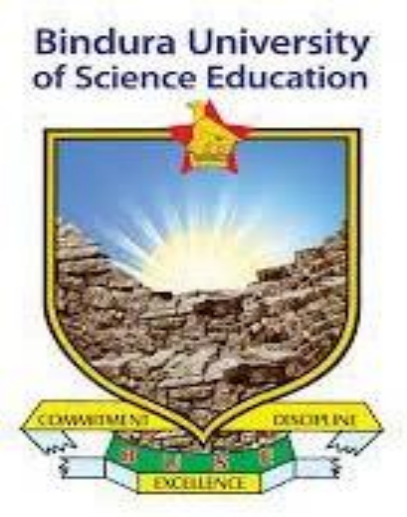


FACULTY OF ENGINEERING DEPARTMENT OF COMPUTER SCIENCE

Design and implementation of a self-healing Network configuration system with Security Policy Validation using automation tools

FINAL YEAR PROJECT

BY

WELLINGTON GWANYANYA

REG NUMBER: B221281B

SUPERVISOR: MR CHITUMA

Submitted in partial fulfilment of the requirements for Bachelor of Science Honours Degree in Network Engineering

APPROVAL FORM

The undersigned certify that they have supervised B221281B Gwanyanya Wellington dissertation entitled "**Design and implementation of a self-healing Network configuration system with Security Policy Validation using automation tools**" submitted in partial fulfillment of the requirements for the Bachelor of Science Honors Degree in Network Engineering



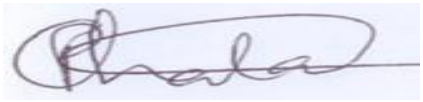
Student:…….

Date:……23/06/2026**……**



Supervisor:……

Date:……23/06/2026**……**



Chairperson:…….

Date:……23/06/2026**……**

Dedication

This work is lovingly dedicated to my family, whose unwavering faith in me has been my greatest source of strength. To my parents and guardians your sacrifices, your hopes for my future, and your constant encouragement carried me through every challenge. Nothing I achieve would be possible without the foundation you built for me.

To my lecturers and mentors, thank you for seeing my potential even when I doubted myself. Your guidance, patience, and belief in my abilities shaped not only this dissertation but also the person I am becoming.

And finally, I dedicate this work to all dreamers and learners who push forward despite obstacles. May this dissertation be a reminder that resilience, curiosity, and support from those we love can transform difficult journeys into meaningful achievements.

Acknowledgments

I would like to express my sincere gratitude to everyone who supported me throughout the completion of this university project.

My deepest appreciation goes to my family, whose constant encouragement, patience, and belief in my potential have guided me through my studies. Your support has been my strength, and I am truly grateful for everything you have done to help me reach this stage.

I am profoundly thankful to my project supervisor and the lecturers in my department for their guidance, constructive feedback, and dedication to my academic growth. Your expertise, mentor-ship, and willingness to assist have been essential to the development and completion of this project.

I would also like to thank my classmates and friends who offered ideas, shared experiences, and provided motivation along the way. Your collaboration and support made this journey more enjoyable and meaningful.

Lastly, I acknowledge the university and the resources made available to me during this research. The facilities, tools, and learning environment provided a strong foundation for carrying out this project successfully.

Abstract

Modern networks face increasing risks of misconfiguration and unauthorized changes, while most automation tools only detect issues rather than correct them. Existing self-healing systems offer automated remediation but are expensive, proprietary, and vendor-dependent, limiting their accessibility. This project develops an open-source, vendor-neutral self-healing network configuration system that automatically detects configuration drift, restores secure baselines, and validates compliance with predefined security policies. Using GNS3, Ansible, and Python, the system integrates configuration deployment, drift detection, security checks, and automated rollback into one unified framework. Testing under simulated failures and attack scenarios demonstrates that the system improves configuration integrity, reduces downtime, and enhances security compliance compared to manual network management. The solution provides a lightweight and affordable alternative for organizations seeking reliable autonomous network resilience without enterprise-level costs

Table of Contents

APPROVAL FORM i

Dedication ii

Acknowledgments iii

Abstract iv

CHAPTER 1 1

INTRODUCTION 1

1.1 Introduction 1

1.2 Background of the Study 1

1.3 Statement of the Problem 1

1.4 Research Aim 1

1.5 Research Objectives 1

1.6 Research Questions 2

1.7 Significance of the Study 2

1.8 Assumptions 3

1.9 Limitations/Challenges 5

1.11 Definition of terms 8

1.12 Summary 9

CHAPTER 2 10

LITERATURE REVIEW 10

2.1 Introduction 10

2.2 Conceptual Framework 10

2.3 Theoretical Framework 14

2.4 Empirical Evidence 14

2.5 Research Gap Analysis 16

2.6 Summary 16

CHAPTER 3 18

RESEARCH METHODOLOGY 18

3.1 Introduction 18

3.2 Research Design 18

3.3 Population and Sample 18

3.4 Research instruments 19

3.5 Reliability and Validity 22

3.6 Data Collection Process 24

3.7 Ethical Considerations 25

3.8 Feasibility Study 26

3.9 System Analysis 28
3.10 System Design 30
3.11 System Implementation and Code Structure 40
3.12 System Testing 41
3.13 Summary 43
CHAPTER 4 43
DATA PRESENTATION, ANALYSIS AND INTERPRETATION 44
4.1 Introduction 44
4.2 Analysis and Interpretation of Results 44
4.3 Summary of Findings 47
Chapter 5 47
Conclusions and Recommendations 48
5.1 Introduction 48
5.2 Conclusions 48
5.3 Recommendations 49
5.4 Future Work 49
5.5 Chapter Summary and Final Conclusion 50
REFERENCES 51

CHAPTER 1

INTRODUCTION

1.1 Introduction

Modern computer networks form the backbone of communication, service delivery, and digital operations for organizations of all sizes. As networks continue to expand in scale and complexity, managing device configurations manually has become increasingly challenging and error-prone. Configuration errors, unauthorized changes, and delayed remediation can lead to service outages, security breaches, and reduced network reliability.

Network automation tools such as Ansible, NAPALM, and configuration backup systems have significantly improved consistency and efficiency in network management. However, these tools are largely reactive, focusing on deployment and detection rather than on autonomous correction. When configuration drift or policy violations occur, human intervention is still required to diagnose the issue and restore the network to a secure and operational state. This delay increases exposure to vulnerabilities and operational downtime.

In response to these limitations, self-healing network systems have emerged, offering automated detection and remediation capabilities. While enterprise solutions such as Cisco DNA Center and Juniper Apstra provide some level of autonomous recovery, they are proprietary, expensive, and tightly coupled to specific vendor ecosystems. As a result, small and medium enterprises, educational institutions, and municipalities are often unable to benefit from these advanced capabilities.

This project addresses these challenges by designing and implementing a vendor-neutral, fully open-source self-healing network configuration system with integrated security policy validation. The system automatically detects unauthorized configuration changes, restores secure baseline configurations, and enforces compliance with predefined security policies without human intervention. Using GNS3 for network simulation, along with Ansible and Python for automation, the proposed solution provides a lightweight and cost-effective approach to improving network resilience.

By integrating configuration management, autonomous remediation, and security validation into a unified framework, this project demonstrates how intelligent automation can enhance reliability, reduce operational overhead, and strengthen security in modern network environment

1.2 Background of the Study

Configuration updates can be deployed, backed up, and monitored using current network automation and configuration management tools like Ansible, NAPALM, and RANCID. However, when illegal changes, incorrect setups, or security breaches take place, these tools still need operator intervention because they function reactively. Even though certain enterprise systems, such as Cisco DNA Center, offer some degree of autonomous remediation, small businesses, educational institutions, and municipalities cannot use them since they are proprietary, costly, and vendor-specific.

This reveals a gap: the lack of a vendor-neutral, lightweight, fully open-source system that can do autonomous rollback and assess configurations against pre-established security policies in addition to detecting unauthorized configuration changes.

By creating a system that provides self-healing capabilities without requiring expensive enterprise gear or subscriptions, this study aims to close that gap.

1.3 Statement of the Problem

Most available configuration management tools detect drift but do not autonomously remediate it. Conversely, existing self-healing systems are proprietary, costly, and dependent on vendor ecosystems such as Cisco DNA Center, Juniper Apstra, and Anuta ATOM, making them inaccessible to SMEs and institutions with limited budgets.

Furthermore, configuration management and security validation are often separate processes, causing delays that can expose networks to vulnerabilities. There is therefore a need for a **vendor-neutral, low-cost, fully automated self-healing framework** that integrates configuration drift detection, autonomous rollback, and security policy validation for improved operational reliability.

Thus, the research problem is the lack of a lightweight open-source system that automatically detects unauthorized configuration changes, restores secure baselines, and enforces security compliance without requiring human intervention.

1.4 Research Aim

To develop an open-source, automated, and self-healing network configuration system capable of detecting unauthorized configuration changes, restoring secure baseline configurations, and validating compliance with predefined security policies

1.5 Research Objectives

i. Simulate a realistic and scalable network environment using GNS3 to provide a controlled testing bed for automation and remediation logic
ii. Develop a centralized configuration management framework using Ansible and Python to automate the deployment and management of network device configurations.
iii. Implement a drift detection mechanism that utilizes bitwise comparison to accurately identify and alert on unauthorized changes to the live configuration.
iv. Construct an autonomous self-healing logic that executes automated rollbacks to restore "Golden" configurations immediately following a detected drift.
v. Validate the system's resilience and responsiveness by testing its performance and security policy enforcement under simulated network attacks and configuration errors.

1.6 Research Questions

i. How can a high-fidelity and scalable network environment be architected within GNS3 to ensure that virtualized Cisco IOSv nodes accurately mirror the configuration behaviors and protocol responses of physical hardware?
ii. What specific architectural combination of Ansible playbooks and Python scripting maximizes efficiency in pushing "Golden" configurations while maintaining a vendor-neutral approach that avoids proprietary lock-in?
iii. Through what technical mechanisms can bitwise configuration comparison be optimized to accurately detect unauthorized drift in real-time while minimizing false-positive alerts during legitimate administrative updates?
iv. What logic-based automation workflows are required to enable self-healing capabilities that can autonomously restore a secure baseline within a predefined time window without necessitating manual operator intervention?
v. How do the system's response latency and resource utilization (CPU/Memory) vary when enforcing security policy compliance under simulated stress scenarios, such as unauthorized port access or routing protocol tampering?

