Cloud Native MANO for Next Generation Mobile Networks

by

Alireza Mohammadi

a dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Sorbonne Université

Ecole Doctorale Informatique, Télécommunications et Electronique

EURECOM, Systèmes de Communication

Thesis Supervisors:

Professor Navid Nikaein,

Mister David Olivier

President of the Jury:

Professor Raymond Knopp

Reviewers:

Associated Professor Roberto Bruschi

Associated Professor Anna Tzanakaki

Examiners:

Professor Serge Fdida

Associated Professor Adrian Kliks

Professor Wolfgang Kellerer

Defense day scheduled by **13 December 2023**

MANO cloud native pour les réseaux mobiles de nouvelle génération

par

Alireza MOHAMMADI

une dissertation soumise en vue de l'obtention partielle du diplôme de

Doctor of Philosophy

à la

SORBONNE UNIVERSITÉ

Ecole Doctorale Informatique, Télécommunications et Electronique

EURECOM, Systèmes de Communication

Journée de soutenance prévue d'ici le **13 décembre 2023**

Declaration of Authorship

Hereby, I, Alireza Mohammadi, declare that this document titled, “Cloud Native MANO for Next Generation Mobile Networks”, and the work presented in it are of my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Doctor of Philosophyat this university.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this project is entirely my work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signature

Date

“Pauca sed Matura.”

[On the personal seal of Carl Friedrich Gauss]

Abstract

Over the past decade, the telecommunication industry has undergone a profound transformation, witnessing substantial shifts in the design, deployment, and operation of network infrastructure. The advent of 5G technology has set forth a new set of requirements and challenges for network operators. Predominantly, these challenges are accentuated at larger scales of the network and the encompassed data, which are the core facilitators for emergent use cases foreseen in the Beyond 5G and 6G. Despite this evolving landscape, the prime objective for network operators remains to be the same: *To optimize the cost-revenue problem*, albeit in a substantially expanded dimension. This demands a transformation from the traditional, manual network management approaches to the modern, and automated, cloud-native systems. This transformation has previously occurred with Self-Organizing Networks (SON) falling short in addressing the challenges of 5G networks [1]. Now, with the surge of cloud-native technologies into the telecommunication domain, this transformation has to be elevated further.

In the context of the cost-revenue optimization, a pivotal factor in network operation costs is **resource consumption**, whereas revenue generation is primarily influenced by sustained quality of service and business **agility** for a rapid service introduction. In the realm of cloud-native 5G and 6G, the former encompasses concepts such as cost and energy optimization, resource sharing, and network slicing, while the latter leverages a majority of DevOps practices, relatively new to the telecommunication industry. The advent of cloud computing and virtualization technologies has empowered network operators to minimize Capital and Operational Expenditures by consolidating Network Functions (NFs) onto commodity hardware. Advancements in cloud technologies have facilitated the instantiation and decommissioning of NFs, shifting network management from lengthy, manual processes to agile, automated Management and Operation (MANO) systems. This paradigm shift introduces a *proactive* network design strategy aimed at meeting service requirements, rather than *retrofitting* services to fit a pre-established network architecture. It has also redefined problem statements such as network slicing, transitioning from a resource division focus to a design approach aligned with slice requirements and available resources. These dual formulations fundamentally alter how MANO systems are designed and operated for the dynamic, software-oriented, cloud-native networks, necessitating a new generation of MANO that adheres to the cloud-native principles like **declarative reconciliation** and **consistent automation**.

On the other hand, both traditional and current MANO systems have lagged behind the rapid advancements in the IT industry, failing to fully exploit the capabilities of cloud-native environments. To bridge this gap, this thesis introduces a **specialized cloud-native MANO for next-generation mobile networks**, conceptualized under the TRIREMATICS project, which embodies cloud-native principles to redefine and revamp the MANO stack fundamentally. Our newly proposed methods significantly outperform existing solutions in all considered metrics, such as agility and overhead. Furthermore, a substantial number of *operations* become scalable only when these modern methodologies are integrated.

In the realm of research, these innovations have unlocked a new domain previously inaccessible due to the inherent limitations of traditional MANO systems. This novel area merges insights from distributed systems, cloud computing, and networking to design highly scalable and reliable systems. This thesis falls under the category of experimental system design, where each idea presented is implemented and evaluated in a real-world setting. This approach has garnered significant interest from researchers and engineers involved in various global and European projects, fostering collaboration and exploration of new opportunities enabled by this platform.

This thesis crystallizes significant contributions pivoted around integral concepts such as **Consistent DevOps** and **Declarative Automation** to realize the envisioned cloud-native MANO. These principles are applied in the context of **multi-x** systems, where 'x' represents various dimensions such as RAN vendor, OS, and cloud, addressing the level of heterogeneity and diversity in the modern networks. Interpreting multi-x as a cloud-native extension to the Open RAN ecosystem, the TRIREMATICS project is conceived and validated through a concrete proof-of-concept prototype for multi-vendor 5G networks. This addresses the complexities of next-generation private and public cloud-native MANO systems by incorporating advanced technologies such as eBPF and recent developments in the cloud-native domain, including Kubernetes.

Resumé

Au cours de la dernière décennie, l'industrie des télécommunications a subi une transformation profonde, marquée par des changements importantes dans la conception ainsi que le déploiement et l'exploitation des infrastructures réseaux. L'emergence de la technologie 5G a introduit un nouvel ensemble de défis et d'exigences pour les opérateurs de réseau. Ces défis sont particulièrement accentués à l'échelle du réseau et ses données associées, qui sont les facilitateurs principaux des nouveaux cas d'usages prévus pour les technologies 5G et 6G. Malgré cet évolution, l'objectif principal des opérateurs de réseau reste le même : *optimiser le problème coût-revenu*, bien que dans une dimension considérablement élargie. Cela nécessite une transformation des approches traditionnelles de gestion de réseau manuelle vers des systèmes modernes et automatisés, basés sur la technologie cloud. Cette transformation a déjà eu lieu avec les réseaux auto-organisants (SON), mais ces derniers se sont avérés insuffisants pour relever les défis des réseaux 5G [1]. Aujourd'hui, avec l'essor des technologies cloud natives dans le domaine des télécommunications, cette transformation est acceleré.

Dans le contexte de l'optimisation coût-revenu, un facteur clé dans les coûts d'exploitation du réseau est la **consommation des ressources**, tandis que la génération de revenus est principalement influencée par le niveau de qualité de service et **l'agilité** commerciale pour un deploiement rapide de services. Dans le domaine des 5G et 6G basées sur le cloud, le premier englobe des concepts tels que l'optimisation des coûts et de l'énergie, le partage des ressources et la découpage du réseau (en anlgais network slicing). Le second tire parti d'une majorité de pratiques DevOps, relativement nouvelles dans l'industrie des télécommunications. L'emergence du cloud computing et des technologies de virtualisation ont permis aux opérateurs de réseaux de minimiser les coûts d'investissement (CapEx) et d'exploitation (OpEx) en déployant des fonctions réseau (NFs) sur des serveurs génériques. Les avancées dans les technologies cloud ont facilité l'instanciation et la decommission des NFs, changeant ainsi la gestion manuelles et longues des réseaux de télécommunication vers un système de gestion et exploitation (MANO) automatisés et agiles. Ce changement de paradigme introduit une stratégie de conception de réseau *proactive* visant à répondre aux exigences de service, plutôt que de *rénovation* les services pour les adapter à une architecture de réseau préétablie. Il a également redéfini les énoncés de problèmes tels que le découpage du réseau, passant d'une approche axée sur la division des ressources à une approche de conception alignée sur les exigences de découpage et les ressources disponibles. Ces doubles formulations modifient fondamentalement la façon dont les systèmes MANO sont conçus et exploités pour les réseaux dynamiques, orientées logiciel, et deployé nativement dans le cloud, nécessitant une nouvelle génération de MANO qui adhère aux principes du cloud tels que la **réconciliation déclarative** et l'**automatisation cohérente**.

D'un autre côté, les systèmes MANO existants ont pris du retard par rapport aux avancées rapides dans l'industrie informatique, ne parvenant pas à exploiter pleinement les capacités des environnements cloud

natifs. Pour combler cette lacune, cette thèse introduit un **MANO spécialisé dans le cloud pour les réseaux mobiles de nouvelle génération**, conceptualisé dans le cadre du projet TRIREMATICS. Ce projet incarne les principes du cloud pour redéfinir et revitaliser fondamentalement la pile MANO. Nos nouvelles méthodes proposées surpassent nettement les solutions existantes dans toutes les métriques considérées, telles que l'agilité et la surcharge. De plus, un nombre important d'*opérations* ne deviennent évolutives que lorsque ces méthodologies modernes sont intégrées.

Ces innovations ont ouvert un nouveau domaine de recherche auparavant inaccessible en raison des limitations inhérentes aux systèmes MANO traditionnels. Ce nouveau domaine fusionne les connaissances issues des systèmes distribués, de cloud computing et des réseaux pour concevoir des systèmes hautement évolutifs et fiables. Cette thèse s'inscrit dans la catégorie de la conception de systèmes expérimentaux, où chaque idée présentée est mise en œuvre et évaluée dans un contexte réel et operationelle. Cette approche a suscité un intérêt considérable de la part des chercheurs et des ingénieurs impliqués dans divers projets mondiaux et européens, favorisant la collaboration et l'exploration de nouvelles opportunités permises par cette plateforme.

Cette thèse cristallise des contributions significatives articulées autour de concepts intégraux tels que **DevOps cohérent** et l'**automatisation déclarative** pour réaliser une pile MANO cloud natif, spécialisé pour des réseaux de télécommunications . Ces principes sont appliqués dans le contexte des systèmes **multi-x**, où « x » représente diverses dimensions telles que le fournisseur RAN, l'OS et le cloud, traitant plusieurs niveaux d'hétérogénéité et de diversité des réseaux modernes. Interprétant le multi-x comme une extension cloud natif de l'écosystème Open RAN, le projet TRIREMATICS est conçu et validé à travers un prototype concret et un pilot pour les réseaux 5G multi-fournisseurs. Cela répond parfaitement aux complexités des systèmes MANO dans le contexte des deploiements dans des clouds privés et publics en intégrant des technologies avancées telles que eBPF et des développements récents dans le domaine cloud natif, notamment Kubernetes.

Résumé du chapitre 1 : Introduction

Le chapitre 1 de la thèse établit la scene pour l'ensemble du travail académique en fournissant une introduction complète et détaillée aux domaines de recherche abordé. Ce chapitre est essentiel car il donne au lecteur une compréhension claire des objectifs, des motivations, et du contexte de la recherche ainsi que l'organisation de la manuscrit dans son ensemble.

Objectifs et Motivations

La première section du chapitre se concentre sur les objectifs et les motivations qui ont guidé la recherche. Il souligne l'importance du domaine de la télécommunication en cloud natif, et en particulier les défis et les opportunités présentés par les technologies de réseau de cinquième et sixième générations (5G et 6G). L'objectif principal est d'améliorer les systèmes existants et d'apporter des innovations dans ces réseaux, tout en abordant les questions de évolutivité, de fiabilité et d'efficacité.

Contributions de la Recherche

Le chapitre détaille également les contributions spécifiques apportées par l'auteur dans ce domaine. Il s'agit notamment de l'élaboration de nouvelles méthodologies pour la gestion des réseaux, de l'optimisation des processus d'automatisation, et de la création de solutions plus efficaces pour la gestion des réseaux à grandes echelles. Chaque contribution est présentée avec un contexte suffisant et une justification de son importance dans le domaine global de la recherche.

Structure de la Thèse et Publications

Enfin, le chapitre se termine en présentant la structure globale de la thèse, en donnant un aperçu des chapitres suivants. Il inclut également une liste des publications académiques générées par la recherche, ajoutant ainsi une couche de validation externe au travail effectué.

Résumé du chapitre 2 : Contexte et l'état de l'art

Le chapitre 2 sert de pierre angulaire à la thèse en établissant le contexte fondamental et le cadre générique du domaine de recherche de cette thèse. Le principal objectif de ce chapitre est de fournir un panorama complet des travaux antérieurs et des concepts clés qui influencent directement ou indirectement le sujet de la thèse.

Structure et L'Architecture

Ce chapitre commence par une discussion sur la structure planétaire de 5G et les défis associés à la gestion des réseaux à grande échelle du cloud. Après avoir présenté les plans de la 5G, le chapitre se concentre sur l'architecture particulière de TRIREMATICS, qui est le projet de recherche principal de cette thèse. En course de cette introduction, nous présentons les comparaisons avec les stacks MANO existants, ETSI-NFV et O-RAN SMO, et soulignons les différences clés.

Nomenclature et Terminologie

Cette section du chapitre se concentre sur la nomenclature et la terminologie utilisées dans la thèse. Il est essentiel de définir ces termes dès le début, car ils sont utilisés tout au long de la thèse. En particulier, les termes tels que *multi-x*, *déclarative*, *idempotent*, et les autres concepts clés du cloud computing sont également définis en détail.

Evolution Chronologique et Générationnelle

Le chapitre poursuit avec une revue détaillée de l'évolution chronologique des technologies et des solutions MANO et cloud. Cette section nous permet de définir quatre générations de solutions MANO qui s'appuient sur les différentes technologies. TRIREMATICS est présenté comme un MANO de dernière génération, entièrement cloud natif et basé sur les principes DevOps et le plan des opérateurs logiciels.

Les Dimensions et les Modèles de Conception

Dans ce chapitre, nous présentons également les dimensions dans lesquelles les solutions MANO peuvent être comparées aux modèles de conception qui peuvent être utilisés pour les concevoir. De nombreux travaux précédents sont présentés dans le contexte de ces dimensions et modèles de conception, permettant au lecteur de comprendre les différences et les similitudes entre les solutions existantes et les TRIREMATICS.

Résumé du chapitre 3 : DevOps Cohérent

Chapitre 3 aborde la conception et la mise en œuvre de plateforme DevOps de TRIREMATICS. Le concept de **DevOps cohérent** est également introduit, en particulier dans le contexte des environnements **multi-x**. Un DevOps cohérent fait référence à un *pipeline* de construction, de test et de déploiement qui fournit des artefacts avec des résultats reproductibles et prévisibles dans différents environnements. En ayant à l'esprit l'agilité commerciale, un DevOps agile et cohérent serait un véritable catalyseur pour tout opérateur de réseau. De plus, le chapitre met en évidence les défis uniques que représente le DevOps dans le domaine des télécommunications, en raison de la nécessité d'une grande échelle et d'une cohérence dans des scénarios multi-x.

Preliminaries

Cette section sert d'introduction au concept de DevOps, qui regroupe un ensemble de meilleures pratiques et d'outils permettant aux organisations de construire, tester et déployer leurs applications dans le cloud d'une manière plus rapide et fiable. Les artefacts, principalement des images conteneur, sont définis de manière exhaustive, ainsi que les complexités des différents environnements d'exécution de conteneurs et formats d'image. La section critique également les malentendus et imprécisions actuels dans la recherche en télécommunications, plaidant pour une approche équilibrée entre la sécurité et la performance dans les systèmes de gestion de conteneurs, spécifiquement au sein du projet HYDRA.

Système de Construction (Build en anglais)

La section suivante du chapitre discute de la procedure entière de construction des images conteneur. Tout d'abord, le mappage des NFs sur les artefacts est présenté, accompagné des critères de sélection des mappages en fonction des mesures de temps de construction et de taille d'image. Ensuite, les recettes de construction sont définies, en mettant l'accent sur les dépendances et leurs expressions déclaratives. Après cela, la stratégie de construction est présentée et comparée à trois autres systèmes en termes de cohérence, de concurrence, et de mise en cache. La présentation de la stratégie de construction contient également une formulation mathématique du problème. Dans une partie séparée de la section, nous démontrons l'importance de la cohérence dans DevOps, tout en respectant la concurrence et la mise en cache. En outre, l'impact de la limitation des ressources sur la cohérence est évalué, en particulier dans le contexte des environnements multi-x. Enfin, la section se termine par une discussion sur l'automatisation de la construction et le système de gestion des versions.

Système de Test et CI/CD

La section suivante du chapitre se concentre sur le système de test, essentiellement la procédure de test des images conteneur dans un environnement multi-x. De plus, la section présente le système CI/CD (de l'integration en continuee au deploiement en continue), qui est responsable de l'automatisation de la procédure de construction des artefacts et de test. En particulier, nous discutons de nouvelles méthodologies pour la gestion des tests en CD pour les réseaux de la prochaine génération. Ces méthodologies sont basées sur le découpage du réseau et la gestion des ressources, et sont conçues pour répondre aux exigences de test des réseaux 5G et au-dela dans un environnement multi-x.

La Structure des Images et les APIs

Le chapitre se termine par une discussion sur la structure des images conteneur et ses APIs. La structure des images conteneurs est définie en détail, en mettant l'accent sur les différentes couches et leurs rôles. Ensuite, les APIs sont présentées, en particulier celles qui sont utilisées pour la gestion de cycle de vie et le tolérance aux pannes.

Résumé du chapitre 4 : Automatisation Déclarative

Le chapitre 4 aborde la conception et l'implémentation de la partie MANO de TRIREMATICS, qui est particulièrement responsable de l'automatisation déclarative de la gestion des NFs.

Le Plan de Opérateur Multi-x

Au début du chapitre, nous identifions les acteurs dans l'écosystème multi-x. Ensuite, nous définissons le concept de plan de opérateur, qui est composé de trois couches. Dans chaque couche, plusieurs exemples de opérateurs logicels sont proposés. Les couche basse et niveau-1 sont détailées en termes de fonctionnalités et de responsabilités dans les sections suivantes du chapitre. Egalement, nous identifions les formulations duals des problèmes de mise à l'échelle et de découpage du réseau ainsi que leur algorithmes associées, qui apparaissent dans le contexte moderne du cloud natif.

Le Gestionnaire de Side-car

Cette partie du chapitre se concentre sur le gestionnaire de side-car et la composition de Pods. Nous introduisons plusieurs nouvelles interfaces pour plusieurs opérations du jour-1 et du jour-2, y compris la (re) configuration, la résolution des dépendances, le tolérance aux pannes, l'observabilité et la gestion fine d'un NF. Nous définissons également les nouvelles mesures pour ameliorer l'efficacité energitique et la qualité de service, qui sont utilisées par les décisions mentionnées ci-dessus.

L’Evulation et les Resultats

Le chapitre se termine par une évaluation des performances du projet ATHENA. En particulier, nous comparons les performances de ATHENA avec OSM en matière de temps de déploiement, de consommation de ressources, de agilité de réponses aux événements de jour-2, et l’effet sur les performances du réseau (latence et débit). Les résultats montrent que ATHENA est surpasse considérablement OSM dans tous les aspects considérés, prouvant qu’il s’agit de la prochaine génération de MANO. Par exemple, notre solution montre une amélioration minimale de 60 % de l’agilité et une réduction moyenne de 90 % des ressources nécessaires par rapport à l’état de l’art sans sacrifier l’efficacité ou la performance.

Résumé du chapitre 5 : Cas d’Usage

Chapitre 5 présente les cas d’usage de TRIREMATICS a travers trois scénarios différents:

1. L’optimization de coûts et d’énergie dans les réseaux privés 5G;
2. La planification et la prédiction de la consommation de ressources dans les réseaux 5G Open RAN;
3. La collecte de mesures énergétiques et le cloud computing vert dans le cadre de la 6G et MANO de la prochaine generation.

Résumé du chapitre 6 : Conclusions

Le chapitre 6 conclut la thèse en résumant la vue d’ensemble du projet TRIREMATICS en form d’un problème d’optimisation global. Ensuite, les travaux futurs sont présentés.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter would incorporate the motivations and contributions made by the author in this area of research in a summarized manner. Furthermore, we introduce the structure of the thesis and the publications made by the author.

The project supporting the thesis material is called TRIREMATICS, which is a portmanteau of the words *Trireme* and *Automatics*. The Trireme is a type of ancient Greek warship that was used by the Athenians, and the Automatics is a term used in the telecommunication industry to refer to the automation of the network operations. The layered structure of TRIREMATICS is the reason behind the name while mixed with the common courtesy of naming projects relevant to the Greek or nautical terms in the cloud native landscape.

1.1 Motivation and Challenges

While 5G networks have been deployed for several years across various countries, and discussions about 6G networks are already underway in the research community, the promises of 5G networks remain *elusive* for network operators, both public and private. This issue is not extensively covered in existing literature but is evident from the author's interactions with multiple operators and companies in the telecommunication industry. We claim the primary challenge lies in the scale and complexity of 5G networks, making them difficult to manage with the current tools and technologies available to network operators. The central motivation and the overarching challenge of this thesis are to address this gap and provide a viable solution.

The first layer of this complexity arises from the convergence of various technologies in 5G and anticipated 6G networks. Such a rich tapestry of technologies emerged from various vendors and technologies increases the heterogeneity of the network, creating a multitude of layers that require *management*. Especially when one considers the long-term viability of the network in terms of the sustainability requirements for the future networks, the complexity of the network *maintenance* increases even further. Traditional management tools are often siloed and not built to handle such diversified networks, thereby leading to operational inefficiencies and increased downtimes. In this backdrop, MANOs in general and cloud-native MANOs in particular offer a harmonized and integrated approach, capable of unifying various layers of the network, to manage resources, services, and applications seamlessly. By leveraging cloud-native principles, this approach promises agility, scalability, and resilience—qualities essential for handling the immense scale and complexity inherent to 5G and 6G ecosystems.

Moreover, the commercial implications of an inefficiently managed network can be severe, impacting not just the quality of service but also the rapid rollout of new features and services, a critical requirement in the rapidly evolving telecom landscape. A cloud-native MANO solution can bring about automation and intelligence in network management processes, thereby significantly reducing operational expenses and time-to-market for new services. The ability to adapt quickly to consumer needs and market trends could mean the difference between leading the industry or playing catch-up. Therefore, the motivation to delve into this research is not merely academic; it is critically aligned with tangible, real-world challenges and opportunities that network operators are currently grappling with.

These motivational remarks raise to certain challenges that are enlisted further in the section. The challenges directly emerge from the motivations. For instance, *consistency* directly addresses the need for reliability and predictability in network operations, ensuring that the theoretical gains of 5G and 6G are actualized in real-world deployments. Meanwhile, a multidimensional sustainability underscores the imperative of long-term viability, emphasizing that the network solutions of today must be designed with environmental considerations and future scalability in mind.

The adopted methodology, as illustrated in figure 1.1, involves an experimental design and development of the problem's premises on a comprehensive realistic platform named TRIREMATICS, which later leads to the abstraction and modeling practices for accurate problem formulation and algorithm design. The primary goal of the experimental design and development aligns with the universally accepted motivation across enterprises: *financial viability*. Insights obtained from this implementation and study enable us to abstract the designs and identify the underlying mathematical models of the problem. These models span from simple graph theory to complex optimization problems and are then leveraged to explore alternative solutions not initially considered in the implementation but have emerged due to the abstraction process. Through iterative comparison and refinement, this cycle continues until an acceptable solution is identified. In an ideal scenario, these abstractions could be further dissected to discover potential solutions initially overlooked by the experimental design and development.

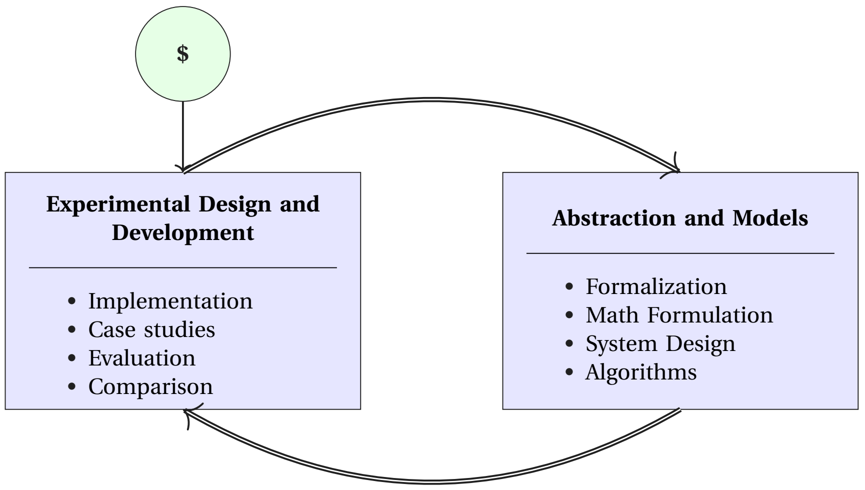


FIGURE 1.1: The study of this thesis follows the loop of experimental design and development and abstraction to identify the problem and its solution. The Dollar sign ($) represents the financial viability of the solution as the primary incentive of the optimizations.

The cost and revenue dynamics of a network are so complex that formulating a comprehensive optimization problem and solving it through mathematical methods is impractical. This fact is revisited in the chapter 6 where we try the formulation of the problem into an optimization problem, and it rapidly expands to a sophisticated non-linear optimization problem. This underscores the importance of experimental design and development practices forming an empirical analysis of the problem. On the other hand, this necessitates a system design study, not only to establish a platform for empirical research but also to implement the outcomes of the abstraction process. The following fundamental challenges are identified and addressed in this thesis using the aforementioned methodology. These challenges are also accompanied by key terms that serve as contributing responses to these challenges:

- An End-to-End (E2E) MANO platform is needed that supports the dynamic and diverse nature of modern 5G or 6G networks → **Multi-x**;
- The results produced from the deployments of such MANO should be reproducible and predictable → **Consistency**;

- The operators should be able to define their network with high-level desired outcomes instead of low-level complex configurations → **Intelligent Declarative Automation**;
- The MANO should be rapid and proactive to the events of the network lifecycle → **Agile Day-*n* Reconciliation**;
- The artifacts used in the MANO need to deliver business agility with little to no overhead → **Agile DevOps and Lightweight MANO**;
- The platform needs to be scalable and extensible to support the future needs of network operators → **Cloud Native**.
- The MANO system needs to be ready for supporting sustainability for the future networks, including observations for energy metrics, fine-grained energy-aware decisions, and support for heterogeneous *recycled* or *reused* compute and radio resources → **Multi-x Sustainability**.

1.2 Contributions

By the methodology and motivation discussed in the section 1.1, the author has made the following major contributions:

Multi-x — Originating from the extension of the Open RAN concept, multi-x offers freedom of choice at every stage of network design and implementation. Our design is not only agnostic to the hardware or software used but also provides open interfaces for extensibility and while it itself remains fungible.

Consistent and Concurrent DevOps — This becomes particularly important in the context of multi-x networks, where predictability and reproducibility are essential for minimizing supply chain risks. Consistency impacts the network's performance, security, and reliability. We have prioritized consistency as a core constraint in our design, leading to a highly concurrent yet consistent DevOps approach for multi-x 5G and 6G networks.

Intelligent, Agile, and, Declarative Automation — Leveraging distributed and cloud-native system designs, we have revolutionized how networks are designed, defined, and operated. This paradigm shift has unlocked numerous benefits and opportunities that were previously unattainable. For instance, declarative automation enables agile and fault-tolerant reconciliation, structured observability, optimized and intelligent decision-making, new problem formulations, and, ultimately, novel use cases. The network lifecycle management is an active process that evolves over time. The agility of the response to the changes in the network while delivering observable and actionable insights enables the operators to make informed decisions and optimize the network in real-time. We show in this thesis a modern, cloud-native reconciliation approach that is agile and fault-tolerant, outperforming the existing solutions by a significant margin.

Operator Plane — A newly defined cloud-native Plane for extending the capabilities of the platform for unforeseen and complex future use cases. This plane is a structured multi-level abstraction that transcends the network concepts to cloud-native API resources.

Multi-x Sustainability — A novel approach to the sustainability of the network, including the energy metric collection and formulation, fine-grained micro-decisions and high-level macro-decisions, and support for multi-x heterogeneous clusters.

1.3 Publications

The following direct publications are made by the author to support the thesis material:

- **A. Mohammadi**, N. Nikaein; *ATHENA: An Intelligent Multi-x Cloud Native Network Operator*, IEEE Journal on the Special Areas in Communications (JSAC), Open RAN Special Issue, 2023 [2]; related to the context of the chapters 4 and 5.
- C. Chen, M. Irazabal, C. Chang, **A. Mohammadi**, N. Nikaein; *FlexApp: Flexible and low-latency xApp framework for RAN intelligent controller*, IEEE International Conference on Communications (ICC), 2023 [3]; a use case of the TRIREMATICS platform for the Open RAN systems.

1.4 Industrial Demos and Projects

Following the methodology represented in the figure 1.1, the author has been involved in several projects and has demonstrated the results of the thesis in the form of industrial demos in several events. These are a few notable mentions:

- Next Generation Mobile Networks (NGMN) International Conference and Exhibition (IC&E), Paris, France, 2022; with a demo titled Cloud Native Open RAN;
- Mobile World Congress (MWC), Barcelona, Spain, 2023; where we presented several demos including one on artifact upgrade in telco DevOps pipeline, multi-x observability, and cloud-native RAN sharing;
- Open RAN Workshop, Taipei, Taiwan, 2023; with a demo on network slicing and sharing in Open RAN as well as a talk on the cloud-native Open RAN;
- European Conference on Networks and Communications (EuCNC), in Gothenburg, Sweden, 2023; with a demo on applications of private 5G networks in the industry with TRIREMATICS as the MANO platform; part of the 5G-Victori project.

1.5 Structure

This project is the result of experimental research in the common intersection of the cloud native and telecommunication. Hence, the study method is deeply intertwined with the implementation and the results of the experiments. The main flow of the project is based on two major concepts:

- Consistent DevOps; the core concept in the Chapter 3;
- Declarative Automation; the building block of the Chapter 4.

In each chapter, first the general design is laid out, then for each particular choice in the design, supporting abstract reasoning alongside implementation considerations are provided. In chapter 3, we performed empirical quantified analysis on a group of mathematically well-defined problems that appear in a DevOps pipeline applied in the context of cloud-native 5G and 6G. The result of this analysis is a set of practices that are incorporated in our CI/CD platform. In chapter 4, we shift the focus to the MANO systems with mostly a qualitative design journey to employ the best of cloud-native technologies in the MANO systems for 5G and 6G.

The thesis is structured as the following:

- Chapter 2 provides a brief background on the cloud native technologies and discusses the related works, grouped into several generations;

- Chapter 3 discusses the challenges and novelties for HYDRA and TRIDENT projects on the axis of the consistent DevOps, alongside several evaluations of the defined approaches;
- Chapter 4 addresses ATHENA, the declarative automation of TRIREMATICS and its design and implementation details for an efficient and cloud native MANO, with a set of evaluations and comparisons to the existing MANO solutions;
- In Chapter 5, we provide several use cases and scenarios to demonstrate the capabilities of TRIREMATICS in the real world scenarios;
- Chapter 6 concludes the thesis and provides a brief outlook on the future work.

It should be noted even though this thesis by itself is not conclusive enough to capture every other aspect of the discussed concepts, it paves the path for the future research in the area of cloud-native telecom in its own influential and innovative manner.

Chapter 2

Background & Related Work

In this chapter, we lay the foundational context of the thesis through the planar structure introduced in section 2.1, which delineates the exact scope of the project. Given this framework, we explore the chronological evolution of Cloud Native[1] telecommunications and its relationship with other models and standards in section 2.4. We also examine the nomenclature in section 2.3 to establish a consistent terminology for the remainder of the thesis.

[1] The term may appear with dash as *cloud-native* or in lowercase, especially when used adjectivally. However, the official spelling according to the Cloud Native Computing Foundation (CNCF) is *Cloud Native*.

The chapter continues with a review of the dimensions along which various MANO solutions can be compared, as discussed in section 2.5. These dimensions, in conjunction with the design patterns outlined in section 2.6, serve as the basis for introducing the four generations of MANO in section 2.7.

To fully comprehend the comparison logic for the related works, one has to take into account that TRIREMATICS is tackling the underlying framework for MANO by introduction of a whole new generational shift. In that sense, we tend to examine the state-of-the-art with the same perspective, reaching out to their underlying architecture rather than the innovations they have brought on the top. One may even argue that those works could be transplanted to the new architecture with minor changes, but the contributions made there are orthogonal to the discussion in this thesis.

2.1 Planes in 5G

To simplify the understanding and development of complex architectures, it is standard practice to divide them into logical components. In telecommunications, this practice dates back to the early days of cellular networks, where the architecture was typically divided into the Radio Access Network (RAN) and Core Network (CN). This division serves to segregate the physical components of the network and administrative domains. When visualizing a generic

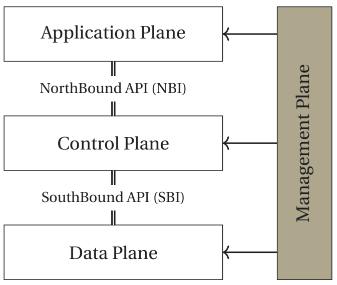


FIGURE 2.1: SDN's three planes and their interfaces. Alternative terminologies exist for SDN that do not apply to the context of this thesis, see [4]. The Management Plane, defined out of the scope for the SDN, is shown here too.

mobile network horizontally, as is commonly done in textbooks, this division appears to bisect the diagram vertically.

With the advent of Software Defined Networking (SDN) in the early 2010s, the concept of dividing a network into horizontal planes gained prominence. Specifically, SDN specifies three distinct planes as illustrated in figure 2.1: the Application Plane, Control Plane, and Data Plane. These planes differ in their roles within the network, whether directly involved in packet processing (Data Plane), configuring the processing (Control Plane), or dictating the general traffic behavior (Application Plane) [4].

The innovation in SDN's planar architecture lies in its ability to enhance programmability, automation, and flexibility, while simultaneously improving network performance and monitoring capabilities. The performance gains are a direct result of decoupling control operations from the data path. SDN has had a transformative impact on various forms of networking, including cloud infrastructures and, subsequently, LTE and 5G networks [5], [6].

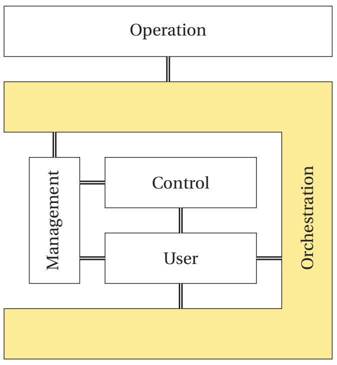


FIGURE 2.2: In our design, the 5G Planes could be roughly pictured like this figure. The Orchestration Plane wraps around the other Planes to provide infrastructure access (south), synchronization (east), and the communication medium between the Management and Operator (north).

Although the terms *plane* and *layer* are often used interchangeably, we distinguish between them using specific formal definitions that will be consistently applied throughout this thesis. The following subsections will explore the planar structure of 5G, adhering to the terminology established here.

Definition 2.1 (Plane)**.** *A plane refers to the horizontal logical partitioning of a system into distinct components based on their influence on data packets within a network or their structural impact on other system elements.*

We identify five distinct Planes in this thesis, as illustrated in figure 2.2: User, Control, Management, Orchestration, and Operation. Subsequent sections will elaborate on each Plane, detailing their internal structures and *Layers* with specific examples from 5G networks where applicable. The particularly new challenges and requirements of each Plane will also be discussed in the context of cloud-native 5G and 6G networks.

Definition 2.2 (Layer)**.** *A layer constitutes a subdivision of a Plane (definition 2.1), grouped by either its functional attributes or the interfaces it consumes or provides.*

Contrary to Planes, there is no standardized definition of Layers within 5G networks; we employ the term solely for categorization purposes in this thesis. While some researchers have proposed additional Planes such as the Application Plane, Intelligence Plane, Cognitive Plane, and Analytics Plane, we omit these from our discussion as they fall outside the scope of this thesis and do not present unique challenges in a cloud-native context.

2.1.1 User Plane

3GPP adopts the concept of Control and User Plane Separation (CUPS) [7] from the SDN architecture.[2] The entire 5G network can be viewed as a complex packet processing system that receives packets from the User Equipment (UE), processes them, and forwards them to the destination on the Data Network (DN). The components of the 5G network directly involved in packet processing, constituting the User Plane (UP), are as follows:

[2] The terms User Plane (UP) and Data Plane (DP) are interchangeably synonymous in the context of 5G.

- Distributed Unit (DU) in the RAN;
- Central Unit User Plane (CU-UP) in the RAN;
- User Plane Function (UPF) in the CN;
- Data Network (DN);

In monolithic RAN architectures, the RAN can be considered as a whole entity in both the User Plane and Control Plane. In LTE, the Serving Gateway (SGW) and the Packet Gateway (PGW) serve the role of the UPF. It is noteworthy that the O-RAN definition of Radio Unit (RU) combined with O-RAN's DU aligns with the 3GPP DU definition. Figure 2.3 shows the 5G UP with its interfaces. The primary challenges for the User Plane in a cloud-native environment include:

1. Minimizing the disparity between nominal and actual throughput on UP links;
2. Reducing the number of hops in packet transmission;
3. Mitigating unwanted latency overhead particular to the containerized and cloud-native settings;
4. Ensuring the requisite level of reliability;
5. Implementing mechanisms to enhance security and privacy;
6. Dynamic service mapping for observability and troubleshooting.

