Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Je-Ruei Yang

Automating Security Operations in Telecommunication Networks with GitOps

Master's Thesis
Espoo, July 30, 2024

Supervisors: Professor Tuomas Aura, Aalto University
Professor Raja Appuswamy, EURECOM
Advisor: Joel Reijonen, M.Sc. (Tech.), Oy L M Ericsson Ab

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Abstract

Author:	Je-Ruei Yang		
Title: Automating Security Operations in Telecommunication Networks with GitOps			
Date:	July 30, 2024	**Pages:**	70+8
Major:	Security and Cloud Computing	**Code:**	SCI3113
Supervisors:	Professor Tuomas Aura, Aalto University Professor Raja Appuswamy, EURECOM		
Advisor:	Joel Reijonen, M.Sc. (Tech.), Oy L M Ericsson Ab		

5G network operators are continuously seeking more efficient methods for network security management. Concurrently, GitOps has emerged as a novel and effective approach to manage applications, configurations, and infrastructure. However, there has been limited research on integrating these two advancements. This thesis explores the integration of GitOps into telecommunication network security operations, proposing a new workflow to address the challenges in traditional methods.

We identify the challenges presented in conventional network security operations and introduce a new workflow based on the GitOps principles. Our work includes designing a declarative model to describe and store resources in Git repositories and implementing GitOps operators to ensure alignment between the desired state in Git repositories and the actual state of the managed network. Additionally, we incorporate various validation strategies within the continuous integration pipeline to enhance the robustness of the workflow.

The evaluation of the workflow involves case studies that compare traditional and GitOps workflows in real-world scenarios, such as updating security configuration, establishing experimental environments, performing disaster recovery, and detecting configuration drift. The analysis shows that the GitOps workflow improves efficiency, visibility, and traceability, eases disaster recovery, and automates configuration drift correction. The findings suggest that mobile network operators should consider adopting the GitOps workflow to enhance their overall security operations.

Keywords:	GitOps, telecommunication network, configuration management, network security management, network automation
Language:	English

Master en Securité et Cloud Computing | Résumé

Auteur:	Je-Ruei Yang		
Titre: Automatiser les opérations de sécurité dans les réseaux de télécommunications avec GitOps			
Date:	30 juillet 2024	**Pages:**	70+8
Spécialisation:	Big Data Security	**Promo:**	2024
Superviseurs:	Professor Tuomas Aura, Aalto University Professor Raja Appuswamy, EURECOM		
Conseiller:	Joel Reijonen, M.Sc. (Tech.), Oy L M Ericsson Ab		

Les opérateurs de réseaux 5G cherchent en permanence des méthodes plus efficaces pour la gestion de la sécurité des réseaux. Parallèlement, GitOps s'est révélé être une approche nouvelle et efficace pour gérer les applications, les configurations et l'infrastructure. Cependant, peu de recherches ont combiné ces deux avancés. Ce mémoire explore l'intégration de GitOps dans les opérations de sécurité des réseaux de télécommunication, en proposant un nouveau workflow pour améliorer les procédés actuels.

Nous procédons par la présentation des difficultés présentes dans les opérations de sécurité des réseaux conventionnelles et introduisons un nouveau workflow basé sur les principes de GitOps. Notre travail inclut la conception d'un modèle déclaratif pour décrire et stocker les ressources dans les dépôts Git et la mise en œuvre des opérateurs GitOps pour assurer l'alignement entre l'état souhaité dans les dépôts Git et l'état réel du réseau géré. De plus, nous intégrons diverses stratégies de validation dans le pipeline d'intégration continue pour renforcer la robustesse du workflow.

L'évaluation de notre implémentation implique des études de cas comparant le workflow traditionnel et celui de GitOps dans des scénarios réels, tels que la mise à jour des configurations de sécurité, l'établissement d'environnements expérimentaux, la réalisation de la récupération après sinistre et la détection des dérives de configuration. L'analyse montre que le workflow de GitOps améliore l'efficacité, la visibilité et la traçabilité, facilite la récupération après sinistre et automatise la correction des dérives de configuration. Les résultats suggèrent que les opérateurs de réseaux mobiles devraient envisager d'adopter le workflow de GitOps pour améliorer leurs opérations de sécurité globales.

Mots-clés:	GitOps, réseau de télécommunication, gestion de la configuration, gestion de la sécurité réseau, automatisation du réseau
Langue:	Anglais

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Tuomas Aura from Aalto University, for his invaluable guidance and feedback. His expertise has been instrumental in shaping the direction of this research. My sincere thanks also go to Professor Raja Appuswamy from EURECOM for his unwavering support throughout the course of the project.

I am grateful to my advisors and colleagues Joel Reijonen, Ville Pesonen, and Adam Peltoniemi from Ericsson for their assistance and insightful discussions. My deepest appreciation goes to my manager, Holger Offermann, for offering me the opportunity to work on such an interesting topic with a remarkable team.

Being part of the SECCLO family has been a wonderful journey filled with joyful memories. Many thanks to my friends, especially the SECCLO 2022 batch, for sharing unforgettable moments over the past two years.

I wish to extend my heartfelt thanks to my family for their unconditional love and support. Special thanks to my father for being my role model and offering fresh perspectives on this thesis.

Last but not least, this work would not have been possible without the encouragement of Yi-Han. Her belief in me has been an endless source of motivation, inspiring me to achieve my best throughout this project.

Espoo, July 30, 2024

Je-Ruei Yang

With the support of the
Erasmus+ Programme
of the European Union

DECLARATION FOR THE MASTER'S THESIS

I warrant, that the thesis is my original work and that I have not received outside assistance. Only the sources cited have been used in this report. Parts that are direct quotes or paraphrases are identified as such.

In Espoo, Finland
Date: July 30, 2024

DÉCLARATION POUR LE RAPPORT DE STAGE

Je garantis que le rapport est mon travail original et que je n'ai pas reçu d'aide extérieure. Seules les sources citées ont été utilisées dans ce projet. Les parties qui sont des citations directes ou des paraphrases sont identifiées comme telles.

À Espoo, Finlande
Date: 30 juillet 2024

Je-Ruei Yang

楊哲睿
YANG Je-Ruei

Abbreviations and Acronyms

2G	Second Generation
3G	Third Generation
3GPP	3rd Generation Partnership Project
4G	Fourth Generation
5G	Fifth Generation
AMF	Access and Mobility Management Function
API	Application Programming Interface
AR	Augmented Reality
CD	Continuous Delivery / Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
CM	Configuration Management
CN	Core Network
CNF	Cloud-native Network Function
CRUD	Create, Read, Update and Delete
DAST	Dynamic Application Security Testing
DNF	Disjunctive Normal Form
DoS	Denial of Service
eMBB	Enhanced Mobile Broadband
ESM	Ericsson Security Manager
GPG	GNU Privacy Guard
GSM	Global System for Mobile Communications
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaC	Infrastructure as Code
IoT	Internet of Things
JSON	JavaScript Object Notation
LAN	Local Area Network
LTE	Long-Term Evolution
MFA	Multi-Factor Authentication

MME	Mobility Management Entity
mMTC	Massive Machine Type Communication
MNO	Mobile Network Operator
MTTR	Mean Time To Recovery
NETCONF	Network Configuration Protocol
NF	Network Function
NFV	Network Function Virtualization
NR	New Radio
NSCM	Network Security Configuration Manager
OAM	Operation, Administration, and Maintenance
PKI	Public Key Infrastructure
PLMN	Public Land Mobile Network
PNF	Physical Network Function
PSTN	Public Switched Telephone Network
RAN	Radio Access Network
REST	Representational State Transfer
SAST	Static Application Security Testing
SBA	Service-Based Architecture
SBI	Service-Based Interfaces
SCA	Software Composition Analysis
SDLC	Software Development Lifecycle
SIEM	Security Information and Event Management
SMS	Short Message Service
SNMP	Simple Network Management Protocol
SOAR	Security Orchestration, Automation and Response
SSH	Secure Shell
TETRA	Terrestrial Trunked Radio
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
URLLC	Ultra Reliable Low Latency Communication
VCS	Version Control System
VM	Virtual Machine
VNF	Virtual Network Function
VoLTE	Voice over LTE
VR	Virtual Reality
WAN	Wide Area Network
WLAN	Wireless Local Area Network
YAML	Yet Another Markup Language / YAML Ain't Markup Language

Contents

Chapter 1

Introduction

In recent years, GitOps has emerged as a promising methodology for orchestrating infrastructure and software applications, representing the evolution of DevOps principles [1, 2]. DevOps principles emphasize collaboration, automation, and continuous improvement and monitoring throughout the software development lifecycle. Built upon these principles, GitOps leverages Infrastructure as Code (IaC), continuous integration and delivery (CI/CD) pipelines, and Git source control to automate configuration management for both infrastructure and applications. The GitOps approach allows automated deployments and effortless rollbacks, thereby promoting consistency, repeatability, and security.

The telecommunication industry has undergone a transformation with the advent of 5G technologies. Despite 5G networks promising enhancements in speed, latency, and capability, which broaden the scope of mobile applications and services, the increased complexity and virtualization introduce a new spectrum of security challenges. Traditional methods of network security management struggle to keep pace with the dynamic virtualized environments, which creates potential vulnerabilities [3]. As telecommunication networks evolve towards virtualized, cloud-native architectures, the demand for efficient and scalable security operations increases.

The present solutions for 5G network security and configuration management are designed to fortify the security and integrity of telecommunication networks [4, 5]. With a policy-based approach, these tools detect network misconfigurations that violate the operator security policies, potentially weakening network security. Additionally, they identify unauthorized or unwanted changes to the network configurations. Lastly, these solutions automate compliance checks to ensure adherence to regulatory standards and industry best practices by streamlining security processes.

While DevOps and GitOps have transformed software development and

5G technologies have revolutionized telecommunication networks, the application of GitOps for security operations within telecommunication networks remains an emerging area. This integration has the potential to improve the efficiency and effectiveness of network security management.

1.1 Problem Statement

The current telecommunication operation processes, along with network security and configuration management systems, suffer from several limitations that hinder their effectiveness:

- **Lengthy change request process.** The process of changing network configurations currently relies on manual and time-consuming approval procedures. The procedures result in delays and inefficiencies.
- **Lack of automation.** In current system, approved changes in configuration require manual operations to implement on the network. This workflow creates opportunities for human errors and introduces delays.
- **Limited visibility and traceability.** The existing system lacks mechanisms to track and document configuration changes accurately, posing challenges for troubleshooting and conducting thorough security audits. Critical information, such as who initiated a change, when it occurred, and its purpose, is often difficult to trace.
- **Absence of continuous reconciliation.** Continuous reconciliation involves continuously detecting and resolving conflicts between the desired and actual states of the network. Given that invalid configurations introduce security risks, the absence of automated processes in existing systems to validate the deployed configurations increases the likelihood of network vulnerabilities.

This thesis tackles these challenges by exploring the integration of the GitOps workflow into the security operation of telecommunication networks, particularly in non-Kubernetes environments. By leveraging the principles of GitOps, we design a workflow for network security and configuration management system that effectively and continuously applies security configurations to safeguard the 5G networks. The workflow offers granular control in change requests and increases visibility, traceability, and rollback capability in configuration management. It also implements reconciliation loops to continuously ensure the desired security posture is enforced in the network. This approach enables secure and efficient operation, facilitates rapid incident response, and enforces continuous compliance.

1.2 Structure of the Thesis

The remainder of the thesis is organized as follows.

Chapter 2 provides the background for the thesis. First, we introduce telecommunication networks, focusing on mobile 5G networks, and the security challenges as well as the technologies in their management and operation. Subsequently, we explore the principles and practices of DevOps and GitOps, along with the advantages of their adoption. Additionally, relevant literature is presented to discuss the integration of GitOps within the telecommunications sector, which provides insights into its applicability and efficacy in this domain.

Chapter 3 provides a detailed explanation of the security management system the thesis is based on. We introduce the terminologies, components, features, and use cases of the system. Subsequently, we explore the integration of a GitOps workflow into the system. Finally, we discuss the requirements and constraints associated with implementing a GitOps workflow within the system.

Chapter 4 proposes a solution to integrate the GitOps workflow into the studied system. This chapter delves into the design and architecture of the proposed solution, along with an in-depth exploration of its implementation details. We discuss the declarative file formats for managing the resources and outline the practical steps involved in executing the proposed workflow. Moreover, we present steps to validate the user-provided files to ensure the robustness of the workflow.

Following this, Chapter 5 then analyzes and evaluates the proposed solution with various real-world scenarios to assess its effectiveness and efficiency. We compare the proposed workflow with the existing one to identify the improvements and benefits. Furthermore, we discuss the benefits and limitations of the proposed workflow along with possible future enhancements and improvements.

Finally, Chapter 6 summarizes the thesis findings and presents the implications of the research outcomes. We also identify potential future research directions and offer reflections on the contributions of the study.

Chapter 2

Background

This chapter introduces the general concepts of telecommunication networks and mobile networks. It then dives into the architecture and components of 5G networks and how such networks are managed, with a particular focus on security. Furthermore, we discuss the evolution from DevOps to GitOps, the latest trend in managing infrastructure and application configurations through infrastructure as code and version control systems. Finally, we provide a review of related work on applying GitOps in telecommunication.

2.1 Telecommunication Networks

A telecommunication network is a system interconnecting various nodes, including fixed-line telephones, mobiles, and network devices such as routers and switches. It utilizes communication links such as cables and radio waves to transmit data and information across long distances. These networks adhere to protocols and operate within different categories, including public switched telephone networks (PSTN) and public land mobile networks (PLMN).

2.2 Mobile Networks

Mobile networks are telecommunication networks that provide wireless connectivity for mobile devices. Examples of these networks include terrestrial trunked radio (TETRA), satellite networks such as Starlink and Globalstar, and cellular networks such as 5G networks. This thesis focuses primarily on cellular-based mobile networks, notably 5G networks.

A mobile network comprises two main components: the radio access network (RAN) and the core network (CN). The RAN connects mobile devices, or user equipments (UE), to the network. The CN manages the network as a whole and provides services, such as authentication and mobility management, to the connected users [6, 7]. Standardization of mobile networks is governed by the 3rd Generation Partnership Project (3GPP), with each generation representing an advancement in mobile technology. The evolution of cellular mobile technologies occurs approximately once every ten years. Table 2.1 shows a summary of the technologies and services offered by each generation of mobile networks.

The 2G mobile network, established in the 1990s, is the first globally accepted mobile standard. It utilizes the Global System for Mobile Communications (GSM), a digital cellular technology, to offer services such as voice calls using circuit-switching and short message service (SMS) with a store-and-forward architecture.

The 3G mobile network, introduced in the 2000s, adopts Universal Mobile Telecommunications System (UMTS) as its core technology. UMTS employs a combination of circuit-switched and packet-switched technologies, with packet switching for data transmission and circuit switching for voice calls. With improved data transfer rates compared to GSM, UMTS allows mobile multimedia applications such as Internet access and video calling.

The 4G network implements Long-Term Evolution (LTE) to enhance data speed and network capacity. LTE is based on packet-switching with an all-IP architecture [8], which allows voice calls to be transmitted over the same data network as internet traffic. This advancement enables services like Voice over LTE (VoLTE) and video services such as streaming and video conferencing.

The state-of-the-art 5G network adopts New Radio (NR) as its primary radio technology. NR offers higher data speeds, ultra-low latency, and massive connectivity compared to previous generations. We introduce the 5G network in detail shortly in Section 2.3.

2.3 5G Networks

Fifth-generation (5G) cellular network is the latest advancement in mobile connectivity, offering higher data speeds, increased capacity, lower latency, and more reliable and efficient services that support a wide range of new applications [9, 10]. These advantages have led to the development of various use cases: enhanced mobile broadband (eMBB) for applications such as video streaming, virtual reality (VR) and augmented reality (AR), Ultra Reliable Low Latency Communication (URLLC) for critical applications like remote surgery or autonomous vehicles, and massive Machine Type Communication (mMTC) for efficient communication among a vast number of low-power

Table 2.1: Mobile network generation comparison.

