NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

INTER-INSTITUTIONAL POSTGRADUATE MASTERS PROGRAM
”Space Technologies, Applications and SeRvices - STAR”

MSc THESIS

Space-Age DevOps: Integrating ESA’s NanoSat MO Framework with Kubernetes for Seamless Satellite Operations

Dimitrios G. Fotiou

Supervisors: Christos Tsigkanos, Professor NKUA

ATHENS

November 2025

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΙΔΡΥΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΔΙΑΣΤΗΜΙΚΕΣ ΤΕΧΝΟΛΟΓΙΕΣ, ΕΦΑΡΜΟΓΕΣ και ΥΠΗΡΕΣΙΕΣ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

DevOps στην εποχή του Διαστήματος: Συνδυασμός του NanoSat MO framework του Ευρωπαϊκού Οργανισμού Διαστήματος με το Kubernetes για απρόσκοπτες δορυφορικές αποστολές

Δημήτριος Γ. Φωτίου

Επιβλέποντες: Χρήστος Τσίγκανος, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΝΟΕΜΒΡΙΟΣ 2025

MSc THESIS

Space-Age DevOps: Integrating ESA's NanoSat MO Framework with Kubernetes for Seamless Satellite Operations

Dimitrios G. Fotiou
S.N.: 7115172100030

SUPERVISORS: **Christos Tsigkanos**, Professor NKUA

EXAMINATION COMMITTEE: **Christos Tsigkanos**, Professor NKUA
Stavros Kolios, Professor NKUA
Nektarios Kranitis, Professor NKUA

Examination Date: 1 December, 2025

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

DevOps στην εποχή του Διαστήματος: Συνδυασμός του NanoSat MO framework του Ευρωπαϊκού Οργανισμού Διαστήματος με το Kubernetes για απρόσκοπτες δορυφορικές αποστολές

Δημήτριος Γ. Φωτίου
Α.Μ.: 7115172100030

ΕΠΙΒΛΕΠΟΝΤΕΣ: Χρήστος Τσίγκανος, Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: Χρήστος Τσίγκανος, Καθηγητής ΕΚΠΑ
Σταύρος Κολιός, Καθηγητής ΕΚΠΑ
Νεκτάριος Κρανίτης, Καθηγητής ΕΚΠΑ

Ημερομηνία Εξέτασης: 1 Δεκεμβρίου 2025

ABSTRACT

The ever-evolving field of space exploration demands innovative approaches to satellite mission operations. While developments within the “New Space” and small satellites era offer unique advantages, including reduced costs and faster development cycles, they also present operational challenges that require system modernization and increased automation at par with advances in general software engineering. We address the operational limitations inherent in contemporary small satellite missions, which, despite leveraging standardized frameworks like the European Space Agency’s (ESA) NanoSat MO Framework (NMF) under the Consultative Committee for Space Data Systems (CCSDS) protocols, lack the agility and automation of modern terrestrial systems. In particular, this thesis presents a pioneering methodology that leverages the power of Kubernetes container orchestrator to transform satellite missions into agile and efficient endeavors, enabling edge computing capabilities directly in space. Driving principles entail modularizing functionalities and establishing standardized guidelines for software development and deployment.

To this end, this thesis presents a detailed methodology for integrating NMF with the Kubernetes container orchestration platform, thereby enabling a modern DevOps-driven approach to satellite mission operations. This transition represents not only an operational enhancement but also an architectural shift, from a traditional Service Oriented Architecture (SOA) to a fully containerized, microservice-based model.

Results obtained demonstrate a paradigm shift in satellite operations. Kubernetes-powered space missions utilizing the NanoSat MO Framework could exhibit improved flexibility, scalability, observability and reliability, all while complying with CCSDS standards. Importantly, this transformation also enables edge computing capabilities onboard satellites, allowing data to be processed in space rather than relying solely on ground-based infrastructure. This reduces communication latency, optimizes bandwidth usage, and supports more autonomous and resilient mission operations. In context, this thesis envisions a space-age DevOps era; the gap between traditional satellite operations and modern DevOps practices can be bridged, ushering in a new era of space exploration.

SUBJECT AREA: Space applications and automations

KEYWORDS: ESA, DevOps, NMF, Kubernetes, OPSSAT, satellites, edge computing

ΠΕΡΙΛΗΨΗ

Ο συνεχώς εξελισσόμενος τομέας της εξερεύνησης του διαστήματος απαιτεί καινοτόμες τεχνικές προσεγγίσεις στις δορυφορικές αποστολές. Ενώ οι εξελίξεις στην εποχή του «Νέου Διαστήματος» και των μικροδορυφόρων προσφέρουν μοναδικά πλεονεκτήματα, όπως μειωμένο κόστος και ταχύτερους κύκλους ανάπτυξης, παρουσιάζουν επίσης λειτουργικές προκλήσεις που απαιτούν εκσυγχρονισμό συστημάτων και αυξημένη αυτοματοποίηση στο ίδιο επίπεδο με τα επίγεια υπολογιστικά συστήματα. Εξετάζουμε τους λειτουργικούς περιορισμούς που ενυπάρχουν στις σύγχρονες αποστολές μικρών δορυφόρων, οι οποίες, παρά την αξιοποίηση τυποποιημένων προγραμματιστικών εργαλειών όπως το NanoSat MO Framework (NMF) του Ευρωπαϊκού Οργανισμού Διαστήματος (ESA) στο πλαίσιο των πρωτοκόλλων της Συμβουλευτικής Επιτροπής για τα Διαστημικά Συστήματα Δεδομένων (CCSDS), δεν διαθέτουν την ευελιξία και τον αυτοματισμό των σύγχρονων επίγειων συστημάτων. Συγκεκριμένα, η παρούσα διπλωματική παρουσιάζει μια πρωτοποριακή μεθοδολογία που αξιοποιεί τη δύναμη του Kubernetes για να μετατρέψει τις δορυφορικές αποστολές σε ευέλικτες και αποτελεσματικές προσπάθειες, καθιστώντας δυνατό το edge computing απευθείας στο διάστημα. Βασικές αρχές αποτελούν ο καταμερισμός των προγραμματιστικών λειτουργιών και η θέσπιση τυποποιημένων κατευθυντήριων γραμμών για την ανάπτυξη και την ανάπτυξη λογισμικού.

Για τον σκοπό αυτό, η παρούσα εργασία παρουσιάζει μια λεπτομερή μεθοδολογία για τον συνδυασμό του NMF με την πλατφόρμα Kubernetes για container orchestration, επιτρέποντας έτσι μια σύγχρονη προσέγγιση στο διάστημα που βασίζεται στις αρχές του DevOps. Αυτή η μετάβαση αντιπροσωπεύει όχι μόνο μια λειτουργική βελτίωση αλλά και μια αρχιτεκτονική μετατόπιση, από μια παραδοσιακή Service Oriented Αρχιτεκτονική σε ένα πλήρως containerized microservice μοντέλο.

Τα αποτελέσματα που ελήφθησαν καταδεικνύουν μια παραδειγματική αλλαγή στις λειτουργίες δορυφόρων. Οι διαστημικές αποστολές με ενσωματωμένο το Kubernetes και το NanoSat MO Framework επιδεικνύουν βελτιωμένη ευελιξία, επεκτασιμότητα, δυνατότητα διαχείρισης και αξιοπιστία, ενώ παράλληλα συμμορφώνονται με τα πρότυπα CCSDS. Είναι σημαντικό ότι αυτός ο μετασχηματισμός επιτρέπει την επεξεργασία δεδομένων στο διάστημα, κατά τα πρότυπα του edge computing, αντί να βασιζόμαστε αποκλειστικά σε επίγειες υποδομές. Αυτό μειώνει την καθυστέρηση στις επικοινωνίες, βελτιστοποιεί τη χρήση εύρους ζώνης και υποστηρίζει πιο αυτόνομες και ανθεκτικές λειτουργίες αποστολών. Στο πλαίσιο αυτό, η παρούσα διπλωματική εργασία οραματίζεται την εφαρμογή των αρχών του DevOps στην εποχή του διαστήματος. Το χάσμα μεταξύ των παραδοσιακών λειτουργιών δορυφόρων και των σύγχρονων πρακτικών DevOps μπορεί να γεφυρωθεί, εγκαινιάζοντας μια νέα εποχή εξερεύνησης του διαστήματος.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Διαστημικές εφαρμογές και αυτοματισμοί

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: ESA, DevOps, NMF, Kubernetes, OPSSAT, δορυφόροι, edge computing

I dedicate this thesis to my family, who have always supported me wholeheartedly to get this far. Special mention to my brother Nikos, with whom we have shared many common steps academically, professionally and in the struggle of life. He was a beacon for me throughout the completion of this work. I also extend my heartfelt gratitude to my partner, Yiouli, and to all my dear friends for the beautiful moments we share, which give me strength and courage during difficult times.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Christos Tsigkanos, for his guidance and support throughout this thesis. I also thank the Department of Informatics and Telecommunications for providing a constructive academic environment.

CONTENTS

LIST OF FIGURES

PREFACE

The Earth is our courtyard, but Space is immense. Somewhere, something incredible is waiting to be known

This always intrigued me in bringing my ”down to Earth” technology knowledge to Space.

Hopefully, my postgraduate master’s program in NKUA proved to be a fundamental first step for this. This thesis is a fruit of this journey.

1. INTRODUCTION

1.1 Problem description

In recent years, the field of space exploration has been revolutionized by the rise of small satellites, often termed ”CubeSats” or ”NanoSats”. These compact and cost-effective spacecraft have made space more accessible than ever, leading to a surge in launches by governments, institutions, companies, and research groups. However, while building and launching these satellites has become easier, managing their operations once in orbit presents significant challenges. The software systems traditionally used for space missions are often rigid, require a lot of manual oversight, and have not kept pace with the rapid advances seen in software development on Earth.

To this end, this thesis was conceived around existing, but also rising, problems of space aviation, particularly from the software perspective. As the scope of space aviation expands, the computational needs get more complex. At the same time, space projects adopt a different life cycle with the use of CubeSats, enabling cheaper and more frequent launches. This opens up new architectural possibilities, such as satellite clusters - constellations of small, cooperative satellites capable of distributed computation. However, space software development still relies heavily on traditional strategies and has yet to adopt modern software engineering practices.

In essence, the proposed solution to these operational limitations is bridging the gap between space systems and modern, efficient software practices from terrestrial industries, a movement known as ”DevOps”. The present thesis tries to put together the missing pieces of the modern space software puzzle and contribute with real world examples to the evolution of the field. In the following, we provide a more in-detail presentation of the context this thesis addresses, as well as the main challenges faced.

1.1.1 Background and context

Space exploration has always been a testament to human ingenuity and ambition. Over the decades, our reach into the cosmos has expanded, and technological advancements have enabled increasingly complex missions. Among these missions, small satellites, or ”CubeSats,” have emerged as a revolutionary platform for space research, Earth observation, and telecommunications. Their relatively low cost and rapid development cycles make them an attractive choice for governmental, commercial and research entities.

However, the deployment and management of small satellite missions pose unique challenges. Traditional methods of mission operations are often characterized by rigid protocols, limited scalability, and high operational costs. These limitations have prompted the exploration of innovative approaches that align with contemporary practices in software development and operations, commonly referred to as DevOps.

The Consultative Committee for Space Data Systems (CCSDS) plays a pivotal role in the

space industry by developing standards for space data and information systems. These standards are designed to ensure interoperability, reliability, and efficiency in space missions. The adoption of CCSDS standards by space agencies and organizations worldwide underscores their significance in the field.

In this context, the integration of Kubernetes, a leading container orchestration platform, with NanoSat MO Framework, a sophisticated software framework designed for small satellites that implements CCSDS principles [1], presents an exciting opportunity. Kubernetes, known for its ability to manage application deployments at scale, can bring the principles of DevOps to the world of satellite missions. NanoSat MO Framework, with its focus on CCSDS Mission Operations services, provides a foundation for building highly efficient and standardized satellite operations.

The central tenet of this work is essentially to redesign NMF's architecture, transforming it from a traditional, monolithic system into a modern one based on microservices, as small, independent, and containerized components orchestrated by Kubernetes. This innovative approach intends to bring the automation and agility of DevOps to satellite operations.

1.1.2 Challenges of the study

The purpose of this study is multifaceted and addresses several significant challenges in the field of space exploration and small satellite missions. These challenges are rooted in the absence of a precedent for contributions to the NanoSat MO Framework, the limited exposure of space missions to DevOps methodologies, the distinct division between ”upstream” and ”downstream” segments in space missions, the scarcity of comprehensive documentation, and the relatively small NanoSat MO Framework community. In particular, we consider the main challenges to entail the following:

1. **Lack of precedent in contributions to NanoSat MO Framework**

   One of the primary challenges this study seeks to address is the absence of a precedent for contributions to the NanoSat MO Framework. While the NanoSat MO Framework has garnered attention as a valuable resource for small satellite missions by the European Space Agency, there remains a lack of documented contributions and integration efforts. This void impedes the development and adoption of NanoSat MO Framework as a comprehensive and extensible solution for modern space missions.

2. **Limited exposure of space missions to DevOps methodologies**

   In the domain of space exploration, the adoption of DevOps practices has been relatively limited. Space missions traditionally resorted to lower level software implementations, i.e., focusing on embedded systems, and lacked large scale automation. Recent developments across multiple technological fronts have made it possible to include higher-level compute capabilities onboard, particularly as compute payloads.

3. **Facilitating collaboration between Upstream and Downstream domains**

   Within the realm of space missions, a noticeable division exists between the upstream and downstream sectors. The former concentrates on hardware development, while the latter primarily serves as service consumers. This can become an obstacle for the adoption of a unified approach and research on new software technologies and lead to slower progress in space software. This study seeks to facilitate collaboration and synergy between these traditionally separate entities by highlighting the significance of end to end DevOps integration with space mission management.

4. **Scarcity of comprehensive documentation**

   Another challenge facing the space community is the relative scarcity of comprehensive documentation for the NanoSat MO Framework and its integration with modern technologies like Kubernetes. Existing documentation may be fragmented, incomplete, or outdated, hindering the adoption and understanding of these technologies. This study seeks to contribute to a more complete set of documentation, making it easier for space agencies, researchers, and developers to utilize the NanoSat MO Framework effectively.

5. **Leveraging a relatively small community**

   Lastly, the NanoSat MO Framework community remains relatively small compared to other software development communities. This study acknowledges the importance of growing this community to facilitate knowledge sharing, collaboration, and the collective improvement of NanoSat MO Framework.

   In summary, the purpose of this study is to address these challenges by introducing the integration of DevOps practices and Kubernetes into the NanoSat MO Framework, fostering collaboration between the upstream and downstream aspects of space missions, contributing to comprehensive documentation, and strengthening the NanoSat MO Framework community. By doing so, this study aims to advance the capabilities and efficiency of small satellite missions and contribute to the ongoing evolution of space exploration.

1.2 Running example

The primary objective of this thesis is to develop a novel Earth-to-Space computational continuum, aimed at enabling seamless and dynamic computation across distributed space platforms. This is achieved by implementing a clustered computing architecture, designed to facilitate the coexistence and interaction of multiple space-related software systems, regardless of their physical location or platform constraints.

Such computational power is increasingly feasible in space nowadays, moving away from embedded systems towards higher-level, more abstract (and more complex) capabilities on board, including GPUs, TPUs, CPUs or special purpose chips.

The central concept is the deployment of a cluster of CubeSats, each operating as a Kubernetes node. These satellite nodes are interconnected, forming a loosely coupled distributed system. A master node maintains an indirect connection to Earth-based edge infrastructure, allowing dynamic coordination and orchestration of computational tasks. Multiple interconnected applications running as a payload upon each satellite could be based in completely different programming languages and frameworks, such as Java, Python and F‘ prime. Based on relative distance and communication feasibility, the Cube-Sat nodes can further extend connectivity to deep space devices. These deep space components may or may not be Kubernetes-compatible, and their integration into the continuum is handled in a flexible manner. This way, an “elastic arm” is created, a flexible and distributed architecture of compute nodes, extending from Earth-based infrastructure through CubeSats in orbit, and potentially reaching out to deep space devices.

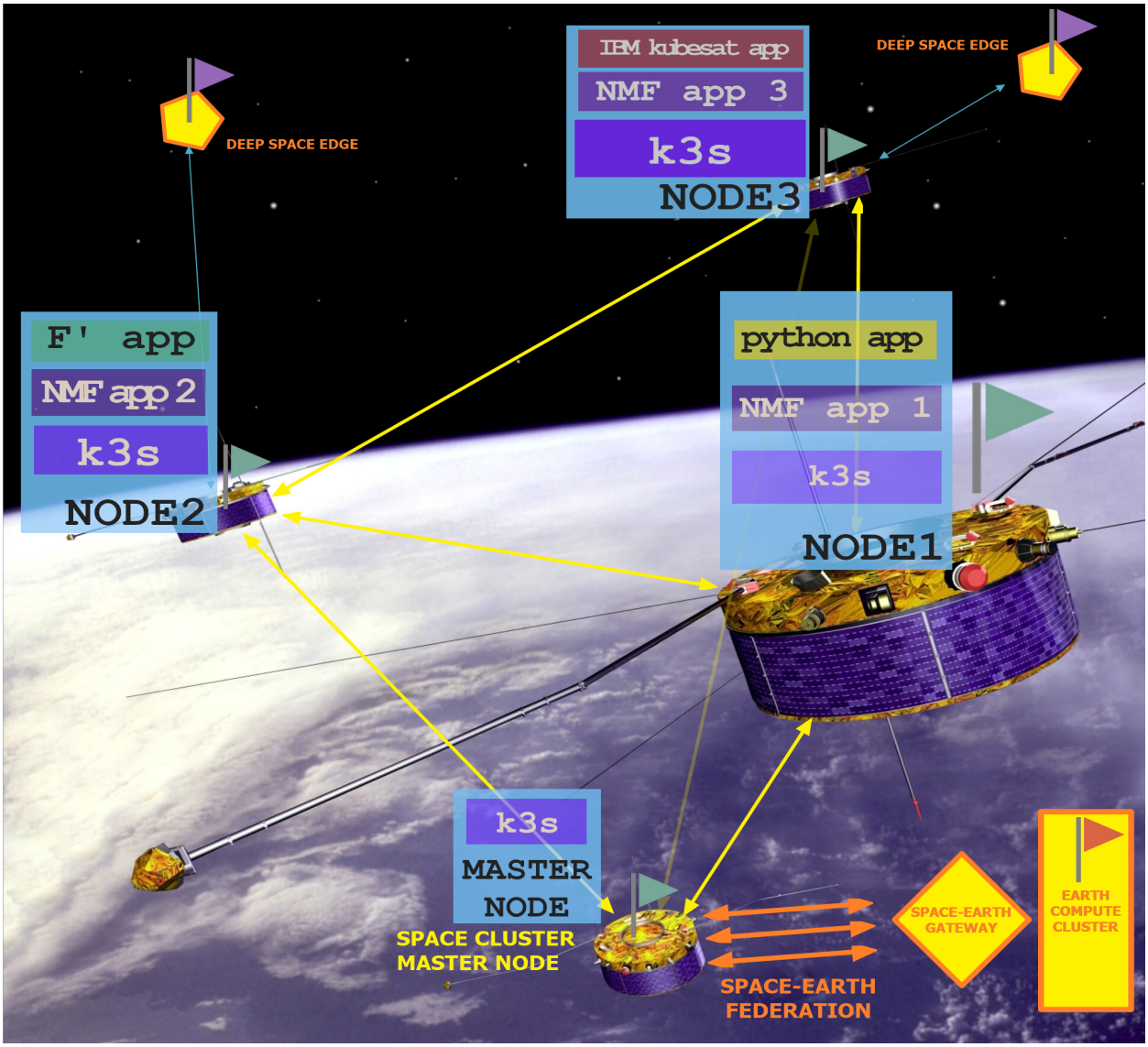

Figure 1.1: Space Edge clustered Kubernetes architecture

To bring a sample of this architecture to life, a well-known software framework already adopted in the space ecosystem is selected for managing onboard services and applications, the European Space Agency's (ESA) NanoSat MO Framework (NMF). NMF is primarily based on Service Oriented Architectures and was not originally designed for full-fledged distributed operations. So it is necessary to break its functionality and services into pieces, containerize them and design a Kubernetes-based deployment solution. This adaptation is essential to support scalable and fault-tolerant operations within a clustered satellite environment.

The OPS-SAT satellite mission is chosen as a proof-of-concept model for this implementation due to its relevance, system requirements, and operational limitations. In order to prototype the proposed system and evaluate its feasibility, a Raspberry Pi device with comparable resources to the OPS-SAT onboard computer is utilized in the context of this thesis. This serves as a proof of concept, facilitating the deployment of the Kubernetes-based NMF solution under constrained hardware conditions. We note that from a software perspective, hardware can be abstracted, and its relevance to the key tenets of this thesis is minimal. In essence, we require a compute environment, capable of accommodating our workload specs in terms of resources and also have access to external devices used for sensing (for example a camera).

1.3 Thesis contributions

This thesis is taking place in a rather uncharted territory. Thus, some contributions can be fundamental, while others might have less impact in the current time frame, but will pave the ground for future added value.

1.3.1 Offer a new architectural proposal for the framework using microservices

The foundational technical contribution from this thesis is the architectural redesign of the Nanosat MO Framework from Service Oriented Architecture (SOA) to microservice architecture [2]. While SOA provided a foundation for modularizing business logic into reusable services, it often resulted in tightly coupled components, complex integration layers and semi-independent deployments. Each application could be deployed, undeployed and managed by the Supervisor. By adopting microservices, the system is decomposed into independently deployable, loosely coupled services. Every application along with the supervisor is now deployed, undeployed and managed by a technology agnostic platform, Kubernetes. This transition enables greater scalability, resilience, and agility, allowing teams to develop, deploy, and maintain services autonomously. The microservices architecture also facilitates continuous delivery and rapid innovation, addressing many of the limitations inherent in monolithic or SOA-based systems.

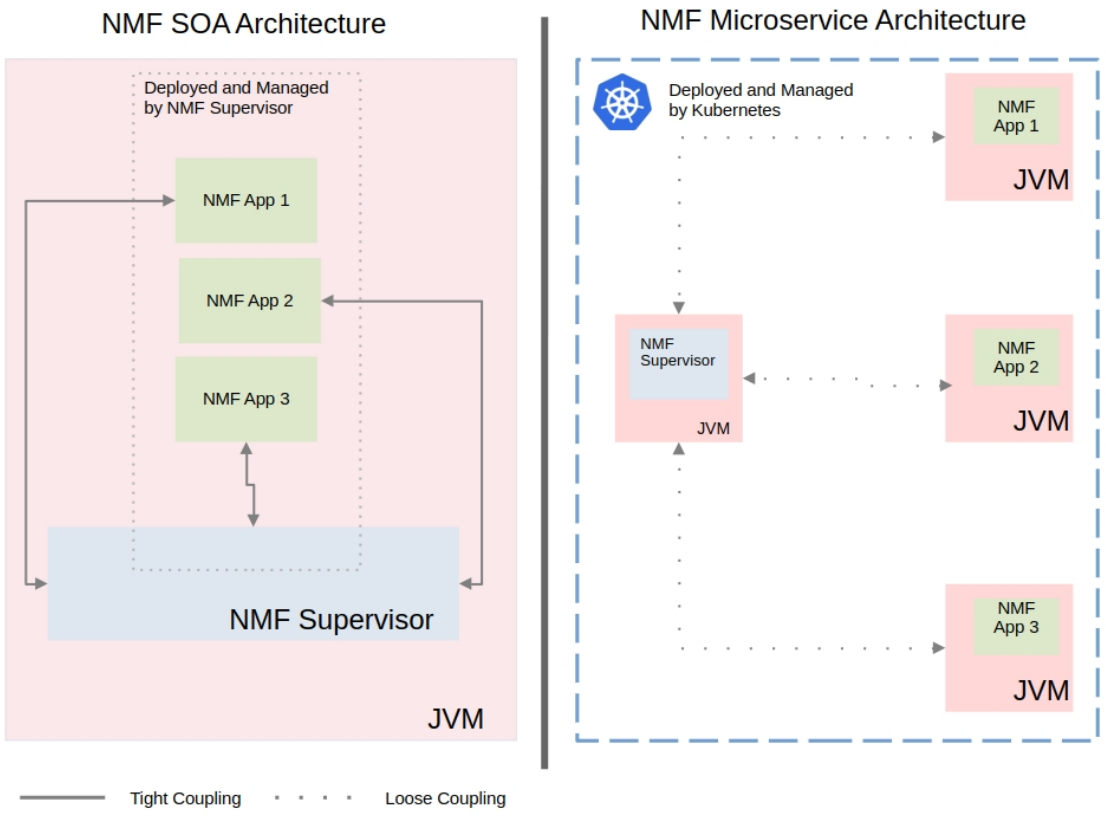

Figure 1.2: NMF SOA & Microservice Architecture Comparison

1.3.2 Containerization of Nanosat MO Framework microservices

The architectural shift to microservices also introduces the important role of containerization. In the previous NMF deployment model, all Java libraries and services were executed within a single JVM instance. While this approach somewhat minimized runtime overhead, it also created challenges, such as classpath conflicts, version-locking of libraries, and the inability to tune JVM parameters for individual components. Containerization addresses these challenges by enabling each NMF service to operate in its own isolated runtime. This design allows services to run with tailored Java runtimes, JRE or JDK, while also supporting the concurrent use of different Java versions (e.g., Java 8 for legacy compatibility and Java 17 for newer features). In addition, container-level isolation eliminates dependency collisions and provides a mechanism for fine-grained JVM tuning, including heap sizing, garbage collection strategies, and startup optimizations, all configured per service rather than globally [3][4]. Beyond the Java runtime itself, containerization also decouples services from a single operating system baseline. Each container can run on a distribution and version best suited to its workload, such as Ubuntu for general-purpose services or

Alpine Linux for lightweight, memory-constrained deployments. Resource allocation becomes more precise as well, with CPU shares, memory limits, and I/O quotas enforced at the container level [5]. This ensures predictable performance and efficient utilization of underlying hardware, even when services have heterogeneous workload characteristics. Overall, containerization transforms NMF application deployment by combining runtime flexibility, OS-level independence, and fine-grained resource control. These properties not only reduce operational friction but also enhance maintainability and scalability in a microservices-based environment.

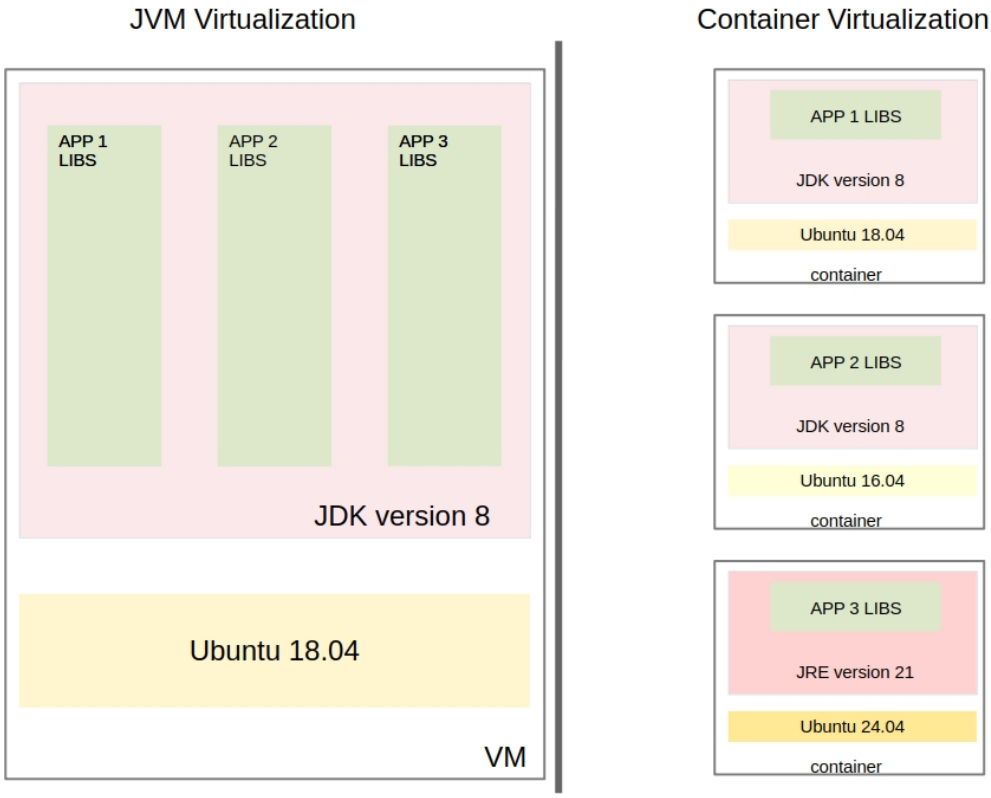

Figure 1.3: Single JVM & Container Virtualization Comparison

1.3.3 Documenting network configurations for containerized environment

During the transition from a setup with all services hosted on a single machine to a distributed environment where services operate independently, difficulties arise in configuring the framework's network connections. Although various configuration options existed beforehand, they were not clearly documented, requiring thorough examination of the source code to determine the appropriate settings. To address this limitation, an alternative ex-

ample configuration is developed and documentation is provided to facilitate its use [6].

1.3.4 Provide the first working example of Nanosat MO Framework running into kubernetes in microservices

As part of this thesis the first documented deployment of Nanosat MO Framework apps in Kubernetes (k3s) is accomplished. We leverage Kubernetes manifests in order to foster the deployment of the apps and maintain their functionality. We also handle the special networking that Nanosat MO Framework requires within the design of our Kubernetes infrastructure. Finally, access to devices (such as cameras and sensors) is granted from within each pod-app by using and parameterizing Kubernetes Device plugin.

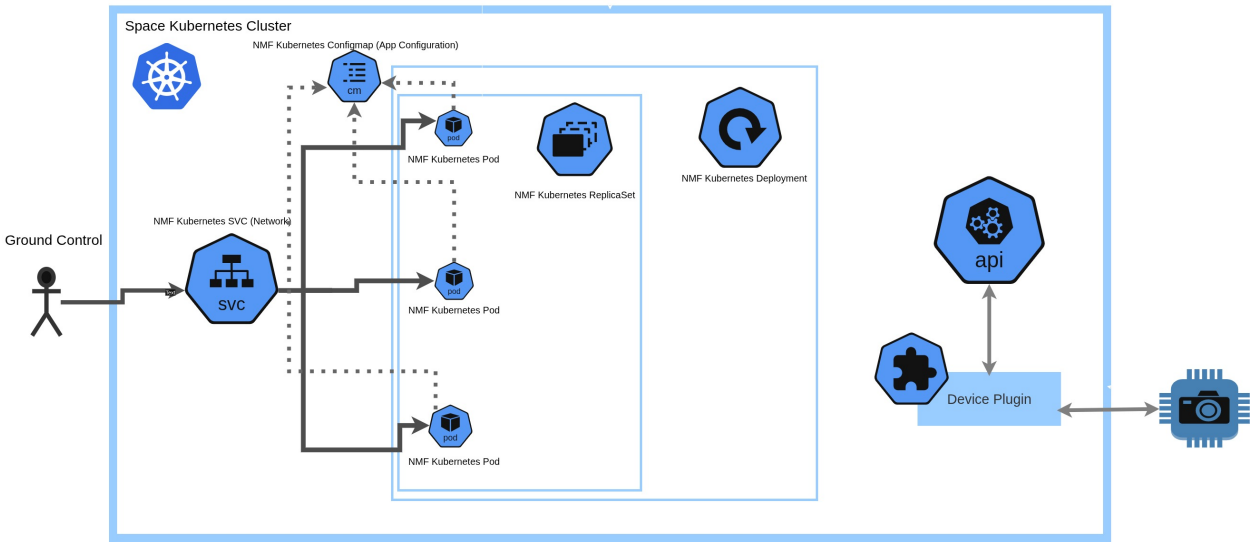

Figure 1.4: NanoSatMO service integration with Kubernetes

1.3.5 Standardized production deployment via Helm Packaging

A Helm Chart package is designed and implemented for the Nanosat MO Framework services. This contribution transforms raw Kubernetes manifests into a repeatable, version-controlled Infrastructure-as-Code (IaC) solution. It establishes a standardized deployment pattern, significantly improving operational velocity, maintainability, and management of complex services. Crucially, the Helm design addresses service dependencies by abstracting and segregating shared components, such as the central Supervisor service, enabling efficient, modular deployment across environments.

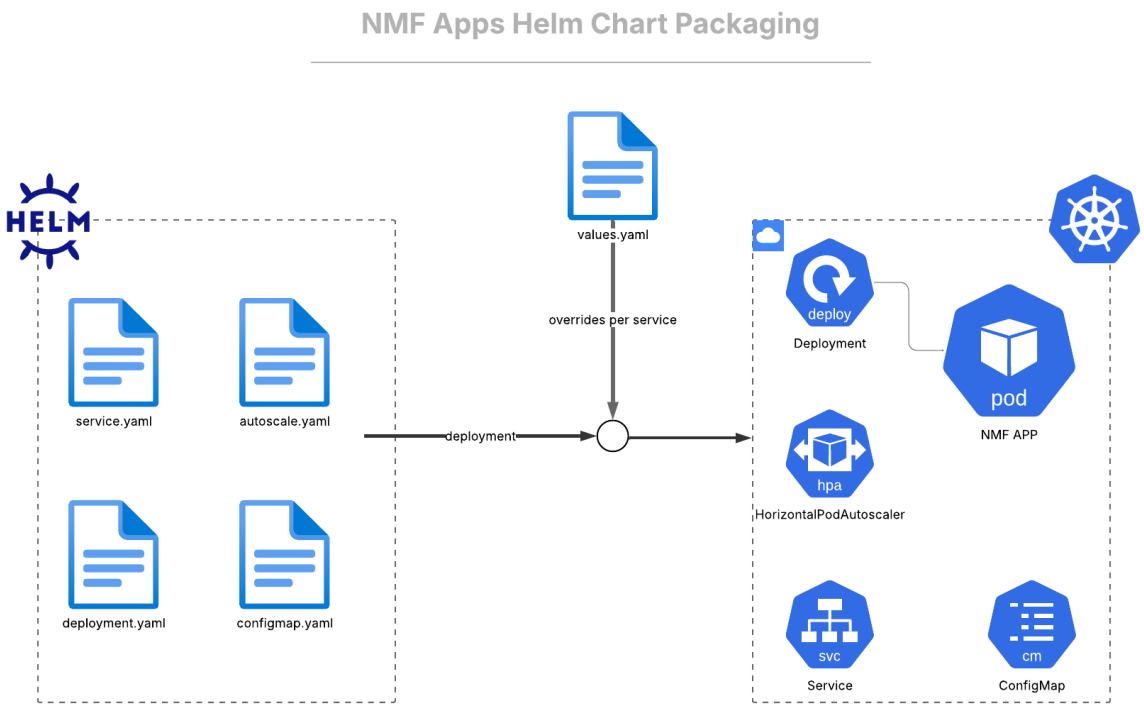

Figure 1.5: NMF Apps Helm Chart Packaging

1.3.6 Enhanced local development velocity with container tooling

To maximize developer productivity and onboard new contributors quickly, a complete Docker and Docker Compose environment is provided for use in local development workstations. This tooling simplifies the entire setup process, allowing the full Nanosat MO Framework stack to be initialized with a single command and requiring only a Docker installation regardless of operating system. This effort validates the work's practical utility and establishes a modern, low-friction entry point for local code development and debugging.

1.3.7 Novel CI/CD strategy for space deployment resilience

We devised and demonstrated a resilient Continuous Integration/Continuous Deployment (CI/CD) pipeline tailored for the unique challenges of space and edge computing. Utilizing Jenkins on a lightweight microK8s cluster, this pipeline specifically addresses the

hypothesis of deploying services in highly latency-bound and connectivity-disrupted environments. The core innovation is the pre-deployment image staging mechanism, which bypasses the unreliable Container Runtime Interface (CRI) download process by copying the new image directly to the node and uploading it to a local registry. This method ensures successful and deterministic deployments despite intermittent connectivity issues.

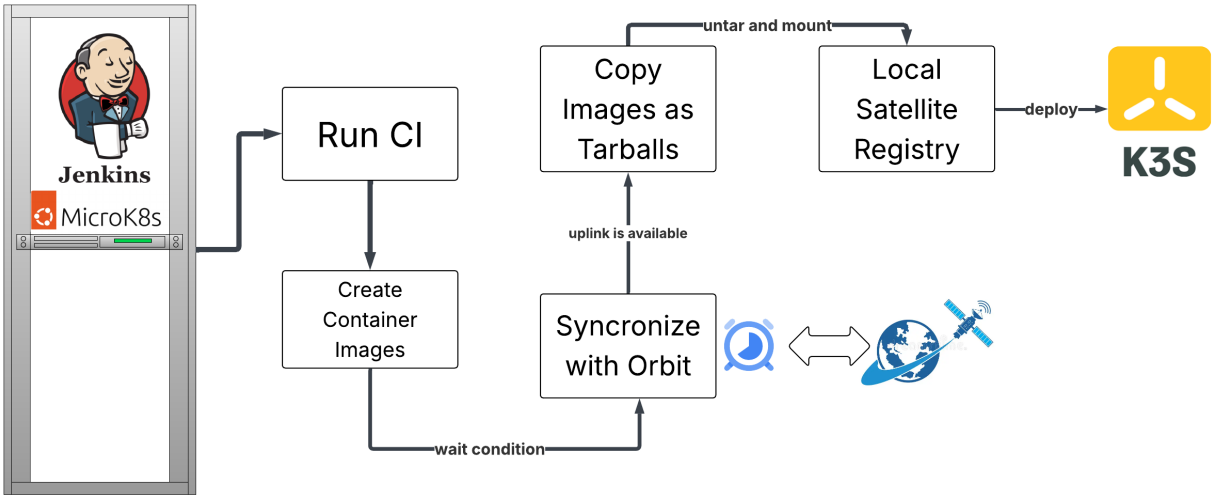

Figure 1.6: Jenkins CI/CD pipeline for NMF apps

1.3.8 Camera adapter for RaspberryPi & image processing app

An NMF camera adapter/platform service is implemented to enable seamless integration and operation of imaging payloads on Raspberry Pi-based nanosatellite hardware. The service provides standardized interfaces for camera initialization and image acquisition.

A new NMF space application was developed to preprocess images acquired in orbit using various techniques, such as Sobel filtering and K-means unsupervised clustering. This application demonstrates the advantages of space edge computing by performing onboard data reduction, extracting and downlinking only the most relevant image features, thereby optimizing bandwidth usage and improving mission efficiency.

2. RELATED WORK

The initial phases of this research are guided by a comprehensive review of existing work in the fields of space systems, edge computing, DevOps and cloud infrastructure. The combination of these technological fields has garnered significant attention in recent years, judging by the quantity and quality of the related real world projects. This chapter presents key initiatives and frameworks that form the foundation of this thesis, highlighting their relevance, influence, or contrast with the final approach.

2.1 K3s (SUSE RGS, Hypergiant)

Description K3s is a highly available, certified Kubernetes distribution designed for production workloads in unattended, resource-constrained, remote locations or inside IoT appliances.

A notable adoption of k3s is found in the collaboration between SUSE Rancher Government Services (SUSE RGS) and Booz Allen Hamilton association, detailed in their report "Advancing the Tactical Edge with K3s and SUSE RGS." This partnership focuses on deploying K3s, a lightweight Kubernetes distribution, to enhance edge computing capabilities in remote and resource-constrained environments. By leveraging K3s, their solution allows battalions to make real-time, data-driven decisions which dramatically improve operational outcomes and increase the probability of mission success [7].

Another initiative involves SUSE RGS collaborating with Hypergiant Industries to deploy Kubernetes clusters on military satellites. This project utilizes K3s to automate the creation of repeatable workloads in orbit, addressing challenges like limited connectivity and the need for rapid software updates in space environments [8].

Evaluation These projects established k3s as a suitable lightweight and reliable Kubernetes flavour both in space and edge environments.

2.2 KubeEdge (Tiansuan constellation)

Description KubeEdge is an open source system for extending native containerized application orchestration capabilities to hosts at Edge. It is built upon Kubernetes and provides fundamental infrastructure support for network, application deployment and metadata synchronization between cloud and edge [9].

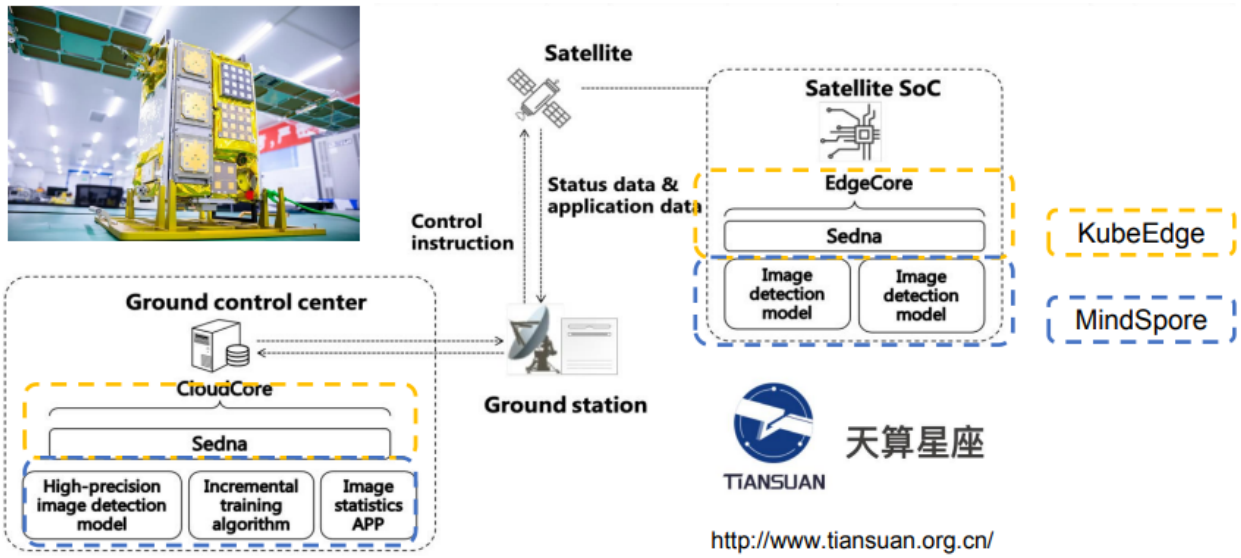

Figure 2.1: Tiansuan constellation architecture integrated with KubeEdge & MindSpore [10]

In 2022, at the main forum of KubeCon and CloudNativeCon European Summit, a high-profile conference in the cloud native field, Huawei published the first open source cloud native AI satellite application solution [10]. They had already launched Tiansuan-1 satellite in September 2021, joining the open satellite research Tiansuan constellation, shipped with CNCF KubeEdge and MindSpore to address the challenges of:

- On-orbit computation to minimize orbit-earth communication for better life span
- Edge-cloud real time inference and incremental deep learning for SAR type task

Evaluation Traditional satellites are closed architectures, with satellites customized for specific tasks, payloads tailored for specific satellites, and software designed for specific payloads. One satellite, one mission, one launch defining its entire lifecycle. This closed system architecture necessarily leads to hardware and software incompatibility, inability to interchange and reuse, and very slow technological iteration.

To address these issues, satellite manufacturers have proposed software-defined satellites, a new type of intelligent satellite centered on space-based computing. These satellites adopt an open system architecture that supports plug-and-play payloads and on-demand application loading.

The KubeEdge project has demonstrated the feasibility of this new cloud-native edge computing paradigm in space.

2.3 KubeSat (IBM Space Tech Team)

Description KubeSat is an open-source project, developed by IBM Space Tech Team, for building a cognitive, autonomous framework for satellite constellations and swarms. It provides the framework needed to develop and operate tasks to be performed on Satellite. Also, it allows for the simulation and optimization of multi-satellite communications. [11]

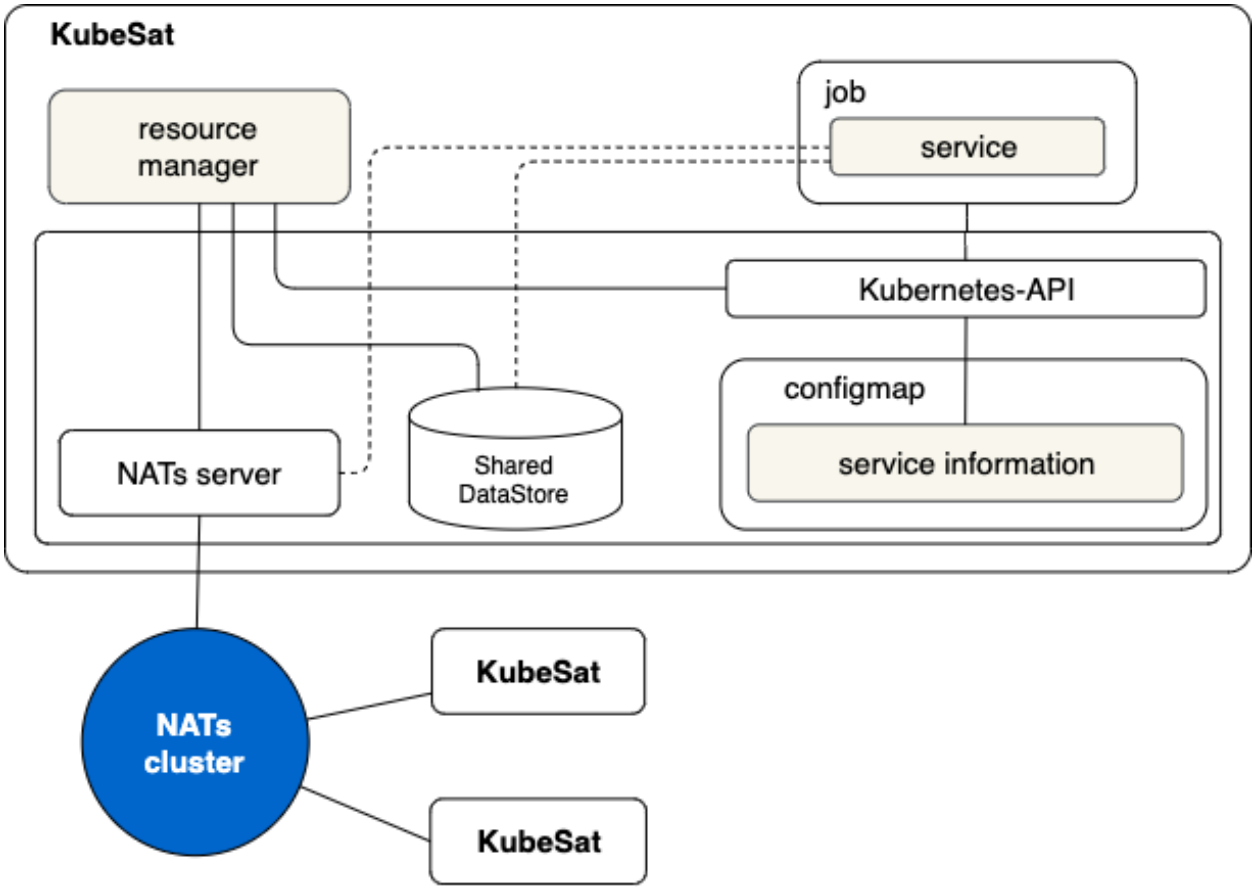

Figure 2.2: IBM KubeSat internal structure [12]

It utilizes the Kubernetes API along with NATs server to cluster different KubeSats running on different satellites.