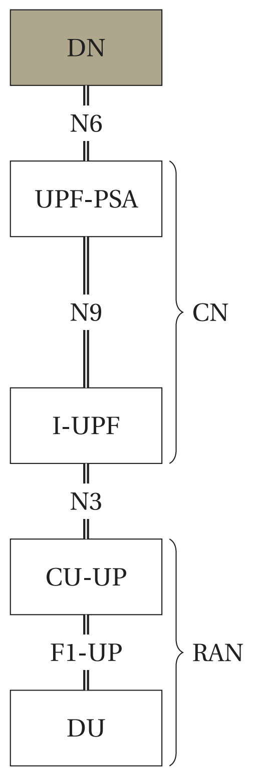


FIGURE 2.3: 5G User Plane with their interfaces. In this figure, we have also shown the decompostion of the UPF into Intermediate UPF and UPF PDU Session Anchor (PSA). In that sense, we tried to depict that I-UPF is closer to the RAN than the UPF PSA.

These challenges are largely agnostic to the underlying hardware and are more influenced by the software stack. In TRIREMATICS, these challenges are primarily addressed through the judicious selection of cloud-native tools in the design of the cloud infrastructure. For example, incorporating Border Gateway Protocol (BGP) in TRIREMATICS allows for improved performance due to lower overhead and no need for encapsulation, or the application of eXpress Data Path (XDP) to reduce the number of hops in packet transmission from container to container. Majority of these challenges are addressed in the section 4.7.

2.1.2 Control Plane

In the 5G network, the Control Plane (CP) is responsible for establishing and configuring data sessions between the User Equipment (UE) and the Data Network (DN). The CP primarily consists of the following components. The Layers in this list follow the definition 2.2.

- Central Unit Control Plane (CU-CP) in the RAN;
- Network Layer, which includes the Access and Mobility Management Function (AMF) and Session Management Function (SMF);
- Slicing Layer, featuring the Network Slice Selection Function (NSSF);
- Exporting Layer, comprising the Network Exposure Function (NEF);
- Data Layer, containing the Unified Data Management (UDM) and Unified Data Repository (UDR).

The introduction of Network Data Analytics Function (NWDAF) [8] as well as Management Data Analytics Function (MDAF) [9] by 3GPP has extended the CP to include a Data Analytics Layer. The O-RAN Alliance has introduced counterparts to NWDAF and MDAF through the RAN Intelligent Controller (RIC) framework [10]. In this thesis, we consider both the RIC and the Data Analytics Layer as components of the CP, albeit strictly limited to the control domain and not extending into the management domain. Our model intentionally excludes MDAF and the FCAPS functionalities of the RIC from its CP, as our Management and Operation (MAO) design pattern already encompasses these functionalities. Incorporating tasks outside their designated Planes could not only blur the separation of concerns but also potentially introduce system conflicts.

In TRIREMATICS, we adopt an *inbound-outbound* metric separation for each Network Function (NF).[3]

[3] In this thesis, RAN components are also referred to as NFs.

Definition 2.3 (Inbound Metric)**.** *A metric that reflects specific measurements about the internal state of an NF. Inbound metrics are collected by the CP, either through the Analytics Layer or the RIC, and are exposed via standard interfaces. These metrics are termed inbound as they are specific to individual NFs and require specialized invasive data extraction methods.*

Definition 2.4 (Outbound Metric)**.** *A metric that provides a general external measurement of an NF, treating the NF as a black-box application. Outbound metrics are collected by the MAO and offer an external perspective on the NFs, typically capturing generic metrics like CPU usage or energy consumption. As* *a general rule of thumb, these metrics are generally accessible from the kernel or the container runtime and are not specific to the NF itself.*

Figure 2.4 elaborates on the concept of inbound and outbound metrics to clarify these definitions. Regardless of the source, the approach of TRIREMATICS is to aggregate all metrics into a unified *data lake* for each beneficiary, subject to their access rights. See [11] for the formal definition of a data lake and its benefits.

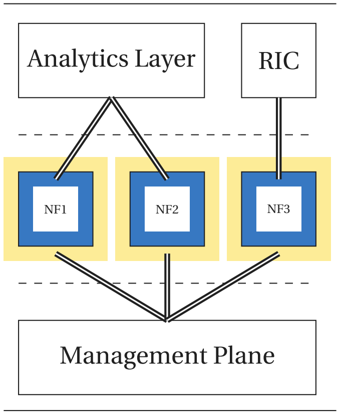


FIGURE 2.4: This figure shows the outside boundary of an NF in blue and the inside boundary in yellow. The APIs and metrics provided for the internal configuration are consumed by the CP and outer ones are consumed by the Management Plane. The black line shows where an NF is not regarded any more as an individual and becomes a generic application to be managed.

The main challenges for the CP in a cloud-native environment are:

1. Supporting the wide range of NFs and the deployment options;
2. Slicing, scoping, migration, and isolation of the NFs;
3. Collection of the analytics from the RAN and the CN in a unified manner as of the other cloud metrics;
4. Access control policies in the Service Based Architecture (SBA) [12] of the 5G CN.

Unlike the UP, where most of the challenges are addressed by the proper selection of cloud-native tools, an intricate design of OAM is required to provide the required functionalities for the CP. For example, even though slicing is majorly a concept in the CP, the OAM is ultimately responsible for the instantiation of the NFs for each slice and guaranteeing the isolation between them. For that matter, even though this thesis is neither about the UP nor the CP, paying attention to their requirements in the OAM domain is what makes the defined tasks in the other Layers possible. Some relevant discussions to these challenges are made in the sections 4.5, 4.6.4, and 4.8.1.

2.1.3 Management Plane

Beyond the UP and CP, the Plane definitions are with respect to the defined components in the UP and CP, not the stream of data. In that sense, the UP and CP components are seen as an application entity that has a **lifecycle** and the Management Plane (MP) is responsible for managing this lifecycle. The lifecycle management includes not only the start and stop of the components, but also the configuration, observability, and maintenance.

Definition 2.5 (Lifecycle)**.** *Any abstract construct in a system design that involves certain manageable states and transitions between them has a lifecycle. In that sense, a lifecycle is equivalent to a finite state machine, where a certain special null state ϕ is as the starting state representing the non-existence of the* *construct. Since naturally, a component could go back to the ϕ state at any time by a deletion operation, the term cycle appears in the name.*

As described in the table 2.1, each of the Planes partakes with respect to certain constructs and under certain domains. The UP is concerned with *data* packets, flows, and streams, and it mostly operates on a per-slice basis. On the other hand, the CP takes care of the *actions* made on the UP and works in each scope of administration. In TRIREMATICS, one could define both slices and scopes to separately manage the UPs and CPs. A scope defines the permeter of which the NFs could access and communicate with each other which might include multiple slices. The MP takes care of the *lifecycle* of the UP and CP components on a per-Workload basis. Inside a Workload (as defined later in the section 4.6), multiple NFs could be present and configured at the same time as NF is a construct belonging to the lower Planes. The Orchestrator comes to the picture when there is a need for *coordination*, whether it be the synchronization or infrastructure access. The Orchestrator takes care of Cluster constructs such as Pods and Nodes. Each Cluster itself needs a separate orchestration. Finally, the Operator is responsible for the *organization* and *abstraction* of the network End-to-End, perhaps spanning over multiple Clusters and sites.

Plane	**Constructs**	**Domain**
User	Data	Slice
Control	Action	Scope
Management	Lifecycle	Workload
Orchestration	Cluster Resources	Site
Operator	Network Abstractions	End-to-End

TABLE 2.1: The Planes and their domains and constructs. The constructs are the main entities that the Plane operates on.

The Management Plane (MP) in a cloud-native environment faces several critical challenges:

1. Ensuring agility and responsiveness in executing management actions;
2. Accommodating a diverse set of configuration and observability models, including legacy systems;
3. Managing dependencies effectively in a distributed setting;
4. Implementing agile yet reliable fault-tolerance and distributed recovery mechanisms, aligned with the dependency graph;
5. Establishing a framework for fine-grained management decisions;

6. Precisely gathering outbound metrics from Network Functions (NFs), with a focus on compute utilization, cost-efficiency, and energy consumption.

These challenges serve as the motivating factors for the design of ATHENA, as elaborated in chapter 4. In particular, the section 4.6 discusses the design of the MP in detail, while the section 4.9.1 provides the results of our experiments on the agility of the design. Section 4.6.2 discusses the configuration management in ATHENA and section 4.6.1 puts forward the dependency resolution mechanism. We study the effects of fault-tolerance and recovery mechanisms in section 4.6.3, metrics collection in section 4.6.4, and micro-decisions for the fine-grained management in section 4.8.3.

2.1.4 Orchestration Plane

The Orchestration Plane serves as a mediator that facilitates resource allocation and communication between the Operator Plane (OP) and the other Planes. In traditional parlance, this function is often encapsulated within the Management and Orchestration (MAO or MANO) frameworks. However, in modern cloud-native environments, Kubernetes has largely assumed this role, extending beyond its original scope as a container orchestrator to manage the entire Orchestration Plane. In multi-site or multi-cluster network architectures, the Orchestration Plane is defined on a per-cluster basis.

An Orchestration Plane design faces several challenges in a cloud-native setting:

1. Efficient and secure discovery of diverse hardware resources, making them addressable by the OP and consumable by the UP and CP;
2. Time synchronization across all hardware and software components;
3. Reliable data transport mechanisms for configuration and observability between the OP and MP;
4. Automated handling of networking and storage requirements for other Planes;
5. Implementation of robust isolation and security protocols;
6. Compatibility with various infrastructure types, including public clouds and on-premise bare-metal systems[4];
7. Scalability and reliability under high-load conditions.

[4] The term *bare-metal* would essentially refer to non-virtualized infrastructure. In the public clouds, only virtual machines can be instantiated, making any bare-metal cloud inherently a private, on-premise solution.

As depicted in figure 2.2, the Orchestration Plane envelops the rest of the mentioned Planes, providing essential functionalities. In simpler terms, it *sandwiches* the other Planes, both at the top and the bottom. The challenges mentioned here are mostly addressed in Kubernetes itself and inherently any solution that is built native to it, including TRIREMATICS. However, certain enhancements are needed to support the variety of telco hardware which are discussed in the section 4.7. For the time synchronization, one could rely on the Precise Time Protocol (PTP) [13] on the network cards which is supported by a simple DaemonSet in Kubernetes.

2.1.5 Operator Plane

The Operator Plane (OP) serves as an overarching Plane built upon the Management Plane (MP) and the Orchestration Plane. It initializes the MP with the system's desired state and continually updates it based on changes made by network operators.[5]

[5] In this thesis, the term *Operator* with a capital 'O' specifically refers to software entities within the Operator Plane, while the lowercase form is used exclusively to discuss human network operators.

The OP offers network abstractions that allow network operators to interact with the system using *intents* and *policies*, rather than low-level configurations. These abstractions facilitate the creation of higher-level constructs like network slices and services, some of which may span End-to-End (E2E) across multiple clusters.

The OP is responsible for scheduling and deploying all required components based on a given network description. While the OP issues commands, the actual deployment of *containers* is carried out by the Orchestration Plane. The OP oversees the lifecycle of not just individual Workloads but also the entire network and its higher-level abstractions.

The primary challenges facing the OP in a cloud-native environment include:

1. Exposing suitable abstractions for building arbitrary higher-level network constructs;
2. Efficiently scheduling and deploying Workloads and Managers across multiple clusters through the Orchestrator;
3. Managing the lifecycle of the network and its abstracted higher-level constructs;
4. Support a diverse range of network topologies, vendors, and hardware;
5. Implementing network policies and intents;
6. Providing the necessary APIs for both human network operators and AI systems to interact with the network.

Based on these challenges, we have introduced ATHENA Operator Plane in the section 4.2, where we discuss how this system of Operators could be structurally extended for various, perhaps unforeseen, use cases. For scheduling, ATHENA acts passively and simply exposes the original APIs from Kubernetes to the network operators. The maturity of these APIs is enough to address any requirements expected from a cloud-native 5G and 6G perspective. However, similar scheduling concepts are incorporated for the slices in the section 4.4.2. This concept is based on how ATHENA as an example enables slices to be an abstract higher-level network construct.

2.2 Architecture

To address the complexities and challenges outlined in the preceding sections, the TRIREMATICS project is divided into multiple subprojects.

ATHENA focuses on implementing the Operator Plane (OP) and Management Plane (MP), offering new abstractions for network, terminal, and slice operations. It employs *declarative* automation to tackle the challenges associated with the MP and OP.

ODIN serves as the intelligent control subsystem for both the RAN and CN, incorporating xApps and rApps as defined by the O-RAN specifications. This subproject is not the focus of this thesis.

GAIA augments Kubernetes with necessary plugins and extensions to form a robust Orchestration Plane. This includes device plugins, networking and storage configurations, time synchronization, and provisioning. Since the design of GAIA involves mostly implementation discussions, GAIA is only briefly discussed in this thesis.

HYDRA and TRIDENT handle the project's artifacts, including their construction, testing, and distribution through Continuous Integration and Continuous Delivery (CI/CD) pipelines. These subprojects also implement DevOps and GitOps practices to ensure artifact *consistency*.

This section mainly explores how each of these subprojects align with established standards such as European Telecommunications Standards Institute (ETSI)-Network Function Virtualization (NFV) and O-RAN Service Management and Orchestration (SMO).

2.2.1 Relation to ETSI-NFV

ETSI-NFV formally defines the Management and Orchestration (MANO) architecture, as depicted in figure 2.5, with four key components:

1. VNF Manager (VNFM): Manages the lifecycle of Virtual Network Functions (VNFs).
2. Virtualized Infrastructure Manager (VIM): Oversees the Network Function Virtualization Infrastructure (NFVI) virtualized resources.
3. NFV Orchestrator (NFVO): Coordinates the overall management and orchestration of the NFVI.
4. WAN Infrastructure Manager (WIM): Extends ETSI-NFV with SDN capabilities.

Open Source MANO (OSM) [14] implements the NFVO and VNFM components of the MANO architecture. It also provides structures for Element Manager (EM) and VNF definition, packaging, and onboarding. OSM further offers Slice Management (SM) functionalities, collectively termed as End-to-End (E2E) Network Service Orchestrator (NSO) in OSM parlance. These are depicted in the figure 2.5.

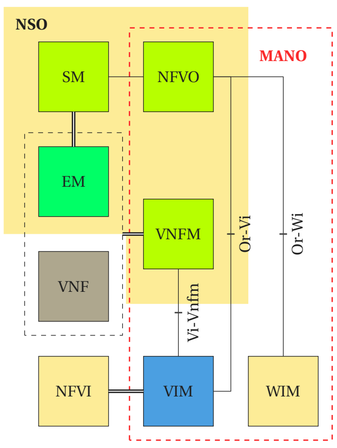


FIGURE 2.5: This figure shows the ETSI-NFV MANO architecture and how OSM is placed in it.

In its latest release (Release THIRTEEN at the time of this writing), OSM supports OpenStack and Kubernetes as preferred VIMs and employs Juju charms for EMs. The NFVI can be implemented using OpenStack or LXD containers, provisioned by Juju. Newer releases also allow VNFs to be containerized using Docker, thus qualifying as Container Network Functions (CNFs).

OSM adopts cloud-native terminology to define three lifecycle phases for network services:

1. Day-0: Network service creation.
2. Day-1: Network service instantiation and initial deployment.
3. Day-2: Network service assurance, monitoring, and maintenance.

In the Day-0, one could design and create the network service packages and onboard them to the OSM, then use them in Day-1 to instantiate the network services.

If one aims to align the ETSI-NFV MANO architecture with TRIREMATICS, the GAIA project would correspond to the VIM and NFVI components. This is because GAIA manages infrastructure provisioning and resource exposure. Kubernetes complements this by filling the remaining part of the VIM, thereby completing our Orchestration Plane. ATHENA serves as the NSO, focusing on

Day-1 and Day-2 operations for Cloud-Native Network Functions (CNFs).[6] ODIN supplements the Slice Management (SM) functionalities and adds capabilities not covered by the ETSI-NFV model. HYDRA and TRIDENT handle Day-0 operations, with the latter also responsible for CI/CD and artifact automation.

[6] The 'C' in the term CNF could mean either cloud-native or containerized, depending on the context. Where the NFs are simply packaged as containers without much regard to the cloud-native principles, they are simply referred to as the containerized. While the OSM is more about containerized workloads, in TRIREMATICS the NFs are treated as *truly* cloud-native constructs.

In TRIREMATICS, the lifecycle *days* are redefined as follows:

- **Day-0**: Artifact creation, planning, and resource definition.
- **Day-1**: Scheduling, deployment, and configuration.
- **Day-2**: Healing, reconfiguration, upgrades, and observability.

All these phases are automated, secure, consistent, repeatable, and green. Given that a process container starts directly with its primary process, lacking an internal init system like a VM, Day-2 operations for a cloud-native CNF do not involve procedures like installation or restart.

In terms of project roles, ATHENA performs tasks similar to an NSO for Day-1 and Day-2 operations concerning CNFs. GAIA, in collaboration with the Kubernetes ecosystem, takes care of the remaining structure. ATHENA is primarily compared with the OSM as a reference solution in chapter 4, given the similarity in their roles. Projects HYDRA and TRIDENT mainly focus on Day-0 operations.

2.2.2 Relation to O-RAN Architecture

The O-RAN SMO architecture is bifurcated into two primary components: the Maintenance and Operation (MAO) and the Non-Realtime RAN Intelligent Controller (Non-RT RIC). Figure 2.6 recreates a diagram from [15] that bridges the O-RAN and ETSI-NFV architectures.

The Federated O-Cloud Orchestration and Management (FOCOM) parallels the Network Function Orchestration (NFO) in its role of managing and orchestrating the O-Cloud. According to [15], FOCOM handles inventory and alarm management for the O-Cloud, while NFO oversees lifecycle, alarm, and performance management for NF deployments within the O-Cloud.

The VIM and the VNFM are subsumed into the Deployment Management Service (DMS). A similar component, termed the Infrastructure Management Service (IMS), is designated for managing O-Cloud infrastructure elements, including DMS instances and hardware accelerators. In the context of ETSI-NFV, these hardware accelerators are considered compute resources. They serve to offload radio processing tasks, either partially, as in look-aside acceleration for encoding and decoding, or entirely, as in inline acceleration for full physical layer processing.

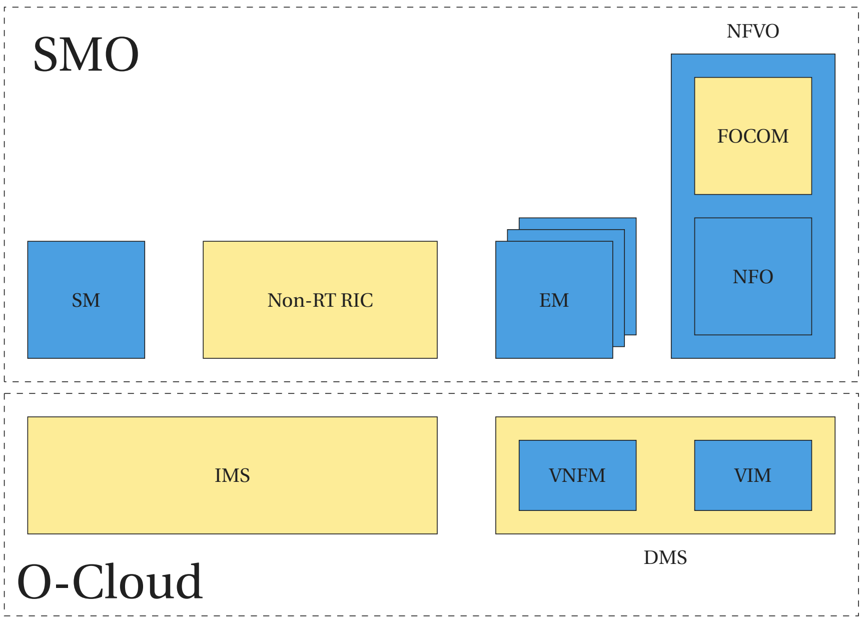


FIGURE 2.6: This figure shows the O-RAN SMO and O-Cloud, recreation of a work by [15]. The figure also shows the components carried from ETSI-NFV in blue and the new components in yellow.

Similar to the mapping presented for ETSI-NFV, one could map the TRIREMATICS to the O-RAN SMO. It should be noted that the mappings are not exact: they only consider how functionally similar are the components in question, not how exactly they are implemented or which interfaces they provide. In most of the O-RAN SMO implementations, they incorporate ETSI-NFV based tools such as OSM or Open Network Automation Platform (ONAP) [16] to provide the required functionalities.

2.3 Nomenclature

This section elaborates on the new terms and concepts introduced in this thesis, as well as existing terms from the cloud-native ecosystem that may be unfamiliar to telecommunications researchers.

2.3.1 Cloud Native Terms

Cloud Native Computing (CNC) is emerging as a pivotal technology in an era where the focus is shifting from operating *individual* computing entities to performing *intelligent* tasks on *large datasets*. The Cloud Native Computing Foundation (CNCF) defines **Cloud Native**[*] as an approach that emphasizes scalability and dynamic environments. It identifies containers, service meshes, microservices, immutable infrastructure, and declarative APIs as key enablers. The ultimate goal of a cloud-native system according to this definition is to create loosely coupled, resilient, manageable, and observable systems that can undergo frequent, high-impact changes with minimal toil.

In this thesis, we explore how TRIREMATICS enables 5G networks to become *truly* cloud-native[7].

[7] We aim to quantify these qualities for an objective comparison between established practices and the innovations introduced in TRIREMATICS

Five cloud-native adjectives frequently appear in this paper, listed as the following: declarative, idempotent, immutable, stateless, and consistent.

Definition 2.6 (Declarative)**.** *A **declarative** system is completely defined by its desired state, not the steps to reach that state. This contrasts with imperative systems, which are defined by the steps to achieve a particular state.*

Declarative systems align with the concept of Open World Assumption (OWA), which considers the system inherently incomplete and interprets the absence of information as an unknown state rather than falsity.

Definition 2.7 (Idempotency)**.** *An **idempotent** operation can be applied multiple times without changing the result beyond the initial application. Mathematically, an idempotent function f satisfies $f(f(x)) = f(x)$, meaning every value in the range of f is a fixed point for f.*

Idempotent operations facilitate the implementation of declarative systems in distributed environments by allowing the same operation(s) to be continuously applied to reach the desired state.

An **immutable** infrastructure is replaced rather than updated, enhancing reliability and resource scheduling. A **stateless** application does not maintain internal state, allowing for horizontal scaling and seamless replacement. It is noteworthy that in some scenarios, the internal state is hidden and non-obvious. For example, any application that relies on the state of a TCP or SCTP connection contains an implicit internal state. The same applies for the case of context stored in attached devices. That is why it is very unlikely to have a truly stateless application in the telco domain outside of the SBA design of the 5G CN. **Consistency** refers to reproducible and predictable outcomes across different environments, a concept further elaborated in chapter 3.

[*] `https://github.com/cncf/toc/blob/master/DEFINITION.md`

2.3.2 Multi-x

The telecommunications environment is a complex and diverse amalgamation of applications and use cases. With the advent of 5G, the list of specialized yet disparate target markets is expanding. Concurrently, network operators are increasingly inclined towards open architectures to avoid vendor lock-in. This dual demand for business application heterogeneity and system interoperability motivates the adoption of **multi-x** frameworks, where ‘x’ can represent vendor, version, node, distribution, runtime, cloud, or instance.

As elaborated in [17], the essence of multi-vendor deployments lies in the seamless interoperation of network functions from different vendors, rather than merely having multiple vendor-specific stacks. While standard interfaces enforce some level of interoperation, particularly at the UP and partially at the CP, they often leave the configuration, management, and monitoring aspects to vendor-specific implementations.

The rise of various containerization technologies, each addressing different performance, security, and isolation concerns, introduces another dimension of multi-x: multi-container technologies. This allows for the selective use of containerization technologies based on internal policies, business contracts, or specific technological advantages. Advanced cloud-native tools like Kubernetes have largely addressed other aspects of multi-x, such as multi-node and multi-cloud, but these solutions are often dispersed and require specialized treatment to meet the unique demands of the telecommunications sector.

2.3.3 Kubernetes Terms

A **container** in this context refers to a Linux process container, which is an isolated environment for running processes, managed by Linux namespaces and cgroups. A more detailed definition is given in the definition 3.2 in section 3.1. **Kubernetes** has emerged as the standard *de facto* for orchestrating these containers across a **cluster** of machines, known as **nodes**. The smallest unit of deployment in Kubernetes is a **Pod**, which is a collection of one or more containers sharing network and storage resources. A **Service** in Kubernetes serves as a software load balancer, providing a stable network interface to a set of **Endpoints**.[8] A **ConfigMap** is a Kubernetes object that holds configuration data that can be consumed by Pods. A **Secret** is a Kubernetes object that holds sensitive data such as passwords and API keys in base64 encoding. The Secrets are mostly useful for binary data, while the ConfigMaps are more suitable for text-based configuration.

[8] In literature, Kubernetes Services are sometimes confused with other types of services such as an application service (in broad sense) or 5G SBA services. None of these are related to Kubernetes Services.

Kubernetes design patterns often utilize two key components: **Controllers** and **Resources**. A Controller continuously monitors the state of Resources and ensures the cluster state aligns with the desired state. These Resources are essentially sets of OpenAPI endpoints in the Kubernetes API server. One could extend the list of the resources by adding a new Custom Resource Definition (CRD) to the cluster which defines a Custom Resource (CR) and is controlled by a Custom Controller (CC). Another common design pattern is the **Sidecar** container, which runs alongside the main container in a Pod to extend or enhance its functionality. In the context of TRIREMATICS, the main container is referred to as a **Workload**, which encapsulates the primary application logic along with auxiliary tools and APIs.

2.3.4 Operator Pattern

ATHENA bases its foundation upon the Operator Framework [18] to assist or replace the human in the loop, approaching an intelligent, ultra-dynamic, and flexible automation. Operator Pattern, as the foundation of the Operator Framework, defines Declarative Operators (DOs) as opposed to the imperative Event-Driven (ED) MANOs. The former method focuses on matching the desired and observed state via idempotent actions without extracting specifications of the trigger or maintaining a state machine [18], but the latter adds listeners to specific events and assumes the state is kept consistent between the operations. In distributed systems, however, a consistent and highly-available state machine is not a safe assumption [19]. When it comes to deployments with Kubernetes, defining state machines not only involves substantial overhead, but also means the pre-existing cloud native tools need to be reinvented, especially on lifecycle management and resource control.

The Operator Pattern and inherently follows up the fundamental control logic principles from Kubernetes.[†] In summary, this relatively novel class of the control logics for the Operators relies on *level-based* designs rather than the *edge-triggered* ones that are common with the EDs. On the other hand, these control systems assume *Open World* conditions, meaning they acknowledge the presence of uncertainty, variability, and unknown factors that can affect the system's behavior. The declarative nature of the Operators in ATHENA is the manifestation of these principles. What, in particular, ATHENA offers on top of that is to adaption and porting of the mentioned designs to the specific

[†] `https://github.com/kubernetes/design-proposals-archive/blob/main/architecture/principles.md`

field of telco by introducing the concept of the OP. This Plane allows a structural and semantic extension of ATHENA for future use cases and scenarios.

2.4 Chronology

For the purposes of this thesis, we examine the concurrent evolution of both the telecommunications industry and cloud-native technologies. In recent years, marked notably by the advent of 5G, the telecommunications sector has undergone several transformative shifts that have fundamentally changed the way radio networks are deployed. Four key trends can be identified in this transition towards a 5G-centric design philosophy, applicable across various system components including the RAN:

- Decomposition of larger systems into smaller, more manageable components, either through disaggregation or microservices architecture;
- Generalization of deployment strategies, facilitated by advancements in SDN, virtualization, or cloud-native technologies;
- A shift in focus from underlying network infrastructure to higher-level abstractions, such as services;
- Clear delineation between different functional planes, including data, control, management, and orchestration.

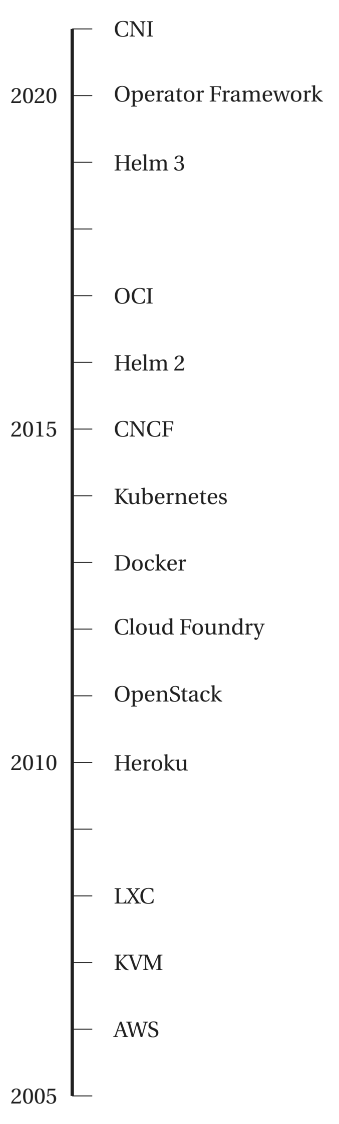


FIGURE 2.7: Timeline of the main events in the cloud native ecosystem. Amazon Web Services(AWS) was the first cloud provider. Kernel-based Virtual Machine (KVM) was the first opensource hypervisor for VMs. Linux Containers (LXC) was the first open-source container runtime. Open Containers Initiative (OCI) released their first stable specification in 2017 and the first stable version of Container Network Interface (CNI) specification was in 2021.

In parallel, virtualization technologies have made significant inroads into the telecommunications domain, offering hardware-agnostic, isolated environments. These technologies operate at various levels, be it hardware-level solutions like Intel VTx, kernel-level implementations like the Linux Kernel-based Virtual Machine (KVM), or operating system-level platforms like the Quick Emulator (QEMU).

Three driving forces continue to propel these transformative changes in the telecommunications sector:

- Financial incentives, primarily through the gradual reduction of the total cost of ownership;
- Enhanced elasticity, enabling quicker adoption of emerging technologies and novel use cases;
- The emergence of diverse markets that defy a one-size-fits-all deployment model.

Despite the rapid pace of these transitions, the momentum shows no signs of slowing, largely because these driving factors remain relevant.

Automation has proven to be an axis for growth and revenue across various industries, and telecommunications is no exception. To capitalize on this, the industry has pivoted from hardware-centric networks to software-defined NFs. These software-based NFs were subsequently encapsulated in virtualized environments, allowing for deployment on general-purpose hardware. This shift laid the groundwork for the ETSI to establish the NFV standard [20], offering a unified framework for NF vendors. This standard was later augmented with SDN to create a fully software-defined network ecosystem. Solutions like OSM [14] and ONAP [16] were introduced to automate the lifecycle management of NFs within this framework. Following the advent of OSM and ONAP, research focus shifted towards higher-level operations such as multi-clustering [21], service onboarding and management [22], and multi-domain orchestration [23], leaving the core MANO architecture largely untouched except for minor adaptations to accommodate CNFs.

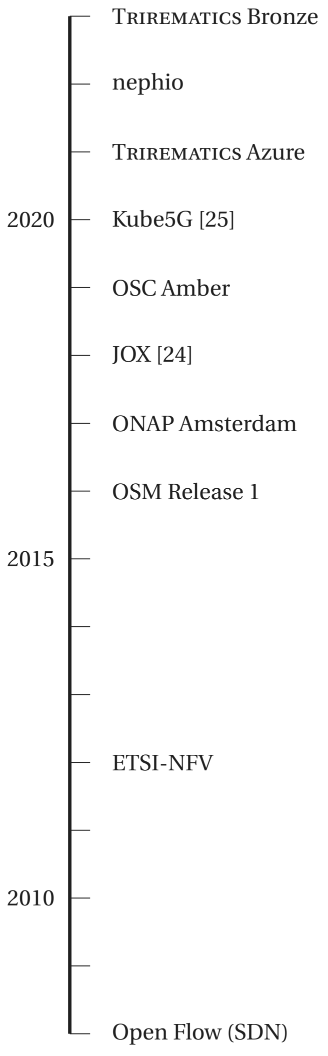


FIGURE 2.8: Timeline of the major release events in the MANO. Nephio was introduced in April 2022, and its first release was in July 2023. The first release of TRIREMATICS was introduced in February 2022, and its second major release is due in October 2023.

The CNFs were the telco industry's response to the cloud-native revolution in the IT sector. Prior to CNFs, the industry's engagement with cloud technologies was confined to the virtualization and cloudification of NFs. While virtualization allowed NFs to be deployed on general-purpose hardware, cloudification transformed NFs into flexible resource pools that could be dynamically allocated and deallocated. Initially perceived as lightweight alternatives to VMs, containers have been repositioned by the cloud-native paradigm as foundational building blocks for cloud systems. As a result, cloud-native design patterns and technologies have evolved with containers at their core. As demonstrated in studies such as [26], simply incorporating Kubernetes as a virtual infrastructure manager or naively converting VMs into containers falls short of realizing the full potential of cloud-native capabilities. Furthermore, dismissing the differences between the containers and VMs, in terms of lifecycle, runtime behavior, and management has led to inefficient and ineffective solutions.

Concurrently, the rapid pace of technological evolution has rendered much of the pre-2015 research on cloud-native and MANO (coinciding with Kubernetes' rise to prominence) either irrelevant, inadequate, or obsolete, yet being referenced in recent works such as [27], [28]. This has resulted in a fragmented and often misleading terminology and taxonomy, exacerbating misconceptions about cloud-native technologies. As the research community continues to evolve, these inconsistencies become increasingly glaring, often due to the irregular intakes of cloud-native principles and technologies scatterd over time.

Approaching cloud native telco in 3GPP standardization of 5G CN has mainly led to Helm-based solutions such as OpenAirInterface (OAI) Helm charts[‡] or Robin Smart Helm.[§] Even though the standard aims for dynamicity, the resulting designs and implementations are of static nature, even for simple matters such as IP address assignment and resolution. This ignorance impels these setups to rely on human operators while they continue to flounder at a large scale. Such malpractices are often justified as minor engineering issues, but they are in fact a reflection of carrying over a common practice from either legacy Physical NFs (PNFs) or comparably outdated design architectures with little to none regard to the nature of the cloud native and cloud deployments.

Several designs and architectures were born in the last few years, with 5G or as a part of it, especially for the RAN. Open RAN, private networking, network slicing, and sustainable and green networking are a few noteworthy examples. Various communities have been formed around these topics, and O-RAN Alliance to date remains one of the most prominent ones, with active participation from major vendors and operators. The O-RAN Alliance iteration of MANO, revived as Operation and Maintenance (OAM) [29], builds upon the same disputable amalgamation of ETSI-NFV without addressing any of the issues discussed in this thesis. In fact, all the existing O-RAN stacks use either OSM or ONAP as their OAM, including but not limited to the O-RAN Software Community (OSC) version of the O-RAN SMO. The core concepts are often renamed and recycled, proving them to be valid recurring ideas, but they are not manifested properly in the modern context of cloud native telco.

2.5 Comparison Dimensions

Before delving into this section, it's crucial to underscore that an effective MANO solution should not only address the challenges outlined in section 2.1 but also offer added value beyond existing IT solutions. Container orchestrators, by their nature, are not designed to handle tasks specific to MANOs, such as telco-optimized operations, semantic understanding of the network, or specialized NF lifecycle management. Given this criterion, Kubernetes and its various distributions that solve generic problems should not be rebranded as MANO solutions. This also excludes solutions like OpenShift[¶] and Rancher[‖] from the competition.

[‡] `https://gitlab.eurecom.fr/oai/cn5g/oai-cn5g-fed`

[§] `https://robin.io`

[¶] `https://www.redhat.com/en/technologies/cloud-computing/openshift`

[‖] `https://rancher.com/`

Since most leading container orchestrators are either Kubernetes variants or built upon it, they could serve as the VIM in an ETSI-MANO architecture or as the foundation for the TRIREMATICS Orchestration Plane. TRIREMATICS is designed to be chiefly agnostic to the underlying orchestrator, focusing instead on providing the missing MANO functionalities.

OS	**MANO**
Linux	Kubernetes
Ubuntu	OpenShift
APT	Helm
Systemd	TRIREMATICS
Daemons	Containers

TABLE 2.2: This table shows the analogy between the Linux OS and the MANO solutions. Linux is the foundation for a lot of modern servers, Kubernetes is the foundation for the modern MANOs. Ubuntu is a Linux distribution, and likewise, OpenShift is a Kubernetes distribution. APT is a package manager for Linux, and Helm plays the same role for Kubernetes. The services in Ubuntu are distributed as daemons that are installed with APT and managed by Systemd. Analogously, the NFs are distributed as containers that are perhaps installed with Helm and managed by TRIREMATICS.

In line with this argument, we should emphasize a MANO solution must offer more than just the installation or instantiation of NFs, tasks that can arguably be accomplished with a few lines of script atop an existing container orchestrator. Therefore, Helm-based solutions like those offered by OpenAirInterface (OAI) Software Alliance (OSA) or Robin (Smart) Helm charts do not qualify as proper MANO solutions. Helm serves as an effective yet straightforward package manager for Kubernetes, providing a packaging format for Kubernetes manifests to simplify their distribution and installation. However, it lacks the specialized intelligence and features one would expect from a MANO solution. In this context, Kubernetes can be likened to an Operating System (OS), and Helm to its package manager. As such, this thesis does not include comparisons between MANO solutions like TRIREMATICS and Kubernetes distributions or enterprise or individual Helm charts. Table 2.2 elaborates more on this analogy.