1.7 Significance of the Study

This study addresses critical challenges in modern network management by introducing an open-source, self-healing configuration system that integrates automated remediation with security policy validations. As network infrastructure grows in complexity, the proposed solution reduces reliance on manual intervention, improves network reliability, and enhances compliance, particularly for organizations with limited resources

These include

Implementation of the proposed system aims to significantly enhance network resilience by introducing an autonomous layer of reliability. By continuously monitoring the infrastructure to automatically detect configuration drift, the framework ensures that any unauthorized or

accidental modifications are identified in real-time. The ability to immediately restore secure baseline configurations without manual intervention minimizes the window of vulnerability and drastically reduces operational downtime. Consequently, the network maintains a consistent state of readiness and stability, ensuring that critical services remain accessible even in the face of frequent or unexpected configuration changes.

Framework significantly strengthens the network's defensive posture by ensuring constant adherence to established security standards. Through the continuous validation of active configurations against a set of predefined security policies, the system proactively identifies deviations that could expose the infrastructure to risk. This persistent auditing process ensures that hardening measures, such as disabling insecure protocols or enforcing access controls, remain in place at all times. By automating this oversight, the research demonstrates a reduction in the window of vulnerability that typically exists between a configuration error and its manual discovery, thereby maintaining a robust and compliant security environment.

This research offers a significant economic advantage by delivering a fully open-source and vendor-neutral alternative to expensive, proprietary self-healing platforms. By leveraging accessible tools such as Ansible, Python, and GNS3, the framework eliminates the substantial licensing fees and hardware lock-in associated with enterprise-grade solutions. This cost-effective approach democratizes access to advanced network resilience capabilities, making autonomous remediation and proactive configuration management attainable for small-to-medium enterprises, educational institutions, and public sector organizations with limited budgetary resources.

Implementation of the framework significantly enhances operational efficiency by shifting the burden of repetitive configuration tasks from human administrators to an automated system. By streamlining the deployment and monitoring processes, the framework minimizes the risk of human error, which is a leading cause of network instability and security vulnerabilities. This reduction in manual intervention not only ensures a higher degree of configuration consistency but also alleviates the administrative workload, allowing network teams to refocus their efforts on strategic infrastructure improvements rather than routine troubleshooting and maintenance.

1.8 Assumptions

A fundamental assumption of this research is that the network devices utilized within the simulated environment fully support remote management and automation via standard communication protocols, specifically SSH. By ensuring that the virtualized Cisco IOSv nodes are configured to accept secure, script-based connections, the study can effectively leverage tools like Ansible and Python for configuration deployment and monitoring. This reliance on industry-standard protocols ensures the framework remains broadly applicable and reinforces the vendor-neutral nature of the solution, as it avoids the need for specialized proprietary agents or legacy management interfaces.

A fundamental premise of this research is that the simulated environment architected within GNS3 provides a high-fidelity representation of real-world network behaviours' and

configuration scenarios. By utilizing authentic Cisco IOSv images, the simulation ensures that the interactions between the automation scripts and the virtual nodes mirror the functional responses of physical hardware. This accuracy is critical for validating the framework's effectiveness, as it assumes that the logic developed for drift detection and autonomous remediation in the virtual lab will be directly applicable to the operational complexities and protocol requirements found in live, physical network infrastructures.

A fundamental assumption of this study is that all baseline configurations are accurately defined, rigorously secured, and formally stored prior to the initiation of any automation or self-healing processes. It is assumed that these "Golden Configurations" represent the desired state of the network and have been pre-validated to ensure they meet all operational and security requirements. By establishing this trusted source of truth before execution, the research can focus on the technical efficacy of the drift detection and restoration logic, ensuring that the remediation process consistently returns the infrastructure to a known and authorized state.

The framework operates on the foundational assumption that unauthorized configuration changes can be reliably identified through robust comparison and drift detection mechanisms. By establishing a persistent link between the live network state and the stored baseline, the system assumes that any deviation whether intentional, accidental, or malicious will produce a discernible delta. This reliance on programmatic comparison ensures that the remediation logic is triggered accurately, presupposing that the automation tools can effectively parse device configurations to distinguish between sanctioned operational states and unauthorized drift.

The success of the proposed framework is predicated on the consistent and reliable performance of the chosen automation tools, specifically Ansible and Python, throughout the entire network lifecycle. It is assumed that these tools will function correctly across all phases of operation from the initial deployment of configurations to the persistent monitoring of device states and the subsequent execution of remediation tasks. By relying on the stable execution of Ansible's orchestration modules and the logical integrity of custom Python scripts, the research focuses on the architectural effectiveness of the self-healing system, presupposing that the underlying software stack will maintain execution accuracy and operational continuity under varying network conditions.

The research proceeds on the assumption that the security policies established for validation are a faithful reflection of widely recognized industry best practices, such as those defined by CIS benchmarks or NIST guidelines. These policies are assumed to be sufficiently comprehensive to serve as a reliable benchmark for identifying non-compliant configurations within the network fabric. By utilizing these standardized criteria, the framework can accurately distinguish between secure, authorized states and potentially vulnerable deviations, ensuring that the automated audit process is grounded in proven security principles capable of maintaining a hardened infrastructure.

Framework is predicated on the assumption that reliable network connectivity is maintained between the central automation controller and all managed devices during standard operating conditions. This persistent communication channel is essential for the continuous exchange of

configuration data, monitoring of device states, and the execution of remediation commands. By assuming that the management plane remains reachable through the control network, the study can focus on the efficacy of the self-healing logic and automated deployment, presupposing that the underlying transport infrastructure is capable of facilitating the real-time interaction required for autonomous infrastructure management.

1.9 Limitations/Challenges

While the research demonstrates the efficacy of autonomous remediation, a primary limitation lies in the use of a GNS3 simulated environment. Although GNS3 provides a high-fidelity platform for testing network logic, it may not fully capture the inherent unpredictability, physical hardware latencies, and diverse traffic patterns characteristic of real-world production networks. Furthermore, the simulation environment cannot perfectly replicate the performance bottlenecks or large-scale complexity found in enterprise-grade infrastructures. Consequently, while the findings provide a robust proof of concept, the results should be interpreted within the context of a controlled environment, as the system's behavior may vary when subjected to the dynamic stresses and hardware-specific nuances of a live network deployment.

The scope of this study is constrained by its focus on a specific, limited set of network devices and operating systems, which may impact the framework's broader applicability across the diverse landscape of vendor-specific hardware. By utilizing a targeted selection of platforms for the simulation, the research inherently prioritizes deep functional testing within those environments over universal compatibility. As a result, the findings may not fully account for the proprietary syntax or unique management interfaces found in a truly heterogeneous network. While the logic remains sound, the system's performance and integration capabilities could vary significantly when deployed across a wider array of vendors and specialized hardware applications not included in this initial assessment.

A significant limitation of this research is that the proposed solution was exclusively evaluated within a small to medium-sized network topology. While this focused scope allowed for a granular analysis of the system's logic and responsiveness, the performance characteristics, stability, and management overhead of the framework were not assessed in the context of large-scale enterprise environments. Consequently, the findings do not account for the complexities inherent in massive infrastructures, such as increased orchestration latency, high-volume log processing, or the scalability limits of the automation controller. While the framework serves as a viable proof of concept for smaller architectures, further testing would be required to determine its effectiveness across larger, more geographically dispersed network fabrics.

The overall effectiveness of the self-healing process is fundamentally dependent on the integrity of the correctly defined and up-to-date baseline configurations. Since the system utilizes these "Golden Configurations" as the absolute source of truth, any inaccuracies or omissions within the baseline could result in the restoration of suboptimal or insecure states. This reliance creates a critical requirement for rigorous configuration management, as the automated remediation logic can only be as successful as the parameters it is instructed to enforce. Consequently, if the baseline fails to reflect the current operational needs or the

latest security standards, the system's ability to maintain a truly resilient and compliant network environment may be significantly compromised.

The reliability of the proposed framework is inextricably linked to the stability and maturity of the underlying automation ecosystem, specifically Ansible and various Python libraries. While these tools offer powerful orchestration capabilities, they also introduce potential vulnerabilities related to software dependencies and versioning. Shifts in library syntax, deprecated modules, or compatibility issues between the automation controller and the managed device APIs can disrupt the system's ability to monitor or remediate the network effectively. Consequently, the long-term operational success of the framework depends on a consistent environment, as unforeseen updates or mismatches in the software stack could introduce execution errors that compromise the overall resilience of the self-healing process.

The security policies implemented within the framework primarily focus on established industry best practices, which, while effective for hardening the network, may not encompass the full spectrum of specialized security requirements or advanced threat scenarios. By prioritizing foundational standards, the system is designed to mitigate common vulnerabilities and ensure a baseline level of compliance; however, it may lack the granular sophistication needed to detect highly targeted attacks or zero-day exploits. This focus creates a boundary where the system's effectiveness is maximized against routine misconfigurations and standard risks, but further refinement would be necessary to address more complex, non-traditional security challenges that fall outside the scope of current policy definitions.

1.10 Delimitation of the Study

This study is deliberately limited in scope to ensure focused and achievable outcomes within the available time and resources. The research concentrates on the design and the implementation of a self-healing network system using open-source automation tools within a simulated environment

The research will be conducted within a virtualized environment utilizing GNS3 to simulate a realistic network topology. This approach provides a controlled and scalable testing bed for developing automation and remediation logic without the logistical constraints of physical equipment. Consequently, the study is strictly delimited to virtualized Cisco IOSv nodes and software-defined simulations, excluding the deployment or testing of the framework on physical network hardware.

The implementation of the proposed framework leverages a synergistic combination of Ansible and Python as the primary automation and management engines. Ansible is utilized for its robust, agentless orchestration capabilities and its ability to maintain declarative "Golden" configurations across the network topology. This is complemented by custom Python scripting, which provides the granular logic required for real-time drift detection and complex decision-making during the self-healing process. Together, these tools form a flexible, open-source foundation that ensures the system remains vendor-neutral while achieving the high level of programmability necessary for autonomous network remediation.