Generation	Technology	Services
2G (1990s)	GSM	Voice calls and text messaging (short message service, SMS).
3G (2000s)	UMTS	Mobile multimedia applications such as Internet access and video calling.
4G (2010s)	LTE	Mobile streaming, video conferencing, and online gaming.
5G (2020s)	NR	Internet of Things (IoT) and real-time applications such as remote surgery.

devices such as sensors or smart meters [9].

This section provides an overview of the underlying architecture and technologies of 5G networks, as well as the security implications associated with their design. These aspects are essential for advanced network management and network automation, which we discuss in Section 2.4.

2.3.1 Service-Based Architecture

5G network uses a service-based architecture (SBA) to address scalability and flexibility. Previous generations of mobile networks from 2G to 4G utilize a reference-based design. The reference-based design focuses on defining the interfaces, protocols, and the information needed to communicate between each pair of network elements. Adding an element requires defining interfaces and protocols with all existing elements. While functional, the approach inherently limits the scalability and flexibility of the network [7].

On the other hand, the modularity of SBA enhances scalability and flexibility in 5G networks. Figure 2.1 depicts how the 5G SBA unifies the interfaces where network functions (NF) are interconnected through a service bus. In this architecture, NFs provide services to other NFs via standardized service-based interfaces (SBI), typically implemented with REST application programming interfaces (API). The SBI uses standard Internet protocols, such as TCP/IP and HTTP/2, and employs either a Request-Response or a Subscribe-Notify communication model between NFs [11]. The well-defined SBI decouples the NFs by abstracting the underlying implementation details of individual NFs. Consequently, each NF is a self-contained module with clearly defined interfaces and can be upgraded and scaled independently without affecting the other functions.

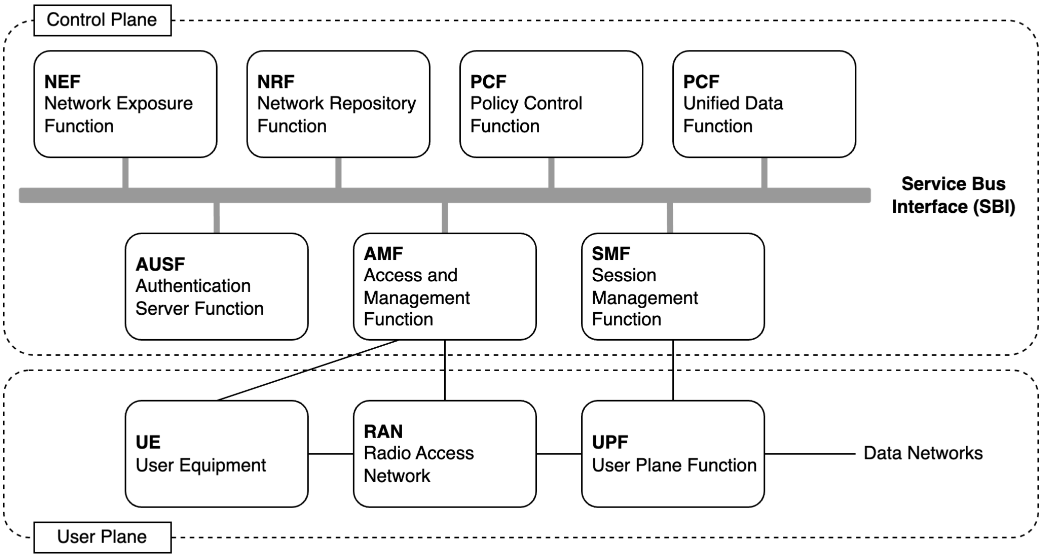

Figure 2.1: 5G service-based architecture [7].

2.3.2 Network Function Virtualization

Network functions (NF) are building blocks of the mobile network that have well-defined external interfaces and functional behavior [12]. These NFs handle critical functionalities for network operation, such as device registration, traffic routing, session management, and user authentication and authorization [11]. Traditionally, NFs resided on dedicated hardware appliances, with each function tied to its own physical hardware. This approach, known as physical network functions (PNF), presented limitations such as inflexibility in scaling, high network infrastructure expenditure, and lock-in to specific hardware vendors [13, 14].

In response to these limitations, network function virtualization (NFV) has emerged as a paradigm shift in 5G networks. NFV provides a more flexible and dynamic NF deployment solution by decoupling the software functionality from the underlying hardware [15]. NFV allows NFs to run within virtualized environments, such as virtual machines (VM) or containers, hosted on commodity hardware. Virtualized NFs, commonly referred to as virtual network functions (VNF), offer advantages including optimized costs and streamlined NF management [16]. Furthermore, to enhance resource utilization and operational flexibility, some NFs leverage cloud-native technologies, such as containerization, to become cloud-native network functions (CNF).

The virtualization strategy complements the modular design of SBA and enables dynamic deployment and scaling of NFs. Moreover, NFV provides flexibility, reconfigurability, and customization for 5G networks [17]. Nevertheless, the dynamic nature of NFV presents challenges for network management. Traditional static configurations and manual provisioning processes are inadequate in such an environment. Section 2.4 introduces network management principles and explores how network automation emerges to manage the complexities of dynamic 5G networks.

2.3.3 Cloud-Native 5G Network

Building upon the principles of SBA and NFV, 5G networks are designed to focus on modularity and virtualization. This approach enables them to cater to the heterogeneous devices they connect to, ranging from mobiles to Internet of Things (IoT) sensors. These devices exhibit diverse traffic patterns, characterized by spatial and temporal fluctuations in volume and attributes such as different quality requirements [18].

In response to the dynamic traffic patterns, 5G networks leverage elastic cloud environments to achieve on-demand scalability, flexibility, and reliability. The integration of SBA, NFV, and elastic cloud infrastructure forms the foundation for developing cloud-native 5G networks to meet the evolving demands of modern network infrastructure.

Containerization is vital in the realization of cloud-native 5G networks. Containers are a more lightweight virtualized technology than VMs as they share the host OS instead of having their own guest OS, as shown in Figure 2.2. By virtualizing and packaging NFs as containers, CNFs facilitate faster deployment and efficient resource utilization compared to traditional VM-based approaches [19]. Moreover, container orchestration platforms such as Kubernetes streamline the management and scaling of CNFs across the cloud infrastructure [20].

The transition towards cloud-native deployments has prompted the development of numerous new solutions. Commercial offerings such as Ericsson Cloud RAN [21] and Nokia Cloud RAN [22] provide solutions for deploying RAN in the cloud. Concurrently, open-source initiatives like Nephio [23] offer tools for deploying and managing cloud-native NFs using Kubernetes.

This migration toward cloud-native underscores the imperative for 5G networks to adopt additional cloud technologies and practices beyond containerization. Sections 2.5 and 2.6 will discuss cloud-native practices ranging from DevOps to GitOps, which are crucial in automating network operations and infrastructure management within the cloud-native 5G network environment.

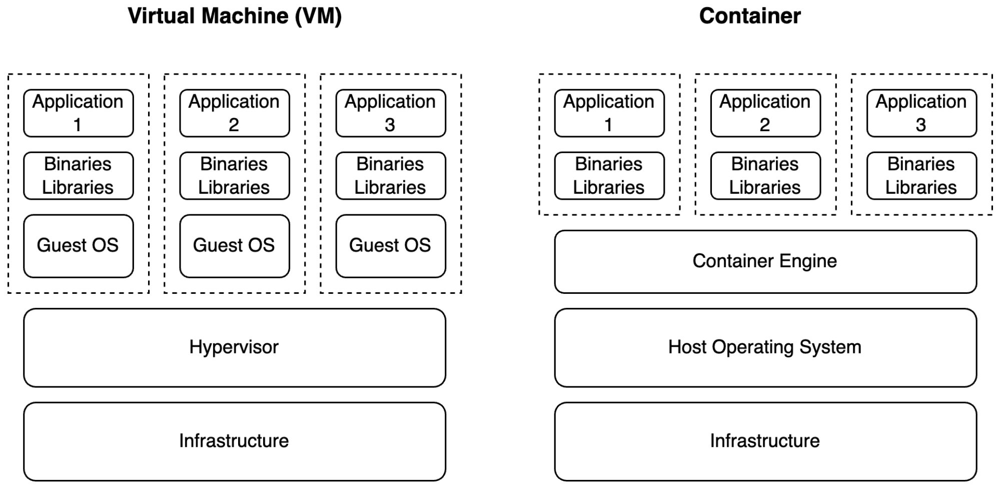

Figure 2.2: Comparison between virtual machines (VM) and containers.

2.3.4 Security Challenges in 5G Networks

The increasing diversity of applications and services, coupled with the growing technology stack in 5G networks, broadens the security surface, therefore introducing security challenges and threats. This section investigates some critical concerns within the 5G security landscape.

The SBA that underpins 5G networks introduces numerous new attack vectors. For instance, malicious actors could exploit the SBI for activities such as denial of service (DoS) attacks to flood NFs with authentication failures, eavesdropping attacks to intercept sensitive data, and man-in-the-middle attacks to manipulate inter-NF communication [24].

Moreover, the virtualized and cloud-native 5G network presents a distinct set of security challenges. For example, attackers can hijack the hypervisor to control the VMs running on top of it and monitor their network traffic. Attackers can also launch DoS attacks on VMs, causing other VMs running on the same physical host to starve for resources [3, 25]. If a public cloud solution is employed, the network faces the risk of cloud intrusion and insider attacks [3]. These threats involve remote attackers attempting unauthorized access to cloud resources or the staff at the cloud service provider exploiting access to the physical servers storing user data.

Attacks on 5G networks are commonly triggered by trivial security oversights, such as misconfigurations and improper hardening of NFs or infrastructure [6]. However, traditional security approaches may be inadequate in discovering and coping with such errors in the ever-evolving and dynamic 5G networks. This inadequacy necessitates enhanced security monitoring and management techniques and tools. In Section 2.4.3 and Section 2.4.4, we investigate network automation and network security management to further explore how these solutions mitigate the security risks associated with 5G networks.

2.4 Network Management

Network management encompasses the design, deployment, monitoring, and operation of a network to ensure its efficiency, reliability, and security. In this section, we first introduce network monitoring and commonly used protocols for network management. Then, we demonstrate how network automation techniques facilitate effective and robust network management. Finally, we explore the security aspects of network management.

2.4.1 Network Monitoring

Network monitoring is the foundation of network management. It refers to the continuous process of observing network activity to identify problems, ensure smooth operation, and detect potential security threats. It helps network operators to understand the current state of the network and reconcile any deviations [26].

Network monitoring tools collect data on traffic patterns, device performance, potential security issues, and more. This data is analyzed to identify and troubleshoot problems, optimize resource utilization, and ensure secure network operation. Additionally, network monitoring can provide insights to detect anomalies and intrusions, support compliance auditing, and enhance the overall network visibility and control.

2.4.2 Network Management Protocols

Network management tools rely on standardized protocols to communicate and interact with the diverse network components. These protocols facilitate essential tasks including configuration management, performance monitoring, and troubleshooting. Some examples are Simple Network Management Protocol (SNMP) for basic device management, Network Configuration Protocol (NETCONF) for advanced configuration management, Hypertext Transfer Protocol Secure (HTTPS) for general-purpose communication, and Command Line Interface (CLI) over Secure Shell (SSH) for manual operations and direct device interaction.

In the telecommunication domain, an array of studies [27–29] underscores the significance of these protocols in the design and implementation of mobile network management systems. These investigations demonstrate the diverse applications of standardized protocols and highlight their efficiency in managing the complexities of modern telecommunication infrastructures.

2.4.3 Network Automation

Network management involves repetitive tasks such as device configuration, software updates, and security patching. Manually managing these tasks at scale can be time-consuming and error-prone, especially in the dynamic and scalable 5G networks. Network automation employs software tools and scripts to streamline these processes, thereby reducing operational expenses, improving network maintenance efficiency, and enhancing scalability [30].

Automation is a key to hardening security in the complex and large-scale modern networks [31]. It ensures security measures are consistently applied across the network and reduces the risk of configuration errors that could lead to security breaches.

2.4.4 Network Security Management

As the networks and cyber threats become increasingly sophisticated, robust security posture and controls become imperative across all layers and resources within the network to safeguard against unauthorized access, data breaches, and other attacks. This necessitates implementing comprehensive network security management strategies to fortify network defenses and mitigate potential risks.

Network security management encompasses deploying specialized tools and techniques to detect and mitigate threats, enforce security policies and configurations, and maintain a secure network environment at scale [31]. These strategies include utilizing functionalities such as security information and event management (SIEM) and security orchestration, automation, and response (SOAR). SIEM identifies potential security incidents, providing insights for security analysts. Meanwhile, SOAR automates security incident response tasks, which improves operational efficiency and reducing response times.

In addition to these measures, the principles of defense in depth and zero trust assume a paramount role in enhancing 5G network security resilience [6, 32, 33]. A defense-in-depth strategy involves deploying multiple layers of security controls throughout the network infrastructure, thereby creating redundant protections to potential threats. Similarly, adherence to the zero-trust model entails continuously verifying and authenticating all network users and devices, regardless of their location within the network perimeter. Notable solutions in the telecommunication industry that embody these functionalities and principles include Ericsson Security Manager (ESM) from Ericsson [4] and NetGuard Cybersecurity Dome from Nokia [5].

2.5 DevOps

Another pillar of the thesis is the operational framework of network security management solutions, encompassing a spectrum from DevOps to the contemporary trend of GitOps.

DevOps has emerged as a paradigm and a set of practices to bridge the gap between the software development (Dev) and IT operations (Ops) teams. This collaborative approach aims to streamline the software development lifecycle (SDLC). It enables faster delivery of high-quality software and reducing process risks compared to the old waterfall model. This section briefly introduces the principles of DevOps, the practices to address these principles, and the benefits of adopting these practices.

2.5.1 DevOps Principles

DevOps encompasses many principles [1, 34, 35], including but not limited to the technical, cultural, and organizational aspects. In this section, we examine the principles that are most relevant to the thesis:

Collaboration. DevOps addresses a cultural shift of enhancing collaboration between software development and IT operations teams. This principle requires team members to share information, broaden their skill sets, and take shared responsibility for the entire lifecycle of a product or service [34]. The cultural shift connects different teams to work towards common goals and results in faster development cycles with better-quality software.

Automation. Automating repetitive tasks such as building, configuring, and testing applications is essential in DevOps. Automation guarantees consistent and repeatable results, improves efficiency, and reduces the potential of human errors [1, 34]. This principle allows team members to free up valuable time and focus on activities that provide more value [36].

Continuity. Every development step should be performed frequently or continuously, including building, testing, and deploying [1]. The continuous approach facilitates a faster feedback loop and enables early detection and resolution of potential issues.

Monitoring. Monitoring metrics should be included in the DevOps pipelines to observe software runtime behavior, such as response time, throughput, and resource utilization [1]. This principle checks how new releases affect the stability of the system and provides insights on how the application can improve from both the development and operations aspects [35].

2.5.2 DevOps Practices

DevOps often implements the following practices to address the aforementioned principles:

Continuous integration. Continuous integration (CI) is a practice that advocates frequent and automatic integration of new code into existing codebase [37]. Ideally, the integration happens as frequently as any change is committed. CI systems typically comprise build pipelines and testing frameworks. The build pipelines automate the building and packaging of the application, while the testing frameworks run a series of tests on the built application. This approach validates the software is in a working state and enables quick identification and fixing of problems [36].

Continuous delivery and continuous deployment. Continuous delivery and continuous deployment, collectively abbreviated as CD, are the practices of continuously preparing software for deployment in production environments. Although both practices aim for frequent releases, they diverge in the extent of automation employed. Continuous delivery ensures the application is in a deployable state, while continuous deployment automates the deployment of the application [38]. While the nuanced distinctions are not central to the thesis, the overall concept of CD represents an extension of CI by guaranteeing the software changes are deployable and are indeed deployed [37]. In the modern DevOps landscape, CI and CD are often integrated into a unified and continuous pipeline known as CI/CD. Tools such as Jenkins [39] and GitLab CI/CD [40] are used for realizing such pipelines and enabling a complete workflow from code commit to production deployment.