MANO solutions can be evaluated along various dimensions, some of which are qualitative and dependent on specific use cases or user preferences, while others are quantitative and measurable. Often, even quantitative dimensions are conflated with marketing terms, rendering comparisons highly subjective. However, there are certain universally positive characteristics for a MANO solution. In this section, we aim to outline a list of dimensions that can serve as a basis for comparing different MANO solutions.

Dimension	**Cost**
Flexibility	Hardware
Openness	Software
Resilience	Operation
Lightweight	Energy
Intelligent	Human

TABLE 2.3: This table shows for an example set of features, how they have emerged from actual costs.

Like any business, network operators aim to continuously optimize their networks to minimize *costs* and maximize *revenue*. These two motifs influence all other dimensions, whether qualitative or quantitative, directly or indirectly. Examples of such relationships are illustrated in table 2.3. Typically, costs are categorized into Capital Expenditure (CapEx) and Operational Expenditure (OpEx). CapEx encompasses the initial investment costs, such as hardware, software licenses, and installation, while OpEx covers ongoing operational costs like maintenance, energy, and human resources. The drive to reduce CapEx has led operators to consider transitioning to public clouds, utilizing commodity hardware, and adopting open-source software. This trend has given rise to a plethora of software claiming to be *carrier-grade*, asserting readiness for production networks while maintaining openness and simplicity.

Qualitative dimensions often stem from requirements set forth by network operators and expected from MANO solution providers. Many of these requirements are inspired by the experiences of enterprises in the IT industry and have yet to be rigorously defined within the context of network operations.

Time-to-Market (TtM) is a crucial metric that measures the time required to introduce a new service to the market. Adopting cloud-native technologies, particularly those aligned with Development and Operations (DevOps) practices, can significantly reduce TtM. However, this reduction is most pronounced when operators employ modern constructs like CNFs. Without such modernization, the choice of MANO solution will have a limited impact on TtM, as illustrated in figure 2.9 using arbitrary values.

TRIREMATICS enables DevOps by its Container Development Kit (CDK), offering NF developers the possibility to create multi-x artifacts that can be immediately and automatically integrated into the TRIREMATICS ecosystem. This approach allows developers to concentrate on the logic of the NFs rather than the intricacies of packaging and deployment. Additionally, TRIREMATICS includes built-in CI/CD automation, streamlining the build and test processes. As elaborated in chapter 3, these DevOps practices are specifically tailored to meet the unique requirements of the telco industry, rather than being a direct adoption of existing IT practices.

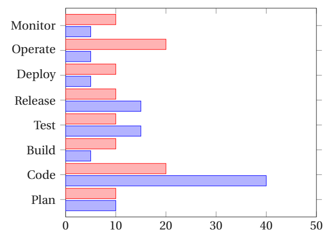


FIGURE 2.9: A rough time breakdown of the DevOps phases for the case of non-cloud native (pink) and the agile DevOps (blue). The times are in percentage of TtM. Looking at this bar chart shows how an agile DevOps transforms the dominance of the TtM to the coding and testing phases and reshapes the distribution. In either case, the majority of the TtM is spent on coding and testing: Using CNFs is the only remedy out of this.

Minimizing **vendor lock-in** is an important strategic objective for network operators, since it enables them to negotiate more competitive prices from vendors. Figure 2.10 illustrates the transition from a vendor lock-in state to a multi-x deployment. While intermediate states offer some cost advantages, achieving a multi-x state is the ultimate goal for maximizing cost-efficiency.

When multi-x strategies are combined with DevOps practices, a variety of CDK platforms can be employed, as summarized in table 2.4. However, HYDRA offers a comprehensive solution tailored specifically for telco use cases. Moreover, the structured Planes in TRIREMATICS make the components *fungible*. This means that components can be easily replaced with equivalent solutions from other vendors, further mitigating the risk of vendor lock-in.

The design of TRIREMATICS is primarily geared towards Private 5G use cases, a

Solution	**Domain**	**DevOps**
Red Hat CDK	Environment	`dC--`
JFrog Artifactory	App build and hub	`-c-P`
CNCF Buildpacks	App build model	`dC-P`
CDF Shipwright	Container build description	`-CeP`
HYDRA	Consistent artifacts methodology	`DCEP`

TABLE 2.4: Comparison between the different CDK platforms. The data in the table is encoded for fitting in the page. The letters D, C, E, and P stand for the Design, Creation, Evaluation, and Publication of the artifacts, where the dash (-) means the platform does not provide the functionality while the lowercase letter defines a rather partial support. None of the solutions other than HYDRA fully supports multi-x for the telco use cases.

domain where the ecosystem is still in its nascent stages. As a result, the transition from PNFs or VNFs to CNFs is not a primary concern. This allows private operators to directly adopt CNFs without the need for transitional steps. This focus on CNFs is a strategic choice, aimed at meeting the specific needs of emerging Private 5G networks.

Numerous industrial efforts, such as those by Rakuten[**] and Telenor[††], aim to address the challenges of the multi-vendor 5G services in cloud-native environments. However, none have fully achieved the multi-x state as depicted in figure 2.10. The authors in [30] elaborated on supporting the cloud native vision for Openstack, while supporting NFs from different vendors. The authors in [31] propose a cloud-native solution for scaling the 4G Mobility Management Entity (MME) for handling the control signaling overhead from RAN. Another work [32] presents a cross-domain slice orchestration, where the main focus is on allowing transparent interoperability between different domains, while each domain uses a distinct orchestration solution. In [33], 5G network slicing in the cloud native environments is considered without any attention on multi-x aspects. The proposed cloud native 5G service delivery platform in [34] fails to support the heterogeneity required in 5G cloud native environments too. The works in [35] propose a cloud-native 5G service platform for orchestrating the deployment in cloud-native environment, without a specific focus on multi-x. All the above-mentioned works lack of a concrete design and prototype for multi-x containerization, and do not address properly the level of heterogeneity demanded in a telco-grade cloud environment.

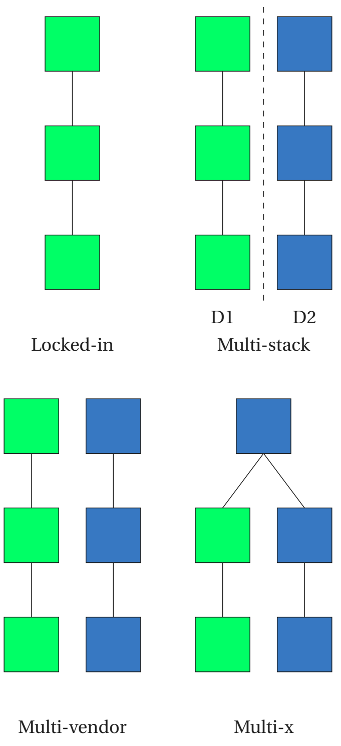


FIGURE 2.10: This figure shows the evolution from the single-vendor locked-in scenario to a multi-x solution. It is important to note that the multi-stack case is often mistaken with the multi-vendor case. In a multi-stack scenario, over multiple deployments, you realize multiple vendors. However, in the multi-vendor case, the operators could have a single MANO stack simultaneously managing multiple vendors. Multi-x takes this one step further and allows the mix-and-matching of the vendors to give an ultimate flexibility to the network operators.

[**] `https://symphony.rakuten.com/products/open-ran-5g`

[††] `https://techblog.comsoc.org/2021/04/20/telenor-trial-of-multi-vendor-5g\-standalone-sa-core-network-on-vendor-neutral-platform/`

In terms of **automation**, a range of qualitative dimensions exist, but the most crucial are *declarative design* and *active reconciliation*. These concepts are relatively new to the telco industry and serve as modern replacements for older constructs like *zero-touch* or *intent-based* systems. An intent-based system aims to understand the operator's intent and translate it into actual configurations. This nebulous concept is more straightforwardly realized through a declarative design. Moreover, a declarative approach enables more intelligent automation, potentially leading to faster convergence compared to imperative designs. Therefore, the realization of a true zero-touch system, where network operators need not intervene, is most feasibly achieved through a declarative design.

Self-Healing, **Fault Tolerance**, and **Agility** are three interrelated terms that TRIREMATICS integrates into its automation framework. A self-healing system is of little use if it is not agile and fault tolerance is essential to make self-healing less frequent. These concepts are elaborated upon in sections 4.6.3 and 4.9.1.

In TRIREMATICS, we insist on the deployments that are *reusable*, *repeatable*, and *reproducible*. These attributes ensure that deployments remain *consistent* and *predictable*, irrespective of the vendor or environment, across multiple iterations. Consistency is well studied in section 3.2.4 with a use case on predictability in the section 5.2.

Lastly, **Scalability** is a critical quantitative dimension for network operators. Unlike traditional IT services, where scaling often involves simple replication behind a load balancer, telco services present unique challenges due to their stateful nature as defined by 5G standards. TRIREMATICS addresses this by offering *slicing* as an alternative scaling mechanism. By combining slicing with replication, TRIREMATICS provides a consistent and compliant scaling solution for 5G networks, as discussed in section 4.5.

2.6 Design Patterns

TRIREMATICS employs five major design patterns that distinguish it from existing solutions:

1. Vendor-neutral DO pattern (see section 2.7);
2. Sidecar management pattern (see section 2.7);
3. Definition of first-class citizens from the telco domain;

4. Exposure of telco devices as addressable and securely assignable resources;
5. Multi-planar augmented analytics and unified actions.

In its OP, Athena treats network constructs like NFs, slices, and terminals as first-class citizens. A first-class citizen construct enjoys separate lifecycle management, instead of being a hidden subcomponent. Unlike other solutions such as OSM, where slices are merely collections of Network Services (NSs), Athena allows for independent definition and lifecycle management of slices. These slices can then be *assigned* and *scheduled* to various NFs, including the RAN, and can be scaled as needed. This results in a simplified, yet powerful, slicing mechanism.

The conventional understanding has long held that there is a trade-off between isolation (and by extension, security) and performance. This belief has been perpetuated by the inefficiencies of virtualization technologies and misconceptions about containers. Trirematics challenges this notion by offering a design that maintains high performance while ensuring robust isolation and security. This is achieved through device plugins in the Orchestration Plane, which automatically detect and advertise device resources like radio units and network terminals. As a result, the containers in Trirematics are unprivileged. A privileged container does not respect *any* known security barriers on the system because it would be executed as a root process, but this bad habit is practiced everywhere on deployment of RAN workloads, including the Helm charts, Docker Compose, or Operator-based deployments ranging over all the providers. Virtualizing the execution environment would help to mitigate to some extent, but it significantly lowers the performance which is not desirable. These resources can then be requested by containers from a resource pool, eliminating the need for root privileges and enhancing both isolation and security, particularly in public cloud deployments.

The analytics and actions in Trirematics are quite diverse and unique. We consider the analytics from multiple Planes and combine them to provide a unified view of the network. The right choice of technology for each Plane is used to provide the best performance and scalability. For the DP and CP, we use the xApps that extract the inbound metrics and send them right away to the data lake, while for the rest of the Planes, we use the Prometheus that continuously scrapes the outbound metrics and stores them in the same data lake. The E2E metrics are calculated at the OP including the energy and cost footprints with their respective breakdowns to the level of UEs, slices, and

NFs. This allows for coordinated and unified actions to be taken across the Planes with low granularity while enabling advanced business intelligence.

Athena considering energy efficiency for decision-making while being *light-weight*. A minimal overhead is a necessity for a green MANO. However, the state-of-the-art is focused either on algorithmic perspectives [36], [37] or on physical To elaborate on the benefits of CNFs compared to the VNFs, one could mention the work in [38] where the authors have pointed our merely *assuming* NFV is green by design may not be true in all cases. Athena not only induces minimal overhead, but also provides extra means to optimize and save energy.

One notable observation about the metrics is the nature of their definition. It is common to define improper and useless metrics that do not reflect the actual state of the NFs. For example, knowing how many successful network attachments have been done in the AMF is not as useful as knowing how many active subscribers are currently using that instance of the AMF. Wherever needed, we have defined the rather useful metrics that are not necessarily available in the existing MANOs.

2.7 Generations

Looking into the architectural evolution of the MANOs as well as the advancement of the cloud native technologies, we have identified four generations of MANOs, as shown in table 2.5. Since the second generation, the support for the CNFs has been added to the MANOs and is becoming the dominant type of the NFs. Trirematics orients itself around supporting CNFs that are packaged with the particular wireframe provided in Hydra enabling multi-x artifacts compatible with the cloud-native models [39], with improved lifecycle in terms of agility, scalability, greenness, downtime, and cost. This alignment is in favor of the Private 5G use cases that this thesis is concerned with. Even though the initial definition of the VNFs was not necessarily bound to the VMs but rather the virtualization of the hardware requirements for the NFs on top of general-purpose hardware, the VNFs are now mostly associated with the VMs. This connection predominantly has slowed down the adaption of the CNFs in the solutions with VNF favoritism like OSM [14] and ONAP [16] and eventually has made them over-complicated. O-RAN's specifications on SMO [40] portray a similar issue, and in several cases, the defined interfaces and services are inapplicable or obsolete for the CNFs. Examples of such services are those concerned with changes in the network interfaces. In common cases, the containers do not have the permissions to modify the network interfaces, and they are indeed configured externally via the network plugins in the orchestrator.

Leaving the VMs aside not only lowers the complexity of the architecture but also allows for a true cloud-native solution. It is worth noting that due to the hardware virtualization in the VMs, they show lower performance compared to the containers. Furthermore, since in private 5G there is little to none infrastructure in place, the network operators could directly benefit from a fully CNF-based environment without an interim VNF adaption step.

One noticeable pattern after the release of OSM [14] and ONAP [16] is that the research community turned its emphasis to higher level operations such as multi-clustering [21] or service onboarding and management [22], ignoring the MANO itself with otherwise adding support for Kubernetes or CNF workloads that never made their way into seminal works. Thus, we have grouped all the OSM-based solutions under the same group in table 2.5 as they do not exhibit significant variations in the scope of MANO itself.

Foundation	**Examples**	**Idiosyncrasies**
OpenStack	Open Baton	VNF-centric, ED Triggers
Juju	OSM, JOX [24]	ED Sidecar charms
Operators	Kube5G [25]	CRDs, Application DOs
Operators	TRIREMATICS, Nephio	Vendor-neutral Logical DOs

TABLE 2.5: This table introduces four generations in order of their appearance. Each generation is mounted on top of a foundation from the cloud native ecosystem while offering a different idiosyncrasy. Examples are given for each generation. The last two generations are built on top of the same foundation, though offer significantly different idiosyncrasies.

ATHENA as the OAM component of TRIREMATICS, realizes the concept of Operators differently from those of the two well-known Operator Patterns, i.e., Redhat Operator Framework[*] and Canonical Charmed Operators[†]. These patterns are intended to be generic, allowing for onboarding of any application. In their design pattern, usually each application from each vendor is associated with its own Operator, in contrast with the vendor-neutral approach of ATHENA. This is the difference between simply incorporating the Operator Pattern and making it native to the design of 5G and 6G MANOs. ATHENA provides Operators for logical entities that are formulating concepts such as network terminals, functions, or slices. This implementation fits our

[*] `https://operatorframework.io/what`

[†] `https://ubuntu.com/engage/collection-of-charmed-operators-whitepaper`

definition of the OP in the section 2.1.5 and allows vendor-neutral mix-and-matches which are otherwise impossible or troublesome to achieve if each vendor is providing its own Operator. Solutions like Kube5G [25], [41] suffer from the exact same issue as the IT Operator Patterns.

TRIREMATICS approach to intelligence is also different from the other solutions. When it comes to building intelligence on top of the MANO, Trirematics as a whole is the enabler and provider for various types of intelligence, including the business intelligence, traffic optimization, energy and cost efficiency, and allocation and assignment algorithms. This is achieved by ATHENA providing the declarative definition and operation of the networks. TRIREMATICS also provides the required data and analytics for the intelligence through its multi-plane architecture. The data is collected from several Planes as efficiently as possible and stored in a *label-based*, descriptive *data lake*.

In the MP, ATHENA Manager is onboarded in each pod as a sidecar container, similar in concept to the sidecar containers in Kubernetes [39] and ETSI-NFV Element Manager (EM) [14]. However, the sidecar pattern in Kubernetes [39] is barely used for management purposes in terms of controlling the lifecycle of an internal application and ETSI EM [14] performs arbitrary operations, some of which are out of the scope of our Manager (e.g., billing), or they are related to VM environments (e.g., installation commands). O-RAN has recently adapted the sidecar pattern for synchronization services [42]. While they have recognized the values of the pattern, it seems its application remains limited to the synchronization. ATHENA Manager performs agile sub-lifecycle operations, observability proxying, configuration management, and dependency resolution. It should be noted that unlike network proxy sidecars, the sidecar model for the Manager in ATHENA is not abolished by the advent and adaption of the extended Berkeley Packet Filter (eBPF) and Xpress Data Plane (XDP) [43] technologies. However, if the functionalities of the Manager are implemented otherwise, it could be discarded for a particular NF. ATHENA remains agnostic to the way the Manager functionalities are provided, and indeed, the Manager is intended to aid rather traditional and non-cloud-native applications to be aligned with the cloud-native paradigms.

O-RAN SMO uses the FCAPS model to define the functional requirements of the MANO. FCAPS is a well-known acronym for Fault, Configuration, Accounting, Performance, and Security in the context of networking and has been around for decades. ATHENA approaches the FCAPS differently than O-RAN SMO. Various levels of abstraction (Operation, Management, and Orchestration) could be involved in each of the FCAPS functionalities and the means to achieve them are based on cloud native principles applicable only to CNFs. Same as the ETSI-NFV, FCAPS is simply too outdated to properly be applied in a cloud-native context.

Chapter 3

Consistent Telco DevOps

In this chapter, we dig deeper into the design and implementation of the HYDRA and TRIDENT projects while delving into the concept of the *consistent DevOps* in the context of multi-x environments. These projects together form the DevOps platform of TRIREMATICS, where HYDRA defines the *artifacts* and their attributes while TRIDENT focuses on the automation and delivery of the said artifacts. After a review on some of the most important concepts related to the containers and DevOps in the section 3.1, we discuss creation, building, testing, and releasing the artifacts with their associated results.

The cloud-native 5G and 6G network designs rely on the software implementations of the NFs, dominantly packaged as containers into the CNFs. These containers like any other modern software artifacts could enjoy from the DevOps best practices to improve the development, testing, and deployment processes. Most importantly, with having the business agility in mind, an agile and consistent DevOps would be an absolute game-changer for any network operator. Furthermore, any 5G network would have a dozen of heterogeneous NFs from different vendors, raising the scales of the DevOps technologies to their limits. As summarized later in the table 3.4, a network operator might need to maintain hundreds of different container images.

The extreme scale of the artifacts, the necessity of consistency in multi-x scenarios, direct influence on the business agility, and the need performance requirements of the NFs, all make the DevOps for telecommunication a completely unique challenge with respect to the IT DevOps. With these challenges in mind, we meticulously journey through the whole DevOps ecosystem to formally define each of its stages and components, and find the most proper choices with respect to the 5G and 6G systems. Throughout this journey, we harvest the best practices established in the IT and make them native and specialized for the telco. For example, in the section 3.4.2 we set forth the differences between CD in IT and CD in telco and how a merger between known 5G protocols and concepts with the DevOps practices could lead to substantial improvements in the business agility.

This chapter goes through three phases of Build, Test, and Release from the DevOps cycle displayed in the figure 3.1, in consequent sections, prepended with the preliminaries to understand the concepts and appended with the resulted structure of the container which would be deployed, Operated, and monitored in the chapter 4. At each stage of DevOps, there are certain challenges to support consistency in multi-x environments while minimizing the overhead and maximizing the agility via concurrent automation. These challenges are in terms of design questions that are responded with empirical analysis over multiple solutions, one of which is the novel solution proposed in this thesis. The table 3.1 summarizes the novelty of each section in this chapter, while the table 3.2 shows which challenges are addressed in each section in terms of the affected metrics or qualities.

Section	**Novel Designs or Algorithms**
3.2.1	Image Set NFs to artifact mapping
3.2.2	Declarative build recipes
3.2.3	Consistent TRIDENT pipelines
3.2.3	Telco multi-x container image caching
3.2.5	Styx GitRegOps versioning and automation
3.3.3	Multi-x integration tests
3.4.2	Slice and Dice deployment strategy
3.4.2	Handover Rolling deployment strategy
3.5.2	Continuous probing and fault detection

TABLE 3.1: The novelty of each section in the chapter 3 with respect to the state of the art.

TRIDENT provides a general framework for the DevOps automation and delivery of the artifacts that could be used by any vendor or integrator to provide artifacts in a multi-x environment. Of course, to protect intellectual properties, some vendors may prefer to limit the multi-x dimension to not incorporate the vendors, yet still the other dimensions of multi-x are applicable and make the discussion of this chapter relevant. Nevertheless, HYDRA provides a containerization skeleton that remains valid across multi-x dimensions, allowing even the artifacts built by different vendors to be interoperable and consistent.

Section	Addressed Metrics, Qualities, or Designs
3.1	Security
3.2.1	Build time, Build resources, Image size
3.2.2	Declarative build description
3.2.3	Consistency, Concurrency factor, Caching gains
3.2.4	Consistency, Concurrency factor with limited resources, Security
3.2.5	Build automation and checkpointing
3.2.6	Cloud-native build automation
3.3.1	Artifact policies
3.3.2	NF testing SDK, Security
3.3.3	Integration E2E testing
3.4.1	Time-to-market
3.4.2	Zero-down-time upgrades
3.4.3	Zero-down-time upgrades
3.5.1	Unified multi-x containers
3.5.2	Management APIs, Status probing and fault detection
3.5.3	Unified multi-x containers

TABLE 3.2: The challenges addressed in each section in the chapter 3 with respect to the metrics and qualities defined in the section 3.1 or design questions.

One of the dimensions of multi-x covers different classes of containerization and container runtimes. Even though the VMs share some similarities with the containers, HYDRA design is not necessarily compatible with the VMs. This limitation does not come as a downside, but rather as a design choice to focus on the most performant and efficient cloud-native technologies.

The discussion in this chapter, in particular in sections 3.5 is focused on single NFs and how the day-1 and day-2 operations are performed on them. The evolution of this discussion on network services formed from multiple NFs is discussed in the chapter 4.

3.1 Preliminaries

DevOps is a collection of best practices and tools that enable organizations to build, test, and release their applications faster and more reliable. There are several extensions for DevOps that might appear in a similar context, three of which are the most relevant to the scope of this thesis:

- **DevSecOps**, which also applies the security best practices to the DevOps.
- **NetOps**, the DevOps best practices in the network infrastructure.
- **GitOps**, to center Git version control as the main source of truth for all the artifacts.

When we are referring to the DevOps in TRIREMATICS, we are implicitly referring to the DevSecOps and GitOps as well, while most of the NetOps operations are defined in GAIA project. All the DevOps practices in TRIREMATICS are with the security in mind and the Git is the only source of truth for all the artifacts. The figure 3.1 shows the DevSecOps cycle in TRIREMATICS in conjunction with the GitOps and NetOps.

Most of the definitions in this chapter are formed around the term *artifact*.

Definition 3.1 (Artifact)**.** *An artifact is any deployable component of a software system, in the most generic sense. It can be a binary, a library, a container image, or a configuration file, or even a collection of other artifacts.*

Unless it is explicitly mentioned, the term artifact in this chapter would refer to the *container images*. A container image itself, in the most general case, is any package of a root filesystem with the necessary metadata, manifests, and configurations to run the software inside the container. There are several container image formats depending on the *container runtime*. For example Snap offers a *SquashFS* image format, while Docker and Podman use the *OverlayFS* image format. A container runtime is a software that is responsible for running the container image as a *container*. The container runtimes would only fork the process tree of the container image and manage the lifecycle of the container without any virtualization or instruction translation.

Defining the term container itself is not straightforward, however, all the different kinds of the containers rely on the same principles and toolchains from Linux, most importantly the concept of *namespaces*.

Definition 3.2 (Container)**.** *A container is a running instance of a container image that has been isolated from the other processes in the system using one or more Linux kernel namespaces.*

The Linux kernel exposes the system differently in each namespace to form a unique perception of the system for each of the processes in those namespaces, allowing them to share the same kernel with different perspectives. The list of the Linux kernel namespaces are the following:

- **PID** that isolates the process trees.

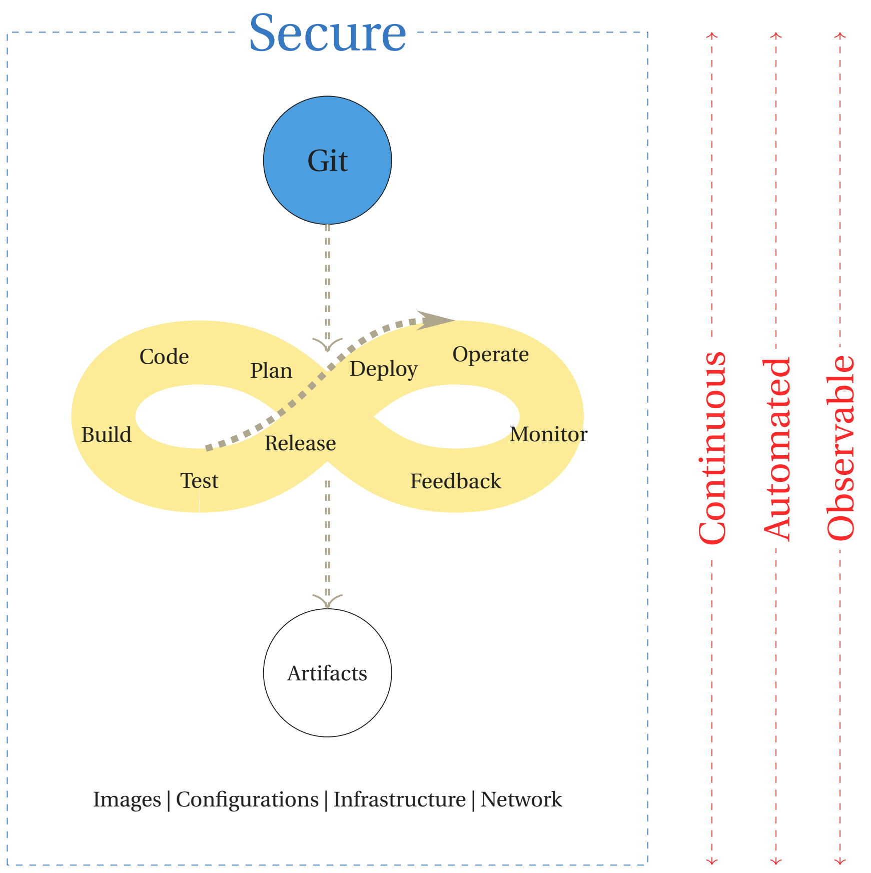


FIGURE 3.1: The DevSecOps cycle in TRIREMATICS in conjunction with the GitOps and NetOps. The figure also depicts the stages in the CI/CD pipeline in the middle. The vetical features of Continuous, Automated, and Observable apply to all the stages of the DevOps.

- **Network** that isolates the network interfaces, routing, and firewall.
- **Mount** that isolates the mount points.
- **UTS** that isolates the hostname and domain name.
- **IPC** that isolates the inter-process communication resources.
- **User** that isolates the user and group IDs.
- **Cgroup** that isolates the resource usage and control.

Cgroups are another common mechanism used in the containers to measure and limit the resource usage of the processes and their access to the system resources. Kubernetes uses the same mechanism to limit the resource usage of the Pods. Prometheus node exporter also exploits cgroups for the metrics collection.

Unlike the VMs, containers share the same kernel with the host and other containers and there is no virtualization layer between the instruction sets in the container and the host. It is crucial to understand that unlike a hypervisor, a container runtime does not directly intervene in the execution process of the container. Some exceptional container runtimes that implement sandboxed containers might have a syscall interception layer in their interface with the host kernel, that provides more security and slows down the execution process, though still there is no virtualization or instruction translation. A similar argument applies to the unikernel containers as well, where the application is compiled with all the required kernel libraries to reduce the number of system calls made to the host kernel without defining a virtualization and instruction translation layer. These huge misconceptions have significantly polluted the current research materials in telco. For example, noticeable survey works such as [44] are prone to these mistakes, leading to improper taxonomy and conclusions.

As mentioned before, there are several types of containers, but the most common groups are the following:

- **System containers**, for example LXD and Sysbox
- **Process containers**, for example containerd and CRI-O
- **Application containers**, for example Snap, Apptainer, and Flatpak

In a system container, the container runtime executes a system init process such as systemd as the first process to *imitate* a lightweight VM. However, still the kernel is shared between the host and the container(s). These container systems commonly namespace all the namespaces. In the process containers, the process tree starts with the main process in the container image and follows its lifecycle. This class also commonly namespace all the namespaces, but there are configurations in which one could share the network or the IPC namespaces to enable direct communication between the containers or host. Finally, the application containers are designed to simply package the binaries and their dependencies and normally only namespace the User and Mount namespaces. Definition of cgroups for the containers varies depending on the container runtime and the level of isolation and security.

Each container runtime provides different security features and isolation levels, such as AppArmor, SELinux, cgroups, and seccomp. These would affect the performance and the security of the container. For a moderate telecommunication workload, there needs to be a balance between the security and the performance. As such, we have chosen four container management systems in HYDRA, namely Snap, Docker, Podman, and LXD. Other stacks of containerization are either not performant enough or do not offer the required level of integration with the rest of the TRIREMATICS platform. Still the critical security practices are applied in TRIREMATICS which to the time of this writing remains unique to TRIREMATICS.

The Open Container Initiative (OCI) is a standardization effort to define the container image format, storage, runtime, and distribution. From the four container runtimes in HYDRA, only Docker and Podman are OCI-compliant. Being OCI-compliant is the only requirement for a container runtime to be supported by Kubernetes. Hence, the workloads using Snap and LXD are not running on Kubernetes environment and are designated for the bare-metal scenarios in TRIREMATICS.

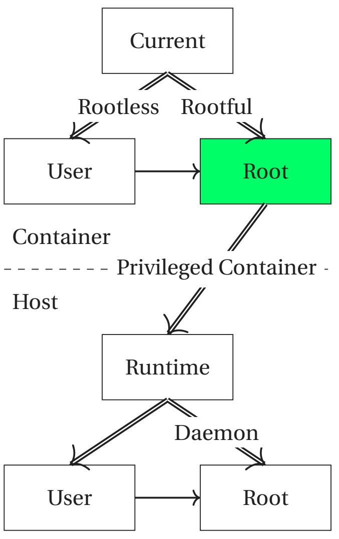


FIGURE 3.2: Each process in the container could potentially escalate itself to the level of the main process in the container that could be root or non-root. A privileged container could escalate itself to the level of its runtime user that could be root or non-root.

To clear up some terms, we briefly explain the structure of a typical (OCI-compliant) container management system. A container management system is a collection of tools for building, running, and distributing container images. The two dominant container management systems are Docker and Podman. Each container management system has a high-level and low-level runtime for the containers, where the high-level one performs all the management tasks and the low-level one is responsible for the execution of the container and communicating with the kernel or systemd. In Docker the high-level runtime is containerd and the low-level runtime is by default runc. However, containerd supports other low-level runtimes such as Kata Containers (runv), gVisor (runsc), Sysbox, and Nabla Containers (runnc). In Podman, the high-level runtime is (a variation of) CRI-O and the low-level runtime is runc or crun.

The main difference between the Podman and Docker system is at the high-level runtime, where Docker uses a system daemon service to manage the containers while podman simply executes the commands in the user space. This allows Podman to execute containers without root privileges and lowers the escalation risk. It should be noted that when it comes to security comparison of the container management systems there are four users involved:

- The user that executes the low-level runtime, which in case of containerd is the root user and for CRI-O is the user that executes the Podman commands.
- The mapped root user from the container to the host, determining the maximum privilege escalation achievable in the container with respect to the host. Unless the container is executed as privileged, the root user in the container is mapped to a non-root user in the host with maximum access level as the user who executed the low-level runtime.

- The user executing the main process in the container in terms of the user namespace inside the container. By default, this user is a root user, however, it can be changed to a non-root user.
- The user executing the current process in the container in terms of the user namespace inside the container.

These users could escalate to each other by the criteria shown in the figure 3.2. Even though the most secure case is the topmost one in the figure, for an average telco workload this mode is infeasible. The NFs normally require full control over their own namespaced environment with some extra capabilities to access certain resources on the host. Hence, the most suitable security type for an arbitrary NF is the one in green: rootful, unprivileged container with carefully selected capabilities and no syscall interception or translation.

Most of the container management systems use a Filesystem in Userspace (FUSE) driver to mount their container images. Snap uses the SquashFS and Docker and Podman use the OverlayFS driver. The OverlayFS is a type of union filesystem that allows mounting multiple layers of filesystems on top of each other, where each layer is a read-only diff of its parent layer, except the topmost layer which is read-write. The container builders use this layered filesystem to cache the results of the previous builds and reuse them in the subsequent builds. Also, when the images are distributed, the layers are distributed separately and the container runtime would only download the layers that are not already available in the local cache. OCI defines the way these images are stored and distributed.

Docker offers Moby Buildx and Buildkit for building the container images, while Podman relies on buildah. Podman builds, distributes, and stores the containers in strict compliance with the OCI standard, however, Docker uses a relaxed extension of the OCI standard known as the Docker Image Specification v2. The recipes for building the container images in Docker Image Specification v2 are called Dockerfiles while OCI prefers the more generic term of a Containerfile.

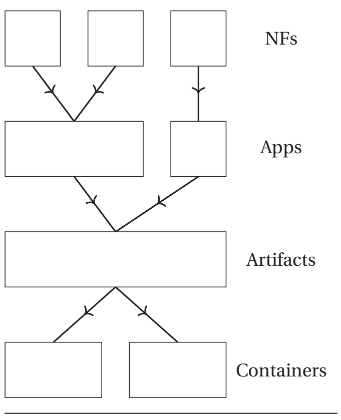


FIGURE 3.3: A schematic representation of the mapping between the NFs, application binaries, artifacts, and containers in runtime. The relationships between the NFs to apps as well as apps to artifacts is many to one, while the relationship between the artifacts to containers is one to many.

3.2 Build System

In this section we discuss the build system of HYDRA and TRIDENT, starting from how the NFs are mapped to the artifacts, to how the recipes are defined, and finally how the artifacts are built and automated. Along the path, we use abstract models where applicable to elaborate on the concepts.

3.2.1 Artifact Mapping

The first question that HYDRA needs to answer is how it maps the NFs to the artifacts that are going to be built and deployed. The specifications normally leave the technical details of implementation for each NF to the vendors, including how the NFs should perhaps be grouped into a group of application binaries or a single application binary. At one extreme, each NF could be a set of *microservice* application binaries and at the other extreme, all the NFs could be implemented in a single application binary. For example, Open5GS has a separate binary for each of the 4G and 5G core NFs, while Amarisoft implements all the functionalities of both the 4G and 5G core together in a single program. The mapping from the NFs to the application binaries is a matter of design preference and openness of the NFs.

Further down the path, the application binaries could be grouped into a single container image or each of them could be in a separate artifact. This is totally different from actually running the binaries in separate containers or a single container. The mapping between the applications to the containers is separate from the mapping of the *network services* and the containers. For example, a single container image might have the binaries for both the AMF and the SMF, however, there would be separate containers using the same image to run the AMF and the SMF. The figure 3.3 depicts these concepts.

The criteria that we picked for HYDRA is to have a single image set per vendor. The image sets would contain one image for the CN, one for the RAN, and possibly one for the UEs, if available. As the time of this writing, HYDRA supports Amarisoft, Open5GS, OAI, SRS and UERANSIM as the vendors. We performed analysis on the mapping between the application binaries and the container images both in terms of the build time and the image size to prove that our criteria is the best choice for HYDRA. This means given the savings on the build time, it makes sense to have extra overhead on the image size. Theoretically, one might imagine container images that cherry-pick the required files from the artifacts to reduce the size of image, but due to its practical complexities we have not considered it in HYDRA. The results are shown in the table 3.3. Due to the particular build strategy in TRIDENT, we only performed the analysis on building Snap packages.

The data in the table 3.3 is obtained by building the OAI core network artifacts on a machine with 40 virtual CPU cores of type Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and 64 GB of RAM. The Snapcraft version for the test is 7.5.2 and the overhead introduced by HYDRA on all the images is balanced. The builds for the Microimage approach are done in sequence to have all the machine

Metric	**Image Set**	**Microimage**
Parallel Build Time	31m22s	17m19s
Resource-based Time	24.510s	165.009s
Image Size	223 MB	831 MB
Number of Artifacts	1	15

TABLE 3.3: The metrics for image set and microimage approaches for the special case of OAI core network. The build times are measured as the maximum across all the different artifcats, while the image size incorporates the sum of the sizes of all the artifacts. This is due to the fact that potentially all the build jobs could be run in parallel.