Evaluation This Github project [12] was archived in 2024 and the stack has not grown into a mature production-ready state. However, it laid some solid foundations for the adoption of DevOps approach, especially for satellite constellations.

2.4 Akri Project

Description Akri is a Cloud Native Computing Foundation (CNCF) Sandbox project. It lets you easily expose heterogeneous leaf devices (such as IP cameras and USB devices) as resources in a Kubernetes cluster, while also supporting the exposure of embedded hardware resources such as GPUs and FPGAs. Akri continually detects nodes that have access to these devices and schedules workloads based on them.

At the edge, there are a variety of sensors, controllers, and MCU class devices that are producing data and performing actions. For Kubernetes to be a viable edge computing solution, these heterogeneous "leaf devices" need to be easily utilized by Kubernetes clusters. However, many of these leaf devices are too small to run Kubernetes themselves. Akri is an open source project that exposes these leaf devices as resources in a Kubernetes cluster. [13]

Akri's solution consists of five components: two custom resources, Discovery Handlers, an Agent (device plugin implementation), and a custom Controller. The first custom resource, the Akri Configuration, is where you name the leaf device that it should look for. At this point, Akri finds it! Akri's Discovery Handlers look for the device and inform the Agent of discovered devices. The Agent then creates Akri's second custom resource, the Akri Instance, to track the availability and usage of the device. Having found your device, the Akri Controller helps you use it. It sees each Akri Instance (which represents a leaf device) and deploys a ("broker") Pod that knows how to connect to the resource and utilize it.

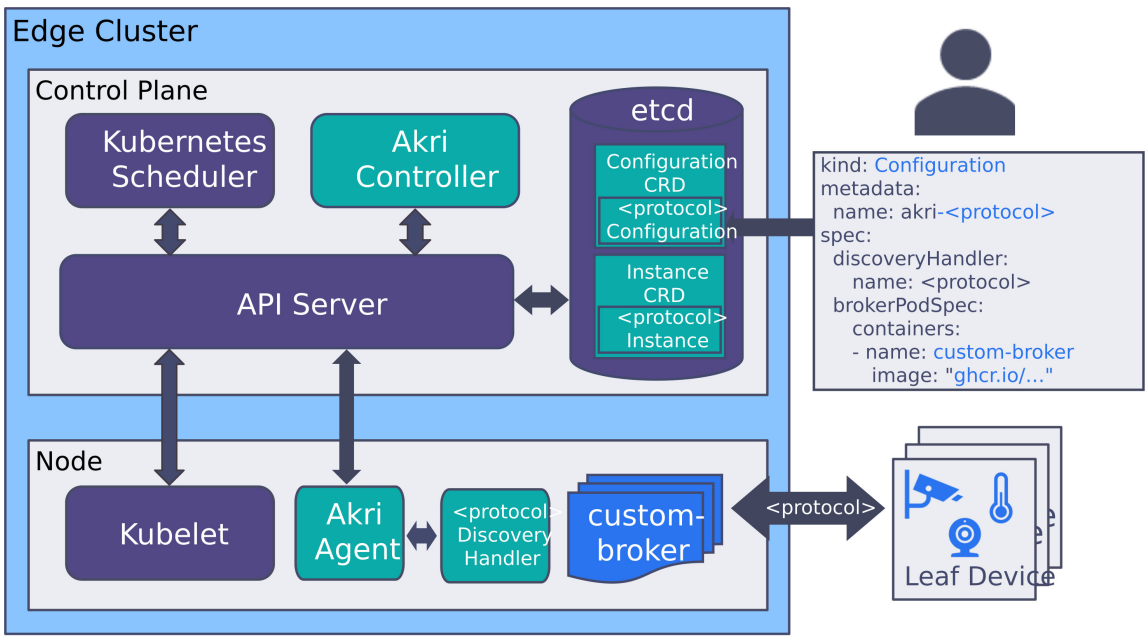

Figure 2.3: Akri Project as device discovery extension to Kubernetes [13]

Evaluation Akri is highly useful in making IoT Devices accessible to Edge Kubernetes Clusters. [14] There is no prior adoption of the Akri project in space, but for sure it would be really intriguing to integrate it in a Space Edge Kubernetes cluster. For example, in a space topology with some satellites that comprise the Kubernetes cluster, Akri could be utilized to harness the functionality of some smaller IoT devices, pico or femto satellites with sensors, GPUs or cameras, deployed even in deep space. Thus deepening the space edge computing capabilities or even enabling a ground to deep space computational continuum.

2.5 F' Prime

Description F´ (or F Prime) is a software framework for the rapid development and deployment of embedded systems and spaceflight applications. Originally developed at NASA's Jet Propulsion Laboratory, F´ is open-source software that has been successfully deployed for several space applications. It has been used for but is not limited to, CubeSats, SmallSats, instruments, and deployables. [15]

Evaluation The use of diverse programming languages, frameworks, and network protocols, such as NASA's F Prime, highlights the need for a technology-agnostic approach to space software and infrastructure. This is where containerization and orchestration become essential, offering solutions that are not tightly coupled to any specific language or technology stack, thus providing support for even cross-organization software implementations.

2.6 Unikernel (SpaceOS, LynxElement)

Description A unikernel is a specialized, single-purpose virtual machine image that combines an application with only the minimal operating system components it requires to run. Unlike traditional operating systems, which support multiple applications and users, a unikernel is compiled into a single binary and runs a single application in isolation. [16]

Unikernels differ from virtual machines and containers in that they eliminate the need for a separate guest operating system or container runtime. Instead, they compile the application together with just the essential OS libraries into a lightweight and highly optimized executable.

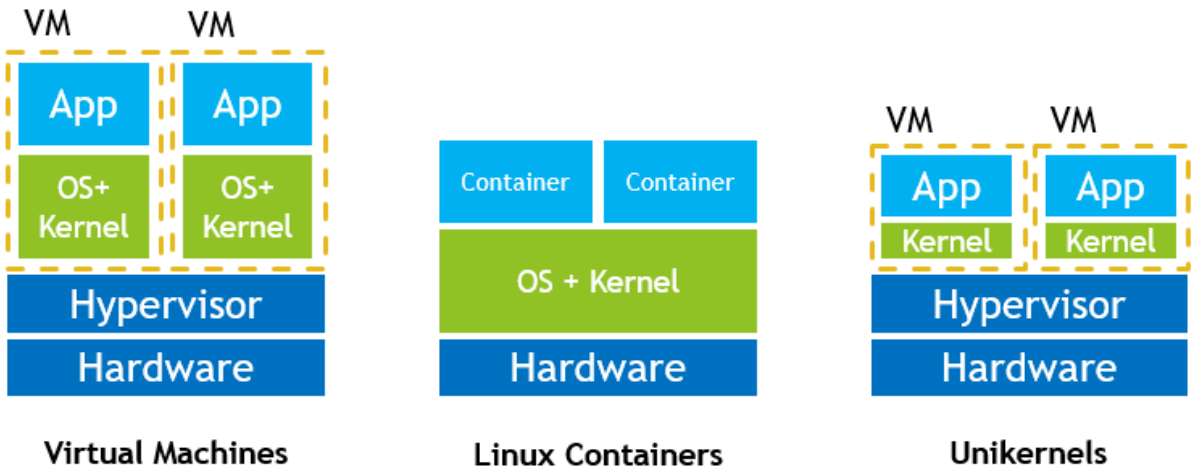

Figure 2.4: Virtual Machine, Container & Unikernel comparison [17]

SpaceOS SpaceOS is a next-generation operating system built specifically for satellite payloads, using unikernel architecture to reduce size, increase security, and optimize performance.

Each application runs with only the code it truly needs, yielding a platform that is extremely lightweight (up to 20× smaller than traditional systems) and Secure by Design from the ground up.

With SpaceOS, satellites become more adaptable and reliable – able to run multiple applications or receive updates seamlessly – all while drastically reducing security risks.

In 2025, SpaceOS will fly on its maiden space mission aboard Clustergate-1 (SpaceX's Transporter-13 rideshare) [18]

LynxElement Lynx Software Technologies (Lynx), a specialist in Mission Critical Edge, released LynxElement, the industry's first unikernel to be POSIX compatible and available for commercial use.

LynxElement unikernel works best for applications requiring speed, agility, and a small attack surface for increased security and certifiability such as aircraft systems, autonomous vehicles and critical infrastructure. The use of Unikernels, which allow pre-built applications using libraries, reduces the attack surface. Unikernels are also very well suited as a component for mission-critical systems with heterogeneous workloads that need the coexistence of RTOS, Linux, Unikernel and bare-metal guests. [19]

Evaluation Unikernels lead to extremely small, fast-booting, and secure deployments, making them particularly well-suited for environments with strict resource constraints, such as embedded systems, cloud microservices, and satellites.

Containers are larger than unikernels. They rely on a host OS, adding some overhead. However, modern space systems (especially LEO and CubeSats) are gaining more compute capacity, making this tradeoff acceptable. Also, containers are characterised by much bigger maturity and ecosystem support, broader familiarity in the developer community, multi-process & multi-language support and portability.

Concluding, unikernels are a paradigm that should be considered seriously in space edge computing. [20] It could be the ideal candidate especially for the deployment of lighter applications in very resource restrictive satellites. However, on a broader scale its utilization could be cumbersome with limited tooling compared to the full-featured containerization ecosystem, thus not preferred for this thesis.

3. BACKGROUND

In order to comprehend deeper the background of this study and understand the methodology implemented, we firstly need to expand on the foundational knowledge, definitions and technologies used.

In the context of deploying services using the NanoSat MO Framework and integrating Kubernetes into space missions, the literature can be broadly categorized into three distinct domains: DevOps-specific literature, space-specific literature, and edge computing literature. These categories collectively bridge the gap between DevOps principles, edge computing paradigms, and the unique requirements of space missions within the NanoSat MO Framework, contributing to a new frontier of research and innovation at the intersection of these domains.

The DevOps-specific literature encompasses research and studies related to DevOps principles, practices, and methodologies. It delves into the application of automation, continuous integration, and continuous delivery in software development and operations. This literature examines how DevOps has transformed various industries, particularly in the realm of information technology.

On the other hand, space-specific literature primarily focuses on the unique challenges and intricacies of space missions, satellite operations, and space data systems. It charts the evolution of small satellite technologies, space standards like CCSDS, and mission-specific case studies.

Finally, what binds Space and DevOps together is the concept of Edge Computing. Space and "Edge" share a lot in common, since both refer to locations that are not easily reached by humans. Similarly to how more and more computing is shifted to the edge, this thesis is an example of how we can achieve greater computing in space.

3.1 DevOps

3.1.1 What is DevOps?

DevOps is a collection of practices, philosophies, and tools designed to enhance the velocity of software development, shorten the time to market, and improve both software quality and system reliability [21]. It fosters collaboration between development (Dev) and operations (Ops) teams, enabling faster delivery through automation, cooperation, quick feedback loops and iterative improvement. Rooted in Agile methodologies, DevOps extends the cross-functional approach to accelerate the building and deployment of applications in a more iterative and efficient way.

The DevOps lifecycle includes a series of stages and activities that integrate development and operations seamlessly [22]. Its goal is to promote continuous integration, delivery, and improvement, ensuring the rapid release of high-quality software. In contrast to conven-

tional methods to the development lifecycle, this model assumes that each stage will be repeated as necessary. The main stages of the DevOps lifecycle include:

1. **Plan**: In this phase, teams outline the project's requirements, scope, and goals. Collaboration among development, operations, and other stakeholders ensures alignment and a shared understanding of the objectives. This stage also involves prioritizing features, setting milestones, and estimating resources. The DevOps workflow is planned with the likelihood of future iterations and likely prior versions in mind.

2. **Code**: During the coding phase, developers write and commit code changes to a common version control system like Git. The code is reviewed collaboratively to ensure compliance with coding standards, best practices, and proper documentation, maintaining both code quality and consistency.

3. **Build**: In the build phase, the code is compiled, tested, and packaged into deployable artifacts. Continuous integration ensures that every code change is automatically built and tested upon commit. Automated pipelines help ensure thorough testing and smooth code integration.

4. **Test**: Testing is crucial for verifying the software's reliability and quality. Automated tests, including unit, integration, regression, performance, and security tests, are run to validate that the code works as intended and is defect-free.

5. **Release**: The release phase occurs when the code has been verified as ready for deployment and a last check for production readiness has been performed. The project will subsequently enter the deployment phase if it satisfies all requirements and has been thoroughly inspected for bugs and other problems.

6. **Deploy**: The deployment phase releases the software across various environments like development, staging, and production. Automated deployment pipelines manage the process, including infrastructure setup, service configuration, and deployment of code artifacts, ensuring consistent, error-free releases.

7. **Operate**: The operation phase dictates that the operations team ensure that infrastructure components stay online and grow with demand as needed. This may mean scaling up or down resources as needed. Cloud environments simplify this by providing methods to monitor the load on your system and automatically spinning up new nodes to properly balance the load. User feedback is also gathered in this phase to be used to drive future development efforts, leading to a better system for the individuals or companies using the product.

8. **Monitor**: During the monitoring phase, product usage, as well as any feedback, issues, or possibilities for improvement, are recognized and documented. This information is then conveyed to the subsequent iteration to aid in the development process. This phase is essential for planning the next iteration and streamlines the pipeline's development process.

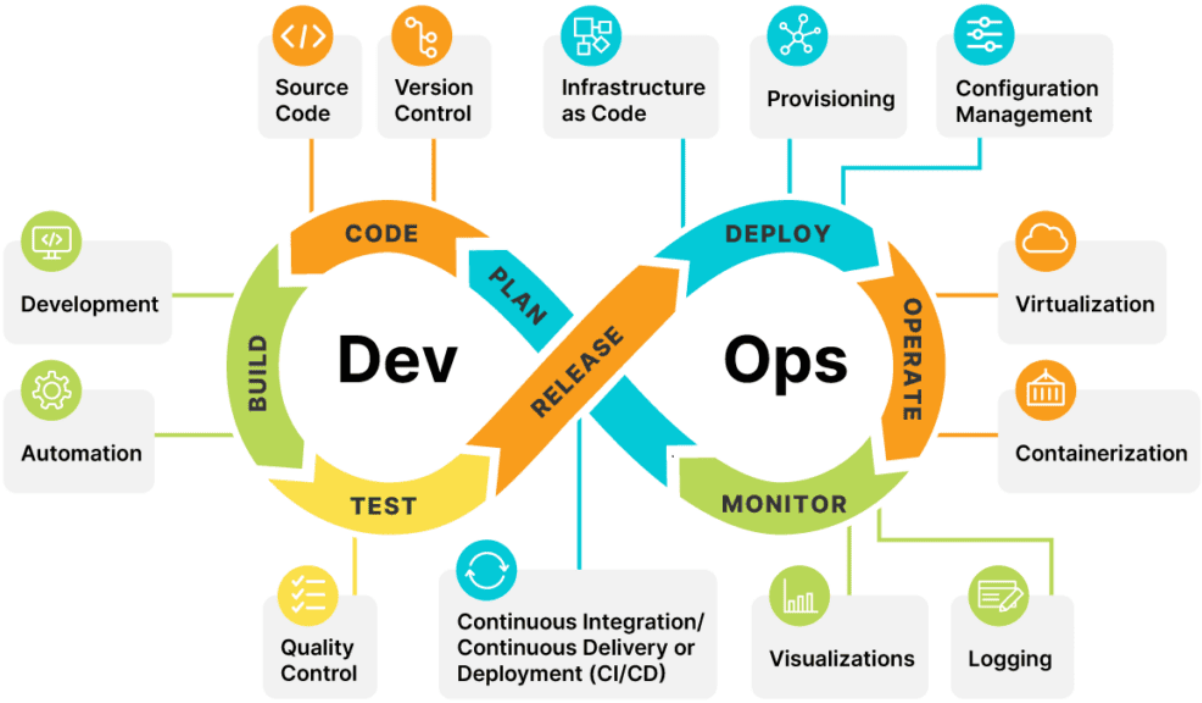

Figure 3.1: DevOps lifecycle [21]

Following, some of the core principles and established tools coupled with the DevOps world are further presented:

3.1.2 CI/CD

CI stands for Continuous Integration, while CD can either stand for Continuous Delivery or Continuous Deployment. The CI-CD concept describes the process of writing and merging code, testing and packaging it, as well as deploying it to production in small rapid cycles.

Continuous Integration is the practice of integrating code changes from multiple contributors into a centralized repository in an automated manner, while running tests and checks to ensure code quality.

A characteristic CI example for an application based on Java and Maven technologies would be: A software engineer develops the code on his local machine, pushes it to a remote Github repository shared with the rest of the team and opens a Pull Request asking for a code review. The maven project is then compiled, unit tests are run and the app is built into the required package (war, jar, ear). SonarQube tool performs code quality reviews where it will measure the quality of source code and generate a report. An artifact repository is used for storing the build artifacts and this whole process is achieved by using a Continuous Integration tool Jenkins.

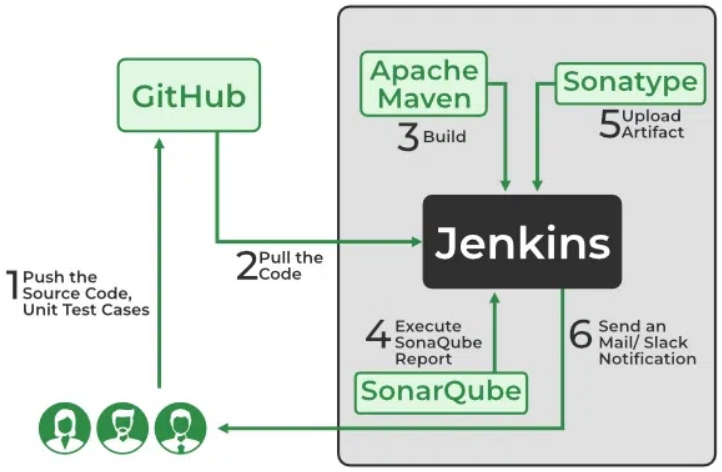

Figure 3.2: Continuous Integration real world scenario [22]

Continuous Delivery is the practice of ensuring that the code is deployment ready by creating deployment artifacts and packages and also deploying it to a staging environment for further quality assurance processes (integration testing, end to end testing etc).

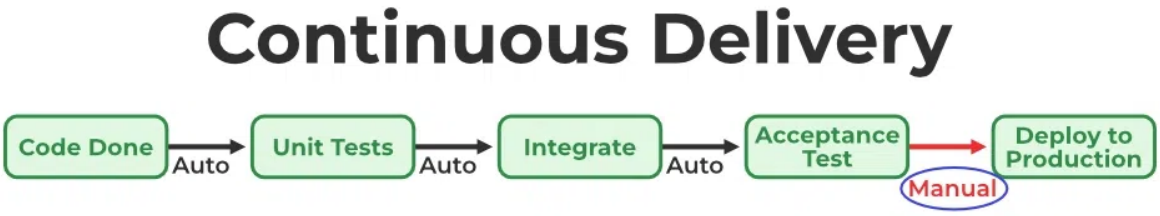

Figure 3.3: Continuous Delivery process [22]

Continuous Delivery is the practice of ensuring that the code is deployment ready by creating deployment artifacts and packages and also deploying it to a staging environment for further quality assurance processes (integration testing, end to end testing etc).

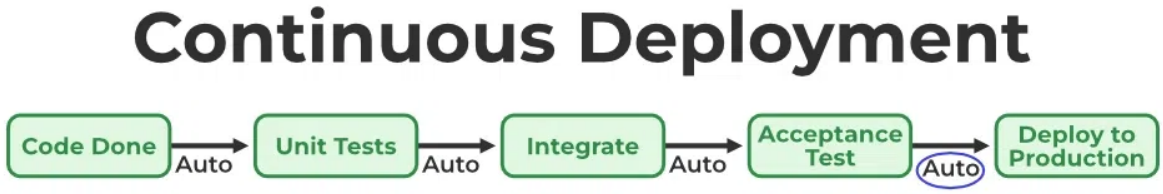

Figure 3.4: Continuous Deployment process [22]

For reference, some of the well-known CI/CD Tools, amongst others, are: Git, Jenkins, Github Actions, Gitlab CI, Argo CD.

3.1.3 Containers

Containerization is the process of packaging software code with the underlying operating system (OS) libraries and dependencies required to run the code. The output is a single lightweight executable, called a container, that runs on any infrastructure.

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. This makes it easy to move the contained application between the environments (dev, staging/test, prod, etc) while retaining full functionality. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

The container shares the kernel of the host OS with other containers, and the shared part of the OS is read-only. Therefore, the containers are lightweight, so you can deploy multiple containers on a single server (or a VM), with different application libraries, programming languages and configuration involved. An entire server needs no more to be dedicated to a single application.

Application containerization enables DevOps teams to automate the build, test, and deployment, which in turn results in faster release cycles and shortened time-to-market.

Additionally, containerization makes it possible to use microservices architecture where applications are built as services loosely coupled that can be developed, deployed, and scaled independently.

Essentially, containerised microservices are a more efficient and effective way to implement DevOps than a monolithic application.

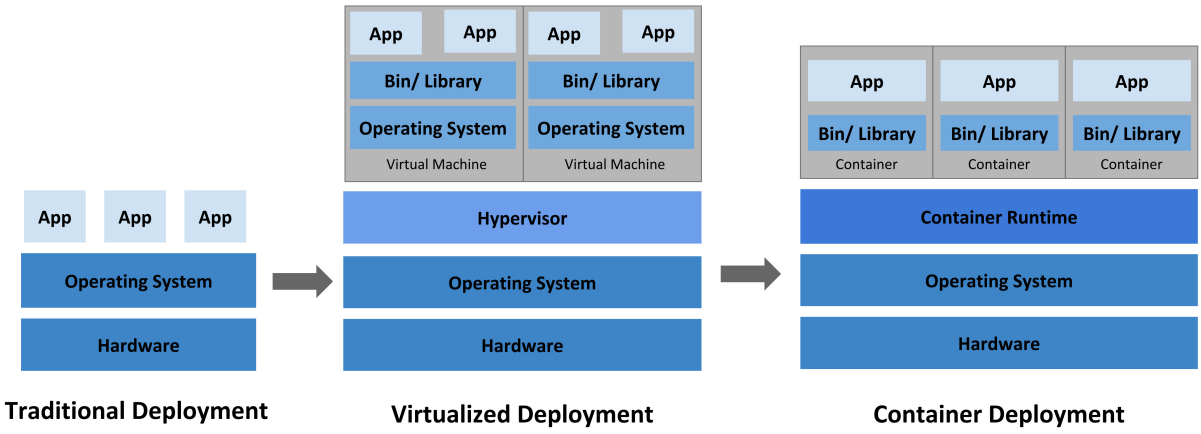

Figure 3.5: Traditional, Virtualized & Container deployment comparison [23]

3.1.4 Kubernetes

Kubernetes is an open-source platform designed, with container orchestration in mind, to automate deploying, scaling, and operating application containers [24]. It simplifies the developer's task of managing containerized applications. Kubernetes is all about optimization - automating many of the DevOps processes that were previously handled manually and simplifying the work of engineers.

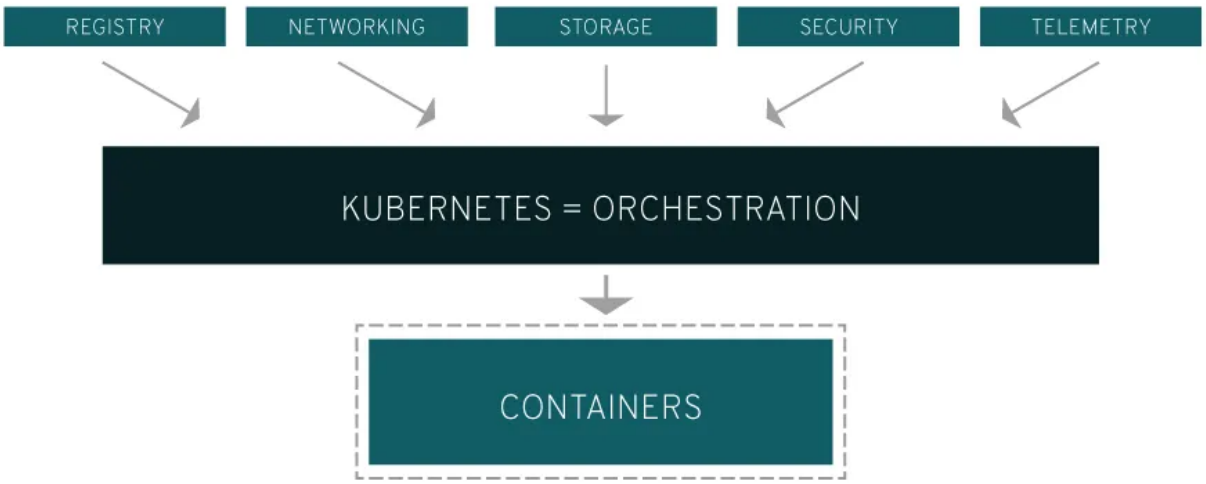

Figure 3.6: Kubernetes container orchestration [25]

Manually managing containers in a production environment without Kubernetes, would require multiple actions by the application owners. It would become a cumbersome process especially as the need for software quality and volume increases [25]. For example

-without using Kubernetes or a container orchestrator in general-, every time a container fails or more containers are needed to cater for increasing traffic, then an SSH connection to a server and commands to launch new containerized apps would be needed. On the contrary, the same incidents would be handled automatically in a Kubernetes environment. Kubernetes is the ideal system that provides this and similar functionalities out of the box in an automated way.

Kubernetes provides an interface to run distributed systems smoothly, while taking care of scaling and failover for applications, providing deployment patterns, and more.

Kubernetes provides, amongst others, the following critical functionalities [26]:

- **Service discovery and load balancing**: Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**: Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, storages from public cloud providers, and more.
- **Automated rollouts and rollbacks**: You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adapt all their resources to the new container.
- **Automatic bin packing**: You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You specify to Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can allocate containers onto your nodes to make the best use of your resources.
- **Self-healing**: Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management**: Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Horizontal & Vertical scaling**: Scale your application up/down and out/in with a simple command, with a UI, or automatically based on CPU usage.

In general, most of the administration actions in a Kubernetes cluster are handled by the control plane, which presents an API and is in charge of scheduling and replicating groups

of containers named Pods. Kubectl is the command line interface that facilitates the interaction with the API to share the desired application state or gather detailed information on the infrastructure's current state.

Each node that hosts part of your distributed application does so by leveraging Docker or a similar container technology. The nodes also run two additional pieces of software: kube-proxy, which gives access to your running app, and kubelet, which receives commands from the k8s control plane. Nodes can also run flannel, an etcd backed network fabric for containers. Kubernetes consists of two basic types of nodes:

- **Control Plane nodes**: the Kubernetes control plane is the management layer inside a Kubernetes cluster. It's a collection of components that work together to manage the cluster's state, coordinate your Nodes, and provide the API server that lets you interact with the cluster. [27]
- **Worker nodes**: the Kubernetes worker nodes handle the application workload. Each of those node types runs a different set of Kubernetes components. [28]

Control Plane Components

- **API Server (kube-apiserver)**: the core server of Kubernetes serving the HTTP API and enabling user-facing tool and access to the cluster. Thus, it is very important that is remains up and running
- **etcd**: this is the "brain" of Kubernetes. It is a distributed key-value datastore. The data regarding the Kubernetes cluster (Kubernetes objects, the current state of each object, configs etc) are all stored in the etcd.
- **Scheduler (kube-scheduler)**: watches over newly created pods that have not been assigned to a node and selects a node for them to run on.
- **Controller Manager**: controllers are a fundamental feature of Kubernetes. A controller is a component that watches over the cluster and when changes are applied, the controller may perform corresponding actions if needed. So the controller manager runs all the necessary controllers.

  Some examples are:

  - Node Controller: responsible for noticing and responding when a node goes down
  - Deployment Controller: creates new pods based on the specs of the Deployment object
  - Cronjob Controller: enables the periodic creation of Kubernetes Job objects
- Cloud Controller Manager: this controller manager is required for managed Kubernetes running on cloud providers and its purpose is to provide access to the provider's API for the control plane.

Worker Node Components

- **Kube Proxy**: it implements the Kubernetes networking layer. In other words it acts like a network abstraction over the nodes, by maintaining network rules on the host and performing connection forwarding.
- **kubelet**: it is the primary kubernetes agent on the node. It registers the node to the kubernetes control plane and watches over pods assigned to the pod by regularly polling the kubernetes API for new pod specs and makes sure they will be implemented.
- **Container Runtime**: it is essential for a Kubernetes worker node to be running a kubernetes runtime, most commonly docker, container-d, so that containers can be created for pods.

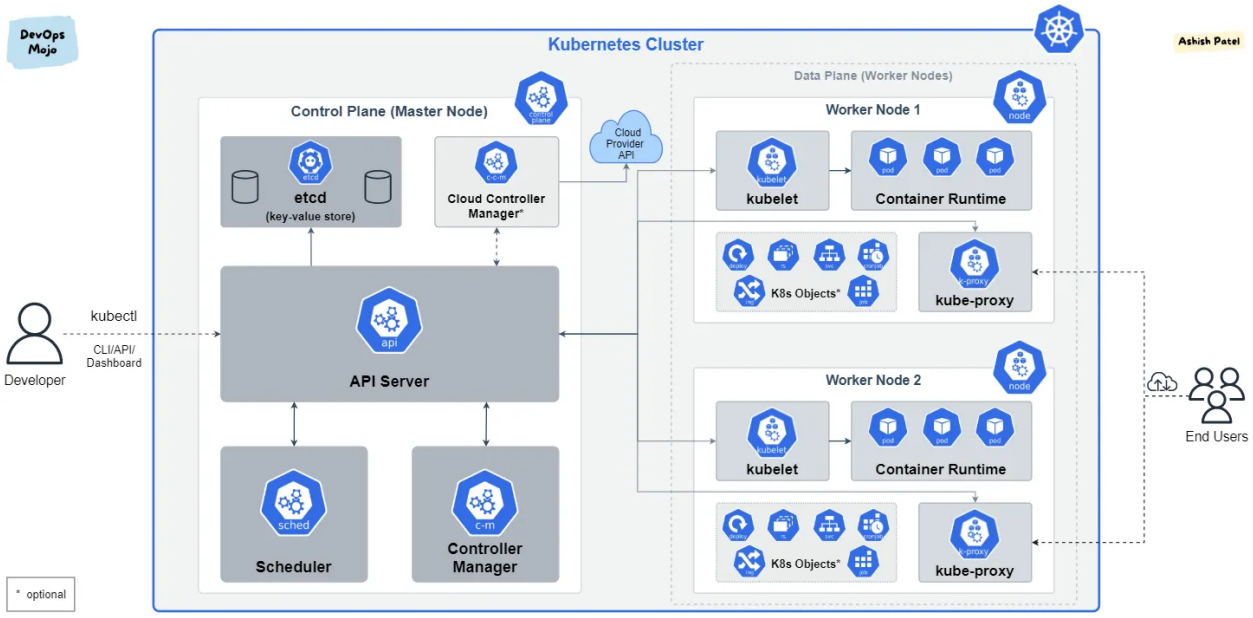

Figure 3.7: Kubernetes cluster architecture [29]

The control plane itself runs the API server (kube-apiserver), the scheduler (kube-scheduler), the controller manager (kube-controller-manager) and etcd, a highly available key-value store for shared configuration and service discovery implementing the Raft consensus Algorithm.

3.1.5 Helm charts

Helm charts, a package manager for Kubernetes releases, focuses on the simplified automated management and deployment of Kubernetes applications. A Helm Chart is a collection of files that describe a set of Kubernetes resources like deployments, services,

and config maps, along with the necessary configuration values. When a chart is deployed, Helm processes these templates with user-provided or default values, rendering Kubernetes manifests. It then communicates with the cluster to create, update, or delete resources.

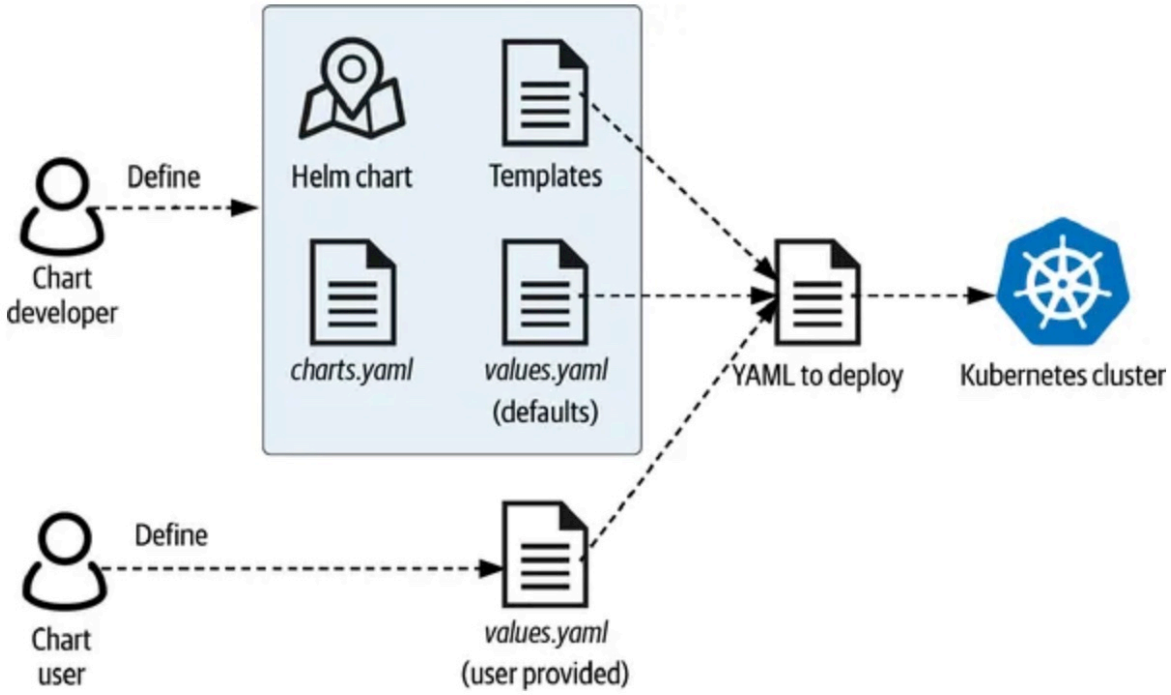

Figure 3.8: Helm charts dynamic configuration [30]

Without helm charts every single yaml configuration file bound to each Kubernetes resource would need to be defined manually. This way, the management of a massive and constantly growing application ecosystem would become progressively inconvenient with increasing complexity.

In order to make the process of Kubernetes configuration simpler and faster, Helm charts are used to preconfigure applications and automate the processes of development, testing and production [31]. Instead of manually writing and managing tons of YAML files, a helm chart is used to install, upgrade, or delete your app release with simple commands.

3.1.6 Docker Compose

Docker Compose is a tool that's used to build and run multi-container applications with Docker, the most popular containerization solution [32]. Compose simplifies the control of an entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.

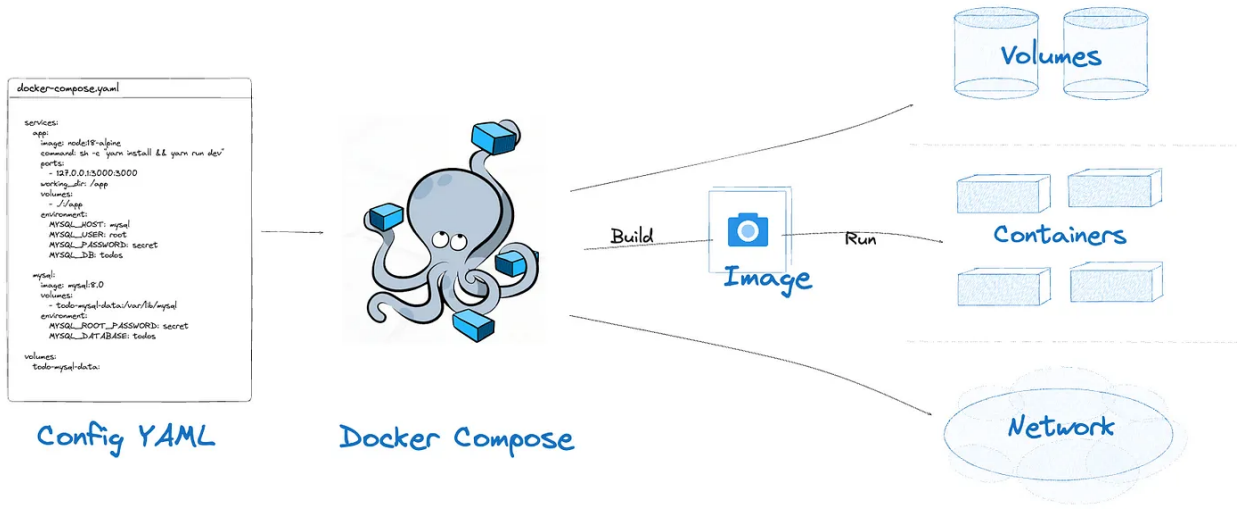

Figure 3.9: Docker Compose for multi-container application management [33]

A very common use case for Docker Compose is to set up local development environments quickly with all required dependencies, like databases, message queues, caches, web service APIs and configurations.

3.2 Space

3.2.1 Small Satellites (CubeSats, NanoSats)

Small satellites, commonly referred to as smallsats, are spacecraft characterized by their relatively low mass and compact size, typically weighing less than 500 kilograms. This category encompasses various subtypes, including minisatellites (100–500 kg), microsatellites (10–100 kg), nanosatellites (1–10 kg), picosatellites (0.1–1 kg), and femtosatellites (less than 0.1 kg).

The emergence of smallsats in space missions is largely attributed to advancements in miniaturization, in-space propulsion, onboard processing, and communication systems [34]. These technological developments have enabled smallsats to perform complex tasks previously reserved for larger spacecraft. Their affordability and shorter development cycles make them accessible to a wide range of organizations, fostering innovation and expanding participation in space activities.

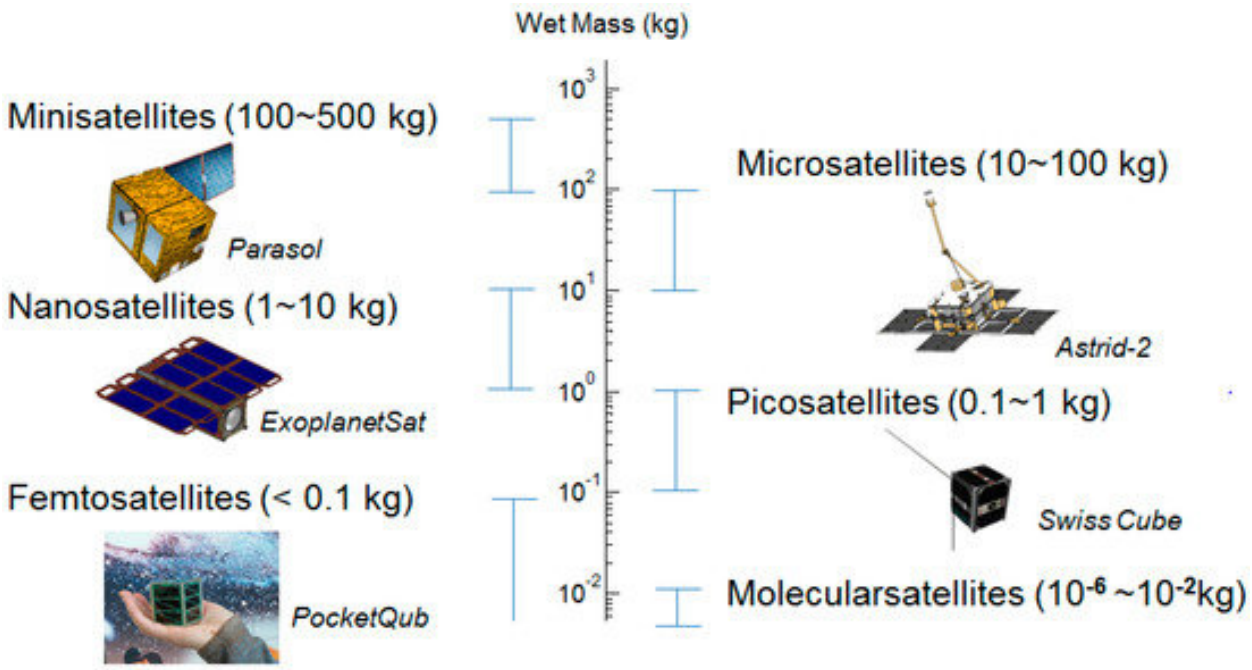

Figure 3.10: Small satellites categories based on mass [35]

One notable subclass of smallsats is the CubeSat, nanosatellites that use a standard size and form factor. The standard CubeSat size uses a "one unit" or "1U" measuring 10x10x10 cms and is extendable to larger sizes; 1.5, 2, 3, 6, and even 12U [36].

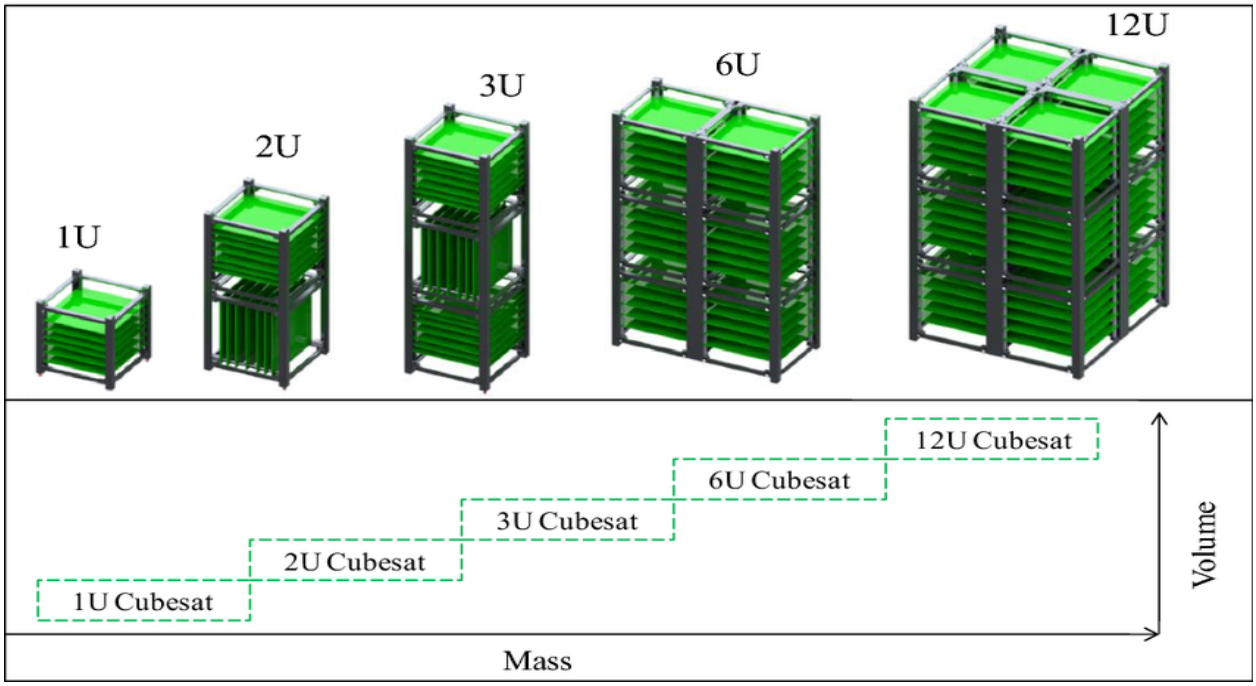

Figure 3.11: CubeSat standard unit measurement [37]

They use commercial off-the-shelf (COTS) components for their electronics and structure. CubeSats are deployed into orbit from the International Space Station, or launched as secondary payloads on a launch vehicle. One such well known CubeSat is OPS-SAT by the European Space Agency (ESA), intended to demonstrate the improvements in mission control capabilities that will arise when satellites can fly more powerful on-board computers.

