et al showed that object recognition does not perform well on blurry frames, and so it is a waste of bandwidth and energy to transmit these blurry frames to a cloudlet for computation.

5.2 A Partial Offload Pipeline for SteelEagle

SteelEagle drones can currently be placed in the "pure offload" part of the spectrum in Figure 5.1, since they do not perform any on-board computation other than the encoding of the raw camera feed to an RTSP H.264 video stream that is transmitted to a cloudlet. As discussed in Section 5.1.1, a pure offload strategy does not support disconnected operation, and impacts the application considerably as network conditions degrade.

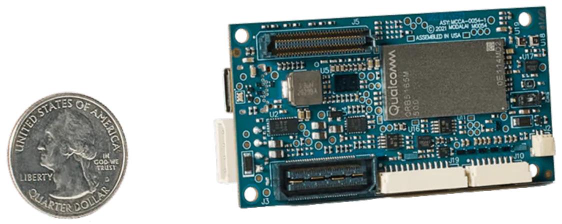


Figure 5.2: Modal AI VOXL 2

Replacing the Onion Omega 2 with the Modal AI VOXL 2 (Figure 5.2) as the communications relay in the SteelEagle pipeline will allow us to pursue a partial offloading strategy that allows SteelEagle drones to gracefully react to changes in network conditions. Intended for use as an AI autopilot in custom drones, the VOXL 2 features a Qualcomm QRB5165 processor and a PX4 flight controller. We do not utilize the PX4 flight controller on the VOXL. Table 5.1 provides information about the computational resources available on the VOXL 2.

The VOXL 2 has support for running Google's Lite Runtime (LiteRT) machine learning models, formely known as TensorFlow Lite (TFLite). Trained TensorFlow and PyTorch models can be converted to LiteRT models by using techniques such as quantization and pruning [41]. Float16 quantization reduces the floating-point precision of the model's weights to 16-bits, and full integer quantization converts them to integers. This reduces the model's size, memory usage and inference latency, making it more suitable for mobile device inferencing. Float16 quantization, for instance, reduces the size of the model by half and causes minimal loss in accuracy. However, GPU acceleration of float16-quantized models requires half-precision floating point support (FP16) [42].

5.2.1 Mapping to the OODA Loop Framework

Figure 5.3 depicts the new pipeline with the VOXL 2. In the existing SteelEagle architecture, as described in Section 3.2, video decoding is performed on the cloudlet since all video stream

	ModalAI VOXL 2
Architecture	64-bit ARM
CPU	Qualcomm Kyro 585 (7nm, released Dec 2019)
	1 x 2.84 GHz high-performance core
	3 x 2.42 GHz high performance cores
	4 x 1.80 GHz low-performance cores
ISP	Qualcomm Spectra 480
GPU	Qualcomm Adreno 650, with support for OpenCL
DSP	Qualcomm Hexagon 698
Memory	8 GB
Weight	16 grams
Power consumption	4-5 W under high load
Operating system	Ubuntu 18.04