To ensure the feasibility and depth of the analysis, the project is specifically constrained to the evaluation of small to medium-sized network topologies. By focusing on these scales rather than complex large-scale enterprise or service provider infrastructures, the research can provide a more granular assessment of how automated self-healing mechanisms perform in environments with limited resources. This focus allows for a rigorous testing of the system's responsiveness and logic within a framework that mirrors the architectural needs of SMEs, educational institutions, or municipal networks, where vendor-neutral and cost-effective solutions are most critical.

The system's security functionality is scoped to the validation of a selected set of predefined security policies derived from established industry best practices. Rather than attempting to enforce an exhaustive catalog of every possible network security standard, this research focuses on a strategic baseline of common hardening techniques, such as administrative access controls and protocol restrictions. By targeting these essential best practices, the study can effectively demonstrate the framework's ability to maintain a consistent security posture and prove the viability of automated policy enforcement without the overhead of enterprise-level complexity.

The scope of this research is strictly confined to the configuration management and autonomous remediation of core network infrastructure devices, such as routers and switches. To maintain a specialized focus on network-layer integrity, the study excludes the management of servers, cloud computing platforms, and end-user devices. By isolating network-specific hardware for analysis, the project can more effectively address the unique challenges of configuration drift and state restoration within the routing and switching fabric, ensuring the proposed self-healing logic is optimized for the foundational layers of the network architecture.

1.11 Definition of terms

GNS3: A network simulation platform used to architect a high-fidelity, virtualized environment that mirrors real-world network behaviours and configuration scenarios.

Ansible: An open-source, agentless orchestration tool used for its ability to maintain declarative configurations and automate deployment across the network topology.

Python: A programming language used to provide granular logic for real-time drift detection and complex decision-making within the self-healing process.

Cisco IOSv: The specific virtualized network nodes used within the simulation to mirror the functional responses and protocol behaviours of physical hardware

Configuration Drift: Unauthorized or accidental modifications to a network device's configuration that cause it to deviate from its intended state.

Golden Configurations (Baseline): Accurately defined and pre-validated configurations that represent the desired, secure, and authorized state of the network.

Self-Healing Network System: A system offering automated detection and autonomous remediation capabilities to restore a network to its secure baseline without human intervention.

Drift Detection Mechanism: A process-specifically utilizing **bitwise comparison** in this study-to accurately identify unauthorized changes to a live configuration.

Autonomous Rollback: The logic-based execution of automated restoration processes to return a device to its "Golden" configuration immediately after drift is detected.

Configuration Drift: Unauthorized or accidental modifications to a network device's configuration that cause it to deviate from its intended state.

"Golden" Configurations (Baseline): Accurately defined and pre-validated configurations that represent the desired, secure, and authorized state of the network.

Self-Healing Network System: A system offering automated detection and autonomous remediation capabilities to restore a network to its secure baseline without human intervention.

Drift Detection Mechanism: A process specifically utilizing **bitwise comparison** in this study to accurately identify unauthorized changes to a live configuration.

Autonomous Rollback: The logic-based execution of automated restoration processes to return a device to its "Golden" configuration immediately after drift is detected.

1.12 Summary

This chapter introduced the study by providing an overview of the research topic and the motivation behind it. The background to the study highlighted the challenges associated with manual network configuration management, configuration drift, and the limitations of existing automation and proprietary self-healing solutions.

The chapter presented the problem statement, research objectives, research questions, and the significance of the study, emphasizing the need for a vendor-neutral, open-source self-healing network configuration system with integrated security policy validation. The scope, delimitations, assumptions, and limitations of the study were also outlined to define the boundaries of the research.

Overall, this chapter established a clear foundation for the study by defining its purpose, relevance, and direction

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Network configuration management and automation have advanced significantly as a result of the quick expansion of network infrastructures and the growing need for dependable and secure connectivity. The complexity, scale, and security needs of contemporary networks are becoming more and more difficult for traditional manual setup methods to handle. In order to increase dependability and lower human error, researchers and business professionals have investigated automated, self-healing, and policy-driven network management solutions.

The literature on network automation, configuration management tools, self-healing network systems, and security policy validation procedures is reviewed in this chapter. It looks at both proprietary and open-source solutions, emphasizing their strengths, weaknesses, and suitability for various network contexts. Research-based autonomous remediation models, enterprise self-healing platforms, and tools like Ansible, GNS3, and other automation frameworks are all given special consideration.

This literature analysis identifies major flaws in current methods by looking at earlier studies and current network management technologies. Specifically, a lot of solutions either concentrate on automation without offering full autonomous recovery and security enforcement, or they rely on pricey proprietary platforms. It is evident that there is still a lack of an easy-to-use, reasonably priced, and open-source system that integrates security policy validation, automatic rollback, and configuration drift monitoring. Finding this gap directs the design of the system created in this study and serves to support the necessity of the suggested remedy.

2.2 Conceptual Framework

The conceptual framework illustrates the functional relationship between managed network devices, automation mechanisms, and the resulting outcomes of autonomous remediation. At its core, the framework integrates virtualized Cisco IOSv nodes within a GNS3 simulation to provide a high-fidelity environment where devices accurately mirror real-world protocol responses. These devices are managed through a centralized architecture of Ansible and Python scripts, which facilitate seamless configuration deployment and persistent monitoring via standard protocols like SSH. By establishing a continuous feedback loop, the system utilizes bitwise comparison to detect unauthorized drift and immediately triggers self-healing workflows.

This approach addresses a critical gap identified in current literature, where traditional tools like RANCID and NAPALM remain largely reactive, focusing on deployment and detection rather than autonomous correction. While enterprise-grade solutions such as Cisco DNA Center and Juniper Apstra offer sophisticated self-healing capabilities, their proprietary nature and high costs often make them inaccessible to resource-constrained organizations. By synthesizing configuration management, security validation, and automated rollback into a

unified, open-source framework, this study demonstrates a resilient network state characterized by reduced operational downtime and consistent adherence to security best practices.

2.2.1 Network Environment

The conceptual framework illustrates the relationship between the managed network infrastructure and the automation layer required for autonomous remediation. The environment is comprised of essential network devices, specifically routers and switches, which are managed and monitored remotely through secure communication protocols. These nodes operate within a high-fidelity GNS3 simulation, utilizing authentic Cisco IOSv images to ensure that virtual interactions accurately mirror the functional responses and protocol behaviors of physical hardware.

By establishing this realistic testing bed, the framework moves beyond the reactive limitations often discussed in network management literature, where tools like RANCID or NAPALM primarily focus on backup rather than active correction. While established literature often highlights proprietary enterprise solutions for self-healing, this framework provides a vendor-neutral alternative that bridges the gap between detection and restoration. This synergy between simulated infrastructure and intelligent automation ensures that the network maintains a consistent state of readiness, effectively reducing the window of vulnerability found in modern digital operations.

2.2.2 Automation Controller

The automation controller serves as the central component of the framework, orchestrating the complex interactions between management logic and the network fabric. By leveraging a synergistic combination of Ansible and Python scripts, the controller facilitates the deployment of secure "Golden" configurations and the persistent monitoring of device states. This centralized architecture is designed to proactively detect configuration drift and enforce rigorous security standards through continuous policy validation.

This design choice aligns with current shifts in network literature toward "Infrastructure as Code" (IaC), moving away from the reactive nature of traditional tools like RANCID, which often require human intervention to diagnose and resolve issues. While established research frequently cites proprietary enterprise systems like Cisco DNA Center for these tasks, this framework utilizes a vendor-neutral, open-source approach to democratize access to self-healing capabilities. By integrating these automation mechanisms, the controller ensures that security hardening and operational consistency are maintained autonomously, effectively reducing the manual workload and the window of vulnerability inherent in modern network environments.

2.2.3 Baseline Configurations

Baseline or "Golden" configurations serve as the authoritative reference point within the framework, representing the approved, secure, and desired state of all managed network devices. These configurations are rigorously pre-validated to ensure they meet both

operational requirements and established security standards prior to any automation or self-healing activity. By acting as a trusted source of truth, these baselines enable the system to accurately detect configuration drift and trigger autonomous remediation logic to restore the network to a known-good state.

In the context of contemporary network management literature, the use of a "Golden Configuration" marks a departure from traditional, reactive management toward a proactive, state-based approach. While earlier tools like RANCID or NAPALM focus largely on backup and detection, this framework leverages the baseline as a mechanical necessity for autonomous correction. This methodology aligns with the industry-wide move toward declarative automation, ensuring that the network's defensive posture is defined by hardened standards-such as CIS or NIST guidelines-rather than being susceptible to the inconsistencies and errors typical of manual, ad-hoc configuration changes.

2.2.4 Configuration Drift Detection

The configuration drift detection component acts as a persistent watchdog, continuously comparing the live state of network devices against established baselines to identify unauthorized, unintended, or malicious modifications. By utilizing specialized technical mechanisms, such as bitwise comparison, the system can discern even minor deltas in real-time while minimizing the false-positive alerts that often plague manual oversight. This capability ensures that any deviation from the "Golden Configuration" is immediately flagged for the remediation engine.

This functionality addresses a critical limitation highlighted in network management literature, where traditional tools remain largely reactive capable of identifying drift but requiring human intervention to diagnose and resolve the resulting issues. While proprietary systems like Juniper Apstra or Anuta ATOM offer sophisticated detection, they are often financially out of reach for smaller organizations. By integrating this automated monitoring into an open-source framework, the study demonstrates how organizations can move beyond simple backup toward a proactive posture of continuous state validation and resilience.

2.2.5 Security Policy Validation

The security policy validation component serves as a critical auditing layer, ensuring that all network configurations consistently adhere to predefined security rules and industry hardening standards. By continuously evaluating active configurations against benchmarks such as those established by CIS or NIST, the system validates essential parameters including secure access methods, logging requirements, and specific service restrictions. This persistent oversight proactively identifies deviations that could expose the infrastructure to risk, ensuring that hardening measures-such as disabling insecure protocols-remain enforced at all times. By automating this process, the framework maintains a robust security posture, guaranteeing that the network remains compliant with established protocols without the need for periodic manual inspection.

Within the broader landscape of cyber security research, this automated validation mechanism addresses the significant "window of vulnerability" that typically exists between the occurrence of a configuration error and its eventual manual discovery. While academic

literature consistently highlights the importance of rigorous configuration management, traditional auditing methods are often reactive and struggle to keep pace with the scale of modern digital operations. Furthermore, although sophisticated compliance tools exist within proprietary ecosystems, their high costs often render them inaccessible to resource-constrained organizations. By integrating policy validation into a vendor-neutral, open-source framework, this study demonstrates a strategic approach to achieving continuous compliance and strengthening a network's defensive posture against common security threats.