The proliferation of smallsats has revolutionized the space industry by enabling satellite constellations that provide enhanced coverage and data collection capabilities. As seen in the following diagram, smallsats comprise the vast majority of spacecraft launches over the past years, leaving a large gap between them and the traditionally larger satellites. Broadband smallsats have dominated the market since 2020, while earth observation and remote sensing smallsats are steadily increasing.

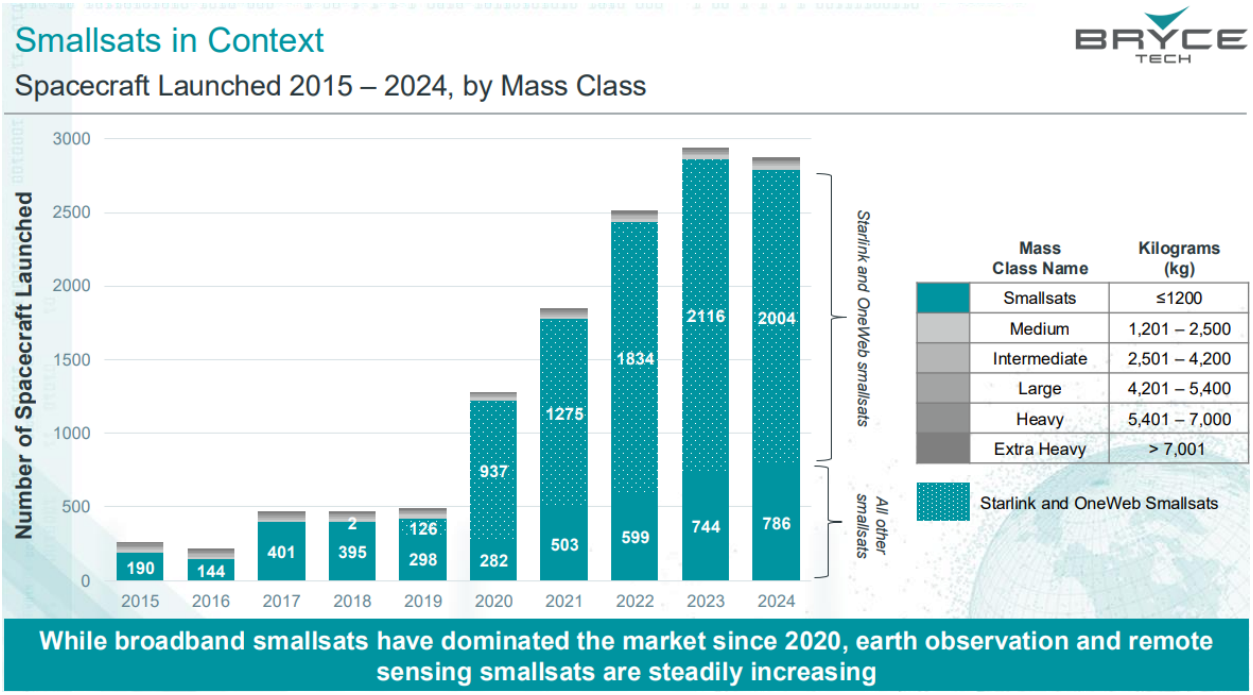

Figure 3.12: Spacecraft Launched 2015-2024, by Mass Class [38]

However, this rapid increase also presents challenges, such as potential overcrowding in low Earth orbit and regulatory concerns regarding frequency allocation and collision avoidance. Addressing these issues requires international cooperation and the development of sustainable practices to ensure the long-term viability of space activities.

In summary, small satellites have become integral to modern space missions, offering cost-effective and versatile solutions for a wide array of applications [39]. Their continued evolution promises to further democratize access to space and drive innovation across multiple sectors.

Key small satellite missions that demonstrate this shift in this new era of faster development and deployment cycles along with more computing power are:

- **OPS-SAT** [40]: Acting as a flying laboratory, ESA's OPS-SAT mission was the first of its kind, with the sole purpose of testing and validating new techniques in mission control and on-board satellite systems. OPS-SAT demonstrated drastically improved mission control capabilities that will arise when satellites can fly more powerful on-board computers. The 3U Cubesat was only 30cm high, but it contained an experimental computer ten times more powerful than any current ESA spacecraft.

- **Φsat-2** [41]: The ESA Φsat-2 mission combines on-board processing capabilities (including AI) and a medium to high resolution multispectral instrument from Visible to Near Infra-Red able to acquire 8 different bands including a panchromatic one. This satellite hosts a multispectral Imager and six AI applications designed to turn images into maps, detect clouds in the images, classify them and provide insight into cloud distribution, detect and classify vessels, compress images on board and reconstruct them in the ground reducing the download time, spot anomalies in marine ecosystems and do wildfire detection.

3.2.2 CCSDS

Founded in 1982 by the major space agencies of the world, the Consultative Committee for Space Data Systems (CCSDS) is a multinational forum for the development of communications and data systems standards for spaceflight. The goal is to enhance governmental and commercial interoperability & cross-support, while also reducing risk, development time and project costs [42].

A major milestone was achieved by the Spacecraft Monitoring & Control (SM&C) Working Group of CCSDS, which defined a service-oriented architecture for mission operations of future space missions, referred to as CCSDS Mission Operations (MO) services. Legacy standards that did not fully exploit the capabilities of modern technologies could finally be decommissioned, paving the way for more advanced and efficient operations. Such an example is the replacement of the Packet Utilisation Standard (PUS) previously used for information exchange [43]. This new architecture consists of a set of standardized end-to-end services linking functions on board a spacecraft with those on the ground, all of which are responsible for mission operations [44].

The architecture allows mission operation services to be specified in an implementation and communication agnostic manner. The MO services are specified in compliance to a reference service model, using an abstract service description language, the MAL. This is similar to how Web Services are specified in terms of Web Services Description Language, WSDL. For each concrete deployment, the abstract service interface must be bound to the selected software implementation and communication technology.

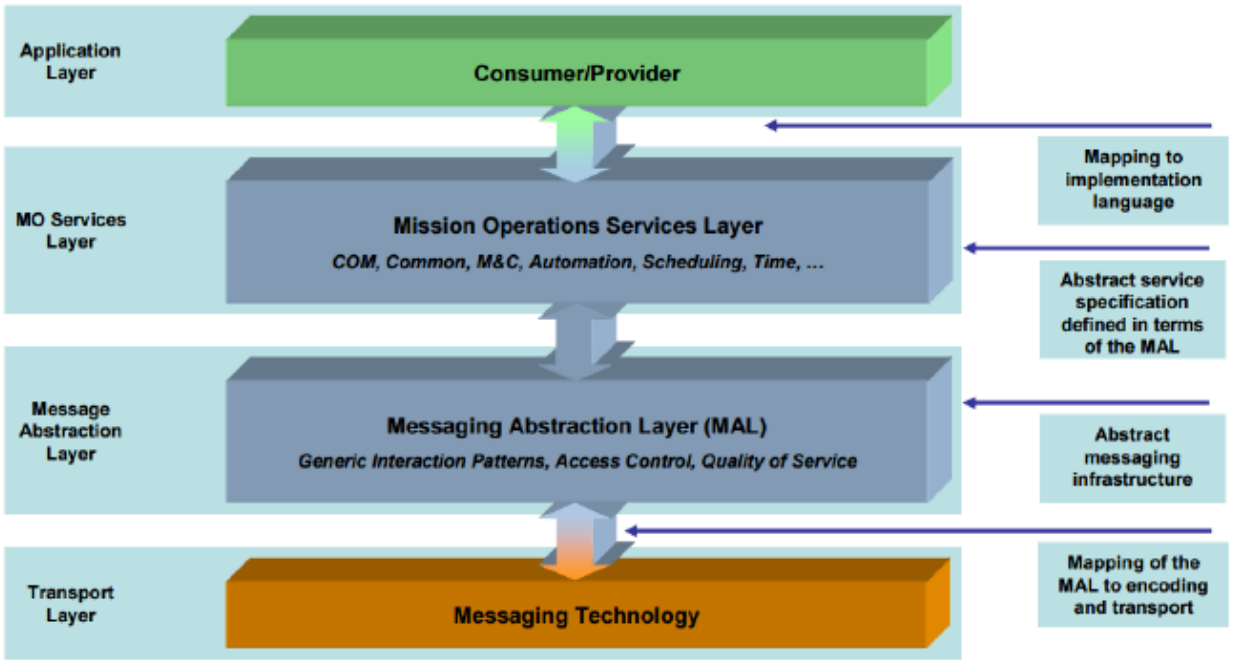

Figure 3.13: CCSDS architecture layers [45]

Standardization of a Mission Operations Service Framework offers a number of potential benefits for the development, deployment and maintenance of mission operations infrastructure:

- Increased interoperability between agencies
- Re-usage between missions
- Reduced costs
- Greater flexibility in deployment boundaries
- Increased competition and vendor independence
- Improved long-term maintainability

The deployment of standardized interoperable interfaces between operating Agencies, the spacecraft and internally on-board would in itself bring a number of benefits. Each organization would be able to develop or integrate their own multi-mission systems that can then be rapidly made compliant with the spacecraft. It does not preclude the reuse of legacy spacecraft, simply requiring an adaptation layer on the ground to support it, rather than many mission-specific bespoke interfaces. In the on-board environment, where software development costs are considerably higher due to platform constraints and reliability requirements, software reuse can bring immense savings.

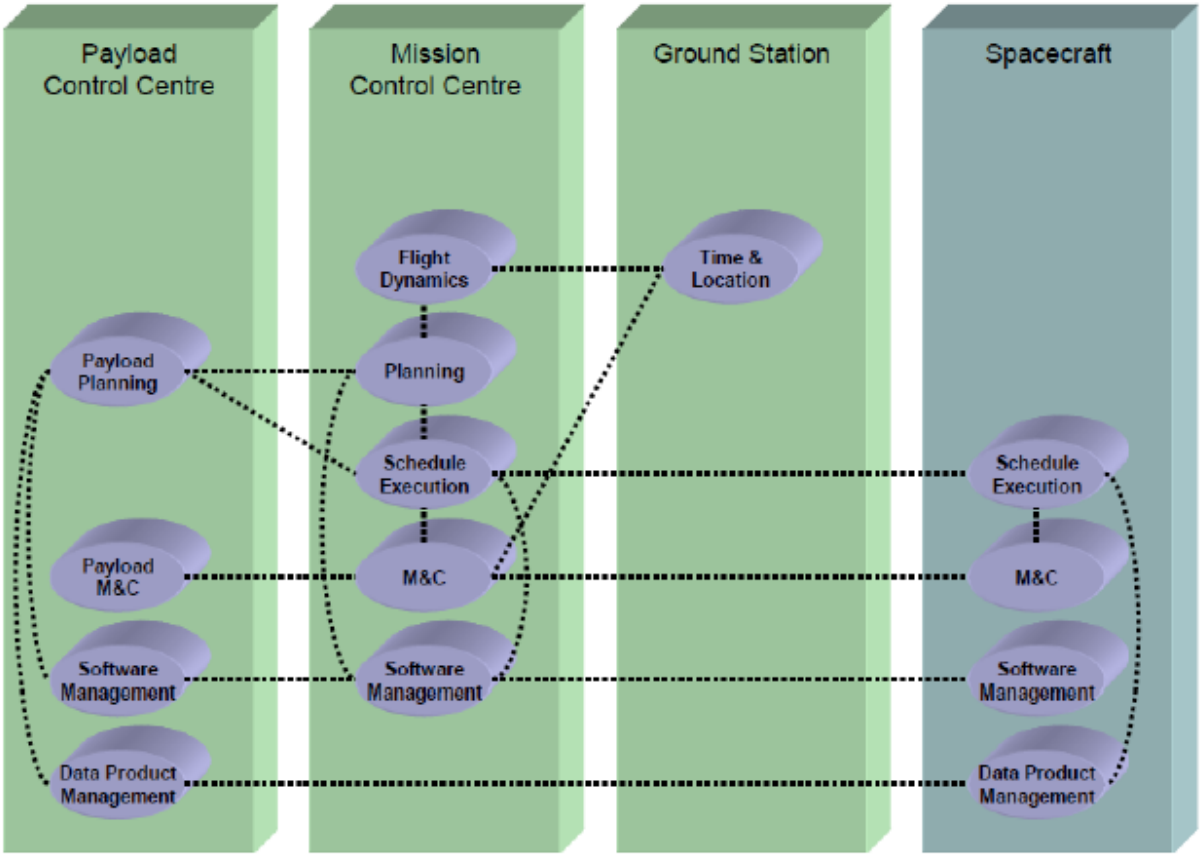

Figure 3.14: Mission operations activities distributed between a Spacecraft and three separate Ground Segment facilities [45]

3.2.3 NanoSat MO Framework

The NanoSat MO Framework [46][47] is an open-source software framework for nanosatellites based on CCSDS Mission Operations services.

It introduces the concept of apps in space that can be started and stopped from ground. Apps can retrieve data from the platform through a set of well-defined MO services. The NanoSat MO Framework supports missions by facilitating the development, distribution, and deployment of apps on satellite missions. The NanoSat MO Framework facilitates the monitoring and control of the satellite and also provides a set of services for software management, enabling the onboard apps to be installed, uninstalled and upgraded remotely from ground. Many possibilities for extensions are available due to its modular and flexible design approach which is not limited to the space segment but extends down to ground by providing all the building blocks for a complete and free end-to-end solution.

The NMF ecosystem was an ideal candidate to harness the improved computational capabilities of the new era of small satellites. So, it was first launched on 18 December 2019 with ESA's OPS-SAT mission [48][49], which provided a fast experiment service for Eu-

ropean and Canadian industry and academia, and completed its mission on 22-23 May 2024. Currently, it is flying on-board ESA's Φsat-2 (phisat-2), a 6U cubesat that will further demonstrate the benefits of using Artificial Intelligence (AI) for innovative Earth observation.

A fundamental module of the NMF installation is the "Supervisor" module. The NanoSat MO Supervisor is an NMF Composite that acts as a supervisor and it is intended to be deployed on the NanoSat segment for an on-board deployment with multiple applications. It supports the discoverability of the providers available on-board and allows different applications to interact with the peripherals on-board via a set of Platform services.

Additionally, it includes software management capabilities such as: starting, stopping, installing, updating and uninstalling applications on-board the spacecraft. This component must be extended to the specific platform and particular mission use case.

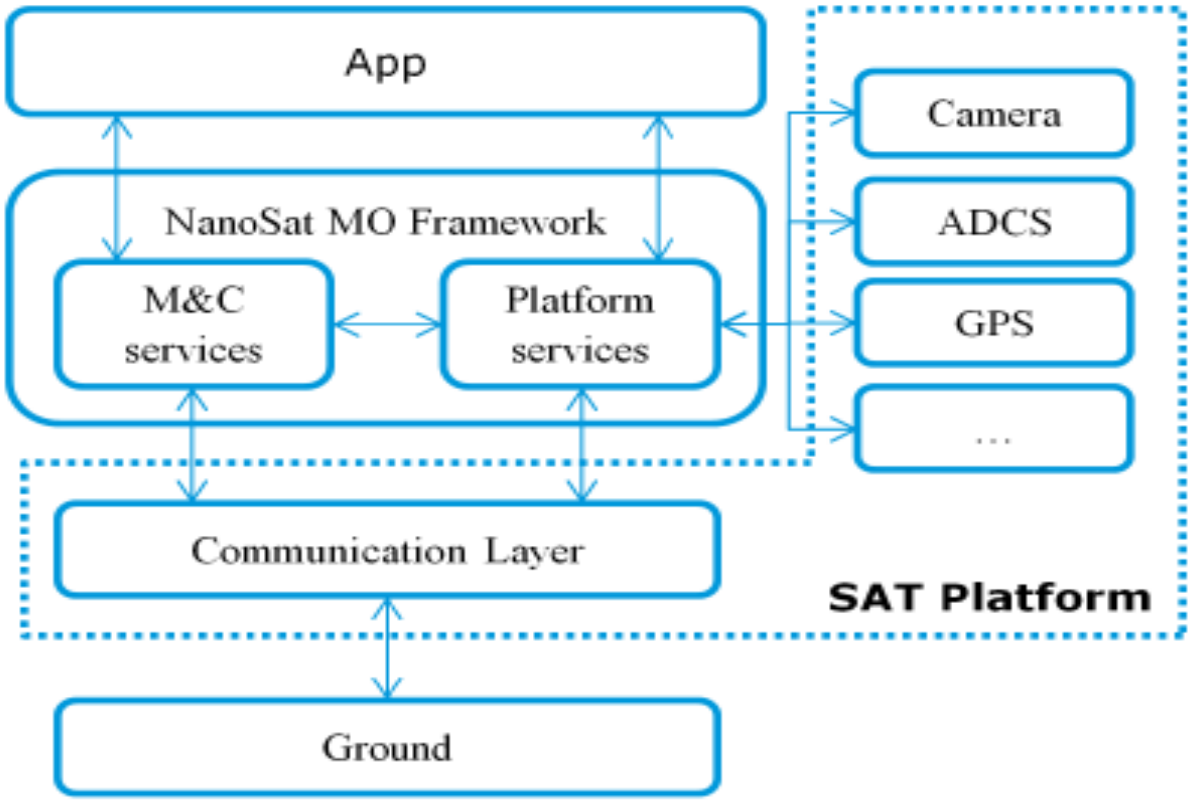

Figure 3.15: Overview of NanoSat MO Framework components [50]

This component includes the Central Directory service that provides discoverability functionality by holding the connections information about all the providers running on-board. Specifically, the NanoSat MO Supervisor provider and the set of registered running provider applications with their respective services.

The implementation of the Platform services to the platform peripherals on the NanoSat MO Supervisor allows multiple consumers to interact with the peripherals available on-

board. The connection between a consumer and a provider of a Platform service is expected to occur via Inter-Process Communication (IPC) [51].

The NanoSat MO Supervisor is able to install packages carrying applications or libraries via the Package Management service. After an application is installed, the Apps Launcher service can be used to start the application and afterwards stop it. The Package Management is able to update a certain package which allows, for example, new features or bug fixes to be ported into an application. Lastly, a package can be uninstalled by the NanoSat MO Supervisor. To give a more specific example of how supervisor works: if we needed to install a new service, then a package java application would be sent to the supervisor and the supervisor would have the responsibility to run that code and monitor the health of the application. Both supervisor and the services are running in the same JVM, as separate processes.

Another useful module of the NMF ecosystem is the Simulator module. The Software Simulator is not an actual service used in mission, rather than a test implementation of the Supervisor. It was developed as part of the NMF SDK in order to mock hardware devices and test services for usage in space missions.

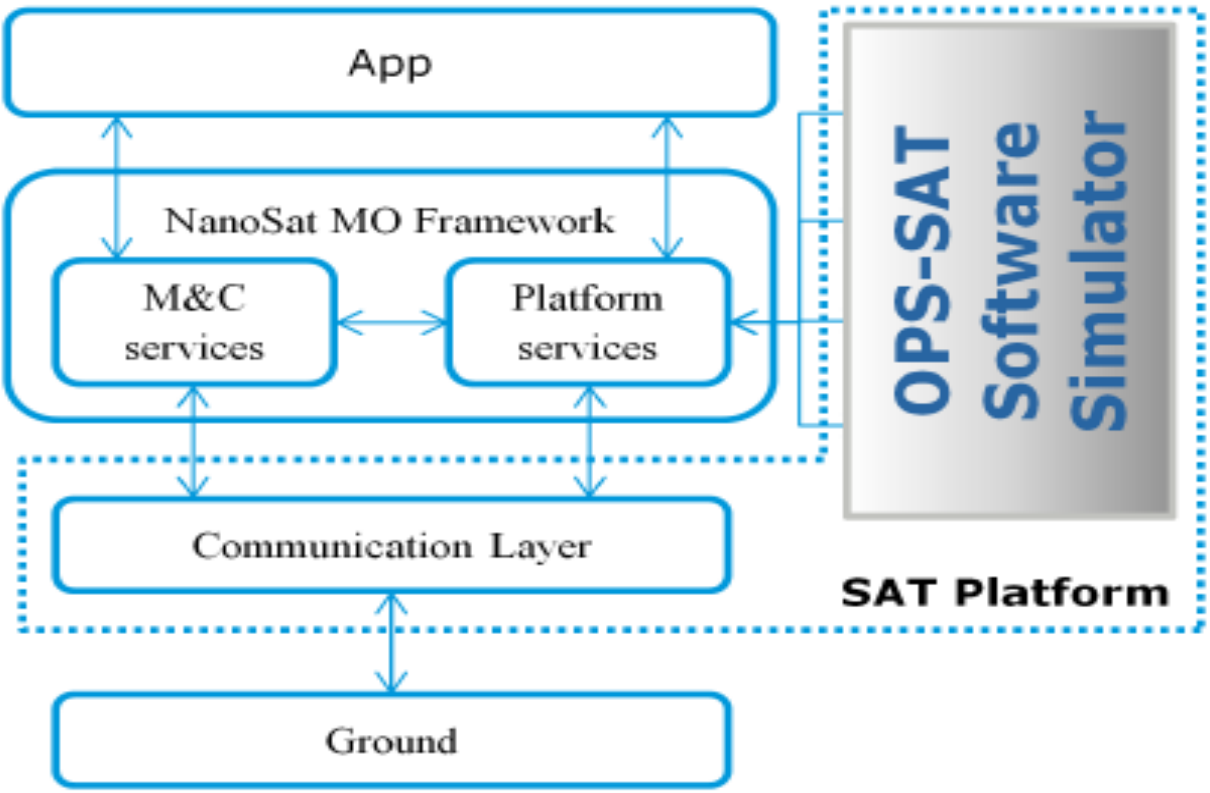

Figure 3.16: NanoSat MO Framework integrated Supervisor-Simulator [50]

Furthermore, the Communication Testing Tool (CTT) is a diagnostic and validation utility within the European Space Agency's Network Management Framework (NMF), devel-

oped to support the testing of space software applications. CTT enables developers to simulate and verify interactions between onboard applications and ground systems from a user-friendly interface. It provides a Graphical User Interface (GUI) to invoke service operations, exchange telemetry and commands, and validate protocol compliance across the Mission Abstraction Layer (MAL).

In implementation terms, it uses the Ground MO Adapter for the connection to the providers. Although CTT is provided as a "ready to be used" application, its source code is also available for developers to learn and get an example of how a client-server architecture could be implemented in NMF.

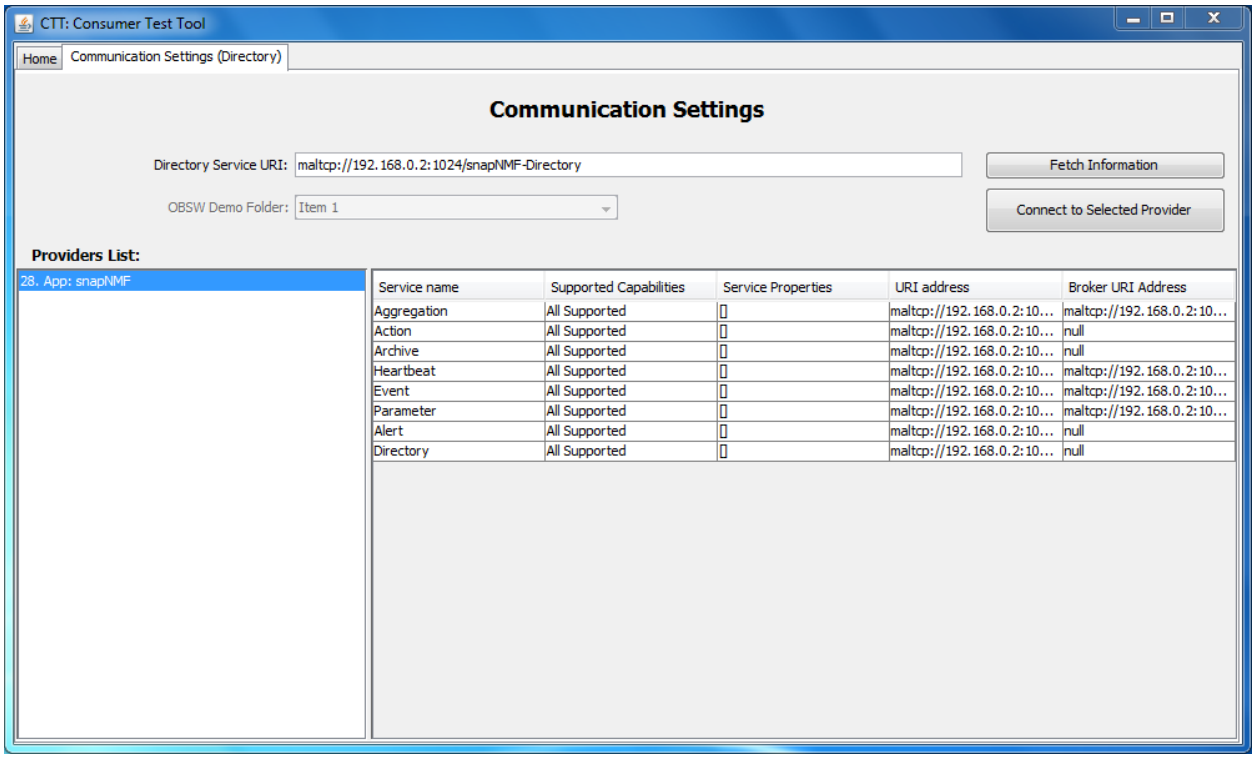

Figure 3.17: Communication Testing Tool GUI [45]

CTT was designed to support the connection to multiple NMF Apps simultaneously via a tabbed view mechanism. Amongst others, it can list registered services and applications, invoke operations ("actions") exposed by the services (such as commanding an onboard camera to take a snapshot), monitor events and status updates and apply configurations to the services.

3.3 Edge Computing

Edge computing represents a paradigm shift in distributed computing, wherein data processing occurs in close proximity to the physical location of data generation, such as sensors and devices, rather than relying solely on centralized servers or cloud infrastructures.

This architecture typically integrates edge servers and connected devices (e.g., laptops, smartphones) into a local network, enabling secure, real-time data processing directly at the source.

The concept of edge computing originated in the 1990s, initially referring to content delivery networks (CDNs) that aimed to reduce latency by distributing website and video content through geographically dispersed servers. In the early 2000s, the functionality of these networks expanded to support application hosting capabilities, thereby giving rise to early edge computing services. These services facilitated a variety of tasks, such as locating dealerships, managing e-commerce operations (e.g., shopping carts), collecting real-time data, and delivering targeted advertising.

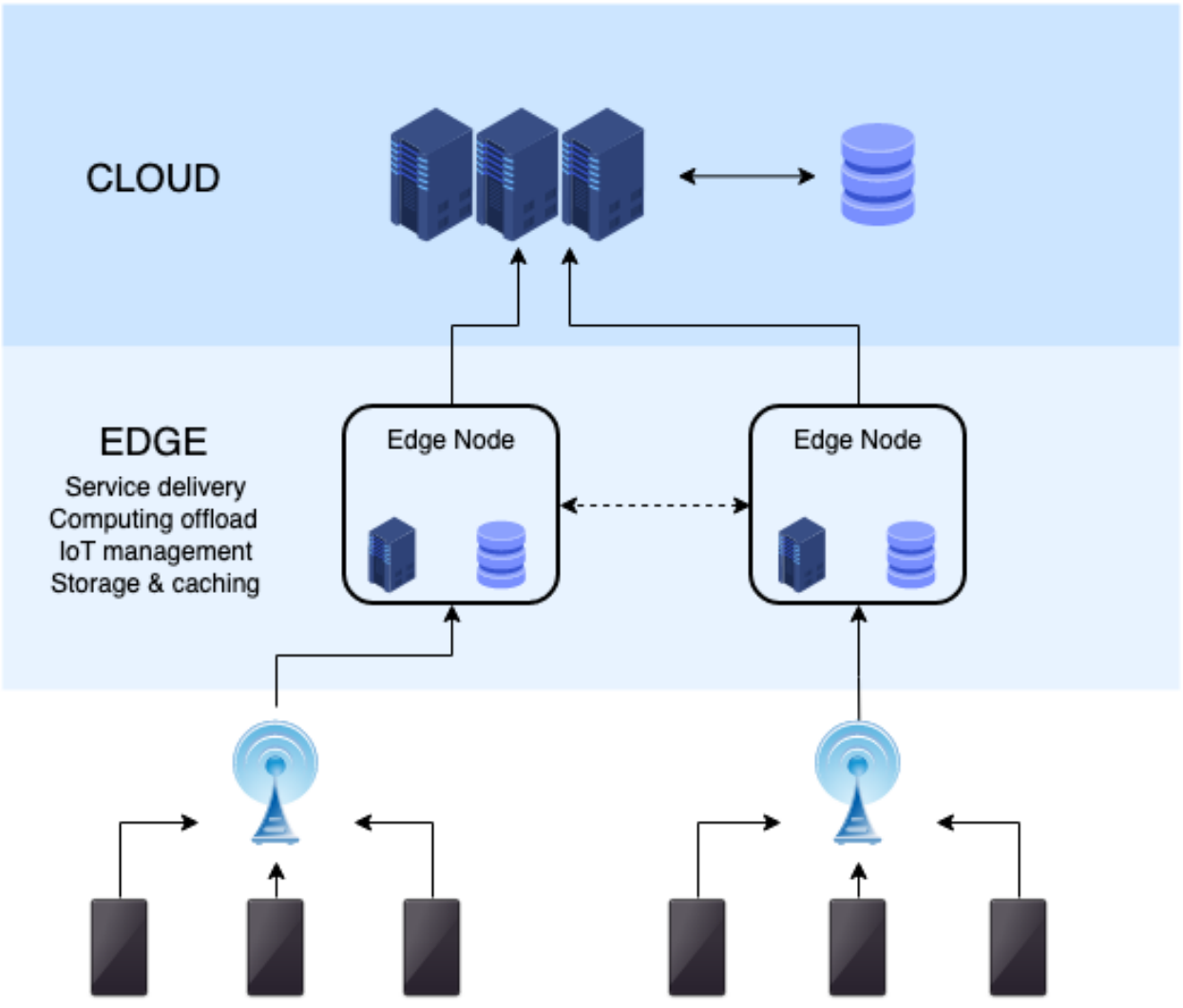

Figure 3.18: The edge computing infrastructure [52]

The evolution of edge computing has been closely associated with the rise of the Internet of Things (IoT), a wide network of interconnected smart devices and sensors. The explosive

growth and increasing computing power of IoT devices have resulted in massive volumes of data. This data scale-up has been further accelerated by the advent of 5G networks, which significantly increase the number and mobility of connected devices.

In the past, the promise of cloud and AI was to automate and speed up innovation by driving actionable insight from data. However, the scale and complexity of data generated by connected devices have begun to exceed the capabilities of conventional network and infrastructure systems. Transmitting all device-generated data to centralized data centers or cloud platforms introduces considerable bandwidth constraints and latency issues.

Edge computing addresses these challenges by enabling data processing and analysis to occur closer to the point of origin. This localized approach significantly reduces latency and alleviates the burden on network bandwidth, enhancing system responsiveness. Particularly in the context of mobile edge computing facilitated by 5G networks, this model supports rapid and extensive data analysis, thereby fostering deeper insights, reduced response times, and enhanced user experiences.

3.3.1 Space Edge Computing

Data generated in space is growing rapidly. A typical commercial imaging constellation collects more than 100 terabytes of data per day. Space data traffic is expected to reach 566 exabytes over the next decade, according to NSR (global market research and consulting firm focused on the satellite and space sectors), with satcom accounting for over 90% of that total [53]

As space missions grow more autonomous and data-intensive, they increasingly rely on computing systems capable of functioning independently of centralized ground control. In response to this shift, space edge computing has emerged as a foundational approach, enabling real-time processing directly at the site of data generation. Rather than routing all data to Earth for analysis, edge computing reduces latency, minimizes bandwidth usage, and supports autonomous decision-making in remote or delay-prone environments such as deep space.

Long communication delays, high radiation levels, and limited energy budgets render traditional cloud-based models insufficient for space applications. Edge computing enables spacecraft, satellites, and planetary rovers to analyze sensor data in real time, make decisions locally, and respond to their environment without waiting for remote instructions.

Scientific missions that venture into deep space, including among others Mars landers, asteroid rendezvous missions, or icy moon probes, face significant communication delays, often ranging from several minutes to over an hour each way. In these environments, transmitting every data packet to Earth for analysis is neither practical nor efficient.

Edge computing allows spacecraft to process scientific data onboard, evaluate its significance, and prioritize transmissions accordingly. For example, an onboard spectrometer analyzing rock samples on a Martian surface can distinguish between ordinary regolith and material with unusual mineral content. Only the latter needs to be flagged for high-

resolution analysis or downlinked for further study.

3.3.1.1 Space Edge

A satellite can be considered the quintessential edge device in many respects. It operates remotely, is often disconnected from terrestrial data centers, and generates vast amounts of data. Equipping a satellite with local computing capabilities enables a range of operational advantages, including reduced latency, enhanced security, and lower data transmission costs.

The concept of space-based edge-hosted payloads aims to minimize the need for data to make repeated round trips to Earth stations or cloud infrastructure on the ground. Given the significant distance from Low Earth Orbit (LEO), Medium Earth Orbit (MEO), or Geostationary Orbit (GEO) to Earth, and the additional time required for analysts to process raw data and return results to the satellite, onboard processing offers a more efficient solution.

3.3.1.2 Ground Edge

In the ground segment, the edge of a satellite network typically refers to the terminal and in some cases, the teleport or gateway. This edge may be located on a plane, boat, vehicle, or other remote site. It generally sits at the outermost point of the network, closest to where data is generated.

Similar to the space-based edge, implementing edge computing at the terminal enhances network performance by bringing workloads closer to the end user. This approach reduces latency, strengthens security, and helps alleviate the costs and bandwidth demands associated with transmitting data to a centralized network.

3.3.1.3 Cloud

The cloud enables two-way communication with the edge, creating an environment that supports easier collaboration across locations, applications, and network functions. Data sent back to the cloud can be processed, stored, or accessed by authorized users within that ecosystem. Additionally, the cloud can send commands, such as software updates or configuration changes, back to edge devices.

As technology continues to advance, many enterprises are increasingly adopting edge cloud solutions (compact versions of centralized cloud infrastructure) and edge servers (essentially micro data centers). These options provide faster data analysis and enhanced processing power without the need to connect to traditional, centralized cloud data centers.

4. ARCHITECTURE AND DESIGN

4.1 Architecture

4.1.1 Current Nanosat MO Framework Architecture

The current NanoSat MO Framework runs on a JVM tightly coupled architecture. On the satellite, the framework hosts multiple space applications (e.g., Space App 1, Space App 2) and a Supervisor component, all running within a single Java Virtual Machine (JVM). The Supervisor manages app lifecycle and interactions with onboard devices.

On the ground segment, one or more NanoSat MO Framework instances provide Ground Services that communicate with the satellite using standardized CCSDS MO protocols, enabling telecommand, telemetry, and data handling.

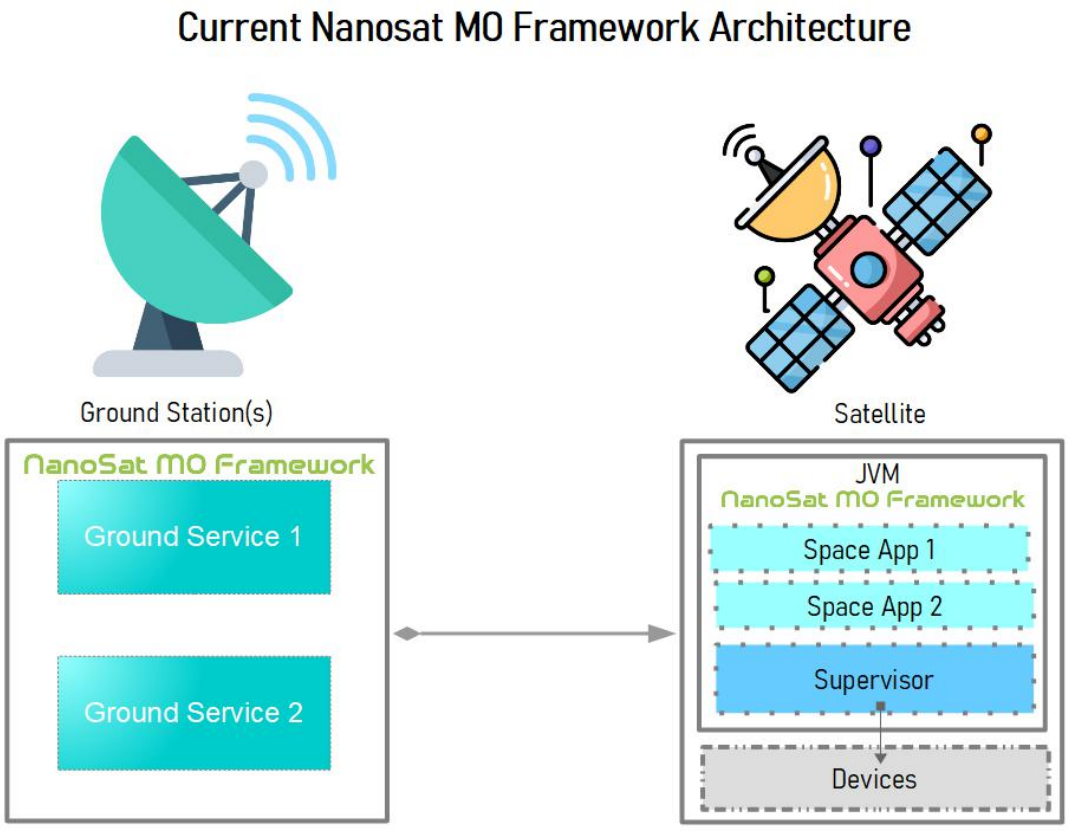

Figure 4.1: Current NanoSat MO Framework Architecture

4.1.2 Proposed Nanosat MO Framework Architecture

The proposed architecture replaces the JVM-based setup with a lightweight Kubernetes (k3s) environment on the satellite. Each space application, supervisor, and NanoSat MO Framework instance runs as an independent container, improving isolation, scalability, and deployment flexibility. A Kubernetes Device Plugin layer interfaces directly with the satellite's hardware devices, allowing containerized apps to access and manage onboard systems efficiently. Ground stations continue to operate NanoSat MO Framework services, maintaining standardized communication with the satellite.

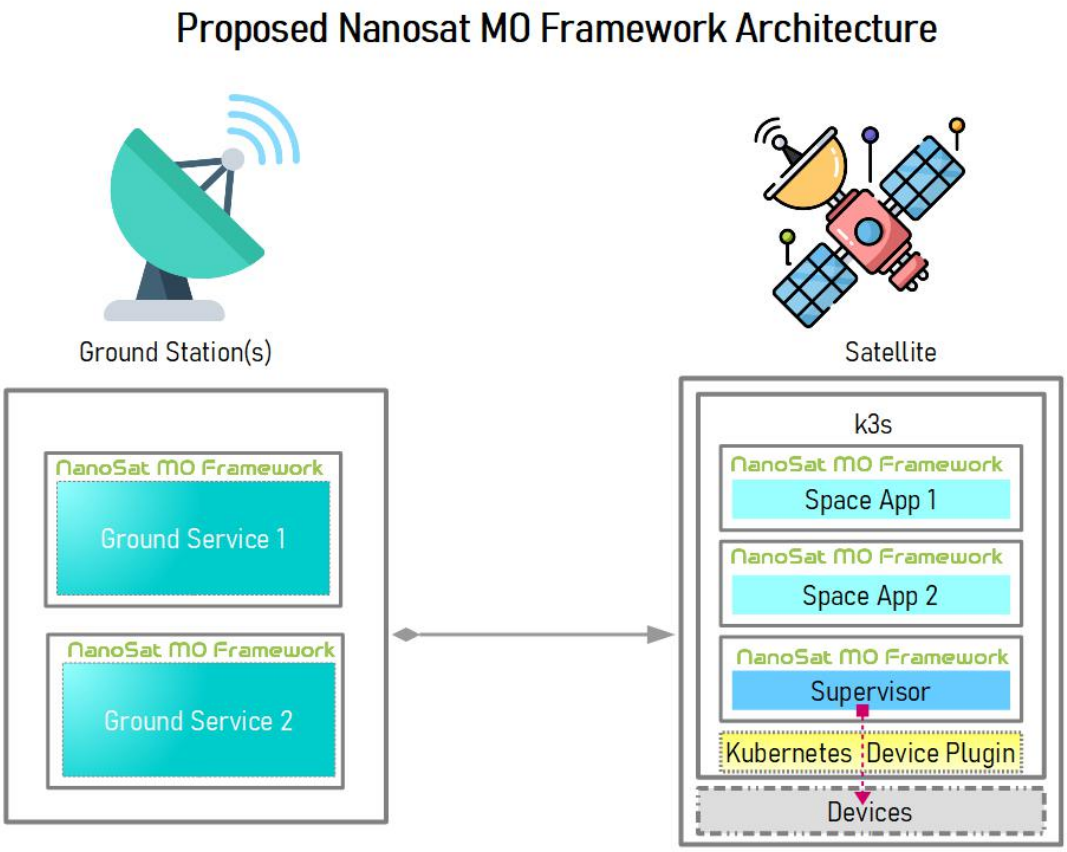

Figure 4.2: Proposed NanoSat MO Framework

4.1.3 Proof Of Concept Architecture

The end to end proof of concept architecture for this thesis comprises three basic elements: the space node, the ground workstation node and the ground CI/CD node.

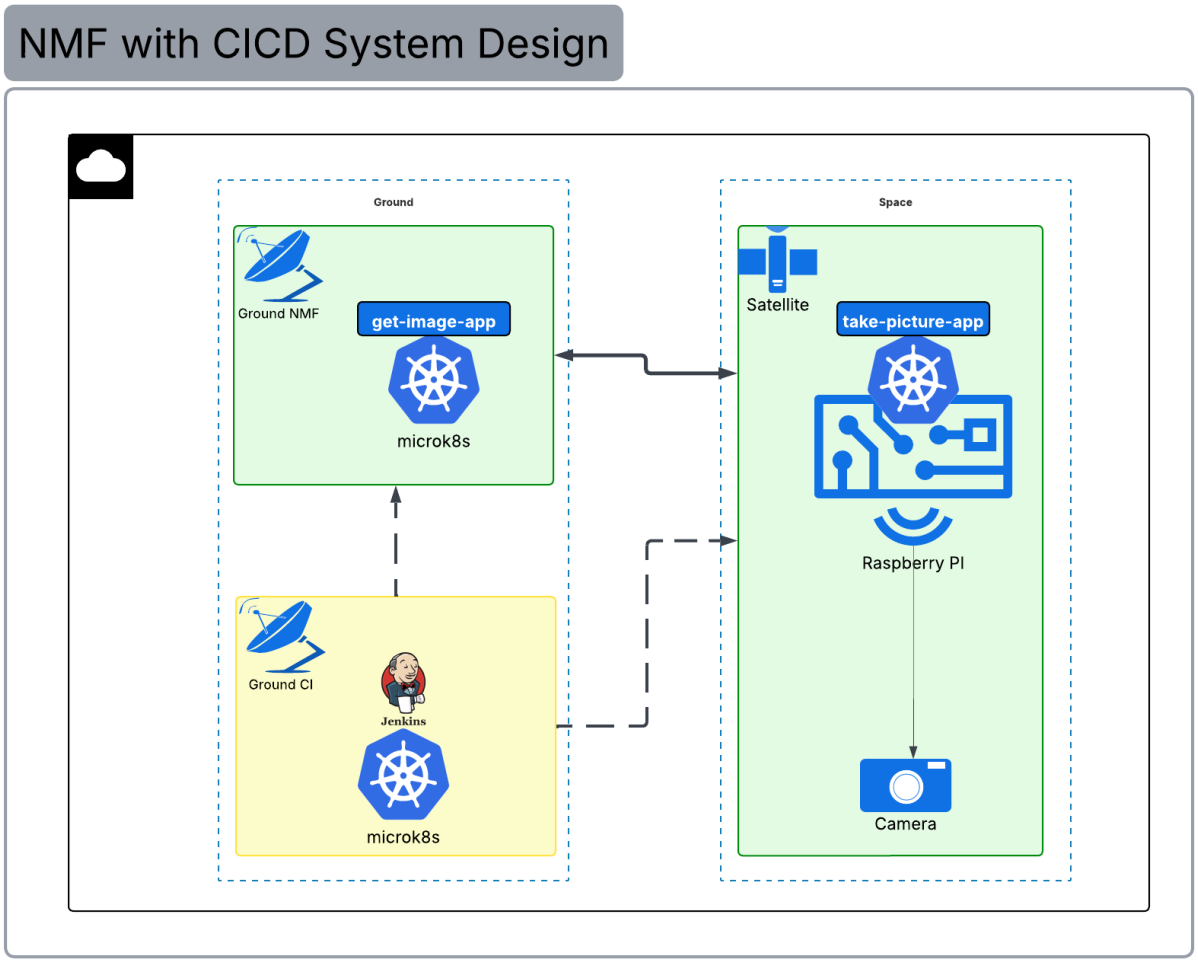

Figure 4.3: Proof of Concept NMF with CICD System Design

Space node Consists of a RaspberryPi, with an attached camera and similar resources as OPS-SAT's computational payload, in which a lightweight k3s cluster [54] is set up and running. A camera acquisition app and Supervisor/platform service is deployed inside this cluster.

Ground workstation node It consists of a development workstation with a ground photo storage app running in a local JVM, which receives preprocessed photos from the space camera acquisition app and stores them locally.