Table 5.1: Technical specifications of the Modal AI VOXL 2 system-on-a-chip

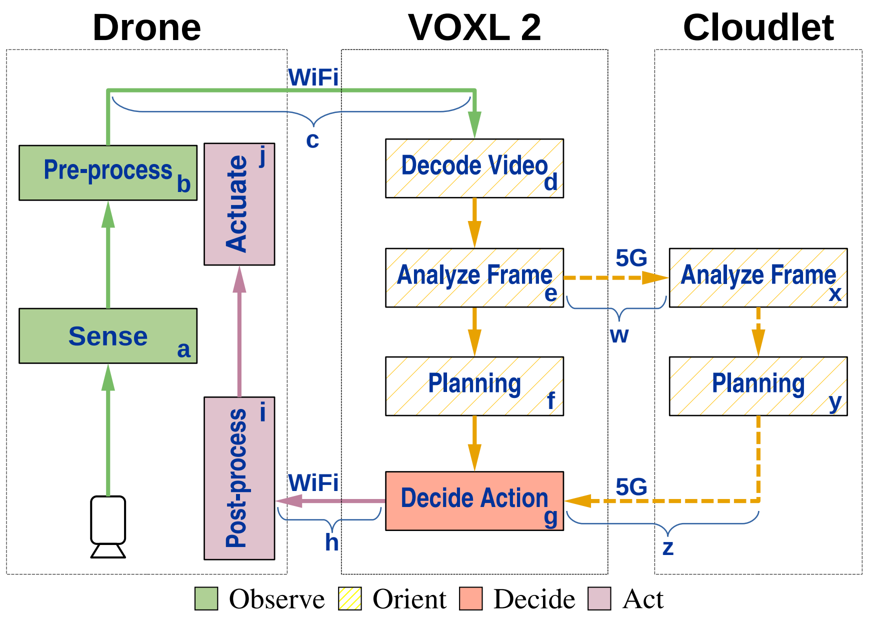


Figure 5.3: Mapping the VOXL 2 pipeline to the OODA loop

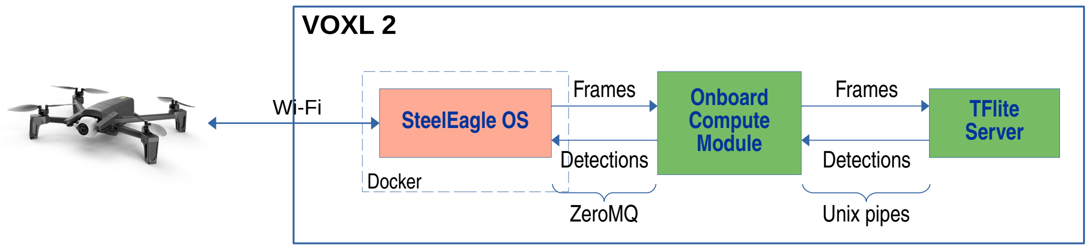
network packets are forwarded to the cloudlet using the Onion Omega as a network relay. To perform onboard computation on the VOXL 2, we must obtain individual frames from the drone's video stream. This requires decoding the H.264 video stream on the VOXL 2, which commences the Orient phase. The VOXL pipeline has an inner and outer Orient loop:

- **Inner loop**: Corresponds to components Orient$_e$ and Orient$_f$, which involve analysis of the decoded frame and planning on the VOXL
- **Outer loop**: Corresponds to components Orient$_w$ through Orient$_z$. Involves offloaded analysis of the decoded frame and planning on the cloudlet

The outer loop involves offloading to the cloudlet, which incurs propagation and transmission delay to and from the cloudlet, Orient$_w$ and Orient$_z$. The cloudlet, being a stationary device, can run more computationally-intensive analysis and planning phases which can yield more accurate results in less time.

To conserve bandwidth, the outer loop is not always exercised. The pipeline can leverage offload shaping techniques (Section 5.1.3) to determine if the frame should be discarded. Then, a remote execution system (Section 5.1.2) can determine if processing should be offloaded. Once frame processing is complete, the pipeline determines whether any drone actuation needs to be performed, and generates the corresponding command.

5.2.2 Onboard Computation Design



Components in red are developed in Python, and components in green in C++

Figure 5.4: VOXL 2 Onboard Computation Pipeline

The VOXL 2 supports the Open Computing Language (OpenCL) standard, which assists in writing parallel programs which leverage its hardware accelerators, including its GPU. The VOXL 2 ships with a software suite that includes various programs designed to be run as `systemd` services. One of these programs, VOXL TFlite Server, performs hardware-accelerated machine learning inference. The inputs and outputs of the programs are sent through an inter-process communication library called `libmodalpipe`, which uses Unix pipes. `libmodalpipe` includes data structures that represent data items such as camera image metadata and the results of inference. For example, `camera_image_metadata_t` specifies image metadata such as width, height, size and format of the image, and `ai_detection_t` includes information relevant to object detection models, such as detected classes, confidence levels, and bounding boxes.

VOXL TFlite Server is designed to take the drone's camera data continuously as input through a Unix pipe and output detections through another Unix pipe. Any service interesting in consuming the inference results can register as a client to the output pipe. We can utilize this framework to perform onboard inference, publishing frames requiring inference to a pipe configured as input to VOXL TFLite Server.

The VOXL software ecosystem is targeted for C/C++ programs. However, the SteelEagle pipeline is currently developed in Python. As a result, a C++ proxy is required that can receive frames from the SteelEagle program and forward them to the VOXL TFLite Server for inference. As Figure 5.4 shows, an onboard compute module was developed that communicates with SteelEagle using the ZeroMQ messaging library, listening for inference requests. ZeroMQ has bindings available for popular programming languages, making it suitable for this use case. A SteelEagle stub provides a convenience API that generates a `protobuf` message that is sent to the onboard compute module. The `protobuf` message includes the contents of the frame as an RGB image along with frame metadata such as width and height. The onboard compute client generates the `camera_image_metadata_t` structure that VOXL TFLite Server expects, and sends it to its input pipe along with the frame data.

Once the VOXL TFLite Server receives an input, it publishes one `ai_detection_t` data structure to the output pipe for each detection on a given frame. There is no metadata published that specifies the number of detections to expect. This works well if a consumer wants a live stream of object detections. However, in our case, we only want to reply to the SteelEagle program once we receive all detections for a given frame. To achieve this, the VOXL TFLite Server program was modified to send a delimiter `ai_detection_t` data structure once all the detections have been sent for a given frame. Once this delimiter is received, the onboard compute module adds all the detections to a single `protobuf` message, and sends it back to the SteelEagle program.