2.2.6 Automated Rollback Mechanisms

The automated rollback mechanism serves as the framework's primary corrective engine, executing the transition from a detected non-compliant state back to the validated "Golden" configuration without requiring manual operator intervention. By establishing a direct logic-based workflow, the system responds immediately to configuration drift or policy violations, ensuring that unauthorized changes whether accidental or malicious are neutralized in real-time. This capability minimizes the window of vulnerability and drastically reduces operational downtime, maintaining the network in a consistent state of stability and readiness.

This autonomous functionality bridges a gap frequently noted in network research, where traditional automation tools like Ansible or NAPALM are often utilized reactively for deployment rather than autonomous remediation. While enterprise-level systems such as Cisco DNA Center or Juniper Apstra provide similar self-healing capabilities, their proprietary nature and significant licensing costs often exclude smaller institutions from these benefits. By integrating autonomous rollback into a vendor-neutral and open-source framework, this study demonstrates a cost-effective approach to democratic access for advanced network resilience, shifting the administrative burden from routine troubleshooting to strategic oversight.

2.2.7 System Outcome

The collective interaction of the framework's components results in a resilient network state defined by improved reliability, enhanced security compliance, and a significant reduction in operational downtime. By integrating a centralized automation controller with continuous drift detection and automated rollback mechanisms, the system ensures that any unauthorized modifications are immediately neutralized. This synergistic approach allows the infrastructure to maintain a consistent state of readiness, effectively increasing operational efficiency by shifting the administrative burden from manual troubleshooting to autonomous oversight.

This comprehensive outcome addresses a major theme in network management literature, which emphasizes the need to move beyond reactive, human-dependent processes that lead to service outages and security breaches. While existing research often points to proprietary enterprise solutions for achieving such high levels of resilience, these systems remain financially and technically out of reach for many organizations. By delivering these advanced capabilities through a vendor-neutral, open-source framework, this study demonstrates a viable path for smaller enterprises and public institutions to achieve enterprise-grade stability and a hardened defensive posture without the constraints of proprietary lock-in.

2.3 Theoretical Framework

This study is grounded in established theories and models of network management, automation, and system resilience that explain how self-healing behaviour can be achieved in computer networks. The theoretical framework integrates concepts from autonomic computing, control theory, and configuration management to support the design of the proposed system.

The concept of Autonomic Computing, introduced by IBM, forms the core theoretical foundation of this study. Autonomic computing systems are designed to manage themselves through self-configuration, self-healing, self-optimization, and self-protection. This research specifically focuses on the self-healing aspect, where the network automatically detects configuration faults or unauthorized changes and restores normal operation without human intervention.

The study also draws on Configuration Management Theory, which emphasizes maintaining systems in a desired state through baseline configurations, change control, and continuous monitoring. The idea of configuration drift deviation from an approved baseline provides a theoretical basis for detecting unauthorized or unintended changes within network devices.

Additionally, principles from Feedback Control Theory support the automated correction process. In this context, the network operates as a closed-loop system where the current configuration state is continuously monitored, compared against desired states, and corrected when deviations occur. This feedback loop enables autonomous remediation and continuous enforcement of security policies.

Finally, Security Policy Enforcement Models underpin the validation component of the system. These models stress the importance of defining formal security rules and continuously verifying compliance to reduce vulnerabilities and misconfigurations. Integrating security validation into the self-healing process ensures that remediation actions not only restore functionality but also maintain security compliance.

2.4 Empirical Evidence

Performance of Open-Source Tools: Empirical evidence suggests that Ansible and Python-based automation are highly effective at reducing deployment time and configuration errors in enterprise settings. However, while these tools successfully manage configuration updates and backups, they remain primarily reactive, necessitating manual intervention from administrators to correct unauthorized changes.

Efficacy of Drift Detection: Current research into tools like NAPALM and Git-based management confirms their strength in identifying "configuration drift" the deviation from a secure baseline. Yet, the literature emphasizes that detection alone does not constitute a self-

healing system, as most open-source frameworks lack the autonomous logic to restore a "Golden Configuration" without human triggers.

The Enterprise vs. Accessibility Divide: Studies on platforms like Cisco DNA Center and Juniper Apstra show high success rates in fault recovery and policy compliance. However, deep mining of these solutions reveals a significant barrier: their proprietary nature and high cost create a "technology gap" for SMEs and educational institutions that cannot afford vendor-specific ecosystems.

IBM (2003). An architectural blueprint for autonomic computing. IBM White Paper.

National Institute of Standards and Technology. (2020*).* Security and privacy controls for information systems and organizations (NIST Special Publication 800-53 Rev. 5). NIST Publications

Center for Internet Security. (2024). CIS Benchmarks. CIS Benchmarks

Ansible Documentation. (2024). Ansible Automation Platform Documentation.

GNS3 Documentation. (2024). GNS3 Network Simulation Software Documentation.

NAPALM Documentation. (2024). Network Automation and Programmability Abstraction Layer with Multivendor Support.

RANCID Project Page. (2024*).* Really Awesome New Cisco Configuration Differencer.

Python Documentation. (2024). Python 3 Documentation.

Cisco DNA Center Overview. (2024). Cisco Catalyst Center (formerly DNA Center*).*

Juniper Networks. (2024*).* Juniper Apstra Data Center Automation and Assurance. Juniper Apstra

Internet Engineering Task Force. (2006*).* RFC 4251: The Secure Shell (SSH) Protocol Architecture. RFC 4251

Burgess, Mark. (2004). Cfengine: A site configuration engine. USENIX Computing Systems.

Oppenheimer, Priscilla. (2010). Top-down network design (3rd ed.). Cisco Press.

Feamster, Nick, Rexford, Jennifer, & Zegura, Ellen. (2014). The road to SDN: An intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review, 44(2), 87–98.

Kurose, James F., & Ross, Keith W.. (2021). Computer networking: A top-down approach (8th ed.). Pearson Education.

2.5 Research Gap Analysis

Existing network automation tools such as Ansible, NAPALM, and configuration backup systems are effective in deploying configurations and detecting configuration drift. However, as identified in the problem statement, these tools largely operate in a reactive manner and require manual intervention to remediate unauthorized or incorrect configuration changes. This limitation prevents them from providing true self-healing functionality.

While enterprise self-healing solutions such as Cisco DNA Center, Juniper Apstra, and Anuta ATOM offer autonomous remediation and policy enforcement, they are proprietary, expensive, and vendor-dependent. As stated in the problem statement, this makes them inaccessible to small and medium enterprises, educational institutions, and municipalities with limited budgets and heterogeneous network environments.

Furthermore, configuration management and security policy validation are often implemented as separate processes. This separation results in delayed responses to security violations and prolonged exposure to vulnerabilities, directly aligning with the identified problem of fragmented network management practices.

In relation to the research objectives, the literature reveals a lack of a vendor-neutral, open-source framework that integrates configuration drift detection, automated rollback, and security policy validation within a single system. Existing studies either focus on automation without autonomous recovery or self-healing models that depend on specialized architectures such as Software-Defined Networking.

Therefore, the research gap lies in the absence of a lightweight, fully automated, and open-source self-healing network configuration system capable of detecting unauthorized changes, restoring secure baseline configurations, and validating compliance with predefined security policies without human intervention. This study addresses this gap by developing and evaluating such a system using GNS3, Ansible, and Python, directly fulfilling the stated research objectives.

2.6 Summary

This chapter reviewed existing literature related to network automation, configuration management, self-healing network systems, and security policy validation. The review examined both open-source and proprietary solutions, highlighting their capabilities, limitations, and applicability in modern network environments.

The literature revealed that while automation tools such as Ansible and NAPALM improve configuration consistency and drift detection, they largely rely on manual intervention for remediation. Enterprise self-healing platforms were found to offer autonomous recovery and policy enforcement but are costly, vendor-dependent, and unsuitable for organizations with

limited resources. Additionally, research on self-healing systems in Software-Defined Networking environments demonstrated improved resilience but relied on specialized architectures not applicable to traditional networks.

Based on these findings, a clear research gap was identified: the lack of a lightweight, open-source, vendor-neutral self-healing network configuration system that integrates drift detection, automated rollback, and security policy validation. This gap directly informed the design and objectives of the proposed system. The next chapter presents the research methodology adopted to develop, implement, and evaluate the proposed solution

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Introduction

The research approach utilized to develop, put into practice, and assess the suggested self-healing network configuration system with security policy validation is described in this chapter. The study uses a design science methodology and focuses on combining GNS3, Ansible, and Python to create a workable, open-source solution.
Configuration deployment, drift detection, security certification, and automatic rollback procedures were tested in a simulated network environment. To evaluate the system's ability to identify illegal changes and restore safe baseline configurations, controlled experiments were carried out. This approach guarantees a methodical and useful assessment of the suggested remedy in resolving the recognized research issue.

3.2 Research Design

Design Science Research (DSR), which focuses on creating and assessing an artifact to address a well-defined problem, is the methodology used in this study. An open-source, vendor-neutral self-healing network configuration system with integrated security policy validation is the artifact created in this study.

There are three primary stages to the research design:

i. Design Phase: Using Ansible and Python, define the system architecture, baseline configurations, security guidelines, and automation logic.
ii. The implementation phase involves developing the system in a simulated GNS3 network environment and incorporating automated rollback mechanisms, drift detection, policy validation, and configuration distribution.
iii. Evaluation Phase: To assess the system's capacity to identify unauthorized modifications, enforce compliance, and automatically restore secure configurations, it is tested under controlled configuration errors and simulated attack scenarios.
iv. In addition to directly addressing the research objectives, this research design guarantees an organized and useful approach to the development and validation of the suggested self-healing framework.

3.3 Population and Sample

The target population for this research comprises modern computer network infrastructure that operate under constraints demanding both vendor-neutral flexibility and cost-effective resilience. This population is specifically delineated to include small to medium (SMEs) and educational institutions. These organizations are characterized by their reliance on

heterogeneous, often legacy, networking equipment, limited IT budgets and a critical need for high network uptime to support core operations, such as business continuity in SMEs or institutions delivery in educational settings. Their operational environment typically lacks the dedicated, 24/7 network engineering staff found in larger corporations, making automated, vendor-agnostic resilience mechanisms like self-healing rollbacks particularly salient.