Ground CI/CD node It consists of a VM with Jenkins setup [55] inside a mikrok8s cluster [56][57]. Jenkins is utilized to build container images for the space applications and push them as compressed tarballs in the k3s space cluster in a scheduled manner. Ansible is used to set up the microk8s cluster from scratch. Ansible is an open source, command-

line IT automation software application that can configure systems, deploy software, and orchestrate advanced workflows to support application deployment, system updates, and more [58].

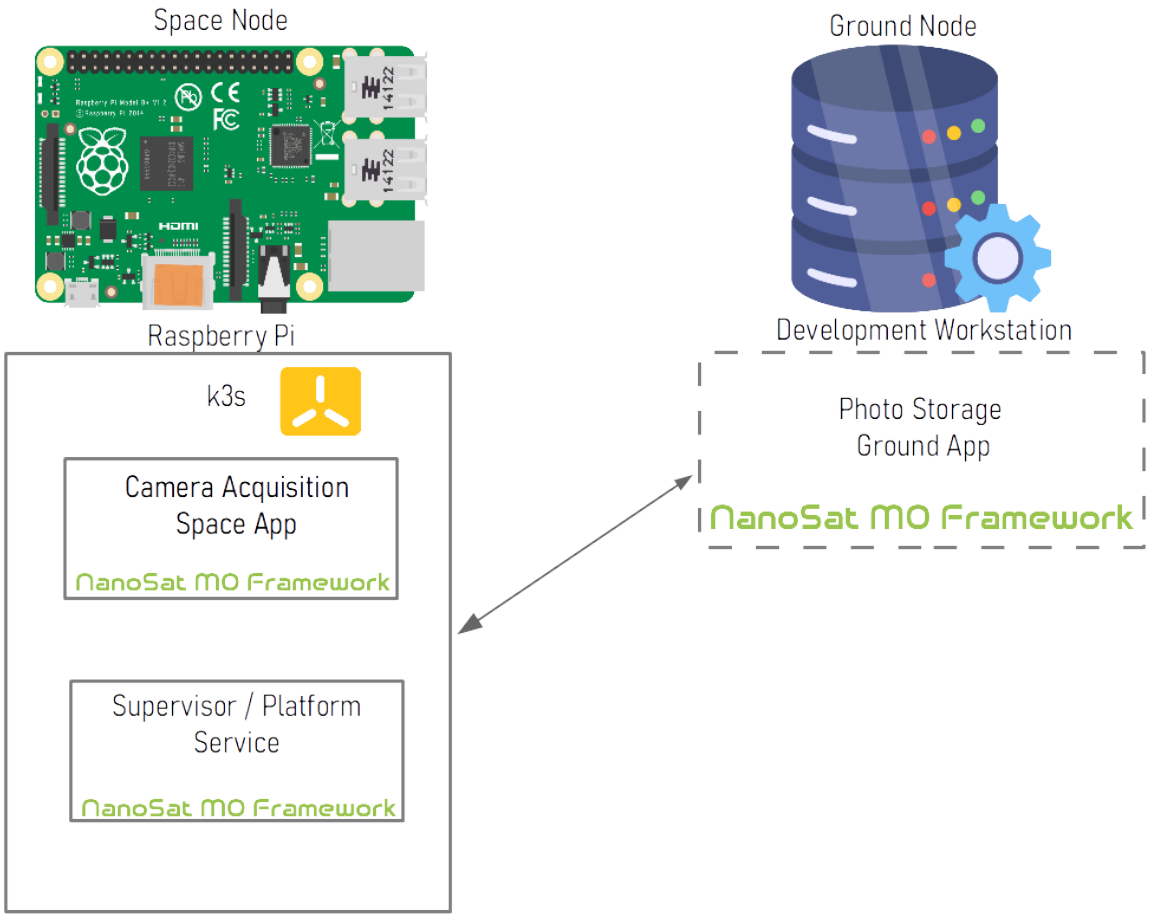

Figure 4.4: Proof of Concept NMF App Architecture

4.1.4 Earth To Space Computational Continuum

By integrating DevOps practices, Kubernetes-based orchestration, and the NanoSat MO Framework (NMF) across satellites, ground stations, and the cloud, it becomes possible to create a seamless Earth-to-Space computing continuum.

In this model, spacecraft operate as edge nodes within a distributed system, running containerized applications close to the data source (e.g., sensors, payloads). Ground segments and cloud infrastructures handle large-scale processing, coordination, and analytics. Using Kubernetes and DevOps pipelines, applications and updates can be deployed, monitored, and managed consistently across all layers, from the cloud to orbiting nanosatellites and even deep-space nodes.

This unified architecture enables real-time data processing, autonomous operations, and scalable mission management, bridging Earth and space into a single adaptive digital ecosystem.

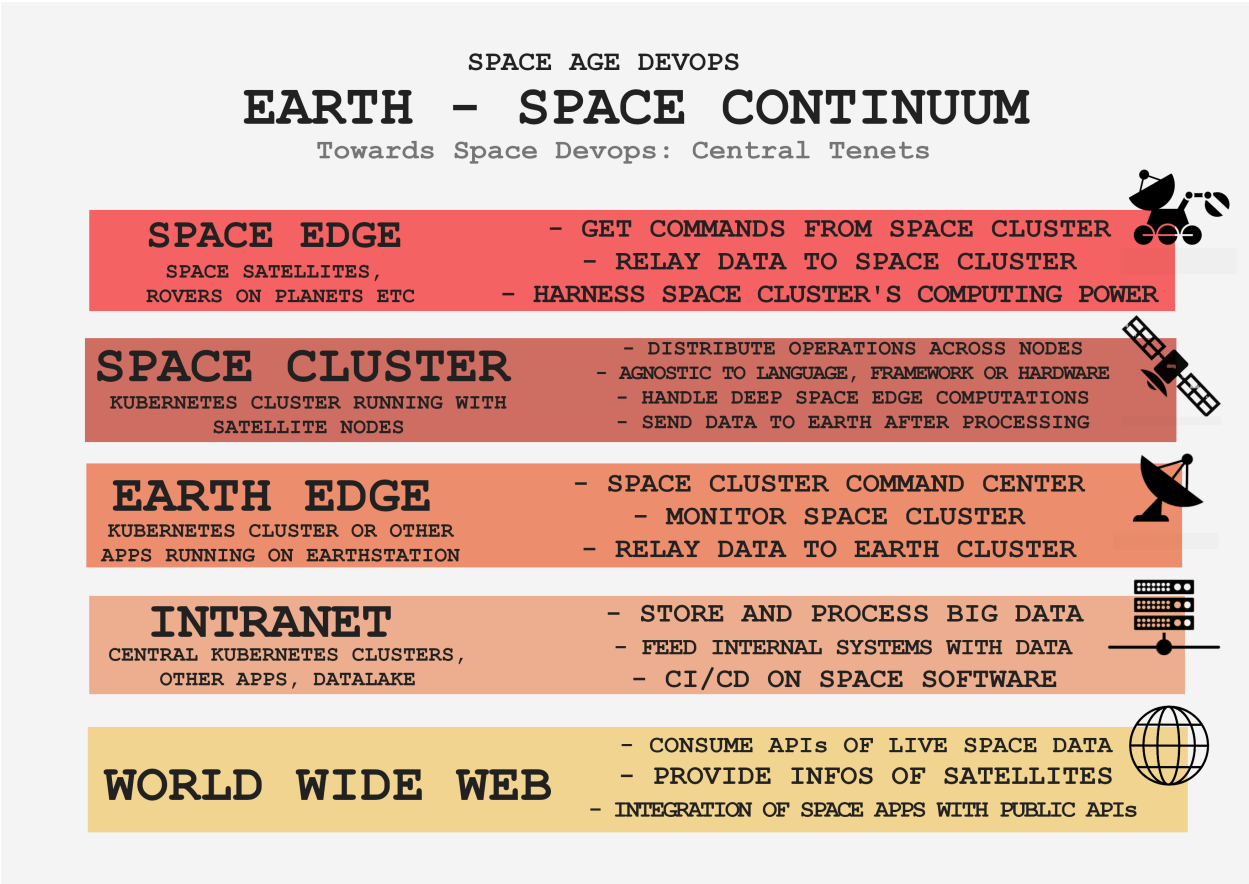

Figure 4.5: Towards Space Devops: Central Tenets

4.2 Methodology

This part of the thesis describes the methodological approach to the resolution of the problem initially formulated. It essentially consists of a breakdown of the path towards converging NanoSat MO Framework with Kubernetes.

4.2.1 Deep-dive into NanoSat MO Framework

The first important step is to get a greater understanding of the NanoSat MO Framework, digging deeper into the CCSDS principles, reviewing all relative available code on ESA's public repositories [59][60][61][62] and observing the actual functionalities of running NMF services.

Thus, in a local workspace the Supervisor service is initially used to start and stop existing NMF services, while getting a grasp of the core functionality provided. Afterward, a custom test application is created based on the NanoSatMO framework. Once it is successfully compiled and deployed, the final step in this stage of the research is to explore the interoperability between ground and space services.

Throughout these steps, all actions and observability are performed using the Consumer Test Tool (CTT). During investigation and initial development, the Supervisor is operated in simulator mode to mock the required underlying infrastructure.

To maintain an organized working environment and follow good development practices, the ESA's public repository is forked into a private workspace, in which branch and commit policies are applied.

4.2.2 Containerize NMF services

The next major step, after gaining a solid understanding of the NanoSat MO Framework, is the containerization of the services. This requires a detailed investigation of how services are originally compiled and executed, as well as an analysis of the scripts and conventions involved, in order to adapt them for containerized environments.

Best practices for containerization are applied, including the use of common base images where possible, like in the case of common libraries. Existing conventions, such as directory structures and paths, are maintained to ensure backward compatibility.

To validate interoperability, testing progresses in stages: first with uncontainerized instances only, then by mixed uncontainerized and containerized instances, and finally between only containerized instances. At each step, functionality is verified using the Consumer Test Tool (CTT), the Supervisor, and the ground/space services. To automate deployment of multiple containers, a Docker Compose file is introduced.

From this stage onward, the Supervisor no longer directly manages the service lifecycle (start/stop). Instead, services are started independently, using docker runtime, and register themselves in the Supervisor. Supervisor remained as a way to interact in a backwards compatible way with those services.

4.2.3 Initial attempt for local Kubernetes deployment

In this step we attempt to create a local Kubernetes deployment, for the NMF ecosystem. Minikube [63] Kubernetes flavour is chosen for the local infrastructure, as it is the ideal lightweight choice for development purposes. Helm charts are created to automate and standardize the deployment of the newly containerized services in the minikube cluster. NMF space and ground apps along with the Supervisor, using simulator mode with mocked underlying hardware.

A significant troubleshooting part of this step is the adaptation of NanoSat MO Framework

to Kubernetes networking. The framework uses MALTCP protocol, which is based on TCP. Kubernetes has a great adaptability regarding communication protocols, however, it is usually hosting apps and services using the application layer TCP/IP network suite [64], for HTTP, SOAP, gRPC transports. A Kubernetes cluster can still accommodate incoming TCP-based traffic from the public network and route it to apps using NodePort Kubernetes Services and binding public static ports to each internal service.

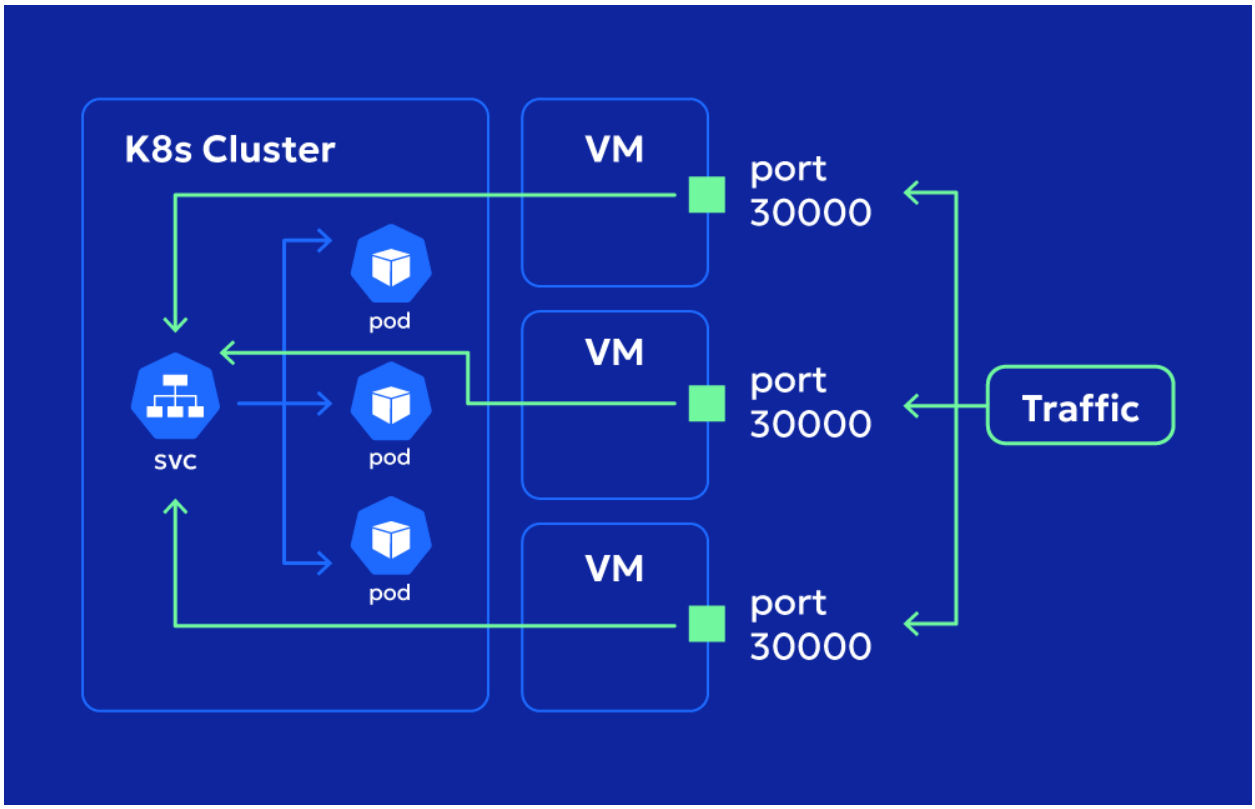

Figure 4.6: NodePort exposes apps on each Node's IP at a static port [65]

Also, the NanoSat MO Framework services exhibit stiff connection requirements. Additional application configurations for host and IP resolution are necessary, after reverse engineering the code and bringing together details from the initial documentation. This process is documented, from now on, in a NMF Kubernetes Networking Guide [6].

4.2.4 Utilize hardware from containerized apps inside Kubernetes

At this stage of the research, all NMF services not dependent on hardware (e.g., ground services) could run within Kubernetes. However, since the framework's primary goal is to deploy services on satellites and utilize onboard hardware (e.g., sensors, cameras, GPS), the next step is to explore effective methods for enabling hardware access from within Kubernetes containers. As containers are largely isolated from the host via Linux

namespaces and cgroups, special configurations are required to grant them access to underlying hardware resources.

An early design consideration is the use of elevated container privileges within a Kubernetes pod to enable access to host hardware. Kubernetes manages container security through Pod Security Standards [66], which define three policy levels: Privileged, Baseline, and Restricted. The "Privileged" policy represents the least restrictive configuration, allowing a pod to bypass most of the isolation mechanisms normally enforced by the container runtime, including direct access to the node's host network, mount host file systems, and directly interact with hardware devices. While this configuration may be appropriate for system-level workloads, such as infrastructure services managed by trusted administrators, it is generally discouraged for application-level workloads due to the significant security risks it introduces. So other approaches need to be investigated also.

4.2.4.1 Kubernetes Device Plugin

Historically, host hardware access in Kubernetes was initially implemented through ad hoc, vendor-specific solutions. For instance, NVIDIA developed a custom integration through a "device plugin" to expose GPUs to containers [67], but such approaches lacked standardization and scalability.

Fortunately, recent Kubernetes versions introduced the Device Plugin framework, providing an official API for developing vendor or hardware specific plugins for mounting devices into Kubernetes, such as GPUs, NICs, FPGAs [68]. This way, there was no longer a need to customize the Kubernetes core code in order to make a device available inside Kubernetes.

The Device Plugin framework was introduced in the Kubernetes v1.8 release as a vendor independent framework to enable discovery, advertisement and allocation of external devices. The feature graduated to Beta in v1.10. With the recent release of Kubernetes v1.26, Device Manager is now generally available (GA) [69].

Within the kubelet, the Device Manager facilitates communication with device plugins using gRPC through Unix sockets. Device Manager and Device plugins both act as gRPC servers and clients by serving and connecting to the exposed gRPC services respectively.

So, the official Kubernetes Device plugin is now integrated in the NMF Kubernetes ecosystem as a full-blown mechanism to access underlying hardware devices. A generic linux device plugin implementation is adopted for this thesis and used as proof of concept for the access of a camera device. Before testing it with the NanoSat MO Framework services, the plugin functionality is validated using a test device and taking pictures from an app running inside minikube.

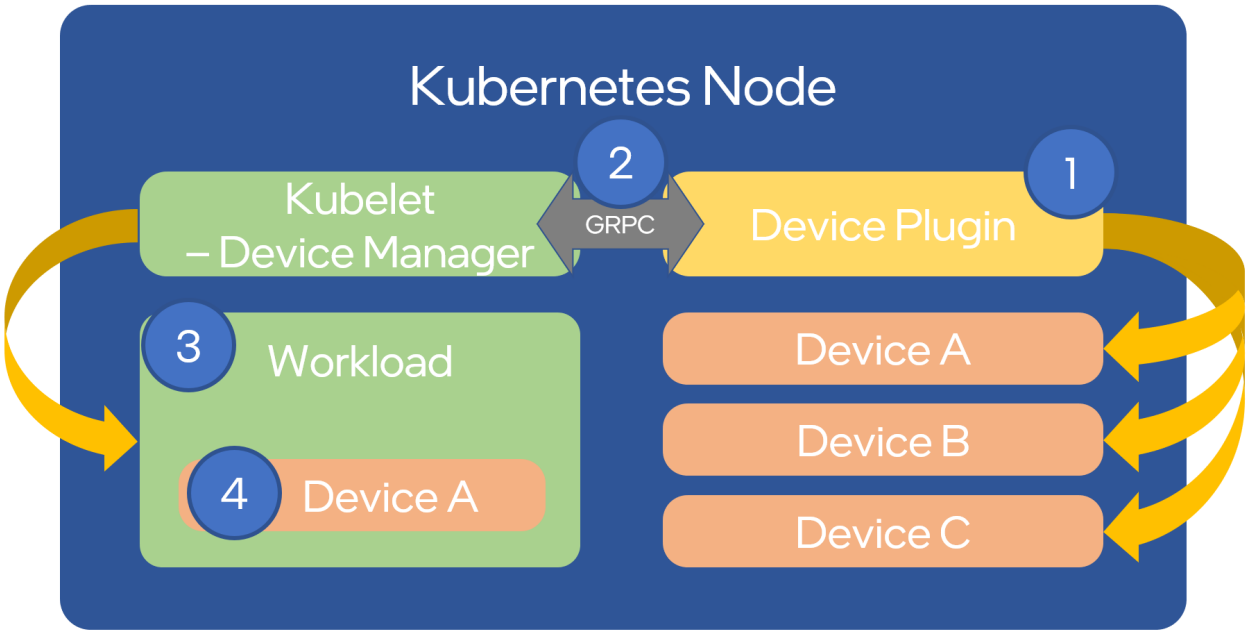

Figure 4.7: Kubernetes Device plugin architecture & gRPC communications [70]

4.2.5 Run final Proof of Concept

In the final proof of concept, we deploy space applications on a k3s cluster upon a Raspberry Pi, acting as a space node, which has computing resources comparable to those of the OpsSat board. In a development workstation, acting as a ground node, all needed ground apps are deployed. A Jenkins set up in a microk8s cluster in a VM is utilized to automate build and deployment of the space apps.

We use an NMF camera space application to capture images through a real camera adapter integrated into the Supervisor service, replacing the previously used mock. Hardware access is managed via Kubernetes device plugin implementation.

Once captured, images are preprocessed onboard using one of two approaches: Sobel filtering for edge detection or K-means clustering for unsupervised segmentation. The processed images are then transmitted via downlink to a ground-based NMF service running on the development workstation, where they are stored.

We validate the full pipeline using this infrastructure and consider the proof of concept successful.

4.3 Design decisions

As part of the implementation, several key design decisions are made to select between alternative approaches. These decisions are guided by trade-offs involving performance, compatibility, system complexity, and alignment with the project's objectives. The following section outlines the most significant choices and the rationale behind them.

4.3.1 Kubernetes flavor selection

After the initial stages of testing and development, the project transitions into the next phase, which focuses on building a proof of concept with advanced infrastructure. The goal is to move beyond basic functionality and develop a setup that better reflects real-world deployment conditions. At this stage, selecting an appropriate Kubernetes distribution becomes a critical decision. Since the time of its creation, Kubernetes has been implemented into a diverse environment of apps, systems and use cases in general, in a multitude of fields; from IoT to data-center domains [71][72].

Given the project's constraints and objectives, the use case aligns most closely with the requirements typically found in IoT scenarios. The most complete flavors of Kubernetes used in IoT are microk8s, k3s and k0s. They offer almost the full capability of the Kubernetes API and require minimum resources.

The system specs of OpsSat, the first NanoSat MO Framework implementation mission in space, need to be taken under consideration. According to its official specifications, OpsSat has two Critical Link MityARM 5CSX in cold redundancy (if one fails, the second one is used). These have a Dual-core 800 MHz ARM Cortex-A9 processor, an Altera Cyclone V FPGA, 1 GB DDR3 RAM, and an external mass memory device with 8 GB [73].

As documented by the comparison depicted in the figure 4.8, OpsSat systems don't provide adequate resources for the microk8s to run smoothly. However, they are more than enough for k3s and k0s, as they have been designed to reduce binary size and memory footprint, targeting resource constrained edge clusters besides developer workstations. Both k3s and k0s support ARM64 system architecture, show high control plane throughput and generally efficient latency and throughput in their overall task completion [74]. Bearing in mind that k3s has the biggest community, sufficient documentation and ease of use, it is chosen as the most suitable Kubernetes flavour for this specific scenario.

	MicroK8s	k3s	k0s	MicroShift
Key Developer	Canonical	Rancher/SuSE	Mirantis	Red Hat
License	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0
Enterprise Support	Yes	Yes	Yes	Yes
GitHub repo	https://github.com/canonical/microk8s	https://github.com/k3s-io/k3s	https://github.com/k0sproject/k0s	https://github.com/openshift/microshift
GitHub stars	6800	21200	105	406
Contributors	146	1796	65	46
First commit	May 2018	January 2019	July 2020	April 2021
Programming Language	Python, Shell	Go	Go	Go
CNCF certified	Yes	Yes	Yes	No
Vanilla Kubernetes	Yes	Yes	Yes	Yes
Single-node cluster	Yes	Yes	Yes	Yes
Multi-node cluster	Yes	Yes	Yes	n/a
Airgap cluster	Yes	Yes	Yes	Yes
High availability	Yes	Yes	Yes	n/a
GPU acceleration	Yes	Yes	Yes	Yes
Operating System	Ubuntu (default), Linux, Windows, MacOS	Linux	Linux, Windows Server 2019 (experimental)	RHEL, CentOS Stream, Fedora, (Windows, MacOS)
CPU Architecture	x86, ARM64, s390x, Power9	x86, ARM64, ARMhf	x86-64, ARM64, ARMv7	x86_64, ARM64, RISCV64
Deployment	Snap Package	Single Binary	Single Binary	RPM Package
Container runtime	containerd (default), kata	containerd (default), docker, custom	containerd (default), custom (e.g., docker)	cri-o (default)
Container network interface	Calico, Flannel	Flannel (default), custom CNI	Kube-Router (default), Calico, custom	Flannel (default), crio-bridge
Control plane datastore	dqlite (default), custom	SQLite (default), PostgreSQL, MySQL, MariaDB, etcd, Embedded etcd	etcd (default), custom (e.g., SQLite, PostgreSQL, MySQL)	etcd (default)
Recommended minimal CPU	2 CPU cores	1 CPU	1 CPU	2 CPU cores
Recommended minimal RAM	4 GB RAM	1 GB RAM	1 GB RAM	2 GB RAM
Advertised memory consumption	540 MB	512 MB	510 MB	n/a

Figure 4.8: Feature comparison of lightweight Kubernetes distributions [74]

4.3.2 Container runtime selection

Docker container runtime and CLI are chosen for local development purposes due to Docker's ecosystem popularity and ease of use. Instead, containerd is adopted in the final proof of concept infrastructure as the most suitable container runtime, being more lightweight.

Containerd is the low-level container runtime that Docker is built upon. Whereas Docker focuses on providing a streamlined user experience, containerd implements the system daemon that lets Docker actually start and run containers [75].

Containerd focuses on providing a stable foundation for other container tools, but Docker is a complete platform for building and operating containers. It's a developer-oriented solution that includes many complementary features not found in containerd.

4.3.3 Jenkins Pipeline: Scheduling, Tarballs & SCP

In our Jenkins-based delivery and deployment setup, standard workflows such as tagging, releasing, and testing remain the same. However, deployment tactics are adapted to meet the unique requirements of space-edge systems [76][77].

Unlike conventional deployments where container images are pulled from an online registry, our Jenkins pipeline builds the images on the ground and then waits for synchronization with the satellite in orbit. When an uplink becomes available, the container images are exported as tarballs and securely transferred using scp command [78]. Once received, the tarballs are unpacked and mounted into the satellite's local container registry and k3s environment.

Using container tarballs instead of performing docker pull operations provides the following practical advantages for in-orbit deployment:

- Predictable and lower host resource usage: pulling images via Docker registry requires multiple concurrent downloads, decompression, and verification steps, which can spike host RAM and CPU usage. Loading tarballs, by contrast, is a sequential, local operation that avoids these transient resource peaks.

- Optimized data transfer: a single, compressed tarball can be efficiently transmitted improving reliability during short uplink windows.

For file transfer, scp was selected for its simplicity over rsync which needed the software installed on both ends [79]. Meanwhile, CFDP (CCSDS File Delivery Protocol) [80] and the Bundle Protocol [81] are purpose-built for space environments, designed to handle long delays and disruption-tolerant networking. However, these protocols were not integrated into this proof of concept due to the complexity of adoption and integration overhead.

4.3.4 Raspberry Pi as the PoC infrastructure

The initial point of reference for NMF is the OPS-SAT mission, which utilizes a Cyclone V SoC FPGA board [82][83]. Raspberry Pi is selected for executing this thesis' running example due to its popularity, simplicity and similarity with Cyclone V in terms of computational resource capabilities (RAM, CPU, storage). Implementing an architecture combining DevOps technologies and principles with space software is under the spotlight, before actually running all those technologies in the real-world underlying infrastructure. Also, gradually it has become a common practice to launch single-board computers like Raspberry Pi and Nvidia Jetson [84] in satellite missions, particularly close to LEO, due to their low cost, availability and increased compute power.

4.3.5 E2E Space-Ground Proof Of Concept

The full-blown proof of concept is conducted in a Raspberry Pi (with the role of a satellite) with a similar system specification to Altera Cyclone V used in OPS-SAT (1 GB RAM) and a camera device connected to it. A k3s cluster is set up in Raspberry-pi with the containerized NMF Supervisor and an NMF camera-acquisition space app deployed in it. The deployment was achieved by the use of automated pipelines with helm charts and Jenkins. Then, the NMF Supervisor (with unmocked camera platform service) was utilized for taking snapshots, successfully retrieving live photos from the camera attached to the underlying hardware, through the Kubernetes device plugin integration. These photos were sent to the camera-acquisition space app to be processed further, with image analysis filters and machine learning algorithms. Eventually, the images are sent to a ground application running on a development workstation.

This proof of concept architecture was designed having in mind of being as close to a real scenario as possible:

- similar computational resources used
- space-ground communication
- data preprocessing in space as part of edge computing logic
- hardware integration with Kubernetes ecosystem
- CI/CD pipelines provisioned for space environment

4.3.6 JDK vs JRE

Maven and JDK-based base image is used to compile and package the Java code in the build stage of the Dockerfile. In the final container image, however, only the JRE (Java Runtime Environment) is included. This is accomplished using a multi-stage Docker build,

where compilation tools are confined to the build phase and excluded from the runtime environment.

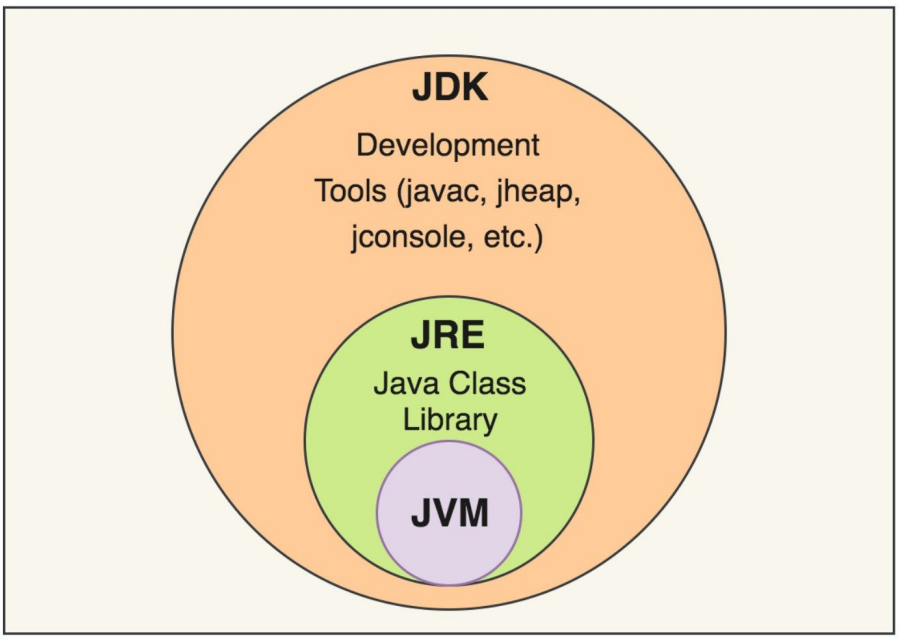

Figure 4.9: JDK, JRE, JVM components of Java ecosystem [85]

The JRE includes the JVM, standard class libraries, and supporting files needed to run Java programs. The JDK includes the JRE plus development tools like javac (compiler), javadoc, jdb (debugger). This approach reduces the final image size, enabling faster deployments and more efficient resource utilization during execution [86].

4.4 Implications and Discussions

- A new version of Nanosat Framework has been released in ESA's Github repository after the start of the development in the forked Github repository. The forked repository has been updated to retrieve any new changes but certainly some new features or improvements, created by the ESA team, could still be left out. The main goal of this thesis remains intact, as it is to pave the way to the new DevOps era in space through the prism of NMF, white highlighting and documenting the way to do it.
- The selection of a Kubernetes distribution should align with available system resources and specific operational requirements. Simulating an OPS-SAT mission on a resource-constrained Raspberry Pi made k3s an optimal choice due to its lightweight footprint and broad community. However, alternative distributions may be more suitable in different contexts, especially given the fact that these distributions grow and transform over time. k0s could be an ideal alternative in the future,

keeping a lightweight footprint while lacking a big support community at the moment. Microk8s supports system architectures like s390x and POWER9 that are not supported by k3s. Also, full Kubernetes (k8s) deployments are preferable for large-scale, resource abundant clusters demanding extensive customization, high availability, and integration with enterprise tooling.

- For the proof of concept, the NMF Supervisor was utilized in hybrid simulator mode, where certain hardware components, such as GPS and ADCS sensors, were simulated giving mocked responses and metrics, while the actual camera device was fully integrated with its corresponding platform service. This setup enabled us to demonstrate end-to-end NMF functionality within a Kubernetes environment without the need to implement full hardware adapters for each device. The hybrid approach was chosen to balance development effort and demonstration scope, focusing on validating system integration rather than exhaustive hardware emulation.

- Raspberry Pi is a good emulation platform for OPS-SAT for software development, NMF integration, and ground-segment testing purposes, but it is not a substitute for real hardware testing or in-orbit operational behavior. It is especially suitable for early-stage development, training, and POCs (like demonstrating NMF functionality in Kubernetes or containerized environments). It is based on Linux-based OS, ARM64 system architecture, utilizing the same range of resources with OPS-SAT's Cyclone V SoC FPGA and providing interfaces for many IO devices. For sure, further validation is required regarding RaspberryPi's hardware radiation tolerance and fault tolerance mechanisms required in space. Also, RaspberryPI is probably lacking out of the box support for specific space-grade sensors (ADCS) and doesn't provide FPGA functionalities.

- Raspberry Pi can be considered conditionally a full example of a satellite on-board computer, particularly in the context of modern, low-cost, LEO missions, prototyping or educational satellites. As the space industry evolves, more small satellites are indeed using commercial off-the-shelf (COTS) hardware like Raspberry Pi, NVIDIA Jetson, or BeagleBone, especially when cost, time-to-launch and flexibility are prioritized. There are real-world examples, like Astro Pi [87], a Raspberry Pi flown to the International Space Station (ISS), used by students for experiments and CubeSats launched from SpaceX with an Nvidia Jetson board for AI-based imaging [88], mission control, or data processing.

- The container image tarballs copied from the Jenkins terrestrial infrastructure could ideally be transferred to the satellite cluster using CCSDS protocols such as CFDP and Bundle protocol. Bundle protocol is a network layer protocol that enables Delay/Disruption-Tolerant Networking (DTN), it forms a store-and-forward overlay network that can move data "bundles" between intermittently connected nodes, an analogy to space IP. CFDP is an application layer protocol for reliable file transfer in space missions, an analogy to space SCP. While CFDP and the Bundle Protocol are well-suited for space communications and disruption-tolerant networking, their integration would have significantly increased system complexity. As a trade-off, these protocols were

not adopted for this proof of concept, prioritizing simplicity and rapid demonstration over full DTN capability.

5. IMPLEMENTATION

5.1 Enhance platform services: Implement RaspberryPi USB camera adapter

In the NMF Supervisor, the Platform services provide access to a satellite's hardware (sensors, actuators, etc.) in a standardized, mission-operations compliant way. To make that possible across different kinds of hardware, adapter classes are used as hardware abstraction layers, one adapter per platform service or subsystem.

For our case, a new adapter, FsWebcameraAdapter, was created and used by the Camera platform service of the supervisor to enable interaction with the actual underlying camera hardware of a Raspberry PI. This functionality was implemented using the Java Webcam Capture library [89] together with fswebcam [90], a lightweight webcam handling tool for Unix-like systems.

```
<dependency>
    <groupId>com.github.sarxos</groupId>
    <artifactId>webcam-capture</artifactId>
    <version>0.3.13-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>com.github.sarxos</groupId>
    <artifactId>webcam-capture-driver-fswebcam</artifactId>
    <version>0.3.13-SNAPSHOT</version>
</dependency>
```

Figure 5.1: Maven artifacts required for RaspberryPi camera platform service

For each platform service, an interface defines the fundamental methods and functionalities to be supported. In this context, the FsWebcameraAdapter implements the set of methods inherited from the Supervisor's Camera Adapter Interface, including operations for obtaining available resolutions and formats, previewing images, and capturing snapshots.

```java
    @Override
    public Picture takePicture(final CameraSettings settings) throws IOException {
        synchronized (this) {
            LOGGER.log(Level.INFO, "FsWebcamAdapter ready to take picture");
            Webcam webcam = Webcam.getDefault();
            if (webcam == null) {
                throw new IOException("Cannot find camera device.");
            }

            webcam.setViewSize(new Dimension((int)
settings.getResolution().getWidth().getValue(),
                    (int) settings.getResolution().getHeight().getValue()));
            LOGGER.log(Level.INFO, "Resolution set.");

            webcam.open();
            LOGGER.log(Level.INFO, "Opened camera");

            final Time timestamp = HelperTime.getTimestampMillis();
            BufferedImage image = webcam.getImage();
            LOGGER.log(Level.INFO, "Snapshot taken");

            webcam.close();
            LOGGER.log(Level.INFO, "Closed camera");

            byte[] data = convertImage(image, settings.getFormat());
            LOGGER.log(Level.INFO, "Image converted to byte array");

            final CameraSettings replySettings = new CameraSettings();
            replySettings.setResolution(settings.getResolution());
            replySettings.setExposureTime(settings.getExposureTime());
            replySettings.setGainRed(settings.getGainRed());
            replySettings.setGainGreen(settings.getGainGreen());
            replySettings.setGainBlue(settings.getGainBlue());

            replySettings.setFormat(settings.getFormat());
            return new Picture(timestamp, replySettings, new Blob(data));
        }
    }
```

Figure 5.2: Take a snapshot

```
    @Override
    public PixelResolutionList getAvailableResolutions() {
        final PixelResolutionList availableResolutions = new PixelResolutionList();
        Webcam webcam = Webcam.getDefault();
        for (Dimension dimension : webcam.getViewSizes()) {
            availableResolutions.add(new PixelResolution(
                    new UInteger((long) dimension.getWidth()), new UInteger((long)
dimension.getHeight())));
        }
        return availableResolutions;
    }
```

Figure 5.3: Get camera's available resolutions

In order to utilize FsWebcamAdapter in the final proof of concept the platforsim.properties configuration file needs to be configured properly, which is used to define simulation parameters for the Platform Services. The Supervisor needs to be executed in hybrid mode and define each adapter class, mocked or real, that is utilized for each platform service. In hybrid mode, the NMF Supervisor uses a mix of real and simulated platform services. That is, the camera subsystem will use a real hardware adapter, while the others are still simulated through the platform simulator.

```
platform.mode=hybrid
pc.adapter=esa.mo.platform.impl.provider.softsim.PowerControlSoftSimAdapter
camera.adapter=esa.mo.platform.impl.provider.real.FsWebcamAdapter
adcs.adapter=esa.mo.platform.impl.provider.softsim.AutonomousADCSSoftSimAdapter
gps.adapter=esa.mo.platform.impl.provider.softsim.GPSSoftSimAdapter
optrx.adapter=esa.mo.platform.impl.provider.softsim.OpticalDataReceiverSoftSimAdapter
sdr.adapter=esa.mo.platform.impl.provider.softsim.SoftwareDefinedRadioSoftSimAdapter
clock.adapter=esa.mo.platform.impl.provider.softsim.ClockSoftSimAdapter
```

Figure 5.4: Configuration to use FsWebcamAdapter in Supervisor hybrid mode

5.2 Enable running NMF into Kubernetes: k3s, Device plugin & CICD pipelines

5.2.1 k3s setup

k3s bundles its own specific Kubernetes version with each release. The K3s release version is typically denoted with a postfix like "+k3sX", where "X" is a number. For example, the release v1.32.5+k3s1 uses Kubernetes v1.32.5. The "+k3sX" postfix allows K3s to

make additional releases while maintaining semantic versioning. So k3s version later than 1.26 is used in this software solution, in order to utilize the device plugin.

Version	Release date	Kubernetes	Kine	SQLite	Etcd	Containerd	Runc	Flannel	Metrics-server	Traefik	CoreDNS	Helm-controller	Local-path-provisioner
v1.32.9+k3s1	Sep 22 2025	v1.32.9	v0.14.0	3.50.4	v3.5.21-k3s1	v2.1.4-k3s1.32	v1.2.7	v0.27.0	v0.8.0	v3.3.6	v1.12.3	v0.16.13	v0.0.31
v1.32.8+k3s1	Aug 25 2025	v1.32.8	v0.13.17	3.49.1	v3.5.21-k3s1	v2.0.5-k3s2.32	v1.2.6	v0.27.0	v0.8.0	v3.3.6	v1.12.3	v0.16.13	v0.0.31
v1.32.7+k3s1	Jul 26 2025	v1.32.7	v0.13.17	3.49.1	v3.5.21-k3s1	v2.0.5-k3s2.32	v1.2.6	v0.27.0	v0.7.2	v3.3.6	v1.12.1	v0.16.13	v0.0.31
v1.32.6+k3s1	Jun 30 2025	v1.32.6	v0.13.15	3.49.1	v3.5.21-k3s1	v2.0.5-k3s1.32	v1.2.6	v0.27.0	v0.7.2	v3.3.6	v1.12.1	v0.16.11	v0.0.31
v1.32.5+k3s1	May 23 2025	v1.32.5	v0.13.15	3.49.1	v3.5.21-k3s1	v2.0.5-k3s1.32	v1.2.6	v0.26.7	v0.7.2	v3.3.6	v1.12.1	v0.16.10	v0.0.31
v1.32.4+k3s1	May 01 2025	v1.32.4	v0.13.14	3.46.1	v3.5.21-k3s1	v2.0.4-k3s2	v1.2.5	v0.26.7	v0.7.2	v3.3.6	v1.12.1	v0.16.10	v0.0.31
v1.32.3+k3s1	Mar 25 2025	v1.32.3	v0.13.9	3.46.1	v3.5.19-k3s1	v2.0.4-k3s2	v1.2.5	v0.25.7	v0.7.2	v3.3.2	v1.12.0	v0.16.6	v0.0.31
v1.32.2+k3s1	Feb 27 2025	v1.32.2	v0.13.9	3.46.1	v3.5.18-k3s1	v2.0.2-k3s2	v1.2.4-k3s1	v0.25.7	v0.7.2	v3.3.2	v1.12.0	v0.16.6	v0.0.31
v1.32.1+k3s1	Jan 28 2025	v1.32.1	v0.13.5	3.46.1	v3.5.16-k3s1	v1.7.23-k3s2	v1.2.4-k3s1	v0.25.7	v0.7.2	v2.11.18	v1.12.0	v0.16.5	v0.0.30
v1.32.0+k3s1	Jan 10 2025	v1.32.0	v0.13.5	3.46.1	v3.5.16-k3s1	v1.7.23-k3s2	v1.2.1-k3s1	v0.25.7	v0.7.2	v2.11.10	v1.12.0	v0.16.5	v0.0.30

Figure 5.5: Dependency matrix for k3s version 1.32.X [91]

k3s provides an installation script that is a convenient way to install it as a service on systemd based systems. To install the latest K3s in the Raspberry Pi (for specific version set environmental variable *INSTALL_K3S_VERSION=vX.Y.Z+k3s1*), the following command is executed: *curl -sfL https://get.k3s.io | sh -*

After running this installation:

- The K3s service will be configured to automatically restart after node reboots or if the process crashes or is killed
- Additional utilities will be installed, including kubectl, crictl, ctr, k3s-killall.sh, and k3s-uninstall.sh
- A kubeconfig file will be written to /etc/rancher/k3s/k3s.yaml and the kubectl installed by K3s will automatically use it

A single-node server installation is a fully-functional Kubernetes cluster, including all the datastore, control-plane, kubelet, and container runtime components necessary to host workload pods. Adding additional server or agent nodes is an option if there is a strict

requirement for additional capacity or redundancy to the cluster. However, single-node server installation is considered ideal for this proof of concept.

The last needed provisioning for the k3s installation in a resource restrictive environment is to limit the amount of RAM that can be allocated to it.

A file is created in /etc/systemd/system/k3s-memory-limit.slice:

```
[Slice]
MemoryAccounting=true
MemoryLimit=4096M  # Set the memory limit to 4 GB
```

Figure 5.6: Slice definition in systemd for k3s memory limits

The k3s service specs under /etc/systemd/system/k3s.service are edited:

```
[Service]
Slice=k3s-memory-limit.slice
```

Figure 5.7: k3s service override assigning k3s to the custom slice

k3s is reloaded by executing commands a) systemctl daemon-reload, b) systemctl restart k3s

5.2.2 Enabling Device plugin to mount camera

Enabling hardware device access within the NMF Supervisor's Platform Services requires the use of the Kubernetes Device Plugin API, the optimal method for exposing such hardware.

A Generic Device Plugin implementation [92], utilizing Kubernetes Device Plugin framework, is adopted. This enables allocating generic Linux devices, such as serial devices or video cameras, to Kubernetes Pods and allows devices that don't require special drivers to be advertised to the cluster and scheduled. To install the generic-device-plugin a Daemonset manifest [93] is included in the helm chart that defines all the appropriate Linux directory paths in which each device is expected to be available from the host machine. Then in the Supervisor helm chart video device capture allocation is enabled for the Supervisor pod. Thus when referring to the "/dev/video0" directory from within the Supervisor app then the host machine's published directory is used.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: generic-device-plugin
  namespace: kube-system
  labels:
    app.kubernetes.io/name: generic-device-plugin
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: generic-device-plugin
  template:
    metadata:
      labels:
        app.kubernetes.io/name: generic-device-plugin
    spec:
      priorityClassName: system-node-critical
      tolerations:
      - operator: "Exists"
        effect: "NoExecute"
      - operator: "Exists"
        effect: "NoSchedule"
      containers:
      - image: squat/generic-device-plugin
        args:
        - --device
        - |
          name: video
          groups:
            - paths:
                - path: /dev/video0
```

Figure 5.8: Part of generic device plugin DaemonSet manifest added in helm chart

```yaml
resources:
  limits:
    squat.ai/video: 1
```

Figure 5.9: Enable video device capture allocation in Pod manifest

5.2.3 CI/CD pipelines

An Ansible role [94] - a reusable automation unit that defines tasks and configurations - is used to install MicroK8s on an Ubuntu virtual machine, where Jenkins will also be deployed. The role is customized to override default settings, specifically modifying the API server's secure port configuration.