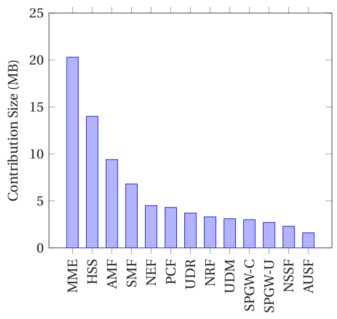


FIGURE 3.4: From the 223MB compressed image size of the OAI core network, 662 MB of uncompressed data is extracted, from which only 78 MB is the total share of the NFs binaries and the rest are the shared libraries or utilities. This figure shows contribution of each NF to that 78MB in sorted descending order.

dedicated, but assumed to be done in parallel. From this table we conclude that the Image Set approach saves 73.16% of the disk space and potentially download time of the images. The figure 3.4 shows the contribution of each NF to the total image size for the Image Set approach.

Before analyzing the build time, we need to introduce two metrics.

Definition 3.3 (Parallel Build Time). *The build time in wall clock time assuming all the jobs could be running in parallel on infinite instance of the same machine. Hence, the Parallel Build Time is defined per artifact mapping for each different setup of the machine. In case of multiple artifacts, the maximum of their build times is considered.*

Definition 3.4 (Resource-based Build Time). *The Resource-based Build Time is total CPU time of a build, assuming all the jobs are running in sequence on a single instance of the same machine. Hence, the Resource-based Build Time is defined per artifact mapping for each different setup of the CPU. In case of multiple artifacts, the sum of their build times is considered.*

The Resource-based Build Time is proportional to how many CPU cores it uses, which directly translates how much cost and energy it would incur. This cost could considered as part of the CapEx for the NFs. Given the definitions, the Image Set approach takes 44.80% longer to build the artifacts on the Parallel Build Time. However, the Resource-based Build Time is actually 85.15% shorter for the Image Set approach. This means a direct cut of CapEx down to one seventh of the Microimage approach.

This mapping is not without consequences though. In a sparse cloud environment where no two NFs sojourn on the same host, the image set approach introduces an overhead of downloading a larger artifact without sharing it with any other NF. However, there are two delicate points to consider in this argument. First, since TRIREMATICS is aiming for green computing, the scheduling of the Pods to the nodes is done with preference of a dense cloud rather than

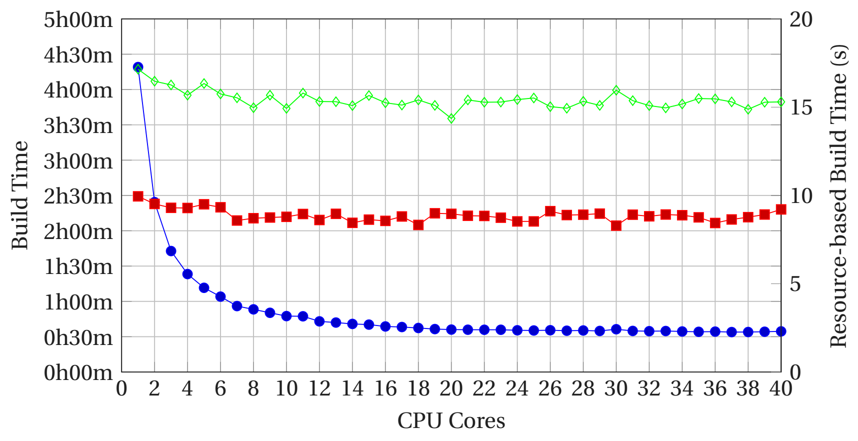


FIGURE 3.5: This figure shows the logarithmic decay of the build time versus the number of CPU cores for building the OAI core network artifact with the Image Set mapping. The measure on the left y-axis is the real clock time, while the measure on the right y-axis is the CPU time. The green line is for the user time and the blue line is for the system time, summed together to form the Resource-based Build Time.

a sparse cloud. Moreover, the images are cached for the subsequent deployments, allowing agile flexibility in changing the node assignment in Day-2 operations. Hence, the overhead defined in the table 3.3 for the image set is in the worst case scenario where the deployment is completely sparse and no caching could be used for running the containers. On the other hand, for the microimage approach we have the extra build time and number of artifacts to consider and maintain. The main practical challenge for maintaining the Image Set style is the cross-compatibility of the libraries used in the same artifact for different NFs. Expectedly, vendors keep the NFs compatible with each other, hence, the issue is not as severe as it might seem.

It is worth noting that not all the CPU cores could be utilized for all the kind of jobs. Hence, the value would fluctuate depending on the type of the job and the CPU architecture. For example, the figure 3.5 shows the build time of the same OAI core network artifact with the Image Set mapping on the same machine for different number of CPU cores. From this figure, one could notice that the build time is not linearly proportional to the number of CPU cores. Above 5 CPU cores, the build time reduces below 10% by adding each CPU core, which we consider as a decent cut-off point for the CPU cores. Furthermore, we notice that the Resource-based Build Time remains almost constant, regardless of the number of CPU cores. This is due to the fact that the build jobs are not CPU-bound and the bottleneck is the disk I/O or networking.

3.2.2 Build Recipes

Generally speaking, the telco artifacts are very complex, composed of several parts to build and lots of dependencies to install. To define a build recipe for such artifacts, currently most of the existing solutions rely on the imperative approaches in which you have to specify the exact steps to be taken for building the artifact. These steps either are realized as a complicated shell script (e.g., OAI) or Dockerfiles or perhaps a combination of both. An imperative build definition is extremely hard to maintain and extend, prone to human errors or inconsistencies, and difficult to parallelize. HYDRA uses a declarative approach based on the Snapcraft build recipes. Each Snapcraft recipe is a YAML file defining the artifact as union of several *parts*. The parts need to define the sources to be fetched, the plugin to be used for building, the packages required during the build, the packages required during the runtime, and any other environment variable or metadata needed. The parts might define dependency on other parts to dictate certain build order. This simple act of redefining the build recipes as a declarative Snapcraft file completely solves the obscurity of the shell scripts, addresses the maintenance challenges, and minimizes the errors in the build recipes.

It should be noted that Snapcraft does not allow the parts to be built in parallel, hence following the same logic, we define the build units in HYDRA per artifact not per part. This does not mean that for example a Make job for the part would be using only one CPU core, but merely that the the other parts would not be built in parallel.

Definition 3.5 (Build Unit)**.** *A build unit is the smallest manageable unit of the build process that could be issued for a parallel build. Each build unit is assigned a certain amount of CPU cores and memory that are used for the whole build process.*

By definition, the build units in HYDRA are the artifacts themselves, defined in separate Snapcraft files. These build units are carefully assigned to the system resources to maximize the concurrency and utilization of the system. Carelessly scaling the build units would actually result in a longer build time due to the overhead of the context switching and the resource contention. We discuss more on the topic of isolation and concurrency in the subsection 3.2.6.

Each artifact might have several variants as defined below:

Definition 3.6 (Artifact Variant)**.** *An artifact variant is a variation of the same artifact with different attributes. The attributes could be the CPU architecture,* *the CPU flags, the base image, the container management system, or the host OS.*

For the sake of *consistency* and lower build time, TRIDENT builds all the variants of the same artifacts based on the same Snapcraft recipe. Furthermore, the skeleton of the artifacts and the Snapcraft recipes all follow the exact same structure to allow automatic instantiation of new artifacts and systematic linting and testing. After the Snap packages are built on a certain CPU architecture, they are extracted from the Snap package format and then copied into the container images for the same CPU architecture. The Dockerfiles and the Containerfiles define how and in which order the files need to be copied into the container image to maximize the cache hits and minimize the build time. There are multiple base images available for each artifact to allow the users to choose the most suitable one for extending and customizing the artifact. Also, each of the images are built and tested on various host OSs to make sure the users can indeed build and run them on their desired OS, without any unattended degradation in the performance. The extracted and copied materials for the container images are regardless of the base image or the host OS, and they only depend on the CPU architecture. In fact, if the Snapcraft files are defined properly and without any dependence on the Snap Core packages, the resulted artifact could simply be lift-and-shifted to any other container management system and using any other base image. There are only two exceptions to this lift-and-shift strategy. The target base image needs to have the libc library of the version greater than or equal to the one in Ubuntu Core 20 (GLIBC 2.31) to allow dynamic linking of the libraries. Due to this constraint, the HYDRA artifacts cannot have base images in Redhat UBI 8 or older, CentOS 8 or older, Ubuntu 18.04 or older, Alpine Linux (since it does not use GLIBC), or the Scratch image. Also, the target run environment needs to have a matching CPU architecture as well as the flags. For example, an artifact built for amd64 architecture with AVX2 flags cannot be run on an arm64 architecture nor on an amd64 architecture without AVX2 flags.

3.2.3 Build Strategies

In the context of multi-x build systems, each variant of an artifact is attributed by its base image, container management system, container image builder, and the host system that it was built on. The official set of the base images supported in HYDRA are Ubuntu 20.04, Ubuntu 22.04, Redhat UBI 9, and the Google Distroless base variant. TRIDENT builds the Snap packages on Ubuntu Core 20 and with both the Multipass and LXD builders. For the Dockerfiles,

Statistics	Value
Vendors	5
NFs	30
Artifacts	130
Variants	2730
Base Images	4
Host Systems	3
Snap Packages	13
Docker Images	52
OCI Images	52
LXD Images	13
Architecture	3

TABLE 3.4: Some general statistcs about the HYDRA and TRIDENT projects. The NFs include the RAN (eNB, gNB, CU, DU), RIC, xApps, and the UEs.

it uses the Moby Buildx builder and the Docker Buildkit builder, while for the Containerfiles it uses the Buildah in the script mode, Buildah in the bud mode, and the Podman builder itself which are all slight different variants of the same builder. The official host systems include Ubuntu 20.04 with amd64 with SSE4.2 and AVX2 flags, Ubuntu 22.04 with amd64 with SSE4.2 and AVX2 flags, Redhat Enterprise Linux 9 with amd64 with SSE4.2 and AVX2 flags, and Ubuntu 22.04 with arm64. It should be noted that not every combination of these attributes is possible and some NFs are infeasible to be built on top of some system architectures. Furthermore, the images are built on different hosts or builders are not expected to be any different. They are built as part of *fluke tests* to make sure the artifacts are indeed portable and reproducible. For this reason, the not every artifact is graduated to release phase in TRIDENT.

To build these large number of variations of artifacts for several vendors, we have analyzed a few systematic build strategies based on build time, concurrency, consistency, and declarative automation. If we represent the directed graph of how different variants of artifacts are built from each other, we have defined a **build strategy**.

Definition 3.7 (Build Strategy)**.** *A build strategy is directed graph with vertices representing the stages for building variants of the artifacts and the edges indicating the destination vertex could be built by only inheriting all the files from the source vertex, except the base image or configurations The feasible build strategies are directed forests in the graph theory terminology.*

The vertices in different connected components or of the same depth in a tree component rooted from the first artifact variant could be built in parallel. Hence, we could declare the concurrency factor c of a build strategy using the following formula:

$$c = \sum_{i=1}^{n} \arg\max_{d,i} |V_{d,i}| \tag{3.1}$$

where n is number of connected components in the graph and V_d is the set of vertices in the depth d, i of the component i. Another interpretation of the concurrency factor would be by dividing the number of independent variants of the artifacts by the total number of variants of the artifacts. This should approximate the mathematical definition of the concurrency factor.

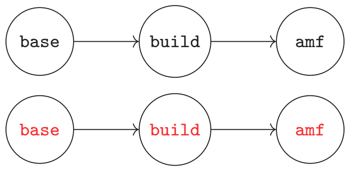
A consistent build strategy is the one which the variants in the same connected component have the exact same binaries other than the base image. Hence mathematically speaking, a consistent build strategy for n artifacts has exactly n connected components. In this thesis we consider four build strategies, namely the Separated, Matryoshka, Jigsaw, and TRIDENT, as depicted in the figures 3.6 and compare them in terms of the concurrency and consistency.

In a Separated build strategy, each artifact is built separately and independent of the other artifacts. This strategy is represented with a null graph and for obvious reasons, it has the maximum concurrency but the minimum consistency. In the Matryoshka (Russian doll) build strategy, the artifacts are built in a hierarchical manner, where the images are built as layers within each other to maximize the cache hits. Designing such a strategy is not trivial and requires a deep understanding of the dependencies between the artifacts. For the general, multi-vendor case of HYDRA, this strategy is simply infeasible. The Jigsaw build strategy does not limit the layers to be hierarchical, but one could cherry-pick the required files from several artifacts to build a new artifact. This is mainly done based on the multi-stage builds in Dockerfiles. Using this method has the downside of missing the caches compared to the Matryoshka strategy, and it could be as complicated as it to design for a generic case. The concurrency of a Matryoshka and Jigsaw build strategies is the same as its number of artifact variants. Neither of them could be considered consistent build strategies, unless each artifact has only one variant.

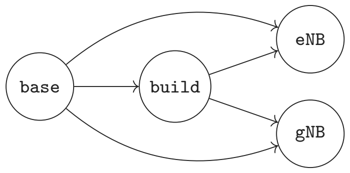
The TRIDENT build strategy builds Snap packages for all the possible variants first, then prunes the build tree by selecting one candidate per CPU architecture. These candidates then are used to generate the Docker, OCI, and LXD images all in parallel over several variants.

A third dimension to consider beyond the concurrency and consistency for a build strategy is the caching factor. In Docker or OCI images, practically each line in the Dockerfile or Containerfile would result in a new layer in the image, where the layers are cached and reused in the subsequent builds. The caching is also used while pulling the images from the registry. To maximize the cache hits, we carefully design the mapping between Snap packages and the Docker or OCI images, while at the same time leveraging the caches in the registry using the Styx build flow scheme, defined in the section 3.2.5. The optimized images first copy the common files between all the artifacts including generic libraries, then the unique files for each vendor, and finally the unique files for each artifact. This gives us the maximum cache hits and the minimum build time.

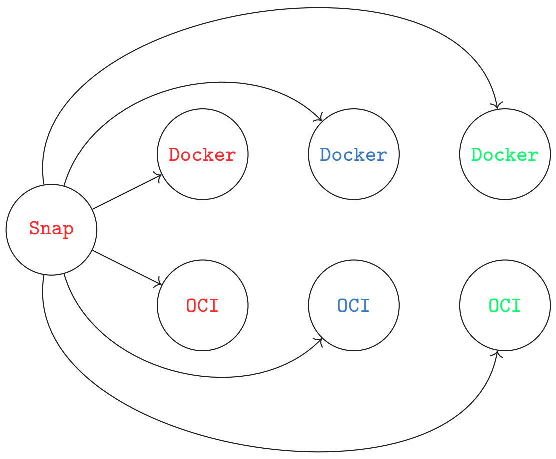
As an example, if we apply this caching strategy to the OAI RAN image, taking into account caches from OAI CN image (vendor-level caching) and AMR RAN image (NF-level caching), we have the pie chart shown in the figure 3.7. This chart indicates that from the 500 MB of the total Docker image size, 73 MB



(a). Two connected components from a Matryoshka build strategy. The different colors are for different variants of builds. This is used in practice for OAI AMF container images released by OAI. Each of the vertices is actually a container image saved during the build process.



(b). One connected component of a Jigsaw build strategy that is used in practice for the OAI RAN images published by OAI. Each of the vertices is actually a container image saved during the build process.



(c). TRIDENT build strategy.

FIGURE 3.6: As shown in this figure, the build strategies could be represented as directed graphs. The nodes are the variants of the artifacts and the edges are the dependencies between the artifacts. The colors are used to differentiate the variants of the artifacts.

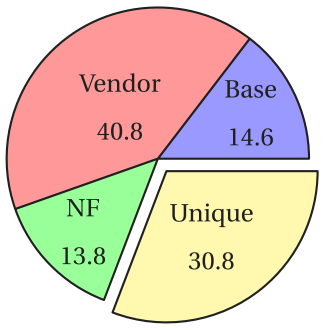


FIGURE 3.7: Cachig pie chart for the OAI RAN image with the decompositon into the base image, vendor-level cache, NF-level cache, and the unique files for the OAI RAN.

is the shared base image, 204 MB is the vendor-level cache, 69 MB is the NF-level cache, and only 154 MB is the actual unique files for the OAI RAN. This would induce a reduction of almost 70% in the build time, image size, and the network traffic.

3.2.4 Consistency and Concurrency

To demonstrate the points made on consistency we have devised a simple, but slightly exaggerated example to surface the issue. In the figure 3.8, we have used two separate base images namely Ubuntu:16.04 and Alpine:3.14 to build two similar artifacts, each of them containing only the OpenSSL package. First of all, notably the versions between the packages are different: in Ubuntu is 1.0.2g, while in Alpine is 1.1.1t. This causes a significant performance difference between the two artifacts. The test has been performed on a machine with 12 multithreaded Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz

CPUs in two independent time slots (to avoid CPU frequency scaling variations). The test command is `openssl speed md5` and the results are shown in the figure 3.8. As it is depicted in the figure, depending on the block size, we could have up to 70% performance difference between the two artifacts. Since OpenSSL is a critical library that is used frequently in programs, this performance difference results in significant unexpected performance variations in the NFs. Furthermore, we observe that some options are not available in the older Ubuntu image, which could be used to further accelerate the performance of the NFs.

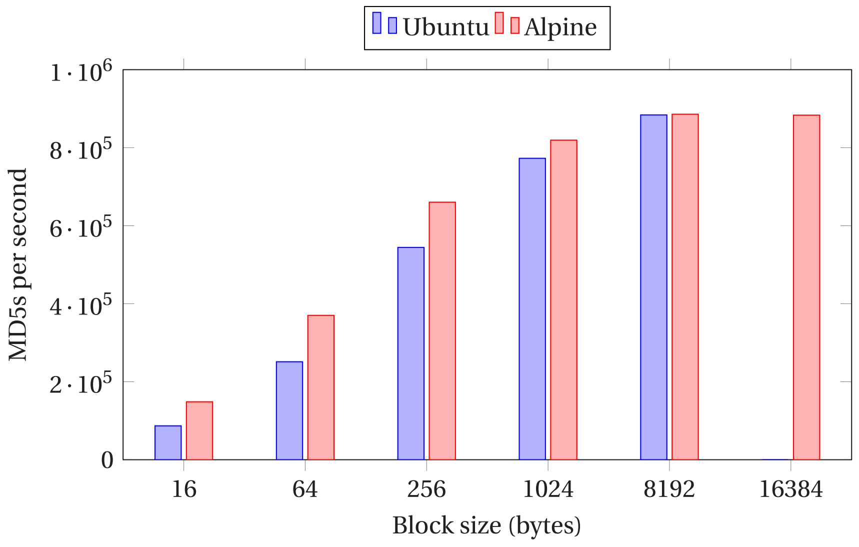


FIGURE 3.8: The performance unpredictability in the case of OpenSSL installed on Ubuntu 16.04 image (hash `b6f507652425`) and Apline 3.14 image (hash `9e179bacf43c`).

A similar issue exists for the vulnerabilities. In the table 3.5, we have listed the number of vulnerabilities found in installing Apache2 server on Ubuntu 22.04 and CentOS 7. The results are obtained using the Trivy* project and are limited only to the httpd package. As it is shown in the table, the number of vulnerabilities is different between the two distributions, reflecting completely different security postures. This issue makes the planning for the security in a multi-x environment very difficult.

S	Ubuntu	CentOS
C	0	0
H	0	0
M	5	16
L	24	5
U	0	0

TABLE 3.5: This table shows the difference between vulnerabilities found in the similar packages in Ubuntu 22.04 and CentOS 7. The vulnerabilities are categorized based on the severity level indicated in the column S, where C stands for Critical, H for High, M for Medium, L for Low, and U for Unknown.

As a result of the two previous experiments, we conclude that the consistency of the environment is a crucial to make the performance and security of the NFs predictable. In TRIDENT, consistency is achieved without much of sacrifice on the concurrency. There are four stages of build in TRIDENT, one for each of the container management systems. The first stage is the Snap build stage, which is the most time-consuming stage. All the variants for the Snap

* `https://github.com/aquasecurity/trivy`

packages are built in parallel, then the rest of the artifacts could all be built in parallel.

As pointed out earlier, the concurrency factor is derivable from the build strategy graph. In case of TRIDENT, given the statistics in the table 3.4 and the formula 3.1, the concurrency factor is 117, resulted from summing up all the Docker, OCI, and LXD variants. However, it should be noted that the concurrency factor is only the theoretical maximum number of parallel builds. In face of limited resources, the actual concurrent build rates needs to be adjusted to avoid the thrashing, context switching, and the memory pressure.

To address the concurrency in a limited resource environment, TRIDENT reserves the resources for each build job, complementing Jenkins[†] scheduling, as its build automation system. As discovered numerically from the figure 3.5, 5 cpu cores per build unit is a good cut-off point. Hence, for each 5 vCPU of the physical node agents, we define one executor in Jenkins, while reserving at least two CPU cores for the host processes, Kubernetes services, and the Jenkins agent process. The basic tasks such as triggers are executed with the `no node` option inside Jenkins' JVM to avoid occupying the executors on the agents. Additionally, we define a set of lockable resources in Jenkins presenting each of the executor slots with the same label. Each build job requests one of these resources to reserve the executor slot, which the actual reserved resource name indicates the exact CPU set to be used. This allows us to define specific CPU sets for each Snap or Docker build job, enabling a proper concurrent build system.

3.2.5 Versioning and Styx

Versioning is a crucial part of the DevOps to track the changes in the artifacts and to allow the users to keep a consistent environment. In TRIREMATICS, we use the Semantic Versioning (SemVer) v2.0.0 for every artifact or project. In particular, in HYDRA we use the following format:

```
<major>.<minor>.<patch>-<app>-<builder>-<base>-<suffix>
```

The `major`, `minor`, and `patch` are from the earliest git release tag parent to the current commit and the `app-version` is the version of the main artifact or repository packaged inside the artifact. The `builder` is the name of the builder used to build the artifact, the `base` is the name of the base image used to build the artifact, and the `suffix` is an optional suffix to differentiate the artifacts built with the CI pipline, patched via the CDK, or built locally. In

[†] `https://www.jenkins.io/`

case of release artifacts, the `suffix` and `builder` are omitted while a separate tag containing only the `major`, `minor`, `patchm` and the app would point to the artifact with the Ubuntu 20.04 base image. Since our DevOps is based on GitOps practices, all the files used for building and testing the artifacts, including the build recipes, the Jenkinsfiles, and the test scripts are versioned in the same repository as the artifacts.

In our CI pipline, we have defined barriers to prevent faulty artifacts to affect the downstream artifacts. This feature that is called **Styx** forces the different build stages to push their resulted artifacts to the registry first and the downstream stages would pull the artifacts from the registry. The pushed artifacts must be versioned, tested, and signed to be accepted by the registry. This allows us to have a consistent pipeline with several checkpoints to follow if the build fails abruptly. The fingerprint of the artifacts is stored in Jenkins and the build history checks the freshness of the artifacts. The images are signed using Cosign and the signatures are stored in the registry.

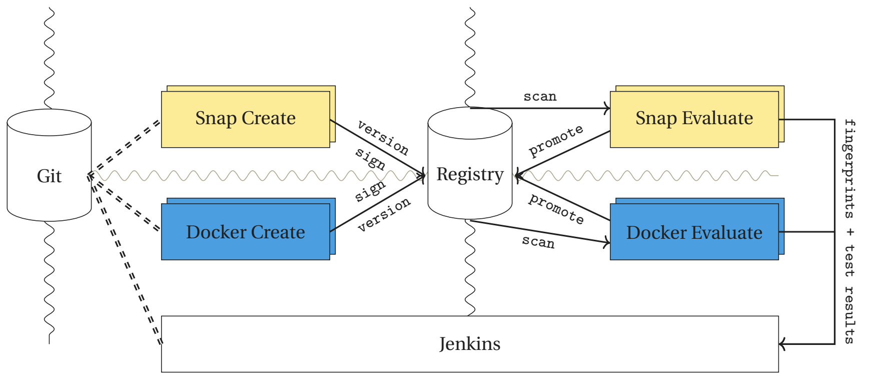


FIGURE 3.9: Styx pipeline examples for Snap and Docker. The curved lines show the Styx barriers and the dashed lines show transfer of metadata or triggers. This figure only shows the CI pipelines.

The figure 3.9 shows how a Styx pipeline is defined in TRIDENT. In this particular example the Snap and Docker pipelines are shown, but the similar arrangement applies to the other two pipelines. First a trigger in Git repository initiates the pipeline for Snap Creation. The resulted build artifacts of different variants are versioned and signed, then pushed to the registry. In the case of Snaps the registry points both to the Snap Store and our Harbor[‡] OCI registry in TRIDENT. The images are scanned for vulnerabilities and the results are stored in the registry. If necessary, the registry would replicate the images to the other registries. Then the Snap Evaluate pipeline is triggered, which pulls the artifacts from the registry and runs the tests. If the tests are successful, the artifacts are promoted to edge channel in the Snap Store and beta repository in the registry.

[‡] `https://goharbor.io/`

The Docker, OCI, and LXD pipelines are only triggered if the Snap Evaluate pipeline is successful and there is a new artifact in the beta repository. A corresponding set of pipelines for Create and Evaluate happens for each of the container management systems, while similarly respecting the Styx barriers, by submitting the signed and versioned artifacts to the registry, pulling it back, running the tests, and promoting the artifacts to the beta repository. They Evaluate pipelines store the resulted fingerprints and test results in Jenkins database. Those are later used for reference and checkpointing the pipelines. If the next trigger has no changes for a certain image set, the pipeline would be skipped for them. If one of the pipelines downstream fails, it retries the pipeline using the same artifact from that point in Styx. After all the pipelines are finished in the CI, the CD stages are triggered to promote the artifacts to the stable repository.

Essentially, Styx goes beyond merely GitOps into a more sophisticated GitRegistryOps. Every source should come from the Git repository and every artifact should be pushed to the registry, while the artifacts in the registry have versions linking them back to the Git repository.

3.2.6 Build Automation

We incorporate GNU Makefiles in HYDRA to define build commands for each of the variants. Some of the options could be defined as environment variables. The Makefiles provide enough abstraction to allow any build automation system to be used with HYDRA, however, not every build automation is aligned with HYDRA goals. Mostly the build automations are defined for building and testing the source code, while HYDRA is just about the artifacts themselves, not their origin source codes.

For an arbitrary software, the maintainers of its source code repository use a build automation system to build and test their source code, most likely inside a container. As a result of a successful pipeline, the source code alongside a reference container image are released. In HYDRA on the other hand, we are maintaining the recipes for building tens of artifacts themselves and the tests are about how the containers behave as a whole, alongside with the APIs added, not just the particular application binaries inside them. Then the containers are deployed and tested in a real environment where all together they form a complex system. In other words, what would be called perhaps a CD pipeline for a normal codebase turns into the CI stage for HYDRA, while the CD stage for HYDRA gets out of the scope for most of the traditional build automations. For this matter, we have chosen Jenkins as the build automation system used in Trident. Jenkins allows bare-metal builds and tests that are specially useful for building the Snap packages and LXD images. It also provides the right type of automation for building large number of multi-x artifacts that have the Trident strategy in mind. Of course, one could use hacks on other build systems to make them *comfortable* with Snap packages or building containers for telecommunication systems, but it usually demands workarounds lowering the security of the build system.

In Jenkins, we use shared custom workspaces that follow the flow of Trident to minimize number of pulls from the Git repositories. Beyond that, as we discussed earlier we use the lockable resources to provide execution isolation. The lockable resource are also to throttle jobs for building the same image set variant or accessing some system resources such as radio devices attached for testing. The variants of an artifact are defined using matrix jobs in Jenkins. All the Jenkinsfiles are defined in declarative syntax where usage of the shell commands are minimized down to mostly calling the makefiles.

3.3 Test System

A proper DevOps platform needs to have a comprehensive test system to ensure the quality of the artifacts. In this section we discuss the test system of Hydra and Trident. The section is divided based on each of the test categories and the tools used for each category.

3.3.1 Linting, Validation, and Compliance

Linting is a process of static analysis of the source code to find the potential errors, bugs, and stylistic issues. It makes sure the source code remains consistent and readable, while the git history stays clean. In Trirematics we use the Mega-Linter[§] project to lint all the source codes. The Mega-Linter is a Docker image that contains more than 70 linters and analyzers for more than 20 programming languages. It is a single Docker image that could be used to lint the source code of any project. The Mega-Linter is integrated with the CI pipeline of Hydra and Trident to lint the source codes before building the artifacts. Beyond that, the Mega-Linter is integrated with the CDK to lint the source codes before committing them to git. Using the git hooks, some of the basic checks are performed before committing the changes to git too.

In Trirematics terminology, validation means statically checking for errors in the configuration files and the build recipes. For example, the YAML files are

[§] `https://github.com/oxsecurity/megalinter` validated against their schemas and the Dockerfiles are validated against the Dockerfile Linter. The validation is performed in the CI pipeline of HYDRA and TRIDENT before building the artifacts, as well as in the CDK before committing the changes to git.

Compliance is a process of checking the artifacts against a set of rules to ensure the artifacts are compliant with the licensing policies. In TRIREMATICS we use the FOSSA[¶] project to check the artifacts against the licensing policies. The FOSSA is a Docker image that could be used to check the artifacts against the licensing policies. Furthermore, certain security and reliability guidelines about building and running containers are validated too.

3.3.2 Unit Testing

Unit testing is a process of testing the smallest testable parts of the software, called units, to ensure they are working as expected. The units in HYDRA are each individual variant of the artifacts. TRIDENT uses the Bash Automated Testing System (BATS)[⊠] to unit test the artifacts. These tests include basic functionality tests as well as smoke tests for each application inside the artifact. In a smoke test, the application is run with a simple configuration to make sure it is running without errors or warnings for a certain predetermined amount of time.

As the final stage of the unit testing, the artifacts are statically analyzed using the Trivy project for any security vulnerabilities or misconfigurations. It should be noted that Trivy does not simply work with a multistage builds such as the one used in HYDRA. To mitigate this issue, we copy the required metadata to the container and use a customized version of Trivy which is pointed out to the right files.

3.3.3 Multi-x Integration Tests

The integration testing is a process of testing a combination of artifacts work together as expected to deliver a particular scenario. The integration tests in TRIDENT are limited to the Docker images, where by the consistency assumption, the rest of the artifacts should work the same. The tests are done using constant configurations defined in several Docker Compose files. These tests

[¶] `https://fossa.com/`

[⊠] `https://github.com/bats-core/bats-core` are the reference scenarios for guaranteed performance with particular applications in the loop. The test results in terms of throughput, latency, and resource usage are generated and reported.

Since the composition models in HYDRA are distributed via Helm charts, some minor integration tests are defined in the Helm charts to make sure the composition models are working as expected. Beyond that, in the CD pipelines of TRIDENT, the multi-x and interoperability tests performed on randomly generated feasible scenarios. These tests cover more application scenarios, and their results are recorded via the observability stack in ATHENA. A particular Operator in ATHENA Operator Plane is responsible for running the tests, collecting the results, and reporting them back. Discussions on the composition models and Operators are deferred to the chapter 4.

3.4 DevOps Continuum

The DevOps practices incorporate one form or another of the three continuums: Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD). Both delivery and deployment continuums are abbreviated as CD. Depending on the type of the final artifact, the CD is either delivery or deployment.

3.4.1 Continuous Integration

The CI journey in TRIDENT starts from the CDK where the artifacts are defined, then continues to Gitlab where we use to store the source codes. In terms of git workflow, we use the Feature Branch Workflow where each feature is developed in a separate branch, but a single *main* branch is used for integration. This branch is tagged for the releases. This strategy simplifies the CI pipeline and the source code management. The commits in Trirematics source codes are done using the Conventional Commits specification with scopes defined for each artifact.

The CI pipeline in TRIDENT is defined and executed in the Jenkins, with one pipeline verifying every push to the *main* or *develop* branch and another one building, testing, and releasing the artifacts for every tag or merge request. The pipelines are defined in the Jenkinsfiles in the Jenkins declarative syntax and stored in the TRIDENT repository, while the source codes themselves are stored in the HYDRA repository. The pipelines are triggered by the webhooks from the Gitlab via the Jenkins Gitlab integration plugins.

Due to massive number of variants in the table 3.4, the push pipelines result into no artifacts and are done only for the changes made in the last commit. This is to minimize the time required to verify the changes and make the development cycles agile. For the tag pipelines, every image set is built, tested, and released in the beta channel of the registries. The tag cycles are no shorter than once a month.

3.4.2 Continuous Delivery

TRIDENT uses Semantic Release[**] to automate the versioning and releasing of the artifacts in a continuous manner. The Semantic Release is a tool that analyzes the commit messages to determine the next version of the artifact and automatically releases the artifact to the registries. It also generates the changelog and the release notes for the artifact. The Semantic Release is integrated with the CI pipeline in TRIDENT to release the artifacts to the beta channel of the registries.

The resulted artifacts that managed to pass the Styx barriers become candidates for the release and are listed in the release notes. The images are distributed via a private Harbor registry that synchronizes the images with the public Docker Hub, Artifact Hub, and the partners' Harbor registries. The Snap packages are also distributed via the Snap Store. The rule of thumb in TRIREMATICS is that every artifact needs to be stored in the registries.

The CD tests are then executed on the artifacts in the beta channel of the registries to tag them for a stable release. These tests start with the multi-x integration tests via Docker Compose as mentioned earlier. The artifacts graduated from the integration tests are release candidates. Then we use the ATHENA to run the multi-x and interoperability tests on the artifacts in a Kubernetes cluster.

For rolling an update on a running system, we need to define a deployment strategy.

Definition 3.8 (Deployment Strategy)**.** *A deployment strategy is a set of rules and procedures for deploying the new version of an artifact while minimizing the downtime and the scope of the effect.*

There are several common deployment strategies in CD, namely blue-green, canary, rolling, and A/B testing . TRIDENT uses different strategies depending on the type of the NFs.

[**] `https://github.com/semantic-release/semantic-release`

For the DP in the CN, the best strategy to use is a mixture of the blue-green and canary deployments, boosted by slicing. We call this deployment strategy **Slice and Dice**.

Definition 3.9 (Slice and Dice)**.** *In a Slice and Dice deployment, n new instances with the new version of the NFs are deployed alongside the $o \geq n$ older existing instances. Then a new slice is created for the set of new instances which gradually the users of n randomly selected older instances are migrated to that slice, hence being served by the new n instances. Then the older n instances are decommissioned. The process is repeated until all the older instances are decommissioned.*

The value $1 + \frac{n}{o}$ is the surge factor, indicating how much more resources is required for rolling out the new version. Let us review an example of this deployment strategy with the UPF NFs: The new version of UPF is deployed first alongside the older version, then using slicing, the traffic is gradually shifted to the new version, while measurements are taken to ensure the new version is working as expected.

For the rest of CN, if the NF is stateless, the rolling deployment is used, otherwise, the blue-green deployment is used to minimize the downtime and context loss. For the RAN, however, it is almost always impossible to deploy two versions sharing the same radio at the same time. In result, we have defined the new deployment strategy called **Handover Rolling**.

Definition 3.10 (Handover Rolling)**.** *In a Handover Rolling deployment, the UEs are handed over to a neighboring cell, while then the older version is being replaced with the newer one, and finally the UEs are handed back to the new version. This in a larger scale would look like a rolling deployment.*

The message sequence chart for this deployment strategy is shown in the figure 3.10 for the example of only one UE connected to a single gNB instance.

After incremental deployment tests are done on the artifacts, the corresponding composition models are updated and a stable release is made. The stable release is then distributed to the customers via the registries alongside the step-by-step upgrade guide.

3.4.3 Continuous Deployment

The final deployment in TRIDENT happens in production environments. After a stable release is made, the corresponding artifacts receive updates following the same deployment strategies done during the testing phase. Alternatively,

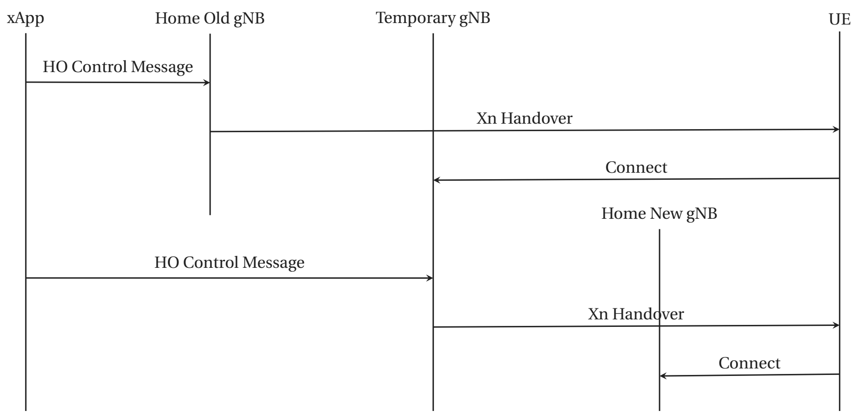


FIGURE 3.10: The message sequence chart for the Handover Rolling deployment strategy. For simplicity the RIC is removed from the figure and the connection between the xApp and the gNBs is show directly. From the figure, the lifecycle of the Home Old gNB ends after the HandOver (HO) and then the lifecycle of the Home New gNB could start with the same resources as the Home Old gNB. The message flows for HO are simplified.

the end users could use rolling updates or complete re-deployments to update their artifacts.