Version control. Version control is the practice of tracking and managing code changes. Version control systems (VCS) facilitate this practice by providing tools and functionality for efficient tracking and management. VCS provides features such as branching and reverting, which encourages experimentation without the fear of disrupting the stability of the existing system. Moreover, VCS enhances visibility into the change history of the codebase, facilitating a more straightforward debug process. Presently, Git is the de facto standard VCS widely adopted in the software development community, owing to its robust features and flexibility.

Infrastructure as Code. Infrastructure as Code (IaC) is an approach to managing and provisioning infrastructure through code, typically declaratively. In the realm of IaC, the operation tasks are handled akin to application code [41]. This approach gains capability when combined with software development practices such as VCS and CI/CD. When changes are committed to the codebase, the automation CI/CD pipeline can trigger IaC tools to test and apply the modifications to the provisioned systems. This streamlined approach ensures consistency and reliability and reduces the risks and effort compared to manual configuration management [42]. Popular IaC tools include Terraform [43], Ansible [44], Puppet [45], and Chef [46]. These tools facilitate defining and managing infrastructure resources as code, simplifying provisioning, configuration, and overall management.

Configuration management. Configuration management (CM) involves storing, retrieving, tracking, and updating all project-related configurations and their interrelationships [36]. These configurations encompass software versions, security policies, networking settings, and OS versions on which applications run. Consistently storing and applying the same set of configurations ensures the uniform behavior of the managed systems throughout their lifecycle. Furthermore, by codifying configurations, CM becomes repeatable and standardized, which facilitates the integration of software development practices such as version control and CI/CD.

2.5.3 DevOps Benefits

By adopting the principles and practices of DevOps, we achieve the following benefits:

Better collaboration. DevOps dismantles traditional silos between development and operations teams, resulting in more seamless and effective communication. This approach fosters a culture of collaboration and shared responsibility throughout the SDLC. By encouraging the exchange of technical knowledge, DevOps enables rapid diagnosis and resolution of issues, thereby streamlining the workflows and improving productivity [47].

Improved product quality. The CI pipeline ensures that every change is automatically tested and verified before being promoted for delivery. This process allows for early detection and mitigation of bugs, thereby minimizing defects and vulnerabilities in the final product [48]. Moreover, as mentioned previously, better collaboration leads to easier cross-functional problem-solving, which contributes to the improvement of the overall product quality.

More frequent releases. The automated CI/CD pipeline seamlessly integrates the building, testing, and deployment processes. This integration reduces the effort required for releases and enabling frequent feature rollouts [48]. Additionally, small and frequent releases are less risky and cause fewer service outages [47].

Reliability and scalability. DevOps incorporates IaC and CM practices where infrastructure and configurations are managed as code. This approach leads to more consistent and automated deployments, improving the reliability and scalability of applications.

2.5.4 DevSecOps

DevSecOps extends the principles of DevOps by integrating security controls and processes throughout the SDLC and the associated automation pipelines. In contrast to traditional software development, where security has typically been an afterthought [35], this holistic approach promotes closer collaboration between development, security, and operations teams, reflecting the collaborative spirit of DevOps [49].

The emphasis on integrating security leads to practices such as shift-left security and continuous security assessment [50]. Shifting security to the left suggests integrating security practices early in the SDLC to identify and fix security issues early in the process. Building upon shift-left security, continuous security assessment emphasizes the ongoing nature of security within DevSecOps and refers to the continuous application of security practices throughout the SDLC. Examples of security practices in DevSecOps pipelines include software composition analysis (SCA), static application security testing (SAST), dynamic application security testing (DAST), and IaC scanning during the development phase, along with configuration drift detection and remediation during the operation phase.

2.6 GitOps

GitOps is an operational framework incorporating DevOps best practices, IaC, and VCS to optimize infrastructure and application management processes. As its name implies, the Git VCS serves as the central component in GitOps methodologies. GitOps leverages IaC stored within a Git repository, combined with automation pipelines, to form its foundational framework [1]. This section delves into the principles and practices of GitOps, explores the benefits of adopting GitOps, and reviews studies about GitOps workflows in the telecommunication domain.

2.6.1 GitOps Principles

The OpenGitOps community defines the key principles as a guideline of GitOps workflows [51]:

Declarative. The managed system should be described declaratively. Infrastructure and application configurations are expressed by the desired state rather than a sequence of instructions to achieve the state. This approach promotes flexibility and simplifies configuration management.

Versioned and immutable. The desired state is stored in a Git repository. The Git repository retains the complete version history and guarantees immutability with cryptographic hashes, meaning committed modifications cannot be altered.

Pulled automatically. Software agents automatically pull the desired state from the Git repository. This means tools within the GitOps workflow automatically retrieve the configurations from the Git repository and eliminate the need for manual configuration updates.

Continuously reconciled. Software agents continuously compare the actual state of the system and the desired state stored in the Git repository. If a discrepancy exists, the agent attempts to reconcile the system to the desired state. This ensures continuous delivery and a self-healing environment.

2.6.2 GitOps Practices

In addition to the DevOps practices, GitOps workflow often incorporates the following practices, guided by the aforementioned principles:

Declarative configuration management. A prevalent practice in GitOps involves utilizing YAML, a human-readable data serialization language commonly used for configuration files, to define the desired state of infrastructure and applications. For instance, Git repositories may contain Helm charts, which are pre-packaged collections of files for deploying and configuring applications in Kubernetes. These Helm charts use YAML files to declaratively configure the desired application deployments and configurations.

Git as the single source of truth. Storing all the desired states in a Git repository establishes a single authoritative source for managing and tracking changes. This centralized approach simplifies collaboration and ensures consistency. Practically, this means that all infrastructure and application configuration changes are codified and stored in a Git repository [2]. As a result, everyone must pull the latest configurations from the Git repository for deployments to ensure uniformity across the system.

Git workflow. Employing Git workflows with pull requests and code reviews ensures a controlled and secure approach to managing changes within a single source of truth. Pull requests present all changes made to the files in Git for code review and approval. The code review, which involves a second pair of eyes to scrutinize code changes, enhances both the quality and security of the updates [2]. Once the changes are reviewed and approved, they are merged back to the main branch and automatically applied to the system by the software agent. This combination of practices adds a layer of control and governance to the automated GitOps process.

Operators and self-healing mechanisms. An operator is a software agent designed to manage and automate the lifecycle of applications or infrastructure, particularly within Kubernetes environments. Operators reconcile the current state of the system with the desired state based on the configurations stored in Git [2]. This approach ensures a self-healing system where any discrepancies between the desired and actual states are automatically corrected without human intervention.

2.6.3 GitOps Benefits

Implementing GitOps practices aligned with GitOps principles offers several key advantages:

Familiarity with Git. Developers are generally already familiar with workflows involving Git such as feature branching or trunk-based development with code reviews. The familiarity allows organizations to benefit from GitOps without the burden of introducing new tools. DevOps teams can seamlessly integrate GitOps practices into their existing workflows, promoting smoother collaboration and faster adoption.

Visibility and traceability. GitOps provides comprehensive visibility and traceability by tracking every change through the Git history. Each commit serves as a complete snapshot of the system state, which offers clear insight into system evolution. Moreover, the Git history includes commits detailing what was changed, who made the change, when it was made, and the reasons for it. The commits are associated with cryptographic hashed checksums for data integrity[1], with the option for users to sign the commits using their GPG (GNU Privacy Guard) key for added authenticity.

Auditability and compliance. GitOps facilitates auditability and compliance by maintaining an audit trail of all changes made to the infrastructure and applications in the Git history. This allows organizations to

[1]Currently, Git uses the SHA-1 algorithm to compute the checksum. However, as SHA-1 has been considered unsafe since 2017 [52], Git is moving toward the more-secure SHA-256.

verify compliance with industry standards and regulations and ensures they meet regulatory requirements [2].

Disaster recovery. With GitOps, organizations have a complete definition of their environment stored in Git, which enables rapid recovery in the event of a disaster [2]. In addition, the revert feature of Git provides stable and reproducible rollbacks to previous states in case of bugs or errors [53], thereby minimizing the mean time to recovery (MTTR).

Simple replication. With all system configurations stored in Git, initiating a new replicated environment is straightforward. GitOps reduces the complexity of creating experimental or production-like environments. Users can expect a consistent experience on freshly deployed systems, which encourages further exploration and optimizations of system operations.

Enhanced security. GitOps enhances security by enforcing rigorous change review processes for infrastructure and application configurations. Additionally, Git has the capability to sign changes, thereby providing cryptographic proof of authorship and ensuring codebase integrity. Furthermore, continuous monitoring and reconciliation of configuration drift strengthen security, ensuring the infrastructure remains consistent and secure.

2.6.4 Related Work in Telecommunication

The telecommunications industry is experiencing a growing adoption of GitOps for automating and managing software deployments. Wörndle et al. [53] demonstrates that through the utilization of GitOps, communication service providers can augment automation and expedite the introduction of new features and updates into their networks, including crucial security patches.

In a study conducted by Scotece et al. [54], the authors emphasize the potential of leveraging DevOps and GitOps practices by telecommunication operators to streamline their deployment processes and reduce operational costs. Moreover, GitOps principles offer a viable approach to managing the lifecycle of CNFs within 5G core networks.

Additionally, Bonati et al. [55] introduce an automation framework based on OpenShift and the GitOps workflow. The framework is capable of automatically testing and deploying software-defined end-to-end 5G and O-RAN-compliant systems instantaneously.

Lastly, Leiter et al. [56, 57] explored the integration of Kubernetes Operators and GitOps principles to enhance automation and efficiency of NF configuration management in cloud-native environments. Their research explains the potential of developing software extensions to streamline NF configuration processes with GitOps methodologies, offering insights into optimizing network management practices with GitOps.

Chapter 3

System Study

In this chapter, we introduce the Network Security Configuration Manager (NSCM), an existing system we used in this study. NSCM is a software designed to monitor and enforce security-related configurations of network nodes. It targets mobile network operators (MNO) managing networks at a substantial scale, as well as mission-critical network operators, such as government agencies operating public safety networks. NSCM is inspired by Ericsson Security Manager (ESM) [4], a cybersecurity platform for 5G networks, and it implements a subset of its features.

NSCM serves as a security management system, which provides visibility and basic automation for network security configurations. It can also be considered a part of the DevSecOps pipelines of a network application on the operation side as it reconciles and remediates configuration drifts of the application during its operation.

The introduction of the system follows a bottom-up approach. Initially, we define the components and the terminologies. Subsequently, we describe how these components interact to realize the features of NSCM. Finally, we discuss the requirements and constraints for integrating a GitOps workflow into such a system.

3.1 NSCM Components and Terminologies

In this section, we present the components and terminologies of NSCM.

3.1.1 Network Node and Node Type

A network node refers to a network element that is managed by NSCM. The network node is a physical or software element of the network that ranges from infrastructure to applications, provided that it is reachable within the network. Examples of network nodes include physical devices such as routers, switches, and firewalls. They can also be network functions and exist in either physical (PNF), virtual (VNF), or cloud-native (CNF) forms. Additionally, entire systems such as Kubernetes clusters or virtual machines can function as network nodes.

Each network node is associated with a node type. The node type defines the category of the node and the operation, administration, and maintenance (OAM) interfaces it exposes. NSCM interacts with the nodes using the OAM interfaces. For instance, an Access and Mobility Management Function (AMF) in a 5G core network may have the node type `core-cnf` and expose a NETCONF interface to monitor and modify its configurations. Similarly, an OpenShift Kubernetes cluster may be categorized as a `kubernetes-cluster` and can be accessed via Kubernetes APIs with tools like `kubectl`.

Additionally, users can attach labels, which are simple strings, to network nodes. Labeling is useful for grouping network nodes with common properties beyond node types, such as geolocations or specific functions. The grouping is helpful when assigning security configurations. For example, users can label base stations in the Helsinki region as `helsinki`, which can then be used to apply region-specific security policies and configurations.

3.1.2 Security Control and Policy

Security control is a measure to protect the confidentiality, integrity, or availability[1] of network nodes. It can be Python scripts, Ansible playbooks, or Shell commands that are executable on the targeted nodes. Each security control supports one or more node types and consists of security variables. The variables define configurable values the control will enforce. For example, a security control concerning password complexity may support all Linux-based nodes and have two security variables: an integer defining the minimum length of the password and a boolean value regarding whether the password should contain special non-alphanumeric characters.

A policy is a set of security controls that serve the same purpose. Policies are usually developed by vendors or MNOs. These policies align with industrial standards and adhere to regulations. Based on the previous example, the password complexity security control may belong to the password management policy. Another security control in the policy could be password lifetime security control, which regulates the maximum password age.

[1]These properties are commonly referred to as the CIA triad, a model to guide information security systems.

The implementation of the security controls and policies is out of the scope of the thesis. We use controls and policies developed by Ericsson to conduct the experiment.

3.1.3 Security Baseline

A security baseline binds security controls with network nodes, creating a many-to-many relationship between them. The binding expresses the intention to enforce certain controls on a group of network nodes. For example, we can define a security baseline named `nf-password` that associates two node types, `core-cnf` and `vnf`, with the password management policy. Similarly, another baseline named `helsinki-log-level` could bind nodes with the label `helsinki` with a logging level security policy.

From the perspective of an MNO, security baselines represent the security objectives they want to achieve on their networks. Figure 3.1 shows the relationship between the components.

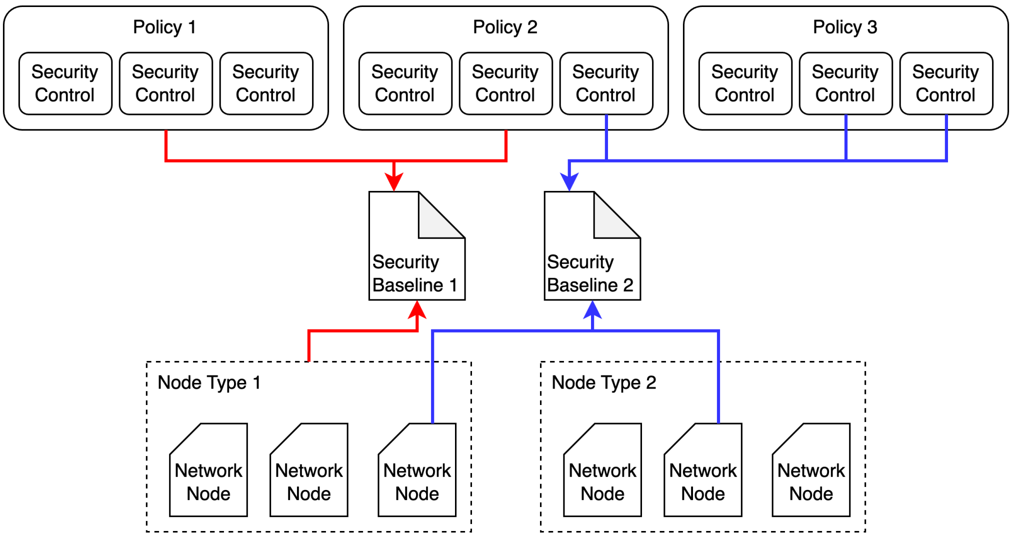

Figure 3.1: Example security baselines. In this example, security baseline 1 associates policies 1 and 2 with node type 1, while security baseline 2 binds two network nodes with three security controls.

3.2 NSCM Features

In this section, we introduce the features provided by NSCM, all of which are accessible through a graphical user interface (GUI) and REST APIs. However, the design and operation of the GUI fall outside the scope of the thesis. While our solution is based on the REST APIs, understanding the existence of the GUI provides context when comparing the GitOps workflow with existing traditional workflows.

3.2.1 Network Node and Policy Management

NSCM stores network nodes, policies, and security baseline information in a database and provides basic create, read, update, and delete (CRUD) features. For network node management, users can manually create network nodes within NSCM with the appropriate node types and export nodes from and import nodes to NSCM in batches. Furthermore, NSCM is capable of integrating with other management tools to automatically discover nodes in the network. Nonetheless, such tools are out of the scope of the thesis.