5.2.3 Adaptive Transmission Rate in the VOXL 2 Pipeline

In the VOXL pipeline, frames are decoded on the VOXL instead of the cloudlet. This means that if the remote execution system decides to offload computation to the cloudlet, it must send individual frames to the cloudlet. This increases bandwidth requirements substantially. Previously, a bandwidth of 5 Mbps was sufficient to send the video stream to the cloudlet. Individual 720p frames average about 180 KB when encoded as JPEG, requiring a bandwidth of 43.2 Mbps to send 30 frames per second.

As Table 5.2 shows, attempting to send 30 frames per second on the VOXL 2, without any flow control, overwhelms the receiver since the VOXL 2 does not have sufficient bandwidth. Only a frame rate of 12.58 frames per second is achievable when sending frames at full-fidelity. We also observe queueing delay leading to significantly higher latency when attempting to send full-fidelity frames. Reducing frame size allows us to achieve better throughput, and gradually reduce latency as network queuing delay and transmission delay decrease.

Restricting our transmission rate to 10 frames per second, Table 5.3, we no longer observe queuing delay. Even with full-fidelity frames we are able to achieve the desired frame rate.

Frame details	Frame size (KB)	Avg. Latency (ms)	Achievable FPS
720p	180	934±30	12.58
720p, 80% JPG quality	98	600±65	28.63
720p, 70% JPG quality	79	304±12	29.95
360p	79	307±20	29.89
360p, 80% JPG quality	38	288±13	29.89
360p, 70% JPG quality	31	286±17	29.85

"Achievable FPS" is calculated at the cloudlet by considering the number of frames that are received in a given time frame

Table 5.2: Offloading Performance on the VOXL 2 (30 FPS)

Frame details	Frame size (KB)	Avg. Latency (ms)	Achievable FPS
720p	180	363±22	9.98
720p, 80% JPG quality	98	343±26	10
720p, 70% JPG quality	79	340±46	9.96
360p	79	312±23	9.95
360p, 80% JPG quality	38	301±27	9.97
360p, 70% JPG quality	31	289±18	9.98

"Achievable FPS" is calculated at the cloudlet by considering the number of frames that are received in a given time frame

Table 5.3: Offloading Performance on the VOXL 2 (10 FPS)

These results show that an adaptive transmission based on current network conditions is essential in the VOXL pipeline. Setting a low static frame rate up front can leave performance on the table if the available bandwidth can sustain a higher frame rate. Similarly, the available the bandwidth might not even be sufficient to sustain the static frame rate set, which is highly possible in commercial cellular networks.

Chen [3] approached this issue through the use of a token-bucket filter, which limits the number of frames allowed to be in flight at any given time. The mechanism replenishes tokens by having the receiver send back a token when a frame has been received. Table 5.4 shows the results when using a token-bucket filter in the VOXL pipeline, sending frames at full-fidelity (720p).

Total Tokens	**Average Latency (ms)**	**Frame Rate (FPS)**
1	398±32	6.03
2	443±40	11.42
3	506±34	11.11

Table 5.4: Drone-to-cloudlet Latency with Different Token Counts

In Chen's approach, the number of tokens used are not dynamic. According to Chen, a small token count minimizes latency at the expense of throughput and resource utilization. This is indeed the observation we make: a token count of one results in a mean latency of 398 ms, and going up to three tokens, which allows up to three frames to be in flight at a time, results in a mean latency of 506 ms. A set-and-forget approach that uses just one token optimizes latency, but can lead to a lower throughput than otherwise possible, especially if network bandwidth improves.

Our observations show the need for future work to achieve an optimal balance of latency and bandwidth based on the current network conditions. A dynamic token limit, for instance, can help achieve this. Obtaining real-time telemetry that indicates the end-to-end transmission time can inform a decision to dynamically alter token counts.

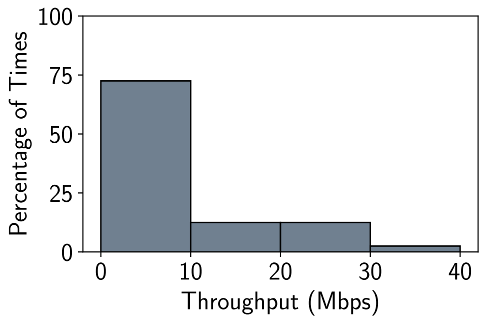
These results also emphasize the fact that bandwidth is a precious and limited resource that can vary over time. Figure 5.5 shows bandwidth measurement obtained using `iperf` on the VOXL using a commercial 5G provider. The mean of 11.28 Mbps is decent, but comes with a standard deviation of 9.34 Mbps and a p1 of 0 Mbps! These findings underscore the importance of conserving bandwidth using early discard techniques discussed in Section 5.1.3.