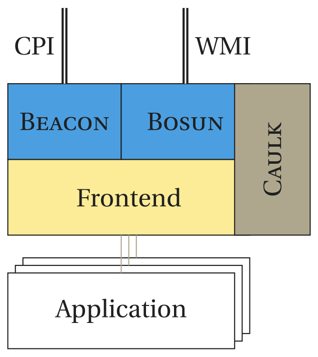


FIGURE 3.11: The structure of the Workload container in HYDRA. The BOSUN and BEACON repsectively provide the WMI and CPI interfaces. The CAULK controller is to provide unified multi-x key-value backend for storing the configuration and state of the applications.

3.5 Container Structure

HYDRA defines a default preferred container structure for the artifacts. This architecture could be violated as long as the artifacts maintain the interfaces with the MANO as defined in the chapter 4. Of course, TRIDENT would only fully support the artifacts that follow the default container structure in HYDRA and it provides no promises for the artifacts that violate the structure.

The default structure is shown in the figure 3.11. In this figure the term *Application* refers to the NF binaries, while **Workload** is the collection of tha applications, their dependencies, configuration files, and the files and APIs imposed HYDRA. In this sense, the Workload's perimeter matches the ones of the container image.

3.5.1 Frontend Scripts

HYDRA provides a set of basic shell scripts named **frontend** to unify the structure of initialization and starting up the application binaries in the container. In particular, they provide the following multi-x functionalities:

1. A unified initialization script for each of the Applications in the Workload.

2. A set of utilities for logging, NUMA policies, device detection, interactive sessions, and environment variables.

3. A configuration script for assigning configuration files to the applications, modifying them using an editor, or generating them from templates.
4. A run script for starting the Applications in the workload with signal trapping and cleanup.
5. A test SDK script for testing the Applications in the workload, used regularly in TRIDENT pipelines.

In terms of lifecycle of the Applications, the most important script is the run script. It allows the Applications to gracefully respond to the signals sent by the container runtime or the *Manager* and performs all the required cleanup before exiting, including closing the file descriptors, removing the temporary, closing the pipes, and killing any (zombie) child processes. The exit status of the Applications are collected and reported to the Manager, alongside a matching against a long list of known exit codes. Using the Linux pipes, this scripts enables access to the internal CLI of multiple Applications in the Workload too.

The test SDK allows definition of any custom tests in BATS format as long as they are placed in the tests directory identified by the `TESTS_DIR` environment variable. The multi-x nature of the frontend allows it to execute the Applications in any environment, including Snap and Docker. The frontend also incorporates some metadata files in JSON format to be used mostly by TRIDENT. The importance of using the test script instead of running tests directly is to make sure the tests are executed in the exact same environment, security context, and configuration as the Applications are run in the production

For the Snap execution environment, frontend also provides a set of hooks to be used by the Snapcraft. These hooks are defined for the installation, refresh, and configuration of the Snap. Snap provides its own backend for keeping a set of key-value pairs for the configuration of the Snap. To make this compatible with our multi-x design, we have defined a minimal controller script named CAULK that implements the same interface as the Snap backend for other execution environments, hence the frontend scripts could load and set those pairs regardless of the execution environment in the exact same way.

3.5.2 Application Programming Interfaces

HYDRA defines two set of Application Programming Interfaces (APIs) for the Workload:

1. The Workload-Manager Interface (WMI), implemented as BOSUN server in the current release.
2. The Container Probing Interface (CPI), implemented as BEACON server in the current release.

Beyond that, HYDRA allows definition of any customary APIs for the Applications in the Workload and exposes them to the MANO. The APIs are declared in the `api.json` file in the `CONF_DIR` directory of the Workloads. The Manager with the right supporting plugins could use these APIs to control the Applications in the Workload in a custom manner. This method for example is used to implement O-RAN O1 interface plugins in ATHENA.

A WMI implementation needs to provide at least these five simple API endpoints:

1. `init` for initializing the applications in the workload.
2. `start` for starting the applications in the workload.
3. `stop` for stopping the applications in the workload.
4. `status` for checking the status of the applications in the workload.
5. `info` for getting the information about the applications in the workload.

These endpoints are defined using a HTTP/1.1 REST API, where the URL format is as the following:

```
<host>:<port>/api/v1/wmi/<endpoint>/<?app>?<arg=val>
```

The scheme, host, and port are defined in the `api.json` file, while the optional `app` parameter is to run the endpoint on a specific Application in the Workload. If not specified, the endpoint is run on a default application named `main` provided by the frontend scripts. Custom arguments could be passed to the endpoint using the query string as key-value pairs.

BOSUN operates in three possible modes: *autonomous*, *managed*, and *standalone*. In the autonomous mode, the API automatically calls itself to initialize and start the `main` application, while in the standalone mode, it waits for the user to interact with the API to perform the actions. In the managed mode, the API server is not started until the Manager has provided the `api.json` file. Essentially, in this mode, it is the Manager who decides where and when it is expecting the API server to be running and BOSUN lifecycle explicitly starts after the Manager initialization. The autonomous mode is used in TRIDENT pipelines for testing, while the standalone mode could be used for running the scenarios manually.

WMI defines a state machine for each Application the Workload as depicted in the figure 3.12. The Manager uses this state machine to manage the lifecycle of the Applications in the Workload. Bosun is also responsible for gathering the exit codes of the applications and reporting them to the Manager when requested using the `status` endpoint.

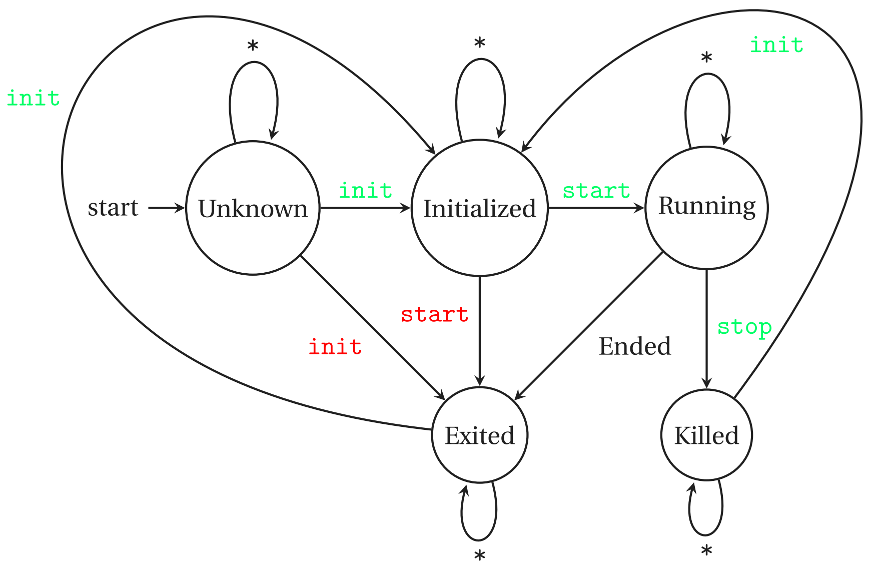


FIGURE 3.12: The state machine for the Applications in the Workload. The states are shown in circles and the transitions are shown in arrow. The transitions are labeled with the corresponding API endpoints. The special character * is used to indicate the rest of the calls. A green color means a successful call and a red color means a failed call.

The CPI is a simple HTTP/1.1 REST API that provides a single API for probing the container status. It provides all the common endpoints for health and readiness checks, including `health`, `healthz`, `healthy`, `live`, `liveness`, `ready`, `readyz`, and `readiness`. The URL format is as the following:

```
http(s)://<host>:<port>/api/v1/cpi/<endpoint>/<?app>
```

The scheme, host, and port are defined in the `api.json` file, while the optional `app` parameter is to run the endpoint on a specific application in the workload. If not specified, the endpoint is run on a default application named `main` provided by the frontend scripts. Docker and Kubernetes use these endpoints as reference for the health and readiness checks of the containers.

The difference between the CPI interface and simply checking the status of the applications using the WMI or the state of the processes using the `ps` command is that the CPI is *continuous*.

Definition 3.11 (Discrete and Continuous Probing)**.** *A discrete probing is a probing that only takes into account the exact moment of the probing. A continuous probing is a probing that takes into account the time interval between* *two probes.*

A discrete probing is very common in Kubernetes. In a continuous probing using the CPI, if the application was not running at any point between two probes, it is considered as *unhealthy*. Hence, beyond the basic endpoints, the CPI also provides two special endpoints namely `dp` and `cp` for discrete and continuous probing respectively. The discrete probing gives an overestimation of the health of the application, while the continuous probing gives an underestimation of the health of the application, hence the true availability of the application is sandwiched between these two values.

To elaborate on these notions, let us formalize the problem. If in a discrete probing, we probe the container at regular intervals with inter-probing interval of I, to detect a fault, the probing needs to be performed when the Application is indeed in a faulty state. The Application might be restarted before the probing has the chance to detect the failure. This is a byproduct of how agile Management works in ATHENA, where the Manager is continuously checking the application status and restarting it if needed, as reviewed later in the chapter 4.

Take the example of figure 3.13, where the application is in a faulty state between $t = 3.5$ and $t = 4.5$. The discrete probing with $I = 3$, starting from $t = 0$ would have no samples in the faulty state. In general, if we define the Fault Interval F as the length of a continuous failure interval, in this case $F = 1$. If $F < I$, it might be cases like the example that the detection of error is impossible. Since F is random and I is constant, finding the right probing interval is a difficult task.

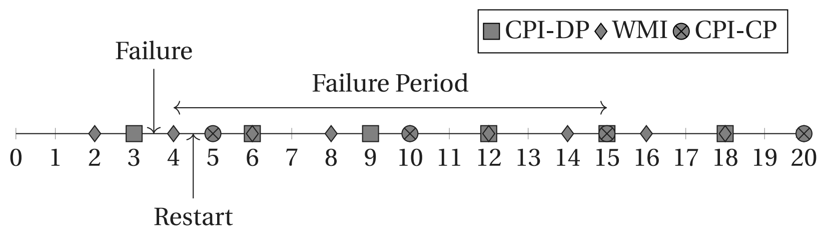


FIGURE 3.13: The discrete and continuous probing of the Application for the example provided in the text.

In the continuous probing, the Application needs to remain healthy for a predefined period of time, $T > I$. In this example $I = 5$ and $T = 6$. This safety measure underestimates the Application's health, but it never misses a failure. The Manager locally performs periodic calls to `status` endpoint in the WMI and records the time of the last successful probing in intervals of S units of time, where $S < I < T$. Since the Application is not restarted unless WMI detects a failure, there could not be any failure that is not detected by the WMI and consequently not reported by the CPI of the Manager. The reported failure period is at least $T+S$, in this case from $t = 4$ to $t = 15$. Assuming an availability metric A for the application, the discrete probing would always overestimate the actual value while the continuous probing would always underestimate it. If the ratio of the successful probes over the total probes in the discrete mode is R_D and for the continuous mode is R_C, then the equation 3.2 is satisfied.

$$R_C \leq A \leq R_D \tag{3.2}$$

3.5.3 Parts and Layers

It worth mentioning that all the artifacts in HYDRA have some common parts in their Snapcraft definitions. These include the `bats`, `api`, and `panacea` parts which are for testing, API server, and the configuration utilities respectively. The `frontend` part is also included in every Snap, but with slight variations depending on the NFs.

Using this information, to maximize the caching in Docker and OCI build processes, we define the layers of the containers as the following, shown in figure 3.14:

1. **Base**: This layer is the base image of the container.
2. **Meta**: Metadata and environment variables depending on the builder might be included as a single or a set of separate layers.
3. **Common**: Files related to the three common parts are included in this layer.
4. **Vendor**: Vendor-shared files are included in this layer, for example the common libraries used in OAI images.
5. **Shared**: The NF-shared files are included in this layer, for example the UHD images and libraries that are used for every RAN artifact.
6. **Specific**: Any other files or layers that are not shared between the artifacts.

As mentioned earlier, this saves 50% on the build time and 30% on the total aggregated size of the images.

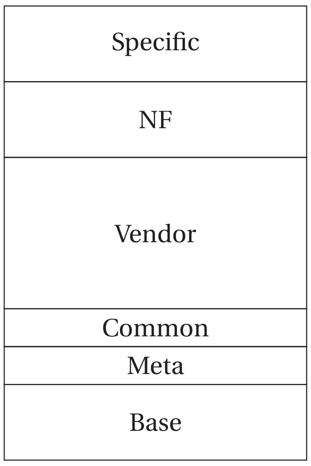


FIGURE 3.14: The layers of the containers in HYDRA. The layers are shown in rectangles with rough height correlated with their expected sizes. The layers are ordered from bottom to top.

Chapter 4

Declarative MANO

In this chapter, we dig deeper into the design and implementation of Management and Operation in the Athena project as the cornerstone of the automation in Trirematics platform. Athena is majorly based on the concept of the declarative automation.

As we laid out earlier in section 2.7, there have been several new design principles emerged with cloud-native that demand a revolution of the MANO designs. The novel designs are so foundational that a mere refactoring of the older MANO solutions would not be sufficient. The key to all of these principles is the declarative automation which has lead to the novel definition of the Operator Plane in Athena in section 4.2.

Unlike chapter 3, the method of study in this chapter is to propose a design alongside with a concrete implementation of a cloud-native, multi-x, declarative MANO. Section 4.3 defines the internal design of the Base Operator, while the section 4.4 dives deeper into the design of the level-1 Operators.

The required level of agility is then realized by the introduction of the sidecar management in the section 4.6 alongside all the internal components of the Manager. We evaluate this agility as well as day-2 operations and the overhead of Athena in the section 4.9.

The concepts of this chapter are largely influenced by the concept of multi-x. On one hand, multi-x MANOs are motivated by the development of the Open RAN systems, and on the other hand, they are supported by the cloud-native principles. Furthermore, the new cloud-native trends in the telco industry are in favor of the multi-x MANOs. For example, the majority of the telco operators would define somewhat a portion of their infrastructure as a set of private clouds. Unlike homogeneous public clouds, these private clouds could be extremely heterogeneous in terms of the hardware, OS, or the container runtime, since each of them have been independently deployed and managed earlier for particular sites. Moreover, with having recycling of the older or spare compute resources as a contributing factor in sustainability, the heterogeneity of the private clouds would even increase over time. In that sense, perhaps a multi-x MANO is a feature for today but a necessity for the future.

This chapter goes through three phases of Deploy, Operate, and Monitor from the DevOps cycle displayed in the figure 3.1 using the MANO concepts. Essentially, we define the Management and Operator Plane in ATHENA while assessing it for Day-1 (Deploy, Configure) and Day-2 (Observe, Reconfigure, Upgrade) operations. At each part of the design, whether in Management or Operation, there are certain challenges to support a declarative, efficient, agile, and sustainable cloud-native MANO. These challenges are addressed in terms of sophisticated, yet extensible and well-defined design patterns emerged from the coalition of the cloud-native and telco principles, with significant novelties in each part of the design. The table 4.1 summarizes the novelty of each section in this chapter, while the table 4.2 shows which challenges are addressed in each section in terms of the affected metrics or qualities.

Section	**Novel Designs or Algorithms**
4.2	Multi-x Operator Plane
4.2	Network resources as cloud-native constructs
4.3.1	Declarative cloud-native network Operation, Network scoping
4.3.1	Network scoping via role-based SBA
4.3.3	MANO programmable policy enforcement
4.4.1	E2E Operation for the UEs and their applications
4.4.2	Declarative first-class citizen slices
4.5.1	Dual formulation of slicing with assignment
4.5.2	Scaling stateful NFs by slicing
4.6.1	Distributed multi-stage network dependency resolution
4.6.4	Inbox and outbox observability
4.6.4	Green metrics for MANO
4.7	Radio devices as cloud resources
4.7	Unprivileged secure CNFs
4.8.1	Multi-source data lake for telco observability
4.8.1	Idempotent and auto-healing MANO
4.8.3	Micro-decisions and Macro-decisions

TABLE 4.1: The novelty of each section in the chapter 4 with respect to the state of the art.

Section	Addressed Metrics, Qualities, or Designs
4.1	Complex multi-x ecosystem
4.2	Cloud-native MANO, Extensibility
4.3.1	Declarative, Cloud-native, Multi-x Operations
4.3.2	Cloud-native, Extensibility
4.3.3	Edge services, Policies in MANO
4.4.1	E2E Operations including the UE and application
4.4.2	Declarative E2E slicing, Slice scheduling and assignment
4.5.1	Slice scheduling and assignment
4.5.2	Stateful scaling
4.6.1	Circular dependencies, Dynamic infrastructure, Agility
4.6.2	Service continuity, Day-2 reconfiguration
4.6.3	Fault tolerance and recovery, Agile lifecycle
4.6.4	Green Observability, Low overhead
4.7	Security, Device discovery and inventory
4.8.1	Fault Recovery, Multi-source observability
4.8.2	Zero-downtime reconfiguration, NF upgrades on-the-fly
4.8.3	Agility, Fine-grained Management

TABLE 4.2: The challenges addressed in each section in the chapter 4 with respect to the metrics, qualities, or design questions.

4.1 Multi-x Operator Ecosystem

ATHENA is formed around the idea of **multi-x** as its motivation, whereby ‘x’ stands for vendor and radio in telco context and container runtime, OS, or cloud provider in the cloud context. Standing at their meeting point, ATHENA covers all the mentioned dimensions from the both realms. It embeds support for simultaneous deployment of workloads from different vendors relying on the different radio devices (exemplified in the section 5.2), while supporting multi-node, multi-cluster, multi -tenant, and multi-runtime deployments on top of K8s. Multi-clustering is supported by using the Border Gateway Protocol (BGP) networking both internally and externally in each cluster. This allows the Pods to be routable from the other clusters. Other common methods are either specific to HTTP (like ingress/egress gateways) or are less efficient due to the encapsulation overhead (like VPNs). Furthermore, multi-networking is added to ATHENA using the Multus CNI plugin[*] . Multi-tenancy

[*] `https://github.com/k8snetworkplumbingwg/multus-cni`

in ATHENA is supported by the Kubernetes namespaces and the corresponding Role-Based Access Control (RBAC) policies. Kubernetes networking policies defined on the top enforces the isolation of the namespaces.

We have foreseen multi-x as the natural generalization of Open RAN which seeks beyond RAN to CN and MANO/OAM itself. Thus, we believe ATHENA as a multi-x Operator would enjoy the same positive expectations of Open RAN as briefed for example in [17], [45]. This includes the reduction of the vendor lock-in, the reduction of the cost, and the increase of the innovation pace. In the figure 4.2, we have demonstrated the variety of vendors as the most important dimension of multi-x by different shapes. This also includes interaction with external elements, not directly under the control of ATHENA (marked by the *External* label in figure 4.2).

Telecom industry has a very complex and multiplayer ecosystem that complicates development and innovation by posing challenges in terms of interaction with other providers for establishing a functional deployment. The existence of the standardization bodies is to address this issue, at least in UP and CP. In accordance to the multiplayer ecosystem of telco, we have separated the concerns on the network parameter, including radio, identity, and slicing with the ones about the composition of the network and how it scales. In this way, the vendors and integrators could evolve and upgrade their software, independent of the network operators. The networks are **declaratively** defined as a Network CR in Kubernetes, whereas the composition of such networks are described in Composition Model CRs that abstract the cloud-related parameters, such as images, resources, and networking as well as the configuration of the workloads. The Composition Model CRs also define the observations that need to be collected from the Workloads as well as the scaling policies for the NFs.

This separation distinguishes between *synthetics* and *semantics* of (multi-x) logical networks. The synthetics defined in the Composition Models govern how a network deployment corresponds to a deployment of Pods in Kubernetes and their interconnections. On the other hand, the semantics of the network in the Network CR assign telco attributes and configuration to those generic deployments, for example, in terms of the radio parameters, the identity, or the slicing. The adaptation of simple yet extensible Composition Models would enable the vendors and the service operators to independently innovate on top of ATHENA.

On this regard, we have detected five major classes of players that might interact with our ecosystem, depicted in the figure 4.1.

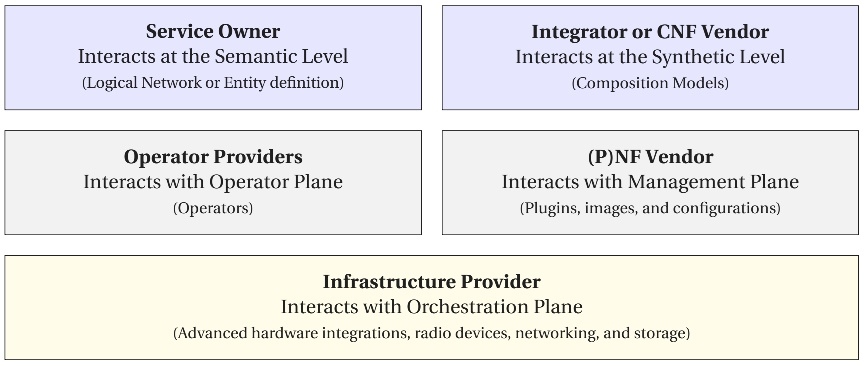

Service Owner
Interacts at the Semantic Level
(Logical Network or Entity definition)
Integrator or CNF Vendor
Interacts at the Synthetic Level
(Composition Models)
Operator Providers
Interacts with Operator Plane
(Operators)
(P)NF Vendor
Interacts with Management Plane
(Plugins, images, and configurations)
Infrastructure Provider
Interacts with Orchestration Plane
(Advanced hardware integrations, radio devices, networking, and storage)

FIGURE 4.1: The major players in TRIREMATICS ecosystem.

1. An Infrastructure Provider who primarily provides orchestration extensions to the hardware control mechanism on the Orchestrator Plane;
2. A PNF Vendor contributes plugins and configurations to the Manager at the Management Plane;
3. An Integrator or CNF Vendor would package and onboard new workloads to TRIREMATICS by defining the new Composition Models;
4. An Operator Provider builds new Operators on top of ATHENA OP;
5. An entity with the Service Owner role would simply use TRIREMATICS and all of its extensions as a turnkey solution to realize another concept on top of it.

4.2 ATHENA and the Operator Plane

ATHENA, according to the figure 4.2, transforms presentations of network resources, network functions, and containers from a traditional Kubernetes resource such as Pod or Service to a logical network via a **Base Operator** capable of controlling and abstracting the networks and their elements. The abstracted format incorporates the network operators' primary concerns, customized to telecom language and use cases, without the underlying infrastructure's complexity. Vendors supply the recipe to *compose* a network in the standardized format of Composition Models in the artifact registry. Each Composition Model is a CR in Kubernetes containing a simple list of Workloads declaring their images, required plugins for configuration, roles in the network, and hardware resources associated with them, without specifying any network-level definitions. The abstraction in the Composition Model is specified independently of how it is deployed or configured, which are logical entities and parameters defined in network design and planning should be represented on top of the synthesized workloads.

The notion of the Operator Plane, in which many Operators cooperate handle logical presentations and network entities in various aspects, is one of the main innovations of ATHENA. The functionality in the Operator Plane might be enhanced in two levels. Operators at Level-1 consume logical network CRs to reveal a new set of CRs targeting a logical item such as a slice or network terminal. The level-2 is built on top of those logical entities, to enable sophisticated and perhaps E2E Operations, like cost optimization. Through the Operator Plane, concepts and functionalities are transcended from physical to logical, abstracting both the network itself and the related concepts.

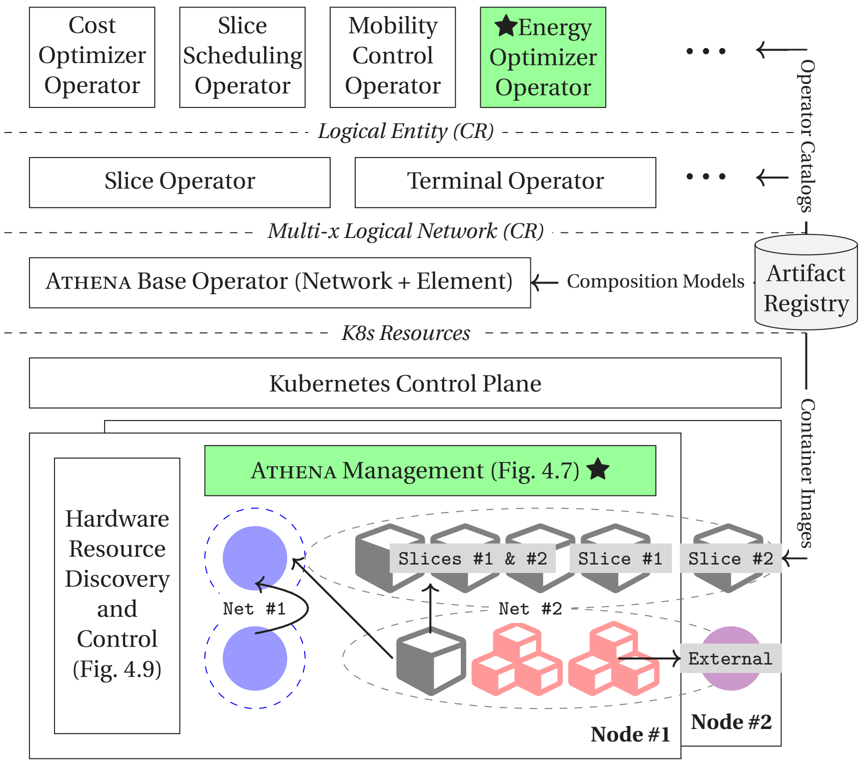


FIGURE 4.2: ATHENA overview of the OP in a nutshell presenting the multiple levels of the OP.

In the figure 4.2, we have shown the two levels of the Operators discussed earlier, with some examples. On the first level, the Slice Operator and Terminal Operators are examples of transition of an inherent concept in telco to a first class citizen CR in Kubernetes. These Operators not only would consume the functionalities provided by the Base Operator, but might have their own independent interaction with the other Operators or Kubernetes. This Service-Based Architecture (SBA) allows arbitrary extensions in the Operator Plane, without the need to modify the Base Operator. On the second level, three kind of examples are presented:

- The Cost Optimizer and Energy Optimizer Operators are the examples of End-to-End business-oriented optimizations providing cloud native solutions to the active industry problems;

- The Slice Scheduler shows how a concept introduced at the level-1 could itself introduce a whole dimension of new Operators on the level-2;
- The Mobility Control Operator is an example of how the MP/OP and CP could interact with each other by defining a new Operator on the OP that would also interact with the RIC.

Each Operator should define new concepts via the CRs and take care of the lifecycle management of those CRs via implementing the logic in of the corresponding concepts. The Operators are themselves Pods in Kubernetes, hence they need to respect the guidelines and limitations of the Pods in terms of access control or resource management. These combined criteria define when one should define a new Operator and when it is sufficient or required to define a client of existing Operators' APIs, perhaps external to the cluster. Each Operator is composed of a few CRs and the Custom Controllers (CCs) which upon invocation by the requests from the orchestrator would *reconcile* the corresponding CRs.

The (Non-RealTime) RIC platform could also potentially participate in the Operator Plane to connect the Control Plane to the Operator Plane. In result the RIC could request and observe xApps or rApps using the CRs provided by the Base Operator. Beyond that, the logics of the other Operators could be influenced and extended jointly by the RIC. For example, the Energy Optimizer Operator would not only consider the RAN as a mere CNF, but it could also trigger control loops in the RIC for fine-grained optimizations. In this situation, from the RIC perspective, the Energy Optimizer Operator behaves as an rApp. The said approach is used in the ODIN project which is out of the scope of this thesis.

One should note that Athena also provides the means and SDKs to build and distribute the Operators for the Operator Plane through OCI-based Operator Catalogs and Operator Lifecycle Manager (OLM)[†] . By relying on the OLM functionalities of the Operator Framework, the dependencies of the Operators on the Athena Base Operators could be explained and a reliable Catalog of them are made and distributed. One major rule in the dependencies is that for each CR, it should be at most one CC and the Operators should try avoiding implementation of the same functionalities. For example, by definition, with the Base Operator, there should be no other Operator that tries to establish Pods or Services independently.

[†] `https://olm.operatorframework.io/`

4.3 Base Operator

As the first level of abstraction, the Base Operator transforms the Kubernetes objects into expressive logical presentations of networks and their associations with each other and the cluster, linked by the provided Composition Models. This Operator hence could deploy and manage end-to-end 4G/5G networks that are requested via declarative intents in the form of CRs. The base Operator itself includes two CCs:

1. The Network CC defines and governs the logical networks (access, core, or edge) as a group of Elements with their associated slices;
2. The Element CC maps the Elements to the NFs using Pods, Services, Deployments, and ConfigMaps in Kubernetes while controlling the scaling and lifecycle of the Elements.

This distinction affects observability, management, and scaling of the networks. It helps to abstract and aggregate the observability of a network as a whole, while being able to observe individual components. Also, the Operators on the Operator Plane could utilize the Element CC directly by managing the network and creating the Element objects on their own. This approach is indeed used in the Terminal Operator to manage network terminals[9]. In that sense, the concurrent instances of the CCs in the Base Operator are adapted to address the unbalanced number of requests processed by the Element CC compared to the Network CC. The separation also defines how a network needs to respond to scaling and slicing. Based on the slices, different number of Elements might be needed to be deployed, while each of the Elements might create multiple replica[10] instances of the NFs.

[9] A network terminal is a UE in the modern 5G terminology.

[10] A replica is an exact copy of a Pod with the same configuration. For example, one cannot provide different slices to two replicas of the same Pod, since it would not be an exact copy anymore.

4.3.1 Network Controller

ATHENA Network CC consumes a Network CR that contains the description of the slices as well as the list of access, core, and edge networks, possibly multiple instances of each. In the access networks definition, the radio parameters including the device and signal features and the cell parameters including the center frequency, the band number, subcarrier spacing and the bandwidth are defined.

Each of the network sections might have a custom networking based on the Multus CNI plugin for each of their network interfaces. The Domain Name System (DNS) records could also be appended and customized for the whole

Network CR, applicable to all the containers in that network as well as the terminals. The network sections incorporate configuration profiles, post configuration assignments, and means for providing extra keys for the configuration. These utilities are used to tweak the configuration of particular NFs or the network as a whole.

Upon invocation, the CC looks up for the Composition Models of each network. Based on the obtained information, the CC builds a topology of the network which later is used by the Management Plane to resolve the dependencies in a distributed manner. ATHENA uses a **Role-based SBA** approach for the network topology, where each NF is enlisted for several roles in a particular **scope** of the network. The scopes are assigned labels to the network sections defining an access control mechanism for the managers to resolve the dependencies. The Base Operator would use scopes in association with the slices to build the configuration for each Element. Each Element receives the list of all the roles implemented in its scope with the corresponding fully qualified names of the NFs playing that role. The precedence of the NFs in the list is defined by the slices associated to them and is resolved by the managers. This topology is scoped and merged with the any provided custom DNS records to form a localized view of the network for each NF and support Multi-Access Edge Computing (MEC) applications. Finally, the Network CC issues Element CRs to be later picked up by the Element CC that works in parallel. For the observability, the Network CC aggregates the status of all of its wrapping Elements to indicate the status of the network itself.

The Network CR also defines the scheduling properties of the Pods of the network. Each network section has optional fields for assigning particular Pod scheduler, specifying the priority and preemption policies, and defining the affinity and anti-affinity rules.

4.3.2 Element Controller

The Element CC builds the corresponding Pods and Kubernetes Services with the proper configuration. During this process, it complies with the best practices of Kubernetes and connects the Pods to the proper container scheduling parameters that are defined for the Element. Element CC supports four types of backend for deployment of actual Pods:

1. Raw Pods, the fastest option where the Element CC would simply deploy the Pods directly;

2. Deployment, the most compliant with the Kubernetes best practices, where the Element CC would deploy the Pods using a Deployment object, providing extra scaling via replication;
3. StatefulSet, similar to the Deployment, but with the extra support for the stateful NFs;
4. DaemonSet, where the Element CC would deploy the Pods using a DaemonSet object, spanning over the whole range of the cluster nodes.

In each of the backends, the Element CC could be managing the object directly or simply observe it passively for the sake of the observability.

This CC would also continuously probe the Pods for custom Kubernetes Conditions that are provided by the Composition Models. These custom Conditions use the Kubernetes Probe interface and might be used to determine the readiness of the Pod, if marked in the Composition Model. The Element CC also records and aggregates the Conditions exported by the Pods for the observability. From the Composition Models, the Element CC would resolve all the storage volumes, custom networks, or additional devices that are required by the Element to assign them properly to the Pods.

4.3.3 Support Services

The Base Operator provides a set of support services that are not implemented as CCs. These services include the custom DNS server and the policy agent. In ATHENA every NF is assigned a unique Fully Qualified Domain Name (FQDN) that properly represents its identity. The DNS records are of the following format:

```
<element-name>.<section>.<network-name>.<namespace>
```

Since the Kubernetes Services are not allowed to be named as such with subdomains, the Base Operator then translates these records to a unique hash that is used to create the corresponding Kubernetes services named **Shadow Services**. The Shadow Services are used for the internal communication between the Managers.

The policy agent is responsible for enforcing the policies and licenses defined in the Composition Models or in the Operators themselves. The policies are defined in the *Rego* language and are processed by the Open Policy Agent

(OPA)[‡] integration in the Base Operator. Each policy is queried for a predefined policy action, for example `data.policy.deny`, to determine if a particular deployment should be denied or not. By the default, all the deployments are allowed.

The Base Operator is also responsible of resolving any conflicting decisions made by the Operators on the higher levels, by simple priority assignments using Kubernetes annotations and authorizations. Each Operator is annotated with a priority number and a level number, which they need to provide with their requests to make sure the decision overriding is done properly. By default the lower the level of the Operator, the higher is its priority.

4.4 Level 1 Operators

In this section, we summarize the internal structure of two of the Operators on the level-1 of the Operator Plane, namely the Terminal Operator and the Slice Operator.

4.4.1 Terminal Operator

The Terminal Operator is defined to handle the network terminals and achieve an E2E control loop that also contains the UE and the applications associated with them. It supports four types of the terminals or UEs:

1. The simulator mode where deploys a simulated 4G/5G UE to connect to the corresponding simulated eNB/gNBs, regardless of the level of the simulation (RF, L2, S1/NG, etc.);
2. The external mode where it is just a presentation of a handset outside the cluster and no container would be deployed in this mode;
3. The internal mode where a container is deployed attached to a physical UE module on the cluster;
4. The backhaul mode to support external network formations that shall use this terminal as a backhaul. The identity of the UE is automatically injected into the databases of the corresponding core networks, given their identification.

The Terminal requests are given to the Operator in the form of a Terminal CR that contains the identity of the UE and the network it should connect to. Via the network identification, the UE's identity is automatically injected into the

[‡] `https://www.openpolicyagent.org/`

databases of the corresponding core networks. In case of the simulators, a proper placement of the UE is done by the Terminal Operator to minimize the latency and undesired effects of the simulation. For the internal mode, the Operator would request for the corresponding Quectel module that contains the SIM with the exact same IMSI as defined in the Terminal CR. The placement of such UEs then is dependent on where the physical modules are connected to the cluster.

The Terminal Operator also could receive a BatchTerminal CR where it contains a list of identity information that should be added to the database and a selection of the number of active UEs. Then the Operator would randomly pick UEs and connect them to the network and report back their overall status. This enables large scale simulations and testing of the network. A Terminal Scheduler Operator or a Mobility Control Operator on the level-2 of the Operator Plane could exploit geographical metadata of ATHENA Workloads alongside the collected metrics to optimize the placement of the UEs and their preferred registered network to maximize the availability of the network for batch deployments. The result would be an adaptive and optimized placement of the UEs in the cluster under any customized strategy, useful for testing the equipments or mobility algorithms.

The Terminal CR also provides means by which an application could be onboarded with the UE. In the simulator or internal modes, a set of application containers could be defined to be deployed in the same Pod as the UE, hence enjoying the same 5G network in their network namespace. These application containers are not managed by ATHENA and could be any arbitrary normal application containers. Four classes of applications are definable in the Terminal CRs as shown in the figure 4.3:

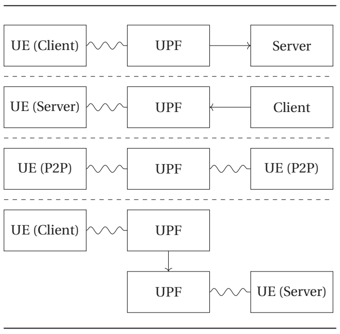


FIGURE 4.3: Application modes in the Terminal CR. The curly lines are to show the connection is not indeed direct and goes through the RANs not shown in the image.

1. Uplink applications where the server-side is located behind the UPF and the client-side is located in the UE's network namespace;
2. Downlink applications where the server-side is located in the UE's network namespace and the client-side is located behind the UPF;
3. Peer-to-Peer (P2P) applications where both the server and the client are located in two different UEs under the same UPF;
4. Multi-UPF scenarios where the server-side and the client-side are located in two different UEs under two different UPFs.

Depending on the scenarios, the Terminal Operator would create the corresponding Services and routes to connect the applications to the network. The part of the application is deployed behind the UPFs need to be deployed by the user on the same cluster. Then Terminal Operator would create a unique Kubernetes Service named after each IMSI that routes the traffic to the corresponding UPFs responsible for that UE. The UPFs would then route the traffic to the corresponding UEs. In the Uplink scenario, no special configuration is needed and the application could be deployed anywhere in the cluster. In the P2P case, the UEs can connect directly or use the DNS names, however, in the multi-UPF, the UEs need to be configured to access to each other's Service names created by the Terminal Operator.