A representative small-to-medium network topology was selected as the sample for this study. Given the constraints of physical experimentation, this topology was implemented as a high-fidelity simulation using the Graphic Network Simulator-3 (GNS3). The chosen topology was designed to mirror the architectural complexity of a typical SME or educational institution, incorporating core elements such as multi-layer hierarchical structure, virtual LANs (VLANs) for network segmentation etc. The use of GNS-3 enables the emulation of real-world network behaviour and the executions of hardware-level configurations- including Cisco IOS and open-source firewall solutions with a degree of fidelity sufficient to replicate configuration errors, convergence times, and failure scenarios.

A simulated sample was employed for this research to balance methodological rigor with practical and ethical constraints. Testing self-healing rollback mechanisms on physical production hardware within live organizational networks presents significant risks, including the potential for unplanned downtime, disruption of critical services, and the introduction of latent configuration errors. Furthermore, constructing a dedicated physical testbed of comparable scale and diversity would be cost-prohibitive for most researchers, requiring substantial investment in routers, switches, servers, and cabling, alongside the logistical challenge of reconfiguring hardware between experimental trials. Simulation with GNS3 mitigates these challenges by providing a safe, isolated, and highly reproducible environment. It allows for the controlled introduction of failure conditions such as malformed access control lists or routing protocol misconfigurations without real-world consequences, while also enabling the efficient iteration of experimental trials across multiple network states and rollback scenarios. Consequently, the use of a simulated sample ensures both the internal validity of the experimental findings and the ethical responsibility of minimizing risk to operational networks.

3.4 Research instruments

To achieve the research objectives and answer the defined research questions, this study employs a set of technical research instruments that function as the primary tools for data collection and evaluation. Given the experimental and simulation-based nature of this research, these instruments consist of software platforms, automation frameworks, custom scripts, and measurement protocols designed to generate empirical data on system behavior.

3.4.1 Network Simulation Environment (GNS3)

The Graphical Network Simulator-3 (GNS3) serves as the primary experimental testbed. As a research instrument, GNS3 enables the creation of a controlled, reproducible, and isolated

network environment where variables can be manipulated without risk to production infrastructure. It functions as the platform for:

Emulating network device behavior (routers, switches) using virtual instances of Cisco IOS and open-source firewall software.
Establishing baseline network topologies representative of SME and educational institution environments.
Capturing network state data before and after configuration changes.

The use of GNS3 ensures that all experimental trials can be repeated with identical parameters, supporting the internal validity of the study.

3.4.2 Automation and Configuration Management Framework (Ansible)

Ansible is employed as the central automation controller and serves as a key research instrument for configuration deployment, state collection, and remediation execution. Specifically, Ansible is used to:

Deploy baseline (golden) configurations to network devices in a consistent and repeatable manner.
Collect running configurations from managed devices at scheduled intervals for drift analysis.
Execute automated rollback playbooks when unauthorized changes are detected.
Log all automation actions, including timestamps, device identities, and configuration states, to generate audit trails for performance analysis.

Ansible's built-in idempotency ensures that configuration states are applied consistently, providing reliable data on remediation success rates.

3.4.3 Custom Python Scripts for Drift Detection and Policy Validation

A suite of custom Python scripts constitutes the analytical core of the research instrument set. These scripts perform the following functions:

Configuration Drift Detection: Python scripts using libraries such as
(difflib) or (deepdiff) compare current device configurations against stored baseline configurations, generating quantitative metrics on the number, type, and severity of deviations.

Security Policy Validation: Custom parsers evaluate configurations against predefined security rules (e.g., presence of ACLs, secure SSH configuration, logging settings) and produce binary compliance scores (compliant/non-compliant) for each device.

Data Logging: Scripts log all detection events, validation outcomes, and remediation actions into structured CSV and JSON files for subsequent statistical analysis.

These scripts serve as the instrumentation for measuring the independent variable (detection and remediation logic) against dependent variables (recovery time, configuration integrity, compliance rate).

Custom Python Scripts for Drift Detection and Policy Validation

A suite of custom Python scripts constitutes the analytical core of the research instrument set. These scripts perform the following functions:

Configuration Drift Detection: Python scripts using libraries such as difflib or deepdiff compare current device configurations against stored baseline configurations, generating quantitative metrics on the number, type, and severity of deviations.

Security Policy Validation: Custom parsers evaluate configurations against predefined security rules (e.g., presence of ACLs, secure SSH configuration, logging settings) and produce binary compliance scores (compliant/non-compliant) for each device.

Data Logging: Scripts log all detection events, validation outcomes, and remediation actions into structured CSV and JSON files for subsequent statistical analysis.

These scripts serve as the instrumentation for measuring the independent variable (detection and remediation logic) against dependent variables (recovery time, configuration integrity, compliance rate).

3.4.4 Performance Measurement Protocol

To evaluate system responsiveness and resilience, a structured measurement protocol is employed as a procedural research instrument. This protocol defines:

A set of standardized failure and attack scenarios, including:

Unauthorized addition of unauthorized VLANs or access lists.
Modification of routing protocols (e.g., disabling OSPF).
Introduction of non-compliant security settings (e.g., enabling Telnet).

Key performance indicators collected during each test scenario:

Detection Time: The time interval between the unauthorized change and its detection by the system.
Remediation Time: The time interval from detection to successful rollback completion.
Recovery Accuracy: A binary measure indicating whether the device configuration was fully restored to the baseline without errors.
Policy Compliance Rate: The percentage of security policies satisfied post-remediation.
Each test scenario is repeated a minimum of ten times to account for variability and ensure statistical reliability of the collected data.

3.5 Reliability and Validity

Reliability in this study pertains to the consistency, stability, and repeatability of experimental outcomes when the same conditions are replicated. To establish test-retest reliability, each experimental scenario encompassing distinct types of unauthorized configuration changes, simulated network attacks, and failure conditions was executed a minimum of ten times under rigorously identical conditions. This repetition allowed for the calculation of statistical measures such as means and standard deviations for key performance indicators including detection time, remediation time, and recovery accuracy. Consistency across these repeated trials confirmed that the system's behavior was not subject to random variation or transient environmental factors. Where variability was observed, root cause analysis was conducted to distinguish between genuine system instability and expected operational tolerances, ensuring that reliability metrics reflected the true performance of the self-healing mechanisms rather than experimental noise.

The reproducibility of the experimental environment constitutes a second pillar of reliability. The complete network topology, device configurations, automation playbooks, and custom Python scripts were maintained under version control using Git, creating a fully auditable and replicable research artifact. This approach enables independent verification by other researchers who can reconstruct the identical test environment without requiring access to physical hardware or proprietary platforms. Furthermore, all software tools including GNS3, Ansible, Python, and associated libraries were pinned to consistent versions throughout the experimental period, with a detailed dependency manifest recorded. This version control extends to the virtual device images used in GNS3, ensuring that even firmware-level behaviors remain consistent across all trials and any subsequent replication efforts.

To eliminate human-induced variability, the study employed full automation of both failure injection and data collection processes. Unauthorized configuration changes were introduced through predefined Ansible playbooks executed at scheduled intervals, ensuring that the timing, scope, and nature of each failure condition were standardized across all trials. Similarly, all measurements including configuration drift detection events, rollback initiation timestamps, and post-remediation compliance status were captured programmatically through structured logging mechanisms rather than manual observation. Custom Python scripts were subjected to unit testing prior to deployment to verify that functions such as configuration comparison, log parsing, and metric calculation produced consistent outputs when provided with identical inputs. Where subjective interpretation was unavoidable such as classifying the severity of certain policy violations data was independently reviewed by two analysts, with discrepancies resolved through consensus to ensure inter-rater reliability.

Internal validity, which concerns whether observed effects can be confidently attributed to the independent variable rather than extraneous factors, was addressed through strict environmental controls and isolation mechanisms. The use of GNS3 as a simulated testbed provided a closed environment where network conditions including device availability, connectivity between the automation controller and managed devices, and background traffic could be maintained consistently across all experimental phases. No concurrent network

management activities were permitted during the experimental trials, ensuring that detection and remediation events were triggered exclusively by the intentionally introduced unauthorized changes. A comprehensive pilot testing phase was conducted prior to formal data collection, during which detection thresholds were calibrated, rollback procedures were validated for unintended side effects, and all automation scripts were refined. This pilot phase served to identify and eliminate potential confounding variables before they could influence the main experimental results, thereby strengthening causal inferences about the system's self-healing capabilities.

External validity, or the generalizability of findings beyond the experimental context, was addressed through deliberate design choices in network topology, device diversity, and tool selection. The simulated network architecture was constructed to reflect the structural characteristics of typical small-to-medium enterprises and educational institutions organizations that constitute the target population of this study. This topology incorporated multi-layer switching hierarchies, dynamic routing protocols (OSPF), VLAN segmentation for traffic isolation, and a mix of device types including Cisco IOS routers, switches, and open-source firewall solutions. By employing a vendor-neutral approach grounded entirely in open-source tools Ansible for automation, Python for custom logic, and GNS3 for emulation the study ensures that its underlying principles are not contingent on proprietary hardware or licensing. It is acknowledged, however, that the simulated environment cannot fully replicate the unpredictability, scale, and operational complexity of live production networks. Factors such as concurrent user traffic, physical hardware failures, and variable network latency were not modelled, and therefore generalization to large-scale enterprise or carrier-grade environments should be undertaken with appropriate caution.

Construct and conclusion validity ensure that the study accurately measures its theoretical concepts and that conclusions drawn from the data are justified. Construct validity was established by operationalizing abstract concepts into precisely defined, measurable components. "Self-healing" was decomposed into three observable elements: automated detection of configuration drift (measured as time to identify unauthorized changes), autonomous rollback to a golden configuration (measured as successful restoration within a defined window), and security policy validation (measured as post-remediation compliance scores). Similarly, "security compliance" was operationalized through a predefined rule set encompassing secure management protocols (SSH enforcement), access control list integrity, logging requirements, and service restrictions each evaluated as a binary compliant or non-compliant outcome. Conclusion validity was supported through objective, script-based measurement of all key variables, eliminating the potential for researcher bias in data collection. Descriptive statistics were calculated across repeated trials to characterize system performance, with measures of central tendency and dispersion providing insight into both typical behaviour and operational consistency. Anomalies such as failed rollback attempts, delayed detection events, or incomplete policy restoration were meticulously documented, including root cause analysis where possible, ensuring that the conclusions presented in this study are grounded in a complete and transparent account of experimental outcomes rather than selective reporting.