For policy management, users can import existing policies or create custom policies. Users can then bind them to network nodes by defining security baselines. The network node and policy management features provide the foundation of security management. They enable the policy enforcement and compliance check features, which we will introduce shortly.

3.2.2 Policy Enforcement

NSCM executes policy enforcement based on a given security baseline. Once started by a user, NSCM will first query the database for the values of control variables in the security controls and the associated network nodes defined in the security baseline. Then, NSCM will access the network nodes using the OAM interfaces and execute pre-defined scripts or commands to enforce the configuration.

The feature automates the configuration of network nodes. The traditional approach of individually logging into network nodes for manual configuration is no longer required, saving man-hours and reducing the risk of human errors.

3.2.3 Compliance Check

Compliance is security specifications defined in policies that network nodes should adhere to. NSCM ensures the compliance of the nodes it manages.

Similar to policy enforcement, a compliance check is run against a security baseline. Compliance check monitors the actual configurations on the network nodes and reports when they deviate from the desired configurations defined in the policies.

The feature is useful for MNOs to detect configuration drifts within their networks. Configuration drifts may occur due to manual operational errors or, more severely, malicious activities. The feature enables MNOs to respond rapidly to potential security incidents.

3.3 Change Management with NSCM

When a user in an MNO wants to adjust security configurations in the network through NSCM, they typically initiate a change request in an out-of-band ticketing system. The content and the rationale for the changes are submitted to the system, awaiting approval from authorized personnel. Once approved, the user, who may or may not be the same individual as the one who initiated the change request, must manually or programmatically apply the changes to NSCM. After the changes have been applied, the user has to trigger the policy enforcement feature in NSCM to deploy the changes to the nodes.

As illustrated, although NSCM has reduced much work by automating policy enforcement across numerous nodes, the change management process remains lengthy and not fully automated. The process is prone to human errors, a challenge we address by integrating a GitOps workflow.

3.4 GitOps Integration

This section details the integration of a GitOps workflow into NSCM. By adhering to GitOps principles, we outline the requirements of integrating GitOps into telecommunication security operations, including defining declarative file formats and implementing software agents to realize the reconciliation loop. We present a high-level comparison between the traditional workflow with NSCM and the GitOps workflow. Furthermore, we discuss the necessary adaptations and customizations due to the unique requirements of NSCM as well as the challenges and benefits of this integration.

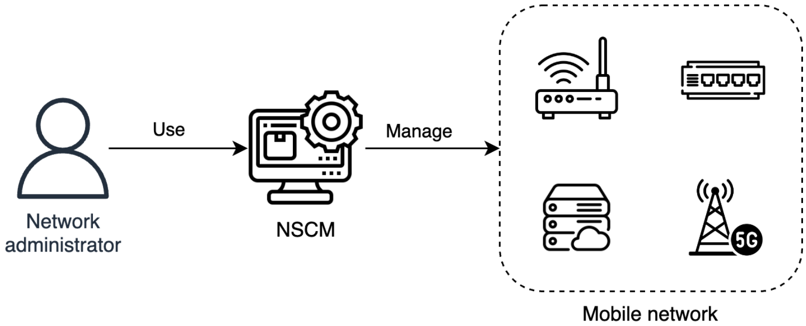

Figure 3.2: Traditional workflow of NSCM.

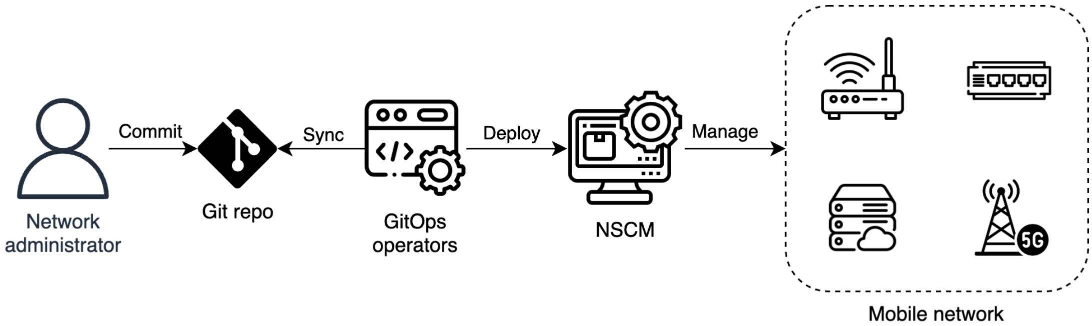

Figure 3.3: GitOps workflow of NSCM.

3.4.1 GitOps Components in NSCM

The primary distinction between the traditional workflow and the GitOps workflow is the interaction point for the user, typically a network administrator. In the traditional workflow of NSCM, illustrated in Figure 3.2, the network administrator directly interacts with NSCM, then NSCM manages and configures the network nodes within the mobile network.

In contrast, in the GitOps workflow, the network administrator commits configuration changes to a Git repository. The GitOps operators then synchronize these changes, deploy them to NSCM, and ensure that the desired state is consistently maintained in the mobile network. Figure 3.3 illustrates the integration of Git repositories and GitOps operators to streamline the management process.

3.4.2 Declarative Configurations in Git

To adopt a GitOps workflow in NSCM, we approach the problem by following the GitOps principles, specifically the *declarative* and *versioned and* *immutable* principles, which suggest that configurations should be stored declaratively in Git. To fulfill these principles, we need to represent the network nodes, the security controls and policies, and the security baselines in a declarative file format, such as YAML or JSON. This data is currently stored in a relational database, presenting the initial challenge of defining the appropriate fields and the level of details to include in the declarative files. The files should contain sufficient detail to accurately describe the resources and enable GitOps tool to construct a concrete execution plan while remaining abstract enough to avoid overwhelming the high-level intent with low-level specifics [58].

For example, to define a password length control in a user-facing YAML file[2], `checkPasswordLength:true` alone might be too vague, while detailing the exact steps to check the password length, such as which file to inspect and how to parse the file[3], might be excessively verbose. The optimal amount of information could be to include a field specifying the required length, as demonstrated in Listing 3.1.

Listing 3.1: Example YAML file for a security control on minimum password length.

```yaml
security-controls:
  - name: password minimum length
    variables:
      - length: 12
```

3.4.3 Software Agent

The *pulled automatically* and *continuously reconciled* principles suggest that we implement software agents in NSCM to continuously apply the latest desired state. Traditionally, GitOps tools such as Flux [59] and Argo CD [60] are used to deploy applications to Kubernetes environments and are considered a part of the CD pipeline. These tools install software agents, often referred to as GitOps operators, in the Kubernetes cluster and run reconciliation loops as depicted in Figure 3.4. The GitOps operators are named to emphasize their role in capturing the operational knowledge of human operators responsible

[2] Relying solely on the password is not a security best practice for authentication; production systems should utilize frameworks such as public key infrastructure (PKI) to establish secure communication between services and multi-factor authentication (MFA) for human users.

[3] On modern Linux, we can check the password length requirement by inspecting whether the file `/etc/pam.d/common-password` has a `minlen=` option and its value for the module `pam_unix.so`.

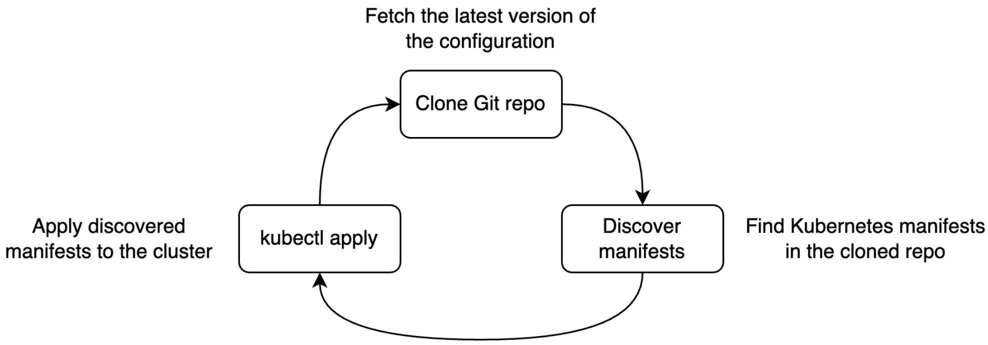

Figure 3.4: The reconciliation loop of a Kubernetes GitOps operator [2].

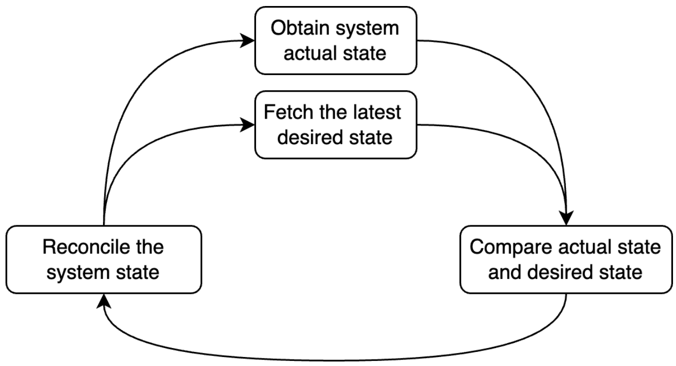

Figure 3.5: The generalized reconciliation loop of a GitOps operator.

for managing the applications and their configurations [61]. Note that operators in the GitOps context refer to the software agents that implement the reconciliation loop, not telecommunication operators. Although GitOps is often related to Kubernetes, the focus of this thesis is to further automate the management and enforcement of configurations in NSCM. As we are not deploying applications to a Kubernetes environment, existing GitOps tools do not fit exactly in this situation.

Nevertheless, we can generalize the reconciliation loop in the Kubernetes GitOps tools. The tasks of the GitOps operators include fetching the latest desired state from a Git repository, monitoring the actual state of the system, comparing the actual and desired state, and reconciling the system state accordingly. Figure 3.5 shows the generalized flow of a GitOps operator. Note that the obtaining and reconciliation of system state are not shown in Figure 3.4 since they are handled by Kubernetes.

From the generalized workflow, we can incorporate existing features in NSCM to form the complete reconciliation loops for the operators. The first operator is the network node and policy management operator, which syncs

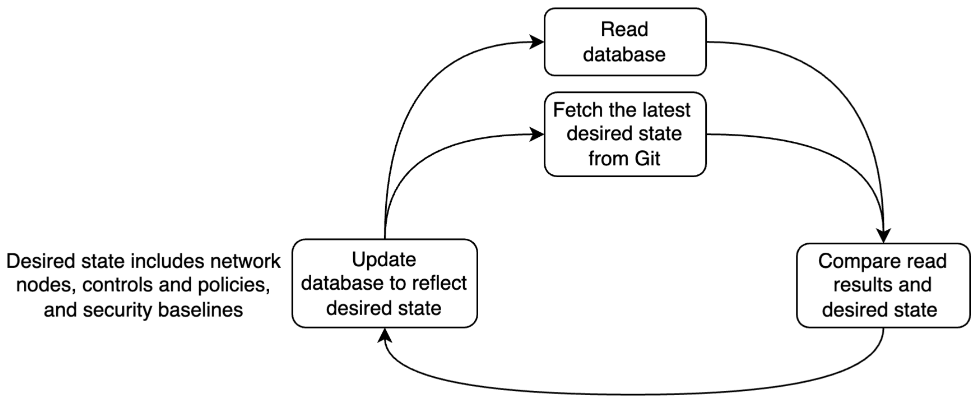

Figure 3.6: The reconciliation loop of the network node and policy management GitOps operator.

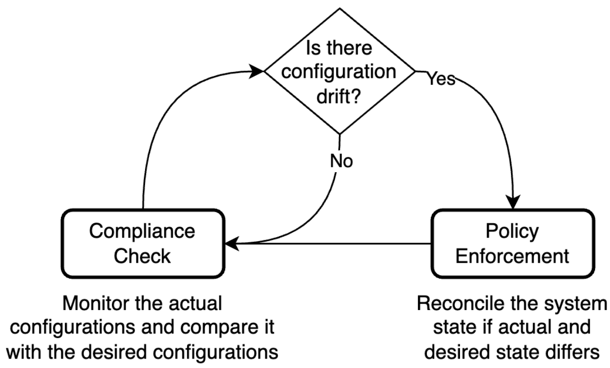

Figure 3.7: The reconciliation loop of the configuration enforcement GitOps operator.

the information stored in Git with NSCM. The operator utilizes the CRUD feature to integrate a GitOps workflow. Figure 3.6 shows the workflow.

The second operator is the configuration enforcement operator, which leverages Compliance Check and Policy Enforcement to ensure security configurations are in position continuously. Compliance Check monitors the actual state and compares it with the desired state in the database. Policy Enforcement applies configurations to the system. Figure 3.7 illustrates the reconciliation loop of the NSCM configuration enforcement GitOps operator, where existing features are marked with boxes with emphasized borders. The loop is rather straightforward as Compliance Check and Policy Enforcement run based on the information stored in the database, which is kept in sync with the Git repository through the network node and policy management operator.

In conclusion, our NSCM GitOps workflow has two major requirements. Firstly, we must define a declarative model to describe the resources including network nodes, controls, policies, and security baselines. Secondly, we need to implement software agents that continuously apply the latest state and configurations to the NSCM system and the environment. However, the primary constraint is that existing GitOps tools do not completely fit our specific use case, necessitating the development of a customized solution.

Chapter 4

Design and Implementation

In this chapter, we elaborate on the details of the implementation of integrating the GitOps workflow into the NSCM. Building on the conclusions drawn in the previous chapter, we recognize the necessity of designing a declarative model to describe the resources in NSCM and developing customized GitOps operators to implement the reconciliation loops.

Therefore, this chapter outlines the file format that describes the resources, the steps involved in the reconciliation loop, and the required configuration of the GitOps operators to achieve the desired functionalities. Additionally, we discuss the validation and error handling in the GitOps operators when invalid or conflicting configuration files are presented.

4.1 YAML File Format

In this section, we provide an overview of the YAML file format used to describe the desired states of network nodes, security policies, and security baselines. We select YAML as the file format since NSCM runs on Kubernetes, and users of NSCM are typically familiar with YAML. To enhance understanding, we provide examples for each resource type. The complete schemas are detailed in Appendix A for reference.

The file format is designed to balance the inclusion of sufficient details to express user intent and enable the GitOps operators to generate concrete execution plans while abstracting away implementation details to reduce noise.

4.1.1 Network Nodes

Network nodes are network elements within the network that are associated with node types. The most important properties of a network node are its node type and the OAM interfaces it exposes. The node type determines the available controls for the network node, while the OAM interface specifies how the NSCM will interact with and manage the node.

Listing 4.1 shows an example of a YAML file that lists two network nodes. The first node is a 5G CNF node with an SSH interface, and the second node is a 4G VNF node that exposes a NETCONF interface. The schema for the network node YAML file is detailed in Listing A.1 using JSON Schema [62].

Listing 4.1: Example YAML file for network nodes.

```
network_nodes:
  - name: Test 5G CNF 1
    description: A 5G CNF that runs in the cloud
    node_type:
      name: 5G-CNF
      version: 1.0.0
    interfaces:
      - name: 5G CNF SSH
        interface_type:
          name: 5G CNF SSH
        ip_address: <ip_address>
        port: 22
  - name: Test 4G VNF 1
    description: A 4G VNF that runs in the VM
    node_type:
      name: 4G-VNF
      version: 1.0.0
    interfaces:
      - name: 4G VNF NETCONF over SSH
        interface_type:
          name: 4G VNF NETCONF over SSH
        ip_address: <ip_address>
        port: 830
```

4.1.2 Security Policies

Security policies are sets of security controls designed to achieve specific security objectives. Each security control comprises one or more security variables that determine the criteria of such control.

Listing 4.2 shows an example YAML file that defines a *Password management policy* with security controls concerning password complexity and lifetime. The schema for the security policy YAML file is detailed in Listing A.2.