```
- name: Change API server secure port to {{
microk8s_config_api_server_secure_port }}
  become: true
  ansible.builtin.replace:
    path: /var/snap/microk8s/current/args/kube-apiserver
    regexp: '--secure-port=\d+'
    replace: '--secure-port={{ microk8s_config_api_server_secure_port }}'
  when: microk8s_config_api_server_secure_port is defined

- name: Replace current secure port to {{
microk8s_config_api_server_secure_port }} in all files
  become: true
  replace:
    path: "{{ item.path }}"
    regexp: 16443
    replace: '{{ microk8s_config_api_server_secure_port }}'
  with_items: "{{ args.files }}"

- name: Replace current server port in kube config file
  replace:
    path: "~/.kube/config"
    regexp: 16443
    replace: '{{ microk8s_config_api_server_secure_port }}'

- name: Replace current secure port 16443 to {{
microk8s_config_api_server_secure_port }} in all files
  become: true
  replace:
    path: "{{ item.path }}"
    regexp: 16443
    replace: '{{ microk8s_config_api_server_secure_port }}'
  with_items: "{{ config_overrides.files }}"
```

Figure 5.10: microk8s secure port overridden in Ansible role

This step requires the prior installation of Ansible package in the VM and execution of a playbook that runs the customized microk8s Ansible role, using the command *"ansible-playbook playbook.yml"*.

```
- hosts: servers
  roles:
    - role: istvano.microk8s
      vars:
        microk8s_plugins:
          dns: "1.1.1.1"
          istio: true
          ingress: true
```

Figure 5.11: Playbook to run Ansible role for microk8s

After the microk8s cluster creation, Jenkins is deployed in it using helm charts. The appropriate values.yaml and Jenkinsfile files are provided to define the pipelines for building NMF apps, exporting them as tarballs and scheduling their deployment-transfer to the satellite.

5.3 Packaging and standardization: Add dockerfiles and Kubernetes Helm charts

Three distinct Dockerfiles have been created to support the Supervisor, Space, and Ground applications container image creation. Each Dockerfile employs a two-stage approach: the first stage uses a Maven 3.8.1 with JDK 11 image to build fat JARs from the repository, ensuring consistent and reproducible builds. The second stage is based on the Eclipse Temurin JRE 11 (Jammy) distribution as a lightweight runtime environment for the deployed containers.

While all Dockerfiles retain a similar approach, each file incorporates its own set of configuration files, environment variables, and service-specific adjustments, enabling tailored behavior for each component while maintaining a unified build and deployment strategy. All Dockerfiles specify the platform option used to produce container images compatible with ARM64 architecture, thus compatible with Raspberry Pi. Instead, in the Supervisor Dockerfile additional libraries required for hardware integration are installed, therefore v4l-utils and fswebcam are included to support the Raspberry Pi FS Camera Adapter.

```
FROM --platform=linux/arm64/v8 maven:3.8.1-jdk-11-slim AS builder
WORKDIR /build
COPY core /build/core
COPY parent /build/parent
COPY mission/simulator /build/mission/simulator
COPY sdk /build/sdk
COPY pom.xml /build/pom.xml
COPY logging.properties /build/logging.properties
RUN mvn clean install -P assembly-with-dependencies


# JRE (not JDK) compatible with arm64/v8
FROM --platform=linux/arm64/v8 eclipse-temurin:11-jre-jammy

# Install camera library dependencies
RUN apt-get update && apt-get install -y v4l-utils && apt-get install -y fswebcam

# Use project version
ARG VERSION=${VERSION}

# Copy supervisor jar and all corresponding property files in the same container directory
WORKDIR /opt/supervisor
COPY --from=builder /build/core/nmf-composites/nanosat-mo-supervisor/target/nanosat-mo-
supervisor-${VERSION}-jar-with-dependencies.jar ./supervisor.jar
COPY logging.properties ./
COPY sdk/sdk-package/src/main/resources/space-common/*.properties ./
COPY sdk/sdk-package/src/main/resources/space-supervisor-root/*.properties ./
COPY mission/simulator/opssat-spacecraft-simulator/platformsim.properties ./

# Run jar
CMD ["java", "-jar", "supervisor.jar"]
```

Figure 5.12: Dockerfile for Supervisor

```
FROM --platform=linux/arm64/v8 maven:3.8.1-jdk-11-slim AS builder
WORKDIR /build
COPY core /build/core
COPY parent /build/parent
COPY mission/simulator /build/mission/simulator
COPY sdk /build/sdk
COPY pom.xml /build/pom.xml
COPY logging.properties /build/logging.properties
RUN mvn clean install -P assembly-with-dependencies


# JRE (not JDK) compatible with arm64/v8
FROM --platform=linux/arm64/v8 eclipse-temurin:11-jre-jammy

# Use given arguments for module fetching
ARG VERSION=${VERSION}
ARG MODULE_PATH=${MODULE_PATH}
ARG MODULE_NAME=${MODULE_NAME}

# Load Supervisor central directory URI from env variable
ENV SUPERVISOR = ${SUPERVISOR}

# Copy module jar and all corresponding property files in the same container directory
WORKDIR /opt/space-module
COPY --from=builder /build/${MODULE_PATH}/target/${MODULE_NAME}-${VERSION}-jar-with-dependencies.jar
./module.jar
COPY logging.properties ./
COPY sdk/sdk-package/src/main/resources/space-common/*.properties ./
COPY sdk/sdk-package/src/main/resources/space-app-root/*.properties ./

# Run jar
CMD ["sh","-c", "java -jar -Desa.mo.nmf.centralDirectoryURI=$SUPERVISOR module.jar"]
```

Figure 5.13: Dockerfile for space app

```
FROM --platform=linux/arm64/v8 maven:3.8.1-jdk-11-slim AS builder
WORKDIR /build
COPY core /build/core
COPY parent /build/parent
COPY mission/simulator /build/mission/simulator
COPY sdk /build/sdk
COPY pom.xml /build/pom.xml
COPY logging.properties /build/logging.properties
RUN mvn clean install -P assembly-with-dependencies


# JRE (not JDK) compatible with arm64/v8
FROM --platform=linux/arm64/v8 eclipse-temurin:11-jre-jammy

# Use given arguments for module fetching
ARG VERSION=${VERSION}
ARG MODULE_PATH=${MODULE_PATH}
ARG MODULE_NAME=${MODULE_NAME}

# Load Supervisor central directory URI from env variable
ENV SUPERVISOR = ${SUPERVISOR}

# Copy module jar and all corresponding property files in the same container directory
WORKDIR /opt/ground-module
COPY --from=builder /build/${MODULE_PATH}/target/${MODULE_NAME}-${VERSION}-jar-with-dependencies.jar
./module.jar
COPY logging.properties ./
COPY sdk/sdk-package/src/main/resources/space-common/*.properties ./
COPY sdk/sdk-package/src/main/resources/ground-consumer-root/*.properties ./

# Run jar
CMD ["sh","-c", "java -jar -Desa.mo.nmf.centralDirectoryURI=$SUPERVISOR module.jar"]
```

Figure 5.14: Dockerfile for ground app

Standardization of the deployment process is further guaranteed through the implementation of Helm charts [95], with two types defined: one tailored to the Supervisor and another shared among the Space applications. Each chart template helps provision the Deployment, ConfigMap, Service (configured as NodePort when publicly exposed), and Horizontal Pod Autoscaler Kubernetes resource for the NMF services. Beyond deployment automation, Helm charts bring version control and reproducibility to the ecosystem, enabling different NMF components to be deployed, updated, or rolled back with minimal effort.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ template "fullname" . }}
  labels:
    app: {{ template "name" . }}
    chart: {{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - name: "http-{{ .Values.service.port }}"
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.port }}
  {{- if eq .Values.service.type "NodePort"}}
      nodePort: {{ .Values.service.port }}
  {{- end }}
      protocol: TCP
  {{- range $p := .Values.app.ports }}
    - name: "http-{{ $p }}"
      port: {{ $p }}
      targetPort: {{ $p }}
      protocol: TCP
  {{- end }}
  selector:
    app: {{ template "name" . }}
    release: {{ .Release.Name }}
```

Figure 5.15: Helm chart template for Service Kubernetes resource with Nodeport default

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ template "fullname" . }}-configmap
  labels:
    app: {{ template "name" . }}
    chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
data:
  transport.properties: |-
    org.ccsds.moims.mo.mal.transport.default.protocol = maltcp://

    #-------------------------------------------------------------------------------------------------------

    # TCP/IP protocol properties
    org.ccsds.moims.mo.mal.transport.protocol.maltcp=esa.mo.mal.transport.tcpip.TCPIPTransportFactoryImpl
    org.ccsds.moims.mo.mal.encoding.protocol.maltcp=esa.mo.mal.encoder.binary.fixed.FixedBinaryStreamFactory
    org.ccsds.moims.mo.mal.transport.tcpip.autohost={{ .Values.configs.useAutohost }}
    org.ccsds.moims.mo.mal.transport.tcpip.host={{ .Values.configs.transportHost | default ""}}
    org.ccsds.moims.mo.mal.transport.tcpip.publishedhost={{ .Values.configs.publishedHost | default ""}}
    org.ccsds.moims.mo.mal.transport.tcpip.port={{ .Values.configs.port | default 1024}}
    #-------------------------------------------------------------------------------------------------------
    org.ccsds.moims.mo.mal.transport.gen.debug=true
    org.ccsds.moims.mo.mal.transport.gen.wrap=false

    org.ccsds.moims.mo.mal.transport.gen.inputprocessors=20

    # MAL Interactions will timeout after that time without updates and throw a MAL DELIVERY_TIMEOUT error
    org.ccsds.moims.mo.mal.interaction.timeout=10000

  platformsim.properties: |-
    #-------------------------------------------------------------------------------------------------------
    # Platform properties
    #-------------------------------------------------------------------------------------------------------
    # Set the platform mode to hybrid, sim or real
    platform.mode={{ .Values.configs.platformProperties.mode | default "hybrid" }}
    {{ range $key, $value := .Values.configs.platformProperties.adapters }}
    {{ $key }}={{ $value }}
    {{ end }}
    #-------------------------------------------------------------------------------------------------------
```

Figure 5.16: Part of helm chart template for Supervisor Configmap Kubernetes resource

```yaml
image:
  repository: ghcr.io/kosmoedge/publish-clock
  tag: latest
  pullPolicy: Always
service:
  port: 32326
  type: ClusterIP
publishClock:
  image:
    repository: ghcr.io/kosmoedge/publish-clock
    tag: latest
    pullPolicy: Always
scale:
  minReplicas: 1
  maxReplicas: 4
  targetCPUUtilizationPercentage: 70
replicaCount: 1
app:
  ports:
    - 32326
    - 11111
  healthcheck:
    port: 32326
    liveness: /
    readiness: /
  envVars:
  - name: SUPERVISOR
    value: maltcp://supervisor.kosmoedge:32325/nanosat-mo-supervisor-Directory
  - name: APP
    value: PushClock
resources:
  limits:
   cpu: 800m
   memory: 256Mi
  requests:
   cpu: 200m
   memory: 128Mi
configs:
  useAutohost: false
  transportHost: "0.0.0.0"
  publishedHost: clock.kosmoedge
  moduleHome: "/opt/space-module"
```

Figure 5.17: Values.yaml that sets the values for the placeholders in helm templates

5.4 Development productivity: Improve local development experience

A Docker Compose file was created to automate the deployment of multiple containers, such as the Supervisor, a Space Camera app and a Space Clock app, thereby significantly improving the local development experience. By launching all services with a single command, the Docker Compose setup ensures a consistent and reproducible environment across different machines and allows developers to test the integration of multiple services.

```
version: "3.9"

services:
  publish-clock:
    container_name: publish-clock
    restart: on-failure
    image: kosmoedge/publish-clock:latest
    environment:
      - SUPERVISOR=maltcp://supervisor.package_default:1024/nanosat-mo-supervisor-
Directory
      - APP=PushClock
    ports:
      - 127.0.0.1:1025:1024
    depends_on:
      - supervisor

  camera:
    container_name: camera
    restart: on-failure
    image: ghcr.io/kosmoedge/camera:latest
    environment:
      - SUPERVISOR=maltcp://supervisor.package_default:1024/nanosat-mo-supervisor-
Directory
      - APP=SnapNMF
    ports:
      - 127.0.0.1:1026:1024
    depends_on:
      - supervisor

  supervisor:
    container_name: supervisor
    restart: on-failure
    image: ghcr.io/kosmoedge/supervisor:latest
    ports:
      - 127.0.0.1:1024:1024
```

Figure 5.18: Docker compose for local multi-container deployment

Complementing this, a Makefile is provided to further simplify local development by offering streamlined commands to build and push container images to a registry for the Supervisor, Space, and Ground applications. It also includes Maven helper targets for generating "fat-jars" of the compiled services, compiling the consumer test tool, and running the containers via Docker Compose. Together, these files reduce manual setup steps, minimize errors, improve workflow consistency, and enhance productivity, making it easier for developers to focus on building and testing mission-critical components in a controlled, portable environment.

```
VERSION := $(shell deployment/scripts/version.sh)
DOCKER_COMPOSE_FILE := deployment/docker-compose.yaml


#---- Build & push service images ----#
# Build & push Supervisor image
supervisor:
    docker buildx build --no-cache --platform linux/arm64/v8 --build-arg VERSION=$(VERSION) -f
deployment/Dockerfile.Supervisor -t ghcr.io/kosmoedge/supervisor:latest . --load

# Build & push space module image
space-module-%:
    docker buildx build --no-cache --platform linux/arm64/v8 --build-arg
MODULE_PATH=sdk/examples/space/$* --build-arg MODULE_NAME=$* --build-arg VERSION=$(VERSION) -f
deployment/Dockerfile.SpaceModule -t ghcr.io/kosmoedge/$*:latest . --load

# Build & push ground module image
ground-module-%:
    docker buildx build --no-cache --build-arg MODULE_PATH=sdk/examples/ground/$* --build-arg
MODULE_NAME=$* --build-arg VERSION=$(VERSION) -f deployment/Dockerfile.GroundModule -t
ghcr.io/kosmoedge/$*:latest .


#---- Helper operations ----#
assembly-jar:
    mvn clean install -P assembly-with-dependencies

consumer-tool:
    mvn clean install && sdk/sdk-package/target/nmf-sdk-$(VERSION)/home/nmf/consumer-test-
tool/consumer-test-tool.sh


#---- Run containers using docker compose --
# Setup network
create_docker_net:
    @ docker network inspect mo-bridge > /dev/null 2> /dev/null && : || docker network create mo-
bridge

# Start all or specific containers
start: create_docker_net
    @ docker-compose -f $(DOCKER_COMPOSE_FILE) up -d $(CONTAINER_NAMES)

# Stop all or specific containers
stop:
    @ docker-compose -f $(DOCKER_COMPOSE_FILE) stop $(CONTAINER_NAMES)

# Stop all containers and remove built images
down:
    @ docker-compose -f $(DOCKER_COMPOSE_FILE) down --rmi local

# Restart all or specific containers
restart: stop start
```

Figure 5.19: Utilizing Makefile to streamline local development lifecycle

```
#!/bin/bash
VERSION=$(mvn exec:exec -Dexec.executable='echo' -Dexec.args='${project.version}' --non-recursive -q)
echo $VERSION
```

Figure 5.20: Script to provide dynamic versioning in container image tags

5.5 End to end image capture, edge processing and ground storage: New NMF Space app a NMF Ground app creation

In order to showcase an end-to-end system for our new processes and infrastructure two NMF applications have been developed. The proof of concept ecosystem consists of a k3s cluster deployed on a Raspberry Pi, with the NMF Supervisor and NMF Camera Acquisition Space App deployed via Jenkins and Helm charts. The NMF Photo Storage Ground App operates on a development workstation.

1. NMF Space App captures snapshots on request using the Supervisor's Camera Platform Service, processes the images on-board using Sobel filtering for edge detection, and transmits only the processed results to the ground application. Performing image processing in space significantly reduces data volume, minimizing downlink bandwidth requirements, as edge-filtered data is far lighter than raw imagery. Sobel filtering is selected for its efficiency in semantic edge detection in satellite imagery, keeping essential structural information rather than full images, achieving competitive results to deep learning methods with minimal computational cost [96].

2. NMF Ground App, built with the Spring Boot framework [97], receives the processed images and stores them locally through a configurable listener for incoming image data. It also provides a REST endpoint for communication with the space component.

```
byte[] finalImage = OpenCVSobelOperator.apply(picture.getContent().getValue());
casMCAdapter.pushBlob(finalImage);

...


public void pushBlob(byte[] image) throws NMFException {
    lastSatImage = new Blob(image);
    LOGGER.log(Level.INFO, "Pushing satellite image to consumers.");
    connector.pushParameterValue("SatelliteImage", lastSatImage);
}
```

Figure 5.21: Part of space camera app code to process image and push it downstream

```java
private static class ImageDataReceivedAdapter extends CompleteDataReceivedListener {

    @Override
    public void onDataReceived(ParameterInstance parameterInstance) {
        String parameterName = parameterInstance.getName().getValue();
        Attribute parameterAttribute = parameterInstance.getParameterValue().getRawValue();
        LOGGER.log(Level.INFO, "Parameter name: {0}, Data content: {1}", new Object[]{
                parameterName, parameterAttribute});
        if (Parameter.LAST_SAT_IMAGE.equals(parameterName)) {
            LOGGER.log(Level.INFO, "Satellite image received.");
            try {
                // Transform to byte array
                Blob image = (Blob) parameterAttribute;
                byte[] imageByteArray = image.getValue();
                // Store it in a file!
                FileOutputStream fos = new FileOutputStream("sat-image.png");
                fos.write(imageByteArray);
                fos.flush();
                fos.close();
                LOGGER.log(Level.INFO, "Satellite image stored.");
            } catch (IOException e) {
                throw new RuntimeException(e);
            } catch (MALException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Figure 5.22: ImageDataReceiverAdapter in ground app listening for incoming images

```java
if (provider.getProviderId().getValue().equals(PROVIDER_CAMERA_APP)) {
    gma = new GroundMOAdapterImpl(provider);
    gma.addDataReceivedListener(new ImageDataReceivedAdapter());
    break;
} else {
    LOGGER.log(Level.WARNING, "Camera Acquisition App not found! Retrying...");
}
```

Figure 5.23: ImageDataReceivedAdapter registration in NMF GroundMOAdapter

A demo Java & Quarkus [98] based application is developed prior to adopting the final solution [99]. The application exposes a REST endpoint that receives commands to capture camera snapshots and process the images. It serves as a testbed for evaluating different image preprocessing techniques following a space edge computing approach.

Three distinct operators are implemented, each applying a different image processing

algorithm:

- OpenCV Sobel Operator: applies Sobel filtering using the OpenCV library [100] to detect intensity gradients and emphasize edges. The operator leverages OpenCV's optimized convolution routines.
- K-means Operator: based on OpenCV library, performs unsupervised clustering on image pixels based on color or intensity. By partitioning the image into k clusters, this operator reduces image complexity and highlights regions of interest, supporting tasks such as segmentation or object boundary detection [101].
- Native Sobel Operator: implements Sobel filtering directly in Java without third-party dependencies [102].

```java
package com.kosmoedge.gr.camera.operator;

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.util.logging.Logger;
import org.opencv.core.Mat;
import org.opencv.imgproc.Imgproc;

import com.kosmoedge.gr.camera.util.OpenCvFileTransformer;

public class OpenCVSobelOperator {

    private OpenCVSobelOperator() {
    }

    private static final Logger logger =
Logger.getLogger(OpenCVSobelOperator.class.getName());

    public static BufferedImage apply(BufferedImage image) throws IOException {
        logger.info("Sobel operation (via OpenCV) is currently being applied...");

        Mat src = OpenCvFileTransformer.bufferedImage2Mat(image);
        Mat dst = new Mat();

        Imgproc.Sobel(src, dst, -1, 0, 1);
        Imgproc.Sobel(src, dst, -1, 1, 0);
        Imgproc.Sobel(src, dst, -1, 1, 1);

        return OpenCvFileTransformer.mat2BufferedImage(dst);
    }

}
```

Figure 5.24: OpenCV Sobel filtering operator

```java
public class KmeansClusteringOperator {

    private KmeansClusteringOperator() {
    }

    private static final Logger logger = Logger.getLogger(KmeansClusteringOperator.class.getName());

    public static BufferedImage apply(BufferedImage inputImage) throws IOException {
        logger.info("Kmeans Clustering operation is currently being applied...");

        Mat image = OpenCvFileTransformer.bufferedImage2Mat(inputImage);
        // Convert image to float type
        image.convertTo(image, CvType.CV_32F);

        // Reshape image to a 2D array of pixels
        Mat samples = image.reshape(1, image.rows() * image.cols());

        int clusters = 10;
        Mat bestLabels = new Mat();
        TermCriteria criteria = new TermCriteria(TermCriteria.EPS + TermCriteria.MAX_ITER, 100, 0.1);
        int attempts = 5;
        int flags = Core.KMEANS_PP_CENTERS;
        Mat centers = new Mat();

        // Perform K-means clustering
        Core.kmeans(samples, clusters, bestLabels, criteria, attempts, flags, centers);

        // Reshape the labels to the original image size
        Mat segmented = bestLabels.reshape(0, image.rows());

        // If we have a big number of cluster better use a random color to each one
        Scalar[] randomColors = null;
        List<Scalar> colorPalette = new ArrayList<>();
        if (clusters > 10) {
            Random rand = new Random();
            randomColors = new Scalar[clusters];
            for (int i = 0; i < clusters; i++) {
                randomColors[i] = new Scalar(rand.nextInt(256), rand.nextInt(256), rand.nextInt(256));
            }
        } else {
            // Custom color palette until 10 clusters
            colorPalette.add(new Scalar(255, 0, 0));      // Red
            colorPalette.add(new Scalar(0, 255, 0));      // Green
            colorPalette.add(new Scalar(0, 0, 255));      // Blue
            colorPalette.add(new Scalar(255, 255, 0));    // Yellow
            colorPalette.add(new Scalar(255, 0, 255));    // Magenta
            colorPalette.add(new Scalar(0, 255, 255));    // Cyan
            colorPalette.add(new Scalar(128, 0, 0));      // Maroon
            colorPalette.add(new Scalar(0, 128, 0));      // Green (dark)
            colorPalette.add(new Scalar(0, 0, 128));      // Navy
            colorPalette.add(new Scalar(128, 128, 0));    // Olive
        }

        // Apply the color to each pixel based on its cluster label
        Mat coloredImage = new Mat(segmented.size(), CvType.CV_8UC3);
        for (int y = 0; y < segmented.rows(); y++) {
            for (int x = 0; x < segmented.cols(); x++) {
                int clusterIndex = (int) segmented.get(y, x)[0];
                if (randomColors != null) {
                    double[] color = randomColors[clusterIndex].val;
                    double[] pixelValues = new double[]{color[2], color[1], color[0]}; //BGR format
                    coloredImage.put(y, x, pixelValues);
                } else {
                    Scalar color = colorPalette.get(clusterIndex);
                    double[] pixelValues = new double[]{color.val[2], color.val[1], color.val[0]}; // BGR format
                    coloredImage.put(y, x, pixelValues);
                }
            }
        }

        return OpenCvFileTransformer.mat2BufferedImage(coloredImage);
    }
}
```

Figure 5.25: OpenCV K-means clustering operator

6. EVALUATION

The expected value of this thesis is multi-dimentional. In general, the main value is to open up a new way of thinking about architecture in space software and eventually lead to more scalable infrastructure. From this point of view we evaluate the work of this thesis on the following factors:

- the resource utilization and the possible overhead that the proposed setup may or may not have, compared to the conventional architecture
- the new features in terms of productivity and resiliency that we get
- the software architecture break-throughs that the present research provides out of the box to be utilized in future designs

6.1 Resource utilization and performance comparison to the conventional architecture

It is important to understand the costs or gains of the proposed architecture in terms of resources and performance. This way one can better evaluate if the tradeoff is worth it.

Evaluation setup The evaluation process is conducted primarily on a Raspberry Pi, which hosts both a new NMF architecture running on k3s and a conventional NMF installation running in the JVM. Each installation is alternately deactivated while metrics are collected from the other to ensure consistent measurement conditions.

Collecting metrics The system is already loaded (only 2G of total memory), so the use of a full-blown metrics collector is avoided. Instead, a lightweight monitoring tool is preferred to minimize overhead. A possible reduction in granularity is considered acceptable, as the objective is not an in-depth performance analysis but rather a high-level overview of resource utilization during defined test periods and workloads. Thus, the Linux tool vmstat [103] is selected for this goal.

Pros of vmstat:

- Lightweight
- Provides overview of many system aspects
- Easy to run and configure
- Real time data

Cons of vmstat:

- Limited Granularity, not many details about root cause of utilization
- System Specific, no app usage
- Raw output

The command used is *vmstat -a -SM -w -t 1 30 | tee vmstat.log*. The command options used are:

- **-a**: Activates the "active/inactive" memory reporting. Instead of just showing free and buffer/cache memory, it provides more granular details:
  - **inactive**: The amount of inactive memory, which is memory that has been accessed but hasn't been used recently and can be reclaimed by the kernel without a major performance penalty.
  - **active**: The amount of active memory, which is currently in use and is less likely to be swapped out.
- **-S M**: Specifies the unit for memory reporting. The -S flag changes the scaling, and M sets it to megabytes. By default, vmstat uses kilobytes.
- **-w**: Activates wide output, ensuring that columns are not truncated, which is helpful when dealing with larger numbers or multiple data points.
- **-t**: Prepends a timestamp to each line of the output, making it useful for analyzing trends over time and correlating performance spikes with specific events.

The metrics are collected every 1 second over a period of 30 seconds. Using the pipe (|) results are exported to the vmstat.log file. Using the vmstat-visualizer tool [104] the first visualization trials for vmstat results take place. However, this tool does not exactly meet our needs, as it works with specific vmstat flags and does not have a comparison feature. So a custom made vmstat-visualizer cli tool [105] is created and utilized specifically for this thesis.

Test scenarios In order to test the overhead of the proposed implementation, basic tests will take place between running NMF on k3s and on conventional JVM. For each setup different cases are examined, such as running an NMF app on k3s without the presence of the supervisor module. As a starting point, monitoring Raspberry Pi without any NMF payload can provide a view of our initial resource utilization.

Test cases monitored:

1. No payload

2. NMF on k3s
   (a) Only k3s
   (b) Camera and Supervisor apps deployed, camera access and 3 consecutive jpg snapshots
   (c) Camera app stopped, benchmarking for Supervisor
   (d) NMF clock application deployed without Supervisor
3. Conventional NMF payload on JVM
   (a) Only Supervisor deployed, no action
   (b) Camera and Supervisor apps deployed, camera access and 3 consecutive jpg snapshots

6.1.1 No payload

In this scenario Raspberry Pi is benchmarked alone without running any workloads or other actions.

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   1    0            0          965           76          689    0    0   402    31  293  477   5   3  86   6   0   0 2025-07-31 23:52:52
   1    0            0          965           76          689    0    0     0     0  291  412   0   0 100   0   0   0 2025-07-31 23:52:53
   1    0            0          965           76          689    0    0     0    32  348  503   1   0  99   0   0   0 2025-07-31 23:52:54
   1    0            0          965           76          689    0    0     0     0  239  368   0   0 100   0   0   0 2025-07-31 23:52:55
   1    0            0          965           76          689    0    0     0     0  216  321   0   0 100   0   0   0 2025-07-31 23:52:56
   1    0            0          965           76          689    0    0     0     0  308  431   0   1  99   0   0   0 2025-07-31 23:52:57
   1    0            0          965           76          689    0    0     0     4  273  427   0   0 100   0   0   0 2025-07-31 23:52:58
   1    0            0          965           76          689    0    0     0     0  318  475   0   0  99   0   0   0 2025-07-31 23:52:59
   1    0            0          965           76          689    0    0     0    20  269  378   0   0 100   0   0   0 2025-07-31 23:53:00
   1    0            0          965           76          689    0    0     0    24  268  394   0   0 100   0   0   0 2025-07-31 23:53:01
   1    0            0          965           76          689    0    0     0     0  325  475   0   0  99   0   0   0 2025-07-31 23:53:02
   1    0            0          965           76          689    0    0     0     0  275  411   0   0 100   0   0   0 2025-07-31 23:53:03
   1    0            0          965           76          689    0    0     0     4  321  486   0   1  99   0   0   0 2025-07-31 23:53:04
   1    0            0          965           76          689    0    0     0     0  248  378   1   0 100   0   0   0 2025-07-31 23:53:05
   1    0            0          965           76          689    0    0     0    12  317  444   0   1  99   0   0   0 2025-07-31 23:53:06
   1    0            0          965           76          689    0    0     0     0  312  452   1   0  99   0   0   0 2025-07-31 23:53:07
   1    0            0          965           76          689    0    0     0     0  273  388   0   1  99   0   0   0 2025-07-31 23:53:08
   1    0            0          965           76          689    0    0     0     0  291  416   0   1  99   0   0   0 2025-07-31 23:53:09
   1    0            0          965           76          689    0    0     0     4  445  727   1   0  99   0   0   0 2025-07-31 23:53:10
   1    0            0          965           76          689    0    0     0     0  220  338   0   0 100   0   0   0 2025-07-31 23:53:11
   1    0            0          965           76          689    0    0     0    12  357  471   0   1  99   0   0   0 2025-07-31 23:53:12
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   3    0            0          965           76          689    0    0     0     0  268  368   0   0  99   0   0   0 2025-07-31 23:53:13
   1    0            0          965           76          689    0    0     0     0  245  370   0   0 100   0   0   0 2025-07-31 23:53:14
   1    0            0          965           76          689    0    0     0     0  249  362   0   0 100   0   0   0 2025-07-31 23:53:15
   1    0            0          965           76          689    0    0     0     4  215  330   0   0  99   0   0   0 2025-07-31 23:53:16
   1    0            0          965           76          689    0    0     0     0  293  412   0   0  99   0   0   0 2025-07-31 23:53:17
   1    0            0          965           76          689    0    0     0    12  289  410   0   1  99   0   0   0 2025-07-31 23:53:18
   1    0            0          965           76          689    0    0     0     0  305  460   0   1  99   0   0   0 2025-07-31 23:53:19
   1    0            0          965           76          689    0    0     0    16  226  328   0   0  99   0   0   0 2025-07-31 23:53:20
   1    0            0          965           76          689    0    0     0     0  232  336   1   0  99   0   0   0 2025-07-31 23:53:21
```

Figure 6.1: Raw metrics for no payload

Overall, the CPU is consistently idle, with **99-100% idle time (id)** for most of the period.

The user (us) and system (sy) CPU usage is minimal, indicating no significant application or kernel processes are demanding CPU time.

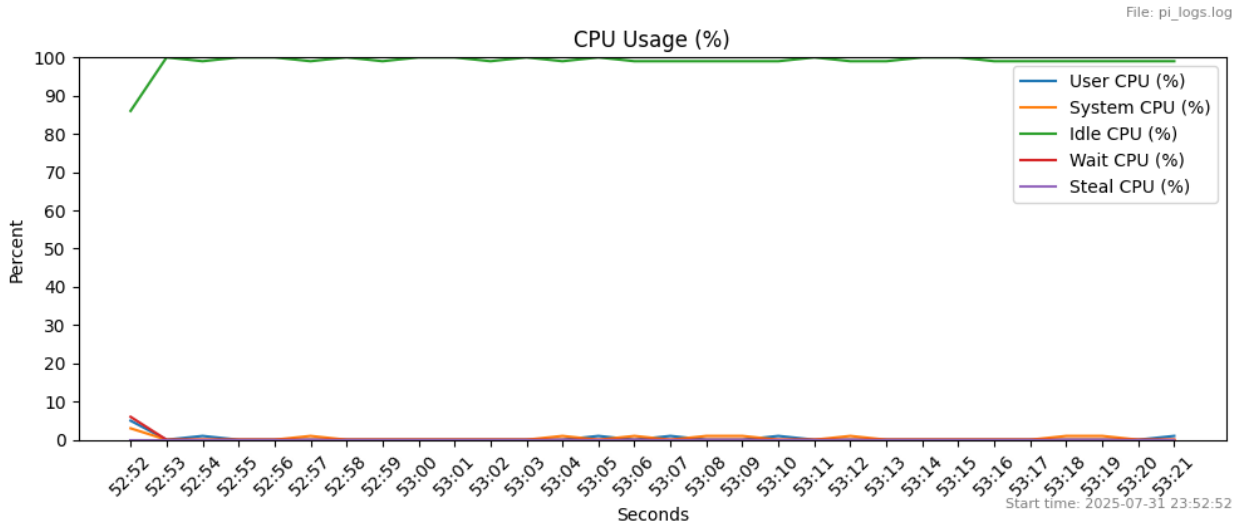

Figure 6.2: CPU usage for no payload (idle CPU metrics removed for visibility)

The number of runnable processes (r) is consistently low, with at most 2 processes waiting for CPU time. This confirms that the system is **not under any significant load**.

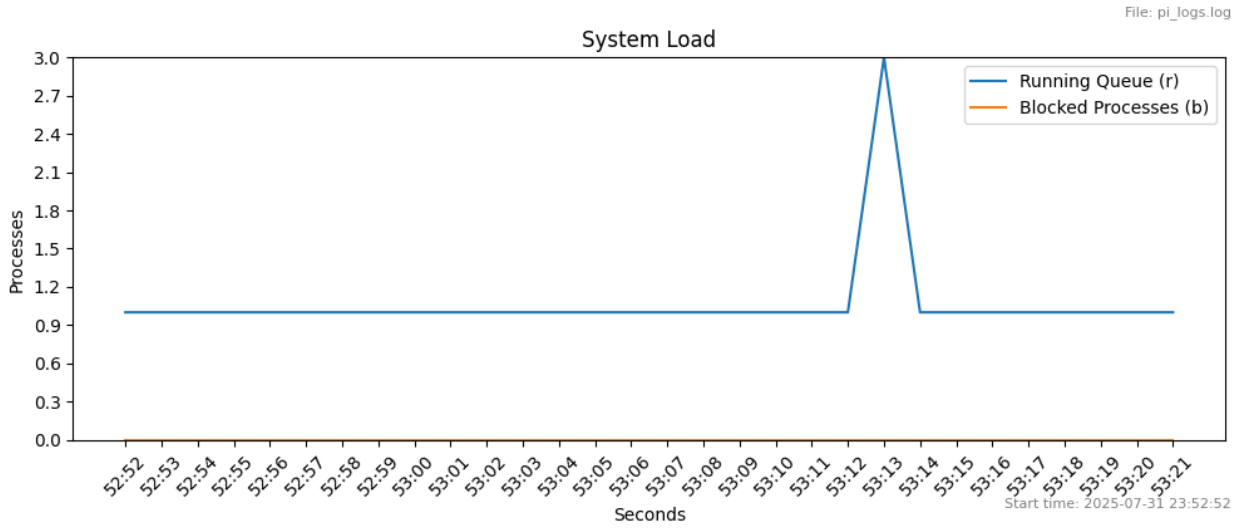

Figure 6.3: System load for no payload

Since most of the memory is in cache, cache was removed from the graph for visibility purposes.

Memory usage is **stable** and well-managed. There is a large amount of **free memory (967 MB)**.

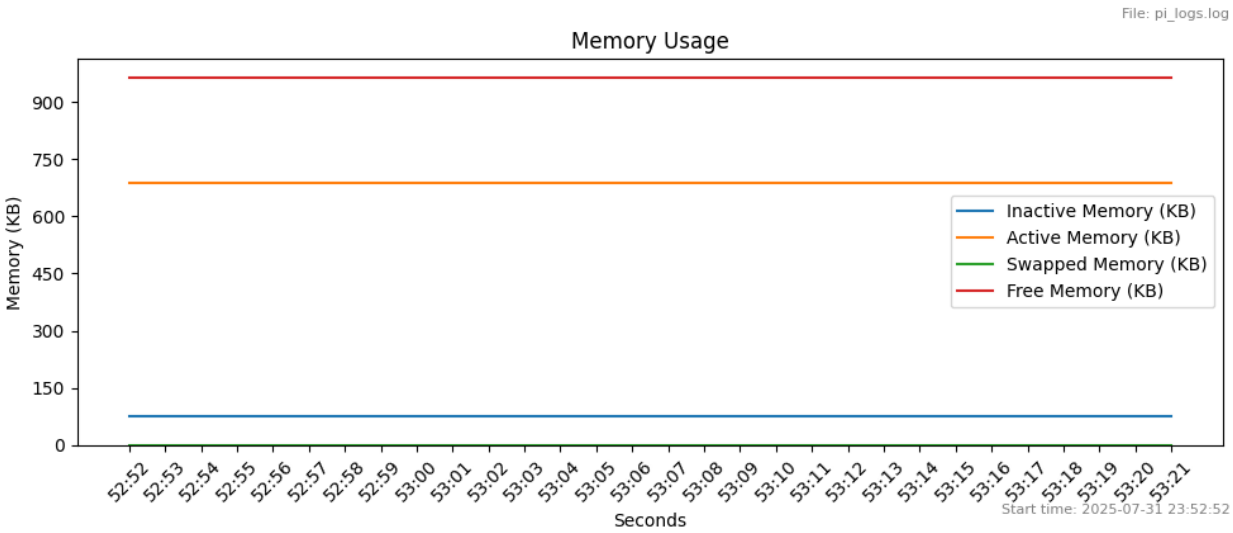

Figure 6.4: Memory usage for no payload

No data is being swapped to or from disk (**swpd, si, and so are all zero**)

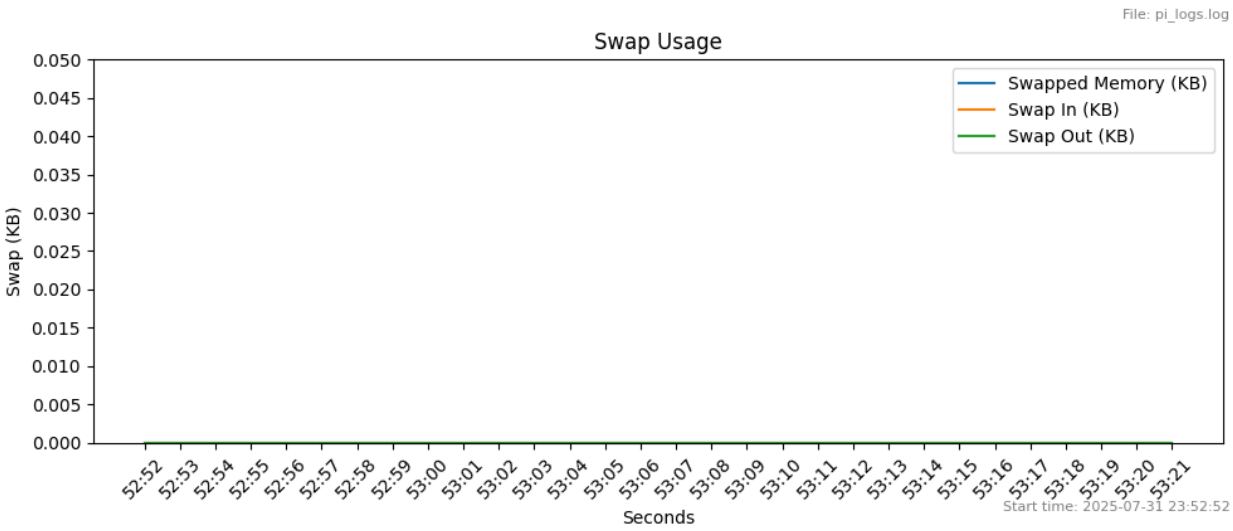

Figure 6.5: Swap usage for no payload

Disk activity is **sporadic and minimal**. While there are a couple of small, brief spikes in block reads (bi) and writes (bo), they are short-lived and do not cause a sustained I/O wait (wa) on the CPU.

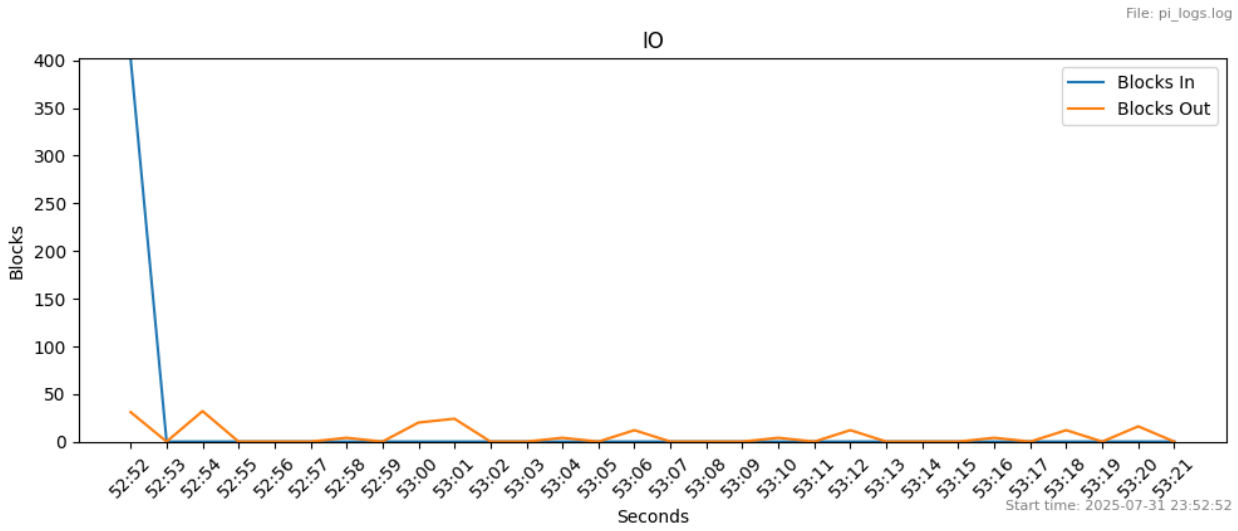

Figure 6.6: IO for no payload

In this scenario, the system does not use many resources. Although almost half of the available RAM (1 out of 2GB) is consumed, probably from the Pi OS.

6.1.2 NMF on k3s

In this bundle of tests, the utilization overhead imposed by running k3s and NMF apps on the system is tested.

6.1.2.1 Only k3s

In this scenario, we seek to benchmark the Raspberry Pi with k3s running standalone.

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   3    0            0          523          128         1059    0    0     4     1   71  111   0   0  99   0   0   0 2025-08-01 10:06:32
   1    0            0          523          128         1059    0    0     0     0 1815 3452   4   1  96   0   0   0 2025-08-01 10:06:33
   1    0            0          523          128         1059    0    0     0    16 2530 4711   4   3  93   0   0   0 2025-08-01 10:06:34
   1    0            0          523          128         1059    0    0     0     0 2061 3776   4   2  94   0   0   0 2025-08-01 10:06:35
   1    0            0          523          128         1059    0    0     0     0 2181 4079   4   3  94   0   0   0 2025-08-01 10:06:36
   2    0            0          523          128         1059    0    0     0    16 1906 3617   4   1  95   0   0   0 2025-08-01 10:06:37
   1    0            0          523          128         1059    0    0     0     0 2108 3965   2   2  95   0   0   0 2025-08-01 10:06:38
   1    0            0          523          128         1059    0    0     0     0 2670 4944   5   3  92   0   0   0 2025-08-01 10:06:39
   1    0            0          523          128         1059    0    0     0   176 2093 3917   4   3  94   0   0   0 2025-08-01 10:06:40
   1    0            0          523          128         1059    0    0     0     0 2359 4400   7   2  92   0   0   0 2025-08-01 10:06:41
   1    0            0          523          128         1059    0    0     0     0 2025 3795   2   1  98   0   0   0 2025-08-01 10:06:42
   1    0            0          523          128         1059    0    0     0     0 1820 3520   2   0  98   0   0   0 2025-08-01 10:06:43
   1    0            0          523          128         1059    0    0     0     0 2064 4008   2   1  97   0   0   0 2025-08-01 10:06:44
   1    0            0          523          128         1059    0    0     0   204 2170 4038   3   1  95   1   0   0 2025-08-01 10:06:45
   2    0            0          523          128         1059    0    0     0   164 2132 3834   3   2  96   0   0   0 2025-08-01 10:06:46
   1    0            0          523          128         1059    0    0     0     0 1963 3705   5   2  94   0   0   0 2025-08-01 10:06:47
   1    0            0          523          128         1059    0    0     0     0 2043 3800   3   2  96   0   0   0 2025-08-01 10:06:48
   1    0            0          523          128         1059    0    0     0     0 3404 6132   6   5  89   0   0   0 2025-08-01 10:06:49
   1    0            0          523          128         1059    0    0     0    32 1949 3607   2   2  96   0   0   0 2025-08-01 10:06:50
   1    0            0          523          128         1059    0    0     0     0 1876 3539   3   2  95   0   0   0 2025-08-01 10:06:51
   1    0            0          523          128         1059    0    0     0     0 2031 3804   3   2  95   0   0   0 2025-08-01 10:06:52
   2    0            0          523          128         1059    0    0     0     0 2140 4011   4   2  94   0   0   0 2025-08-01 10:06:53
   2    0            0          523          128         1059    0    0     0     0 2180 3907   3   2  95   0   0   0 2025-08-01 10:06:54
   2    0            0          523          128         1059    0    0     0   100 2222 4095   4   2  93   0   0   0 2025-08-01 10:06:55
   1    0            0          523          128         1059    0    0     0     0 2389 4300   7   3  90   0   0   0 2025-08-01 10:06:56
   2    0            0          523          128         1059    0    0     0     0 2217 4066   5   3  93   0   0   0 2025-08-01 10:06:57
   1    0            0          523          128         1059    0    0     0    20 2078 3621  15   2  84   0   0   0 2025-08-01 10:06:58
   3    0            0          523          128         1059    0    0     0     0 2359 4536   2   1  97   0   0   0 2025-08-01 10:06:59
   1    0            0          523          128         1059    0    0     0     0 2102 4034   1   2  97   0   0   0 2025-08-01 10:07:00
   1    0            0          523          128         1059    0    0     0   248 2167 3986   2   1  97   0   0   0 2025-08-01 10:07:01
```

Figure 6.7: Raw metrics for standalone k3s

The system is idle for most of the time. Usually remains above 90%. User (us) and system (sy) CPU usage are low, rarely exceeding 7%. There is almost no waiting (wa) or stealing (st).