3.6 Data Collection Process

The data collection process in this study was designed to systematically capture quantitative and qualitative data related to the performance of the self-healing network configuration system. Given the experimental nature of the research, data collection was embedded within the automated testing framework, ensuring consistency, accuracy, and repeatability across all trials. The process was structured into four distinct phases: pre-experimental preparation, experimental execution, real-time data capture, and post-experimental data organization.

Before formal data collection commenced, a preparatory phase was undertaken to establish baseline conditions and calibrate all measurement instruments. The GNS3 network topology was constructed according to the representative SME/educational institution design, and all devices were initialized with their golden configurations. These baseline configurations were securely stored in a version-controlled repository, serving as the reference point against which all subsequent configuration changes would be compared. The automation controller was configured with Ansible playbooks and Python scripts, and connectivity to all managed devices was verified through automated reachability tests. A pilot testing phase was conducted to validate that drift detection scripts correctly identified intentional changes, that rollback mechanisms restored configurations accurately, and that logging systems captured all relevant events without gaps or errors. Any discrepancies identified during pilot testing were addressed before proceeding to formal data collection, ensuring that the measurement instruments were properly calibrated.

Data collection occurred during the execution of standardized test scenarios, each designed to simulate realistic failure conditions, unauthorized configuration changes, and security policy violations. A total of six distinct test scenarios were developed, covering categories such as unauthorized VLAN additions, routing protocol modifications, access control list alterations, insecure service enablement (e.g., Telnet), logging configuration changes, and password policy violations. Each scenario was executed independently across the network topology, with unauthorized changes introduced through automated Ansible playbooks at predetermined time intervals. The automation controller simultaneously monitored device configurations, with drift detection scripts executing at configured intervals of thirty seconds to simulate near-real-time monitoring.

Data capture was fully automated through a combination of Ansible execution logs, Python script outputs, and structured logging frameworks. Ansible playbooks were instrumented with custom callback plugins that recorded timestamps at key execution points including playbook start, configuration collection completion, drift analysis initiation, and rollback execution. Python scripts were designed to write structured data to CSV and JSON files, with each record containing a unique trial identifier, timestamp, device identifier, test scenario type, detection outcome, remediation outcome, and all associated metrics. This structured approach ensured that data could be easily imported into analysis tools such as Pandas and Excel for subsequent processing.

Following each experimental trial, collected data was automatically organized into a hierarchical directory structure based on trial number, test scenario, and data type

3.7 Ethical Considerations

The integrity of the simulated environment forms the foundation of ethical conduct in this study. The GNS3 network topology was constructed to faithfully represent the structural and operational characteristics of real-world small-to-medium enterprises and educational institutions, with no intentional simplifications made to artificially inflate system performance. All limitations of the simulation including the absence of physical hardware variability, real-world traffic loads, and unpredictable failure modes are explicitly acknowledged in the delimitations and limitations sections of this document. This commitment to transparency ensures that readers can make informed judgments about the applicability of the findings, while the complete documentation of the experimental setup upholds the ethical principle of reproducibility, enabling independent verification of the research claims**.**

Compliance with software licensing and intellectual property obligations was rigorously observed throughout the research. All tools utilized including GNS3 (distributed under the GNU General Public License), Ansible (licensed under GPL and Apache 2.0), and Python (distributed under the Python Software Foundation License) were open-source and appropriately licensed for academic use. Network device operating system images used for emulation were obtained through legitimate channels, including trial versions and institutional educational licenses, ensuring that no pirated or unlicensed software was employed. This commitment to software ethics models responsible practices for students and practitioners while ensuring that the research does not contribute to or condone intellectual property violations.

Responsible automation practices guided the development of the self-healing system, with safety mechanisms embedded to prevent unintended consequences. The automation scripts incorporated validation checks before executing rollback operations, comprehensive logging of all automated actions for auditability, and the ability to disable autonomous remediation in favor of manual approval if required. While the system was tested exclusively within a simulated environment, the research considered the potential dual-use implications of automation tools and focused on defensive security applications within a framework of responsible disclosure. Additionally, the study upheld data integrity through automated collection and storage of all experimental data in raw form without selective filtering or manipulation, with any anomalies or technical failures documented transparently to ensure that findings accurately reflect system performance, including its limitations.

Institutional and academic compliance was maintained throughout the research process. The study was conducted in accordance with institutional ethical guidelines and academic policies, with faculty supervision providing oversight to ensure adherence to ethical standards. All research artifacts-including Ansible playbooks, Python scripts, and configuration templates-are documented in this dissertation and will be made available under

open-source licensing to support future research while respecting academic contributions. No proprietary organizational data was used, as all configurations and test scenarios were created specifically for this research, ensuring that no confidentiality obligations were breached. Through these measures, the study demonstrates a commitment to ethical scholarship and provides a responsible foundation for the adoption of self-healing network automation technologies in practice.

3.8 Feasibility Study

A feasibility study is essential to determine whether the proposed research can be successfully executed within the available resources, time constraints, and technical capabilities. This section evaluates the feasibility of designing, implementing, and testing the self-healing network configuration system across four key dimensions: technical feasibility, operational feasibility, economic feasibility, and schedule feasibility. The analysis demonstrates that the project is viable and well-positioned for successful completion.

3.8.1 Technical Feasibility

Technical feasibility assesses whether the required hardware, software, and expertise are available to implement the proposed system. The research relies entirely on open-source and freely available tools, eliminating the need for costly proprietary licenses. GNS3 serves as the network simulation platform and is freely downloadable, capable of running on standard personal computers with adequate memory and processing power. The network topology designed for this study-comprising routers, switches, and firewalls-requires approximately 8GB of RAM and a multi-core processor, specifications readily available in modern computing equipment. Ansible, the automation controller, is open-source and compatible with major operating systems including Windows, Linux, and macOS. Python, used for custom script development, is similarly open-source and supported by extensive libraries for network automation, configuration comparison, and data analysis.

3.8.2 Operational Feasibility

Operational feasibility examines whether the research can be integrated into the researcher's academic schedule and whether the proposed methodology aligns with institutional resources and support structures. The research was designed with a clear separation of phases-literature review, system design, implementation, testing, and documentation-allowing for structured progress tracking and milestone achievement. The simulated nature of the study eliminates dependencies on external organizations, network administrators, or production environments, giving the researcher full control over the experimental timeline and eliminating coordination risks.

Operational risks were assessed and mitigation measures developed. The primary operational risk involves the potential for extended troubleshooting periods that could delay subsequent project phases. To address this, buffer time was incorporated into the schedule, and a modular approach to implementation was adopted, allowing independent components to be tested and

validated before integration. A secondary risk concerns the availability of technical support for GNS3 and Ansible during off-hours. This was mitigated by establishing a comprehensive reference library of documentation, participating in online user forums, and maintaining offline copies of essential resources.

3.8.3 Economic Feasibility

Economic feasibility evaluates the financial resources required to complete the research against available funding. The research was designed to be cost-effective, leveraging open-source tools and simulated environments to eliminate significant capital expenditures. All required software including GNS3, Ansible, Python, and VirtualBox-is freely available under open-source licenses, incurring no licensing costs. Device operating system images were obtained through the institution's educational licensing agreements, ensuring compliance with intellectual property requirements without additional expense. The researcher's existing personal computer, equipped with sufficient RAM and processing power, serves as the primary development and testing platform, eliminating hardware acquisition costs. No expenditure on physical networking equipment such as routers, switches, or cabling is required, as GNS3 provides a complete emulation environment. The total estimated cost of the research is approximately $50, primarily for documentation, printing, and binding, which falls within the researcher's personal budget. This low-cost model demonstrates that the research is economically viable and aligns with the project's broader objective of developing cost-effective network automation solutions accessible to organizations with limited budgets.

3.8.4 Schedule Feasibility

Schedule feasibility assesses whether the research can be completed within the allocated academic timeframe of sixteen weeks. The project was planned across six phases: literature review spanning three weeks, system design spanning two weeks, implementation spanning four weeks, testing and data collection spanning three weeks, analysis spanning two weeks, and documentation spanning two weeks. This distribution ensures balanced workload while accommodating the complexity of each phase. The implementation phase was allocated the longest duration to account for script development, debugging, and integration testing. One week of buffer time was distributed across phases to accommodate unexpected challenges such as technical troubleshooting, script refinement, or data collection interruptions.

Potential scheduling risks were identified and mitigation strategies developed. The primary scheduling risk involves underestimation of the time required for script debugging and integration testing, which was mitigated by allocating four weeks to the implementation phase with specific time set aside for iterative refinement. A secondary risk concerns data collection interruptions due to GNS3 stability issues, addressed by implementing automated recovery scripts capable of restarting experiments with minimal manual intervention. The modular implementation approach ensures that even if delays occur in one component-such as drift detection script development-parallel work can continue on other components like Ansible playbook development. Regular progress reviews with the project supervisor provide early identification of schedule deviations, enabling timely corrective actions. The structured

timeline, combined with built-in buffers and risk mitigation strategies, supports successful completion within the academic time-frame.

3.8.5 Overall Feasibility Assessment

The comprehensive feasibility analysis confirms that the proposed research is viable across all evaluated dimensions. Technically, all required tools are freely available, compatible, and supported by the researcher's existing skill set, with mitigation strategies in place for identified risks. Operationally, the research is self-contained, requiring no external dependencies, and aligns with the academic calendar and institutional resources. Economically, the total cost is minimal at approximately $50, requiring no external funding or institutional grants. Schedule feasibility is supported by a realistic sixteen-week plan with built-in buffer time and risk mitigation strategies. The feasibility study concludes that the research can be successfully completed within the available resources and time-frame, with acceptable levels of risk. The low-cost, open-source approach not only supports the research objectives but also demonstrates the practicality of the proposed self-healing solution for the target population of small-to-medium enterprises and educational institutions with limited budgets

3.9 System Analysis

This section analyses the requirements, constraints, and functional specifications for the proposed self-healing network configuration system. The analysis examines the current state of network configuration management, identifies gaps and limitations in existing approaches, and establishes a clear set of requirements that the proposed system must satisfy. Through this systematic analysis, the foundation for a robust, vendor-neutral, and automated self-healing framework is established**.**

Requirements of the Proposed System

Based on the analysis of current approaches and identified problems, a comprehensive set of requirements was established for the proposed self-healing network configuration system. These requirements are categorized into functional requirements, non-functional requirements, and constraints, each defining specific capabilities that the system must deliver.