Listing 4.2: Example YAML file for security policies.

```yaml
security_policies:
  - name: Password management policy
    controls:
      - control: Password complexity control
        variables:
          - min_length: 12
          - special_char: true
      - control: Password lifetime control
        variables:
          - max_age: 90
```

4.1.3 Security Baselines

Security baselines associate security policies and controls with network nodes. Users have the flexibility to select controls and policies by their names, and they can customize values for control variables as needed. In addition, network nodes can be chosen based on their types, names, and labels.

Listing 4.3 illustrates a YAML file containing two security baselines. The first baseline links the *Password management policy* with all nodes having the node type of `5G-CNF` or are labeled `5g-core`. The second baseline binds two password-related controls with nodes whose names are equal to `MME`[1] or match the regular expression `^Test 4G VNF [0-9]+$`, for instance, `Test 4G VNF 1` and `Test 4G VNF 15`.

In the current implementation, selectors that begin with ^ and end with $ are treated as regular expressions and matched using the Python module `re` module. Selectors not enclosed by these symbols are treated as simple strings and matched with exact equality.

It should be noted that if multiple selectors are specified in the `bindings` block, these selectors form a union relationship. For example, consider the first security baseline in Listing 4.3, which selects network nodes that are either of type `5G-CNF` *or* are labeled `5g-core`. The same logic applies when both `security_policies` and `controls` are presented in the same baseline. The security baseline will include all controls under the specified policies as well as all listed security controls. The schema for the security baseline YAML file is detailed in Listing A.3.

[1] The Mobility Management Entity (MME) is a network function in 4G networks that manages user equipment mobility, including authentication, session setup, and handovers between base stations.

Listing 4.3: Example YAML file for security baselines.

```yaml
security_baselines:
  - name: 5G core CNF password management baseline
    bindings:
      node_type:
        - 5G-CNF
      labels:
        - 5g-core
    security_policies:
      - policy: Password management policy
  - name: 4G VNF password management baseline
    bindings:
      network_nodes:
        - MME
        - ^Test 4G VNF [0-9]+$
    controls:
      - control: Password complexity control
        variables:
          - min_length: 8
          - special_char: false
      - control: Password lifetime control
        variables:
          - max_age: 120
```

4.2 Network Node and Policy Management Operator

The goal of the network node and policy management operator is to sync the desired state stored in Git to the running system, which is the NSCM in this study. The state includes network nodes, security policies, and security baselines that bind the nodes and the policies.

The operator implements a reconciliation loop comprising three high-level steps: fetching the desired state from the Git repository and the actual state from the system, comparing the desired and actual states, and taking action to eliminate any deviations between them before starting the loop anew. The operator runs at predefined intervals, such as every 30 seconds, to ensure continuous reconciliation of the desired and actual states. For simplicity, the operator is currently implemented as a standalone Python program that runs in an environment with network access to the NSCM system. Eventually, it will be integrated as one of the microservices within the NSCM system.

Figure 4.1 and 4.2 provide detailed illustrations of these steps from different perspectives. Figure 4.1 shows the flowchart of the reconciliation loop, while Figure 4.2 elucidates the information flow between components.

4.2.1 Configuring the Operators

To enable the GitOps operators to function, we must provide them with the location of the NSCM and the repositories, along with the necessary credentials through environmental variables[2]. In addition, we specify the branch and path in a repository where the corresponding files are located. These configurations are stored in YAML files. Listing 4.4 shows a sample configuration file. In this example, network nodes, security policies, and security baselines are stored in different repositories.

Listing 4.4: Example configuration file for NSCM GitOps operators.

```yaml
nscm:
  url: <domain name or IP>
  username: ${NSCM_USERNAME}
  password: ${NSCM_PASSWORD}
network_node:
  repo_url: https://repo.site/username/network-node.git
  branch: main
  path: /
  username: ${NODE_REPO_USERNAME}
  password: ${NODE_REPO_PASSWORD}
security_policy:
  repo_url: https://repo.site/username/policy.git
  branch: main
  path: /
  username: ${POLICY_REPO_USERNAME}
  password: ${POLICY_REPO_PASSWORD}
security_baseline:
  repo_url: https://repo.site/username/baseline.git
  branch: main
  path: /
  username: ${BASELINE_REPO_USERNAME}
  password: ${BASELINE_REPO_PASSWORD}
```

[2]Currently, the credentials are stored in a file that should not be committed in Git, with each line formatted as `<key>=<value>`. We then use the command `env $(cat credentials | xargs) python3 nscm_gitops_operator.py` to inject the environmental variables. In a production environment, a proper secret management tool should be used to manage sensitive information centrally.

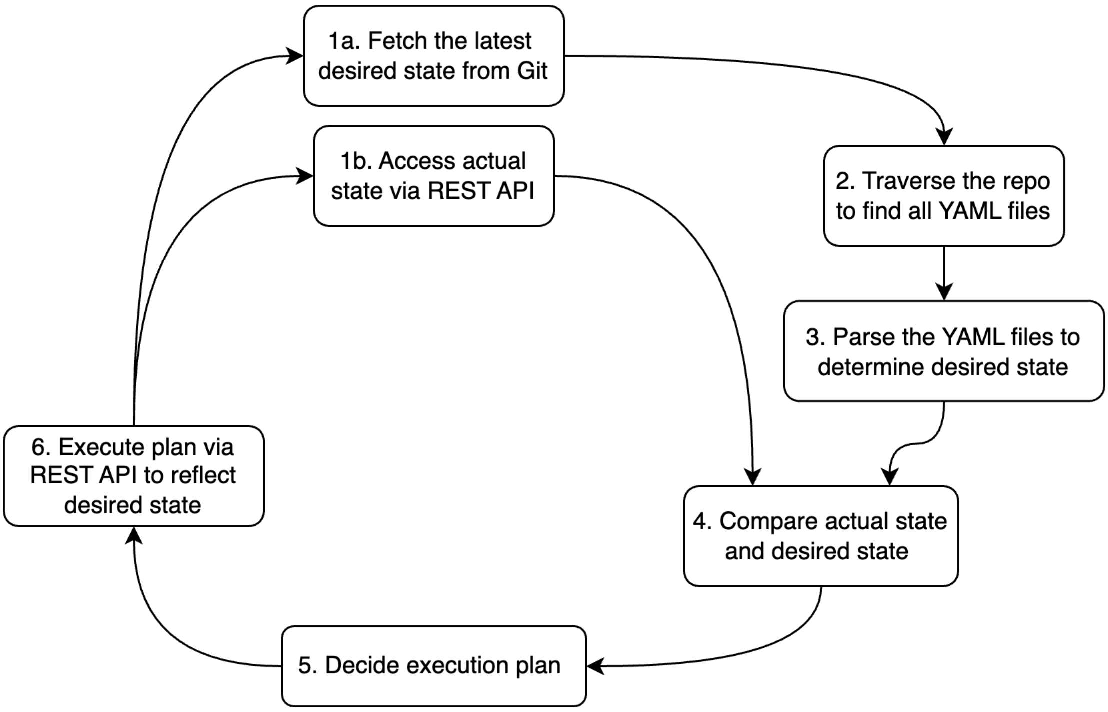

Figure 4.1: NSCM network node and policy management operator reconciliation loop flowchart.

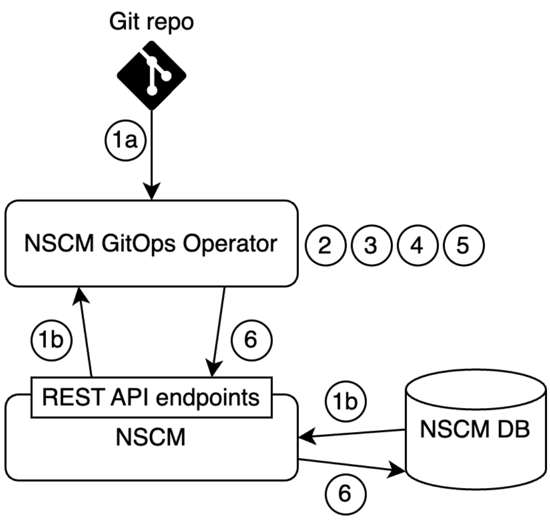

Figure 4.2: NSCM network node and policy management operator information flow. The numbering corresponds to the sequence of events depicted in Figure 4.1.

In the example above, the repository setting is configured per resource, meaning each resource has a distinct repository to store the desired state. However, users can also opt to place all resources in the same repository and organize them into different folders. This approach may be more reasonable as configurations in `network_node` and `security_policy` are closely related to those available in `security_baseline`. In this setup, the `repo_url`, `username`, and `password` fields would likely be identical for all the resources. Thus, we introduce the `default` block in the operator configuration to reduce repetition, as shown in Listing 4.5.

It is worth mentioning that users can overwrite values in the `default` block by specifying different values in the resource-specific blocks. The flexibility is beneficial when applying special branching strategies or when different credentials are required for different parts of the repository. The design choice provides users with the convenience and flexibility needed to tailor the configuration to their specific requirements.

Listing 4.5: Example configuration file with default block.

```
nscm:
  url: <domain name or IP>
  username: ${NSCM_USERNAME}
  password: ${NSCM_PASSWORD}
default:
  repo_url: https://repo.site/username/nscm-gitops.git
  branch: main
  username: ${REPO_USERNAME}
  password: ${REPO_PASSWORD}
network_node:
  path: nodes/
security_policy:
  path: policies/
security_baseline:
  path: baselines/
```

It is important to note that, in the example above, users should place only deployable YAML files in the specified paths. In practice, however, repositories might contain other YAML files that are not intended for GitOps operations. This can be addressed by checking the schema of the YAML files before deployment or by using a `.gitignore`-style file to exclude certain file patterns. While addressing the flexibility issue is beyond the scope of the thesis, it remains essential when productizing the operators.

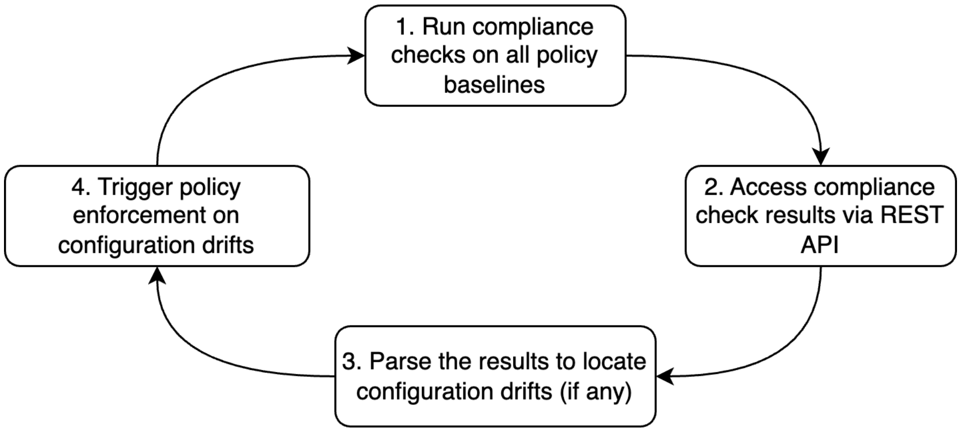

Figure 4.3: NSCM configuration enforcement operator reconciliation loop flowchart.

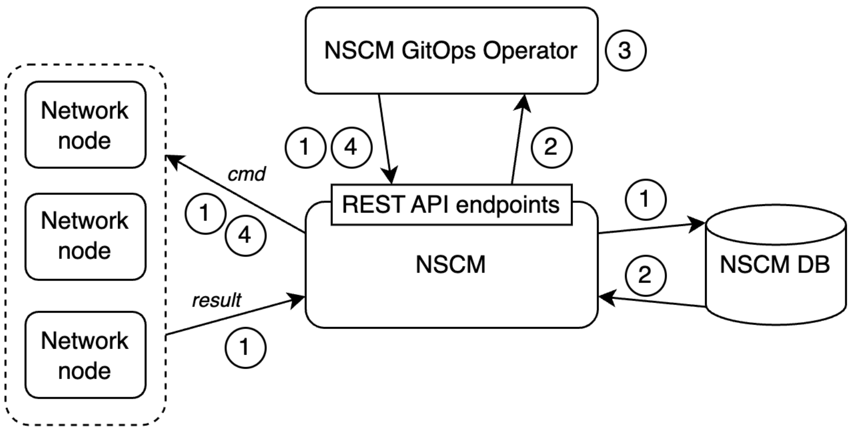

Figure 4.4: NSCM configuration enforcement operator information flow. The numbering corresponds to the sequence of events depicted in Figure 4.3.

4.3 Configuration Enforcement Operator

The objective of the configuration enforcement operator is to ensure that the policies and controls defined in security baselines are enforced on the targeted network nodes. The reconciliation loop of the operator is illustrated in detail in Figure 4.3, while Figure 4.4 demonstrates the information flow within the operator.

No additional configurations are necessary for the configuration enforcement operator. It relies on the NSCM location and credentials provided, which are reused from the settings of the network node and policy management operator.

4.4 GitOps Workflow

With the operators in place, we demonstrate the GitOps workflow within NSCM. The final component of the workflow is a Git server that offers code review and pull request functionalities. The Git server can be either self-hosted or as a service. Popular Git server solutions include GitHub [63], GitLab [64], Gerrit [65], and Gitea [66]. Our solution is based on GitLab, as it offers a self-hosted option that is suitable for storing sensitive information locally during experiments. Additionally, GitLab provides comprehensive code review and pull request tools and user-friendly interfaces, making it well-suited for our needs.

In the workflow, users initiate changes by modifying the declarative configuration files and committing them to the Git history. Subsequently, they create pull requests to provide the rationale behind the changes and begin a code review process. Once the changes are reviewed and approved, they are merged into the target branch, typically the main branch. As discussed earlier, the GitOps operators ensure the latest changes on the main branch are synchronized with the NSCM database and, consequently, enforced on the network nodes it manages. Figure 4.5 shows the flowchart of the GitOps workflow.

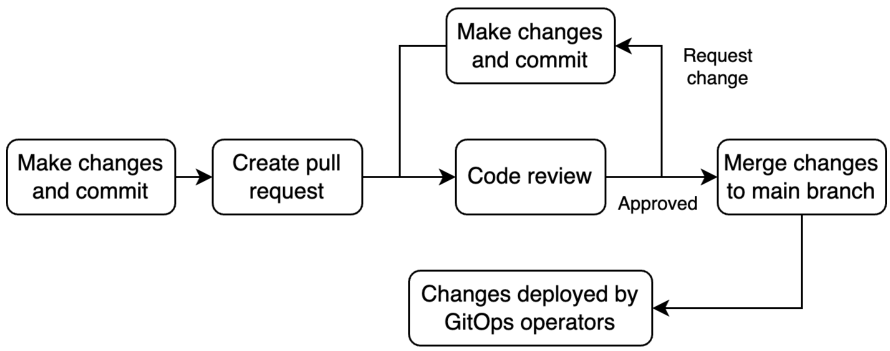

Figure 4.5: NSCM GitOps workflow.

In the event of a network incident caused by recent changes, users might need to roll back the network configurations to a previous stable state. In such situations, users can utilize the `git revert` command to undo the changes. This can be done either directly on the target branch in case of emergency or by following the pull request and code review processes illustrated in Figure 4.5 for better control and oversight, ensuring that the rollback is thoroughly reviewed and validated before it is applied.

4.5 Validation in the Workflow

Given that resource YAML files are user-provided inputs, there is a probability that these files may not adhere to the predefined format or may contain erroneous content. Therefore, it is essential to validate and check these files before applying them to NSCM.

Instead of applying changes directly to NSCM and notifying users of errors or warnings as they occur, which could lead to invalid content being merged into the main branch and potentially create a slow and expensive remediation process, a better approach is to implement validation in the CI pipeline of the resource repositories. This allows reviewers to be informed about the validity of changes while they are still in the pull request stage. As mentioned previously, our solution, hosted on GitLab, provides comprehensive functionalities to build a CI pipeline and perform code reviews. This ensures that only validated changes are merged to the main branch.

In this section, we outline the checks that can be performed in the CI stage to validate resource YAML files. Given that the resource CI cannot access the running NSCM system, all the available checks are static. Additionally, we discuss errors that can only be identified at runtime and how these situations are handled.

