Master's thesis

Master's Programme in Computer Science

Automated Configuration Validation for Cloud-Native Network functions

Kaif Shahahusen Jamadar

May 22, 2025

FACULTY OF SCIENCE

UNIVERSITY OF HELSINKI

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Kaif Shahahusen Jamadar

Työn nimi — Arbetets titel — Title

Automated Configuration Validation for Cloud-Native Network Functions

Ohjaajat — Handledare — Supervisors

Prof. Valtteri Niemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 22, 2025	42 pages, 1 appendix pages

Tiivistelmä — Referat — Abstract

As cloud-native network functions (CNFs) become integral to modern telecommunications, ensuring the security and reliability of their configurations is critical for maintaining operational integrity and mitigating vulnerabilities. However, the complexity of managing CNF configurations, driven by their dynamic nature and evolving compliance requirements, presents significant challenges. The thesis introduces a static configuration validator tool designed for cloud-native network functions. This tool integrates seamlessly into Continuous Integration Continuous Deployment (CI/CD) pipelines such as GitLab to automate the verification of configuration files against established security and operational policies.

The results of the validator tool are fed back into the CI/CD pipeline, offering immediate feedback to developers and operators. This feedback loop ensures that misconfigurations are identified and rectified early in the development lifecycle, aligning with shift-left security principles and preventing potential operational disruptions and security risks.

In conclusion, this thesis presents a static configuration validator that integrates with CI/CD pipelines to support the early detection of misconfigurations in CNF environments. The validator operates during development stages to verify configuration files against defined policies, helping ensure adherence to current security and operational standards. Its integration into automated workflows enables consistent checks, supporting a more systematic approach to configuration management. This method addresses challenges related to configuration complexity and evolving compliance requirements in CNF deployments, with the goal of improving reliability and reducing post-deployment issues.

Avainsanat — Nyckelord — Keywords

Cloud-Native Network functions, Configuration, Network functions, Telecommunication networks, 5G

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networks study track

Contents

1 Introduction

The introduction of 5G networks has brought significant advancements in mobile communications, including enhanced mobile broadband (eMBB), ultra-reliable low-latency communications (URLLC), and massive machine-type communications (mMTC). One of the key changes is the shift from hardware-centric infrastructures to more flexible models based on Network Function Virtualization (NFV) and Cloud-Native Network Functions (CNFs) [1, 2]. NFV enables the separation of network functions from dedicated hardware, while CNFs build on this by using containerized, microservice-based architectures. These technologies offer improved scalability, modularity, and deployment flexibility [1, 2]. As part of this transition, correct and consistent configuration of CNFs is required to maintain interoperability and efficient use of resources [4].

Cloud-native approaches have changed how CNFs are deployed and managed, using tools like Docker and Kubernetes to automate and orchestrate services [1, 2]. However, the shift to these dynamic and distributed systems introduces challenges in configuration management. CNFs are composed of interconnected components that must be configured correctly to function reliably and securely. Minor misconfigurations can affect service availability, introduce security risks, or cause compliance issues [1, 4]. Tools like Ericsson Security Manager (ESM) [9] may detect these misconfigurations during post-deployment checks, requiring costly corrections and potentially affecting the credibility of the deployment. Identifying and resolving these issues later in the lifecycle increases workload and may delay release timelines. Therefore, managing configurations carefully from the early stages is important for reducing risk and operational overhead.

Default configurations applied during the initial deployment phase are particularly important. These defaults must meet current security and compliance requirements to avoid potential vulnerabilities in production environments. Errors introduced at this stage are more difficult to identify later and may affect system behavior when CNFs are scaled or modified. Maintaining configuration consistency across environments becomes more complex as deployments evolve. As security standards change, previously accepted configurations may become outdated, requiring updates to stay aligned with best practices. Keeping configurations current over time is an ongoing task that benefits from automated approaches.

Automated configuration validation tools support this need by checking configuration files against defined security and operational policies. They help identify issues before deployment and support efforts to keep CNFs aligned with updated standards. These tools can assist developers who may

not be experts in security, ensuring configurations meet requirements without manual tracking of all policy changes.

This thesis focuses on CNF configuration validation during development. It presents a containerized, platform-independent validator that integrates with CI/CD pipelines and checks configuration files against version-controlled policy sets. By using practices such as declarative configuration and version control concepts commonly found in GitOps models [18] the validator provides consistent evaluation of configurations before deployment. Full GitOps workflows such as continuous reconciliation are not applied in this work.

The aim of this thesis is to support early detection of configuration issues, allowing teams to address them during development. This helps reduce the likelihood of errors persisting into production environments and contributes to the reliability and maintainability of CNFs in cloud-native network systems.

2 Background

This chapter introduces the technological foundations necessary to understand the configuration of Cloud-Native Network Functions (CNFs) in modern telecommunication networks. It begins by outlining the historical evolution of mobile networks, leading to the adoption of software-based and cloud-native approaches in 5G. The chapter then explains key concepts such as Network Function Virtualization (NFV), Service-Based Architecture (SBA), and CNFs, focusing on how these developments have changed the way network functions are designed and deployed. It also discusses the role of container technologies like Docker and orchestration platforms like Kubernetes, which are essential for running CNFs. Finally, the chapter highlights the increasing complexity of CNF configuration and the need for structured management practices.

2.1 Evolution of telecommunication networks

The evolution of telecommunication networks over the past decades represents one of the most significant technological transformations in modern history. From the initial stages of analog voice transmission to the current sophisticated, all-IP, cloud-native networks, each mobile communication generation has built upon the accomplishments and insights of its predecessors.

The advent of the first generation (1G) of mobile networks marked the beginning of wireless communication, introducing analog voice services through Frequency Division Multiple Access (FDMA) [8]. Despite its groundbreaking impact, 1G networks were plagued by substantial limitations such as poor security, limited capacity, and unreliable handoffs. Early implementations, including the Advanced Mobile Phone System (AMPS) and Total Access Communication System (TACS), operated around 150 MHz with data speeds reaching up to 2.4 kbps [7, 8]. The inception of mobile networks laid the foundational groundwork for future advancements.

The second generation (2G) introduced technologies like the Global System for Mobile Communications (GSM) and Code Division Multiple Access (CDMA). These technologies not only provided clearer voice calls but also introduced SMS (Short Message Service) and MMS (Multimedia Message Service), enhancing network capacity through Time Division Multiple Access (TDMA) [7, 8]. The integration of General Packet Radio Service (GPRS) and Enhanced Data rates for GSM Evolution (EDGE) marked the advent of packet-switched data services, enabling basic internet access [7].

The development of third-generation (3G) networks came with the establishment of the Third Generation Partnership Project (3GPP) in 1998. 3GPP aimed to standardize protocols for mobile systems globally and to foster the development of a unified 3G system based on GSM core networks [7]. This collaboration led to the creation of the Universal Mobile Telecommunications System (UMTS), which employed the Wideband CDMA (WCDMA) air interface [8]. Launched around 2000, third-generation networks supported multimedia services, higher data rates up to 2 Mbps, global roaming, and laid the groundwork for internet-based applications [7, 8]. The coordinated efforts by 3GPP ensured interoperability across regions and vendors, facilitating a smooth transition of 3G into a global standard.

The development of fourth-generation (4G) networks, particularly through Long Term Evolution (LTE), marked a full embrace of an all-IP (Internet Protocol) architecture [8]. Standardized by 3GPP, LTE moved away from circuit-switched systems to packet-switched operations for both voice (via Voice over LTE or VoLTE) and data, achieving speeds up to 1 Gbps under optimal conditions [7, 8]. The introduction of the Evolved Packet Core (EPC) included modular network elements like the Serving Gateway (SGW) and Packet Data Network Gateway (PGW), enhancing flexibility, scalability, and efficiency. Notably, the evolution towards 4G and LTE began the shift towards Network Function Virtualization (NFV), where traditional hardware appliances started to be replaced by software running on commercial servers, marking an early stage of network cloudification [7].

With the progression towards 5G, 3GPP introduced the concept of Service-Based Architecture (SBA), a pivotal architectural milestone first defined in 3GPP Release 15. SBA restructured network functions as independent, discoverable, service-driven microservices communicating over standardized APIs [7]. This architectural shift aligned 5G with modern cloud-native principles such as scalability, elasticity, and flexibility, underpinning innovations like network slicing and dynamic resource allocation [7]. SBA transitioned from tightly coupled network elements to a distributed, service-oriented model, facilitating greater automation and swift service deployment essential for the expansive scale of IoT (Internet of things) and ultra-low-latency applications anticipated in 5G networks. The fully cloud-native core (5GC) represents the shift from physical network appliances to virtualized and cloud-based network functions, a transition that has been carefully developed and standardized by 3GPP.

A fundamental aspect of this evolution is the transformation of network functions themselves. Traditional hardware-dependent network functions—comprising routers, switches, and firewalls as physical appliances—have evolved into virtualized network functions (VNFs) and, more recently,

cloud-native network functions (CNFs). These are deployed as microservices orchestrated by container platforms like Kubernetes, offering enhanced agility, scalability, and cost efficiency [7].