5.2.4 OODA Loop of VOXL 2 SteelEagle Pipeline

Using terminology from Figure 5.3, we evaluate the performance of each component of the VOXL pipeline.

Observe$_{ab}$

The Observe$_{ab}$ component of the OODA loop in the new pipeline is the same as in the case of the Onion Omega (Section 4.4.2.1), including on-drone sensing and pre-processing, which involves



Mean: 11.28±9.34 Mbps p1: 0 Mbps

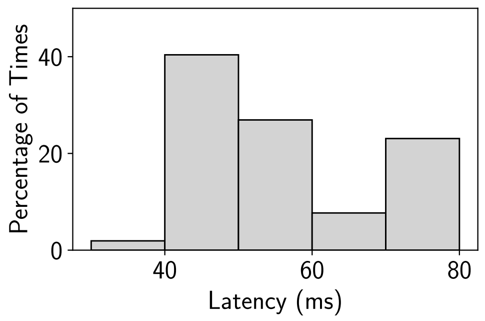
Figure 5.5: Bandwidth Measurement on the VOXL Using 5G Modem

generation of an RTSP H.264 video stream from the drone camera's raw video frames. We use the same drone, the Parrot Anafi, for the VOXL pipeline. As before, this component has a mean of 253 ms, with a standard deviation of 12 ms, and a p99 of 277 ms.

Observe$_c$

Observe$_c$ involves a short Wi-Fi segment from the drone to the VOXL 2, with the drone positioned centimeters away. Measurements obtained using the `ping` tool show that the round-trip latency has a mean of 10.33 ms, with a standard deviation of 1.56 ms. This provides us with an estimate of one-way propagation delay of 5 ms. Bandwidth on this segment cannot be measured since the drone is a black box.

Orient$_d$

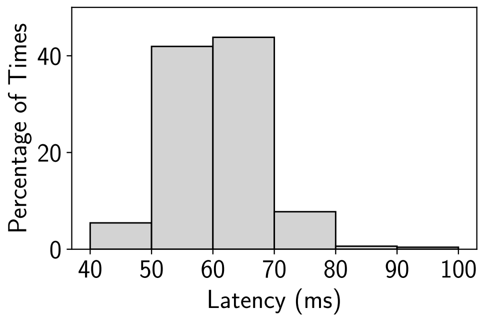


Mean: 55±12 ms p99: 76 ms

Figure 5.6: Orient$_d$ Measurements

Orient$_d$ involves the decoding of the RTSP video stream generated by the drone. Figure 5.6 shows the measurements of the video decoding. The latency distribution has a mean of 55 ms, with a standard deviation of 12 ms, and a p99 of 76 ms. This is comparable to the decoding latency we obtained for the Onion Omega pipeline (Section 4.4.2.3), which had a mean of 32 ms, with a standard deviation of 13 ms and a p99 of 59 ms. In the case of the Onion Omega, the video decoding occurred on the cloudlet, where as now it happens on the VOXL.

Orient$_e$



Mean: 61±7 ms p99: 80 ms

Figure 5.7: Orient$_e$ Measurements Using a Float16-quantized YOLOv5 Model

Orient$_e$ corresponds to Stage 2 of the Onion Omega pipeline, discussed in Section 4.4.2.3. Figure 5.7 shows measurements for running a machine learning model on the VOXL. We use a float16-quantized YOLOv5 model for inference, which is used for detecting objects. We obtain a mean latency of 60.58 ms, with a standard deviation of 6.98 ms and a p99 of 80 ms.

Orient$_f$

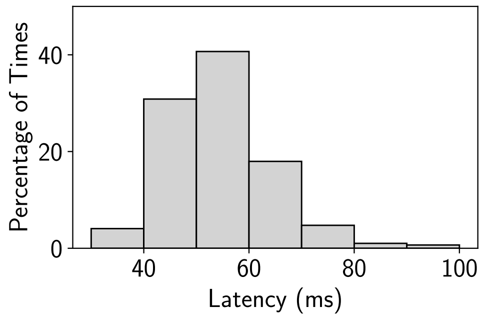
Orient$_f$ involves processing on the VOXL with the inference results obtained from Orient$_e$. The performance of this component is very application-specific, so we ignore it for the purposes of this assessment.

Orient$_w$

Orient$_w$ involves the latency for sending a frame to the cloudlet for offloaded computation. Figure 5.8 shows that this latency has a mean of 54 ms, with a standard deviation of 10 ms and a p99 of 83 ms.