4.5.1 Validating YAML Files with Schemas

The first step of validation involves checking the files against their respective schemas. The schemas of the resources are in JSON format and can be found in Appendix A. We use the `PyYAML` package [67] to read the YAML files and the `jsonschema` package [68] to validate them against their schemas. Listing 4.6 demonstrates a sample program that can be executed in the CI pipeline to ensure compliance with these schemas.

This validation step can identify various types of errors in the YAML files, such as syntax errors, missing required fields, incorrect data types, and invalid values. If the YAML files contain invalid syntax or do not adhere to the schema, the program will fail with a detailed error message This allows for early detection and resolution of the issues.

Listing 4.6: Sample Python script to validate YAML file content.

```
import json

from jsonschema import validate
import yaml

```

```
yaml_file = "/path/to/yaml/file"
schema_file = "/path/to/schema/file"

# exception is raised when yaml_file has syntax errors
instance = yaml.safe_load(open(yaml_file, "r"))
schema = json.load(open(schema_file, "r"))

# exception is raised when the instance does not adhere
    to the schema
validate(instance=instance, schema=schema)
```

4.5.2 Logical Error Checks

In addition to schema validation, we also check for logical errors in the YAML files and provide detailed reasons for any failures:

Duplicate network nodes. Since multiple YAML files might define network nodes, there is a risk of unintentionally duplicating nodes across different files. We check for uniqueness by verifying that the combination of the node name and node type is unique across all files.

Duplicate security controls. A policy can contain numerous controls, making it difficult for users to notice duplicate controls within the same policy. We ensure uniqueness by checking that each control name is unique within a policy.

Overlapping policies. In a security baseline, users can specify both policies and individual controls. However, it is possible that some specified controls are already included in the specified policies. To address this, we list all controls within the specified policies and check for any overlaps with the individually specified controls.

Invalid node selection. In a security baseline, users specify criteria for selecting network nodes. Nevertheless, these criteria might not correspond to any network nodes in the system. To validate this, we first list all existing network nodes in the configurations and then search for nodes based on the selectors given in the security baseline. If no nodes match the given criteria, the security baseline is considered invalid.

Conflicting policies. A network node can be bound to multiple baselines, and sometimes these baselines may contain the same control with different variable values. These distinct values might not always result in a conflict. For example, if network node A is bound to security baseline 1, which specifies the minimum password length of 10, and to security baseline 2, which specifies the minimum password length of 12, there is no contradiction. Security baseline 2 is simply stricter than baseline 1. However, conflicts arise in cases where two baselines specify divergent SSH login messages, as following one baseline would violate the other. Due to the context-dependent nature and semantics of these controls, it is challenging to automatically determine conflicts. Therefore, we raise warnings about potential conflicts but leave the final decision to the users.

It is worth noting that the checks for *duplicate network nodes*, *invalid node selection*, and *conflicting policies* only apply to static nodes defined in the YAML files. For more dynamic nodes, such as ephemeral containers or pods in Kubernetes, these checks are more challenging to implement. Further research and development are required to effectively handle dynamic node configurations.

4.5.3 Limitations of Static Validation

While static checks are useful to ensure the correctness of the changes and to gain confidence for the reviewers, they cannot capture all potential errors. Many of the issues mentioned in this section stem from the fact that the controls are developed by third parties, such as vendors or MNOs, and we use them without having direct oversight. In this section, we enumerate errors that can only be identified in runtime:

Unsupported node type. Controls specified in a security baseline may not support certain nodes in the same baseline, and this incompatibility can only be detected when the GitOps operators attempt to deploy the security baselines to NSCM.

Nonexistent control variables. Control variables specified in the configuration files might not actually exist within the controls. Users will be notified when the GitOps operators fail to deploy these configurations to NSCM.

Invalid control variable values. Some control variables have specific value ranges. For example, the password maximum lifetime control might have a variable specifying the allowed maximum lifetime in days, which is required to be a positive integer. However, the CI pipeline cannot validate these requirements and thus cannot flag invalid values.

4.5.4 CI Pipeline in Self-Hosted GitLab

To create a validation pipeline in a self-hosted GitLab, we first need to register machines[3] where the CI jobs will be executed. Next, we define the CI pipeline using the `.gitlab-ci.yml` file. Listing 4.7 demonstrates a sample CI definition where `schema-test` validates the YAML files against their respective schemas and `conflict-policy-test` examines the policies in the baseline to identify potential conflicts.

Listing 4.7: Sample `.gitlab-ci.yml` file for validation in CI pipeline.

```
stages:
  - test

schema-test:
  stage: test
  image: python:3.12
  before_script:
    - pip install PyYAML jsonschema
  script:
    - python3 tests/schema_test.py --node-path nodes/ --policy-path policies/ --baseline-path baselines/

conflict-policy-test:
  stage: test
  image: python:3.12
  before_script:
    - pip install PyYAML
  script:
    - python3 tests/conflict_policy_test.py --node-path nodes/ --baseline-path baselines/
```

Both jobs will be triggered whenever a new commit is made in this example. If the tests do not pass, a red cross appears to indicate the failure. Figure 4.6 shows an example of a failed pipeline (`conflict-policy-test` job is omitted in this pipeline for simplicity) in a merge request attempting to merge branch `update-conf` into `main`. By clicking on the failed pipeline, we can trace the logs and identify the areas where the test failed, as illustrated in Listing 4.8.

[3]We use the Docker executor to run CI jobs in separate and isolated containers to ensure a clean and reproducible environment for each job. Please refer to the documentation for more details on setting up a Docker executor: `https://docs.gitlab.com/runner/executors/docker.html`.

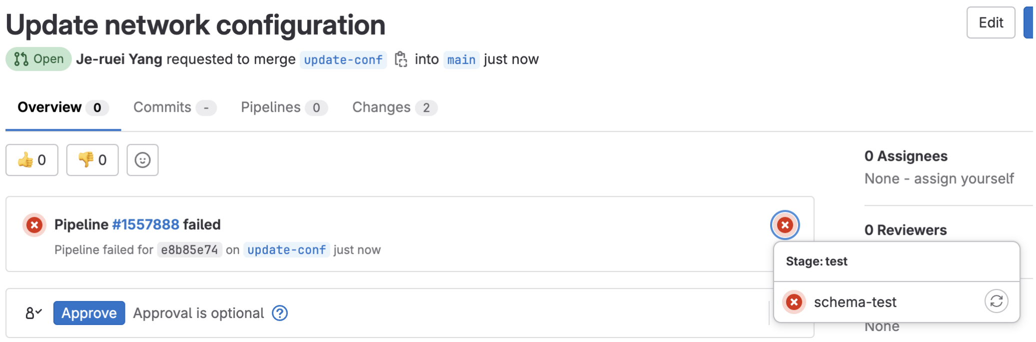

Figure 4.6: Failed schema validation in a merge request.

Listing 4.8: Sample CI pipeline log for validation.

```
<setup logs omitted>
$ python3 tests/schema_test.py --node-path nodes/ --
   policy-path policies/ --baseline-path baselines/
Validating nodes/5g-cnf-test-nodes.yaml...
YAML syntax error: nodes/5g-cnf-test-nodes.yaml
<traceback omitted>
Failed

Validating nodes/4g-vnf-test-nodes.yaml...
Schema validation error: nodes/4g-vnf-test-nodes.yaml
<traceback omitted>
Failed

Validating nodes/baseband-test-nodes.yaml...
Passed

Some validations failed

<clean up log omitted>
ERROR: Job failed: exit code 1
```

Note that tests such as *invalid node selection* and *conflicting policies* require multiple types of resources to be placed under the same repository in the current implementation. A potential enhancement could involve allowing the parameters to accept not only local paths but also remote repositories with paths. However, this would necessitate extra effort such as secret management of credentials, which is beyond the scope of the proof of concept workflow we are building.

It is also important to note that the test scripts are currently placed in the same repository as the resources, which is not ideal for production-level systems. A better approach is to package the validation scripts into proper Python packages that can be downloaded from registries such as PyPI [69]. This way, the scripts can be installed directly as dependencies in the pipeline, which improves maintainability and scalability.

Chapter 5

Analysis and Evaluation

In this chapter, we highlight the benefits of adopting a GitOps workflow in telecommunication security operations. We then consolidate the claimed benefits by comparing the traditional workflow and the proposed GitOps workflow across several scenarios. Finally, we identify limitations and challenges in the current implementation and the GitOps workflow in general.

5.1 GitOps Benefits in Security Operations

We evaluate and analyze the benefits of adopting a GitOps workflow in telecommunication security operations.

5.1.1 Streamlining Change Management Process

In the telecommunication industry, MNOs typically had to navigate a lengthy out-of-band ticket-based system to approve changes to security configurations. Before adopting the GitOps workflow, information relevant to the changes, such as content, purpose, responsible person, and timestamp, was stored externally and not well-integrated with NSCM. Moreover, approved changes required manual operations to be reflected in NSCM and the network, creating delays and increasing the risk of human errors.

With the GitOps workflow, changes are reviewed and approved using pull requests and automatically deployed by the GitOps operator. Information on commits and pull requests is stored centrally within the Git repositories and well-integrated with the NSCM system by the GitOps operators. This approach reduces the manual effort required to apply changes and minimizes the risk of human error.

While GitOps automates the technical aspects of configuration management, traditional ticket-based systems remain essential for broader organizational needs such as planning and knowledge sharing. By integrating GitOps with the existing systems, MNOs can achieve a comprehensive change management process that combines the efficiency and automation of GitOps with the planning and visibility provided by ticketing systems.

5.1.2 Better Visibility and Traceability

The GitOps workflow enhances visibility and traceability by tracking every change through the Git history. The history includes commit details such as the content of the change, the author, the timestamp, and the rationale behind it. Additionally, the pull requests have information such as the review comments and the identities of the reviewers. This record is useful for troubleshooting, auditing, and ensuring accountability within the system.

5.1.3 Reduced Time to Recovery

The GitOps workflow offers quick rollback to a previous stable state in case of incidents. With Git repositories storing the entire history of configurations, each commit acts as a complete snapshot of the desired state of the system. Users can quickly and easily roll back to a previously stable state using straightforward Git commands, such as `git revert`. Additionally, the automation inherent in the GitOps approach minimizes the manual steps required for rollback, further speeding up the recovery process.

5.1.4 Root Cause Analysis of Incidents

The commit history provides insights into the sequence of changes leading up to an incident. This facilitates root cause analysis and helps understand the impact of specific changes. The ability to view detailed commit messages, diffs, and pull request discussions ensures that users have the context they need to make informed decisions for incident response and recovery.

5.2 Case studies

In this section, we demonstrate the scenarios that illustrate the advantages of the proposed solution. We compare the workflow before and after the adoption of GitOps and examine the capabilities of the GitOps workflow within telecommunication security operations.

5.2.1 Updating Security Configurations

As discussed previously, updating security configurations in a telecommunication network typically involves lengthy change request processes, navigating multiple inadequately integrated systems, and numerous manual operations. While the NSCM has automated many trivial tasks, there is still room for improvement.

By implementing a GitOps workflow, the process is streamlined. Changes are reviewed within the same system where the desired state is stored and automatically deployed without human intervention. Figure 5.1 and Figure 5.2 illustrate the traditional and the GitOps workflows, respectively. In the proposed GitOps workflow, operations are automated by the GitOps operators, and the number of manual operations from the user is reduced from three to one, thereby minimizing delay and the potential for human error.

5.2.2 Deploying Multiple Experimental Environments

Prior to deploying NSCM into production, users typically create testing or staging environments to experiment with the system. The GitOps workflow is advantageous in this scenario because users no longer need to manually configure the network nodes, policies, and baselines whenever they create a new experimental environment. Instead, they only need to configure the GitOps operators to fetch the desired Git repositories and deploy them to the target environments. Figure 5.3 compares traditional and GitOps workflows for deploying multiple experimental environments, with the grey boxes indicating the newly automated steps in the GitOps workflow. This approach reduces the hours of manual work required to configure the resources in NSCM.

Furthermore, users can store different environments in the same Git repository using branches. With proper branching strategies, users can test different configurations in separate environments and cherry-pick commits or merge branches as needed. This approach facilitates experimentation and making it easier to manage multiple experimental setups. However, users should be cautious when cherry-picking commits and merging branches to avoid overlaps, conflicts, and invalid node selections.

5.2.3 Disaster Recovery

Before the introduction of the GitOps workflow, rolling back to a working version can be challenging. When unexpected incidents occur after changes to multiple control variables, manually identifying and reverting the changes could be difficult. Users might need to examine the change request system

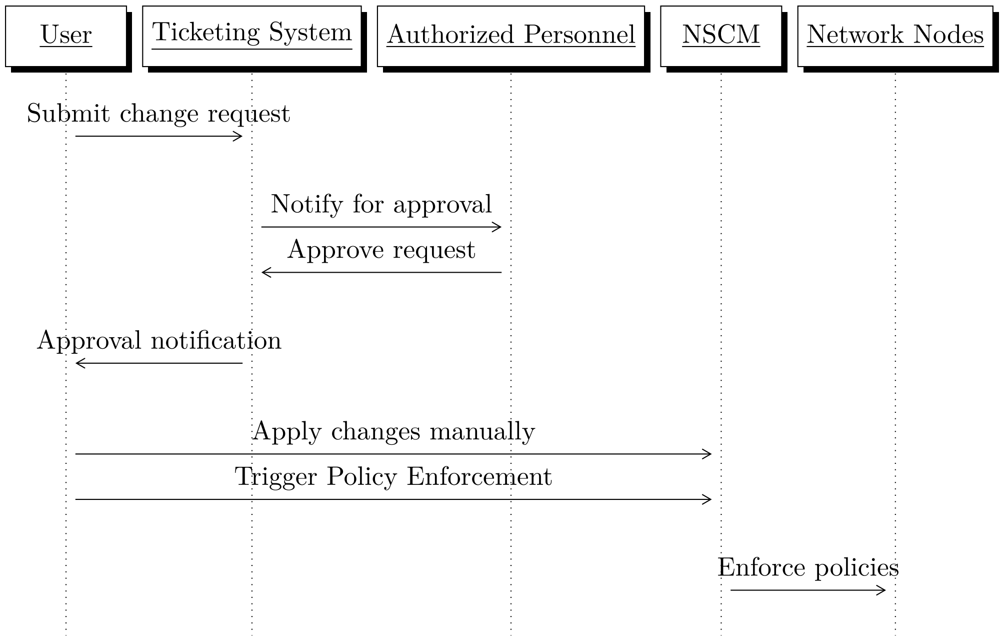

Figure 5.1: Traditional workflow for updating security configurations.

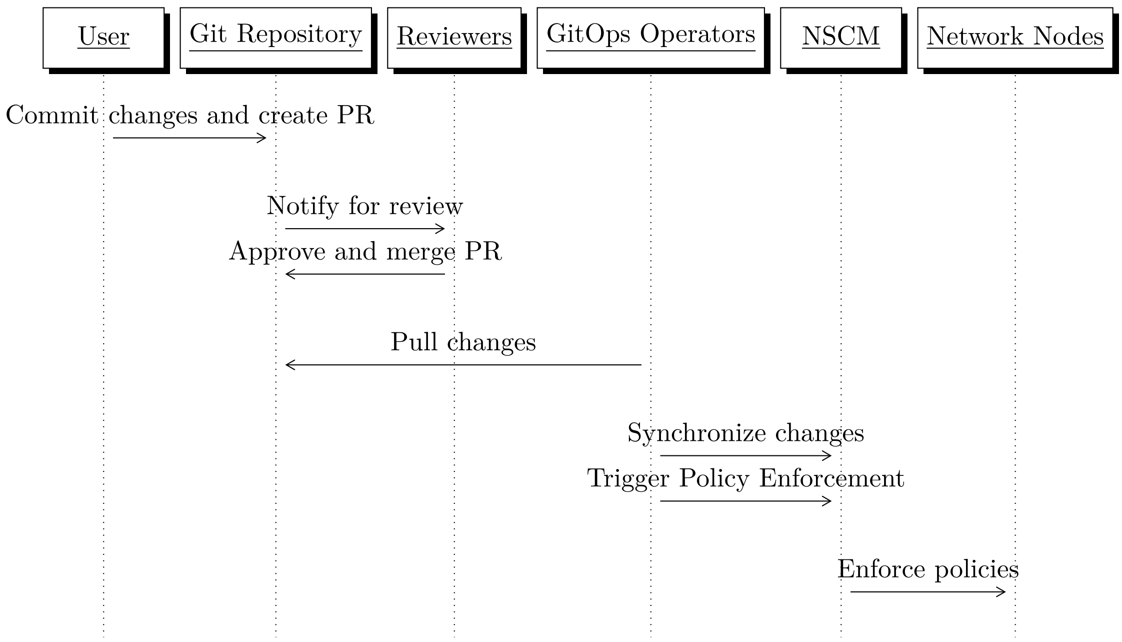

Figure 5.2: GitOps workflow for updating security configurations.