Table 2.1 Evolution of telecommunication network [7, 8]

Generation	Type of Network Function	Architecture	Switching Method	Key Technical Changes
1G	Hardware-based, analog	Circuit-switched, no mobility management	Circuit Switching	- Analog FM signals - No encryption or data services - Voice-only
2G	Hardware-based, digital	Circuit-switched with basic packet overlays (2.5G)	Circuit Switching + limited Packet Switching (GPRS)	- Digital voice (GSM/CDMA) - Introduction of SIM cards - Support for SMS and low-speed data
3G	Hardware-centric, but modular	Circuit and Packet co-existing (CS+PS domains)	Hybrid: Circuit for voice, Packet for data	- Higher data rates (2 Mbps peak) - IP for data - Multimedia services (video call) - Global roaming improvements
4G	Virtualization(NFV)	All-IP, Flat EPC (Evolved Packet Core)	Full Packet Switching (VoIP)	- End of traditional circuit switching - All services over IP - Start of Network Function Virtualization (NFV) - Flat Architecture
5G	Cloud Native(CNFs)	Service-Based Architecture (SBA)	Full Packet Switching with Service Interfaces	- Microservices for Network Functions - APIs between functions (HTTP/2)

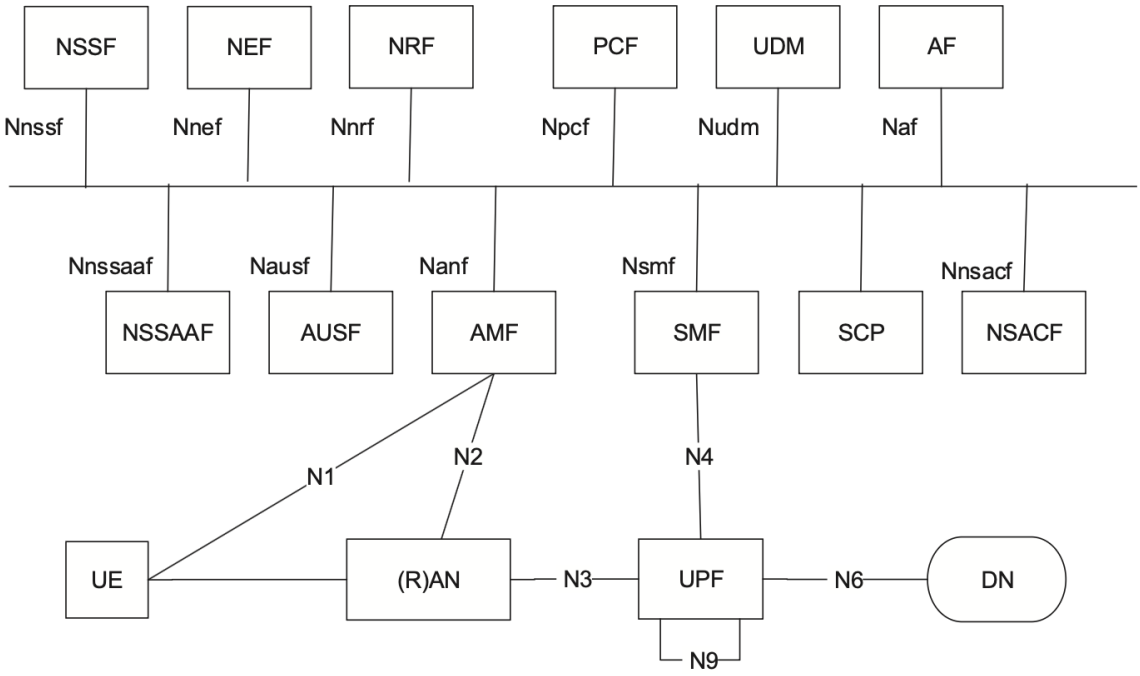
				- Massive scalability (IoT, slicing) - Containerization (Kubernetes, Docker) - AI and automation integration

2.2 Service Based Architecture (SBA)

In a Service-Based Architecture (SBA), 5G Core components operate as loosely coupled services that communicate using standard APIs over HTTP/2, as defined in 3GPP TS 23.501 [19]. While this enables scalability and flexibility, it also introduces increased configuration complexity. Each network function (e.g., AMF, SMF, UPF) must be deployed as a distinct service with its own service registration, discovery, and security settings—such as IP binding, ports, and TLS credentials [19]. This shift results in a larger number of independent configuration files, each with unique dependencies and interfaces that must be securely and consistently defined.

The distributed architecture increases the potential for misconfigurations due to the number of service-to-service communications that must be secured and validated. Unlike previous architectures, SBA requires that API contracts, access control rules, and service discovery parameters be explicitly configured across multiple components. This makes configuration management more error-prone and increases the importance of automated validation tools that can enforce policy compliance across services.

Figure 2.1 Service Based Architecture [19]



In SBA, Network Functions (NFs) interact dynamically through service-based communication interfaces, typically using RESTful APIs over HTTP/2 and exchanging data in JSON format. Each NF acts as either a producer or consumer of services, enabling flexible, scalable interactions suited to modern mobile and IoT use cases [19].

This architecture promotes interoperability and agility by decoupling components, allowing independent development and deployment of functions such as the Access and Mobility Management Function (AMF), Session Management Function (SMF), and User Plane Function (UPF). These functions register and discover services through the Network Repository Function (NRF), which maintains a catalogue of available services. When an NF requires a service, it queries the NRF to dynamically locate a suitable provider [19]. This mechanism supports dynamic scaling, failure recovery, and vendor diversity, but it also significantly increases configuration complexity [17].

Unlike monolithic or tightly coupled systems in previous generations, SBA-based networks rely on a high number of independently deployed CNFs, each with its own configuration requirements. These include requirements about service registration parameters, IP bindings, ports, authentication settings, and TLS configurations. The configuration for each NF must align with standardized policies to ensure proper service discovery, secure communication, and system interoperability. Misconfigurations, such as incorrect API endpoints or outdated certificates, can prevent service registration or cause runtime failures [17].

In addition to core functions, SBA includes components like the Network Slice Selection Function (NSSF), Policy Control Function (PCF), Network Exposure Function (NEF), and Service Communication Proxy (SCP), each with distinct configuration dependencies. These components enable advanced features like dynamic slicing, traffic policy enforcement, and external API access, but they also add layers of configuration to be validated and maintained [19].

The transition to the modular and distributed architecture aligns with broader trends in cloud-native systems and supports continuous integration and deployment practices. However, the new architecture also introduces configuration management challenges that cannot be reliably addressed through manual inspection alone. Configuration for each CNF must not only satisfy internal dependencies but also comply with global operational and security policies defined at the network level [17]. This increasing complexity highlights the need for tools that can automatically validate CNF configurations against predefined, version-controlled policies, ensuring consistency and reducing risks in highly dynamic SBA-based environments.

2.3 Network Function Virtualization (NFV)

Network Function Virtualization (NFV) refers to the architectural approach of decoupling network functions from proprietary hardware and deploying them as software instances on commercial off-the-shelf (COTS) infrastructure. Initially introduced in 4G-era networks, NFV marked a significant departure from tightly coupled hardware appliances by enabling network functions to operate as virtual machines (VMs) on general-purpose servers [1, 2].

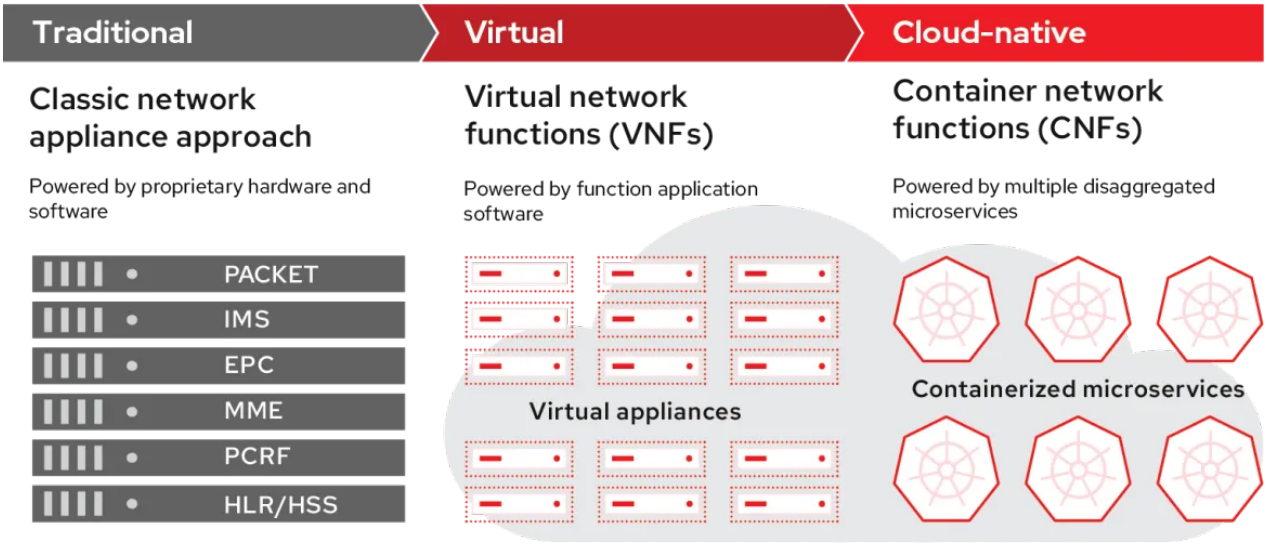
In the context of early 5G deployments and transitional networks, NFV continues to play a role by supporting virtualized network functions (VNFs) within core and edge data centers. This provides flexibility in how resources are allocated and allows operators to scale services more efficiently than in hardware-bound systems. NFV remains a relevant foundation in environments where full cloud-native transitions are either ongoing or not yet feasible.