We can observe small peaks and it is clear that only on 2 occasions total CPU usage exceeded 10%.

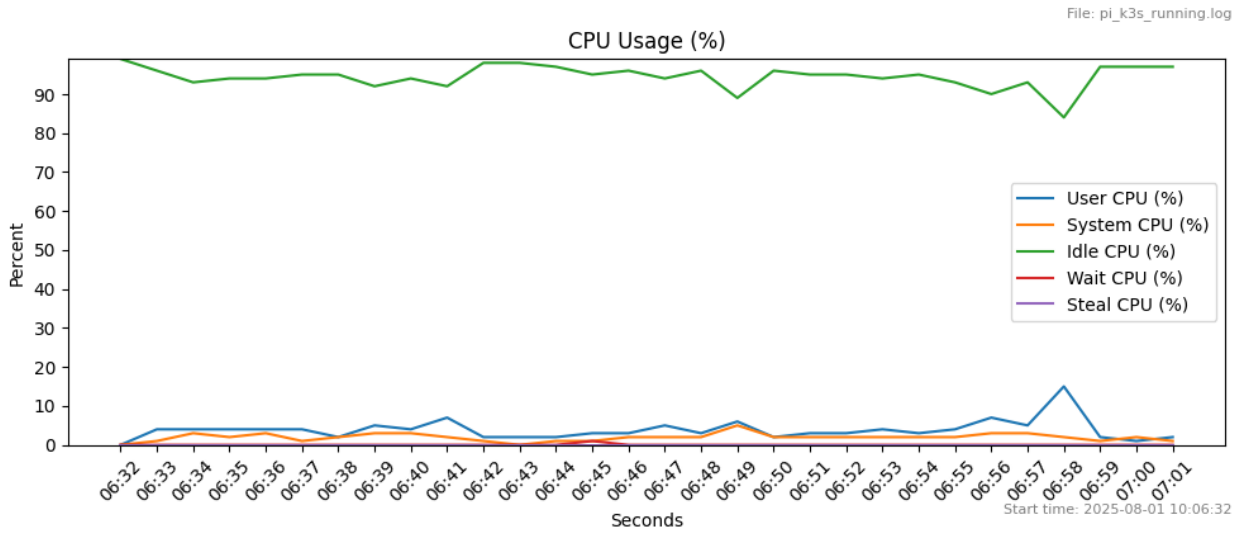

Figure 6.8: CPU usage for standalone k3s

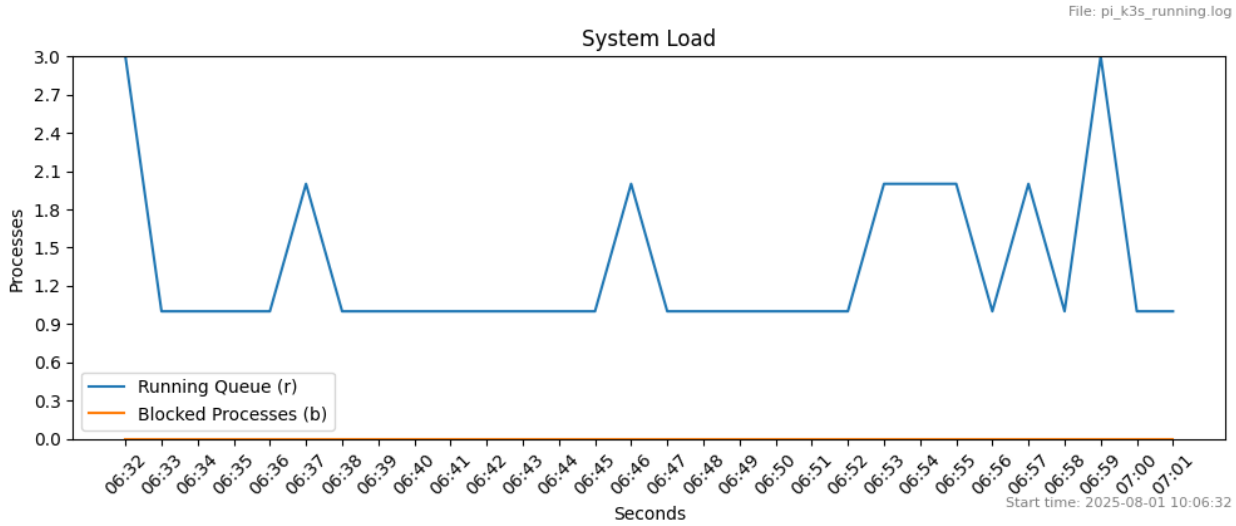

Figure 6.9: System load for standalone k3s

The free memory is around 500Mb, so compared to the scenario without any payload (neither k3s nor NMF), there is a ~400Mb overhead due to k3s installation.

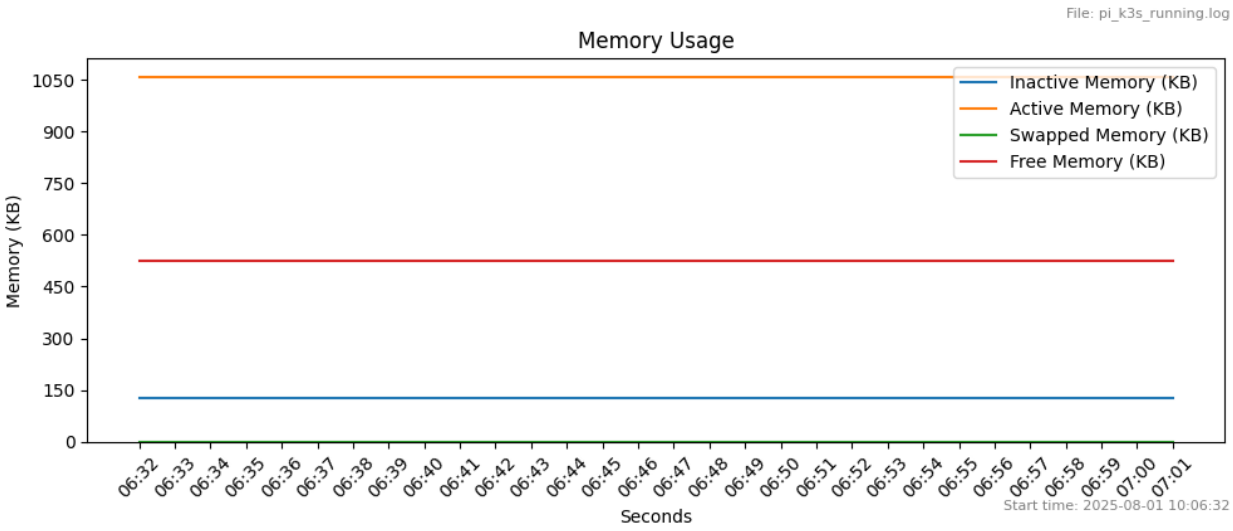

Figure 6.10: Memory usage for standalone k3s

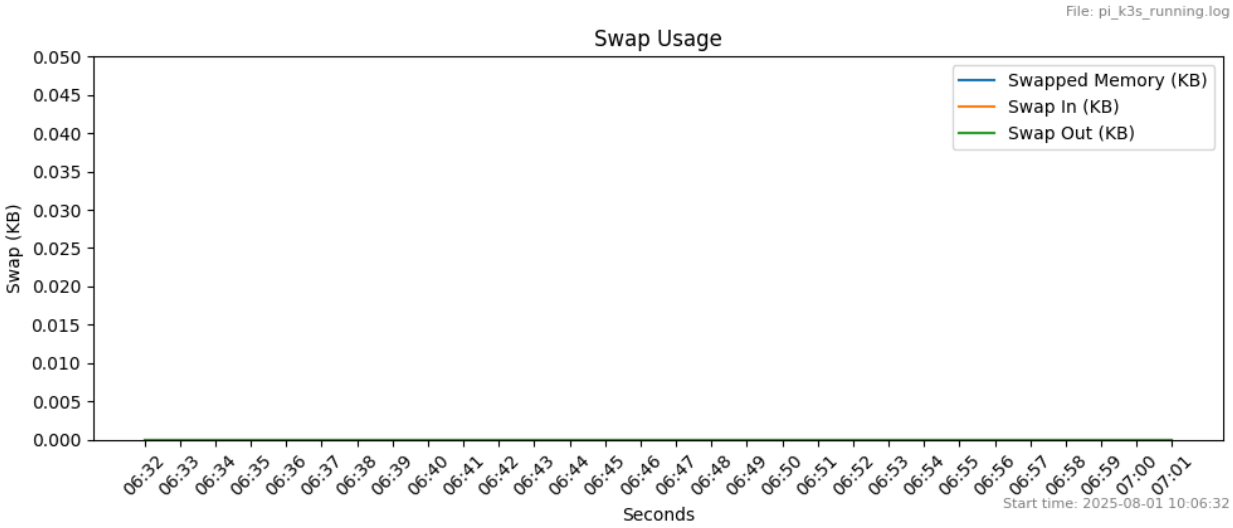

Figure 6.11: Swap usage for standalone k3s

Disk activity remains low. However, there is an observable increase in the output (~300b) compared to the activity in the first example.

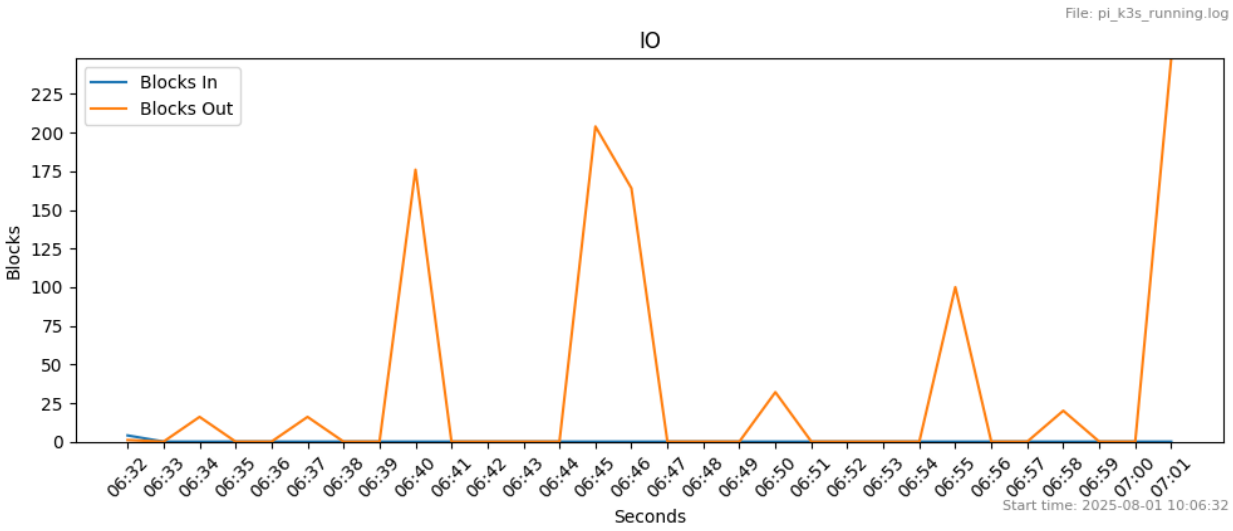

Figure 6.12: IO for standalone k3s

6.1.2.2 Camera and Supervisor apps deployed, camera access and 3 consecutive jpg snapshots

In this scenario, we monitor Raspberry Pi with k3s, an NMF space camera app and Supervisor deployed.

The following actions have an impact on the measurements: 1) Consumer Test Tool connects to Supervisor module 2) The providers are discovered 3) The App's Action provider is selected 4) The Action "TakeSnap.jpg" is submitted 3 consecutive times

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   1    0            0           66         1271          368    0    0    64    15  627 1148   4   2  94   1   0   0 2025-08-24 00:03:37
   1    0            0           66         1271          368    0    0     0     0 2759 4992   2   2  97   0   0   0 2025-08-24 00:03:38
   2    0            0           66         1271          368    0    0     0     0 2373 4443   1   1  97   0   0   0 2025-08-24 00:03:39
   1    0            0           66         1271          368    0    0     0     0 2202 4229   3   1  96   0   0   0 2025-08-24 00:03:40
   1    0            0           66         1271          368    0    0     0   516 2321 4328   4   2  94   0   0   0 2025-08-24 00:03:41
   1    0            0           66         1271          368    0    0     0    96 2618 4966   2   1  97   0   0   0 2025-08-24 00:03:42
   7    0            0           66         1271          368    0    0   116     0 2424 4496   2   2  96   0   0   0 2025-08-24 00:03:43
   1    1            0           66         1271          368    0    0   300     0 2527 4447   3   2  74  21   0   0 2025-08-24 00:03:44
   1    1            0           66         1271          368    0    0   148     0 2462 4375   1   1  73  25   0   0 2025-08-24 00:03:45
   1    1            0           66         1271          368    0    0   136     4 3329 6109   4   2  70  24   0   0 2025-08-24 00:03:46
   1    2            0           66         1271          368    0    0   228    64 3877 6657  10   5  63  22   0   0 2025-08-24 00:03:47
   6    0            0           66         1271          368    0    0  1044   164 3025 5219   4   4  75  17   0   0 2025-08-24 00:03:48
   1    0            0           66         1271          368    0    0     0    60 2485 4388   2   1  96   0   0   0 2025-08-24 00:03:49
   1    0            0           66         1271          368    0    0     0     0 2242 4187   3   1  96   0   0   0 2025-08-24 00:03:50
   1    0            0           66         1271          368    0    0     0     4 2103 4098   3   2  96   0   0   0 2025-08-24 00:03:51
   1    0            0           66         1271          368    0    0     0     0 2501 4576   4   2  95   0   0   0 2025-08-24 00:03:52
   1    0            0           66         1271          368    0    0     0    84 2045 3788   2   1  97   0   0   0 2025-08-24 00:03:53
   2    0            0           66         1271          368    0    0     0     4 2335 4292   4   2  95   0   0   0 2025-08-24 00:03:54
   3    0            0           66         1271          368    0    0     0     0 2120 3946   1   1  97   0   0   0 2025-08-24 00:03:55
   1    0            0           66         1271          368    0    0   204  1464 4321 7376   9   4  85   3   0   0 2025-08-24 00:03:56
   3    0            0           66         1271          368    0    0     4     4 18984 37001  12   7  81   0   0   0 2025-08-24 00:03:57
   2    0            0           66         1271          368    0    0     0   496 2197 4175   4   1  94   1   0   0 2025-08-24 00:03:58
   1    0            0           66         1271          368    0    0     0    20 2077 3987   1   0  99   0   0   0 2025-08-24 00:03:59
   1    0            0           66         1271          368    0    0     0   956 3004 5237   5   2  92   2   0   0 2025-08-24 00:04:00
   4    0            0           66         1271          368    0    0     0   148 15504 29906   7   5  88   0   0   0 2025-08-24 00:04:01
   1    0            0           66         1271          368    0    0     0   476 17621 34150  13   7  79   1   0   0 2025-08-24 00:04:02
   2    0            0           66         1271          368    0    0     0     0 2466 4642   2   1  97   0   0   0 2025-08-24 00:04:03
   2    0            0           66         1271          368    0    0     0  1068 3278 5470  15   2  82   2   0   0 2025-08-24 00:04:04
   1    0            0           66         1271          368    0    0     0     8 33266 65396  11  10  79   0   0   0 2025-08-24 00:04:05
   1    0            0           66         1271          368    0    0     0   476 3684 6563  11   3  86   1   0   0 2025-08-24 00:04:06
```

Figure 6.13: Raw metrics for fully deployed NMF apps on k3s

The system is mostly idle, but some spikes of about 20% usage are visible. Some waiting processes also appear during those spikes. The first spike occurs during the fetch of the provider information from the customer test tool. The rest depict the action of taking the three snapshots. CPU usage increased close to 40% for the initial provider connection and at about 20% for the snapshots.

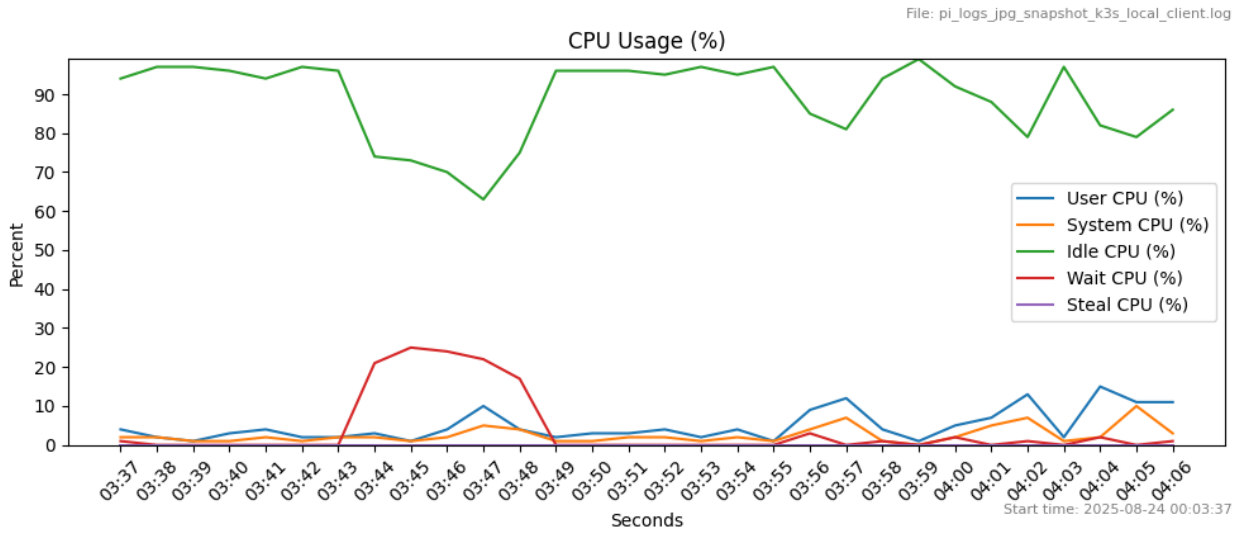

Figure 6.14: CPU usage for fully deployed NMF apps on k3s

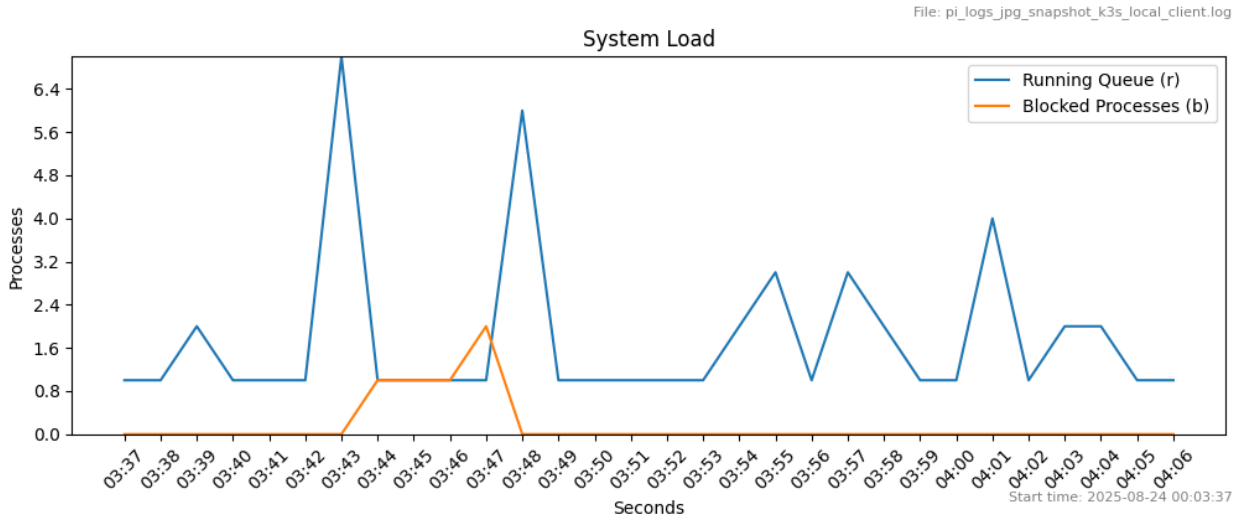

Figure 6.15: System load for fully deployed NMF apps on k3s

Free memory has dropped a lot. However, there is still inactive memory (~1271 MB), while the active memory is stable (~368 MB). This means that the system probably increased cached files, resulting in reduced free memory.

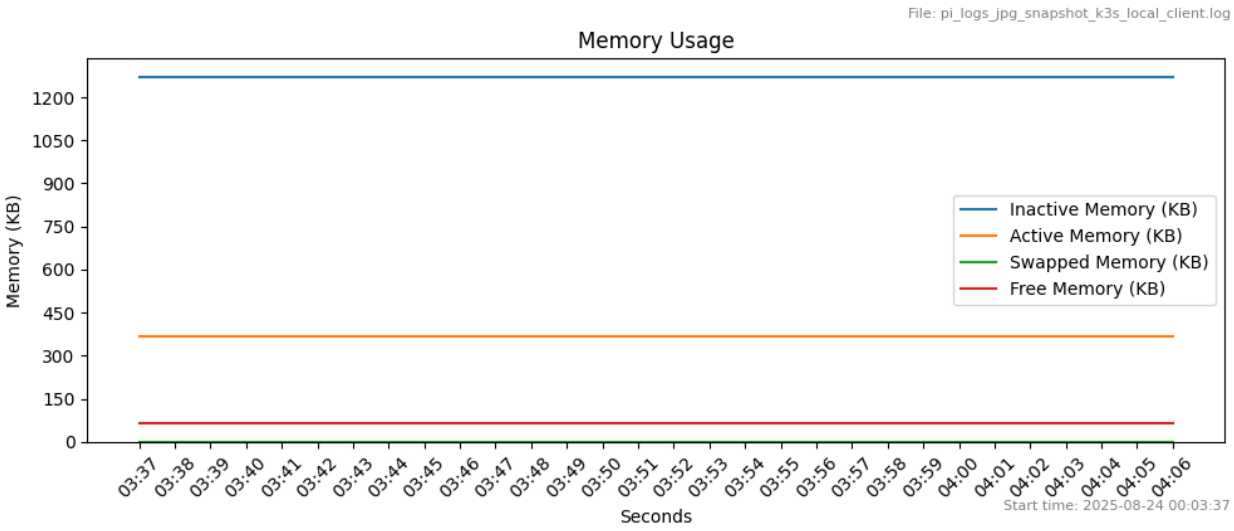

Figure 6.16: Memory usage for fully deployed NMF apps on k3s

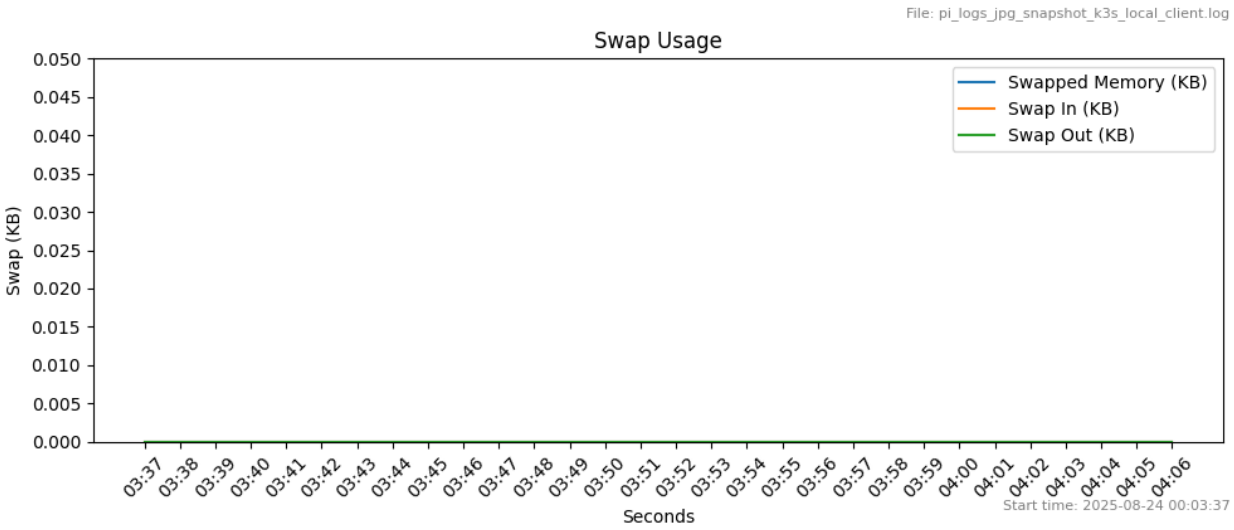

Figure 6.17: Swap usage for fully deployed NMF apps on k3s

Disk utilization looks healthy. There is still no swap, but an increase in input (~200b) and output (~200b) can be observed.

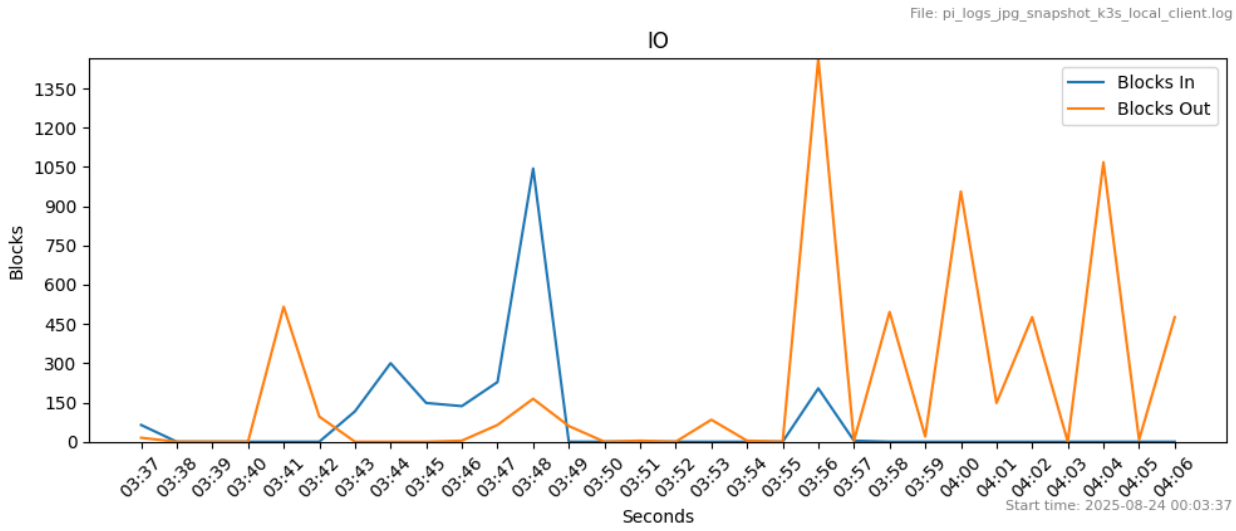

Figure 6.18: IO for fully deployed NMF apps on k3s

6.1.2.3 Camera app stopped, benchmarking for Supervisor

In this scenario, the focus is on the Raspberry Pi with k3s and Supervisor deployed. We want to isolate the contribution of the Supervisor to the workload.

```
--procs-- ----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   3    0            0          226          102         1386    0    0   344    41  675 1187   8   2  85   5   0   0 2025-08-24 23:53:00
   1    0            0          226          102         1386    0    0     0     0 2791 4734   2   3  96   0   0   0 2025-08-24 23:53:01
   1    0            0          226          102         1386    0    0     0     0 2343 4222   5   2  93   0   0   0 2025-08-24 23:53:02
   1    0            0          226          102         1386    0    0     0     0 2054 3734   1   0  99   0   0   0 2025-08-24 23:53:03
   1    0            0          226          102         1386    0    0     0     0 2068 3885   2   1  97   0   0   0 2025-08-24 23:53:04
   1    0            0          226          102         1386    0    0     0     0 1982 3744   1   1  98   0   0   0 2025-08-24 23:53:05
   1    0            0          226          102         1386    0    0     0    36 2450 4539   2   2  97   0   0   0 2025-08-24 23:53:06
   1    0            0          226          102         1386    0    0     0     0 2181 3963   1   2  97   0   0   0 2025-08-24 23:53:07
   2    0            0          226          102         1386    0    0     0     0 2380 4398   3   2  95   0   0   0 2025-08-24 23:53:08
   1    0            0          226          102         1386    0    0     0    16 2551 4297   2   4  94   0   0   0 2025-08-24 23:53:09
   1    0            0          226          102         1386    0    0     0     0 2621 4802   3   3  95   0   0   0 2025-08-24 23:53:10
   1    0            0          226          102         1386    0    0     0     0 2030 3766   3   1  96   0   0   0 2025-08-24 23:53:11
   2    0            0          226          102         1386    0    0     0    36 2204 4021   3   1  96   0   0   0 2025-08-24 23:53:12
   1    0            0          226          102         1386    0    0     0     0 1975 3711   2   1  97   0   0   0 2025-08-24 23:53:13
   1    0            0          226          102         1386    0    0     0     0 2185 3900   4   1  95   0   0   0 2025-08-24 23:53:14
   1    0            0          226          102         1386    0    0     0     0 2213 4099   3   2  95   0   0   0 2025-08-24 23:53:15
   1    0            0          226          102         1386    0    0     0     0 2020 3684   4   1  95   0   0   0 2025-08-24 23:53:16
   1    0            0          226          102         1386    0    0     0     0 2081 3796   2   2  97   0   0   0 2025-08-24 23:53:17
   1    0            0          226          102         1386    0    0     0    28 2397 4400   3   1  96   0   0   0 2025-08-24 23:53:18
   1    0            0          226          102         1386    0    0     0     0 2630 4707   5   2  93   0   0   0 2025-08-24 23:53:19
   1    0            0          226          102         1386    0    0     0    32 2769 4918   8   3  90   0   0   0 2025-08-24 23:53:20
   2    0            0          226          102         1386    0    0     0     0 2350 4344   1   1  98   0   0   0 2025-08-24 23:53:21
   1    0            0          226          102         1386    0    0     0     0 1974 3697   2   0  98   0   0   0 2025-08-24 23:53:22
   1    0            0          226          102         1386    0    0     0    24 1967 3665   1   0  98   0   0   0 2025-08-24 23:53:23
   1    0            0          226          102         1386    0    0     0     4 1963 3696   2   1  98   0   0   0 2025-08-24 23:53:24
   1    0            0          226          102         1386    0    0     0   460 2267 4091   4   1  96   0   0   0 2025-08-24 23:53:25
   1    0            0          226          102         1386    0    0     0     0 2081 3816   2   1  97   0   0   0 2025-08-24 23:53:26
   1    0            0          226          102         1386    0    0     0     0 2171 3891   3   2  96   0   0   0 2025-08-24 23:53:27
   1    0            0          226          102         1386    0    0     0    36 2281 4303   3   1  96   0   0   0 2025-08-24 23:53:28
   1    0            0          226          102         1386    0    0     0     0 1915 3574   2   2  97   0   0   0 2025-08-24 23:53:29
```

Figure 6.19: Raw metrics for NMF Supervisor deployed on k3s

At the system level, there is moderate context switching and interrupts since the system is lightly loaded. The system remains mostly idle. Small increases in usage, less than 10% occur occasionally.

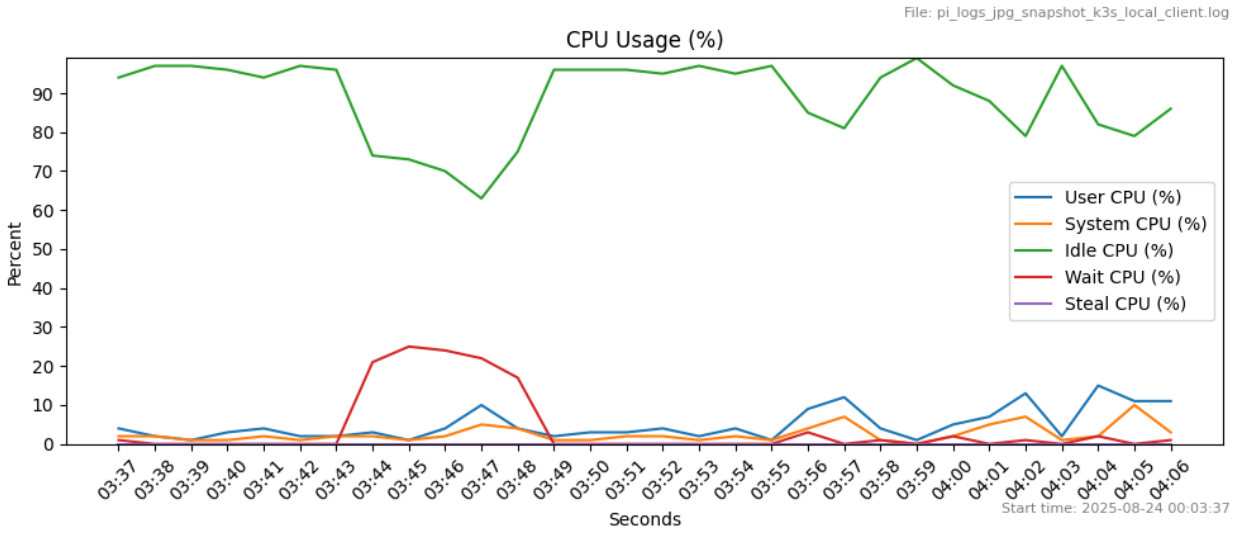

Figure 6.20: CPU usage for NMF Supervisor deployed on k3s

There is a very low system load, with a few running processes and no blocked ones.

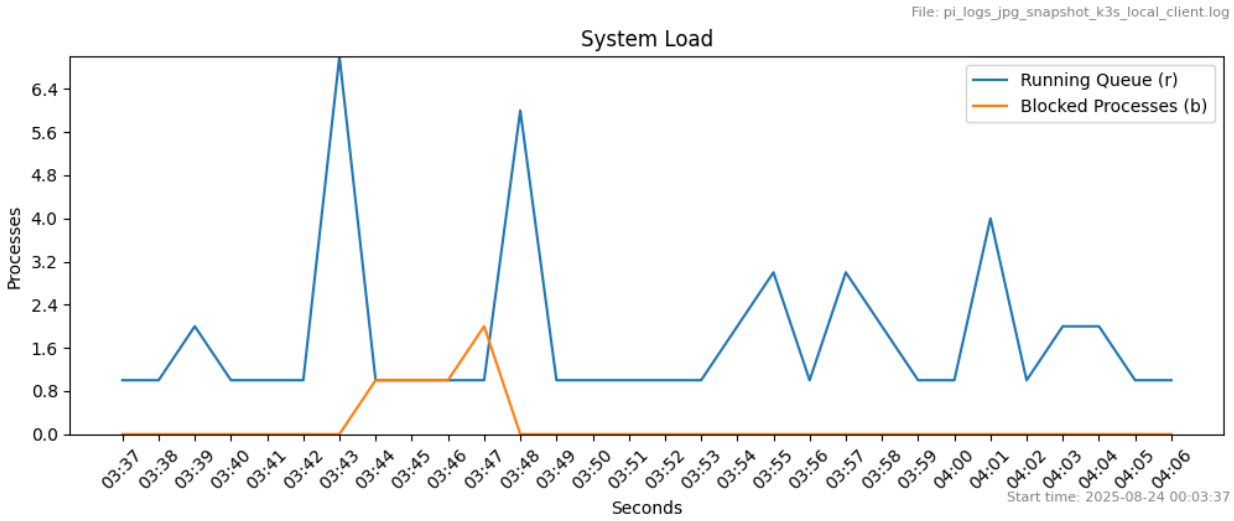

Figure 6.21: System load for NMF Supervisor deployed on k3s

Free memory is stable over 200MB.

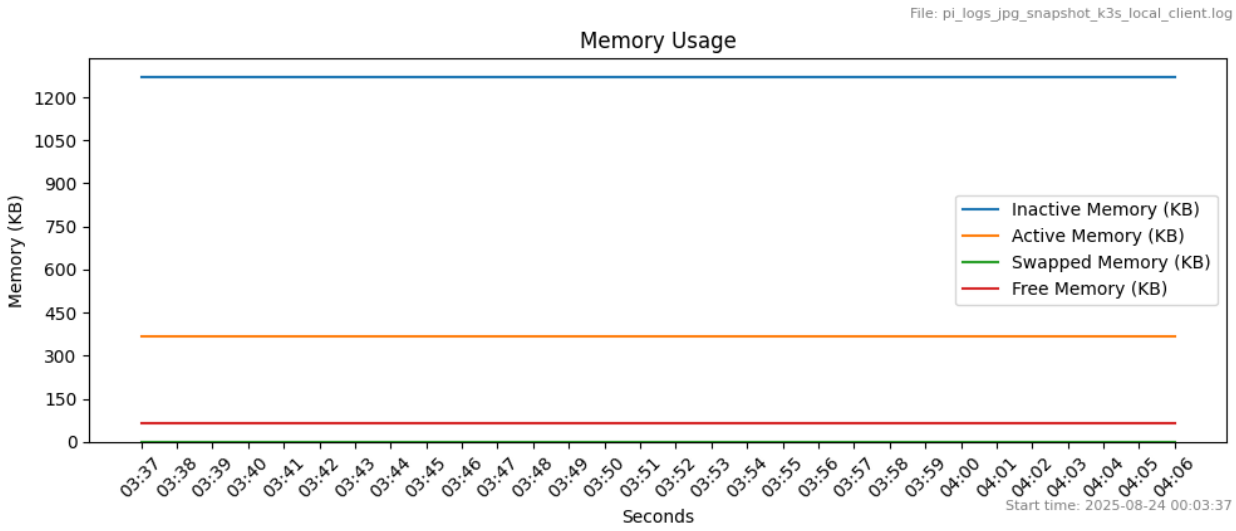

Figure 6.22: Memory usage for NMF Supervisor deployed on k3s

No swap is observed.

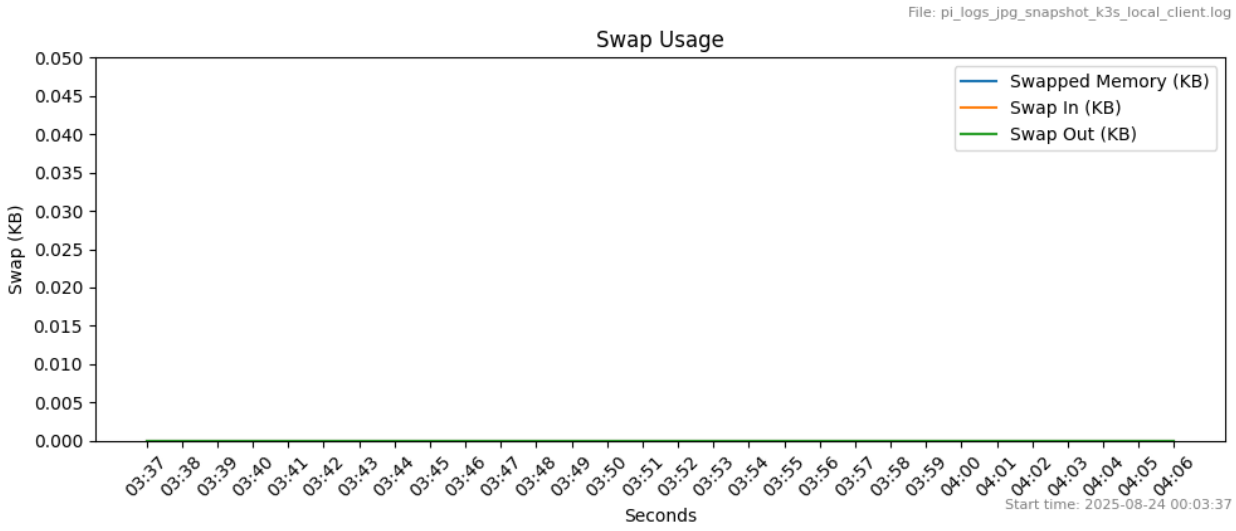

Figure 6.23: Swap usage for NMF Supervisor deployed on k3s

There is low block in and block out and occasionally some events take place.

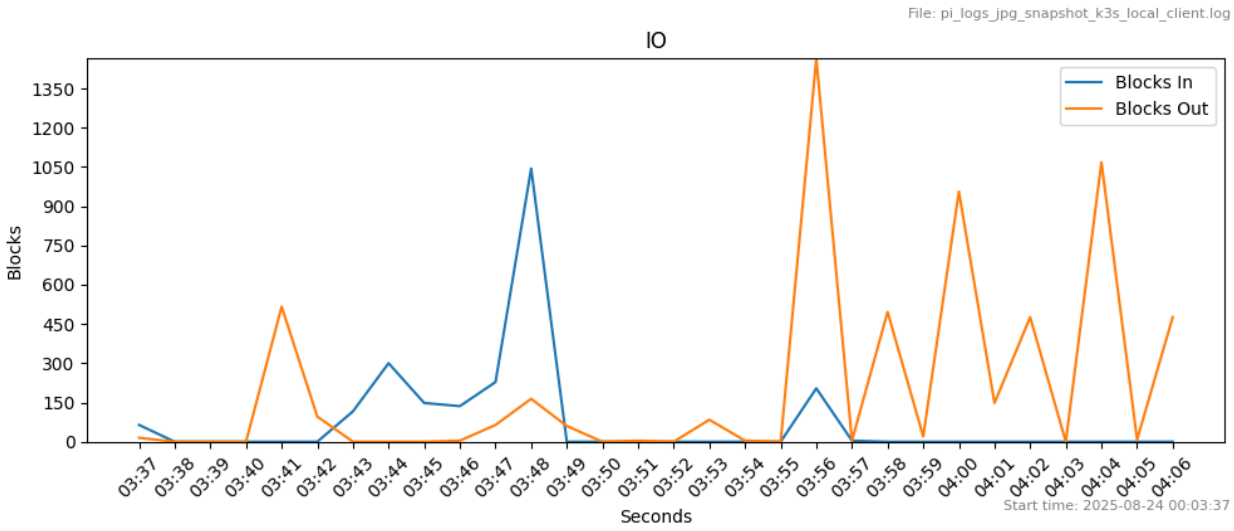

Figure 6.24: IO for NMF Supervisor deployed on k3s

6.1.2.4 NMF clock application deployed without Supervisor

Raspberry Pi with k3s running and an NMF clock app deployed. Interactions take place with the clock app. However, this app does not require the supervisor to run, as no hardware access is involved. The emphasis is on the impact of the supervisor on the resource utilization.

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   3    0            3          222          944          541    0    0    37    34  593 1085   3   2  94   1   0   0 2025-08-25 01:14:48
   1    0            3          222          944          541    0    0     0   236 2723 4963   2   2  96   1   0   0 2025-08-25 01:14:49
   1    0            3          222          944          541    0    0     0   212 2789 4946   2   2  95   1   0   0 2025-08-25 01:14:50
   1    0            3          222          944          541    0    0     0   212 2989 5569   3   4  93   0   0   0 2025-08-25 01:14:51
   2    0            3          222          944          541    0    0     0   216 2368 4362   5   2  93   1   0   0 2025-08-25 01:14:52
   1    0            3          222          944          541    0    0     0   216 2050 3672   2   1  97   1   0   0 2025-08-25 01:14:53
   2    0            3          222          944          541    0    0     0   220 2142 3943   3   1  95   1   0   0 2025-08-25 01:14:54
   1    0            3          222          944          541    0    0     0   592 2721 4789   5   2  93   1   0   0 2025-08-25 01:14:55
   1    0            3          222          944          541    0    0     0   236 2383 4181   4   2  95   0   0   0 2025-08-25 01:14:56
   1    0            3          222          944          541    0    0     0   212 2171 3881   2   1  96   1   0   0 2025-08-25 01:14:57
   1    0            3          222          944          541    0    0     0   216 2320 4142   3   2  95   1   0   0 2025-08-25 01:14:58
   1    0            3          222          944          541    0    0     0   212 2084 3705   2   1  96   1   0   0 2025-08-25 01:14:59
   1    0            3          222          944          541    0    0     0   528 2633 4536   4   2  93   2   0   0 2025-08-25 01:15:00
   1    0            3          222          944          541    0    0     0   228 2634 4804   2   2  95   1   0   0 2025-08-25 01:15:01
   1    0            3          222          944          541    0    0     0   216 2243 4078   2   2  95   1   0   0 2025-08-25 01:15:02
   1    0            3          222          944          541    0    0     0   212 2077 3722   1   1  97   1   0   0 2025-08-25 01:15:03
   1    0            3          222          944          541    0    0     0   212 2384 4251   4   2  93   1   0   0 2025-08-25 01:15:04
   1    0            3          222          944          541    0    0     0   216 2280 3984   4   2  94   0   0   0 2025-08-25 01:15:05
   1    0            3          222          944          541    0    0     0   216 2691 4682   3   1  95   1   0   0 2025-08-25 01:15:06
   1    0            3          222          944          541    0    0     0   212 2281 4121   3   2  94   1   0   0 2025-08-25 01:15:07
   1    0            3          222          944          541    0    0     0   216 2253 3971   5   1  93   1   0   0 2025-08-25 01:15:08
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
   r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
   2    0            3          222          944          541    0    0     0   236 2223 3995   3   2  95   0   0   0 2025-08-25 01:15:09
   1    0            3          222          944          541    0    0     0   216 2433 4271   4   2  94   1   0   0 2025-08-25 01:15:10
   1    0            3          222          944          541    0    0    16   368 3743 6783   4   5  91   1   0   0 2025-08-25 01:15:11
   1    0            3          222          944          541    0    0     0   260 2384 4197   4   2  94   1   0   0 2025-08-25 01:15:12
   1    0            3          222          944          541    0    0     0   212 2005 3547   1   2  97   0   0   0 2025-08-25 01:15:13
   1    0            3          222          944          541    0    0     0   212 2318 4097   3   2  95   1   0   0 2025-08-25 01:15:14
   1    0            3          222          944          541    0    0     0   220 2962 5016   4   4  92   0   0   0 2025-08-25 01:15:15
   1    0            3          222          944          541    0    0     0   596 2032 3609   2   2  96   1   0   0 2025-08-25 01:15:16
   1    0            3          222          944          541    0    0     0   236 2167 3797   3   2  95   1   0   0 2025-08-25 01:15:17
```

Figure 6.25: Raw metrics for NMF Clock app without Supervisor deployed on k3s

The idle CPU is over 90% all the time. There is some usage on the user and system levels.