Orient$_x$

Table 5.5 shows the performance of various YOLO models on a cloudlet. The mean latency varies from 28 ms for the smallest YOLO model to 42 ms for the largest. The latency of onboard inference, Orient$_e$, is greater than that of the largest YOLO model on the cloudlet, which emphasizes the mobility penalty that the VOXL incurs.



Mean: 54±10 ms p99: 83 ms

Figure 5.8: Orient$_w$ Measurements

Model	**Latency** (ms)	**Throughput** (fps)	**mAP**
YOLOv5s	28	25	56.8
YOLOv5m	37	20	64.1
YOLOv5l	42	20	67.3

The inference and throughput were obtained on our cloudlet. The mean average precision (mAP) is from the YOLO documentation.

Table 5.5: YOLOv5 Performance on Cloudlet (from [43])

Orient$_y$

Orient$_y$ performs the same function as Orient$_f$ on the VOXL. It involves processing with the inference results obtained from Orient$_x$. The performance of this component is very application-specific, so we ignore it for the purposes of this assessment.

Orient$_z$

Orient$_z$ involves a 5G segment from the cloudlet to the VOXL 2. We re-use the measurements from Orient$_w$ for this component.

Act$_h$

Act$_h$ involves a Wi-Fi segment from the VOXL 2 to the drone. We obtain the same measurement for Act$_h$ as Orient$_c$, with a estimate of 5 ms obtained using the `ping` utility.

Act$_{ij}$

Act$_{ij}$ is the same as Act$_{fg}$ in the Onion Omega pipeline, discussed in Section 4.4.2.5, involving drone post-processing and actuation. For the actuation of the Parrot Anafi drone's gimbal, it has a mean of 173 ms and a standard deviation of 15 ms.

5.2.5 Overview of the VOXL Onboard Computation OODA Loop

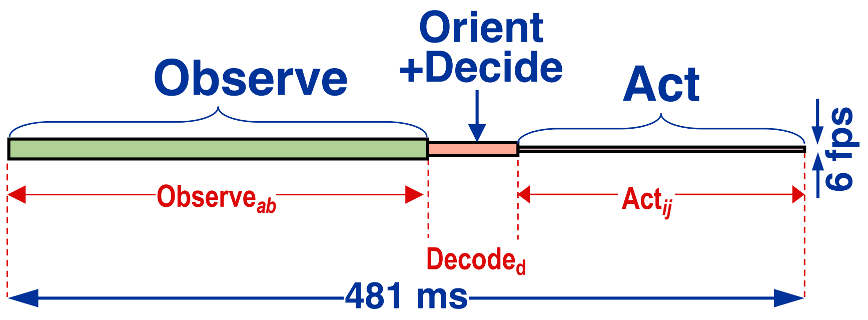


Figure 5.9: VOXL OODA Loop Throughput and Latency for Offloaded Computation

Figure 5.9 provides a visual depiction of the OODA loop using onboard computation on the VOXL 2, showing a 481 ms OODA loop. In Figure 4.18, we provided a similar visual depiction of the OODA loop for the Onion Omega payload where we achieved a latency of 527 ms. With the VOXL, we eliminate the Observe$_c$ and Act$_e$ components that involved 4G LTE segments to and from the cloudlet, eliminating 70 ms of latency from the OODA loop (Sections 4.4.2.2 and 4.4.2.4). At the same time, onboard video decoding on the VOXL takes 55 ms, an increase of 23 ms from decoding on the cloudlet. This gives the VOXL about a 46 ms tighter OODA loop compared to offloading to the cloudlet.

However, as discussed in Section 4.4.3, these depictions assume that no time is spent on inference and planning stages, which can involve the use of deep neural networks and Kalman filters. As such, these depictions capture the best-case OODA loops. Consequently, since onboard inference on the VOXL takes longer than that on the cloudlet, the benefit of onboard computation over offloading decreases. For instance, as Table 5.5 shows, object detection on the cloudlet takes anywhere from 28 ms to 42 ms depending on the model. But doing the same on the VOXL with a quantized object detection model can take upwards of 60 ms (Figure 5.7). This reduces the advantage of onboard computation on the VOXL by at least 18 ms on average.

Overall, the VOXL currently has a 28 ms tighter OODA loop factoring in inference. It is easy for hardware on the cloudlet to close the gap by using sophisticated modern GPUs. In our setup, the cloudlet is configured with an NVIDIA GTX 1080 Ti which is almost seven years old at this point.