However, as 5G networks evolve, the industry is increasingly adopting Cloud-Native Network Functions (CNFs), which represent a further shift from NFV-based VNFs. Unlike VNFs, CNFs are designed to run in containerized environments and are orchestrated by platforms such as Kubernetes. The transition builds upon NFV's virtualization concepts but emphasizes microservices, lightweight packaging, and automated orchestration for greater scalability and resilience. Thus, NFV serves as a foundational step in the broader evolution of network architecture.

2.4 Cloud Native Network Functions (CNFs)

Cloud-Native Network Functions (CNFs) represent a key step in the evolution of network function deployment. They build on the concepts introduced by Network Function Virtualization (NFV), which separated network functions from proprietary hardware by using virtual machines. CNFs extend this approach by using a microservice-based, containerized architecture that is designed for operation in dynamic and distributed cloud environments [2, 19]. In this model, individual network functions are broken down into smaller components that can be deployed, managed, and scaled independently.

Figure 2.1, Evolution of network functions [14]



To support deployment and operations, CNFs are typically orchestrated using platforms such as Kubernetes. Kubernetes provides core functionalities such as container scheduling, automated recovery, scaling, and service discovery. This allows operators to adjust resources in response to changing conditions without relying on manual intervention. CNFs also benefit from lightweight packaging and faster startup times compared to traditional virtual machines, improving portability and responsiveness across public, private, or hybrid cloud environments [19].

Alongside container orchestration, cloud-native systems often incorporate Continuous Integration and Continuous Deployment (CI/CD) practices. These allow development teams to apply updates more frequently and introduce changes incrementally. While these changes bring operational advantages, they also increase the rate at which configurations must be managed and validated. The challenge is not only in defining the right configuration at once but in maintaining its correctness over time and across different environments [17].

CNFs require configuration for various parameters such as service registration, authentication, API endpoints, and network ports. In large-scale deployments, each CNF may have environment-specific configurations, which must remain consistent to ensure correct operation. Configuration errors introduced at this stage, whether due to oversight, duplication, or evolving policy requirements can result in issues during deployment, including degraded service performance or unintended behavior [17, 19].

Because CNFs are deployed in distributed environments and scale dynamically, configuration errors may not be detected immediately. In some cases, incorrect settings may propagate silently across instances or clusters, leading to increased operational risk. This is especially relevant when configuration files are reused or manually adapted without systematic verification. The evolving nature of security policies and operational requirements further complicates configuration management. Settings that were valid during development may no longer meet compliance or security standards at a later time [19].

To manage this complexity, a more structured approach to configuration is needed. This includes not only defining configuration parameters correctly right from the beginning but also ensuring that these settings remain valid as systems evolve. The goal is to reduce the potential for post-deployment issues that are costly and time-consuming to resolve. A misconfigured default setting, for example, can be difficult to trace once deployed in a distributed system and may introduce behavior that is inconsistent with policy or intended operation.

In the context of CNFs, effective configuration practices play a central role in supporting system reliability and maintainability. Ensuring consistency, correctness, and compliance across all configuration files is essential, particularly in environments where services are composed of many loosely coupled components. As network functions become more modular and cloud-native, maintaining configuration accuracy becomes more challenging but also more critical.

By recognizing these challenges early in the lifecycle, developers and operators can reduce risk, improve operational outcomes, and maintain alignment with evolving standards. This provides necessary context for exploring tools and methods that help validate configuration files as part of the CNF development and deployment process.

2.5 Docker and Kubernetes

Docker is an open-source platform used to package software into standardized units called containers. These containers encapsulate the application code along with its runtime environment and dependencies, providing consistency across different execution environments [10]. Containers

are significantly more lightweight than virtual machines, as they share the host operating system kernel and start up much faster, making them well-suited for highly dynamic systems such as those seen in modern telecommunications networks. In the context of Cloud-Native Network Functions (CNFs), Docker provides a practical mechanism for encapsulating network functionality into modular, reproducible units. This enables CNFs to be instantiated quickly, replicated across distributed nodes, and integrated more easily into automated workflows [5].

Docker also facilitates the management of configuration files and runtime parameters in a controlled manner, which is particularly relevant when deploying CNFs in edge and core network environments. Open-source efforts such as the OpenShift Example CNF project [20] demonstrate containerized networking applications running in standardized Docker environments. These examples help illustrate how Docker can provide a consistent structure for building and deploying telecom-grade workloads in both lab and production environments, without tying deployments to a specific hardware or hypervisor layer.

Kubernetes is an open-source container orchestration platform designed to manage the deployment, scaling, and lifecycle of containerized applications. It provides declarative control over infrastructure, allowing operators to define desired states for applications and automatically reconcile them against the actual state. This model is particularly applicable to CNFs which may need to be distributed across multiple availability zones, scaled based on load, or restarted in case of failure. Kubernetes manages these operations through built-in scheduling, health checks, and resource controllers [5].

Kubernetes also introduces mechanisms for managing configurations and secrets, providing a structured way to inject environment-specific parameters into running CNFs without modifying the container image itself. This supports better separation between application logic and configuration, which is critical for secure and reproducible deployments. Open initiatives such as the CNF Testbed project [21] demonstrate how Kubernetes primitives can be applied to networking workloads, offering reference implementations for deployment, observability, and fault recovery in cloud-native telecom environments.

Both Docker and Kubernetes are foundational to the modern CNF deployment model. While Docker handles containerization and packaging of network functions, Kubernetes enables automated orchestration and lifecycle management at scale. Their use in CNF environments helps enable consistent, automated, and reproducible deployment patterns, which are prerequisites for robust configuration validation practices explored in this thesis.

2.6 GitOps

GitOps is a methodology that applies version control and automation principles to the management of infrastructure and applications. In this model, any change to the system—whether to the application or infrastructure—is made by modifying files in a Git repository, which then triggers an automated process to synchronize the live environment with the declared desired state [18].

At its core, GitOps builds upon practices from DevOps and Infrastructure as Code (IaC), but introduces stricter guidelines for how deployment pipelines are constructed and how state reconciliation is performed. Rather than relying solely on manually triggered updates or ad hoc scripting, GitOps ensures that all operational changes are traceable, auditable, and reproducible through version-controlled commits.

The core GitOps principles are a set of practices that define how Git is used to manage infrastructure and application deployments[18]:

- **Declarative configuration**: All infrastructure and application configurations are written in a declarative format mostly YAML format (YAML Ain't Markup Language) [11], describing the intended state of the system.
- **Versioned and immutable**: All changes are made through Git commits, providing a full audit trail. Each state is versioned, allowing for rollback and reproducibility.
- **Automatically applied**: CI/CD systems or controllers monitor the Git repository and apply changes automatically when a commit is detected.
- **Continuously reconciled**: In full GitOps workflows, agents or operators observe the runtime environment and continuously compare it to the declared state in Git, restoring it if deviations are found.

In the context of the thesis, not all GitOps practices are implemented in their entirety. Rather, selected GitOps principles are adopted where appropriate to support secure and reliable configuration validation. Specifically, the thesis embraces:

- **Declarative configuration**: CNF configuration files are written in YAML, which is a declarative format. This allows configurations to be described in terms of desired states, which are then validated against policies.
- **Versioned and immutable storage**: Configuration files, policy catalogs, and mapping files are stored in Git-based version control systems (e.g., GitLab). This ensures that any change is recorded, reviewable, and reproducible.

- **CI/CD integration**: The validation process is integrated into the CI/CD pipeline, enabling automated execution of configuration checks upon each commit. This reflects the "automatically applied" principle, although without continuous reconciliation.

While the thesis does not employ full GitOps deployment automation (e.g., pull-based operators or live-state reconciliation), it leverages the GitOps philosophy to support structured, traceable, and automated validation of CNF configurations during development. This aligns with the goal of ensuring that CNFs are deployed with secure and policy-compliant configurations from the outset, thereby reducing the risks associated with manual misconfigurations in cloud-native 5G environments.

3 Design and Implementation

This chapter presents the design and implementation of an automated configuration validator. The validator is developed as a lightweight, platform-agnostic tool that statically analyzes CNF configuration files and verifies their compliance with predefined security and operational policies.

The design follows a modular architecture that aligns with DevOps and GitOps principles, integrating seamlessly into existing CI/CD pipelines. It supports YAML-based CNF configurations and uses a version-controlled policy catalog to validate configurations consistently across environments. This ensures early detection of misconfigurations during development and promotes policy adherence throughout the software delivery lifecycle.