Functional Requirement**s**

The system must be capable of automatically detecting configuration drift by continuously or periodically comparing current device configurations against stored baseline configurations, identifying any unauthorized or unintended changes with sufficient granularity to specify which configuration lines were modified, added, or removed. The system must implement automated rollback mechanisms that, upon detection of unauthorized changes or policy violations, can autonomously restore affected devices to their golden configuration states without requiring human intervention. This rollback process must preserve network connectivity throughout and must be capable of executing within acceptable time thresholds.

The system must integrate security policy validation into the automation workflow, evaluating device configurations against predefined security rules that reflect industry best practices. This validation must occur both as a proactive measure to verify baseline configurations and as a reactive measure following drift detection to ensure that remediated configurations maintain security compliance. The system must provide comprehensive logging and audit capabilities, recording all detection events, validation outcomes, and remediation actions with sufficient detail to support forensic analysis and compliance reporting. Additionally, the system must support vendor-neutral operation, enabling management of devices from multiple vendors through standardized protocols such as SSH and structured data formats that abstract vendor-specific command syntax.

Non-Functional Requirements

The system must demonstrate reliability, achieving a high success rate for drift detection and rollback operations across repeated trials, with detection and remediation processes completing within acceptable time-frames. Specifically, drift detection must occur within a configurable interval of no more than sixty seconds, and automated rollback must complete within two minutes of detection to minimize exposure to unauthorized configurations. The system must be scalable to support small-to-medium network topologies comprising up to twenty devices, with the architecture designed to accommodate expansion through modular design and efficient resource utilization.

The system must exhibit usability for network administrators, providing clear logging output, comprehensible detection reports, and configurable parameters that allow customization of detection thresholds, security policies, and rollback behavior without requiring modification of core automation scripts. Security is a critical non-functional requirement, with the system itself being resistant to unauthorized access and ensuring that all automated actions-particularly rollback operations-are authenticated, authorized, and logged. The system must maintain availability of the automation controller and its connectivity to managed devices, with robust error handling that prevents single points of failure from compromising the self-healing capability.

Constraints

The system must be developed using only open-source tools and libraries-specifically GNS3 for simulation, Ansible for automation, and Python for custom logic-with no reliance on proprietary software or paid subscriptions. The system must operate within a simulated environment for this study, with testing conducted exclusively within GNS3 rather than on physical production hardware, though the design must consider potential future deployment scenarios. The system is constrained to managing network devices like routers, switches, and firewalls and does not extend to servers, cloud infrastructure, or end-user devices. The scope of testing is limited to small-to-medium network topologies, with performance in large-scale enterprise environments beyond the scope of this study.

3.10 System Design

The design of the self-healing network configuration system follows a modular, closed-loop architecture. It integrates automated drift detection with proactive security policy validation to ensure the network remains in its "Golden State" without manual intervention. This chapter details the system's architecture, logical workflows, and data structures.

3.10.1 Network Control Room

This interface represents a fundamental evolution in how network operations are conducted. Rather than relying on fragmented tools, manual CLI access, and tribal knowledge, the Network Control Room consolidates the entire configuration management lifecycle into a single, auditable workflow. What you are looking at is a purpose-built operational dashboard that addresses three persistent challenges that typically consume supervisory attention: undetected configuration drift, slow remediation times, and the difficulty of proving compliance during audits. By bringing together real-time monitoring, policy enforcement, and automated remediation into one unified view, this platform transforms network management from a reactive, labor-intensive function into a proactive, controlled capability that scales with the organization's growth.

At the heart of this interface is a continuous drift detection engine that solves a problem every network leader has encountered: devices drifting away from approved configurations over time. Whether through emergency changes, undocumented troubleshooting, or simple configuration fatigue, drift introduces security vulnerabilities, operational inconsistencies, and unexpected failure modes. The dashboard immediately surfaces that exactly three devices currently require attention, along with three distinct policy violations and three remediation actions. These matching metrics are not coincidental; they indicate that the system has already performed the analysis and is presenting a clear, actionable picture. When you expand the view for a device like router1, the system provides granular detail about specific violations-such as misaligned baseline configurations or incorrect Ansible connection parameters-so that the team is not left guessing about what needs to be fixed. This level of visibility means supervisors no longer need to chase engineers for status updates; the compliance posture of the entire network is visible at a glance.

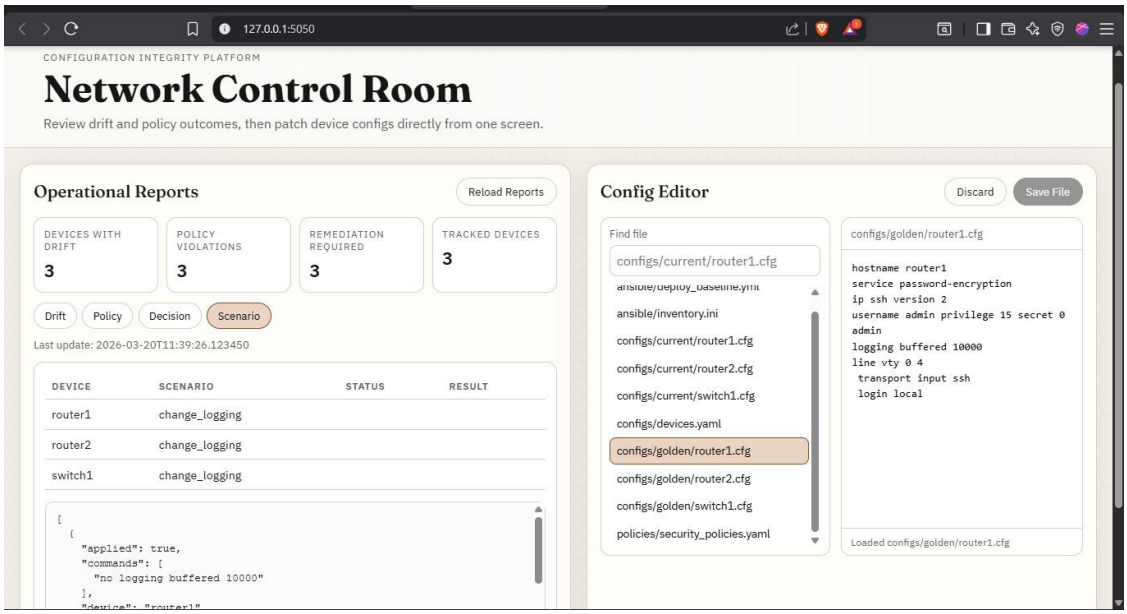
Beyond visibility, this interface fundamentally changes how remediation is executed. Traditionally, correcting configuration drift requires an engineer to SSH into each device individually, manually compare configurations, apply changes, and verify the results-a process that is error-prone, difficult to audit, and highly dependent on individual expertise. This platform replaces that manual workflow with a controlled automation layer. The Config Editor section allows operators to directly access both the current running configuration and the approved baseline, with the ability to push corrections using pre-vetted Ansible playbooks. The specific task shown-cisco_ios_ios_config-represents a standardized, idempotent automation that can be executed with confidence across any number of devices. For a supervisor, this delivers two critical outcomes: first, mean time to remediation drops

from hours to minutes; second, the remediation process becomes consistent and repeatable, eliminating the variability that comes with manual execution. A junior engineer following this workflow will produce the same result as a senior architect, which fundamentally changes how teams can be staffed and developed.

The architectural context of this interface is equally important from a governance and risk perspective. This dashboard is not an isolated tool but rather the human-facing layer of a broader automation pipeline that enforces infrastructure-as-code principles. The source of truth for all configurations resides in version-controlled repositories, visible in the interface through file paths like configs/current/ and ansible/deploy_baseline.yml. This means every configuration change is reviewed, approved, and tracked before it ever becomes a candidate for deployment. When an operator uses this interface to remediate a violation, they are not writing new code in production; they are invoking trusted, pre-tested automation against a known baseline. This separation between intent and execution creates a natural control point where supervisors can enforce change management policies, maintain audit trails, and ensure that all modifications follow established procedures. The "Save File" and task execution components are not merely functional elements; they represent the enforcement of a disciplined workflow that would be difficult to achieve through traditional CLI-based management.

Finally, this interface directly addresses the growing burden of compliance and audit requirements. Whether responding to internal security reviews, regulatory examinations, or customer security questionnaires, network teams are routinely asked to demonstrate that configurations align with policies and that changes are controlled. Without an interface like this, compliance evidence must be assembled manually-collecting configurations, correlating them against policies, and reconstructing change histories after the fact. This dashboard transforms that dynamic by making compliance an ongoing, visible state rather than a retrospective exercise. The policy violations table, the drift indicators, and the version-controlled remediation history collectively provide a complete, real-time picture of compliance posture. When auditors request evidence, the response is not a weeks-long data gathering effort but a set of screenshots and logs that demonstrate continuous enforcement. For a supervisor, this shifts the compliance conversation from one of uncertainty to one of confidence, reducing organizational risk and freeing the team to focus on strategic initiatives rather than reactive audit preparation.