5.2.6 Optimal Use of Onboard Compute

Compared to the cloudlet, the VOXL 2 offers reduced flexibility, requiring engineering effort to create quantized models that can run on it. The requirement for using OpenCL, a framework that target heterogeneous compute, presents a large learning curve and supports fewer libraries out of the box compared to popular frameworks such as CUDA and ROCm. Consequently, it is most practical to implement critical functionality that does not frequently change on the VOXL. Functionality that is not critical to safe drone flight can be implemented using offloading, which benefits from the flexibility to run any off-the-shelf model without modification because of its generality. This flexibility is incredibly valuable, allowing for a much more agile development cycle. Tasks critical to safe flight, such as following mission waypoints and avoiding obstacles, benefit from the resiliency offered by onboard computation.

Our results show that while utilizing onboard computation resources using the VOXL offers a tighter OODA loop than offloading, the margin is small compared to the overall OODA loop latency. The drone's sensing, pre-processing, post-processing, and actuation (Observe$_{ab}$ and Act$_{fg}$) still dominates the overall pipeline. However, onboard computation is extremely valuable for resilience in the face of network disruptions.

As the bottlenecks for Observe$_{ab}$ and Act$_{fg}$ are resolved, onboard computation will offer a much more significant advantage. A range of issues will come to the forefront when this happens. One such issue relates to the fusion of computational results from onboard and offloaded computation. Onboard computation would eventually offer inference results at a much higher rate than what offloaded computation could. What happens if the results disagree with each other, for instance if only one result indicates the risk of imminent collision? The result from the offloaded computation will always be more delayed, while the result from the onboard computation will always be less accurate. It is not clear which result to trust in such a situation. One possible strategy involves first assessing the risk associated with an incorrect or delayed actuation. If either results indicates the risk of destruction, it may be wise to exercise extra caution and actuate to avoid destruction. If the risk level is low, there are other ways in which the disagreement can be resolved, such as by getting a closer look at the target.

Chapter 6

Conclusion and Future Work

This thesis analyzed the OODA loop of SteelEagle, an edge-enabled autonomous drone system. We performed an in-depth analysis of the system's performance to identify bottlenecks, and discussed approaches to resolve the limitations of SteelEagle using a new payload with onboard computational capabilities. We end our discussion with concluding remarks and a discussion of future work.

6.1 Conclusion

Considering the ability of aerial drones to effectively tackle many difficult problems such as search and rescue and industrial inspections, this thesis discusses the advantage of cost-effective autonomous drone operation. We identified the performance of autonomous drone systems, particularly in terms of latency, as a critical attribute that determines the practicality and versatility of the system. Consequently, we performed a detailed investigation into the SteelEagle pipeline, measuring the latency and bandwidth of various components.

We discussed initial attempts to benchmark the system that led to identifying the FFmpeg video decoding software as a system bottleneck. We switched to a different video decoding software that reduced drone-to-cloudlet latency by a factor of two. Recognizing the need for a systematic approach to benchmarking the system, we devised a technique to obtain granular measurements from each component of the pipeline. We mapped the components to the OODA loop framework to organize our measurements and assist in comprehension of the results. Using this technique, we found the H.264 encoding of the real-time video stream on the drone to be the next bottleneck in the system.

The onboard H.264 encoding currently makes up almost half of the total OODA loop latency. Our experiments used the Parrot Anafi drone, but the video encoding performance of other drones may be better. Certainly, the agility of the system will be handicapped by this component unless we switch to drones that are designed with an emphasis on minimizing latency. First-person view (FPV) drones, for instance, are designed with low latency as a priority as they are intended to be flown manually with high precision in obstacle-dense environments. The results from our experiments emphasize the importance of measuring the latency of the onboard sensing and pre-processing steps when choosing between drones for latency-sensitive applications.

Recognizing the dependence of SteelEagle on the network for offloading, we explored integrating the VOXL system-on-a-chip device as a payload on SteelEagle drones. Utilizing onboard compute requires decoding of the drone's video stream on the VOXL. This requires us to deal with bandwidth challenges when offloading computation to a cloudlet, as sending full-fidelity frames over the network requires high bandwidth availability. We showed that the OODA loop of onboard computation on the VOXL is better than that of the Onion Omega, although the margin is small compared to the overall OODA loop latency.

6.2 Future Work

There are many avenues for future work that will help improve the practicality and versatility of SteelEagle.