This chapter details the core components of the validator system: the structure of CNF configuration files, the design of the policy catalog and mapping mechanism, and the functionality of the validator engine. The chapter also covers the practical integration of the validator into a GitLab CI/CD pipeline, demonstrating how the validator tool automates static configruation checks and provides actionable feedback.

3.1 CNF configuration files

CNFs rely heavily on structured configuration files to define their behaviour, typically defined in YAML format. YAML is widely adopted for cloud-native applications due to its human-readable syntax, support for hierarchical data structures, and compatibility with container orchestration platforms such as Kubernetes. The use of YAML also facilitates integration with Infrastructure as Code (IaC) and GitOps workflows, where configurations are version-controlled and declaratively defined.

Each CNF within a cloud-native application maintains its own dedicated configuration file. This decoupling ensures that function-specific parameters are independently defined and updated, promoting modularity and reusability. For example, in a typical 5G core deployment, different CNFs such as the Access and Mobility Management Function (AMF), User Plane Function (UPF), Charging Function (CHF), and Session Management Function (SMF) are implemented as standalone microservices. Accordingly, their configurations are encapsulated in individual files commonly named amf.yaml, upf.yaml, chf.yaml, and smf.yaml respectively.

These configuration files encapsulate operational parameters that are critical for CNF functionality, including service interfaces, IP bindings, protocol configurations, security credentials, retry

mechanisms, and logging preferences. Since CNFs often operate in high-availability, multitenant environments, misconfigurations — even if minor — can propagate rapidly, leading to runtime failures, degraded performance, or security vulnerabilities. Hence, maintaining consistency, correctness, and compliance in these YAML configurations is paramount.

To illustrate the structure and content of a typical CNF configuration file, Listing 3.1 presents an example for the AMF. The AMF is responsible for managing user registration, connection, and mobility procedures. It serves as a central signaling point between user equipment (UE) and the 5G core network and must therefore be configured to handle interface bindings, security policies, and service registration settings with precision.

Listing 3.1: Example *amf.yaml* configuration file [6]

```
info:
  version: 1.0.9
  description: AMF initial local configuration

configuration:
  amfName: AMF # the name of this AMF
  ngapIpList:  # the IP list of N2 interfaces on this AMF
    - 127.0.0.18
  ngapPort: 38412 # the SCTP port listened by NGAP

  # Service-based Interface (SBI) Configuration
  sbi:
    scheme: http # the protocol for sbi (http or https)
    registerIPv4: 127.0.0.18 # IP used to register to NRF
    bindingIPv4: 127.0.0.18  # IP used to bind the service
    port: 8000 # port used to bind the service
    tls: # the local path of TLS key
      server:
        scheme: https
        private_key: sysconfig/open5gs/tls/amf.key
        cert: sysconfig/open5gs/tls/amf.crt
        verify_client: true
        verify_client_cacert: sysconfig/open5gs/tls/ca.crt
      client:
        scheme: https
        cacert: sysconfig/open5gs/tls/ca.crt
        client_private_key: sysconfig/open5gs/tls/amf.key
        client_cert: sysconfig/open5gs/tls/amf.crt

  # NAS Security Configuration
  # the priority of integrity algorithms
  # the priority of ciphering algorithms
  security:
    integrityOrder:
      - NIA2
    cipheringOrder:
      - NEA0
      - NEA2
# Optional NGAP Information Elements (IE)
  ngapIE:
    mobilityRestrictionList: # Mobility Restriction List IE, refer to TS 38.413
      enable: true # append this IE in related message or not
    maskedIMEISV: # Masked IMEISV IE, refer to TS 38.413
      enable: true # append this IE in related message or not
    redirectionVoiceFallback: # Redirection Voice Fallback IE, refer to TS 38.413
      enable: false # append this IE in related message or not

  # retransmission timer for paging message
  t3513:
    enable: true     # true or false
    expireTime: 6s   # default is 6 seconds
    maxRetryTimes: 4 # the max number of retransmission
```

```
logger: # log output setting
  enable: true # true or false
  level: debug # how detailed to output, value: trace, debug, info, warn, error, fatal, panic
  reportCaller: false # enable the caller report or not, value: true or false
```

Listing 3.1 is an example of a configuration file for AMF, it illustrates how various operational parameters are defined using a declarative YAML structure. Below are brief explanations of a few key configuration blocks:

- **amfName**: Specifies the name of the AMF instance. This identifier is used during service registration and logging.
- **ngapIpList and ngapPort**: Define the IP address and port for the N2 interface, which handles signaling between the AMF and the RAN.
- **sbi**: Configures the Service-Based Interface, including protocol, IP bindings, port, and TLS settings. These values control how the AMF communicates securely with other 5G core components via APIs.
- **security**: Sets the preferred integrity and ciphering algorithms (e.g., NIA2, NEA0, NEA2) for secure message exchange.
- **t3513**: Manages the paging retransmission timer, determining how long the AMF waits and how many retries it performs when paging a UE.
- **logger**: Specifies logging preferences such as verbosity (debug) and whether to include caller details in the logs.

This configuration structure ensures modularity and clarity, making it easier to manage, validate, and maintain as part of an automated CI/CD pipeline. The selected fields demonstrate how critical operational and security-related parameters are captured and controlled at deployment time.

3.2 Policy catalog

The policy catalog is a collection of *policies*, each composed of multiple *controls*, that are applied to validate configuration files across various CNFs. Policies are typically developed by Mobile Network Operators (MNOs) or vendors based on industry best practices, internal security requirements, and regulatory guidelines. They are maintained under strict version control, usually via Git, ensuring traceability, reproducibility, and the ability to audit changes over time.

A policy within the catalog defines a set of expected configuration patterns or validation requirements. These policies are composed of one or more *controls*, with each control specifying

conditions that must be met in a CNF's configuration file,such as the presence of TLS certificates, proper logging parameters, or valid cipher suite declarations. While some policies may be specific to a particular network function (e.g., an AMF), others are more generic and can be applied across multiple CNFs. For example, a policy enforcing secure communication or standardized logging can be reused by any CNF that supports those configuration blocks. The catalog is designed to be modular and flexible, allowing different combinations of policies to be mapped to different CNFs as needed. The policy catalog is usually defined as YAML.

Listing 3.2 Example policy catalog file.

```
policies:
  - name: "CHF generic policy"
    policy_id: "chf_general"
    description: "this policy contains a typical chf related configurations"
    controls:
      - "sbi"
      - "tls"
      - "cgf"
      - "rfDiameter"
      - "logger"

  - name: "AMF generic policy"
    policy_id: "amf_general"
    description: "this policy contains a typical amf related configurations"
    controls:
      - "logger"
      - "tls"
      - "security"
      - "mobilityRestrictionList"
      - "t3513"

controls:
  - name: "logger"
    description: "this control checks if logs file is present"
    enable: true
    level: info
    reportCaller: true

- name: "sbi"
    description: "this control is for the sbi configuration"
    scheme: "https"
    registerIPv4: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"
    bindingIPv4: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"

- name: "cgf"
    description: "cgf configurations"
    enable: true
    hostIPv4: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"
    port: "^(6553[0-5]|655[0-2]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|\d{1,4})$"
    listenPort: "^(6553[0-5]|655[0-2]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|\d{1,4})$"
```

```
- name: "tls"
    description: "this control checks if tls certificates is present"
    server:
      scheme: "https"
      private_key: "^.*\\/.*\\.key$"
      cert: "^.*\\/.*\\.crt$"
      verify_client: true
      verify_client_cacert: "^.*\\/.*\\.crt$"
    client:
      scheme: "https"
      cacert: "^.*\\/.*\\.crt$"
      client_private_key: "^.*\\/.*\\.key$"
      client_cert: "^.*\\/.*\\.crt$"

- name: "security"
    description: "this is the security configuration"
    integrityOrder:
      - "NIA2"
    cipheringOrder:
      - "NEA0"
      - "NEA2"

- name: "rfDiameter"
    description: "rfDiameter used in chf nf configurations"
    protocol: "tcp"
    hostIPv4: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"
    port: "^(6553[0-5]|655[0-2]\d|65[0-4]\d{2}|6[0-4]\d{3}|[1-5]\d{4}|\d{1,4})$"

  - name: "pfcp"
    description: "The listen IP and nodeID of the N4 interface on this UPF"
    addr: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"
    nodeID: "^(\d{1,3}\.){3}\d{1,3}(:\d{1,5})?$"
    retransTimeout: "2s"
    maxRetrans: 3

- name: "mobilityRestrictionList"
    description: "Mobility Restriction List IE"
    enable: true

  - name: "t3513"
    description: "retransmission timer for paging message"
    enable: true
    expireTime: '6s'
    maxRetryTimes: 3
```

Listing 3.2 defines an example of a policy catalog. The policy defined with *policy_id: amf_general* consists of a group of five controls intended to validate key aspects of the *amf.yaml* configuration file. Each control corresponds to a specific configuration block that commonly appears in CNF deployments, particularly those involving the Access and Mobility Management Function (AMF). The purpose of these controls is to help ensure that critical parameters are defined correctly and follow expected formats or constraints.