4.4.2 Slice Operator

Slicing in 5G enables sharing the same infrastructure for varieties of E2E services that mostly have incompatible requirements. We respect this service-oriented outlook of the network by including slices at the highest level of abstraction. On the Base Operator, each Network CR has a list of slices with their corresponding Public Land Mobile Network (PLMNs) and Data Network Names (DNNs)[11]. The network definitions on the same file are allowed to *filter* their slices based on PLMN, DNN, Slice Service Type (SST), or an exhaustive list of the slice indices. The slice picking process is independent of the connection between the access and core networks and the Basic Operator performs no checks for feasibility of the slice allocations.

[11] DNNs are equivalent to the concept of Access Point Names (APNs) in 4G.

On contrary, the Slice Operator *assigns* slices to the networks based on the scheduling mechanisms defined for it, which results to a Network CR with exhaustive listing of the slices at the end. Having slice assignment at the MANO level enables optimized compute, network, and radio resources all together. We recognized the assignment problem for slices to virtual networks is analogous the problem of assigning pods to the nodes. Thus we devised a slice scheduling framework based on the Kubernetes scheduler design[§] to assign slices to the networks, depending on the provided filtering and scoring criteria defined in the CR, following these steps:

1. **Sorting** out the slices for assignment;
2. **Filtering** *feasible* allocations based on the criteria in the CR;
3. **Scoring** and sorting the eligible assignments;
4. **Admission** of the assignment with possible extra actions regarding coordination with controllers, high-availability, or scaling.

[§] `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/`

Each of these stages could be extended with plugins in a similar fashion that Kubernetes allows for the scheduler extensions, and they could include affinity and anti-affinity rules. Even though this method compared to the original filtering would be capable of more complex allocations, still by its mathematical formulation, this method is limited to the results achievable by a form of greedy algorithm (matroids to be exact). Open interfaces on the Operator Plane allows for a fully customized scheduler in terms of another Operator on level-2.

ATHENA assigns a global readiness status to the slice to indicate its availability. Modifying or deleting a slice object would trigger the corresponding action to be called on the controllers' APIs. Managing these events as well as how the rest of statistics would be collected and organized is dependent on the controller and out of ATHENA scope. We consider ATHENA slice CRs the best anchor point for marrying the similar concepts in the RAN, the CN, and the RIC to a single slice object. Actions on this object would trigger the corresponding actions on the RIC or NSSF, for example, to enable a declarative E2E slicing in the network.

4.5 Slicing, Scaling, and Replication

In ATHENA the concepts of slicing, scaling, and replication are related to each other. As mentioned earlier, each Network CR defines a list of slices and how they are filtered for each of the access, core, or edge networks. The filtering could have been done by the assignments from the Slice Operator. At this point the Base Operator reads the Composition Model to create the corresponding Elements of that network section. However, depending on the preferences defined in the Composition Model, each of the Elements listed might be scaled to support the assigned slices. There are four types of slice scaling that could be defined in the Composition Model:

1. Scale per unique PLMN, useful for the NFs that cannot support multiple PLMNs at the same time;
2. Scale per unique DNN, to perhaps have separate UPFs supporting each of the applications;
3. Scale by the SSTs, to separate the needs of the NFs for their services;
4. Scale per slice index, to have a separate instance of the NF for each slice.

In this section we model and study both the assignment and the scaling problems for the slices in ATHENA. We exemplify the approaches using a simple set of slices provided in the table 4.3.

ID	PLMN	Type	Differentiator	DNN
1	00101	eMBB	0x000001	operator
2	00101	eMBB	0x000002	internet
3	00102	eMBB	0x000001	operator
4	00102	uRLLC	0x000001	internet
5	00102	mMTC	0x000001	operator

TABLE 4.3: Example details of 5 slices filtered for an access network section.

4.5.1 Slice Assignment

If we imagine each slice definition has a few attributes much lower than the number of slices that would be assigned to the networks, and likewise, the number of attributes of the networks is an order lower than the number of the networks in question, then we could consider the efficiency of the assignments too. Following the table 4.3, and the figure 4.4, the slices are sorted by their SSTs and DNNs. The figure reflects the filtering and scoring procedures as we go through that sorted list for four RAN instances identified by their vendors and radio devices as example attributes that could be used for filtering and scoring. The four instances are OAI running on B200 SDR, OAI running on N300 SDR, Amarisoft (AMR) running on B200 SDR, and AMR running on the Amarisoft SDR50 devices.

At the filtering stage, due to limitations of OAI in supporting non-eMBB slices, we filter it out for both IDs 4 and 5. This knowledge is provided as part of the scheduling logic in the Slicing Operator. For a delay-critical uRLLC slice, usage of SDR50 on PCI interface is preferred over UHD-B200 on USB, and for mMTC, the presence of ID 4 on the fourth RAN, lowers its score due to anti-affinity rule. The Slice Operator would intelligently perform this filtering and scoring based on the Network CR and the logic implemented in the Operator.

On the other hand, for eMBB slices the supported bandwidth of the device becomes the deciding factor which for both IDs 1 and 3, the N300 instance could support more bandwidth resources (50MHz each in fair assignments) than a single B200 instance (56MHz). At this stage, an energy efficient action is to colocate the IDs 1 and 3 on the same machine to reduce the power usage by lowering the number of active nodes. Finally, the ID 2 is marked with high availability to serve all the users of PLMN `00101` with basic connectivity, hence it is assigned to all the RAN instances but the fourth one that has received negative score due to the anti-affinity of uRLLC.

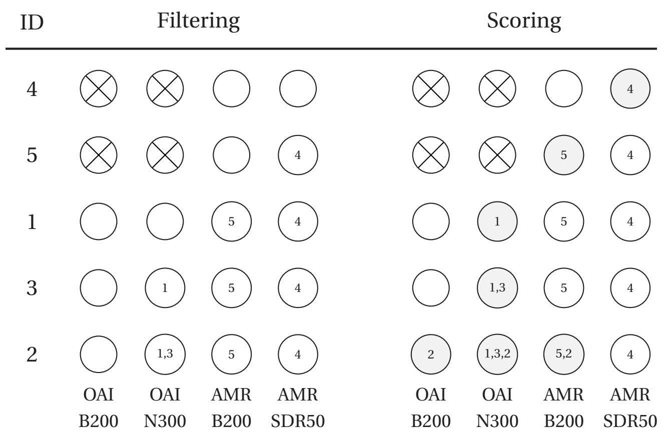


FIGURE 4.4: Slice scheduling example for 5 slices over 4 RAN instances. This figure shows how a sample filtering and scoring procedure could resolve into a complicated assignment of the slices to the RANs by the Slice Operator.

Based on our assumptions and considering n slices with c core networks, a access networks, and e edge networks, then we could conclude the following theoretical bound for the algorithm's execution time

$$\Theta(nlog(n)) + \Theta(c+a+e) + O(n(c+a+e)) + O(n(c+a+e)) \tag{4.1}$$

where the first term comes from sorting n slices, the second on is filtering all the networks, the third for scoring them, and the last one for admission of the slices. Even though in the filtering phase, we need to consider all the networks (exact bound), the next two phases may have lower number of networks. Thus, according to the equation 4.1, we have $\Theta(nlog(n)) + O(n(c+a+e))$ as the order of execution time for our algorithm. This bound shows the algorithm scales very well with the number of networks (linear) and slices.

We could compare slice allocation algorithms according to two extremes:

1. Defining a new dedicated network per slice;
2. Assigning all the slices to all the networks.

Both of the mentioned methods involve higher resource usage, where the former is obviously doing so by having unnecessary networks provisioned and the latter is vulnerable to the peak provisioning by allocating resources for the sum of the slice demands. For instance, consider the case of the example slices in the table 4.3. Provisioning all the RAN instances for uRLLC means all of them should perhaps be deployed with a commercial CNF that supports this kind of slice, which significantly increases CapEx. On the other hand for the first option, we need at least one other RAN instance which adds up to OpEx and we loose the functionalities like high availability of the slices.

4.5.2 Scaling and Replication

To elaborate the mentioned capabilities on the scaling, let us consider a RAN Composition Model, containing a CU and a DU Element, where the CU is set to scale with DNNs perhaps to connect separately to individual UPFs for each DNN and the DU is set to scale with the SSTs to optimize the resources for each service type. Given the sample slices defined in the table 4.3, we need two instances of CU and three instances of DU, with the connections given in the figure 4.5.

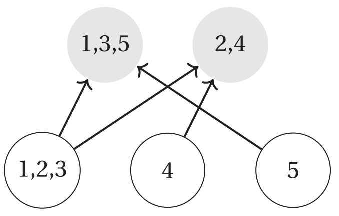


FIGURE 4.5: How the slicing defined in the table 4.3 would affect the scaling of the CUs and DUs in an imaginary composition model. The hollow circles are the DUs scaled up to three instances connecting to the two CUs shown by the gray circles. The numbers in the circles show which slices are supported by each instance.

Each of the circles in the figure 4.5 represents a separate Element, however, since they have been emerged from a single description in the Composition Model, they would be assigned to the same Shadow Service. It is by the Management Plane to make sure each of the Elements is actually connected to the right instance of the Element served under the same Shadow Service. This concept is elaborated in the section 4.6.1.

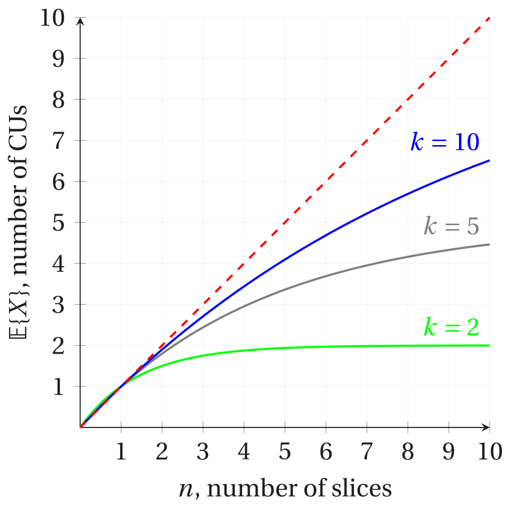


FIGURE 4.6: The theoretical gain of scaling definition in ATHENA compared to scaling per slice definition. k is the number of different DNNs and n is the number of slices.

Since ATHENA allows each individual NF scaled in accordance to the slices, then scaling procedure would be simple and efficient. For example consider a specific core network design that has a User Plane Function (UPF) capable of supporting only one slice at a time. If by for example NFV definition [20], one tackles this setup, then he needs to either consider a new NS for the UPF and scale it independently, or scale the core network as whole. The first approach would need a complex management with the resource waste of a new

NS definition and the second one demands significantly more and unnecessary resources to scale other parts of the NS as well. Consider the example of the figure 4.5, where each NF is scaled based on the provided option. If taken the approach of scaling the entire RAN NS defined as one CU and one DU instance, supporting the same slices, we would have used twice the instances. Thus, ATHENA offers 50% resource reduction in this specific example.

We could generalize this example by considering k different DNNs for n slice definitions in ATHENA. The same generalization may be used for the other parameters. Denoting the number of CUs which would be equal to the number of DNNs with the random variable X, we could write the equation 4.2, where X_k is an indicator random variable for each DNN.

$$X = \sum_{i=1}^{k} X_k \Rightarrow \mathbb{E}\{X\} = \sum_{i=1}^{k} \mathbb{E}\{X_k\} = \sum_{i=1}^{k} \mathbb{P}\{X_k = 1\} \tag{4.2}$$

Each indicator variable is from a Bernoulli distribution with the success probability of $1-(1-1/k)^n$, thus we end up with the value in the equation 4.3.

$$\mathbb{E}\{X\} = k - k(1-\frac{1}{k})^n \tag{4.3}$$

In the figure 4.6, we have plotted this value and comparing it with n, standing for a scaling per slice definition, for three different value of k. The plot shows that if k is much smaller than n, the gain would be significant. A good example for that scenario is to consider SSTs instead of DNNs which could practically take only three values. In any case, the saving is absolute and clear by the plot.

Beyond this scaling, since each instance of the Elements creates a Deployment in Kubernetes, the Pods assigned to that element might be scaled by **replication**. This type of scaling is ignorant to the slices and is done by the Kubernetes itself. The resulted Pods are supposed to be exactly the same and the scaling is done to increase the availability of the NFs. Each of the Pods in this case should be equally capable of serving all the slices assigned to the Element.

4.6 Sidecar Manager and Pod Design

ATHENA Management Plane works as an extension of its Operator Plane, but in a distributed manner to take local, short-term decisions rather than global, longer-term ones. The former supports macro-decisions whereas the latter defines micro-decisions. A distinguishing example of these oppositions is captured in the green capabilities of ATHENA, marked with a star in the figure 4.2. The Energy Optimizer Operator would take decisions concerning day-long Operations using perhaps AI/ML algorithms, whereas the Manager supports micro-decisions to control fine-grained, semantic lifecycle of the NFs for energy savings. The energy saving capabilities, whether observation or control, are integrated to the ATHENA design, perfectly fitting in the distinction of the Manager and Operator. In definition of the interfaces between the Manager and Operator or the Manager and the Workloads, we use open interfaces with simple, yet strict protocols that are less prone to protocol ossification [46] while being modernized and cloud native, preferring *de facto* standards over the *post facto* practices[12].

[12] *De facto* refers to a situation that exists in reality, even though it may not be officially recognized or legally established, wheras *post facto* refers to something that occurs or is done after the fact, usually referring to laws or rules that are applied retrospectively.

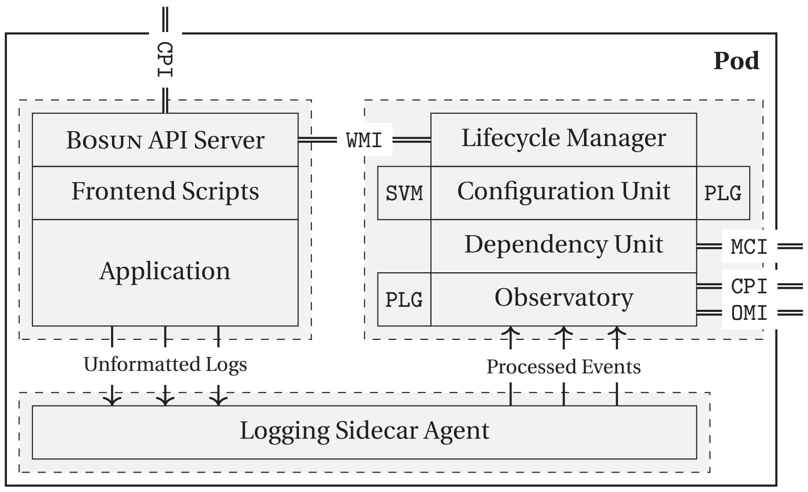


FIGURE 4.7: Sidecar Manager and Agents in ATHENA. The interfaces are shown by the double lines with their names on top.

The Pod in our proposed sidecar design depicted in the figure 4.7 is composed of up to four types of containers, two of which are mandatory and two are optional.

1. The Workload containers that have the Applications with some frontend helper scripts, depending on the applications, that are with a generic API server that implements the Workload Manager Interface (WMI);
2. A single Manager sidecar container;
3. Optional utilities agents such as for logging or collecting metrics, many of these agents could be replaced with their eBPF counterparts;
4. An optional set of sidecars for loading the external applications or the test suites, which are not managed or used by the Manager.

Compare the figure 4.7 with the figure 3.11 to understand how the container structure in HYDRA emerges into the Pod design in ATHENA. BOSUN is the name of the default implementation of the WMI server in the Workload container. The Workload and Manager containers share a common volume where the Manager would place the configuration files for the application to consume.

ATHENA Manager, depicted in the figure 4.7, is actually the distributed part of the MANO with several new interfaces required to facilitate Day-2 operations and observability. There are several options to choose for where to locate the Manager. For example, the Managers could have been placed as node daemon services, hence a more traditional approach of managing the NFs per node. However, we chose the sidecar pattern for the Manager to:

- **Decouple** the Management Plane from the other planes, common in cloud native designs;
- **Accelerate** adaptation to cloud native for the rather traditional NFs;
- Keeping the workloads **lightweight** and **portable** to any environment by keeping the cloud-specific interfaces independent of the NFs.

This approach is similar to the ETSI-EM model, with cloud native adaptations. On the other side of spectrum, solutions such as Facebook's Magma[¶] Converged Core try to deeply integrate the cloud native services into the NFs themselves. For that matter, for example, the codebase adapted from OAI MME to Magma MME has been polluted with a lot of dependencies from gRPC, Protobuf, Prometheus, and Redis, making it less portable and more difficult to maintain.

The Manager is chiefly responsible for performing seven tasks, each of them discussed in one of the following subsections.

4.6.1 Dependency Resolution

The Managers of each Element may communicate with each other through the Management and Composition Interface (MCI) in the Management Plane. Each Manager is given a *discovery* list from the Base Operator, by which it could resolve each of the *roles* to a set of potential FQDNs. Each FQDN is resolved to a Shadow Service by the Base Operator's DNS server, which is then resolved to the IP address the Kubernetes Service behind it. Depending on the scaling policy defined in the Composition Model, the Base Operator might have created multiple instances of the Element that only differ in the slices they support. In that sense, all of these Elements would be under the same

[¶] `https://magmacore.org/`

Shadow Service. An indifferent Manager, could then resolve the DNS only once and get assigned to an arbitrary load-balanced instance of the Element. However, in the general case, the Managers would discover all the endpoints behind that Shadow Service to process the entries according to their slicing.

The MCI is a simple JSON-based REST API with two endpoints:

1. `resolve/{interface-name}` to resolve the IP address of a particular interface of that Element such as `resolve/xn` or `resolve/e2`;
2. `depends/{interface-name}` works similar to the `resolve` endpoint, but it would wait for the readiness of the dependee and would return an error if the dependee is not ready;

Optionally, in each request the Manager could send a slice matching template. The slice matching template is a list of slice definitions similar to what is defined in the Network CR, however with regular expressions instead of the exact values. If this template is provided, the MCI would only return successfully if the the template matches with the slices served in the dependee and would provide the resulted matched slices in the response.

The **Dependency Unit** in the manager is responsible for implementing both the client and server side of the MCI. Beyond that, it continues querying the same MCI endpoints to make sure the dependency is still valid. If the dependency has failed for any point in time, the Dependency Unit would react depending on the time of the original call made to that Element.

Definition 4.1 (Strong and Weak Dependency)**.** *A dependency is considered strong if the dependant needs the dependee to be on ready state continuously. On the other hand, a weak dependency is a mere dependence on the IP address or other parameters of the Pod from another Element, used to fill in the details in the configuration.*

Weak dependency is represented by a call to the `resolve` endpoint. If the original call was made to the `resolve` endpoint, the Dependency Unit would only trigger a reconfiguration if the IP address or the slice response list of the dependee has changed. On the other hand, a strong dependency means the dependant needs the dependee to be on ready state, so the call is not returned successfully, until the managed is running and healthy. This is represented by a call to the `depends` endpoint. If the original call was made to the `depends` endpoint, the Dependency Unit would trigger a reconfiguration and potentially a restart regardless of the change in the IP address or the slice response list. A readiness in the MCI context means the application was running healthy and the conditions were met for a *continuous* period of time, T, without any interruptions. The default reaction to a failed dependency might be changed by a policy defined in the annotations in the Composition Model.

Having the strong and weak dependencies enables the Managers to resolve even circular dependency graphs, conditioned that every cycle has at least one weak dependency edge. This means if we consider an arbitrary directed graph with edges colored either strong or weak, then the graph is solvable with the Dependency Unit if and only if it does not contain any cycle fully colored with strong edges. To prove this, we could just remove the weak edges from the graph and the remaining graph would be a DAG. Every DAG could be solved by a topological sort, which is exactly what the Dependency Unit does. The weak links could all be resolved at the same time as the first stage before traversing the DAG.

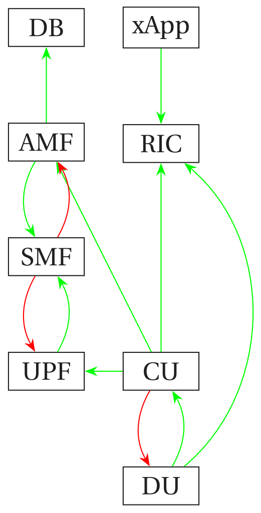


FIGURE 4.8: A practical example of a dependency graph in a 5G network. These graphs like this are solvable by the Dependency Unit. The network is a sample Open RAN with a minimal 5GC deployment. This figure's graph matches the actual deployment requirements from OAI. The green arrows are strong dependencies and the red ones are indicators of weak dependencies.

Figure 4.8 shows a practical example of a dependency graph in a 5G network. In this figure, optimally, DB, RIC, and SMF would become ready first, followed by AMF, UPF, and xApp at second stage, then CU at third stage and finally DU has all the dependencies resolved and becomes ready at the last stage. The length of the longest directed path in the dependency graph would determine the maximum number of stages. However, in practice, each of the Elements would take some time to become ready. To figure out the maximum latency we should consider longest weighted path in the dependency graph.

4.6.2 Configuration Management

The Manager essentially generates the configuration files for the Workload and its Applications by calling the right configurator plugins (indicated by PLG in the figure 4.7) and placing them in the shared volume between the Workload and Manager container. A configurator plugin must implement at least three functions:

1. `configure` is called via the corresponding input files to generate the configuration for the Applications in Day-1;
2. `reconfigure` is called upon detection of changes in the inputs or dependencies, it should update the configuration files and report back if it would be necessary to restart the Application via the WMI.
3. `dep-check` is called regularly to ask the plugin to verify validity of any of the dependencies it used for the configuration in Day-1 or Day-2.

The Manager listen for the Linux filesystem notifications via Shared Volume Manager (SVM) and calls for reconfiguration and restart if necessary, based on the decision in the plugins. The SVM uses Linux *inotify* to detect changes introduced in the filesystem. In that sense, the `reconfigure` call combined with SVM is the first mile towards the genuine **Service Continuity** in 4G /5G networks by enabling Day-2 actions that require no restarting of the network using specific plugins develop for each vendor. However, still restarting the Application should take considerably shorter time than restarting the container by avoiding the initialization for the containers.

The Composition Models define which configurator plugins should be used for each of the Applications in the Element. In these Composition Models, ATHENA supports various type of input sources for the configurations, including downloading them from a remote location, using a ConfigMap, or using a Secret. The plugins are called after the Manager has made all the required files locally available. The Configuration Unit would call the plugins separately for each configuration index defined in the Composition Model and passed to the Manager. The Base Operator makes sure that the Manager has all the necessary mount points and permissions for the Configurations Unit to function properly.

The Manager is shipped with a default set of plugins that are generic enough to be sufficient for most of the use cases. One major plugin that could potentially be used to implement any arbitrary configuration file is based on the Go text template engine[⁸] . Using this engine, the input files to the plugin are just templates with placeholders. Since a Go template engine is a Turing-complete language, it could be used to implement any arbitrary configuration logic via a sequence of `if` and `range` statements.

The O-RAN's alternative ATHENA configuration management approach is to use the O-RAN O1-CM interface to configure the NFs. O1-CM is largely built around the idea of static configurations that don't change frequently. Manual configurations or batch updates are too slow and can create bottlenecks. Including all the other features that we discussed in this section, it seems that O1-CM is not a competent candidate for the configuration management in ATHENA. Nevertheless, the O1-CM as a plugin for the Manager is supported to enable configuration of legacy Applications.

4.6.3 Lifecycle Management and Fault Tolerance

The Manager is essentially responsible for the lifecycle of the Applications inside the Workload container. The lifecycle control is done via simple calls such as `init`, `start`, `status`, and `stop` for the corresponding Application in the

[⁸] `https://golang.org/pkg/text/template/`

Workload container via the WMI in form of an infinite observe -decide-act loop acting on the state machine shown in the figure 3.12. Evident by the results in the section 4.9.1, this method has the maximum agility for managing the lifecycle of an Application in a containerized cloud native environment. The Manager implements a client for the WMI server in the Workload container as defined in the section 3.5.2.

The faults in ATHENA are grouped into the direct and indirect faults. A direct fault is when one of the components would crash or fail to start. This component could be either of the followings:

1. If the Application inside the Workload fails, this failure is captured via its exit code in the Frontend Scripts and then becomes available to observe via the WMI interface, where the Manager would respond by restarting the Application almost immediately (subject to the validity of the dependencies);
2. If the WMI server inside the Workload fails, since it is the initial entry point to the Workload, the Workload container as a whole would crash, being restarted by the Kubernetes (or more accurately the CRI in Kubelet via the container runtime);
3. If the Manager fails, a similar scenario to the previous one would happen, with the difference of the Manager would just silently be restarted without forcing the Workload container to restart;
4. If the Pod fails, gets evicted, or its node becomes drained or unavailable, the Pod would be perhaps restarted or rescheduled by the Kubernetes, accelerated by the Base Operator.

Probes and CPI would ascend the visibility of the faults to the Operator Plane from the Management Plane. The time to detect and resolve the direct faults are studied in the section 4.9.1.

The indirect faults are the cause for service degradation and disruption. They appear in the logs or the metrics of the workloads and are left for the Operators in the Operator Plane to handle. ATHENA does not directly react to the indirect faults, but it provides the means for the other Operators in the Operator Plane to do so via its observability stack.

In the O-RAN's traditional FCAPS model, the fault management is based on the O-RAN O1 interface. For a generic Workload, this interface seems to be irrelevant and inefficient in the cloud native context. O1 is simply too slow to adapt to the rapid instantiation, scaling, and termination of CNFs, leading to false alarms or missed issues especially considering the agility offered by the WMI. Additionally, they often rely on centralized, heavyweight monitoring solutions that can introduce latency and are antithetical to the decentralized, microservices-based architecture of cloud-native applications. Consequently, using traditional FM for CNFs can result in inefficiencies, higher operational costs, and reduced agility in responding to faults.

4.6.4 Observatory

The **Observatory** is a unit of the Manager responsible for the consumption of the events from the logging agent and Condition plugins (PLG) and expose the overall and detailed status of the Pod for probing via Container Probing Interface (CPI). It also needs to gather, group, and pre-process metrics defined for the workload and expose them to metric collectors such as Prometheus[**] using Open Metrics Interface (OMI).

The conditioner plugins implement `probe` and `metrics` function where given the parameters, they generate a related Condition object (exposed via CPI) or metrics (exposed via OMI) respectively. These plugins are used to transform the traditional monitoring information to the cloud native equivalents. The plugins are fed with the observation queries that define a source and a drain for the information with a certain data format. The plugins are responsible for the data collection and conversion, while the Observatory would periodically call them to get the latest metrics and conditions for feeding back to the CPI or OMI.

In terms of metrics, O-RAN offers the O1-PM interface. The performance indicators monitored by traditional O1-PM may not provide comprehensive visibility into the health and efficiency of cloud native functions. Furthermore, they are not compatible with the native monitoring tools used in cloud environments. The O1-PM also suffers from scalability issues due to its reliance on the traditional protocols with a centralized approach.

Accounting in ATHENA depends on observability and collecting the metrics that is coordinated by the Management and Orchestration Planes. The processed metrics are consumed by the Operator Plane to take actions such as Cost Optimization or Energy Optimization. All the metrics in ATHENA are grouped into the *outbox* and *inbox* metrics. The inbox metrics are collected

[**] `https://prometheus.io`

from inside the boundaries of the application and differ from one NF to another. Example of inbox metrics are those which are mainly collected and processed by the controllers such as RICs. The outbox metrics are collected from outside the boundaries of the application and are common for all the Applications, such as the CPU and memory usage. Some of the outbox metrics like energy or cost are composite, second-order metrics that could be derived from several other inbox or outbox metrics. ATHENA unifies these metrics to Prometheus, ready to be consumed by the Operator Plane. Similar to the Accounting Management, the Performance Management in ATHENA is based on the metrics and is collected and coordinated by the Management and Orchestration Planes.

Methods of performance or accounting management that like O-RAN O1-PM fully or partially delegate the task of the measurement to the NFs themselves are potentially biased, inaccurate, and prone to the side effects of the measurement itself. Moreover, operating legacy NETCONF-based interfaces at scale becomes inefficient and challenging in the medium to large scale cloud-native applications, due to limitations of its transport protocol and improper data models.

ATHENA does not directly involve itself in key managements, authentication, or authorization. For most cases, the orchestration plane, Kubernetes in this context, provides the means to handle these issues. However, ATHENA device management and secured, isolated, and rootless containers are the cornerstones of a secure network. Otherwise, any kind of security barriers could be breached by a faulty or malicious workload.

ATHENA introduces new metrics for the green decision-making that is are actionable insights of the network. In that regard, we define some new terms.

Definition 4.2 (Decision Projection)**.** *We define a decision projection resulted from a decision made in system is represented by the sequence of periodic observation of the interested metrics in a particular SLA.*

For example, if we consider the DL goodput and the RTT as the metrics of an SLA, any sequence of pairs (t, r) of non-negative real numbers could be potentially a decision projection. For simplicity, in the rest of the section we assume one-dimensional projections, i.e., only one metric is in question for the SLA. Multidimensional projections should be a straight extension of the discussion.

Definition 4.3 (Quality Reduction Metric)**.** *Quality Reduction Metric (QRM)* *metric is defined by the average of point-by-point ratio of an alternative decision projection D_a over the null decision projection D_0.*

To be more verbose, if $D_0 = (d_{01}, d_{02}, \cdots)$ and $D_a = (d_{a1}, d_{a2}, \cdots)$, then we have the equation 4.4, where $D_* = (d_{*1}, d_{*2}, \cdots)$ is the SLA desired values.

$$\text{QRM} = \sum_{i=1}^{n} \frac{d_{0i} - d_{ai}}{d_{*i}} \tag{4.4}$$

Since QRM is defined as a ratio, it is a dimensionless number with no unit of measurement. Thus, the values over several dimensions of the decisions could be easily averaged over the dimensions to form a single metric that presents how bad the alternative decision is.

Definition 4.4 (Power Saving Score)**.** *The Power Saving Score (PSS) is defined by the average of point-by-point ratio of the null decision projection D_0 over the alternative decision projection D_a, where the metric of interest is the power consumption.*

If we define the power metric equivalents of D_0, D_a, and D_* as D'_0, D'_a, D'_* then we have the equation 4.5.

$$\text{PSS} = \sum_{i=1}^{n} \frac{d'_{0i} - d'_{ai}}{d'_{*i}} \tag{4.5}$$

The power replaced as the metric is relative, without considering the base usage of the system. PSS is dimensionless as well. Consequently, we could adjust the balance between the QRM and PSS in a meaningful manner.

In most cases, the QRM needs to be normalized to give a more sensible view of the quality reduction. If we define the estimated ratio of affected users of a group of interest as the Affected Users Ratio (AUR), then the comparison should be made between QRM × AUR and PSS. For example, if the group of interest is the users of an operator in a town, then the AUR is defined as the ratio of the average number of the users experiencing the quality reduction over all the population of the users in the town. The metrics presented here are with simplified assumptions to demonstrate how the Observatory works. Extensive studies using similar yet more sophisticated and accurate metrics given on other works, for example [47] now could be verified with ATHENA on real physical setups rather than mere mathematical studies.

4.7 Devices, Networks, and Volumes

In TRIREMATICS, GAIA is responsible for completing the Orchestration Plane and provide the utilities demanded by the other Planes. In that regard, GAIA unifies the device management for telco devices in a cloud native manner. What makes this device management particularly outstanding is its level of automation and consecutive isolation that it actually brings to the access network workloads. The abstraction in the device management is based on the device capabilities to make pools of homogeneous devices that are distinguished by their geographical locations.

GAIA defines addressable and assignable resources for any form of device resources such as radio devices (whether hot-plug, or network-based, or PCI-based) or accelerators (e.g., GPU, FPGA). These resources are exposed as Kubernetes node resources via a device plugin and could be allocated to any of the Pods upon request. Defining a device plugin improves the isolation and the security of the containers by following the Kubernetes security structure, and on the other hand opens up a whole spectrum of possibilities for the logics to be implemented in the Operator Plane.

For a device to be allocated for a container, it should be requested in the Pod description. ATHENA takes care of defining the device requests and limits that are processed by Kubelet on the nodes containing the right node resources to allocate them for the containers. A device plugin communicates with Kubelet on a UNIX socket exposing the devices that are made available via systemd, as shown in the figure 4.9. Systemd is the most-commonly used init daemon across all the Linux distributions which comes with a device management mechanism called *udev*. Given specific rules to udev, we govern the naming, management, and initialization of the devices on the nodes, to achieve a harmonized representation of the resources. Later the GAIA device plugin automatically and dynamically detects the devices and advertise them as node resources in Kubernetes.

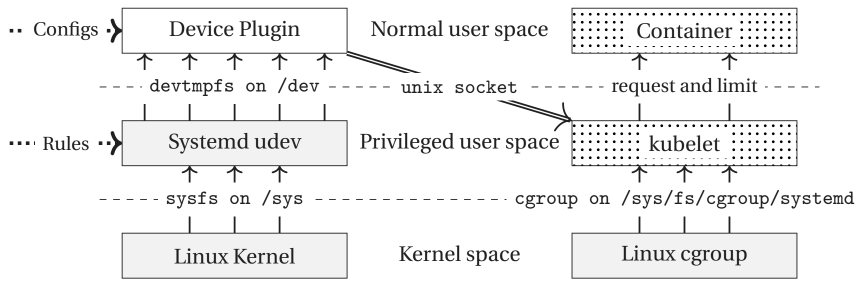


FIGURE 4.9: Device management stack in GAIA based on udev.

The physical devices in GAIA are transformed to logical software devices by

Linux systemd and udev in the smallest operational units for each device. These units are categorized by their capabilities into homogeneous sets of devices. The compatability of the devices with the NFs are left for the vendors to decide in the Composition Models. Similar to the NUMA topology, we define neighborhood metadata for the devices that are used in the container scheduling. The neighborhoods could refer to the physical connections between the devices, such as belonging to the same physical device, sharing the same PCI bus, and having the same RF ports and antennas. They could also refer to the geographical proximity or synchronization.

Two particular set of devices are in our interest: First, the radio unit devices such as USRPs or SDR cards that could be hot-plugs (through USB, like UHD-B2xx), on-the-network devices (through Ethernet or SFP, like UHD-N3xx or AW2S), or on-the-board (through PCI or PCIe, like Amarisoft SDR50). Secondly, the network terminals in particular Quectel LTE modules connected either through USB or PCI. For each of the classes, specific rules are defined to name the devices in a consistent manner, regardless of the kernel version or the device ports that they are connected to. For the network terminals in particular, the devices are named after the IMSI of the SIM card in their slot, hence the UEs are uniquely identifiable across the cluster. At the same time, each network terminal is also advertised as a unified resource to provide a pool of UEs.

GAIA also detects the extra devices that are listed in the Composition Models automatically during the deployment and exposes them via the same interface. It should be noted that radio devices are defined in the Network CR not the Composition Model CR following multi-x definition. The Manager would configure the corresponding Workloads to operate on the specified device.

ATHENA utilizes the Multus plugin to support multiple network interface association with the containers, including the SR-IOV plugins for faster networking. In the Network CRs, for each network protocol interface such as Xn, F1, E1, or NG, one could use a different Multus network attachment definition name that should be configured externally. In addition to Multus, the Managers on the Management Plane recognize and respect these network interfaces and during the distributed resolution processes for `depends` and `resolve` API calls, they respond with the proper interface addresses.

Moreover, we have used cutting-edge networking technologies in the cloud such as Calico [††] to establish incredibly fast data planes while advertising

[††] `https://projectcalico.docs.tigera.io/about/about-calico`

pod IPs using BGP. In this way, ATHENA workloads are allowed to communicate effectively and efficiently with physical deployments or workloads external to ATHENA domain. We also use Calico's eBPF dataplane to speed up the networking and reduce the CPU usage while providing the monitoring capabilities for the Operator Plane.

To further improve the performance, ATHENA supports NUMA topologies on the cluster when it is assigning the computing resources to the workloads. The NUMA regions are selected carefully by the Base Operator to optimize caching. Also Linux eBPF is used to monitor and control the network traffic and energy consumption of the workloads without a need for a proxy, a sidecar container, or any other invasive or intrusive methods for that purpose. In particular, we use KEPLER[‡‡] for energy monitoring and control.

4.8 Day-2 Operations

In this section we discuss the Day-2 Operations in ATHENA and how they are supported by the Management Plane and Operator Plane.

4.8.1 Observability, Auto-healing and Idempotency

The declarative design of ATHENA avails the Base Operator of auto-healing properties, i.e., whenever an Element or Pod is deleted or evicted, it would reconsider the topology and adjust the associated parameters. The healing in action is merely repeating the deployment, since the control loops of ATHENA are designed to be **idempotent**. Thus, the effect of applying them repeatedly or under failure should result in the exact desired state.