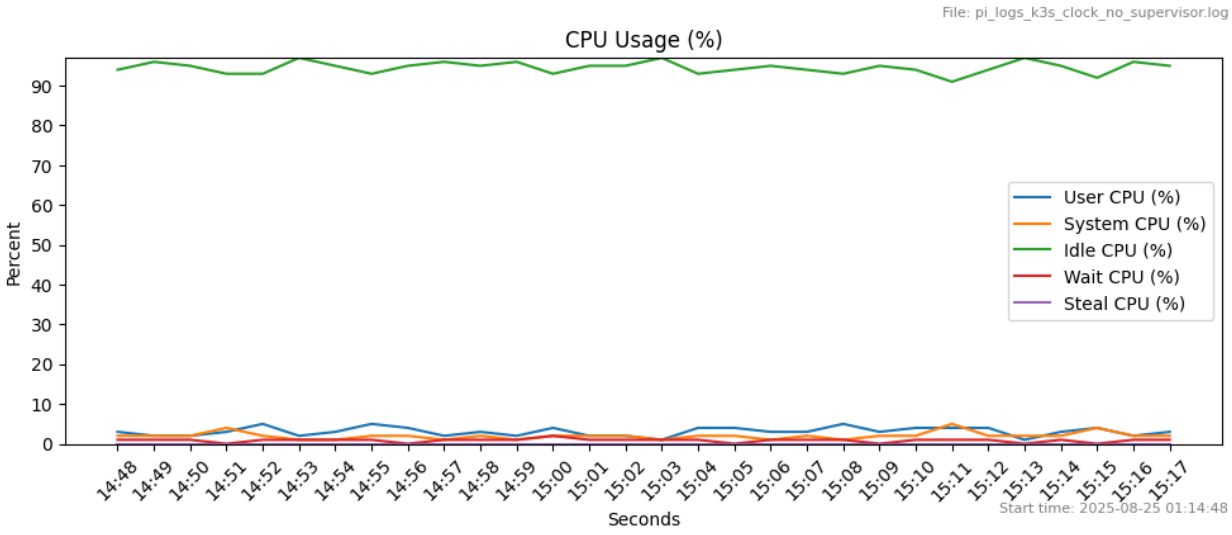

Figure 6.26: CPU usage for NMF Clock app without Supervisor deployed on k3s

There are no blocked processes.

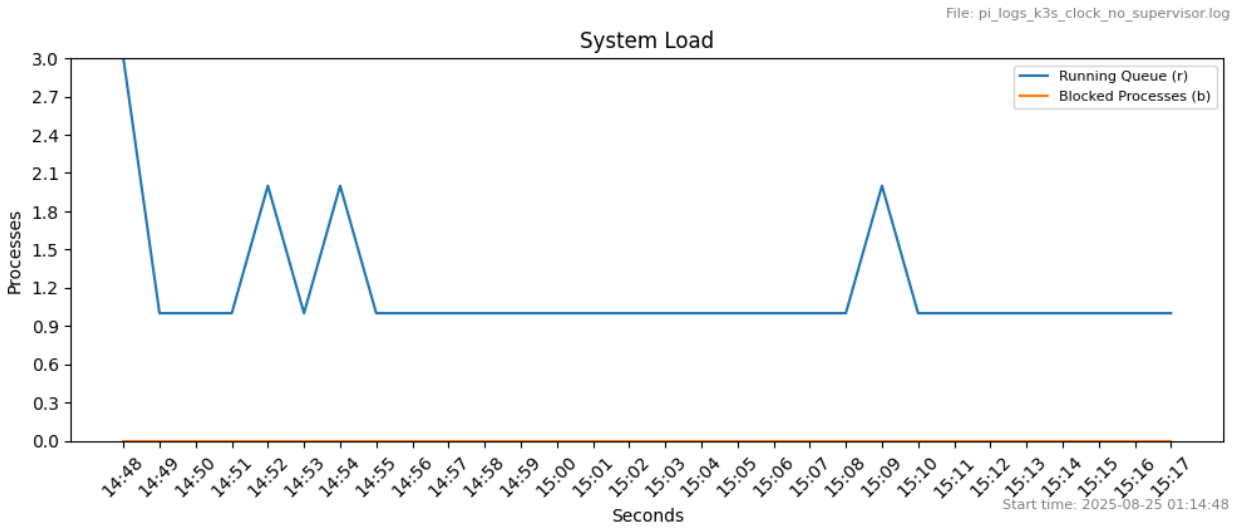

Figure 6.27: System load for NMF Clock app without Supervisor deployed on k3s

Free memory is over 200MB, while there is active memory ~500MB and the inactive memory is more than 900MB.

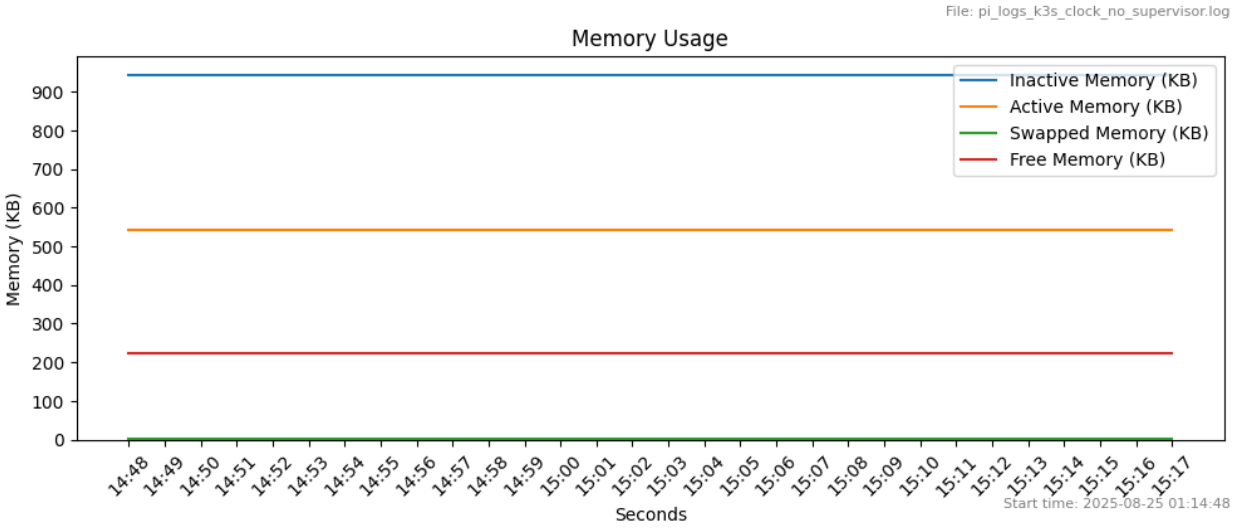

Figure 6.28: Memory usage for NMF Clock app without Supervisor deployed on k3s

No disk swapping is observed.

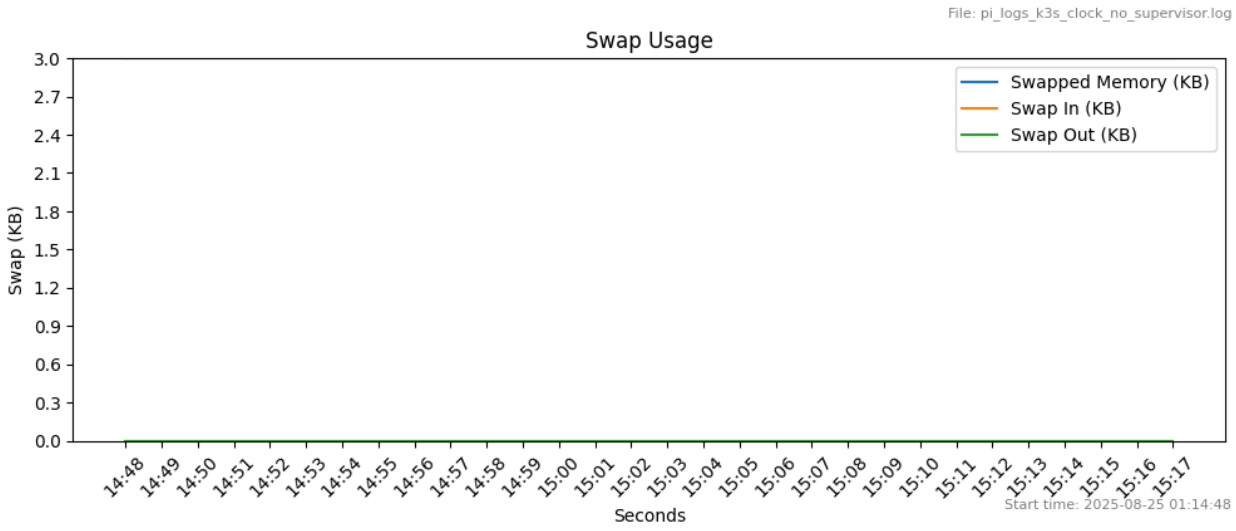

Figure 6.29: Swap usage for NMF Clock app without Supervisor deployed on k3s

From time to time there is some moderate block output, with minimal block input events.

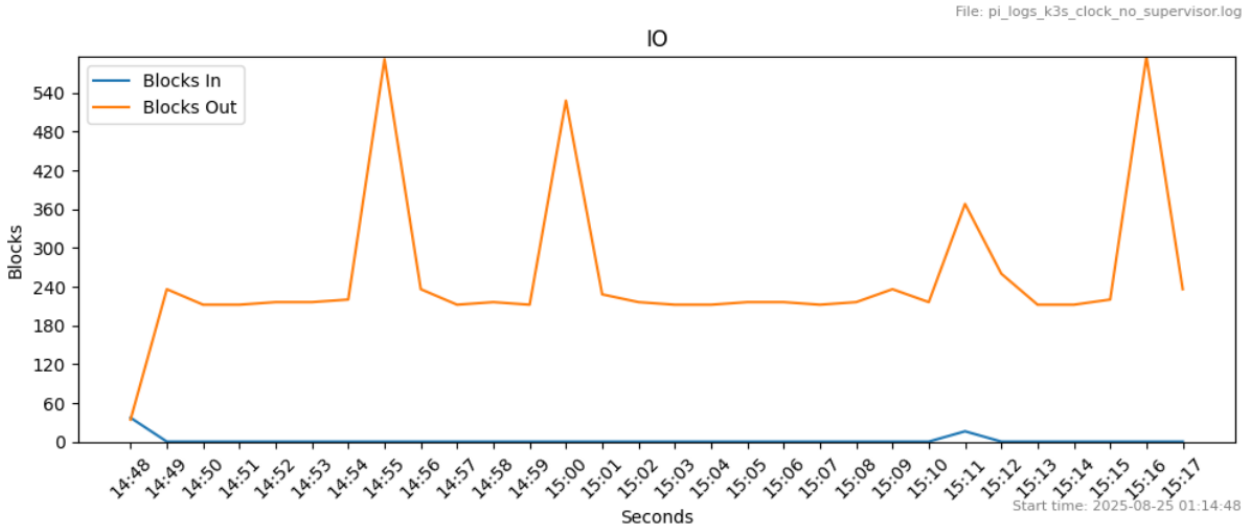

Figure 6.30: IO for NMF Clock app without Supervisor deployed on k3s

6.1.3 Conventional NMF payload on JVM

A set of performance tests with the NMF installation running conventionally with JVM follows.

6.1.3.1 Only Supervisor deployed, no action

This scenario examines the supervisor running standalone in JVM using the conventional NMF setup. No other NMF app is running.

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
  r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
  2    0          199          514          767          387    0    0    57   173  471  812   5   1  92   1   0   0 2025-08-25 02:25:48
  1    0          199          514          767          387    0    0     0   232 1532 2041   1   1  98   1   0   0 2025-08-25 02:25:49
  1    0          199          514          767          387    0    0     0   228 1555 2063   2   1  97   0   0   0 2025-08-25 02:25:50
  1    0          199          514          767          387    0    0     0   248  849 1215   1   1  98   1   0   0 2025-08-25 02:25:51
  1    0          199          514          767          387    0    0     0   244  669 1020   2   0  98   1   0   0 2025-08-25 02:25:52
  1    0          199          514          767          387    0    0     0   232  588  866   0   1  99   1   0   0 2025-08-25 02:25:53[10
  1    0          199          514          767          387    0    0     0   232  601  873   1   1  98   1   0   0 2025-08-25 02:25:54
  1    0          199          514          767          387    0    0     0   228  791 1122   1   1  98   1   0   0 2025-08-25 02:25:55
  1    0          199          514          767          387    0    0     0   228  633  940   1   1  98   1   0   0 2025-08-25 02:25:56
  1    0          199          514          767          387    0    0     0   244  816 1147   1   1  97   1   0   0 2025-08-25 02:25:57
  1    0          199          514          767          387    0    0     0   252  575  853   1   1  99   0   0   0 2025-08-25 02:25:58
  1    0          199          514          767          387    0    0     0   232  531  802   0   0  99   1   0   0 2025-08-25 02:25:59
  2    0          199          514          767          387    0    0     0   580  826 1199   2   1  96   1   0   0 2025-08-25 02:26:00
  1    0          199          514          767          387    0    0     0   340  689  988   1   1  98   1   0   0 2025-08-25 02:26:01
  1    0          199          514          767          387    0    0     0   272  782 1098   3   0  96   1   0   0 2025-08-25 02:26:02
  1    0          199          514          767          387    0    0     0   232  672  962   0   0  99   0   0   0 2025-08-25 02:26:03
  1    0          199          514          767          387    0    0     0   256  531  842   1   0  98   1   0   0 2025-08-25 02:26:04
  3    0          199          514          767          387    0    0     0   228  586  922   0   0  99   1   0   0 2025-08-25 02:26:05
  1    0          199          514          767          387    0    0     0   228  668 1038   0   1  98   1   0   0 2025-08-25 02:26:06
  1    0          199          514          767          387    0    0     0   236  736 1052   1   1  98   1   0   0 2025-08-25 02:26:07
  1    0          199          514          767          387    0    0     0   240  589  884   1   0  99   0   0   0 2025-08-25 02:26:08
  2    0          199          514          767          387    0    0     0   228  676  999   2   0  98   1   0   0 2025-08-25 02:26:09
  1    0          199          514          767          387    0    0     0   236  644  998   1   1  98   1   0   0 2025-08-25 02:26:10
  1    0          199          514          767          387    0    0     0   228  782 1246   1   1  98   1   0   0 2025-08-25 02:26:11
  1    0          199          514          767          387    0    0     0   260  688 1032   1   1  98   1   0   0 2025-08-25 02:26:12
  1    0          199          514          767          387    0    0     0   240  583  878   1   1  98   0   0   0 2025-08-25 02:26:13
  1    0          199          514          767          387    0    0     0   228  588  859   3   0  97   1   0   0 2025-08-25 02:26:14
  1    0          199          514          767          387    0    0     0   240  730 1009   0   0  99   1   0   0 2025-08-25 02:26:15
  1    0          199          514          767          387    0    0     0   264  636  953   1   0  99   1   0   0 2025-08-25 02:26:16
  1    0          199          514          767          387    0    0     0   232  719 1000   1   1  99   0   0   0 2025-08-25 02:26:17
```

Figure 6.31: Raw metrics for standalone conventional NMF Supervisor

The CPU is idle more than 96% of the time. Rare CPU wait occurs (<1%). System usage is really low.

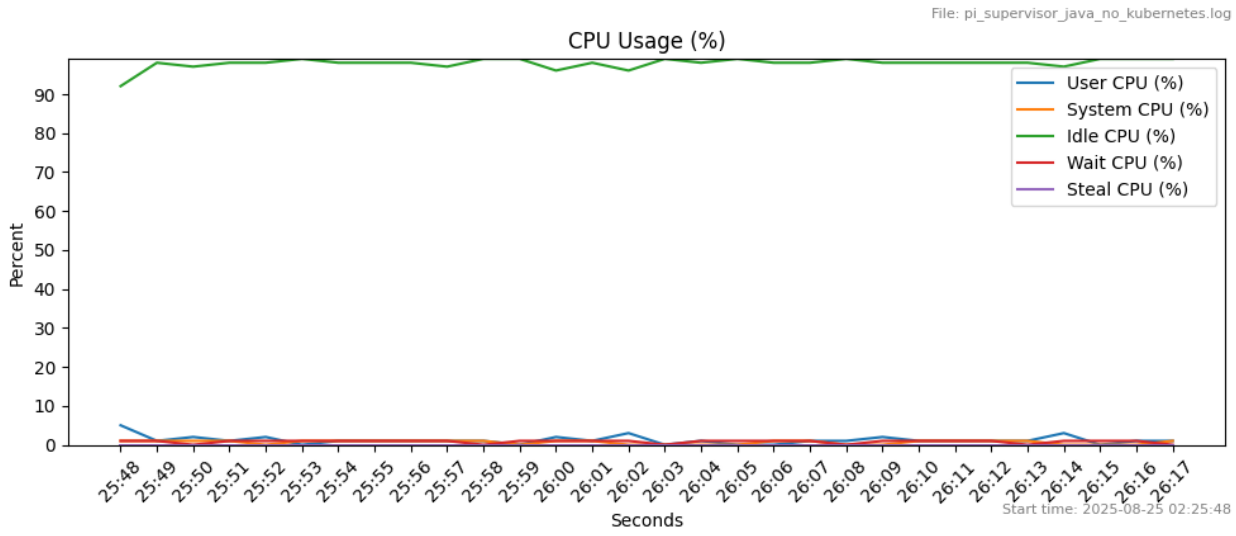

Figure 6.32: CPU usage for standalone conventional NMF Supervisor

There are no blocked processes.

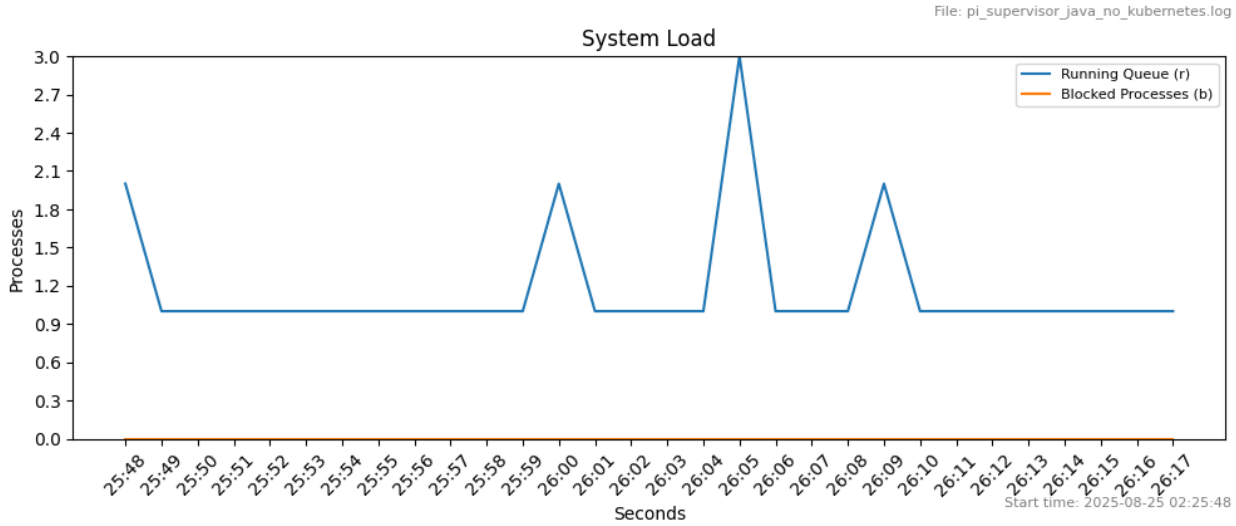

Figure 6.33: System load for standalone conventional NMF Supervisor

Free memory is above 500MB, with inactive memory at ~750MB and active low at ~400MB. Some memory swapping (compression) is used.

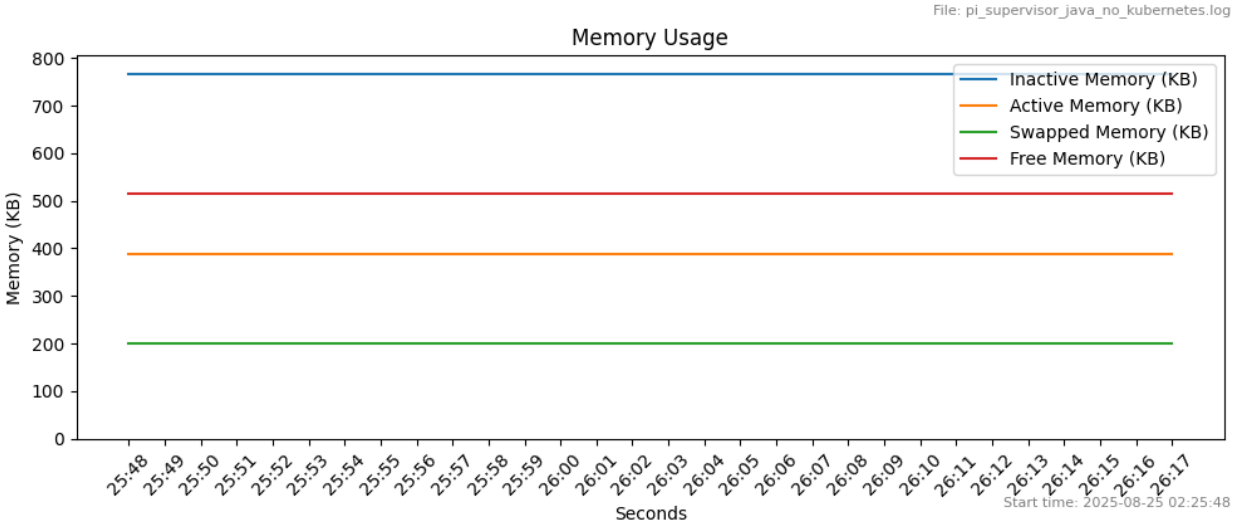

Figure 6.34: Memory usage for standalone conventional NMF Supervisor

No swap activity (although some swap is in use).

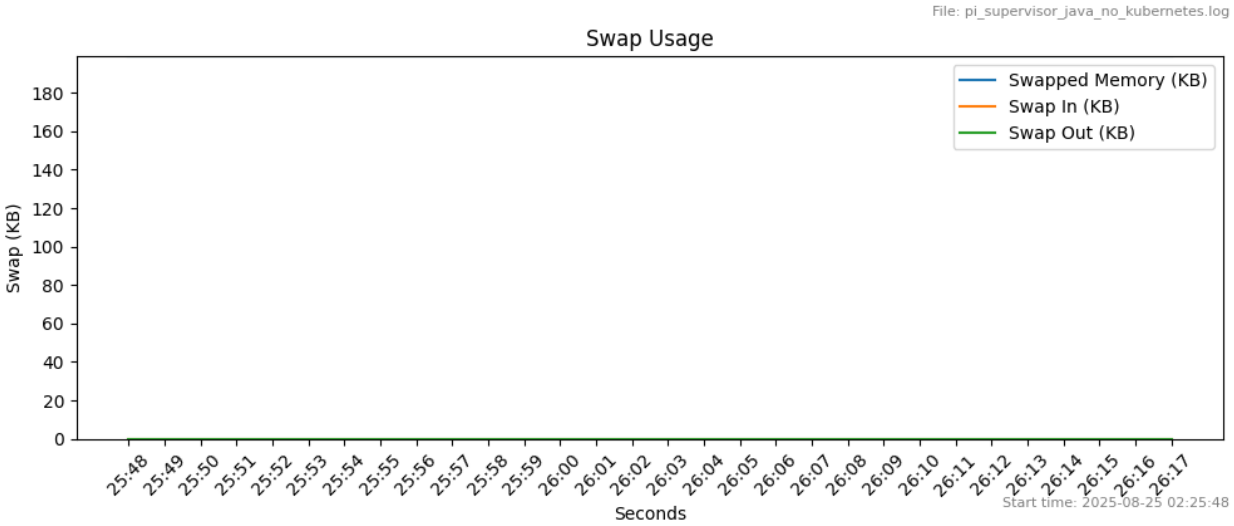

Figure 6.35: Swap usage for standalone conventional NMF Supervisor

Occasional block output and almost no block input.

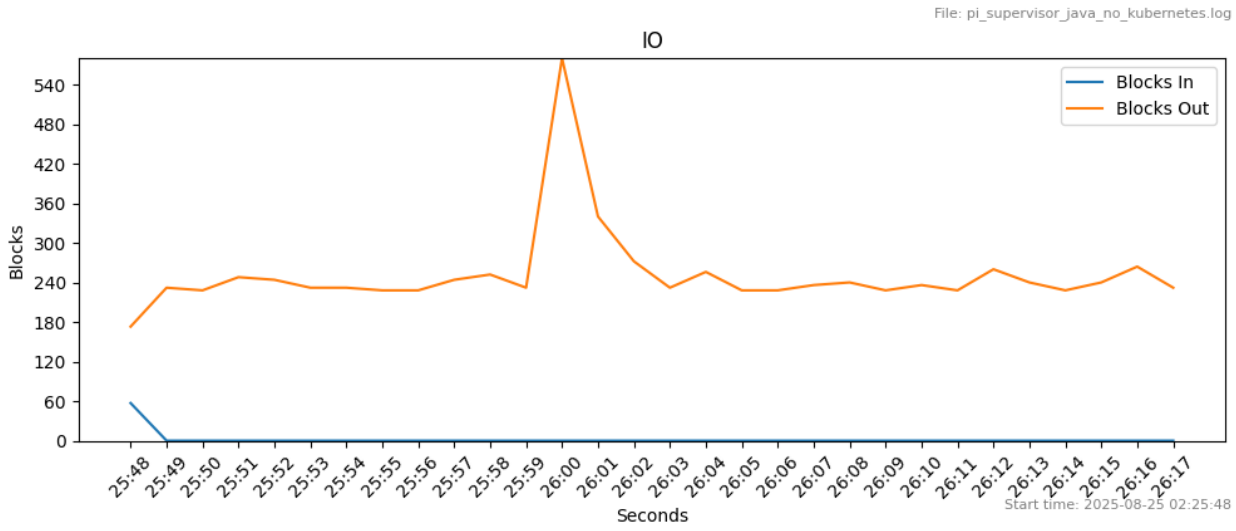

Figure 6.36: IO for standalone conventional NMF Supervisor

6.1.3.2 Camera and Supervisor apps deployed, camera access and 3 consecutive jpg snapshots

This is the main scenario across system designs with the camera app running via the NMF Supervisor.