The "*tls"* control is designed to verify that secure communication is enabled. It checks that both the server and client sections of the TLS configuration use the "*https"* scheme and that file paths for keys and certificates match the required format. This helps reduce the risk of missing or incorrect certificate definitions, which are essential for secure service-based interface (SBI) communication.

The "*security"* control evaluates whether the appropriate integrity and ciphering algorithms are included in the configuration. In this case, it expects algorithms like "*NIA2"* for integrity and "*NEA0"* or "*NEA2*" for encryption. These are standard algorithms used in 5G network deployments to ensure confidentiality and message integrity, especially during signaling between network functions.

The "*logger"* control ensures that logging is enabled and configured with a minimum log level of info, along with the "*reportCaller*" flag set to true. Proper logging supports troubleshooting, observability, and operational visibility, making this a valuable component for network monitoring and diagnostics.

It is also important to understand how these policies are applied. While policies can be designed to be reusable or specific, they are typically selected and associated with CNFs as part of the configuration governance process. This association is not arbitrary but rather determined based on the function, role, and security requirements of each CNF. For example, some controls like "*tls"* or "*logger"* may be applicable across multiple CNFs, whereas others like "*t3513"* may be relevant only to one network function or a few network functions.

This flexible structure allows policies to be composed and reused as needed. It supports different deployment scenarios where not all CNFs share the same requirements. A single CNF may inherit multiple relevant policies, and conversely, not all policies need to apply to every CNF.

It is worth noting that this policy catalog operates at the configuration validation level. It is designed to work independently of platform-specific policy engines, such as those used in container orchestration environments. While Kubernetes-native tools like OPA Gatekeeper [22] or Kyverno [23] are effective for enforcing runtime and infrastructure-level rules (e.g., limiting privileged containers or restricting image sources), they may not evaluate telecom-specific configuration semantics or domain-specific formats such as those used in CNF YAML files.

3.3 Policy mapping and the application

Each Cloud-Native Network Function (CNF) in an application must be validated against the correct set of configuration policies. To manage this, the validator uses a *policy mapping file*. This file acts as a guide that connects each CNF like AMF, UPF, and CHF to one or more relevant policies defined in the policy catalog.

For example, *amf.yaml* might be validated using a policy called "*amf_general"*. Similarly, the *upf.yaml* and *chf.yaml* files are mapped to their respective policies. These mappings are written in YAML format and stored in the application's Git-based repository.

The policy mapping file is version-controlled along with the application's configuration files. This ensures that any change in policy assignments is traceable and auditable over time.

During development, when the CI/CD pipeline runs, the validator tool reads this mapping file. Based on the mappings, it loads the right policies and applies them to each configuration file. This automated check helps maintain consistency, security, and correctness across all CNFs.

Listing 3.3 Policy mapping file

```
amf:
  - amf_general

chf:
  - chf_general

upf:
  - upf_general
```

As shown in Listing 3.3, the configuration for 'amf' includes the policies with policy ids (see section 3.2) 'amf_general', 'chf_general', and 'upf_general'. The validator tool is tasked with validating these policies. Using the mapping file as a guide, the validator tool systematically examines each configuration file linked to the network functions.

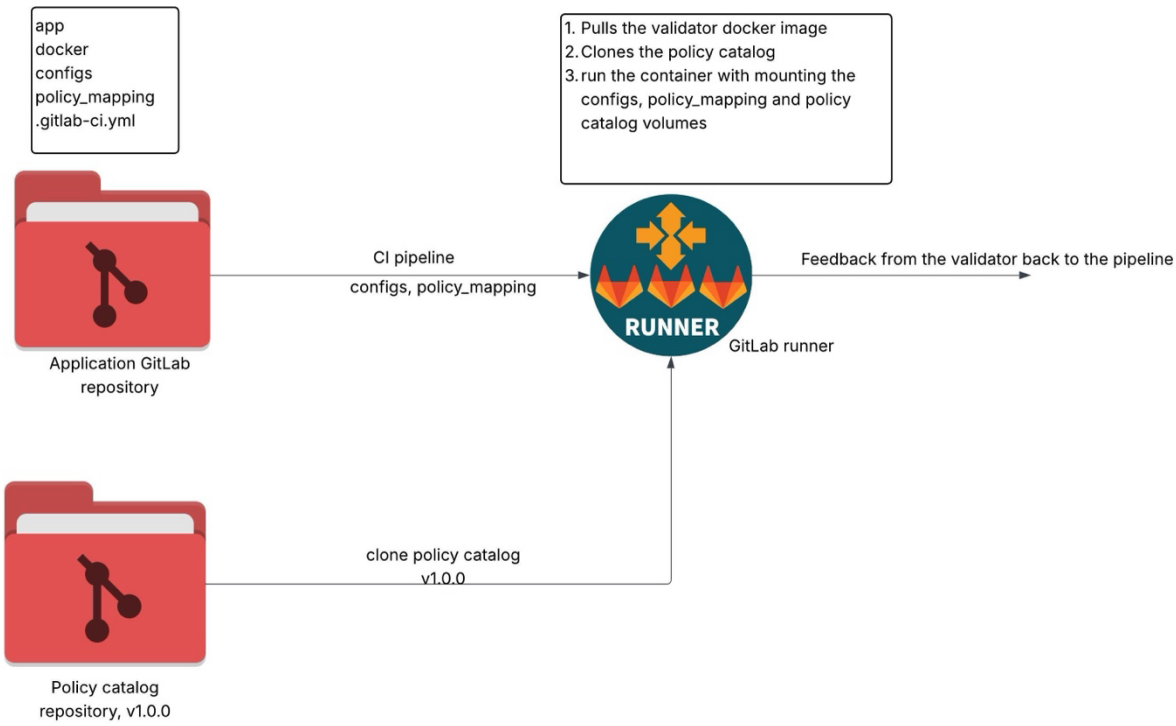
3.4 Validator

This section explains the implementation of the validator tool developed for static validation of configuration files associated with Cloud-Native Network Functions (CNFs). The validator is packaged as a lightweight Docker image, which allows seamless integration into CI/CD pipelines.

Within the CI/CD pipeline, the validator is executed inside a container by the GitLab runner. All required inputs, CNF configuration files, the policy mapping, and the policy catalog are made available within the runner's working directory. These inputs are passed into the validator container during runtime, using appropriate path bindings defined in the pipeline configuration (refer to Listing 3.6 for the *.gitlab-ci.yml* file).

The validator first loads the policy catalog, which is defined in YAML, and converts it into JavaScript Object Notation (JSON) format for internal use (see Listing 3.4). This conversion is required because the validation engine utilizes JSON-compatible tools such as JMESPath [15] for querying policy conditions. JSON's structure also enables simpler traversal and matching against configuration attributes [12].

Figure 3.1 Validation mechanism



After converting the catalog, the validator reads the policy mapping file to determine which policies are associated with each CNF (e.g., amf.yaml → amf_general). Using this mapping, it loads the configuration files and performs validation checks against the expected fields and values defined in the relevant controls.

If non-compliance is detected, for example, using *http* instead of *https*, or setting a logging level to *debug* when *info* is required, the tool records these issues. It then returns a corresponding exit code to the pipeline.

This validation step is integrated into the CI/CD flow and does not require manual invocation. The

job’s outcome, including success or failure status and detailed feedback (example in listing 3.5), is stored as an artifact, supporting traceability and auditability throughout the software lifecycle.

By verifying configurations at an early stage in development, this validator helps enforce consistent and secure CNF deployments. Its integration into automated workflows supports shift-left practices and reduces the risk of misconfigurations entering production environments.

Listing 3.4 Policy Catalog in JSON format

```
{
 "policies": [
   {
    "policy_id": "amf_general",
    "controls": ["logger", "tls", "security"]
   }
 ],

 "controls": [
   {
    "name": "logger",
    "level": "info",
    "reportCaller": true
   }
 ]
}
```

Listing 3.5 Feedback from the validator

```
  "amf": {
     "logger": {
        "level": "expected: info, found: debug",
        "reportCaller": "expected: True, found: False"
     },
  }
```