In ATHENA, we incorporate the ReadinessGate feature of Kubernetes[*] to handle complicated cases of readiness. This feature is designed to be used for events that are not resolvable directly by the Pod itself, like readiness of an external volume or cluster job. However, in telco workloads, readiness is a multistage event which might not necessarily be expressible as a binary readiness status. The readiness could rely on various internal state changes that are reacted to by the other elements as a dependency. For example, an arbitrary DU might incorporate different threads to handle F1 and F2 connections, and while not ready for serving the UEs over radio, it could be ready for F1 setup. Thus, exposing different readiness levels not only improves the observability

[‡‡] `https://github.com/sustainable-computing-io/kepler`

[*] `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle.`

but helps the Management Plane to handle complicated, multistage dependency matrices. We have overloaded this capability through Base Operator custom probes. The custom probes defined for the container in the Composition Model are individually probed by the Base Operator and *patched* as a Kubernetes Condition resource to the Pod.

As we discussed earlier, ATHENA categorizes the metrics into the inbox and outbox, where the inbox metrics are collected from the interior of the boundaries of the application, while the outbox metrics are collected from its exterior. Each of those metrics could be either in a local or global scope, depending on how many NFs are involved in that metric. There are six producers of the metrics in a typical Open RAN scenario in ATHENA.

E2 Metrics These inbox local metrics are collected via the E2 interface from each individual NF or E2 node in this context. A specialized monitoring xApp targets each of the E2 nodes and subscribes for various monitoring service models such as O-RAN KPM and caches the collected metrics in a small local storage. The xApp gradually pushes the pre-processed metrics as a time series to the database backend of Prometheus, where they are stored for further processing or visualization using Grafana[†] . Since the xApp might have significant amount of data to push, it uses compression techniques to minimize the network traffic. In that sense, the bottleneck of the E2 metrics collection is the network bandwidth between the xApp and the RIC. Even though all the metrics would scale by the number of NFs, the E2 metrics would remain as the dominant size.

Manager Observatory The Manager's Observatory is tasked to perform local outbox measurements that are scraped by Prometheus (pull-based) and they finally would end-up in the same backend data lake. Due to their pull-based nature, the minimum interval between the scrapes is 1 second.

Base Operator The Operator Plane has a global scope and the Base Operator exports some metrics about the topology of the network and its composition via Prometheus pull-based mechanism. These metrics are also considered outbox.

Non-RT RIC Unlike the metrics collected from E2, Non-RT RIC may export global inbox metrics that are later scraped by Prometheus. All the metrics are merged into the same backend and grouped by the NF identity.

[†] `https://grafana.com`

eBPF In Athena, we use eBPF to collect some local outbox metrics from the NFs mainly concerning with their energy consumption. Also, eBPF is used to generate global outbox metrics based on the network traffic. The eBPF metrics are scraped by Prometheus and stored in the same data lake.

Node Exporter The Node Exporter is a Prometheus exporter that collects metrics from the underlying host machine. The process of collecting metrics is the default Prometheus implementation. The data are mixture of cAdvisor[‡] and Kubernetes metrics server[§] . The metrics from the Node Exporter are considered local outbox.

4.8.2 Reconfiguration, Upgrade, and Immutability

Each CR has mutable and immutable constructs. Changes in structure of the containers, such as their image or resources as well as changes in the radio device or identity of the networks would result to recreation to preserve the immutability of the setup. Keeping the building blocks immutable is crucial to make scaled instances consistent and predictable [48]. We call such actions an **upgrade** and Athena minimizes the down-time of the NFs during an upgrade as proven by example in the section 5.2. The agility of Athena and containers combined is the key to achieve the low down-time. However, a **reconfiguration** action, triggered by changes in mutable constructs are handled by the distributed processing of the Management Plane with zero down-time. An example of changing the network topology and its effect on the service quality is given in the section 5.2. Reconfiguration in day-2 is of paramount importance in telco to achieve service continuity and flexibility at the same time.

4.8.3 Micro-decisions and Agility

The Manager in Athena provides the support for performing micro-decisions including, but not limited to, short-term green optimizations via WMI.

Definition 4.5 (Micro-decision)**.** *A micro-decision is a decision that is concluded and intended for a short period of lifecycle and could be invalidated and overwritten by another decision in a short time.*

These micro-decisions could be used to perform short-time optimizations, in particular significant in the green computing [47], [49]. To elaborate on an example of micro-decisions, we have considered an extension to the WMI with

[‡] `https://github.com/google/cadvisor`

[§] `https://github.com/kubernetes-sigs/metrics-server` two new API calls. A `freeze` endpoint in the WMI intends to freeze the corresponding workload. By default, it is pointing to the `stop` endpoint internally, though each vendor could provide customized actions that would not necessarily stop the instance, but just put it in the sleep or freeze mode. Likewise, the `thaw` endpoint implements the awakening action and by default is pointing to `start`.

4.9 Evaluations

We have completely implemented the mentioned design of ATHENA at all layers. The code is mostly written in Golang with over 10k lines of code for the Operators in total, 3k lines of code for the Manager and its plugins, and another 3k for the extensions to orchestrator. For the evaluation of ATHENA, we have used this implementation and onboarded Amarisoft (AMR), OAI, and Software Radio Systems (SRS) already as vendors. The cluster under the test contains 9 machines with Redhat Enterprise Linux 8, CentOS 8, Ubuntu 20.04, or Ubuntu 18.04 installed on them. The radio devices supported on the cluster are USRP B210, USRP N300, AW2S RRH, AMR SDR50, and AMR SDR100.

4.9.1 Lifecycle Improvements and Agility

To simulate the effects of a failure and analyze how the *observe-decide-act* loop of the Manager would behave, we have onboarded OAI gNB workload on ATHENA then performed the following experiments:

1. Stop the RAN process inside the container to simulate a workload failure where in ATHENA, the Manager would detect the issue and act accordingly by restarting the application process;

2. Stop the main process of the container to simulate a full-scale container failure where Kubernetes intervenes to detect the error and then goes through crash loop back-off that takes considerable amount of time for recovery;

3. Finally we stop the Pod's sandbox container to simulate an overall pod failure that causes all the containers in the Pod to restart.

As reflected in the figure 4.10, ATHENA approach is considerably faster on each part. Besides, the main process of the container is chosen carefully to be the stable BOSUN API server for WMI that exhibits very low chance of failure, hence no transition to the longer recovery cycles of Kubernetes. It should be noted that time for observation is composed of the minimum number of probings with the standard period of one seconds each to reach the conclusion about the failure. Killing a container causes a faster detection since it would be triggered with one failed probe but for the Pod the minimum failed probe becomes two. Decisions or acting on the containers takes longer than pods since Kubernetes categorizes the container failures as application failure and goes into crash loop backoff, but a pod failure is counted as of Kubernetes and is recovered faster. This comparison is not to show Kubernetes is slow, but to indicate in this particular setup, ATHENA is able to provide more agile recoveries compared to the generic-purpose vanilla Kubernetes.

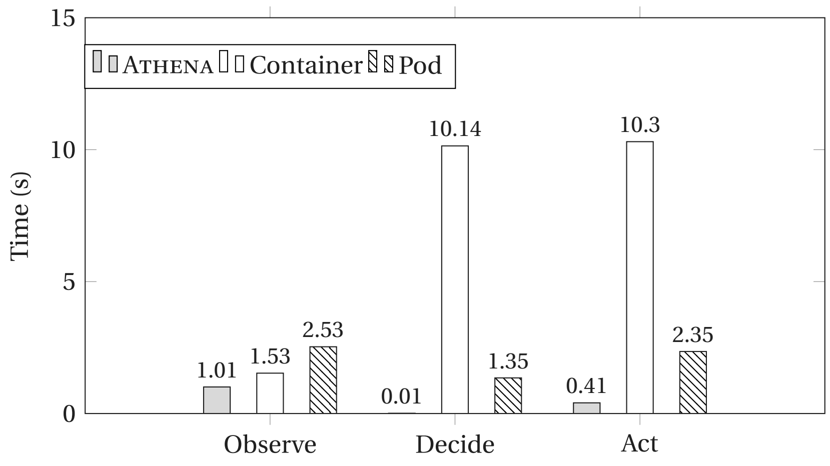


FIGURE 4.10: Comparison of observe-detect-act cycles in ATHENA and vanilla Kubernetes.

To reflect on the agility of the deployments in ATHENA, we have done a concise analysis of lifecycle of an E2E deployment of CNFs on both ATHENA and ETSI-OSM [14] release 11. This data is gathered in the table 4.4, where one could observe ATHENA is far more agile than the OSM. The data is averaged over several deployment scenarios with the same Kubernetes version (v1.24). The two clusters have the same machines running Ubuntu 18.04.1 (kernel 5.4.0-107-generic) over 8 virtual CPU cores of Intel Core i7-8550U, at 1.80 GHz baseline, scalable upto 4 GHz and with RAM size of 32 GBs. The data is collected by the timestamp of the corresponding objects or logs. Since Kubernetes does not record timestamps of shorter than one second, for some actions in ATHENA, we could just give the upper bound of 1 second.

We have performed the comparison with Charmed deployment of OSM and for onboarding a Charmed unit to have a similar structure of MANO. We have taken the OSM control plane components such as LCM, VCA, or RO as Operation Plane and the Model Operator and the Charms as the Management Plane. The data is collected for onboarding a single CNF workload, regardless of image pull times, and after both Operation Planes running and ready.

Phase	Delay in OSM	Delay in ATHENA	Improvement
	MANO Gen. #2+	MANO Gen. #4	
Operator to Pod Creation	4 second	< 1 second	**75%**
Management Tasks	13 seconds	3 seconds	**77%**
Status Propagation	< 1 second	< 1 second	-
Day-2 Operations	5 seconds	2 seconds	**60%**
Deletion and Clean Up	52 seconds	5 seconds	**90%**

TABLE 4.4: ATHENA and OSM timeline compared.

On deletion phase, OSM uses long graceful shutdown timeouts by default (30 seconds), but in ATHENA the API server gracefully terminates the process internally by detecting the terminate signal from Kubernetes, hence it does not need to wait for the timeouts. Note that, for the numerous iterations of a deployment, the ratio is more important than the difference, because in large number of sequential deployments the small differences grow to a lot.

4.9.2 Performance and Overhead

To form a conclusive image of our platform, we demonstrate the E2E 5G network throughput UDP and TCP throughput achievable in ATHENA, compared to a fully PNF deployment on bare metal and a deployment with Snaps, shown in the figure 4.11 for OAI on B210 (SISO), AMR on SDR50 (MIMO 2x2), and AMR on SDR100 (MIMO 2x1). All the tests are performed on the same machine running RedHat Enterprise Linux 8 with 12 cores of CPU of type Intel Core i9-10920X at 3.50GHz base frequency and 64GB of RAM. The figure shows the data OAI gNB monolithic as the RAN and minimal deployment model of OAI 5GC (only AMF, SMF, and UPF) as the CN, where the UPF is deployed using OAI SPGW-U module, all on containerd as the container runtime. The configuration of the RAN is the same over all the setups: 5G-SA FR1, 106 PRBs, 40 MHz bandwidth, TDD band 78 with pattern of 7DL:2UL slots and 6DL:4UL symbols. For the FR2 setup we used 5G-NSA where the LTE cell is configured with 20 MHz bandwidth in FDD band 66 and the NR cell is configured with 100 MHz bandwidth, 120 kHz subcarrier spacing, and TDD band 261 with the pattern of 3DL:1UL for the slots 10DL:2UL for the symbols. In this case the results show the aggregated throughput of the two cells that are running with MIMO 2x1. The data shows negligible difference between the throughput on different setups and the slight variation could only be caused by minor aerial differences.

Plane	Resource	ATHENA		Charmed OSM	Improvement
		REQUEST	LIMIT		
Operator	CPU	10m	500m	151m	**93.38%**
	Memory	64MiB	128MiB	1.8GiB	**96.45%**
	Image Size	-	61.3MB	719.97MB	**91.49%**
Management	CPU	10m	30m	135m	**92.59%**
	Memory	15MiB	64MiB	50MiB	**70%**
	Image Size	-	145MB	183.52MB	**20.99%**
Orchestration	CPU	10m	50m	100m	**90%**
	Memory	30MiB	64MiB	2.67GiB	**98.88%**
	Image Size	-	21MB	694.30MB	**98.88%**

TABLE 4.5: Overhead comparison between ATHENA and Charmed OSM.

We have evaluated the total resource usage of ATHENA Manager and Operators (Base, Terminal, and Slice Operators all together), as the table 4.5. Doing some calculations will lead us to the estimated energy consumption of ATHENA as the MANO energy overhead in long term. Of course this overhead would scale linearly with the number of deployed networks, but the constant factor is small enough to be negligible compared to the resource and energy usage of the network components themselves. Gathered in the table 4.5, one finds a comparison between Charmed OSM and ATHENA in terms of the computing resources, as well as the corresponding total image sizes. Unfortunately, OSM does not define limits for its resource usage, so we relied on K8s top command. The limits defined would ensure ATHENA cannot consume beyond the defined resources by which it receives improved QoS control in Kubernetes. For OSM we have considered the Model and Charmed Operators, LCM, PLA, and POL as the necessary parts of the Operator Plane, while taking into account an average for Charmed Operators for workloads as replacement for the Management Plane, The observability stack of OSM is considered optional for fair comparison. It contains Prometheus and Grafana which are available for ATHENA as well but are not considered as our contribution to the observability stack. The RO, databases, messaging system (Kafka and ZooKeeper), and authentication management (Keystone) are counted as part of the Orchestration Plane. We have not counted in the extra overhead of having Juju, LXD, and MicroStack installed as underlying parts of the Charmed OSM setup, but ATHENA needs nothing more than a minimal Kubernetes stack to Operate on top of it.

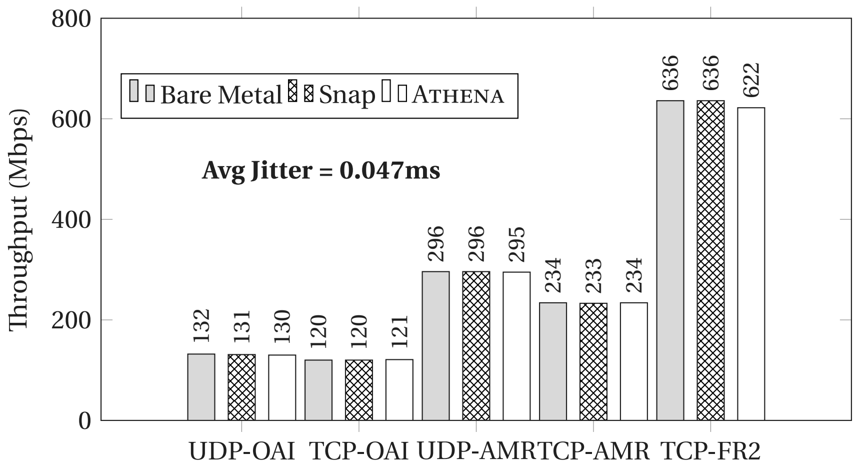


FIGURE 4.11: Comparison of the throughput between bare metal, Snap, and Kubernetes deployments of TRIREMATICS.

Chapter 5

Use cases

Though TRIREMATICS design model encompasses the support for numerous use cases in 4G, 5G, and beyond, we emphasize on three use cases foreseen in 5G and beyond, which are in particular attention of TRIREMATICS too.

5.1 Private Networking and Optimization

To demonstrate the capabilities of a platform for private networking use cases, one should consider at least the following three substantial domains:

- Network **Customization** by defining customized, multi-x stacks with possibly network sharing;
- Cost and Energy **Optimization** across variations of deployment decisions, constrained by the desired Key Performance Indicators (KPIs);
- **E2E** capabilities including observability and automation with the UEs in the loop, since in many cases, the UEs would be deployed and controlled by the owner of the private network.

In common use cases in 5G Private Networking, such as Industry 4.0, a specific range of the UEs would be deployed by the owner of the network itself, unlike the traditional public networks where the UE belongs to the end customer and is out of the deployment loop. Furthermore, the DevOps platform introduced in the chapter 3 enables consistent network customization by *patching* the artifacts.

To further develop these ideas, we consider an illustrative example of a private network operator. Consider the scenario of an event in a museum to exhibit recently found relics for a short period of 10 days in a town. The network operator wishes a **short-lived** private network to serve some augmented media to the visitors, demanding 80 Mbps downlink UDP payloads inside the venue. The goal is to optimize the cost of the deployment while trying to respect the regulations imposed by the city hall for the energy efficiency with 50 Watts power budget per base station. Peeking the data from table 4.4 and 4.5, it can be seen that the overhead incurred by ATHENA and the agility of its operations makes it suitable for short-lived networking including network leasing and temporary service boosts following the user demands. Measuring the agility of ATHENA on deployment of a simple, E2E network including the UEs, we have observed that the UE gets network service in less than 2 minutes in total, yielding around 0.01% of the total time of the service lifecycle in this example.

Due to its declarative design, ATHENA simplifies tailoring and enables customization of the networks. This includes network sharing capabilities as well as relying on the external network components, outside the ATHENA domain of control. Network sharing in ATHENA happens in two levels: Network-level and Composition-level. At the network-level we address the Multi-Operator Radio Access Network (MORAN) and Multi-Operator Core Network (MOCN) scenarios while at the composition-level, how the internals of an access or core network should be scaled based on the sharing options in accordance to the slice definitions. This would ultimately result in other sharing options like sharing DU or CU in RAN as demonstrated earlier in the section 4.5. One could also incorporate RAN or CN instances that are external to the cluster or alien to ATHENA, but it should be noted that 3GPP standards should still be respected by the deployer and ATHENA would not enforce a connection that would otherwise be rejected in the standard. Events are issued in these cases, and the element status could change to the failed status, depending on the vendor's preference.

ATHENA Cost Optimizer Operator is an optional Operator on the Operator Plane which given a Service-Level Agreement (SLA) object as well as cost constraints, it would iterate on various options in the network deployment to find the minimum cost deployment that still satisfies the SLAs. The Composition Models are incorporated with metadata that is particularly useful for pricing and cost optimization. The DevOps platform generates parts of this metadata as how many resources were consumed to produce the images in that particular Composition Model. The total cost of a network then is calculated by simple summation over the cost of each Composition Model used. These would be computed into the CapEx while the OpEx is estimated by the resource consumption over a period of time.

As an instructive example, consider a private network operator who intends to set up a network chosen from one of the four setups as provided in the table 5.2, with the minimum monetary cost. Indisputably, many more setups and variations over different dimensions could be explored and compared in the same manner via ATHENA. Given the scenario, Cost Optimizer Operator sweeps the four options with a tolerable range of $[0, 100]$ Watts for the power usage (twice the regulation, but still fair penalty) and $[0, 2\%]$ for maximum packet loss to improve the QoS. Noticeably, as seen from this example, the Cost Optimizer Operator defines hard constraints and soft constraints, allowing exploration of the options outside the original SLA space. Since essentially the SLAs themselves indirectly translate into the generic cost-revenue problem, their violation as long as it yields a better total revenue over the cost, is acceptable. Hence, for example, if allowing power consumption out of the valid regulation range, enables usage of much cheaper radio equipment, perhaps the total cost of the radio plus the penalty would still be less than the cost of the more expensive radio equipment.

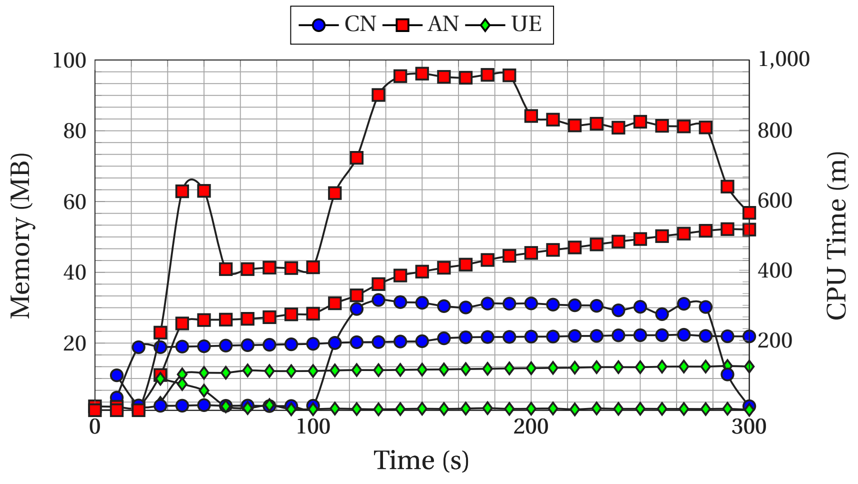


FIGURE 5.1: CPU and RAM usage of option 1 in table 5.2. The CPU time does not have a unit, and it is measured by how many miliseconds of CPU time are consumed by the container at a 1-second interval. Hence, the normalized concumption would reach the maximum value of 1000 as visible from the graph.

Using its observability capabilities on top of Prometheus over the OMI, essentially ATHENA gets the computing resource consumption of the deployed scenarios in real time. Each scenario for the Cost Optimizer Operator is unique, hence a set of pre-defined, static measurements would not be helpful more than giving some approximate intuitions in general. The figure 5.1 shows resource consumption of the first option in the table 5.2, where the CPU consumption is summed over the 10 CPU cores of type Intel Core i9-10920X at 3.50GHz, and averaged for intervals of 10 seconds for smoother plotting. Then the overall cost of the scenario would be calculated via an integral over a standard interval of 5 minutes, where the last 3 minutes have a single UE deployed transmitting on 80Mbps downlink via iperf3, placed in a Faraday cage.

The integral would include the pricing model specified to the Cost Optimizer Operator as possibly non-linear function. In general, if we consider a cost function $C_r : \mathbb{R} \rightarrow \mathbb{R}^+$ for a time-varying resource $r(t)$, the cost of the resource over a period of time $[0, T]$ would be calculated as the equation 5.1.

$$C_r = \int_0^T C_r(r(t))dt \tag{5.1}$$

In this terminology, C_r does not need to be linear, but it should be a monotonically increasing function. The majority of current cost models used by the public cloud providers are linear, described by a mere constant coefficient. Hence, in this model, the function C_r is of the simple form of $C_r(x) = \alpha x$. However, we expect in the near future, non-linear cost models, which penalize higher usage of, for example, energy, would replace the traditional models to encourage more sustainable and green computing. For the sake of simplicity and proper reference, we have used Google Cloud Platform (GCP) pricing policy as an example reference in the table 5.2, by extracting the cost of the resources as given in the table 5.1.

Resource	**Cost**
CPU	\$0.0108/h
RAM	\$0.0044/h
Disk	\$0.0004/h
Network	\$0.0001/h

TABLE 5.1: The linear pricing model of the Google Cloud Platform (GCP) as an example reference. The values are defined as USD per hour, however, before being used in the integration of the equation 5.1, they have to be converted to USD per second to match the time interval used for the approximation of the CPU resource consumption.

In the table 5.2, the CapEx is calculated based on the cost of the hardware and the software licenses. The average OpEx reflects the average cost per hour for the entire 5-minute duration, whereas the peak OpEx only considers the portion of the iteration where the UE is active. The power measurements are collected by the Cost Optimizer Operator from a digital Watt-meter deployed on the machines and connected to their power supplier, and it factors out the idle usage of the machine. It should be noted that radio devices consume power even on standby; hence the idle power is measured with no cards or devices connected to the motherboard.

#	Setup	CapEx (k\$)	Avg OpEx (\$/h)	Peak OpEx (\$/h)	Power (W)
1	AMR SimpleRAN + SDR50	15	73.04	91.87	8
2	AMR SimpleRAN + B200	14	100.25	112.54	9
3	OAI Monolithic + B200	2	141.18	171.20	10
4	OAI O-RAN + AW2S	8	211.69	235.10	90

TABLE 5.2: Example of 4 iterations for Cost Optimization. The costs are calculated in the 5-minute interval and then extrapolated for an hourly cost.

The total cost C for the whole n days of deployment could be then simply calculated via the formula in the equation 5.2.

$$C = \text{CapEx} + n \times 24 \times \text{OpEx} \tag{5.2}$$

where 24 is the number of hours per day, and the units are taken from the table 5.2. This gives the cost for a single node, however; considering the range of each deployment and the space to cover, the number of the nodes might change which is left to the network operators to decide. As a naive approximation, one might consider that the cost should perhaps simply be multiplied by their count.

ATHENA Cost Optimizer Operator closes its loop by involving ATHENA Terminal Operator through the Operator Plane via injection of a Terminal object, which also provides the required live testing utilities. This triggers the Terminal Operator to inject the UE identity to the CN database, create the Element, and provide the required testing facilities to measure KPIs, including Round-Trip Time (RTT) and throughput. Reacting to the corresponding event that Terminal Operator would issue after successful attachment and setup of the UE to the network, the Cost Optimizer Operator starts measuring the KPIs of the E2E deployment. In result, we have the figure 5.2, where the best choices are indicated by their numbers in the table 5.2.

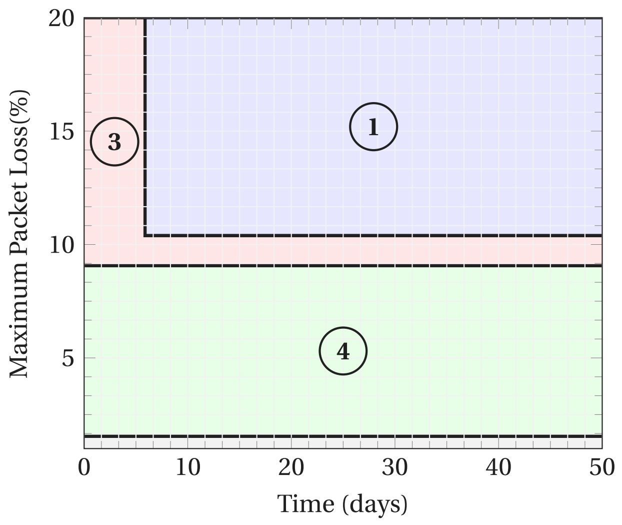


FIGURE 5.2: If the minimum cost is calculated, considering the maximum packet loss, the dominant best option would be the fourth for the ranges below 9%, while for higher than this value, depending on the number of days, the best choice would change from the third to the first. The small white region at the bottom of the graph is an infesible region, unachievable by any of the options.

The figure 5.2 shows the regions of optimization for the packet loss criteria using the formula in the equation 5.2. The temporal dimension is important in this kind of region graphs, since it helps to flatten the CapEx over OpEx in the long term. Instead of producing just the single-point result for $n = 10$, the Cost Optimizer Operator produces these region graphs. As a result, new options could be explored by the operator, for example, if the event is extended to 20 days, the best choice would be changed from the third to the first option. This example clearly demonstrates how the novel designs in TRIREMATICS open up new possibilities that have never been considered in a traditional problem definition.

On the figure 5.3, the plot presents the variation of best option based on the cost of average usage (60% of time in the peak) depending on the period of deployment and power consumption constraints. On the right, we considered the packet loss reported on the iperf3 measurements with the cost of deployment on peak usage. Due to the given time period and SLAs of our scenario, the best choice for a single node setup is to have the first option which could guarantee less than one percent packet loss. It has to be noted that the conditions and scenario are relaxed for the sake of simplicity, but the procedure is valid for any arbitrary scenario.

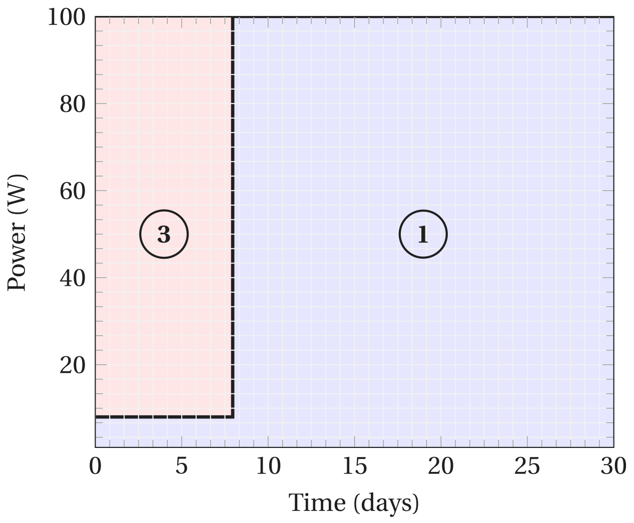


FIGURE 5.3: If the minimum cost is calculated, the choice would be down between the first and third options, depending on the number of days.

It is worth noting that ATHENA is reproducible and consistent by design, hence the whole process of each iteration could be completely automated. Otherwise, the results of the experiments could not be trusted. Besides, the agility of the platform indicated in the table 4.4 is crucial since the deployment time would determine the time for each individual iteration, yet still the reconfiguration capabilities of ATHENA could be exploited to reduce deployment time from two highly correlated network deployment options. Unlike the deployment of the final result, for each iteration of the test, almost 30% of the time is spent for the UE to get fully attached and ready to transmit. This then brings the question of how the arbitrary periods of 5 minutes and 3 minutes used in this example would be chosen in practice. One option is to let the Cost Optimizer Operator continuously calculate its estimations in table 5.2 until they converge to a stable value. Hence, the experiment continues as long as the calculated hourly cost values based on the interpolation of the data are unstable, capped by the maximum of actual one hour.

5.2 Open RAN and Emerging Networks

Open RAN is an initiative led by operators to break from vendor lock-ins and mix-and-match components of the network, even beyond the 3GPP specifications through the newly defined open interfaces. We have built the idea of multi-x as the extension to the same doctrine mixed with the fundamental goals of cloud native, which goes beyond the RAN itself to the MANO/OAM. ATHENA defines open interfaces in between its components that allow them to be replaced at any time with any customized implementation of the same API, making its design fungible. We need to emphasize that simply using open source tools does not correspond to an open interface multi-x solution, since not only all the open-source licenses could be applied to commercial applications, but also there could be preferences to use one solution over the other, simply due to relations of their maintainers. ATHENA is agnostic to the cloud provider, Kubernetes distribution, container runtime, or OS. Taking ETSI-OSM again as an example, one immediately notices the maintainers of OSM have taken irreplaceable design decisions, for example, in picking how their DevOps stack should be deployed.

Aligned with the Open RAN, as a cloud-native O-RAN OAM, ATHENA supports concurrent deployment of Open RAN components as well as dynamic transition between any of the options for deploying the RAN with minimal downtime and human intervention. To illustrate this, we have composed a scenario in which ATHENA progressively deploys a complex network of varying vendors and splits in multiple requests to resemble a growing network of an operator. The figure 5.4 shows how the CPU usage of the whole cluster as an example of computation resources changes during the growth of the network and its correlation with the scale of new workloads introduced. Based on the number of Elements and networks deployed in each phase, we have done an extrapolation of the resource consumption.

In the figure 5.5, we have tried to perform a linear regression on the data points of the figure 5.4, to predict the resource consumption based on the number

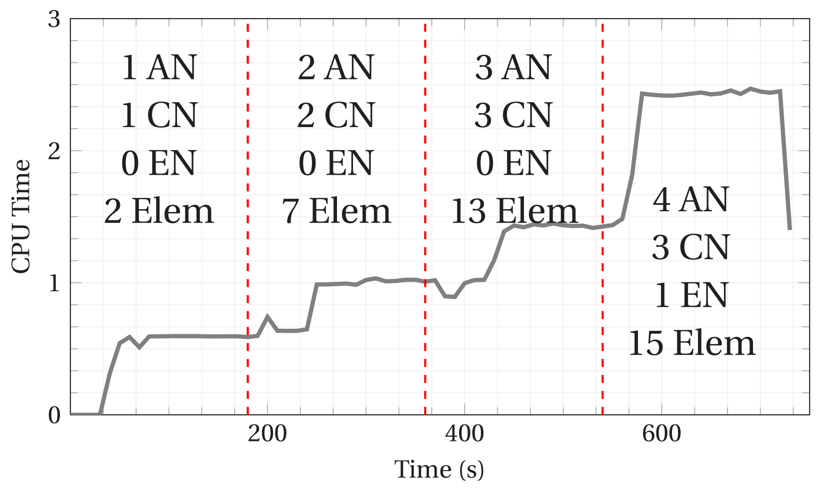


FIGURE 5.4: The CPU usage of the cluster as an example of computation resources changes during the growth of the network and its correlation with the scale of new workloads introduced. 'AN' stands for Access Network, 'CN' for Core Network, and 'EN' for Edge Network.

of Elements and networks. The results have higher correlation score with Elements, due to the fact of variations in the network compositions and vendors used in the setup. This means to predict and provision resources for an Open RAN deployment; one should rely on the number of Elements to be deployed rather than the number of networks. The R^2 value for fitting the data linearly to the Elements and networks are respectively 0.9876 and 0.9672, which for these sample sizes are acceptable[13]. For the figure 5.6, we have used OAI RAN in RF simulator mode, and in intervals of 5 minutes, one 5G SA RAN and UE simulators have been introduced. The prediction has been made based on the number of elements which indicates higher non-linearity, despite the larger sample size.

[13] One expects R^2 nataurally increase with higher number of samples

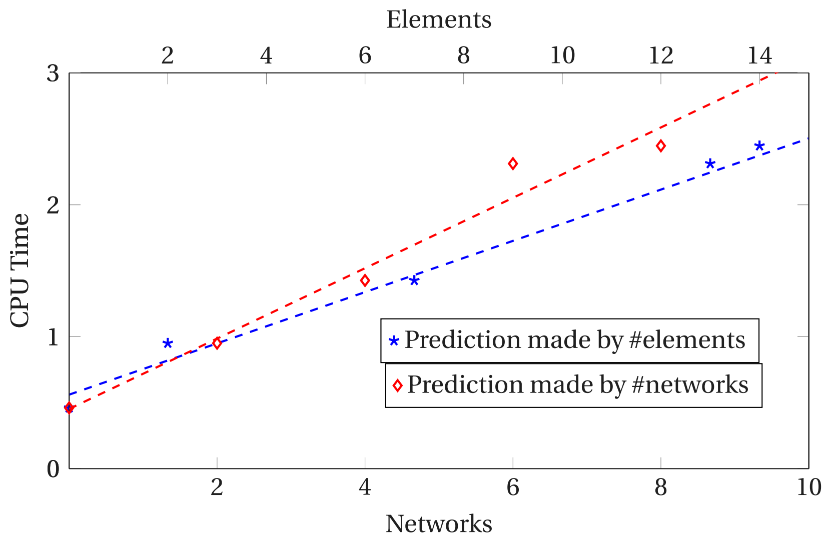


FIGURE 5.5: Linear regression on the data points of the figure 5.4, to predict the resource consumption based on the number of Elements and networks. CPU time in this figure essentially translates to the number of CPU cores used by the containers.

We hypothesize in the Open RAN scenario depicted in the figure 5.4 the variations offered by different deployment models and vendors have absorbed some uncertainty caused by the distribution of nodes in the cluster. However, in a more homogenous scenario of the figure 5.6, other factors become more dominant. On the other hand, in the figure 5.7, we have adjusted the Pod scheduling via ATHENA Mobility Control Operator to place the simulated UEs on the same machine with the simulated gNB. This has drastically changed the resource usage regime into a highly linear but significant CPU time. The RTT experienced by the user drops from an abnormal amount of 88ms to 17ms on average, showing the effect of placement on the QoS as well. In conclusion, a multi-x platform is the key enabler to unlock the Open RAN potential not only for deployment of tailored networks, but also for performing valid and reliable measurements.

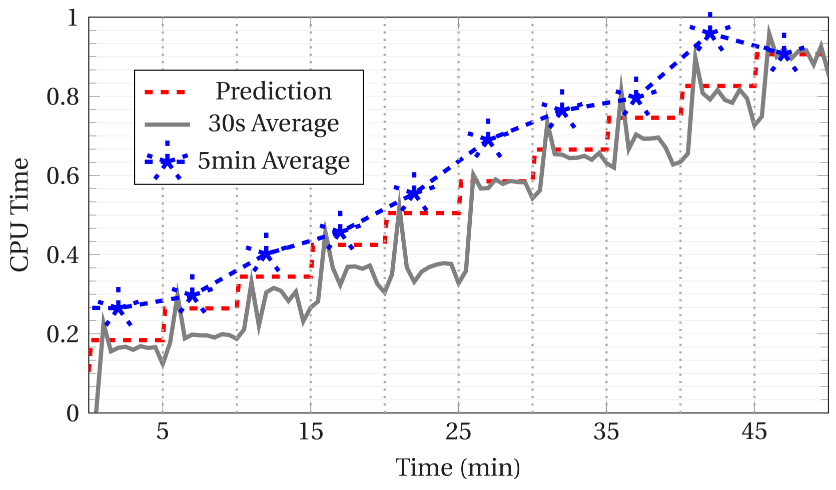


FIGURE 5.6: The CPU usage of the cluster as an example of computation resources changes during the growth of the network. The figure also shows the predictions made for the amount of resources based on the number of Elements and the averaged CPU usage over a period of 5 minutes.

With these examples of predicting resource consumption, we have arguably demonstrated the capabilities of ATHENA to be used as a platform for Open RAN and emerging networks. ATHENA allows the operators to predict and provision the right amount of resources even for the heterogeneous ecosystem of Open RAN. Furthermore, the support for multi-x and multi-vendor deployments, as well as the ability to perform valid and reliable measurements, makes ATHENA a suitable platform for Open RAN.

These progressive growth examples are not the only ways an Open RAN network could evolve. A large set of those evolutions are grouped on the term of Day-2 operations, which are the operations that happen after the initial deployment of the network. To reflect on the day-2 operations of ATHENA, we present two operations in the figures 5.8 and 5.9, respectively.