```
--procs-- -----------------------memory---------------------- ---swap-- -----io---- -system-- ----------cpu---------- -----timestamp-----
  r    b         swpd         free        inact       active   si   so    bi    bo   in   cs  us  sy  id  wa  st  gu               EEST
  2    0          198          300          779          600    0    0    51   156  430  731   5   1  93   1   0   0 2025-08-25 02:55:14
  1    0          198          300          779          600    0    0     0   260 1326 2124   1   1  98   1   0   0 2025-08-25 02:55:15
  1    0          198          300          779          600    0    0     0   260 1056 1664   1   1  98   1   0   0 2025-08-25 02:55:16
  1    0          198          300          779          600    0    0     0   228  702 1055   1   0  99   0   0   0 2025-08-25 02:55:17
  1    0          198          300          779          600    0    0     0   232  630  964   0   0  98   1   0   0 2025-08-25 02:55:18
  1    0          198          300          779          600    0    0    16   232  736  980   7   0  92   1   0   0 2025-08-25 02:55:19
  1    0          198          300          779          600    0    0     0   252  750 1067   1   0  99   0   0   0 2025-08-25 02:55:20
  1    0          198          300          779          600    0    0     0   232 1387 1959   7   1  92   1   0   0 2025-08-25 02:55:21
  1    0          198          300          779          600    0    0     0   244  626  911   0   0  99   1   0   0 2025-08-25 02:55:22
  1    0          198          300          779          600    0    0     0   252  592  882   0   1  99   1   0   0 2025-08-25 02:55:23
  1    0          198          300          779          600    0    0     0   232  620  899   1   0  99   1   0   0 2025-08-25 02:55:24
  1    0          198          300          779          600    0    0     0   252  659  966   0   1  99   0   0   0 2025-08-25 02:55:25
  1    0          198          300          779          600    0    0     0   280  708 1020   1   1  98   1   0   0 2025-08-25 02:55:26
  2    0          198          300          779          600    0    0     0   228  868 1112  10   1  89   0   0   0 2025-08-25 02:55:27
  1    0          198          300          779          600    0    0     0  1488 1696 2500   4   1  92   3   0   0 2025-08-25 02:55:28
  1    0          198          300          779          600    0    0     0   472  878 1221   0   0  98   1   0   0 2025-08-25 02:55:29
  1    0          198          300          779          600    0    0     0   232  647  933   0   0  99   1   0   0 2025-08-25 02:55:30
  1    0          198          300          779          600    0    0     0  1252 1828 2472   4   2  92   3   0   0 2025-08-25 02:55:31
  1    0          198          300          779          600    0    0     0   240  586  876   0   0  99   0   0   0 2025-08-25 02:55:32
  1    0          198          300          779          600    0    0     0   476 1081 1354  10   1  89   1   0   0 2025-08-25 02:55:33
  1    0          198          300          779          600    0    0     0  1240 1705 2302   4   1  92   3   0   0 2025-08-25 02:55:34
  2    0          198          300          779          600    0    0     0   496  881 1190   1   0  98   1   0   0 2025-08-25 02:55:35
  1    0          198          300          779          600    0    0     0   236  691  994   0   0  99   1   0   0 2025-08-25 02:55:36
  1    0          198          300          779          600    0    0     0   228  633  897   0   0  99   0   0   0 2025-08-25 02:55:37
  1    0          198          300          779          600    0    0     0   228  630  897   0   0  99   1   0   0 2025-08-25 02:55:38
  1    0          198          300          779          600    0    0     0   228  806 1142   1   1  98   1   0   0 2025-08-25 02:55:39
  1    0          198          300          779          600    0    0     0   252  670  952   1   0  98   1   0   0 2025-08-25 02:55:40
  1    0          198          300          779          600    0    0     0   232  713 1011   1   1  98   1   0   0 2025-08-25 02:55:41
  1    0          198          300          779          600    0    0     0   240  739 1103   1   1  97   0   0   0 2025-08-25 02:55:42
  1    0          198          300          779          600    0    0     0   252  649  912   1   1  98   1   0   0 2025-08-25 02:55:43
```

Figure 6.37: Raw metrics for fully deployed conventional NMF camera app and Supervisor

The user CPU usage occasionally reaches 10%. CPU is mostly (>89%) idle. There is an occasional wait.

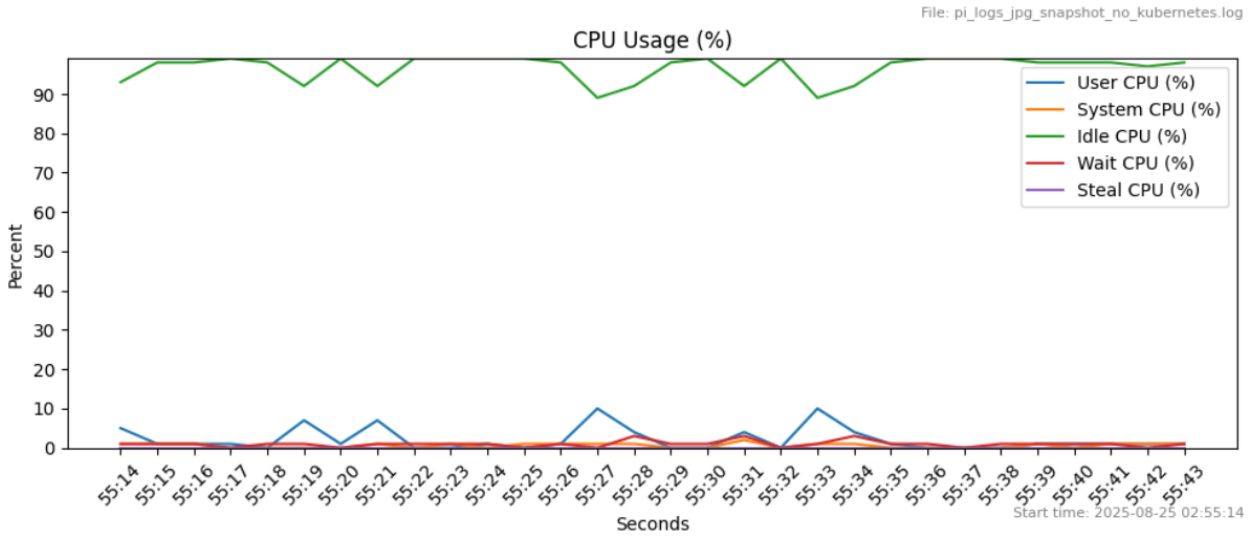

Figure 6.38: CPU usage for fully deployed conventional NMF camera app and Supervisor

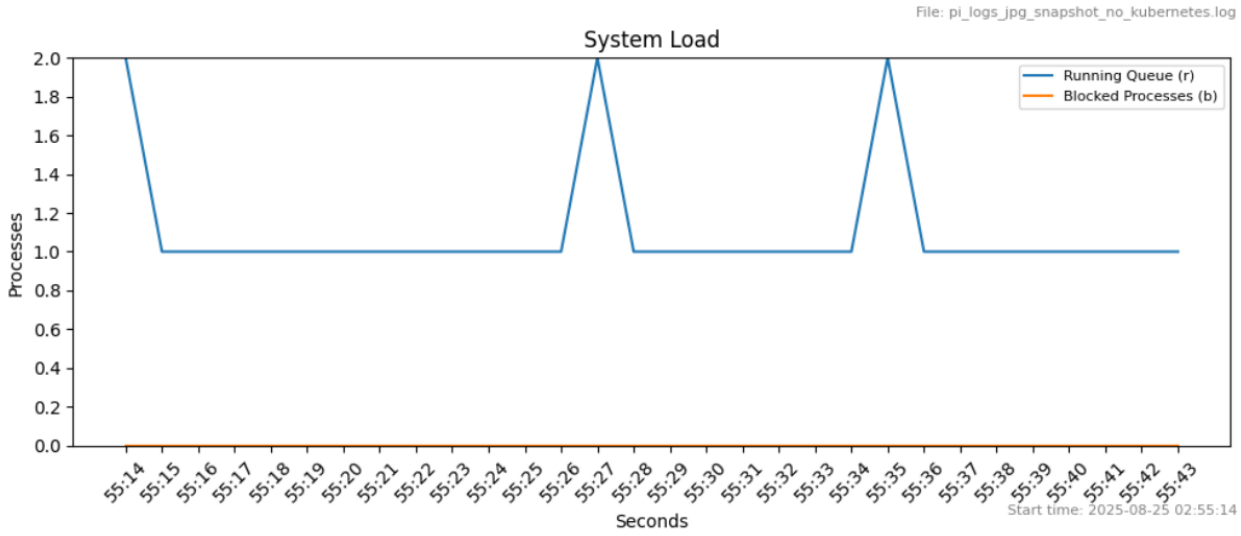

Figure 6.39: System load for fully deployed conventional NMF camera app and Supervisor

Free memory is stable at ~300MB, while the actively used memory is ~600MB and inactive memory is ~750MB. Swap memory compression is close to 200MB.

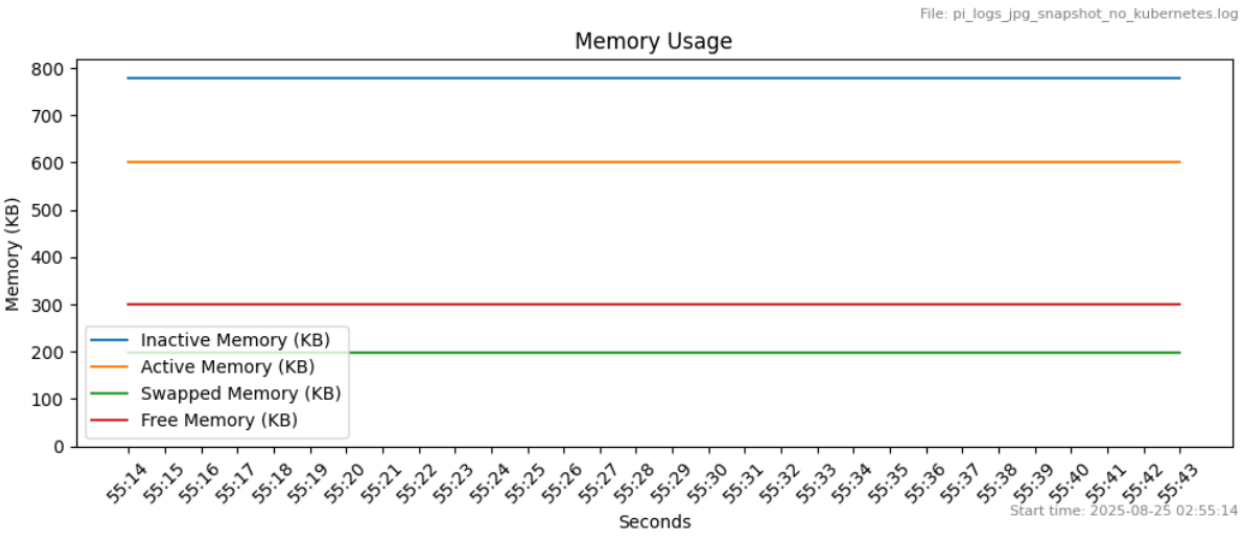

Figure 6.40: Memory usage for fully deployed conventional NMF camera app and Supervisor

No swap in or swap out is observed.

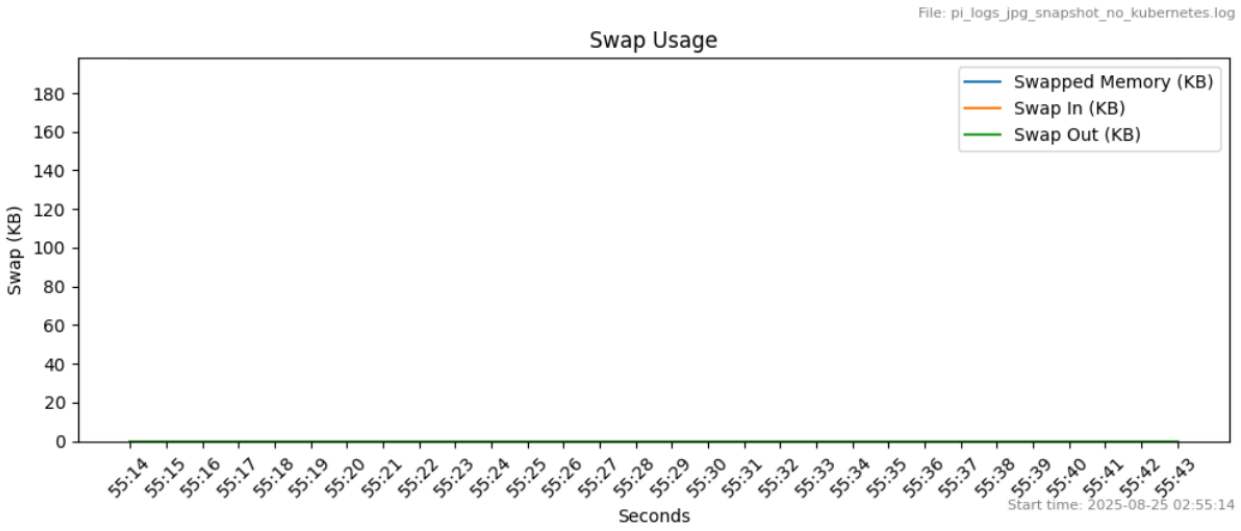

Figure 6.41: Swap usage for fully deployed conventional NMF camera app and Supervisor

Block output is occasionally high, with momentary spikes up to 1488, while the rest of the time it is moderate, above 200.

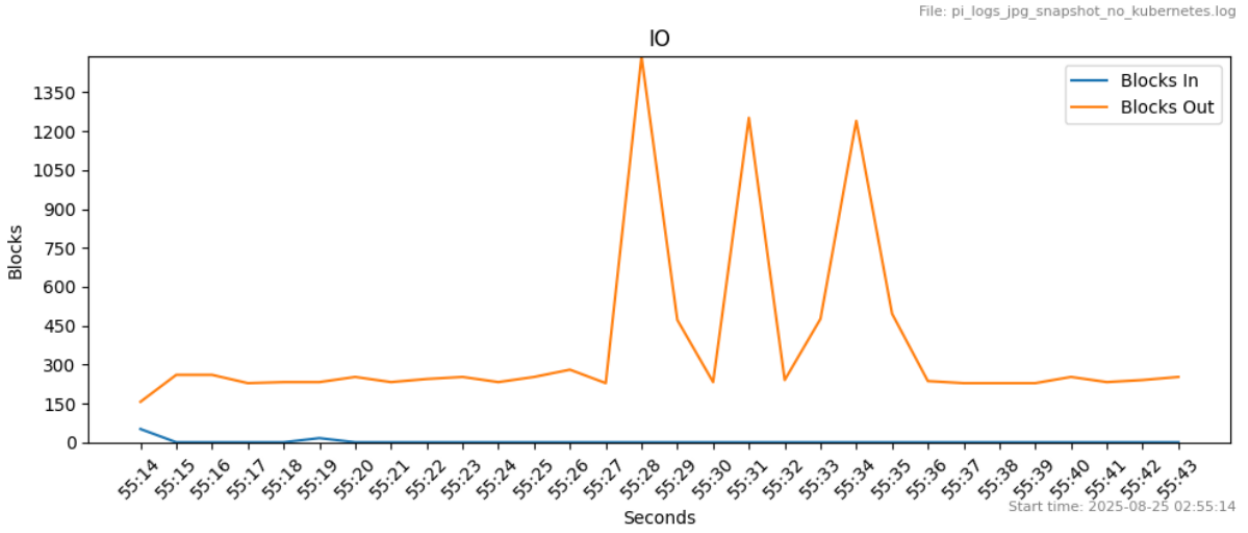

Figure 6.42: IO for fully deployed conventional NMF camera app and Supervisor

6.2 Summary

Below are presented the key summaries of the metrics collected.

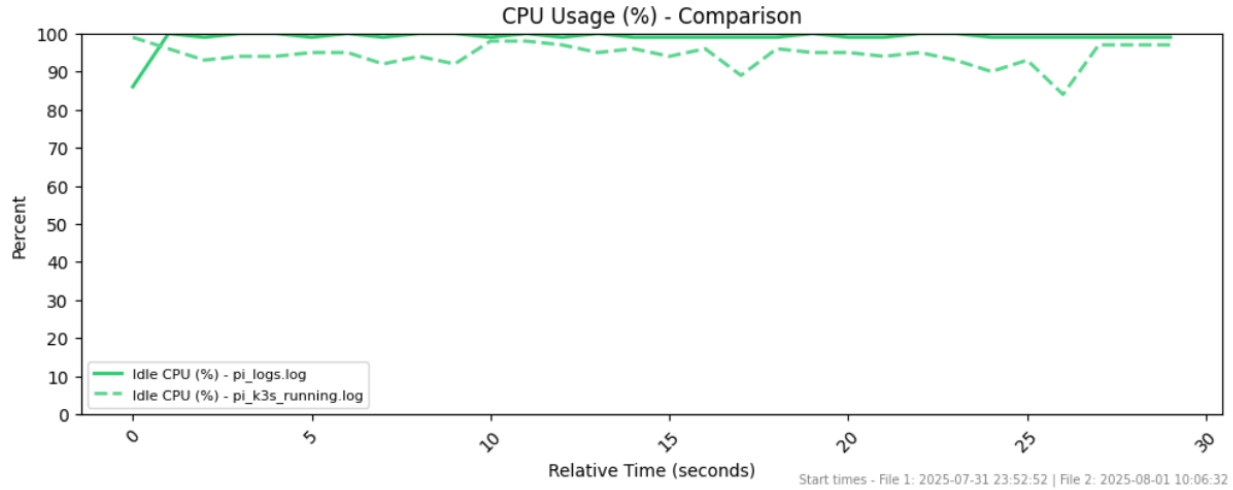

Figure 6.43: CPU usage comparison: only k3s versus only RaspberryPi

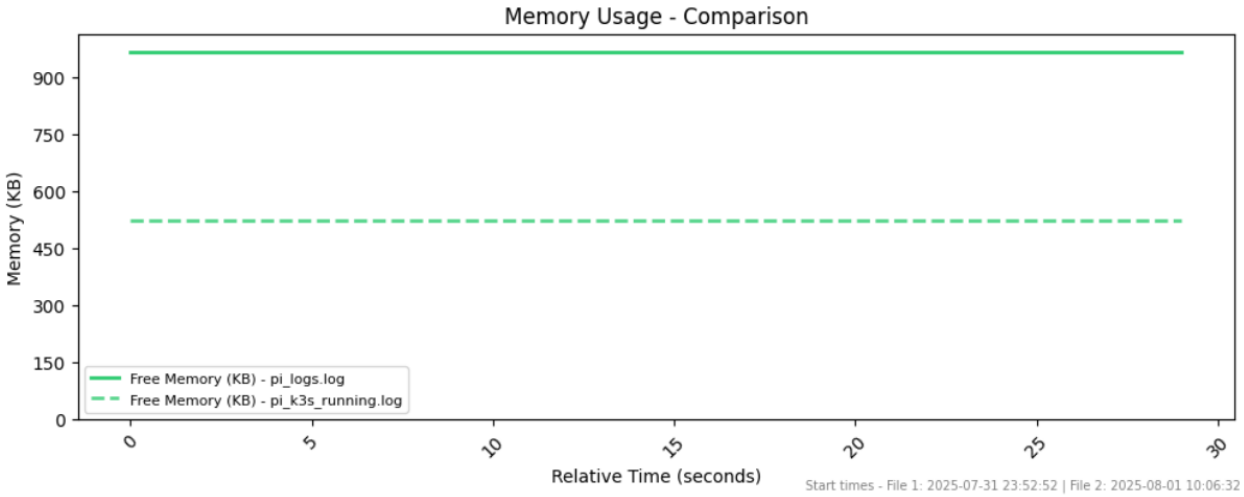

Figure 6.44: Free memory comparison: only k3s versus only RaspberryPi

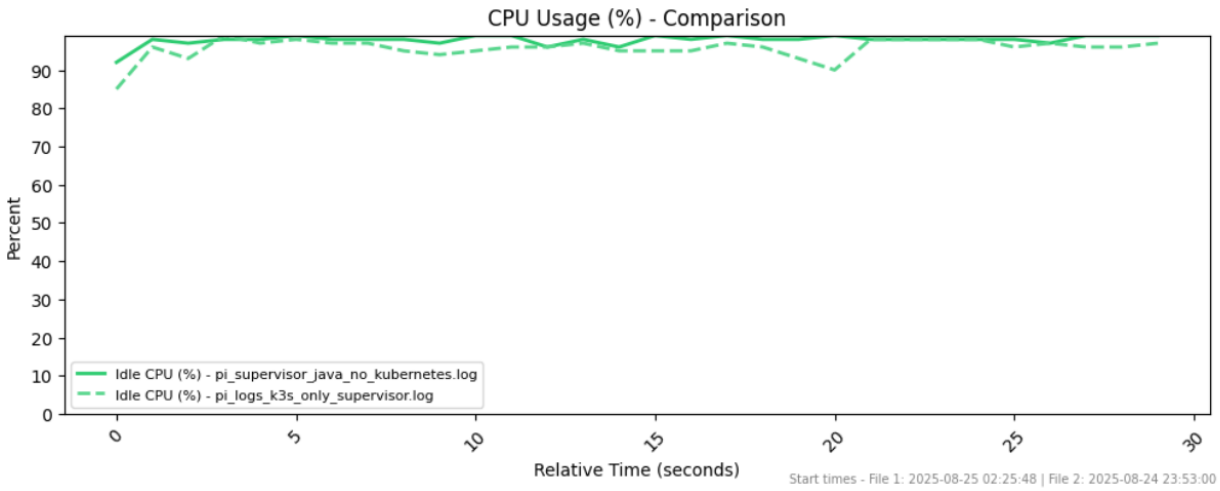

Figure 6.45: CPU usage comparison: k3s with Supervisor versus conventional Supervisor

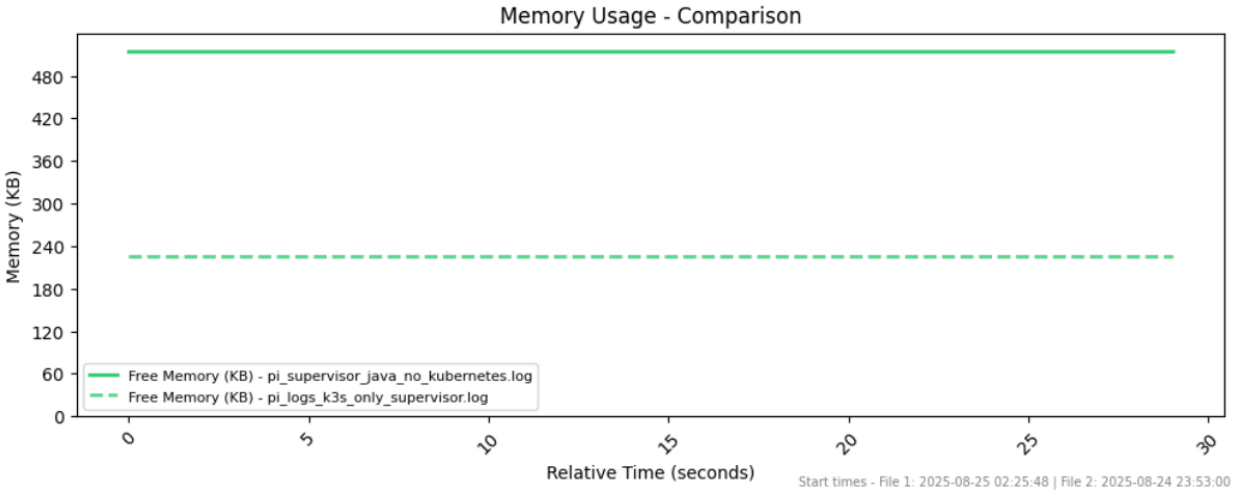

Figure 6.46: Free memory comparison: k3s with Supervisor versus conventional Supervisor

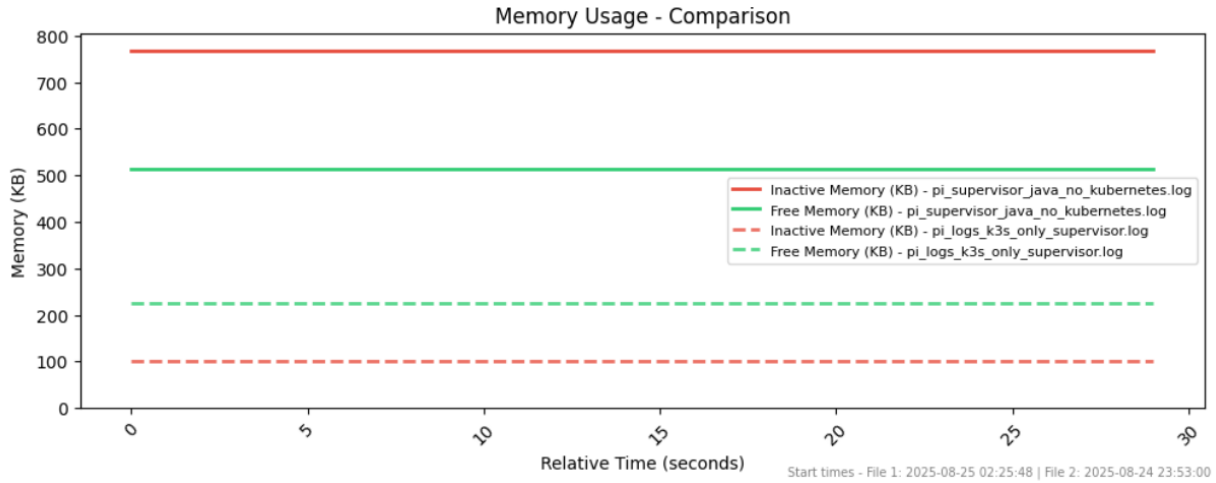

Figure 6.47: Free and inactive memory comparison: k3s with Supervisor versus conventional Supervisor

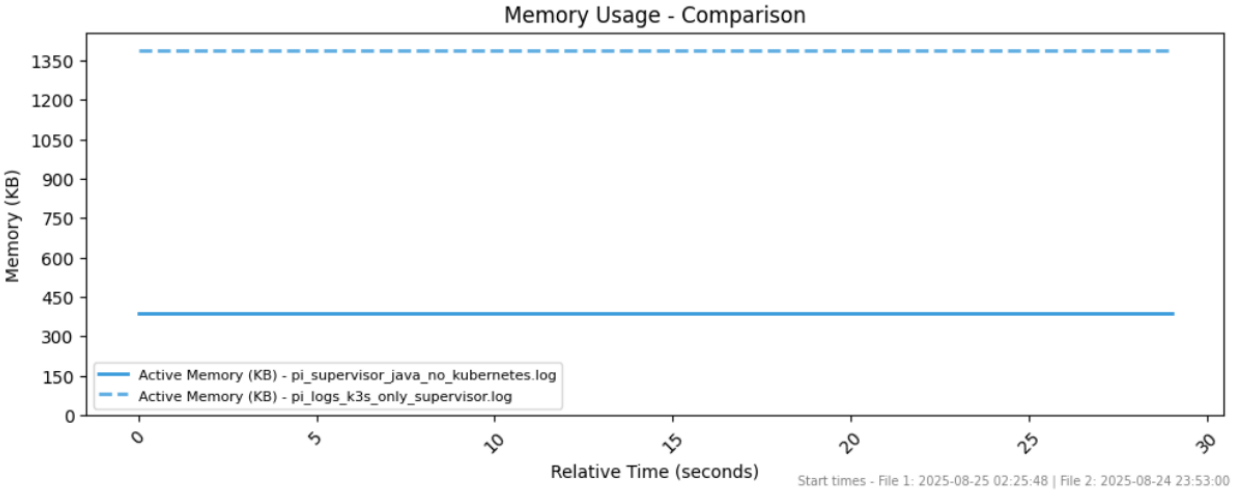

Figure 6.48: Active memory comparison: k3s with Supervisor versus conventional Supervisor

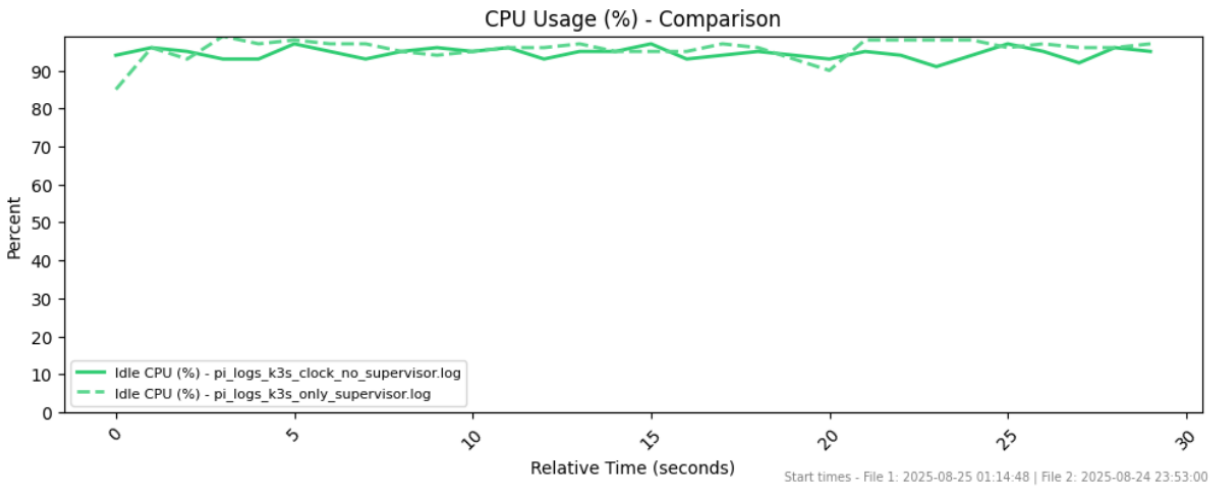

Figure 6.49: CPU usage comparison: k3s with Supervisor vs k3s with clock app

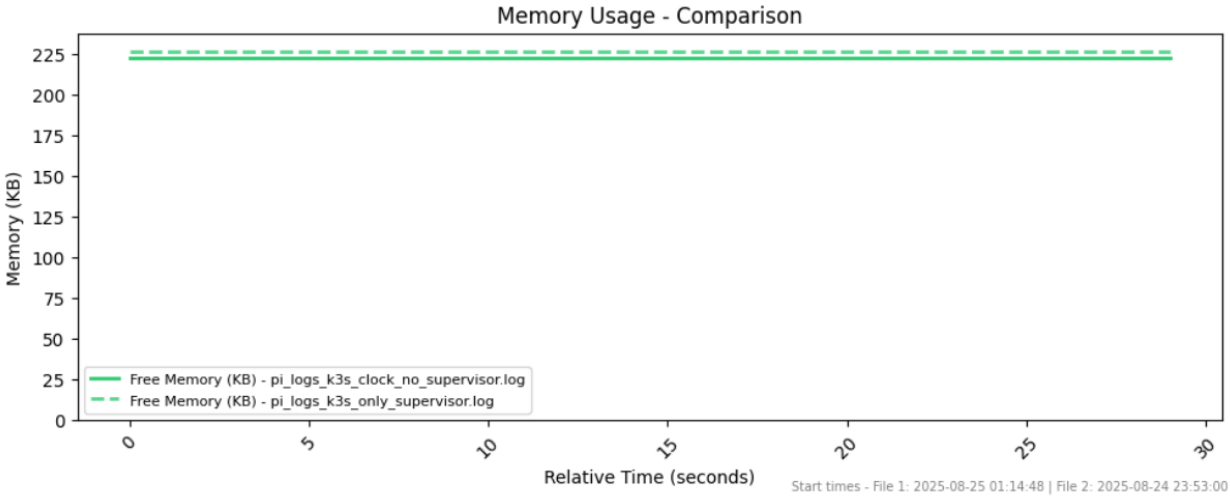

Figure 6.50: Free memory comparison: k3s with Supervisor vs k3s with clock app

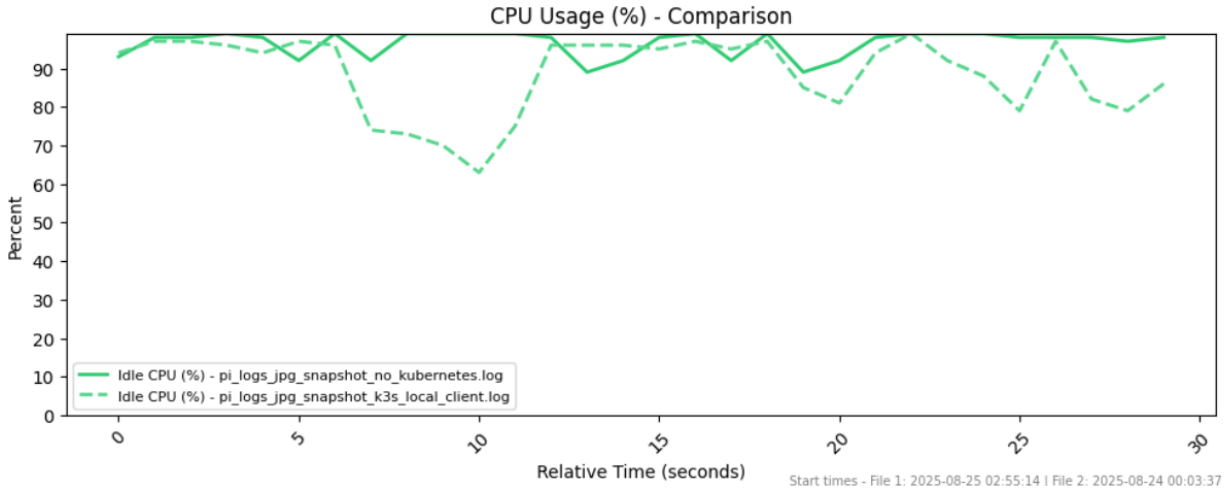

Figure 6.51: CPU usage comparison: Take snapshots with k3s vs with conventional NMF

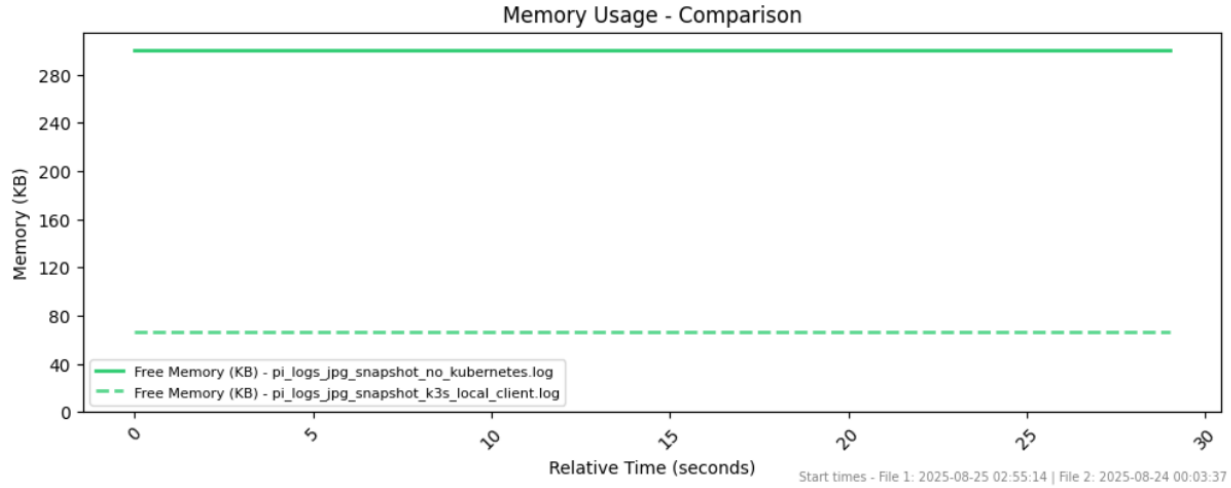

Figure 6.52: Free memory comparison: Take snapshots with k3s vs with conventional NMF

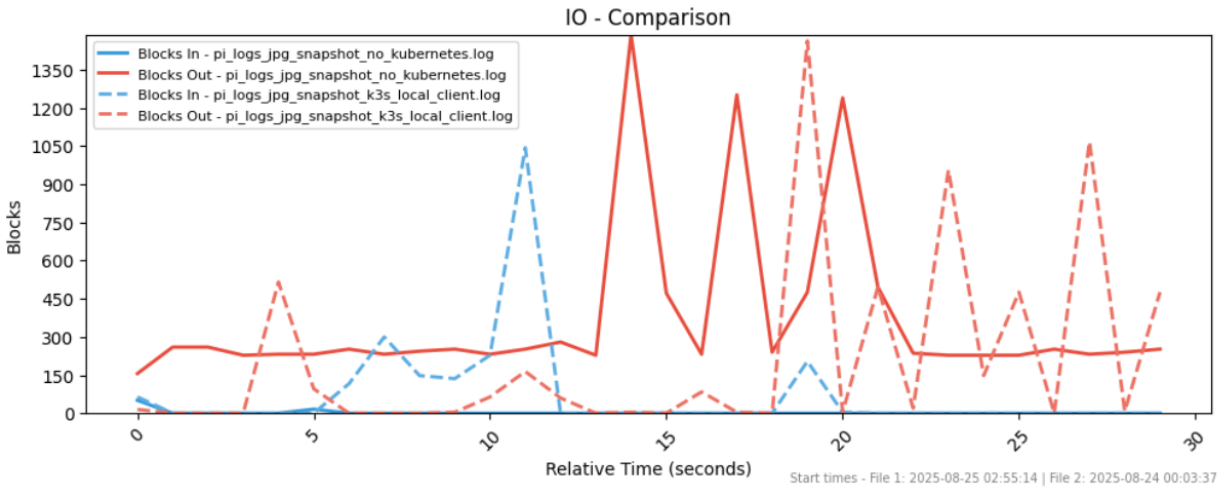

Figure 6.53: IO comparison: Take snapshots with k3s vs with conventional NMF

As a conclusion, the overhead brought by the proposed solution integration of NMF with Kubernetes is low. It should also be noted that the overhead becomes negligible as the number of services deployed grows.

Using Kubernetes will prove to be a much more scalable solution for a strong software satellite ecosystem. In the new proposed way using DevOps and CI/CD, there would be automated pipelines for building (container images and tagging) and deploying NMF services, while utilizing the inherent strengths of Kubernetes, such as health checks, automated rollbacks.

It is clear that in the "happy path" when everything runs smoothly, the operational effort is reduced by more than 75%.

The value of CI/CD with Kubernetes skyrockets in case of a failure, where the whole process would have to be re-run from scratch to deploy the old artifact, doubling the amount of work and attention. Now, Kubernetes ensures downtime to be 0 seconds for deployments, whereas the downtime by shipping a faulty version in the old flow would be at least at the scale of minutes (~10) even with the most lightweight jar.

7. CONCLUSION & FUTURE WORK

This thesis addresses the growing disparity between rapid advancements in hardware on board contemporary ”New Space” missions and the comparatively stagnant software operations methodologies and technology adoption. Traditional approaches, while functional, lack the agility, automation, and scalability that are standard in modern terrestrial software development. To bridge this gap, this thesis integrates ESA's NMF with the Kubernetes container orchestration platform, effectively introducing the ”Space-Age DevOps” paradigm.

This work provides the first documented proof-of-concept of deploying and operating NMF services on a lightweight k3s distribution. Key findings include the specific containerization strategy for NMF components, the development of network configurations necessary for this onboard distributed environment, and the standardization of application deployment via Helm packaging. Furthermore, the thesis presents a novel CI/CD methodology tailored to the context of space operations, which utilizes a pre-deployment image staging mechanism to ensure deterministic deployments despite intermittent connectivity. We consider our primary contribution to be a novel architectural blueprint that refactors NMF from its traditional (but flight-validated) Service-Oriented Architecture into a fully containerized, microservices-based model. This was achieved by running a PoC on resource-constrained hardware similar to the actual environment setting in a satellite which validates core functionalities, from onboard image capture and edge processing to ground storage, while adding a minimal resource consumption overhead. By facilitating this fusion of space-specific frameworks with mainstream DevOps tools, this thesis enhances mission flexibility, reliability and portability but also enables advanced edge computing capabilities, with a view towards more autonomous, resilient, and powerful satellite operations in the New Space era.

The incorporation of DevOps principles into the domain of space exploration promises substantial benefits. Traditionally, space missions have adhered to stringent schedules, often characterized by extended development cycles and complex mission operations. The infusion of DevOps practices injects a new level of agility, efficiency, and adaptability into these missions. By embracing automation, continuous integration, and rapid deployment, space missions can respond more dynamically to evolving mission requirements and unforeseen challenges.

The robust monitoring and error-handling capabilities inherent to DevOps practices enhance the reliability and resilience of space systems. In a domain where mission success is paramount, these gains are of immeasurable value. The introduction of DevOps methodologies to space missions fundamentally changes the landscape by enabling space agencies and organizations to iterate, adapt, and innovate more swiftly.

Some of the key advancements provided are:

- Application self-healing

- Better compute resource management
- More flexible and resilient infrastructure on board that can support more complex software architectures
- New version deployment for satellite software apps becomes a trivial task.
- Satellite services can be scaled up or down, reducing operation costs (like energy consumption), manually when needed or automatically based on a rule set. The restriction of a maximum 1 replica per application imposed by the conventional NMF architecture is overcome.
- Better software administration, with integrated monitoring functionalities, easily parameterized and configurable applications.
- Elaborate network and security rules setup.
- Out of the box job scheduling which can be really helpful in correlation with satellite's orbital characteristics
- Add a technology/language agnostic layer of abstraction, enabling the usage of diverse frameworks for the same satellite.
- Adhering with the industry standards for software development

7.1 Future work

Future work may advance the presented ”Space-Age DevOps” paradigm by i) a refinement of the core framework and ii) its operational scalability. A primary objective is the re-architecting of the NMF Supervisor, stripping it of redundant, Kubernetes-handled application management functionalities to operate purely as a lightweight Platform Service dedicated exclusively to hardware abstraction. Also, this streamlined architecture could be validated by executing the PoC on the actual OPSsat target platform, moving beyond the current hardware emulation. To fully realize the framework-agnostic benefits of the proposed model, further PoCs may integrate applications developed in diverse languages and frameworks, such as Python runtimes, demonstrating true polyglot operations on orbit. On the orchestration front, a significant advancement can be the development of a custom Kubernetes Operator; this operator will manage deployment complexity and execute optimized, autonomous resource management, such as scheduling service activation and deactivation based on orbital parameters, mission schedules, or service dependencies. Furthermore, this work lays the foundation for a subsequent direction focused on extension for multi-satellite clusters and constellations, in which multiple satellites are interconnected and act as nodes in a Kubernetes cluster. Finally, some steps in the direction of using “space-native” networking protocols could be adopted, using disruption-tolerant CCSDS File Delivery Protocol (CFDP) for container image transferring instead of scp. Practical realization of these goals, however, would be significantly accelerated by enhanced documentation and more active community engagement including from ESA.

ABBREVIATIONS - ACRONYMS

NMF	NanoSat MO Framework
ESA	European Space Agency
CI	Continuous Integration
CD	Continuous Deployment/Delivery
k8s	Kubernetes
CCSDS	Consultative Committee for Space Data Systems
JVM	Java Virtual Machine

REFERENCES

[1] C. Coelho, S. Cooper, M. Merri, M. Sarkarati, and O. Koudelka, "NanoSat MO Framework: Drill down your nanosatellite's platform using CCSDS Mission Operations services," in *68th International Astronautical Congress (IAC)*, September 2017. [Online]. Available: https://www.researchgate.net/publication/320267505_NanoSat_MO_Framework_Drill_down_your_nanosatellite%27s_platform_using_CCSDS_Mission_Operations_services

[2] A. Ahmad. Monolithic vs Microservices vs SOA – Architecture Comparison Guide. Design Gurus. Accessed Nov. 25, 2025. [Online]. Available: https://www.designgurus.io/blog/monolithic-service-oriented-microservice-architecture

[3] Containerize your Java applications. Microsoft. Accessed Nov. 25, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/developer/java/containers/overview

[4] N. Thomson and S. Gerring. Java on containers: a guide to efficient deployment. Datadog. Accessed Nov. 25, 2025. [Online]. Available: https://www.datadoghq.com/blog/java-on-containers/

[5] G. Kocur. JVM and Kubernetes walk into a bar. SoftwareMill. Accessed Nov. 25, 2025. [Online]. Available: https://softwaremill.com/jvm-and-kubernetes-walk-into-a-bar/

[6] NMF services network configuration for Kubernetes. Accessed Nov. 25, 2025. [Online]. Available: https://github.com/kosmoedge/kosmoedge-docs/blob/main/docs/space-age-devops/nmf-network-howto-k8s.md

[7] Advancing the Tactical Edge with K3s and SUSE RGS. SUSE RGS. Accessed Nov. 25, 2025. [Online]. Available: https://www.rancher.com/assets/pdf/government/SUSE_RGS_and_Booz_Allen_FINAL_RGS_Version.pdf

[8] S. Mohanty. Putting Kubernetes in Space! Medium. Accessed Nov. 25, 2025. [Online]. Available: https://sanjimoh.medium.com/putting-kubernetes-in-space-e0918a3dce26

[9] The world's first cloud-native edge computing satellite cloud-edge integrated solution is verified in space. KubeEdge. Accessed Nov. 25, 2025. [Online]. Available: https://kubeedge.io/case-studies/satellite/

[10] Z. Huang, X. Hu, and Y. Bao. Incremental Deep Learning For Satellite with KubeEdge and MindSpore. KubeEdge. Accessed Nov. 25, 2025. [Online]. Available: https://kccnceu2022.sched.com/event/yuEH/keynote-incremental-deep-learning-for-satellite-with-kubeedge-and-mindspore-xiaoman-hu-community-operation-

[11] M. Chauhan. IBM Open Sources Space Situational Awareness, Kubesat Projects. TFIR. Accessed Nov. 25, 2025. [Online]. Available: https://tfir.io/ibm-open-sources-space-situational-awareness-kubesat-projects/

[12] IBM Space Tech - Github Repository. IBM. Accessed Nov. 25, 2025. [Online]. Available: https://github.com/IBM/spacetech-kubesat

[13] Akri Project - Github Repository. Akri Project. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/project-akri

[14] Akri: Making IoT Devices Accessible to Your Edge Kubernetes Clusters. Akri Project. Accessed Nov. 26, 2025. [Online]. Available: https://www.youtube.com/watch?v=mcKNistZkrY

[15] F': A Flight Software Embedded Systems Framework. NASA. Accessed Nov. 26, 2025. [Online]. Available: https://nasa.github.io/fprime/

[16] Unikernels: Rethinking Cloud Infrastructure. Accessed Nov. 26, 2025. [Online]. Available: http://unikernel.org/

[17] Unikernels - Github Repository. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/cetic/unikernels

[18] SpaceOS: The Secure Unikernel OS for Satellite Payloads. Parsimoni. Accessed Nov. 26, 2025. [Online]. Available: https://parsimoni.co/

[19] Industry's first commercial Unikernel with POSIX compatibility. Lynx Software Technologies. Accessed Nov. 26, 2025. [Online]. Available: https://www.electronicspecifier.com/industries/security/industry-s-first-commercial-unikernel-with-posix-compatibility/?utm_source=chatgpt.com

[20] T. Pfandzelter, J. Hasenburg, and D. Bermbach, "Towards a Computing Platform for the LEO Edge," in *4th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '21)*, April 2021. [Online]. Available: https://moewex.github.io/academic/publication/2021-leo/2021-leo.pdf

[21] T. Ho. What is DevOps? Principles, Life Cycle, and Best Practices. Accessed Nov. 26, 2025. [Online]. Available: https://reliasoftware.com/blog/what-is-devops

[22] DevOps Lifecycle. GeeksforGeeks. Accessed Nov. 26, 2025. [Online]. Available: https://www.geeksforgeeks.org/devops/devops-lifecycle/

[23] Kubernetes Overview. Accessed Nov. 26, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/overview/

[24] What is Kubernetes? Mirantis. Accessed Nov. 26, 2025. [Online]. Available: https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-is-kubernetes/

[25] S. Rasal. Why you should use Kubernetes?? Accessed Nov. 26, 2025. [Online]. Available: https://faun.pub/why-you-should-use-kubernetes-bf395bef52de

[26] H. Sangshetti. Kubernetes: Architecture and Components explained. Accessed Nov. 26, 2025. [Online]. Available: https://medium.com/@himanshusangshetty/kubernetes-architecture-and-components-explained-e489e98db15d

[27] J. Walker. Kubernetes Control Plane: What It Is How It Works. Accessed Nov. 26, 2025. [Online]. Available: https://spacelift.io/blog/kubernetes-control-plane

[28] J. Evans. A few things I've learned about Kubernetes. Accessed Nov. 26, 2025. [Online]. Available: https://jvns.ca/blog/2017/06/04/learning-about-kubernetes/

[29] A. Patel. Kubernetes — Architecture Overview. Accessed Nov. 26, 2025. [Online]. Available: https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11

[30] A. Gogineni, "Helm for Continuous Delivery of Serverless Applications on Kubernetes," *International Journal of Innovative Research in Engineering Multidisciplinary Physical Sciences*, vol. 9, no. 6, December 2021. [Online]. Available: https://www.ijirmps.org/papers/2021/6/232138.pdf

[31] S. Gokhale, R. Poosarla, S. Tikar, S. Gunjawate, A. Hajare, S. Deshpande, S. Gupta, and K. Karve, "Creating Helm Charts to ease deployment of Enterprise Application and its related Services in Kubernetes," in *2021 International Conference on Computing, Communication and Green Engineering (CCGE)*. IEEE, September 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9776450

[32] Docker Compose. Docker. Accessed Nov. 26, 2025. [Online]. Available: https://docs.docker.com/compose/

[33] M. Sharma. File Structure of docker-compose.yml File. Accessed Nov. 26, 2025. [Online]. Available: https://builder.aws.com/content/2qi9qQstGnCWguDzgLg1NgP8lBF/file-structure-of-docker-composeyml-file

[34] P. D. P. Kogut. Small Satellites: Technology That Democratizes Space. EOS Data Analytics. Accessed Nov. 26, 2025. [Online]. Available: https://eos.com/blog/small-satellites/

[35] S. W. Paek. Synthetic Aperture Radar. Accessed Nov. 26, 2025. [Online]. Available: https://www.researchgate.net/publication/343821902_Synthetic_Aperture_Radar

[36] What are SmallSats and CubeSats? NASA. Accessed Nov. 26, 2025. [Online]. Available: https://www.nasa.gov/what-are-smallsats-and-cubesats/

[37] M. A. Zafrane, A. A. Bouchahma, B. Abes, B. Zafrane, and M. F. Bengabou, "E-design and manufacturing approach for Cubesat solar panel deployment mechanism," *International Journal on Interactive Design and Manufacturing (IJIDeM)*, vol. 16, no. 4, March 2022. [Online]. Available: https://www.researchgate.net/publication/359254168_E-design_and_manufacturing_approach_for_Cubesat_solar_panel_deployment_mechanism

[38] Smallsats by the Numbers 2025. BryceTech. Accessed Nov. 26, 2025. [Online]. Available: https://brycetech.com/reports/report-documents/smallsats-2025/

[39] O. Liubimov, I. Turkin, V. Pavlikov, and L. Volobuyeva, "Agile Software Development Lifecycle and Containerization Technology for CubeSat Command and Data Handling Module Implementation," *MDPI Computation*, September 2023. [Online]. Available: https://www.researchgate.net/publication/373966364_Agile_Software_Development_Lifecycle_and_Containerization_Technology_for_CubeSat_Command_and_Data_Handling_Module_Implementation

[40] OPS-SAT Spacecraft. Nanosats Database. Accessed Nov. 26, 2025. [Online]. Available: https://www.nanosats.eu/sat/ops-sat

[41] Phi-Sat-2 Spacecraft. Nanosats Database. Accessed Nov. 26, 2025. [Online]. Available: https://www.nanosats.eu/sat/phi-sat-2

[42] Consultative Committee for Space Data Systems (CCSDS). Accessed Nov. 26, 2025. [Online]. Available: https://ccsds.org/

[43] M. Merri and M. Sarkarati, "Retire Legacy Technology with the CCSDS MO Services," in *14th International Conference on Space Operations*, May 2016. [Online]. Available: https://www.researchgate.net/publication/303098729_Retire_Legacy_Technology_with_the_CCSDS_MO_Services

[44] M. Merri, "Cheaper, Faster, and Better Missions with the CCSDS SMC Mission Operations Framework," in *AIAA SPACE 2009 Conference Exposition*, June 2012. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2009-6718

[45] C. Coelho, "A Software Framework for Nanosatellites based on CCSDS Mission Operations Services with Reference Implementation for ESA's OPS-SAT Mission," Ph.D. dissertation, Graz University of Technology, 2017. [Online]. Available: https://github.com/esa/nanosat-mo-framework/blob/ops-sat/sdk/sdk-package/src/main/resources/docs/Dissertation__Cesar_Coelho.pdf

[46] C. Coelho, O. Koudelka, and M. Merri, "NanoSat MO framework: When OBSW turns into apps," in *IEEE Aerospace Conference 2017*, March 2017. [Online]. Available: https://www.researchgate.net/publication/316588347_NanoSat_MO_framework_When_OBSW_turns_into_apps

[47] ——, "NanoSat MO Framework: Achieving On-board Software Portability," in *SpaceOps 2016 Conference*, May 2016. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2016-2624

[48] C. Coelho, M. Merri, O. Koudelka, and M. Sarkarati, "OPS-SAT Experiments' Software Management with the NanoSat MO Framework," in *AIAA SPACE 2016*, September 2016. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2016-5301

[49] D. Evans, "OPS-SAT: Operational Concept for ESA'S First Mission Dedicated to Operational Technology," in *14th International Conference on Space Operations*, May 2016. [Online]. Available: https://www.researchgate.net/publication/303098634_OPS-SAT_Operational_Concept_for_ESA%27S_First_Mission_Dedicated_to_Operational_Technology

[50] S. C. Suteu, "Ops-sat software simulator," Master's thesis, Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering., September 2016. [Online]. Available: https://github.com/esa/nanosat-mo-framework/blob/ops-sat/sdk/sdk-package/src/main/resources/docs/Software_Simulator_Master_Thesis.pdf

[51] V. Sharma. Inter-Process Communication in Operating Systems: A Comprehensive Guide with Real-life Examples and Code. Accessed Nov. 26, 2025. [Online]. Available: https://medium.com/@the_daft_introvert/inter-process-communication-in-operating-systems-a-comprehensive-guide-with-real-life-examples-and-c508cf3bfb1a

[52] Edge computing. Accessed Nov. 26, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Edge_computing

[53] L. Bernstein. How Edge Computing Is Changing Space from Terminal to Satellite. Accessed Nov. 26, 2025. [Online]. Available: https://www.kratosspace.com/constellations/articles/how-edge-computing-is-changing-space-from-terminal-to-satellite

[54] k3s: Lightweight Kubernetes. Accessed Nov. 26, 2025. [Online]. Available: https://k3s.io/

[55] What Is Jenkins and How Does it Work? Intro and Tutorial. Accessed Nov. 26, 2025. [Online]. Available: https://codefresh.io/learn/jenkins/

[56] microk8s: The effortless Kubernetes. Accessed Nov. 26, 2025. [Online]. Available: https://microk8s.io/

[57] P.-S. Feng. Setting up Jenkins on MicroK8s. Accessed Nov. 26, 2025. [Online]. Available: https://dev.to/psfeng/setting-up-jenkins-on-microk8s-4b8a

[58] How Ansible works. Red Hat. Accessed Nov. 26, 2025. [Online]. Available: https://www.redhat.com/en/ansible-collaborative/how-ansible-works

[59] NanoSat MO Framework - Github Repository. ESA. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/esa/nanosat-mo-framework

[60] CCSDS MO services - ESA's Java implementation - Github Repository. ESA. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/esa/mo-services-java

[61] NanoSat MO Framework - mission tailoring for OPS-SAT - Github Repository. ESA. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/esa/nmf-mission-ops-sat

[62] CCSDS MO Wiki - Github Repository. ESA. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/esa/CCSDS_MO

[63] minikube. Accessed Nov. 26, 2025. [Online]. Available: https://minikube.sigs.k8s.io/docs/

[64] T. Panhalkar. Overview of the TCP/IP Networking Model. Accessed Nov. 26, 2025. [Online]. Available: https://info-savvy.com/overview-of-the-tcp-ip-networking-model/

[65] NodePort in Kubernetes: A Glossary Overview. Accessed Nov. 26, 2025. [Online]. Available: https://zesty.co/finops-glossary/nodeport-in-kubernetes-a-glossary-overview/

[66] Pod Security Standards. Accessed Nov. 26, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/security/pod-security-standards/

[67] NVIDIA device plugin for Kubernetes - Github Repository. Nvidia. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/NVIDIA/k8s-device-plugin

[68] Device Plugins. Kubernetes. Accessed Nov. 26, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/

[69] Kubernetes 1.26: Device Manager graduates to GA. Accessed Nov. 26, 2025. [Online]. Available: https://kubernetes.io/blog/2022/12/19/devicemanager-ga/

[70] D. Golubovic, D. Gaponcic, D. Guerra, and R. Rocha. Efficient Access to Shared GPU Resources: Part 2. Accessed Nov. 26, 2025. [Online]. Available: https://kubernetes.web.cern.ch/blog/2023/03/17/efficient-access-to-shared-gpu-resources-part-2/

[71] M. Kozorovitskiy. Kubernetes Everywhere Enables Simplified Heterogeneous Deployment: Edge, Prem, Cloud. Accessed Nov. 26, 2025. [Online]. Available: https://www.suse.com/c/rancher_blog/kubernetes-everywhere-enables-simplified-heterogeneous-deployment-edge-prem-cloud/

[72] Kubernetes for IoT: Bringing the Best of Edge and Cloud Computing. Accessed Nov. 26, 2025. [Online]. Available: https://www.i2k2.com/blog/kubernetes-for-iot-bringing-the-best-of-edge-and-cloud-computing/

[73] OPS-SAT Wiki. Accessed Nov. 26, 2025. [Online]. Available: https://en.wikipedia.org/wiki/OPS-SAT

[74] H. Koziolek and N. Eskandani, "Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift," in *ICPE '23: ACM/SPEC International Conference on Performance Engineering*, April 2023. [Online]. Available: https://programming-group.com/assets/pdf/papers/2023_Lightweight-Kubernetes-Distributions.pdf

[75] J. Walker. Containerd vs. Docker: Container Runtimes Comparison. Accessed Nov. 26, 2025. [Online]. Available: https://spacelift.io/blog/containerd-vs-docker

[76] A. Ahmed and G. Pierre, "Efficient Container Deployment in Edge Computing Platforms," 2017. [Online]. Available: https://www.researchgate.net/publication/335471071_Efficient_Container_Deployment_in_Edge_Computing_Platforms

[77] balenaCloud: code deployment to devices grouped in a fleet. Accessed Nov. 26, 2025. [Online]. Available: https://docs.balena.io/learn/deploy/deployment/

[78] Bash scp Command - Secure Copy. Accessed Nov. 26, 2025. [Online]. Available: https://www.w3schools.com/bash/bash_scp.php

[79] How To Use Rsync to Sync Local and Remote Directories. Accessed Nov. 26, 2025. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-use-rsync-to-sync-local-and-remote-directories

[80] M. C. of the Consultative Committee for Space Data Systems, "Ccsds file delivery protocol (cfdp)," CCSDS, Tech. Rep., July 2020, recommendation for Space Data System Standards. [Online]. Available: https://ccsds.org/Pubs/727x0b5e1.pdf

[81] ——, "Ccsds bundle protocol specification," CCSDS, Tech. Rep., September 2015, recommendation for Space Data System Standards. [Online]. Available: https://ccsds.org/Pubs/734x2b1.pdf

[82] Cyclone® V FPGA and SoC FPGA. Accessed Nov. 26, 2025. [Online]. Available: https://www.altera.com/products/fpga/cyclone/v

[83] D. Evans, "OPS-SAT: Designing a Mission from the Ground Upwards," in *14th International Conference on Space Operations*, May 2016. [Online]. Available: https://www.researchgate.net/publication/303098768_OPS-SAT_Designing_a_Mission_from_the_Ground_Upwards

[84] F. Schöttl and J. Gottfriedsen. Near-Real-Time Fire Detection Leveraging Edge AI in Space: Transforming Thermal Earth Observation with NVIDIA. Nvidia. Accessed Nov. 26, 2025. [Online]. Available: https://www.nvidia.com/en-us/on-demand/session/gtc25-s72067/

[85] Understanding the difference between JDK, JRE, and JVM. Accessed Nov. 26, 2025. [Online]. Available: https://www.boardinfinity.com/blog/understanding-the-difference-between-jdk-jre-and-jvm/

[86] M. Haeussler. Dockerfiles, Jib ..., what's the best way to run your Java code in Containers? Accessed Nov. 26, 2025. [Online]. Available: https://www.youtube.com/watch?v=HFhIqfKn_XI

[87] Astro Pi: a Raspberry Pi used on the ISS for student-led science and coding experiments. Accessed Nov. 26, 2025. [Online]. Available: https://astro-pi.org/

[88] T. Pultarova. SpaceX to launch 1st space-hardened Nvidia AI GPU on upcoming rideshare mission. Accessed Nov. 26, 2025. [Online]. Available: https://www.space.com/ai-nvidia-gpu-spacex-launch-transporter-11

[89] Webcam Capture API - Github Repository. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/sarxos/webcam-capture

[90] fswebcam - Small and simple webcam for *nix. Accessed Nov. 26, 2025. [Online]. Available: https://manpages.ubuntu.com/manpages/bionic/man1/fswebcam.1.html

[91] k3s - Release Notes v1.32.X. Accessed Nov. 26, 2025. [Online]. Available: https://docs.k3s.io/release-notes/v1.32.X

[92] Kubernetes Generic Device Plugin - Github Repository. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/squat/generic-device-plugin

[93] Generic device plugin DaemonSet basic manifest. Accessed Nov. 26, 2025. [Online]. Available: https://raw.githubusercontent.com/squat/generic-device-plugin/main/manifests/generic-device-plugin.yaml

[94] Ansible role to install microk8s on Ubuntu - Github Repository. Accessed Nov. 26, 2025. [Online]. Available: https://github.com/istvano/ansible_role_microk8s

[95] NMF Helm Charts - Github Repository. Accessed Nov. 28, 2025. [Online]. Available: https://github.com/kosmoedge/deployments/tree/main/helm-charts/src

[96] J. Abraham and C. Wloka, "Edge Detection for Satellite Images without Deep Networks," May 2021. [Online]. Available: https://arxiv.org/pdf/2105.12633

[97] Spring Boot. Accessed Nov. 28, 2025. [Online]. Available: https://spring.io/projects/spring-boot

[98] Quarkus - Supersonic Subatomic Java. Accessed Nov. 28, 2025. [Online]. Available: https://quarkus.io/

[99] Camera App Demo - Github Repository. Accessed Nov. 28, 2025. [Online]. Available: https://github.com/kosmoedge/camera-app-demo

[100] Edge Detection Using OpenCV. Accessed Nov. 28, 2025. [Online]. Available: https://opencv.org/blog/edge-detection-using-opencv/

[101] R. Ratnakumar and S. J. Nanda, "A low complexity hardware architecture of K-means algorithm for real-time satellite image segmentation," *Multimedia Tools and Applications*, vol. 78, no. 3, May 2019. [Online]. Available: https://researcher.manipal.edu/en/publications/a-low-complexity-hardware-architecture-of-k-means-algorithm-for-r/?utm_source=chatgpt.com

[102] L. Haidar. Sobel vs. Canny Edge Detection Techniques: Step by Step Implementation. Accessed Nov. 28, 2025. [Online]. Available: https://medium.com/@haidarlina4/sobel-vs-canny-edge-detection-techniques-step-by-step-implementation-11ae6103a56a

[103] T. Carrigan. Linux commands: exploring virtual memory with vmstat. Accessed Nov. 28, 2025. [Online]. Available: https://www.redhat.com/en/blog/linux-commands-vmstat

[104] vmstat Visualizer. Accessed Nov. 28, 2025. [Online]. Available: https://knziiy.net/vmstat-visualizer/

[105] extended vmstat Visualizer. Accessed Nov. 28, 2025. [Online]. Available: https://github.com/kosmoedge/vmstat-visualizer