Fgure 3.2, Validation process simulation

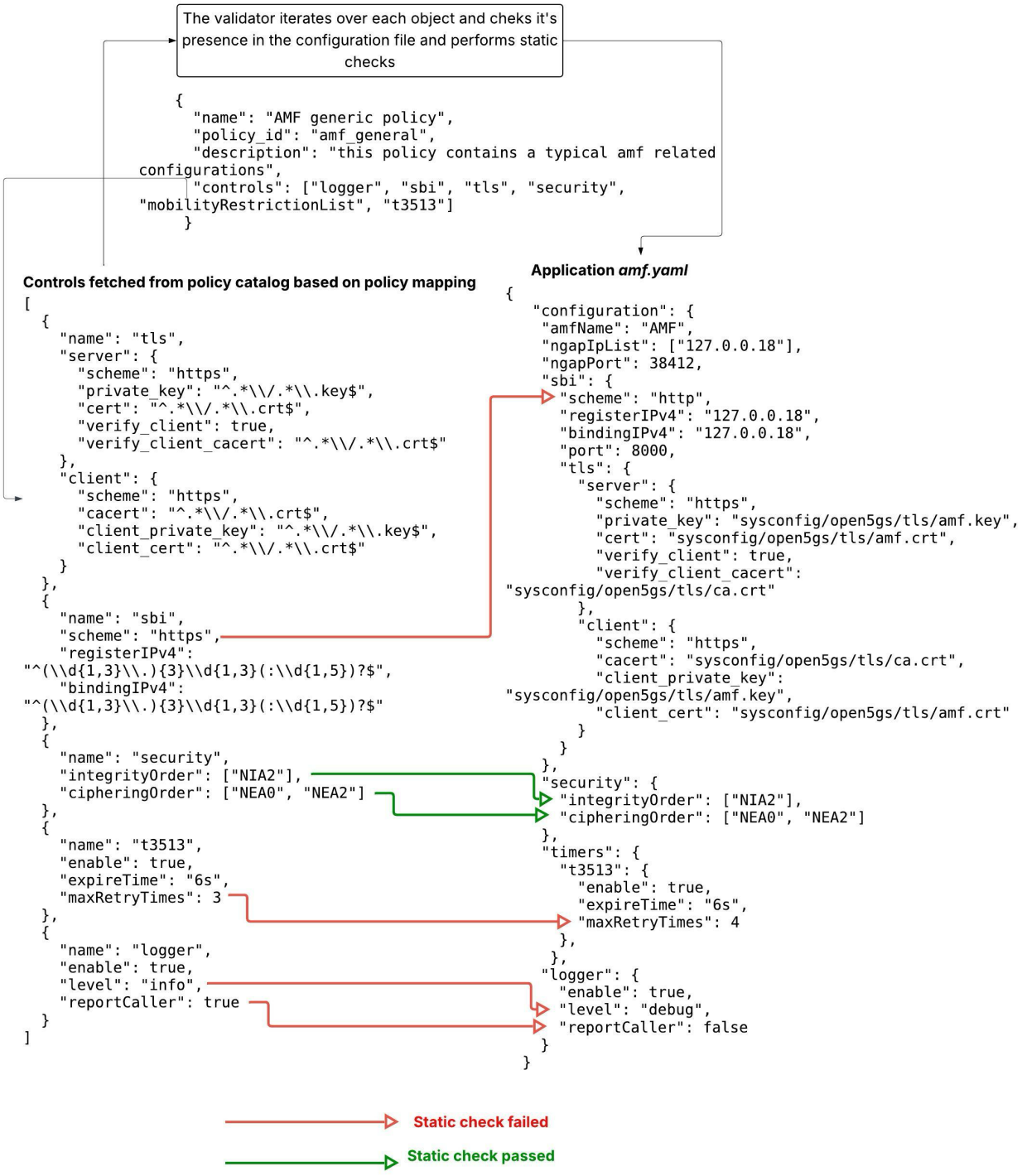


Figure 3.2 demonstrates the validation mechanism.

3.5 Pipeline

For the proof of concept, a CI/CD pipeline has been implemented using GitLab CI/CD [13]. This setup ensures that every time a commit is pushed to the repository, an automated validation process is triggered. The GitLab repository serves as the central storage for the application code, and the .gitlab-ci.yml configuration file defines the pipeline's workflow, including its stages and jobs. Listing 3.6 illustrates the structure of the .gitlab-ci.yml file, which governs the execution sequence of the pipeline. In the “validate” job, a specific version of the policy catalog, referenced through the POLICY_VERSION variable, is cloned. The validation process is executed within a Docker container, initiated through a docker run command that binds necessary volumes, ensuring the required files are accessible within the containerized environment. Since the pipeline is executed on a shell runner hosted on the local machine, it does not require pulling the Docker image, as the image is already available. This means that the runner executing the pipeline is configured to run directly on the host machine’s shell environment, allowing it to access pre-installed dependencies, including the Docker engine. However, it is important to note that this setup is chosen exclusively for this proof of concept to simplify execution and debugging. In a real-world deployment, this pipeline can be implemented using different types of GitLab runners, depending on the specific infrastructure requirements. For example, Docker executor could be used to encapsulate the entire validation process within isolated Docker containers, ensuring a clean and reproducible environment for each execution. Additionally, Kubernetes runners could be leveraged to scale the validation process dynamically, allowing multiple validation jobs to run in parallel across a cluster. Custom runners tailored to an organization's needs, such as using virtual machines or cloud-based environments, can also be employed to achieve the same functionality. The flexibility in selecting a runner type ensures that this validation pipeline can be adapted to different CI/CD architectures, making it highly portable across various environments.

Figure 3.3 An example of GitLab pipeline

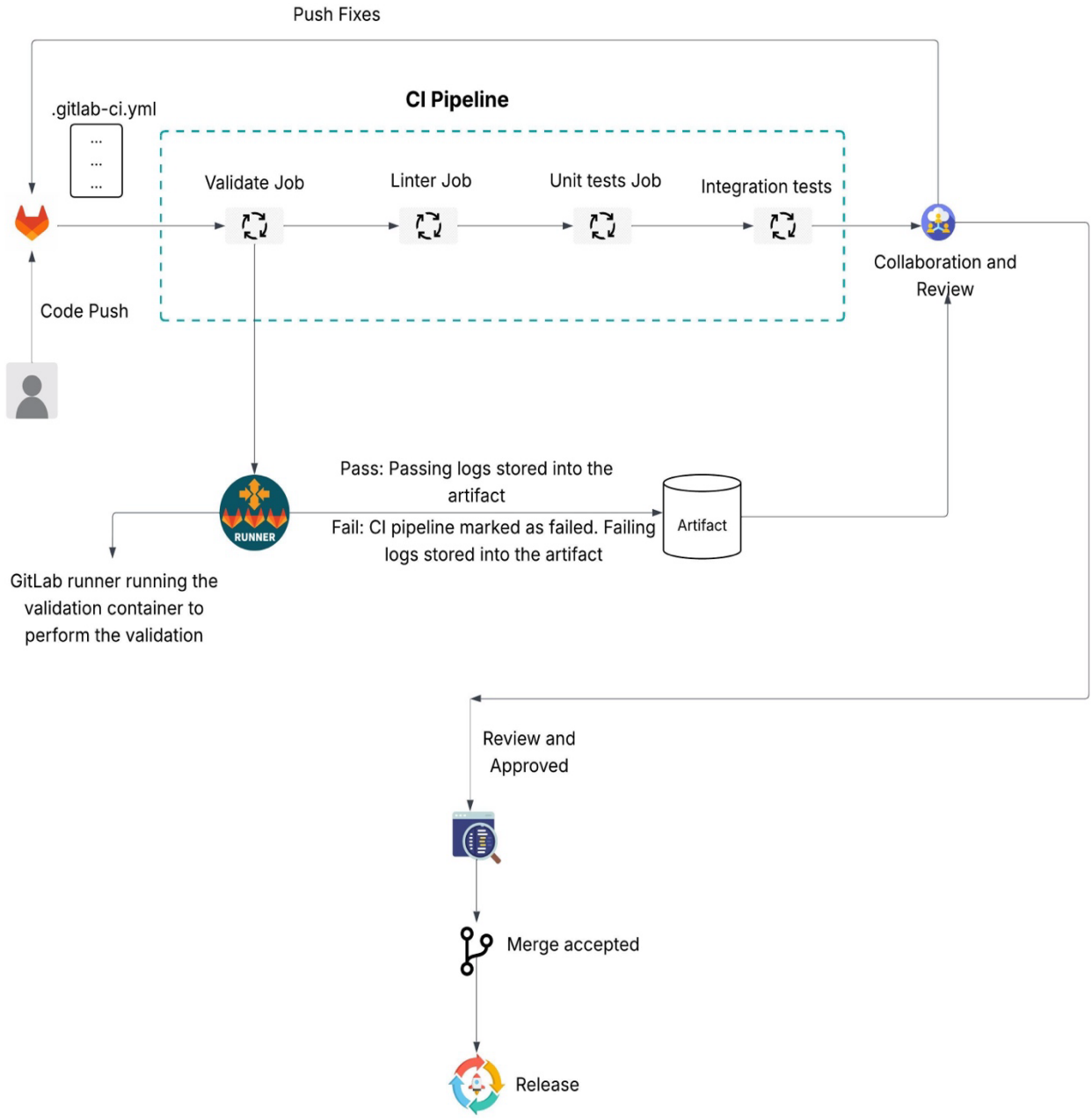


Figure 3.3 depicts a typical CI/CD pipeline, which incorporates various jobs including unit testing, linting, and integration testing. In addition to these, the pipeline also features a validation job. This job is responsible for the static validation of configuration files, as shown in Figure 3.2.

The validation task is executed by a GitLab runner, which pulls a Docker image designed for this purpose. Within this environment, the validation process outlined in Section 3.4 is carried out. After completing the validation, the tool within the Docker container returns an exit status to indicate the results. A return code of '0' means that the validation was successful, indicating compliance with all policy requirements. Conversely, a return code of '1' signals a failure due to non-compliance in one or more configuration files.