- **Reducing system latency.** Exploring new drones for use in SteelEagle that have lower onboard sensing and pre-processing latency, leading to a tighter OODA loop that increases system agility.
- **Resiliency during network degradation.** A remote execution system that takes current network conditions into account when deciding whether to offload computation increases system resiliency, allowing its use in a wider range of applications that involve drone operation in areas with unreliable network coverage. This thesis discussed the feasibility of onboard computation using the VOXL 2, but future work needs to architect system designs that allow the drone to make decisions on where to execute computation based on current conditions.
- **Combining onboard and offloaded execution results.** Onboard and offloaded execution have inherent limitations. Offloaded computation offers higher accuracy but adds latency. Onboard computation supports less accurate models but can potentially offer lower inference time. Approaches that allow combining these two computational results, such as the one explored by Srinivas et al [44] to provide hints from a large model operating on an edge server to the onboard model, can help attain the best of both worlds.
- **Improving benchmarking precision.** We discussed the measurement error in our technique to measure the performance of each stage of the system. Error mainly originates from the refresh rate of the display used to show the Unix timestamp, which can add upwards of 17 ms of error to our latency measurements. Currently, our measurements are much larger in magnitude than the measurement error. As we optimize our OODA loop and achieve latency less than 100 ms, we will require a more precise way to measure the system. One possible approach is to use LEDs connected to a single-board computer, and measure the time taken for the LED to turn on instead of relying on Unix timestamps.

Bibliography

[1] Mahadev Satyanarayanan. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile: Mobile Comp. and Comm.*, 18(4):19–23, January 2015. 1.1

[2] Mahadev Satyanarayanan, Guenter Klas, Marco Silva, and Simone Mangiante. The seminal role of edge-native applications. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 33–40, 2019. 1.1

[3] Z. Chen. *An application platform for wearable cognitive assistance*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2018. 1.1, 5.2.3

[4] J. Flinn. *Cyber foraging: Bridging mobile and cloud computing via opportunistic offload*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, San Rafael, CA, 2012. 1.1

[5] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, page 1–7, New York, NY, USA, 1996. Association for Computing Machinery. 1.1

[6] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009. 1.1, 3.1

[7] Mihir Bala, Thomas Eiszler, Xiangliang Chen, Jan Harkes, James Blakley, Padmanabhan Pillai, and Mahadev Satyanarayanan. Democratizing drone autonomy via edge computing. In *Proceedings of the Eighth ACM/IEEE Symposium on Edge Computing*, SEC '23, page 40–52, New York, NY, USA, 2024. Association for Computing Machinery. 1.2, 3.1, 3.2, 3.2, 4.4

[8] BAE Systems. De Havilland Tiger Moth & Queen Bee. https://www.baesystems.com/en/heritage/de-havilland-tiger-moth---queen-bee. [Accessed 11-08-2024]. 2.1

[9] A Brief History of Drones — iwm.org.uk. https://www.iwm.org.uk/history/a-brief-history-of-drones. [Accessed 11-08-2024]. 2.1

[10] de Havilland DH82B Queen Bee; de Havilland Aircraft Museum — dehavilland-museum.co.uk. https://www.dehavillandmuseum.co.uk/aircraft/de-havilland-dh82b-queen-bee/. [Accessed 11-08-2024]. 2.1

[11] NOVA — Spies That Fly — DH.82B Queen Bee (UK) — PBS — pbs.org. https://

www.pbs.org/wgbh/nova/spiesfly/uavs_05.html. [Accessed 11-08-2024]. 2.1

[12] NOVA — Spies That Fly — Ryan SPA 147 (USA) — PBS — pbs.org. https://www.pbs.org/wgbh/nova/spiesfly/uavs_12.html. [Accessed 11-08-2024]. 2.1

[13] Ferran Giones and Alexander Brem. From toys to tools: The co-evolution of technological and entrepreneurial developments in the drone industry. *Business Horizons*, 60(6):875–884, 2017. 2.1, 2.3

[14] Jürgen Scherer, Saeed Yahyanejad, Samira Hayat, Evsen Yanmaz, Torsten Andre, Asif Khan, Vladimir Vukadinovic, Christian Bettstetter, Hermann Hellwagner, and Bernhard Rinner. An autonomous multi-uav system for search and rescue. In *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, DroNet '15, page 33–38, New York, NY, USA, 2015. Association for Computing Machinery. 2.1, 3.2

[15] Teodor Tomic, Korbinian Schmid, Philipp Lutz, Andreas Domel, Michael Kassecker, Elmar Mair, Iris Lynne Grixa, Felix Ruess, Michael Suppa, and Darius Burschka. Toward a fully autonomous uav: Research platform for indoor and outdoor urban search and rescue. *IEEE Robotics & Automation Magazine*, 19(3):46–56, 2012. 2.1

[16] Jake N. McRae, Christopher J. Gay, Brandon M. Nielsen, and Andrew P. Hunt. Using an unmanned aircraft system (drone) to conduct a complex high altitude search and rescue operation: A case study. *Wilderness & Environmental Medicine*, 30(3):287–290, 2019. 2.1, 3.2

[17] Jan C. van Gemert, Camiel R. Verschoor, Pascal Mettes, Kitso Epema, Lian Pin Koh, and Serge Wich. Nature conservation drones for automatic localization and counting of animals. In Lourdes Agapito, Michael M. Bronstein, and Carsten Rother, editors, *Computer Vision - ECCV 2014 Workshops*, pages 255–270, Cham, 2015. Springer International Publishing. 2.1