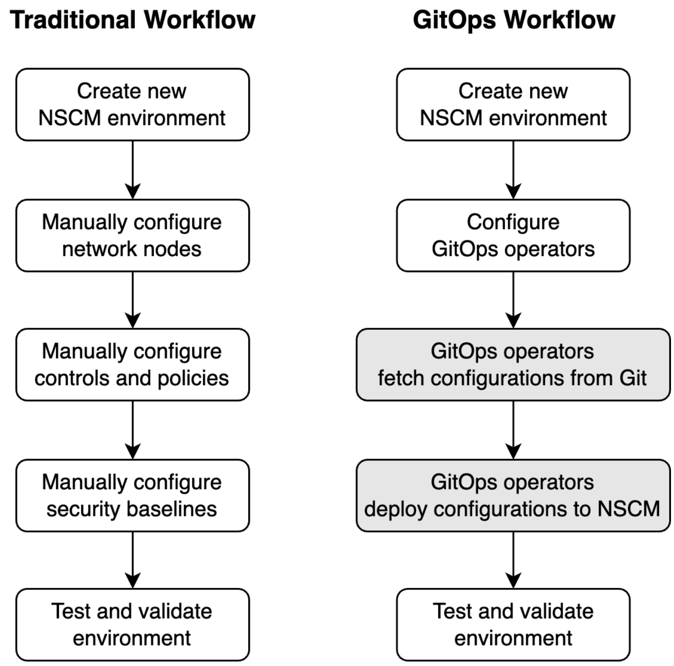

Figure 5.3: Comparison of traditional workflow and GitOps workflow for deploying experimental environments.

or analyze NSCM logs to determine when and what has been changed to cause the incident. With the GitOps workflow, users can easily identify the changes through pull requests and roll back to any previous point in time using commands such as `git revert`. Figure 5.4 and Figure 5.5 show the traditional and the GitOps workflows for disaster recovery, respectively. Note that the *Check Git history* step in Figure 5.5 is much more efficient than the *Check ticketing system* step in Figure 5.4 as it provides the context of the sequence of changes.

Moreover, integrating mechanisms that trigger rollbacks automatically in response to predefined failure conditions can be considered. Immediate action is taken and recorded in Git as soon as an issue is detected. Such proactive measures can be helpful in maintaining high availability and reliability of network services.

5.2.4 Configuration Drift

Without the GitOps workflow, configuration drifts caused by malicious actors or unintentional human errors cannot be detected and corrected in real time. Although users can set compliance checks to run periodically, they still

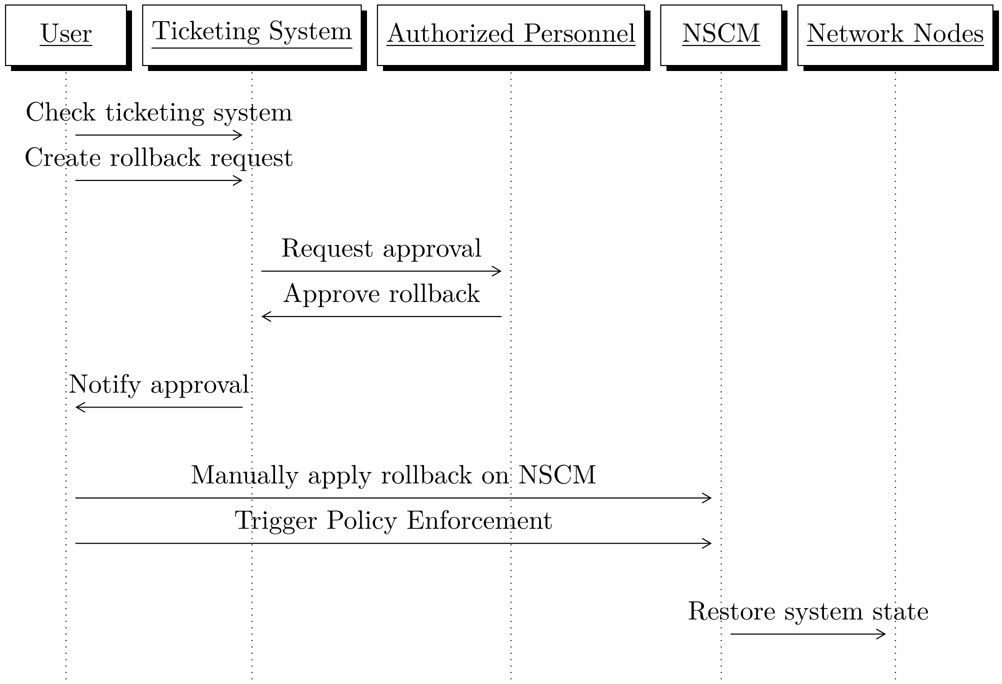

Figure 5.4: Traditional workflow for disaster recovery.

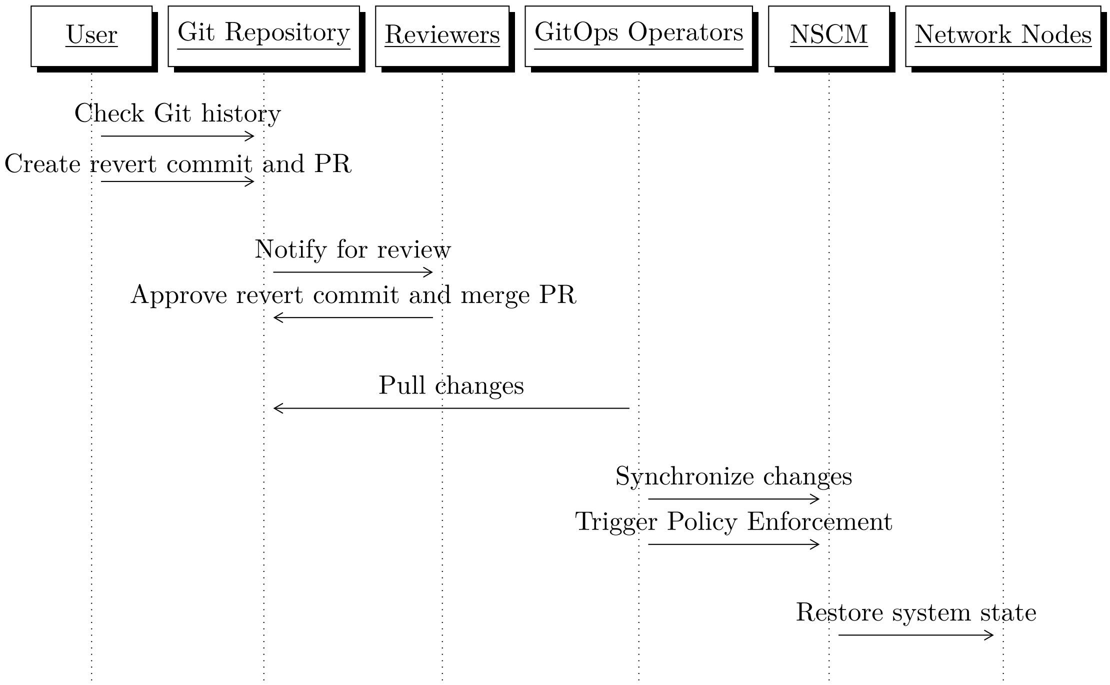

Figure 5.5: GitOps Workflow for disaster recovery.

require manual intervention to address configuration drifts identified in the compliance check results. With the GitOps workflow, security configurations are continuously and automatically monitored and managed, which reduces the risk of configuration drifts in the networks.

5.3 Limitations and Challenges

Although adopting a GitOps workflow provides numerous benefits, we also recognize limitations and challenges in the workflow. In this section, we list some limitations and challenges in our GitOps workflow.

5.3.1 GUI Compatibility

Our current solution completely relies on pure Git operations. However, many network administrators are used to using the NSCM GUI. In our proposed workflow, changes made from the GUI will be over-written by the GitOps operator to maintain the single source of truth stored in Git repositories.

The solution could be enhanced by integrating a Git client into the NSCM and abstracting Git operations from the users. For example, when users make changes in the NSCM GUI, the system automatically commits the changes and creates a pull request on behalf of the logged-in user. Moreover, NSCM could include a GUI to manage the review and approve processes, maintaining network administration processes similar to before while adopting the GitOps approach in the background.

5.3.2 Expressiveness of Selector

Currently, the node selectors in the security baseline only offer the *union* semantic. As a result, more complex selection logic, such as selecting *base stations labeled as 5G and not in the Helsinki region*, cannot be expressed in our current declarative model.

Introducing logical operators in the declarative model could provide a solution. By designing a new schema to support arbitrary logical expressions, users would gain the ability to specify more nuanced selection criteria. For example, combining `AND`, `OR`, and `NOT` operators could allow for complex configurations, such as selecting nodes that meet multiple intersecting or exclusive conditions.

Another possible solution is to express boolean logic with disjunctive normal form (DNF). We can achieve this with nested lists in the YAML files.

For instance, Listing 5.1 shows a selector that captures *base stations labeled as 5G and not in the Helsinki region plus all 4G VNFs.*

Listing 5.1: Example selector in disjunctive normal form (DNF).

```
selector:
  - ["label:5g", "!label:helsinki", "node_type:base_station"]
  - ["label:4g", "node_type:4g_vnf"]
```

However, this added expressiveness comes with trade-offs. Complex logical expressions can result in convoluted YAML files, which potentially lead to losing the clarity and simplicity of using a declarative model in the first place. To address this issue, it is essential to conduct deeper user research to understand the specific needs and use cases of network administrators.

5.3.3 Sequence of Operations in Security Changes

In security operations, the sequence of operations is crucial for maintaining the integrity and security of the system. When changing a policy in the network, the order of operations, such as adding-before-removing or removing-before-adding, can significantly impact network security.

For example, if we are updating firewall rules from a more permissive to a more restrictive one, the default rule determines the correct sequence to follow. If the default rule allows all traffic, we should add the new restrictive rule before removing the old permissive rule to ensure no unintended traffic is allowed. Conversely, if the default rule denies all traffic, we should remove the old rule before adding the new rule to avoid conflicts that might create unexpected behavior.

Our current implementation does not provide a way to specify the order of operations nor guarantee the intermediate state during transitioning. However, we can coordinate the two operators so that when the Network Node and Policy Management Operator is updating the resources in NSCM, the Policy Enforcement Operator waits until the changes are applied before enforcing the new policies. This delegates the responsibility of ensuring the correct sequence of operations to the NSCM. An alternative solution is to already ensure the correct sequence in the operators, but that requires understanding the semantics of the changes, which is challenging due to the complexity and dynamic nature of security controls.

Chapter 6

Conclusions

In this chapter, we summarize the findings and discuss how the result would be useful for NSCM users to improve their workflows. We then propose future areas that need further research and conclude the thesis.

6.1 Summary of Findings

In this thesis, we identify the challenges in traditional telecommunication security operations and propose a new workflow based on the GitOps principles to address these issues. We develop a declarative model to describe the resources in NSCM in YAML format, which allows the resources to be stored in Git for tracking and managing changes to them. Additionally, we implement GitOps operators where the reconciliation loops continuously ensure the actual state in NCSM and the network align with the desired state stored in the Git repository. To enhance the robustness of the solution, we also incorporate network nodes, security policies, and security baselines validation within the CI pipeline.

We evaluate the proposed workflow through case studies, analyzing aspects such as automation, visibility, traceability, disaster recovery, and configuration drift correction. While the workflow offers numerous benefits, we also pinpoint limitations in the current implementation. The study provides a comprehensive overview of the advantages and challenges associated with adopting GitOps in telecommunication security network management. It suggests that the telecommunication industry should consider adopting GitOps to improve its network security management processes.

6.2 Future Work and Improvements

In this section, we list potential future work and improvements that could be considered if the GitOps operators go into production.

6.2.1 Integration with NSCM

As mentioned in Section 4.2, the current implementation is a standalone Python program that is not shipped with the NSCM product itself. To provide a better user experience, the GitOps operator should not require a separate installation but should be available out-of-the-box with NSCM. This requires packaging and containerizing the GitOps operators into microservices included in the NSCM build process.

6.2.2 Scalability and Performance Optimization

Since this work is a proof-of-concept workflow that demonstrates the possibility of adopting GitOps workflow into telecommunication security operations, the system has yet to be deployed at scale. The operators are implemented straightforwardly and have yet to be optimized for performance when managing hundreds or thousands of resources simultaneously. Addressing scalability and performance will be crucial for deploying production-level GitOps operators.

6.2.3 Flexibility in Repository Structure

As discussed in Section 4.2.1, the current implementation requires users to place only deployable YAML files in specified folders. In real-world scenarios, users might prefer different folder structures based on their operational needs and existing practices. To improve usability and prevent accidental deployment of irrelevant files, the system could be enhanced to support customizable repository layouts by incorporating mechanisms to distinguish between deployable and non-deployable YAML files, possibly through schema validation or `.gitignore`-style files.

Another solution could be having a catalog file that lists the files to be included in the deployment instead of including all files in the specified folder by default. This way, multiple sets of configurations can exist as separate catalog files which reuse the same component files. While this approach can be flexible and powerful, it also introduces complexity when there are large numbers of configuration files. The optimal choice will depend on the specific use cases and requirements of the users.

6.2.4 Secret Management

Currently, the system does not address the secret management of login information for network nodes. Future improvements could involve integrating with existing secret management tools or implementing a custom secret management system to securely handle credentials and other sensitive information within the GitOps workflow.

6.2.5 Better Node Selector in Security Baseline

The current model supports only basic *union* semantics for node selection, limiting the ability to define complex selection logic. Enhancing the node selector capabilities to support more sophisticated logical expressions, such as combining `AND`, `OR`, and `NOT`, would provide greater flexibility in defining security baselines. This enhancement would require careful consideration to maintain the simplicity and readability of the declarative model.

6.2.6 Compatibility with NSCM GUI

While the current implementation relies heavily on Git operations, many network administrators are accustomed to using the NSCM GUI. Future work could integrate GitOps functionalities into the NSCM GUI, allowing administrators to manage configurations and review changes within a familiar interface. This could involve developing a GUI that handles commit operations, pull requests, and code reviews to ensure a smooth transition to a GitOps workflow without compromising user experience.

6.2.7 Gradual Deployment Strategies

In the current implementation, all approved changes to security configurations are immediately applied across the entire network. However, this approach carries inherent risks, such as the potential for widespread disruption if a newly implemented configuration contains errors. A solution to mitigate the risk could be incorporating a gradual deployment process, such as a canary deployment, into the workflow. This strategy reduces the risk of global failures and ensures issues can be identified and addressed before full-scale deployment.

Bibliography

[1] Florian Beetz and Simon Harrer. GitOps: The evolution of DevOps? *IEEE Software*, 39(4):70–75, 2021.

[2] Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, and Jesse Suen. *GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux*. Simon and Schuster, 2021.

[3] Ijaz Ahmad, Shahriar Shahabuddin, Tanesh Kumar, Jude Okwuibe, Andrei Gurtov, and Mika Ylianttila. Security for 5G and beyond. *IEEE Communications Surveys & Tutorials*, 21(4):3682–3722, 2019.