The pipeline records this validation result and stores it in an artifact file. If this job or any other job in the pipeline fails, it indicates that the CI pipeline was unsuccessful, prompting the need for additional fixes and a restart of the process.

Once all tests in the pipeline have been successfully passed, a reviewer typically evaluates and, if satisfied, approves the merge request. The approved changes are then merged into the master branch, completing the integration process.

Listing 3.6 Example *".gitlab-ci.yml"* file

```
stages:
  - validate

variables:
  POLICY_REPO: "https://gitlab-ci-
token:${CI_JOB_TOKEN}@${COMPANY_INTERNAL}/ejamkai/validator-policy-catalog.git"
  POLICY_VERSION: "v2.0.0"
  DOCKER_IMAGE: "validator:2.0.0"
  CONFIG_PATH: "configs"
  POLICY_PATH: "/policy_repo"
  POLICY_MAPS: "policy_maps"

validate:
  stage: validate
  tags:
    - shell
  before_script:
    - mkdir -p public
    - mkdir -p "${POLICY_PATH}"
    - echo "Cloning policy repository..."
    - git clone --depth 1 --branch "${POLICY_VERSION}" "${POLICY_REPO}" "${POLICY_PATH}"

  script:
    - echo "Starting validation process..."
    - echo "Catalog version ${POLICY_VERSION}" > public/result.txt
    - echo "Running Docker container for validation..."
    - |
      set -o pipefail
      sudo docker run \
        -v "$(pwd)/${CONFIG_PATH}:/app/configs" \
        -v "$(pwd)/${POLICY_MAPS}:/app/policy_maps" \
        -v "${POLICY_PATH}:/app/policy_repo" \
        "${DOCKER_IMAGE}" 2>&1 >> public/result.txt
      STATUS=$?
      echo "Docker run completed with exit code $STATUS" >> public/result.txt
    - echo "Validation completed. Checking Docker exit status..."
    - cat public/result.txt
    - STATUS=$(cat public/result.txt)
    - echo "Exit status $STATUS"
    - |
      if [ "$STATUS" -ne 0 ]; then
        exit 1
      fi
  after_script:
    - echo "Cleaning up created directories..."
    - rm -rf "${POLICY_PATH}"
    - echo "Cleanup complete."
```

```
artifacts:
  paths:
    - public/result.txt
  expire_in: 100 days
  when: always
```

Listing 3.6 illustrates an example of the pipeline YAML file, which is included in the application repository. The development team is responsible for defining this pipeline, which specifies several important elements. These include the URL to the policy catalog repository, detailed in the variable "POLICY_REPO," the version of the policy, the Docker image along with its version, and the paths to the configuration files and policy mapping file.

In the validation job, before the execution of the script, the job first clones the policy catalog repository. Following this, it executes the 'docker run' command, which includes volume binding to ensure the necessary files are accessible within the Docker container. The output from this 'docker run' command is subsequently captured and stored in a file named 'public/results.txt', which is preserved as an artifact.

Listing 3.7 Pipeline artifact containing failure information

```
Catalog version v1.0.1
2025-04-27 15:46:52,646 - ERROR - Validation failed: some config files are not compliant.
2025-04-27 15:46:52,646 - ERROR - Validation result: {
  "chf": {
    "rfDiameter": {
      "protocol": "expected: tcp, found: udp"
    },
    "logger": {
      "level": "expected: info, found: debug",
      "reportCaller": "expected: True, found: False"
    }
  },
  "amf": {
    "logger": {
      "level": "expected: info, found: debug",
      "reportCaller": "expected: True, found: False"
    },
    "sbi": {
      "scheme": "expected: https, found: http"
    },
    "t3513": {
      "maxRetryTimes": "expected: 3, found: 4"
    }
  },
  "upf": {
    "logger": {
      "level": "expected: info, found: debug",
      "reportCaller": "expected: True, found: False"
    },
    "pfcp": {
      "retransTimeout": "expected: 2s, found: 1s"
    }
  }
}
```

Listing 3.8 Pipeline artifact containing success information

```
Catalog version v1.0.0
2025-04-27 12:13:05,640 - INFO - All config files are compliant with policy requirements.
Docker run completed with exit code 0
```

Listings 3.7 and 3.8 represent the outputs generated by the docker run command, which are subsequently stored as artifacts. These artifacts provide crucial details regarding the validation process, including specific compliance issues detected within the configuration files. For instance, in Listing 3.7, the validation output highlights a policy violation within multiple network function configurations particularly related to loggin and scheme.

Additionally, the artifact records the version of the policy catalog against which the configurations were validated, ensuring traceability and consistency in the compliance assessment.

4 Analysis and Future Work

This chapter assesses the effectiveness of the configuration validator tool developed for Cloud-Native Network Functions (CNFs). The impact of the validator on the security and reliability of CNFs is examined by analyzing its integration with CI/CD pipelines and its role in automating the validation process. Additionally, the chapter outlines the current limitations and challenges, and proposes potential improvements for future development.

4.1 Security and Compliance

The configuration validator tool streamlines the process of ensuring that Cloud-Native Network Functions (CNFs) adhere to security policies throughtout the development phase. By integrating this tool into CI/CD pipelines, developers receive immediate feedback on configuration errors, facilitating swift rectification and preventing the deployment of non-compliant configurations. This proactive approach upholds high security and compliance standards across cloud-native network functions.

The validator demonstrated its capability by performing checks on all network function configuration files within the free5GC [6] GitHub repository a publicly available project dedicated to 5G core mobile networks in less than a second. Among the issues identified were configurations where logging levels were set to 'debug' instead of the expected 'info', and where insecure communication schemes such as 'http' were used instead of 'https'. While these misconfigurations might appear minor, when they are detected late, typically during product testing just before deployment, they trigger a chain of time-consuming steps. The configurations must first be corrected, followed by repeated rounds of integration and security testing to ensure no regressions or new issues were introduced. This process may delay product releases by days or even weeks. The resulting impact is not only financial due to prolonged testing cycles and resource usage, but also reputational, especially in fast-paced markets where timely and secure delivery is expected. In some cases, these issues are not caught during testing at all and make their way into production, where they may be flagged by external security audits or runtime compliance tools. Discovering such problems post-deployment requires even more costly remediation and can seriously affect customer trust.

The validator tool aims to detects these issues during development within seconds, minimizing rework and reducing the risk of flawed CNFs reaching production. This supports the broader

practice of shifting security left within CI/CD pipelines, promoting more secure and efficient deployments. Through early detection, the validator improves development efficiency and helps ensure compliance, reducing reliance on manual testing phases. It mitigates the costs tied to redeployments, debugging, prolonged downtime, and potential losses due to regulatory penalties or missed business opportunities. By automating the validation process, the tool also decreases the dependency on manual reviews, enabling developers to focus on implementation rather than compliance. This leads to improved productivity while embedding essential configuration checks earlier in the lifecycle. The tool thus contributes to a stronger security posture and aligns with established principles in technology governance and risk management.

Another application of the validator and its feecabck can be that a compliance matrix can be developed to correlate different versions of an application with versions of the policy catalog. This matrix helps in tracking the status of compliance standards and their impact on CNF configurations.

Table 4.1 Compliance Matrix

Application version	Policy catalog version	Status	Description
v1.0.0	v1.0.0	Passed	All configurations are compliant
v1.0.0	v1.0.1	Failed	Amf validation failed
v1.0.1	v1.0.1	Passed	All configurations are compliant

Table 4.1 is an example of such a compliance matrix which can be stored in a separate location such as a wiki page. This file can be appended with data by the CI/CD pipeline at the time of creating the artifact (see section 3.5) for the pipeline run.This matrix serves as a dashboard to quickly identify which changes of the policy and which version of the policy catalog led to failures in configuration validations. It aids in diagnosing issues more swiftly and informs updates to both the configurations and the policies. This tool is particularly useful for teams managing multiple versions of CNFs and policy catalogs, ensuring that all deployed versions meet the required compliance standards.

4.2 Limitations

Although the validation tool offers significant advantages in improving CNF security and operational compliance, it is not without its limitations.

4.2.1 Dependency on Policy Catalog

The effectiveness of the validation tool is reliant on the comprehensiveness and relevance of the policy catalog. If the catalog is not comprehensive or updated frequently to reflect the latest security standards and best practices, the tool may fail to catch new types of misconfigurations or security vulnerabilities. Keeping the policy catalog up-to-date requires continuous effort.

4.2.2 Lack of contextual awareness

The validator tool operates on a static set of rules defined in a policy catalog, which does not account for runtime context or environmental conditions. Some security misconfigurations are only identifiable in the context of specific deployment environments or runtime behaviors. For example, configurations that are valid in isolation may conflict with policies or cause issues when deployed alongside other CNFs, particularly when shared resources (e.g., ports, volumes, secrets) are involved. The lack of contextual awareness means the validator may produce false negatives, passing configurations that later result in security flaws during runtime. It is critical that the policy catalog is carefully curated to include only those checks that are meaningful and reliable when applied in isolation.