[18] Luis F. Gonzalez, Glen A. Montes, Eduard Puig, Sandra Johnson, Kerrie Mengersen, and Kevin J. Gaston. Unmanned aerial vehicles (uavs) and artificial intelligence revolutionizing wildlife monitoring and conservation. *Sensors*, 16(1), 2016. 2.1

[19] Federal Aviation Administration. Drones by the numbers (as of 10/1/24). https://www.faa.gov/node/54496, 2024. Accessed: 2024-11-07. 2.1

[20] DJI. Dji mini 4k. https://store.dji.com/product/dji-mini-4k?vid=166281, 2024. Accessed: 2024-11-08. 2.2

[21] Jacob Stoner. Drone acronym bvlos: Beyond visual line of sight. https://www.flyeye.io/drone-acronym-bvlos/, June 2024. Accessed: 2024-11-22. 2.5

[22] FAA. Small unmanned aircraft systems (uas) regulations (part 107). https://www.faa.gov/newsroom/small-unmanned-aircraft-systems-uas-regulations-part-107, October 2020. Accessed: 2024-11-22. 2.3

[23] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *SIGOPS Oper. Syst. Rev.*, 31(5):276–287, October 1997. 3.1, 5.1.1

[24] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm

in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, 2012. 3.1

[25] Stephen R. Ellis, Katerina Mania, Bernard D. Adelstein, and Michael I. Hill. Generalizeability of latency detection in a variety of virtual environments. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 48(23):2632–2636, 2004. 3.1

[26] Mahadev Satyanarayanan. Edge computing. *Computer*, 50(10):36–38, 2017. 3.1

[27] Gopika Premsankar, Mario Di Francesco, and Tarik Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275–1284, 2018. 3.1

[28] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. The role of edge offload for hardware-accelerated mobile devices. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 22–29, New York, NY, USA, 2021. Association for Computing Machinery. 3.2

[29] Onion.io. Omega2: The $5 linux computer with wi-fi. https://onion.io/omega2/. Accessed: 2024-10-20. 3.2

[30] John R. Boyd. Patterns of conflict. https://www.ausairpower.net/JRB/poc.pdf, 1986. Slides from a presentation by John R. Boyd. 3.3

[31] Oliver Morton. Defence technology: The information advantage. *The Economist*, 335(7918):3, Jun 10 1995. Copyright - Copyright Economist Newspaper Group, Incorporated Jun 10, 1995; Last updated - 2022-11-19; CODEN - ECSTA3. 3.3

[32] Shilpa George, Thomas Eiszler, Roger Iyengar, Haithem Turki, Ziqiang Feng, Junjue Wang, Padmanabhan Pillai, and Mahadev Satyanarayanan. Openrtist: End-to-end benchmarking for edge computing. *IEEE Pervasive Computing*, 19(4):10–18, 2020. 4.2

[33] Parrot. Parrot ANAFI Product Sheet. https://www.parrot.com/assets/s3fs-public/2021-02/anafi-product-sheet-white-paper-en.pdf. Accessed: 2024-11-08. 4.3

[34] Community Project. FFmpeg. https://ffmpeg.org/. 4.3.1

[35] Community Project. OpenCV. https://opencv.org/. 4.3.1

[36] Parrot. PDrAW. https://developer.parrot.com/docs/pdraw/overview.html. 4.3.1

[37] Jason Flinn, SoYoung Park, and M. Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, page 217, USA, 2002. IEEE Computer Society. 5.1.2

[38] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, page 49–62, New York, NY, USA, 2010. Association for Computing Machinery. 5.1.2

[39] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti.

Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 301–314, New York, NY, USA, 2011. Association for Computing Machinery. 5.1.2

[40] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. The case for offload shaping. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, page 51–56, New York, NY, USA, 2015. Association for Computing Machinery. 5.1.3

[41] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017. 5.2

[42] Nhut-Minh Ho and Weng-Fai Wong. Exploiting half precision arithmetic in nvidia gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017. 5.2

[43] Mihir Bala, Aditya Chanana, Xiangliang Chen, Qifei Dong, Thomas Eiszler, Jingao Xu, Padmanabhan Pillai, and Mahadev Satyanarayanan. The ooda loop of cloudlet-based autonomous drones. In *Proceedings of the Ninth ACM/IEEE Symposium on Edge Computing*, To appear in SEC '24, 2025. 5.5

[44] Vidya Srinivas, Malek Itani, Tuochao Chen, Sefik Emre Eskimez, Takuya Yoshioka, and Shyamnath Gollakota. Knowledge boosting during low-latency inference, 2024. 6.2