[4] Ericsson Security Manager. `https://www.ericsson.com/en/portfolio/new-businesses/security-and-risk-management/security-manager` (Accessed Mar. 28, 2024).

[5] NetGuard Cybersecurity Dome. `https://www.nokia.com/networks/security-portfolio/netguard/cybersecurity-dome/` (Accessed Mar. 28, 2024).

[6] A guide to 5G network security 2.0. Technical report, Ericsson, 2021. `https://www.ericsson.com/4a66f8/assets/local/security/09172021-a-guide-to-5g-network-security-2.0.pdf` (Accessed Mar. 22, 2024).

[7] University of Helsinki. Core 5G and beyond [MOOC], 2023. `https://6g.mooc.fi` (Accessed Mar. 22, 2024).

[8] Ramije Zeqiri, Florim Idrizi, and Halim Halimi. Comparison of algorithms and technologies 2G, 3G, 4G and 5G. In *2019 3rd International symposium on Multidisciplinary studies and Innovative Technologies (ISMSIT)*. IEEE, 2019.

[9] Ramraj Dangi, Praveen Lalwani, Gaurav Choudhary, Ilsun You, and Giovanni Pau. Study and investigation on 5G technology: A systematic review. *Sensors*, 22(1):26, 2021.

[10] Naser Al-Falahy and Omar Y Alani. Technologies for 5G networks: Challenges and opportunities. *IT Professional*, 19(1):12–20, 2017.

[11] Qiang Tang, Orhan Ermis, Cu D Nguyen, Alexandre De Oliveira, and Alain Hirtzig. A systematic analysis of 5G networks with a focus on 5G core security. *IEEE Access*, 10:18298–18319, 2022.

[12] GR NFV 003 - v1.8.1 - network functions virtualisation (NFV); terminology for main concepts in NFV. Technical report, ETSI, 2023. `https://www.etsi.org/deliver/etsi_gr/NFV/001_099/003/01.08.01_60/gr_NFV003v010801p.pdf` (Accessed Apr. 7 2024).

[13] Bo Yi, Xingwei Wang, Keqin Li, Min Huang, et al. A comprehensive survey of network function virtualization. *Computer Networks*, 133:212–262, 2018.

[14] Teodor B Iliev, Nikolay T Nikolaev, Grigor Y Mihaylov, Elena P Ivanova, Ivaylo S Stoyanov, and Dimitar I Radev. A critical view of the NFV dataplane implementations for 5G networks. In *2018 26th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE, 2018.

[15] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications surveys & tutorials*, 18(1):236–262, 2015.

[16] Sofiane Imadali and Ayoub Bousselmi. Cloud native 5G virtual network functions: Design principles and use cases. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 91–96. IEEE, 2018.

[17] Massimo Condoluci and Toktam Mahmoodi. Softwarization and virtualization in 5G mobile networks: Benefits, trends and challenges. *Computer Networks*, 146:65–84, 2018.

[18] Dirk Wubben, Peter Rost, Jens Steven Bartelt, Massinissa Lalam, Valentin Savin, Matteo Gorgoglione, Armin Dekorsy, and Gerhard Fettweis. Benefits and impact of cloud computing on 5G signal processing: Flexible centralization through cloud-RAN. *IEEE signal processing magazine*, 31(6):35–44, 2014.

[19] Wissal Attaoui, Essaid Sabir, Halima Elbiaze, and Mohsen Guizani. VNF and CNF placement in 5G: Recent advances and future trends. *IEEE Transactions on Network and Service Management*, 2023.

[20] Adrián Pino, Pouria Khodashenas, Xavier Hesselbach, Estefanía Coronado, and Shuaib Siddiqui. Validation and benchmarking of CNFs in OSM for pure cloud native applications in 5G and beyond. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2021.

[21] Ericsson cloud RAN. `https://www.ericsson.com/en/ran/cloud` (Accessed Mar. 27, 2024).

[22] Nokia cloud RAN. `https://www.nokia.com/networks/mobile-networks/airscale-radio-access/cloud-ran/` (Accessed Mar. 27, 2024).

[23] Nephio. `https://nephio.org/` (Accessed Mar. 27, 2024).

[24] Hans Christian Rudolph, Andreas Kunz, Luigi Lo Iacono, and Hoai Viet Nguyen. Security challenges of the 3GPP 5G service based architecture. *IEEE Communications Standards Magazine*, 3(1):60–65, 2019.

[25] Yu-Lun Huang, Borting Chen, Ming-Wei Shih, and Chien-Yu Lai. Security impacts of virtualization on a network testbed. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 71–77. IEEE, 2012.

[26] Sihyung Lee, Kyriaki Levanti, and Hyong S Kim. Network monitoring: Present and future. *Computer Networks*, 65:84–98, 2014.

[27] Madanagopal Ramachandran, T Archana, V Deepika, A Arjun Kumar, and Krishna M Sivalingam. 5G network management system with machine learning based analytics. *IEEE Access*, 10:73610–73622, 2022.

[28] Jae Seung Lee, Jaewoo Park, Jeeyon Choi, and Moon-Sik Lee. Design of a management plane for 5G open fronthaul interface. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 946–948. IEEE, 2020.

[29] Tse-Han Wang, Yen-Cheng Chen, Sin-Jie Huang, Kai-Sheng Hsu, and Chung-Hua Hu. Design of a network management system for 5G open RAN. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 138–141. IEEE, 2021.

[30] Danish Rafique and Luis Velasco. Machine learning for network automation: overview, architecture, and applications [invited tutorial]. *Journal of Optical Communications and Networking*, 10(10):D126–D143, 2018.

[31] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. Automation for network security configuration: State of the art and research trends. *ACM Computing Surveys*, 56(3):1–37, 2023.

[32] Jonathan Olsson, Andrey Shorov, Loay Abdelrazek, and Jorden Whitefield. 5G zero trust–a zero-trust architecture for telecom. *Ericsson Technology Review*, 2021(5):2–11, 2021.

[33] Hans Byström, Hsin-Yi Chen, and Scott Poretsky. Zero trust architecture for advancing mobile network security operations. Technical report, Ericsson, 2024. `https://www.ericsson.com/49313a/assets/local/reports-papers/white-papers/2024/zero-trust-architecture.pdf` (Accessed Mar. 27, 2024).

[34] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. Dimensions of DevOps. In *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings 16*, pages 212–217. Springer, 2015.

[35] Håvard Myrbakken and Ricardo Colomo-Palacios. DevSecOps: a multivocal literature review. In *Software Process Improvement and Capability Determination: 17th International Conference, SPICE 2017, Palma de Mallorca, Spain, October 4–5, 2017, Proceedings*, pages 17–29. Springer, 2017.

[36] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[37] Jennifer Davis and Ryn Daniels. *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale.* O'Reilly Media, Inc., 2016.

[38] Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120. IEEE, 2017.

[39] Jenkins. `https://www.jenkins.io/` (Accessed Mar. 27, 2024).

[40] GitLab CI/CD. `https://docs.gitlab.com/ee/ci/` (Accessed Mar. 27, 2024).

[41] Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren. *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution, 2021.

[42] Kief Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[43] Terraform. `https://www.terraform.io/` (Accessed Mar. 27, 2024).

[44] Ansible. `https://www.ansible.com/` (Accessed Mar. 27, 2024).

[45] Puppet. `https://www.puppet.com/` (Accessed Mar. 27, 2024).

[46] Chef. `https://www.chef.io/` (Accessed Mar. 27, 2024).

[47] Mali Senapathi, Jim Buchan, and Hady Osman. DevOps capabilities, practices, and challenges: Insights from a case study. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 57–67, 2018.

[48] Leah Riungu-Kalliosaari, Simo Mäkinen, Lucy Ellen Lwakatare, Juha Tiihonen, and Tomi Männistö. DevOps adoption benefits and challenges in practice: A case study. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, pages 590–597. Springer, 2016.

[49] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Security as culture: a systematic literature review of DevSecOps. In *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pages 266–269, 2020.

[50] Roshan N Rajapakse, Mansooreh Zahedi, M Ali Babar, and Haifeng Shen. Challenges and solutions when adopting DevSecOps: A systematic review. *Information and software technology*, 141:106700, 2022.

[51] OpenGitOps. `https://opengitops.dev/` (Accessed Mar. 19, 2024).

[52] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I 37*, pages 570–596. Springer, 2017.

[53] Peter Wörndle, Stephen Terrill, and Torsten Dinsing. Automating telecom software deployment with GitOps: A declarative, GitOps-based approach to software delivery and deployment enables communication service providers to increase automation and speed up the introduction of new features and updates-including security fixes-into their networks. *Ericsson Technology Review*, 2023(2):2–10, 2023.

[54] Domenico Scotece, Asma Noor, Luca Foschini, and Antonio Corradi. 5G-kube: Complex telco core infrastructure deployment made low-cost. *IEEE Communications Magazine*, 2023.

[55] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Pietro Brach del Prever, and Tommaso Melodia. 5G-CT: Automated deployment and over-the-air testing of end-to-end open radio access networks. *arXiv preprint arXiv:2311.03206*, 2023.

[56] Ákos Leiter, Attila Hegyi, István Kispál, Pál Böõsy, Nándor Galambosi, and Gábor Zsolt Tar. Gitops and kubernetes operator-based network function configuration. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. IEEE, 2023.

[57] Ákos Leiter, István Kispál, Attila Hegyi, Péter Fazekas, Nándor Galambosi, Péter Hegyi, Péter Kulics, and József Bíró. Intent-based 5G UPF configuration via Kubernetes operators in the edge. In *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 186–189. IEEE, 2022.

[58] Christian Posta. Configuration as data, GitOps, and controllers: It's not simple for multi-cluster, 2021. `https://www.solo.io/blog/configuration-as-data-gitops-and-controllers-its-not-simple-for-multi-cluster/` (Accessed Apr. 11, 2024).

[59] Flux CD. `https://fluxcd.io/` (Accessed Apr. 11, 2024).

[60] Argo CD - declarative GitOps CD for Kubernetes. `https://argo-cd.readthedocs.io/en/stable/` (Accessed Apr. 11, 2024).

[61] Operator pattern | Kubernetes. `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (Accessed May 13, 2024).

[62] JSON schema. `https://json-schema.org/` (Accessed Jun. 7, 2024).

[63] GitHub. `https://github.com/` (Accessed May 14, 2024).

[64] GitLab. https://about.gitlab.com/ (Accessed May 14, 2024).

[65] Gerrit code review. https://www.gerritcodereview.com/ (Accessed May 14, 2024).

[66] Gitea. https://about.gitea.com/ (Accessed May 14, 2024).

[67] PyYAML. https://github.com/yaml/pyyaml (Accessed Jun. 7, 2024).

[68] jsonschema 4.22.0 documentation. https://python-jsonschema.readthedocs.io/en/stable/ (Accessed Jun. 7, 2024).

[69] PyPI · the Python package index. https://pypi.org/ (Accessed Jun. 10, 2024).

Appendix A

NSCM Resource Schemas

Listing A.1: Schema of the network nodes.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "description": "Schema for network nodes configuration",
  "properties": {
    "network_nodes": {
      "type": "array",
      "description": "List of network nodes",
      "items": {
        "type": "object",
        "description": "An network node entry",
        "properties": {
          "name": {
            "type": "string",
            "description": "Name of the network node"
          },
          "description": {
            "type": "string",
            "description": "Description of the network node"
          },
          "node_type": {
            "type": "object",
            "description": "Type of the network node",
            "properties": {
              "name": {
```

```
            "type": "string",
            "description": "Name of the node type"
          },
          "version": {
            "type": "string",
            "description": "Version of the node type"
          }
        },
        "required": [
          "name",
          "version"
        ]
      },
      "interfaces": {
        "type": "array",
        "description": "List of OAM interfaces for the network node",
        "items": {
          "type": "object",
          "description": "An interface entry",
          "properties": {
            "name": {
              "type": "string",
              "description": "Name of the interface"
            },
            "description": {
              "type": "string",
              "description": "Description of the interface"
            },
            "interface_type": {
              "type": "object",
              "description": "Type of the interface",
              "properties": {
                "name": {
                  "type": "string",
                  "description": "Name of the interface type"
                }
              },
              "required": [
```

```
                    "name"
                  ]
                },
                "ip_address": {
                  "type": "string",
                  "description": "IP address of the interface",
                  "format": "ipv4"
                },
                "port": {
                  "type": "integer",
                  "description": "Port number of the interface",
                  "minimum": 1,
                  "maximum": 65535
                }
              },
              "required": [
                "name",
                "interface_type",
                "ip_address",
                "port"
              ]
            }
          }
        },
        "required": [
          "name",
          "node_type",
          "interfaces"
        ]
      }
    }
  },
  "required": [
    "network_nodes"
  ]
}
```

Listing A.2: Schema of the security policies.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
```

```
  "type": "object",
  "description": "Schema for security policies and controls configuration",
  "properties": {
    "security_policies": {
      "type": "array",
      "description": "List of security policies",
      "items": {
        "type": "object",
        "description": "A security policy entry",
        "properties": {
          "name": {
            "type": "string",
            "description": "Name of the security policy"
          },
          "controls": {
            "type": "array",
            "description": "List of security controls within the policy",
            "items": {
              "type": "object",
              "description": "A security control and its associated variables",
              "properties": {
                "control": {
                  "type": "string",
                  "description": "Name of the security control"
                },
                "variables": {
                  "type": "array",
                  "description": "List of variables associated with the control. Each variable is a key-value pair.",
                  "items": {
                    "type": "object",
                    "description": "A key-value pair representing a variable for the control. The value must be a primitive type.",
                    "additionalProperties": {
                      "type": [
                        "string",
```

```
                        "number",
                        "boolean",
                        "null"
                      ],
                      "description": "The value of the variable. Must be a primitive type."
                    }
                  }
                }
              },
              "required": [
                "control"
              ]
            }
          }
        },
        "required": [
          "name",
          "controls"
        ]
      }
    }
  },
  "required": [
    "security_policies"
  ]
}
```

Listing A.3: Schema of the security baselines.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "description": "Schema for security baselines configuration",
  "properties": {
    "security_baselines": {
      "type": "array",
      "description": "List of security baselines",
      "items": {
        "type": "object",
        "description": "A security baseline entry",
```

```
        "properties": {
          "name": {
            "type": "string",
            "description": "Name of the security baseline"
          },
          "bindings": {
            "type": "object",
            "description": "Bindings for the security baseline",
            "properties": {
              "node_type": {
                "type": "array",
                "description": "List of node types",
                "items": {
                  "type": "string"
                }
              },
              "network_nodes": {
                "type": "array",
                "description": "List of network nodes by their names. Regex patterns are supported.",
                "items": {
                  "type": "string"
                }
              },
              "labels": {
                "type": "array",
                "description": "List of labels",
                "items": {
                  "type": "string"
                }
              }
            },
            "anyOf": [
              {
                "required": [
                  "node_type"
                ]
              },
              {
                "required": [
```

```
                  "network_nodes"
                ]
              },
              {
                "required": [
                  "labels"
                ]
              }
            ]
          },
          "security_policies": {
            "type": "array",
            "description": "List of security policies",
            "items": {
              "type": "object",
              "properties": {
                "policy": {
                  "type": "string",
                  "description": "Name of the policy"
                }
              },
              "required": [
                "policy"
              ]
            }
          },
          "controls": {
            "type": "array",
            "description": "List of controls",
            "items": {
              "type": "object",
              "properties": {
                "control": {
                  "type": "string",
                  "description": "Name of the control"
                },
                "variables": {
                  "type": "array",
                  "description": "List of variables for the control",
                  "items": {
                    "type": "object",
                    "additionalProperties": {
```

```
                      "type": [
                        "string",
                        "number",
                        "boolean",
                        "null"
                      ]
                    }
                  }
                }
              },
              "required": [
                "control"
              ]
            }
          }
        },
        "required": [
          "name",
          "bindings"
        ],
        "anyOf": [
          {
            "required": [
              "security_policies"
            ]
          },
          {
            "required": [
              "controls"
            ]
          }
        ]
      }
    }
  },
  "required": [
    "security_baselines"
  ]
}
```