4.2.3 Challenges with Configuration File management

As the product becomes more complex, so does the amount of configuration files. These files are not only for CNFs but also for other parts of the application such as Docker configuration files. Identifying the correct configuration files specific to CNFs can be challenging, requiring precise paths and mounting to the validator tool container to ensure accurate validation. This task becomes even more difficult if different configuration files are located in disparate locations.

4.2.4 Uncertain performance at Large-Scale

While the validator has proven effective in controlled proof-of-concept environment, its performance and scalability in larger, more complex deployments have not been thoroughly tested. In large-scale CNF environments, like those found in modern telecommunications networks, the

tool's ability to efficiently process and analyze extensive configuration data without substantial delays or resource overhead is essential.

4.2.5 Static check limitations

The validator is rule-based and deterministic, which limits its ability to detect emergent or anomalous configurations that are not explicitly defined in the policy catalog. Unlike some modern security tools that use heuristics, learning models, or pattern detection to identify suspicious but syntactically correct configurations, this tool lacks the capability to flag such issues, potentially missing zero-day misconfiguration types.

4.3 Future Work

The validator tool can be improved by integrating it with a real-time security compliance monitoring tool such as Ericsson Security Manager (ESM) [9]. A new step can be added to the CI/CD pipeline, which runs only after the static validation step passes. In this stage, the CNF application is deployed to a controlled testing environment, where ESM can perform a runtime compliance check. This would allow security to be validated not only from static configurations but also based on the application's behavior in an actual deployment scenario. Combining static validation with runtime validation offers stronger and more complete assurance of security before production.

For the issue of contextual awareness, one possible improvement is to mark certain policies as "deployment-dependent" or "runtime-sensitive" within the catalog. These policies can be validated only in later stages of deployment, rather than during static checks. This separation helps avoid false positives or negatives during early validation while still enforcing stricter checks in testing or staging environments.

To improve handling of complex file structures and configuration file discovery, the validator can be enhanced to support directory scanning with filtering rules. This would help automatically locate CNF-specific files based on naming conventions or directory structure, reducing the risk of overlooking relevant configurations.

The validator's integration with CI/CD pipelines can be made easier by creating reusable templates and sample pipelines for common platforms such as GitLab, Jenkins, or GitHub Actions. These templates can serve as starting points for teams and reduce setup time.

To address the validator's limitations at scale, future work should include performance testing in simulated large-scale CNF deployments. This will help measure processing time, resource use, and

scalability. If necessary, parallel processing or caching mechanisms can be introduced to support larger environments.

The lack of support for templated or dynamic configurations such as Helm charts can be partly mitigated by adding a rendering phase to the validator pipeline. This would involve resolving values before validation begins, so that only finalized configuration files are checked.

Severity tagging for policy violations can also be added. This would help teams prioritize which issues to address first. Categories like “critical,” “warning,” and “informational” would help reduce noise and focus developer attention on the most impactful problems.

5 Conclusion

The thesis addressed the configuration challenges in Cloud-Native Network Functions (CNFs) and proposes a static, automated validation tool that integrates with CI/CD pipelines. To address the challenge of misconfiguration of CNFs, a policy-driven static validation approach was developed. The validator tool was designed to parse CNF configuration files written in YAML, convert them to JSON, and evaluate them against a centralized, version-controlled policy catalog. This process ensures that configurations meet predefined security and operational standards before deployment.

The validator integrates into CI/CD workflows and provides immediate feedback to developers during the development phase. This early validation helps prevent misconfigurations such as incorrect protocol settings or insecure logging, from reaching production environments. By automating this process, the tool reduces reliance on manual checks and supports more consistent and secure CNF deployments.

While the tool effectively improves configuration security and policy compliance during development, further work is needed to address scalability in large-scale deployments and to support dynamic or templated configurations. Additionally, the static nature of validation limits its ability to detect context-specific issues that may only surface at runtime. Enhancements such as runtime validation integration, policy severity tagging, and support for Helm-based templates could expand its applicability.

Overall, the solution contributes a practical method for introducing automated, policy-based configuration checks into modern CNF development workflows. It offers a foundation for improving configuration governance in cloud-native telecom environments, where agility, reliability, and compliance are critical to operational success.

The thesis used a large language model (LLM), ChatGPT by OpenAI, to support the writing process. It was used to correct grammar, improve sentence structure, and make the text clearer. All technical work, including the design, implementation, analysis, and results, is the author's own. The model was not used to generate or validate research data, code, or original technical solutions, but solely to support writing refinement and document presentation.

Bibliography

[1] S. Imadali and A. Bousselmi, "Cloud Native 5G Virtual Network Functions: Design Principles and Use Cases," *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, Paris, France, 2018, pp. 91-96, doi: 10.1109/SC2.2018.00019. Page: 1-5, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8567377

[2] ETSI, "Network Functions Virtualisation (NFV): Management and Orchestration", Blog post, https://www.etsi.org/technologies/nfv Last accessed: 22/05/2025

[3] Axel Sukianto, UpGuard, "Cloud Misconfiguration: The Risk to Your Data, Point 8: Lack of Validation, Blog post, https://www.upguard.com/blog/cloud-misconfiguration, Last accessed 22/05/2025

[4] Greg Young, "Misconfigurations: The Biggest Threat to Cloud Security", InfoSecurity Magazine, https://www.infosecurity-magazine.com/blogs/misconfigurations-threat-to-cloud, Last accessed: 22/05/2025

[5] J. Shah and D. Dubaria, "Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform," *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, USA, 2019, pp. 0184-0189, doi: 10.1109/CCWC.2019.8666479,https://ieeexplore.ieee.org/abstract/document/8666479/references#references

[6] free5GC ,GitHub repository https://github.com/free5gc/free5gc Last accessed: 21/02/2025

[7] Jyotsna Agrawal , Rakesh Patel , Dr. P.Mor , Dr. P.Dubey and Dr. J.M.keller, “Evolution of Mobile Communication Network: from 1G to 4G” https://ijmcr.com/wp-content/uploads/2015/11/Paper11100-1103.pdf

[8] Azar Abid Salih1 , Subhi R. M. Zeebaree2 , Ahmed Sinali Abdulraheem3 , Rizagr R. Zebari4 , Mohammed A. M.Sadeeq5 , Omar M. Ahmed6, “Evolution of Mobile Wireless Communication to 5G Revolution”, Academic paper, https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q=Evolution+of+Mobile+Wireless+Communication+to+5G+Revolution+&btnG= , Last accessed: 22/05/2025

[9] Ericsson Security Manager. Product documentation, https://www.ericsson.com/en/portfolio/networks/security-and-risk-management/security-manager Last accessed: 22/05/2025

[10] Docker, Product documentation https://docs.docker.com/get-started/docker-overview/ Last accessed 22/05/2025

[11] RedHat, What is YAML?, Blog post, https://www.redhat.com/en/topics/automation/what-is-yaml Last accessed: 22/05/2025

[12] Introduction to JSON, Technical documentation https://www.json.org/json-en.html Last accessed: 22/05/2025

[13] Get started with GitLab CI/CD, Technical documentation, Product documentation, https://docs.gitlab.com/ci/ Last accessed: 22/05/2025

[14] VNF and CNF, what's the difference? Red Hat, Published July 28, 2022, https://www.redhat.com/en/topics/cloud-native-apps/vnf-and-cnf-whats-the-difference, Last accessed: 22/05/2025

[15] JMESPath https://jmespath.org/ Last accessed: 22/05/2025

[16] X. Wang et al., "Holistic service-based architecture for space-air-ground integrated network for 5G-advanced and beyond," in China Communications, vol. 19, no. 1, pp. 14-28, Jan. 2022, doi: 10.23919/JCC.2022.01.002.

[17] H. C. Rudolph, A. Kunz, L. L. Iacono and H. V. Nguyen, "Security Challenges of the 3GPP 5G Service Based Architecture," in IEEE Communications Standards Magazine, vol. 3, no. 1, pp. 60-65, March 2019, doi: 10.1109/MCOMSTD.2019.1800034.

[18] F. Beetz and S. Harrer, "GitOps: The Evolution of DevOps?," in IEEE Software, vol. 39, no. 4, pp. 70-75, July-Aug. 2022, doi: 10.1109/MS.2021.3119106.

[19] 3GPP, System architecture for the 5G System (5GS) (3GPP TS 23.501 version 17.5.0 Release 17), Technical documentat, Product documentation, https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/17.05.00_60/ts_123501v170500p.pdf, Last accessed: 22/05/2025

[20] Openshift, Example CNF, GitHub repository, Technical document, Product documentation, https://github.com/openshift-kni/example-cnf, Last accessed: 22/05/2025

[21] CNF testbed, GitHub repository, https://github.com/cncf/cnf-testbed, Last accessed: 22/05/2025

[22] Rita Zhang (Microsoft), Max Smythe (Google), Craig Hooper (Commonwealth Bank AU), Tim Hinrichs (Styra), Lachie Evenson (Microsoft), Torin Sandall (Styra), OPA Gatekeeper, Kubernetes Blog post, https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/, Last accesses 22/05/2025

[23] Kyverno, Product website, Technical document, Product documentation, https://kyverno.io/docs/introduction/, Last accessed: 22/05/2025

Appendix A

Figures

Listings