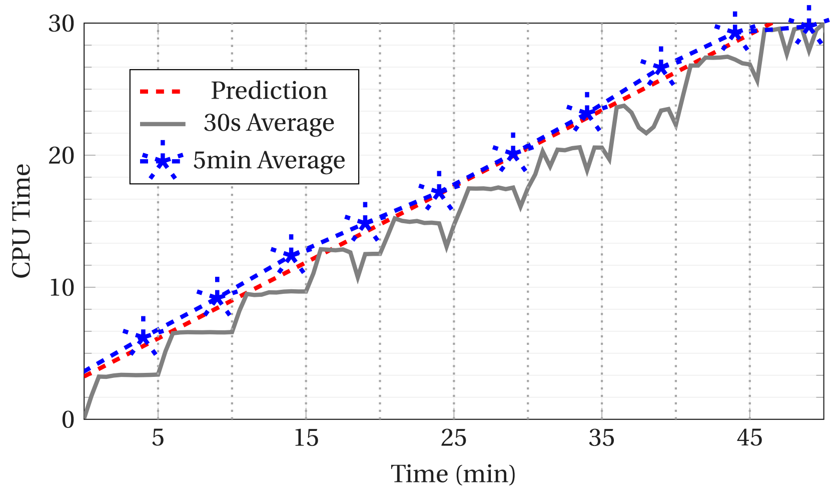


FIGURE 5.7: Whether the UEs are placed on the same machine with the gNB or not, the regimes of prediction are different. Compared to the figure 5.6, the CPU time is significantly higher, but the RTT is much lower.

- A zero down-time reconfiguration of AMR RAN in response to topology change and addition of a new core network;
- An upgrade procedure, where the AMR CN that was connected to an AMR RAN instance gets upgraded to a newer version, and in response, the UE performs a connection reestablishment procedure.

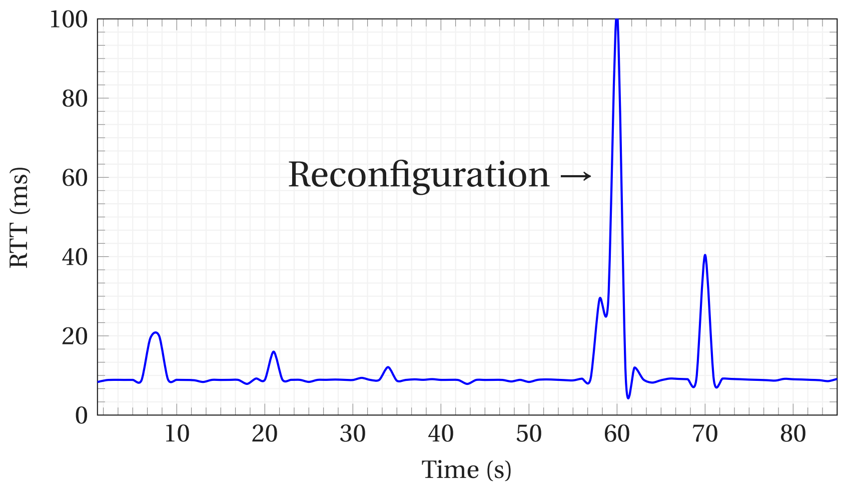


FIGURE 5.8: A zero down-time reconfiguration of AMR RAN in response to topology change and addition of a new core network. The arrow points out the effect of the reconfiguration to a peak in the RTT values.

For the first trial, the figure 5.8 shows the E2E RTT of the UE, which without a down-time, experiences only 3 seconds of service degradation. This shows how ATHENA enables major network reconfigurations without any service outage or restarting any of the containers, Pods, or applications. The changes made in the network description are propagated in the matter of a few seconds to the affected Elements, portraying an extremely dynamic and agile network deployment

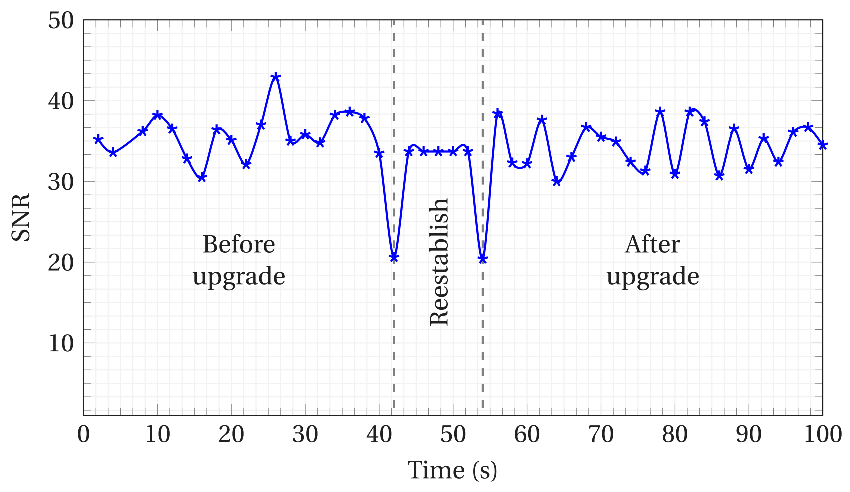


FIGURE 5.9: An upgrade procedure, where the AMR CN that was connected to an AMR RAN instance gets upgraded to a newer version, and in response, the UE performs a connection reestablishment procedure. The phases of the transition are shown in the figure.

The second trial in the figure 5.9, plots the SNR values observed in the RAN for one particular UE. We have chosen the SNR values as a reference to show the changes at the lowest level of the RAN, in the PHY layer. When the SNR gets abnormally low, the UE is transitioning between phases. Since the bearer between the RAN and the CN gets lost, the UE tries a service connectivity request. The request is rejected due to the context loss in the CN, but the UE tries again by establishing a new NAS session over a new RRC. The NAS request fails because the UE tries SUCI instead of SUPI, but the CN is unable to reveal the concealed identity of the UE. Finally, a second reestablishment is successful, and after 12 seconds of lack of E2E service, the UE is back on again. For upgrades of the service once each month, 12 seconds of service outage is equivalent to 99.9995% service reliability. Thus, ATHENA is one of the few platforms able to continuously deliver (CD) with the required service reliability of 5-nines or 6-nines.

5.3 Green MANO/OAM

Sustainability and energy efficiency have been listed already in the requirements for 6G [50], [51], while the foundations to achieve them are already established by the lean-design, fine-grained controlling, and variety of optimizations in 5G. The design of ATHENA is solicitous to sustainability and green computing, by providing built-in features in its observability and control mechanisms. To realize green MANO/OAM one should consider two factors:

1. The extra overhead introduced by the MANO/OAM specifically in terms of computation resources should be minimal;
2. The MANO/OAM should provide means by which algorithms on energy optimizations could be applied.

One particular concern that gains significance on transition from a simple, all-in-one access or core network to disaggregated solutions that are of interest to OpenRAN is the overhead that a MANO design introduces which scales with the number of deployed units in most cases. The flexibility and Day-2 operation offered through macro-decisions and micro-decisions in ATHENA are the key to resolve over-provisioning and achieve a sustainable deployment. These decisions span over the whole spatio-temporal dimension as demonstrated in the following examples.

The figures 5.10 and 5.11 demonstrate a tradeoff between QoS and energy saving in general; the solid black lines are the optimizations made by ATHENA, and the doubly white lines are the baseline. The area in between the graphs, colored in green, is the actual power saved. The decisions, either micro or macro, would affect user performance, either in terms of service outage or QoS drops. Either way, one could encapsulate this as a Quality Reduction Metric (QRM) as discussed in the section 4.6.4. The former metric should be normalized by the number of affected users to form a more accurate picture of the impact. The Power Saving Score (PSS) shows the ratio of average normalized power consumption where the baseline power is deducted from the average values to show only the difference. The operator defines these two metrics based on the observed raw data as well as his desired balance between them. The Energy Optimizer Operator configures its decision as well as Manager's to resolve the tradeoff accordingly. The metrics are ratios and have no dimensional units, hence mathematically comparable. Both the following experiments on power consumption are done with OAI gNB over USRP B210.

To showcase an example of a micro-decision in ATHENA specifically for green MANO/OAM, we provide a scenario in which the tradeoff between availability and energy-efficiency stands out. Imagine a UE that is moving around a facility to collect and upload some data. A few gNB nodes are deployed along the path, but since the data collection from the devices is frequent yet geographically scattered, the energy optimization becomes important. The required UL throughput is 5 Mbps, and any value below it counts as violation of SLA. When the UE's uplink channel gets worse enough to toss the balance between the mentioned metrics, the UE is dropped from the RAN side; otherwise the bad uplink would require heavy computation to provide a weak service. This

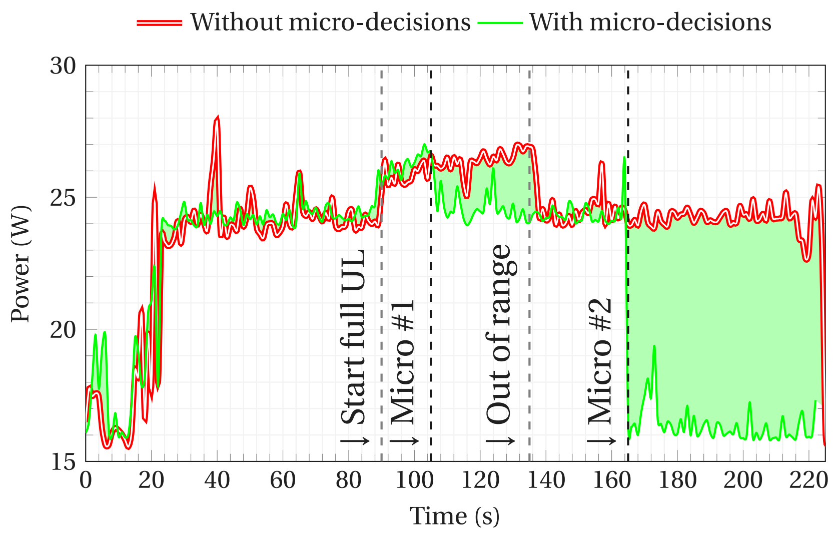


FIGURE 5.10: This figure shows two examples of micro-decisions in ATHENA. The first one concerning a drop-off of the UE in favor of energy saving, whereas the second one is a freeze decision to temporarily set a RAN instance to standby.

decision, as shown in the figure 5.10, saves on average 1.74 W (17.4% PSS) per UE with 61.54% QRM. Also, an unused RAN instance would be temporarily set to standby and remain in sleep mode for the duration on which there is no apparent activity from the UEs. The RAN is awakened at the moment that a UE is activated in its vicinity on a proximate node, and the whole process of detection and activation of the dormant instance all together takes less than 1 second, but it allows us to have on average 7.83 W (78.3% PSS) power saving per gNB. In this case, QRM could be considered almost zero for an agile MANO/OAM.

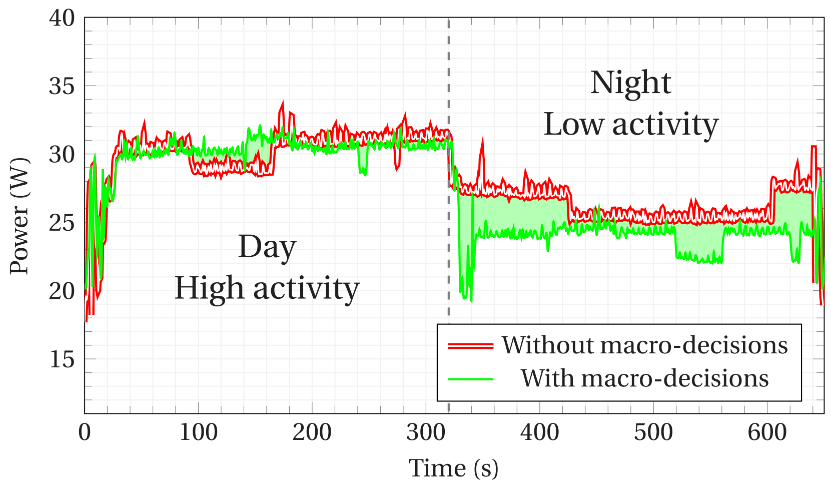


FIGURE 5.11: This figure shows an example of macro-decisions in ATHENA. The decision follows the daily pattern of traffic and changes the bandwidth of the gNBs accordingly.

This could be used to intelligently remote-control very far away sites, where for a significant portion of time there might be no user activity, and for example, the start trigger could also be based on a handover signal. ATHENA Base Operator incorporates the geographical distribution of the nodes in its decision-making by using the Kubernetes region and zone labels that could provide metadata about the geographical location of the nodes in the cluster. Thus, one could use this metadata to extract the list of potential UEs in the vicinity of a RAN instance and take actions accordingly. Evidently, this information could be enriched by deploying specific utilities on each site to measure and report real-time radio signal levels in its proximity or even by the statistics extracted from the network instances themselves. To view this in scale, we have considered three different RAN instances deployed on different sites and the UE traverses from one site to another continuously and randomly. The total power usage of the sites as well as the baseline (without micro-decisions) are plotted in the figure 5.10, where one could observer significant power saving of around 60% in total. Even though we have just considered freeze micro-decision, the WMI could be expanded to take more effective actions such as reducing the bandwidth to handle variations in the presence of the UEs, not just by their disappearance.

Moreover, ATHENA exposes some Workload-dependent statistics via its OMI interface in Manager where later could be processed by an Energy Optimizer Operator on the Operator Plane to take decisive actions with respect to energy saving and green computation for longer cycles, like day and night shifts, exploiting patterns discovered by AI/ML [52], [53]. For example, in the figure 5.11, we have shown a saving of on average 2.24 W (22.4% PSS) per gNB for lowering the bandwidth from 40 MHz to 10 MHz during the second half of the trial period due to the pattern enforced by the Energy Optimizer Operator, causing 77.36% QRM during the nighttime. If we consider 20% active users during the nighttime, normalized QRM becomes 15.47%. As a result of our macro-decision, to save 20% power consumption, the users will lose 80% of the throughput, however, normalized with the number of users at nighttime, say 20%; we get a more reasonable loss of 16%. Of course, AI/ML could be handy in processing this information and adjusting the Energy Optimizer Operator's parameters, to make sure the decision is desirable. Weighting the QRM and PSS the same makes this particular decision favorable. Utilizing the delegated interfaces to the access and core network controllers, these actions span to compute, radio, and network resources across the whole deployment.

The micro- and macro-decisions may be combined to maximize the energy efficiency as shown in the table 5.3. The Energy Optimizer Operator considers a green budget per user as part of its SLA, indicating the maximum tolerable QRM over PSS value. This budget would be first spent on macro-decisions and then for micro-decisions. Since the Manager observes RAN state directly and the QoS in real-time, it could by itself decipher the remaining budget from the tradeoff parameters and monitoring the RAN. Because of their timescale, the micro-decisions could adapt very fast to the side effects of the macro-decisions.

Decision	PSS per UE	QRM
Micro-only	17.4%	61.54%
Macro-only	22.4%	77.36%
Together	39.8%	71.46%

TABLE 5.3: The table shows the effect of combining micro- and macro-decisions on the power saving score (PSS) and quality reduction metric (QRM).

During our experiments in the figures 5.10 and 5.11, we noticed commonly-used tools like powertop[*] are incapable of capturing true power usage of the RAN, because their scope is limited only to the ACPI interface of the CPU, even though the RAN could include RF devices and accelerators that are not reported via the same interface. Thus, we used a battery-powered machine that provides power readings from the battery. The setup is done via USRP B210 which takes all the power from the USB port of the machine. These metrics are exposed to the manager container via SysFS by the device manager for micro-decisions and another node agent reads the coarser data, with higher periods, exposing them to the Prometheus for the usage in the Operator Plane.

We believe by adaptation of ATHENA, since now there are means to analyze and Operate based on energy consumption; the infrastructure providers would gradually provide better energy measurement equipment in their cluster, which enable sophisticated green Operations on ATHENA.

[*] `https://github.com/fenrus75/powertop`

Chapter 6

Conclusion

In this chapter, we reveal the final optimization problem that makes the flow of the thesis connected. Based on this problem statement and the materials provided in the earlier chapters, we discuss the future work. In chapter 3, we performed empirical quantified analysis on a group of mathematically well-defined problems that appear in a DevOps pipeline applied in the context of cloud-native 5G and 6G. The result of this analysis is a set of practices that are incorporated in our CI/CD platform. In chapter 4, we shift the focus to the MANO systems with mostly a qualitative design journey to employ the best of cloud-native technologies in the MANO systems for 5G and 6G. The nature of these studies are completely different from one another, but they both contribute to the same optimization problem and the same level of qualities in the final solution.

This thesis signifies a monumental step in addressing the unique challenges that 5G and 6G network operators face, particularly in the realm of MANO. By focusing on cloud-native principles and practices such as consistent DevOps and declarative automation, this research not only responds to the pressing need for reliable and efficient network management but also acts as a foundation for further innovations. The introduced specialized cloud-native MANO, presents a viable, scalable, and fundamentally redefined solution that fills existing gaps in traditional and current MANO systems. The efficacy of this approach is validated through a real-world proof-of-concept, offering not just theoretical insights but also practical tools and methodologies that drastically outperform existing solutions in terms of agility and overhead.

By aligning the design of MANO systems with cloud-native tenets, this work sets the stage for a unified, sustainable approach to network management. In particular, principles like consistent automation echo the crucial need for reliability, while the focus on multi-x systems captures the essence of diverse and heterogeneous networks. The research not only contributes valuable knowledge to academia but also delivers actionable solutions that are used in its empirical evaluation.

6.1 Global Optimization Problem

Concisely speaking, during this thesis, we have studied a global optimization problem for cloud-native 5G and 6G. As stated earlier, this optimization is in the form of a cost-revenue problem for the network operators. A generic cost-revenue problem calculates the difference between the revenue and the cost across the whole network, for all the services. In this thesis, we model the revenue as a direct function of business agility: The faster the new services are built, tested, and deployed, the more revenue the network operator can generate. Meanwhile, the cost is modeled as a function of the resources used to realize the services.

Thus, if we consider the revenue for the service i as $r_i : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, it would be a monotonically decreasing function of t_i, the time it takes to build, test, and deploy the service i. This variable t_i itself is the sum of the *Dev* portion of time, d_i and the *Ops* portion of time, o_i, where the former could be improved by introducing better concurrency and more resources, but the latter is mostly agnostic to the resources used. The chapter 4 optimizes o_i independently; since regardless of the value of d_i, a lower o_i would increase the revenue, without introducing cost. The problem statement of chapter 4 essentially breaks down o_i to multiple Day-1 and Day-2 operation timers and empirically improves them over the state-of-the-art. Furthermore, in chapter 4, we reduce the overhead introduced by the *Ops* which results into constant reduction of the resources, hence the cost.

The optimization of d_i is more delicate though. First of all, as introduced in chapter 3, we have to maintain consistency while addressing multi-x concurrent build jobs. This is introduced as a condition for the optimization problem. Secondly, we take into account four decisions as given in the table 6.1. The Artifact Mapping would determine how much Resource-based Time and Image Size would be used, which directly translate to the compute and storage costs. We also discussed that a proper mapping could potentially reduce the time needed for some of Day-2 operations, hence reducing the o_i. We compared our mapping with the commonly used Microimage approach and demonstrated that our approach is more efficient. The Build Strategy affects the consistency and concurrency of multi-x artifact variants, which is studied for four different types of strategies, proving TRIDENT build strategy gives the best consistency and concurrency. Consistency is our condition of the optimization problem, and concurrency is the main factor that affects the d_i. The Caching Strategy sets the storage and download time, which directly translate to storage and network costs. We demonstrated the caching and compressing factors of our build system in chapter 3. Finally, the Job Assignment affects the parallel build time, the most important factor in d_i. The effects of this decision only become visible if we consider the realistic scenarios in which the total resources are limited. This limit is introduced as a condition for the optimization problem.

Var	**Decision**	**Effect**	**Reference**
$\mathscr{M}$	Artifact Mapping	Resource-based Time and Image Size	3.2.1
$\mathscr{S}$	Build Strategy	Consistency and Concurrency	3.2.3
$\mathscr{C}$	Caching Strategy	Storage and Download Time	3.2.3
$\mathscr{A}$	Job Assignment	Parallel Build Time and Concurrency	3.2.3

TABLE 6.1: Summary of decisions in DevOps optimization problem and how they affect the cost and revenue.

Eventually, by defining $\mathscr{M}$ as the Artifact Mapping, $\mathscr{S}$ as the Build Strategy, $\mathscr{C}$ as the Caching Strategy, and $\mathscr{A}$ as the Job Assignment, we can define the optimization problem could be formulated as the equation 6.1. It is not noted in the equation, but the term c_i for the cost of the service i depends also on the overhead introduced by the *Ops* which is minimized in chapter 4.

$$\begin{aligned} \max_{\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}} \quad & \sum_{i=1}^{n} r_i(o_i(\mathscr{M}) + d_i(\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}))) - c_i(\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}) \\ \text{subject to} \quad & \text{Consistency}(\mathscr{S}) \\ & \text{Limited Resources}(\mathscr{A}) \end{aligned} \tag{6.1}$$

Since still this formulation is overly complex, we approximate the problem by assuming that the objective is linearly separable by its variables. Hence, in each section of the chapters of the thesis, we mostly pay attention to optimizing the objective with respect to one of the variables regardless of the others. To justify this format of formulation, we could replace the cost with a constant driven from the design of the system and merge the cost function as a condition for the optimization problem into the limit the resources. This results into the equation 6.2 with the objective function further reduced to include the argument of the revenue function since it is a decreasing function of the time.

$$
\begin{aligned}
\min_{\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}} \quad & \sum_{i=1}^{n} o_i(\mathscr{M}) + d_i(\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}) \\
\text{subject to} \quad & \text{Consistency}(\mathscr{S}) \\
& \text{Resources}(\mathscr{M},\mathscr{S},\mathscr{C},\mathscr{A}) \leq \text{Threshold}
\end{aligned}
\tag{6.2}
$$

Now in this formulation, o_i would reduce to the agility of the MANO system while d_i represents the DevOps agility. The optimization of o_i is done in chapter 4 by employing a novel *design* whereas for d_i we used the best practices of DevOps fine-tuned for an agile telco multi-x environment with the empirical analysis of the chapter 3.

6.2 Cloud Native Qualities

In this thesis, we have introduced a set of qualities that are essential for a cloud-native 5G and 6G network. These qualities are in fact derived from the decisions that we made in the overall design of the platform as listed in the table 6.2.

Decision	**Consequence**	**Reference**
Privileges	Security	3.1
Build Recipes	Declarative DevOps	3.2.2
Versioning Model	Security and Consistency	3.2.5
Operator Plane	Extensibility and Declarative Reconciliation	4.2
Sidecar Management	Agility	4.6
Multi-Source Data Lake	Observability	4.8.1

TABLE 6.2: Summary of decisions in DevOps optimization problem and how they affect the qualiities of the platform.

6.3 Future work

A considerable contribution of this thesis is forming a foundational prototype for the full lifecycle of a cloud-native 5G and 6G network. A wide range of future work could be built on top of this prototype, whether on top of the Operator Plane or as just use cases on top of the platform.

Furthermore, we have introduced several interesting research questions, especially for DevOps that could be studied either in the specific context of cloud-native 5G and 6G or in general. There are four categories of extensions that could be studied in the future.

- Defining new use cases on top of the platform and evaluate them in different environments or dimensions of multi-x;
- Introduction of new Operators to the Operator Plane, support for new types of resources, or onboarding new set of NFs;
- Defining new metrics, algorithms, and strategies to replace the ones that we have used in this thesis;
- Find new models and abstractions on the current design of the platform.

As an example, the following questions could be immediately driven from the current work:

1. Optimal caching strategy for a multi-x pipeline considering the availability; related to the section 3.2.3.
2. Study of the general formulation for the limited resource allocation for concurrent and consistent build jobs; related to the section 3.2.4.
3. Dynamic non-uniform resource allocation for concurrent and consistent build jobs; related to the section 3.2.4 and the section 3.2.1.
4. Optimal scheduling algorithm for Open RAN and slicing with prediction and trending analysis; related to the section 4.5 and the section 5.2.
5. Integrated energy metrics with breakdown to the level of UEs, applications, and NFs, considering all the explicit and implicit contributors to the energy consumption; related to the section 5.3.
6. AI/ML-aided intelligent for OAM oriented with business goals by extending the Operator Plane, perhaps using the novel technologies such as Large Language Models (LLMs); related to the section 4.2.

All

[1] Stephen S. Mwanje, Guillaume Decarreau, Christian Mannweiler, Muhammad Naseer ul Islam, and Lars-Christoph Schmelz, “Network management automation in 5g: Challenges and opportunities,” in *27th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC 2016, Valencia, Spain, September 4-8, 2016*, IEEE, 2016, pp. 1–6. DOI: 10.1109/PIMRC.2016.7794614. [Online]. Available: https://doi.org/10.1109/PIMRC.2016.7794614.

[2] Alireza Mohammadi and Navid Nikaein, “Athena: An intelligent multi-x cloud native network operator,” *IEEE Journal on Selected Areas in Communications, JSAC Special Issue on Open RAN: a New Paradigm for Open, Virtualized, Programmable, and Intelligent Cellular Networks, 28 November 2023*, 2023.

[3] Chieh-Chun Chen, Mikel Irazabal, Chia-Yu Chang, Alireza Mohammadi, and Navid Nikaein, “Flexapp: Flexible and low-latency xapp framework for ran intelligent controller,” in *ICC 2023, IEEE International Conference on Communications, 28 May-1 June 2023, Rome, Italy*, IEEE, Ed., Rome, 2023.

[4] ONF TR-502, “Sdn architecture,” ONF, Tech. Rep., 2014.

[5] AbdulAziz AbdulGhaffar, Ashraf S. Mahmoud, Marwan H. Abu-Amara, and Tarek R. Sheltami, “Modeling and evaluation of software defined networking based 5g core network architecture,” *IEEE Access*, vol. 9, pp. 10 179–10 198, 2021. DOI: 10.1109/ACCESS.2021.3049945. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3049945.

[6] Robert Schmidt, Mikel Irazabal, and Navid Nikaein, “Flexric: An sdk for next-generation sd-rans,” in *CONEXT 2021, 17th International Conference on Emerging Networking EXperiments and Technologies, 7-10 December 2021, Munich, Germany (Virtual Conference)*, ACM, Ed., © ACM, 2021. This is the author&#039;s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in CONEXT 2021, 17th International Conference on Emerging Networking EXperiments and Technologies, 7-10 December 2021, Munich, Germany (Virtual Conference) http://doi.org/10.1145/3485983.3494870, Munich, 2021.

[7] Technical Specification Group Core Network and Terminals, “Ts 29.244 v17.4.0,” 3GPP, Tech. Rep., 2022.

[8] 3GPP, “3gpp ts 23.288 v18.3.0: Architecture enhancements for 5g system (5gs) to support network data analytics services(release 18),” 3rd Generation Partnership Project (3GPP), Technical Specification (TS), 2023. [Online]. Available: https://www.3gpp.org/DynaReport/23288.htm.

[9] ——, “3gpp ts 28.533 v17.4.0: Management and orchestration; architecture framework (release 17),” 3rd Generation Partnership Project (3GPP), Technical Specification (TS), 2023. [Online]. Available: https://www.3gpp.org/DynaReport/28533.htm.

[10] O-RAN Alliance, "O-ran.wg3.ricarch-r003-v04.00: O-ran near-rt ric architecture 4.0," O-RAN Alliance, Technical Specification, 2023. [Online]. Available: `https://www.o-ran.org/specifications`.

[11] Rihan Hai, Christoph Quix, and Matthias Jarke, "Data lake concept and systems: A survey," *CoRR*, 2021. [Online]. Available: `https://arxiv.org/abs/2106.09592`.

[12] 3GPP, "3gpp ts 23.501 v18.3.0: System architecture for the 5g system (5gs); stage 2 (release 18)," 3rd Generation Partnership Project (3GPP), Technical Specification (TS), 2023. [Online]. Available: `https://www.3gpp.org/DynaReport/23501.htm`.

[13] *Ieee standard for a precision clock synchronization protocol for networked measurement and control systems*, IEEE Std 1588-2008, IEEE, 2008.

[14] OSM End User Advisory Group, "Osm scope, functionality, operation and integration guidelines," ETSI, Tech. Rep., 2019.

[15] Shinji Mizuta, Anil Umesh, Yoshihiro Nakajima, and Yuya Kuno, "Initiatives toward virtualized ran," en, *NTT Technical Review*, vol. 20, no. 11, pp. 52–63, 2022. DOI: `10.53829/ntr202211fa7`.

[16] ONF, "Onap architecture overview," ONF, Tech. Rep., 2018.

[17] Mavrakis Dimitris and Saadi Malik, "Open ran: Market reality and misconceptions," ABI research, Tech. Rep., 2020.

[18] Jason Dobies and Joshua Wood, *Kubernetes Operators*. O'Reilly Media, Inc., 2020, pp. 67–77.

[19] Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002, ISSN: 0163-5700. DOI: `10.1145/564585.564601`. [Online]. Available: `https://doi.org/10.1145/564585.564601`.

[20] ETSI GR NFV-REL 010, "Network functions virtualisation (nfv) release 3; reliability; report on nfv resiliency for the support of network slicing," ETSI, Tech. Rep., 2019.

[21] Borja Nogales, Ivan Vidal, Diego R. Lopez, Juan Rodriguez, Jaime Garcia-Reinoso, and Arturo Azcorra, "Design and deployment of an open management and orchestration platform for multi-site nfv experimentation," *IEEE Communications Magazine*, vol. 57, no. 1, pp. 20–27, 2019. DOI: `10.1109/MCOM.2018.1800084`.

[22] Sevil Dräxler, Manuel Peuster, Holger Karl, Michael Bredel, Johannes Lessmann, Thomas Soenen, Wouter Tavernier, Sharon Mendel-Brin, and George Xilouris, "Sonata: Service programming and orchestration for virtualized software networks," May 2016.

[23] G. Biczók, Manos Dramitinos, László Toka, Poul E. Heegaard, and Håkon Lønsethagen, "Manufactured by software: Sdn-enabled multi-operator composite services with the 5g exchange," *IEEE Communications Magazine*, vol. 55, pp. 80–86, 2017.

[24] Kostas Katsalis, Navid Nikaein, and Anta Huang, "Jox: An event-driven orchestrator for 5g network slicing," in *NOMS 2018, IEEE/IFIP Network Operations and Management Symposium, 23-27 April 2018, Taipei, Taiwan*, IEEE, Ed., Taipei, 2018.

[25] Osama Arouk and Navid Nikaein, "Kube5g: A cloud-native 5g service platform," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6. DOI: `10.1109/GLOBECOM42002.2020.9348073`.

[26] 5G Americas, "5g and the cloud," Tech. Rep., 2019.

[27] Alcardo Alex Barakabitze, Arslan Ahmad, Rashid Mijumbi, and Andrew Hines, "5g network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106 984, 2020. DOI: 10.1016/j.comnet.2019.106984. [Online]. Available: https://doi.org/10.1016%2Fj.comnet.2019.106984.

[28] Massimo Condoluci and Toktam Mahmoodi, "Softwarization and virtualization in 5g mobile networks: Benefits, trends and challenges," *Computer Networks*, vol. 146, Sep. 2018. DOI: 10.1016/j.comnet.2018.09.005.

[29] O-RAN.WG10.OAM-Architecture-R003-v08.00, "O-ran operations and maintenance architecture," O-RAN Alliance, Tech. Rep., 2023.

[30] Parth Yadav and Vipin Kumar Rathi, *Kupenstack: Kubernetes based cloud native openstack*, 2021. arXiv: 2106.02956 [cs.DC].

[31] P. C. Amogh, G. Veeramachaneni, A. K. Rangisetti, and B. R. Tamma, "A cloud native solution for dynamic auto scaling of mme in lte," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, 2017, pp. 1–7.

[32] Jose Ordonez-Lucena, Christos Tranoris, João Rodrigues, and Luis M. Contreras, "Cross-domain slice orchestration for advanced vertical trials in a multi-vendor 5g facility," in *2020 European Conference on Networks and Communications (EuCNC)*, 2020, pp. 40–45.

[33] Sameerkumar Sharma, Raymond Miller, and Andrea Francini, "A cloud-native approach to 5g network slicing," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 120–127, 2017.

[34] Kay Haensge, Dirk Trossen, Sebastian Robitzsch, Michael Boniface, and Stephen Phillips, "Cloud-native 5g service delivery platform," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2019, pp. 1–7. DOI: 10.1109/NFV-SDN47374.2019.9040042.

[35] Osama Arouk and Navid Nikaein, "5g cloud-native: Network management automation," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2.

[36] Rajkarn Singh, Cengis Hasan, Xenofon Foukas, Marco Fiore, Mahesh K. Marina, and Yue Wang, "Energy-efficient orchestration of metro-scale 5g radio access networks," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488786.

[37] Carla-Fabiana Chiasserini, Giada Landi, Marco Capitani, Francesco Malandrino, and Claudio Casetti, "An optimization-enhanced mano for energy-efficient 5g networks," *IEEE/ACM Transactions on Networking*, vol. 27, Jul. 2019. DOI: 10.1109/TNET.2019.2931038.

[38] Raffaele Bolla, Roberto Bruschi, Franco Davoli, Chiara Lombardo, and Jane Frances Pajo, "Debunking the "green" nfv myth: An assessment of the virtualization sustainability in radio access networks," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 180–184. DOI: 10.1109/NetSoft48620.2020.9165481.

[39] Bilgin Ibryam and Roland Huß, *Kubernetes Patterns*. O'Reilly Media, Inc., 2019, pp. 131–134.

[40] O-RAN.WG1.O-RAN-Architecture-Description-v07.00, "O-ran architecture description," O-RAN Alliance, Tech. Rep., 2022.

[41] David Breitgand, Vadim Eisenberg, Nir Naaman, Nir Rozenbaum, and Avi Weit, "Toward true cloud native nfv mano," in *2021 12th International Conference on Network of the Future (NoF)*, 2021, pp. 1–5. DOI: `10.1109/NoF52522.2021.9609908`.

[42] O-RAN.WG6.O-Cloud Notification API-v03.00, "O-cloud notification api specification for event consumers," O-RAN Alliance, Tech. Rep., 2022.

[43] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle, "Performance implications of packet filtering with linux ebpf," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, 2018, pp. 209–217. DOI: `10.1109/ITC30.2018.00039`.

[44] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia, "Open, programmable, and virtualized 5g networks: State-of-the-art and the road ahead," *CoRR*, 2020. [Online]. Available: `https://arxiv.org/abs/2005.10027`.

[45] Dariusz Wypiór, Mirosław Klinkowski, and Igor Michalski, "Open ran; radio access network evolution, benefits and market trends," *Applied Sciences*, vol. 12, no. 1, 2022, ISSN: 2076-3417. DOI: `10.3390/app12010408`. [Online]. Available: `https://www.mdpi.com/2076-3417/12/1/408`.

[46] Korian Edeline and Benoit Donnet, "A bottom-up investigation of the transport-layer ossification," in *2019 Network Traffic Measurement and Analysis Conference (TMA)*, 2019, pp. 169–176. DOI: `10.23919/TMA.2019.8784690`.

[47] Nikolaos Sapountzis, Stylianos Sarantidis, Thrasyvoulos Spyropoulos, Navid Nikaein, and Umer Salim, "Reducing the energy consumption of small cell networks subject to qoe constraints," pp. 2485–2491, Feb. 2015. DOI: `10.1109/GLOCOM.2014.7037181`.

[48] Brendan Burns, Joe Beda, and Kelsey Hightower, *Kubernetes Up & Running; Dive into the Future of Infrastructure*, Second Edition. O'Reilly Media, Inc., 2019, pp. 3–6.

[49] Kyuho Son, Hongseok Kim, Yung Yi, and Bhaskar Krishnamachari, "Base station operation and user association mechanisms for energy-delay tradeoffs in green cellular networks," *Selected Areas in Communications, IEEE Journal on*, vol. 29, pp. 1525–1536, Oct. 2011. DOI: `10.1109/JSAC.2011.110903`.

[50] FG-NET2030; Focus Group on Technologies for Network 2030, "Network 2030 - additional representative use cases and key network requirements for network 2030," ITU-T, Tech. Rep., 2020.

[51] Tongyi Huang, Wu Yang, Jun Wu, Jin Ma, Xiaofei Zhang, and Daoyin Zhang, "A survey on green 6g network: Architecture and technologies," *IEEE Access*, vol. 7, pp. 175 758–175 768, 2019. DOI: `10.1109/ACCESS.2019.2957648`.

[52] Leonardo Militano, Anastasios Zafeiropoulos, Eleni Fotopoulou, Roberto Bruschi, Chiara Lombardo, Andy Edmonds, and Symeon Papavassiliou, "Ai-powered infrastructures for intelligence and automation in beyond-5g systems," in *2021 IEEE Globecom Workshops (GC Wkshps)*, 2021, pp. 1–6. DOI: `10.1109/GCWkshps52748.2021.9682117`.

[53] China Mobile Research Institute, "C-ran the road towards green ran," China Mobile, Tech. Rep., 2020.