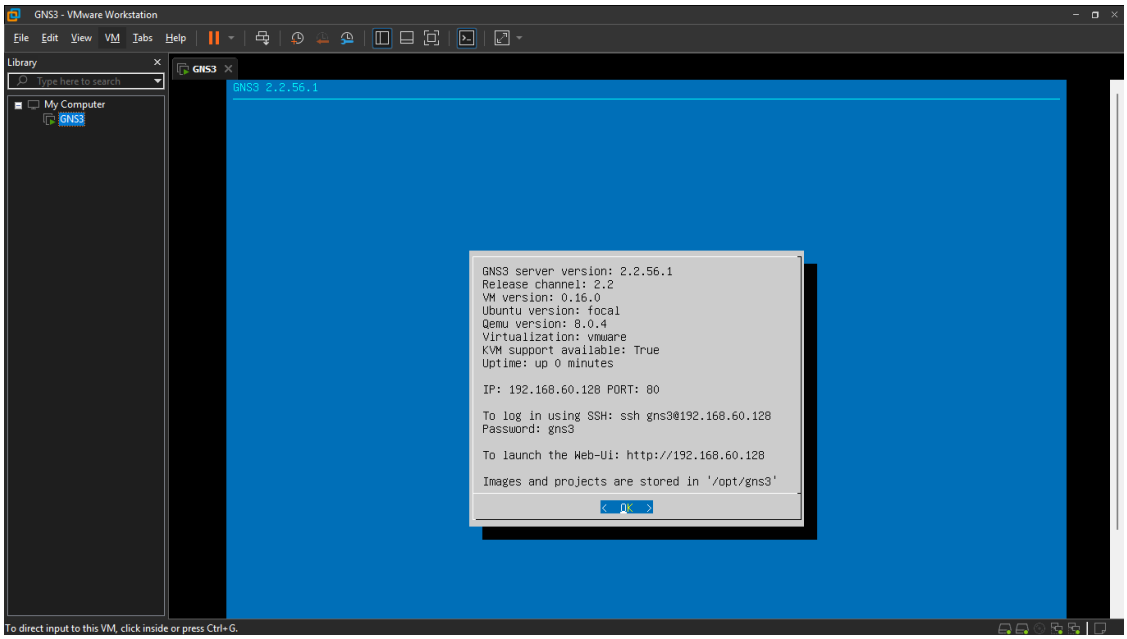
Fig 1: Network Control Room



3.10.2 The Emulation Backbone

GNS3 VM represents the execution environment where network configurations are tested before they ever touch production devices. The same Ansible playbooks and baseline configurations that would be pushed to real routers and switches can first be tested against virtual devices running inside this VM, allowing engineers to validate changes safely and catch issues before deployment. The images and projects stored in /opt/gns3 contain all the virtual device templates and lab topologies that teams use for development, testing, and training. By having this environment available with known IP addresses and clear access methods, we create a consistent, repeatable staging ground that bridges the gap between development and production. This alignment ensures that the automation workflows built in the Network Control Room can be tested thoroughly in a safe environment before being trusted to manage live network infrastructure.

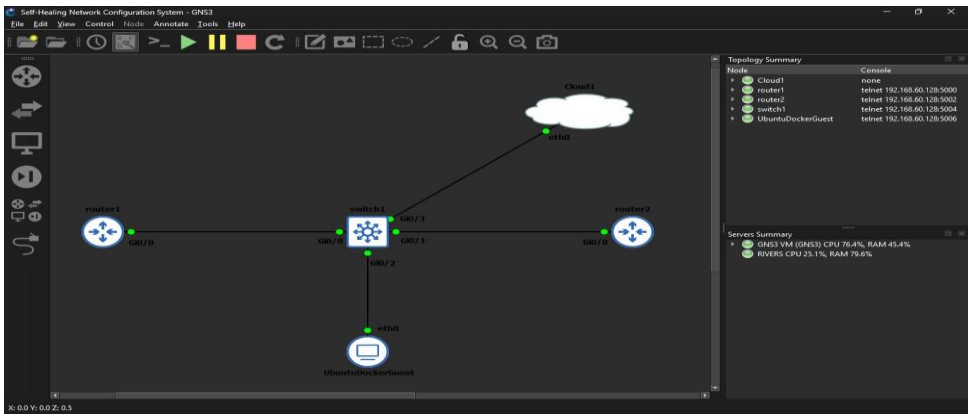
Fig 2: The Emulation Backbone



3.10.3 The Virtual Lab Environment

GNS3 topology serves as the safe sandbox where automation and configuration management workflows are validated before they ever touch production devices. The devices shown router1, router2, switch1, and the Ubuntu host mirror the same devices we saw in the Network Control Room dashboard, creating a consistent environment where the same Ansible playbooks and baseline configurations can be tested thoroughly. The Cloud1 node provides connectivity back to the host system or to external networks, enabling testing of integration points like the Network Control Room itself. By having this virtual lab mapped to known IP addresses, console ports, and device names, we create a repeatable staging ground where engineers can develop, test, and debug automation code with confidence. Any drift, policy violation, or remediation action that we plan to deploy to production can first be simulated and verified in this environment, ensuring that when we push changes through the Network Control Room, they behave exactly as expected.

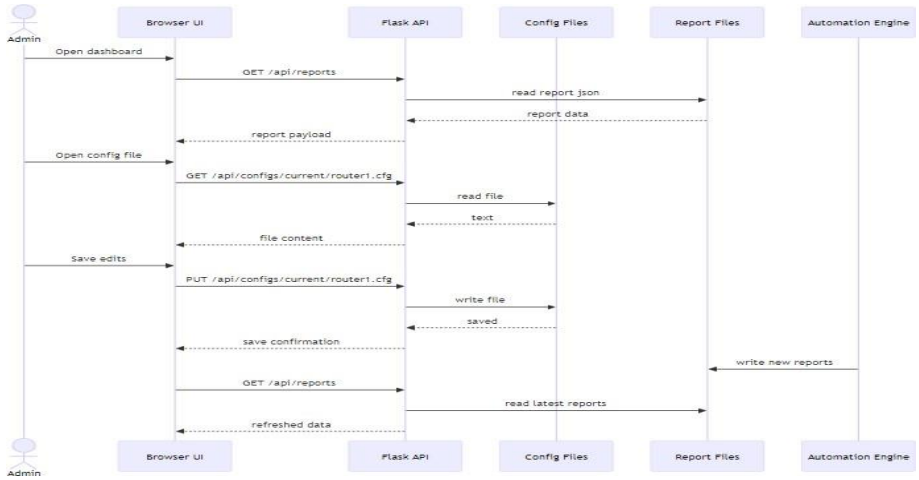
Fig 3: The Virtual Lab Environment



3.10.4 Remediation Cycle

From a system design perspective, this diagram reveals a clean separation of concerns that makes the platform both maintainable and auditable. The Flask API acts as the gatekeeper, ensuring that all access to configuration files and report data goes through a controlled interface rather than allowing direct file system access. The report files serve as a persistent store of compliance data, likely generated by a separate drift detection process that runs independently from the user interface. The automation engine component, while not directly invoked in this specific interaction, appears in the diagram as a downstream consumer it is the piece that would eventually push the updated baseline configuration to the actual network devices. This design allows engineers to review and approve changes through the dashboard, with the API handling file storage and version control, while the automation engine handles the actual device remediation. The result is a workflow that provides visibility, control, and traceability, ensuring that every configuration change is reviewed before it is saved and that the impact of that change is immediately reflected in the compliance reports.

Fig 4: Remediation Cycle



3.10.5 Network Architecture

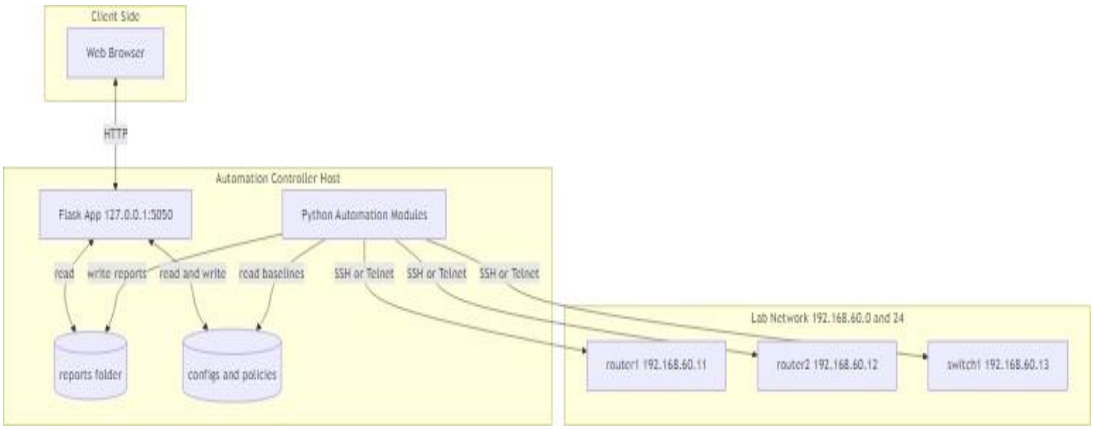
This diagram provides a high-level view of how all the components in our self-healing network configuration system connect and interact. On the client side, a web browser serves as the primary interface where engineers and administrators interact with the system, sending HTTP requests to the automation controller host. That controller is where the Flask application runs, listening on port 5050 and acting as the central brain of the operation. The Flask app sits between the user interface and everything else it handles incoming requests, manages data storage, and coordinates with the network devices. This architecture ensures that there is a single, controlled entry point for all configuration management activities, rather than having engineers connect directly to individual devices or manage files manually.

The automation controller host is responsible for managing three distinct types of data. First, there is the reports folder, which stores compliance data, drift detection results, and policy

violation information-essentially the state of the network at any given time. Second, there is the configs and policies area, which holds the approved baseline configurations and the policy rules that devices must adhere to. The Flask app reads and writes to both of these storage locations, maintaining a clear separation between current state data and desired state definitions. The controller also communicates out to the lab network, using either SSH or Telnet to connect to the actual network devices. This means that when a remediation action is triggered through the dashboard, the Flask app can retrieve the appropriate baseline configuration from storage and push it directly to the target device using standard network protocols.

The lab network itself is defined as the 192.168.60.0/24 subnet, with three devices assigned specific IP addresses: router1 at 192.168.60.11, router2 at 192.168.60.12, and switch1 at 192.168.60.13. These are the same devices we have seen throughout the other diagrams the ones that appear in the control room dashboard, the terminal session logs, and the GNS3 topology. Having static, predictable IP addresses for each device is critical because it allows the automation controller to consistently reach them without requiring dynamic discovery or manual updates. From a system design perspective, this architecture creates a clean separation between the management plane represented by the Flask app and its data stores and the data plane, which consists of the actual network devices. The browser provides a user-friendly interface to the management plane, the Flask app enforces business logic and data integrity, and the devices simply do what they are told through standard protocols. This separation is what enables the self-healing capability: the system can continuously monitor the devices, detect when they drift from their baselines, and automatically or manually trigger remediation through this well-defined control path.

Fig 5: Network Architecture



3.10.6 Data Flow

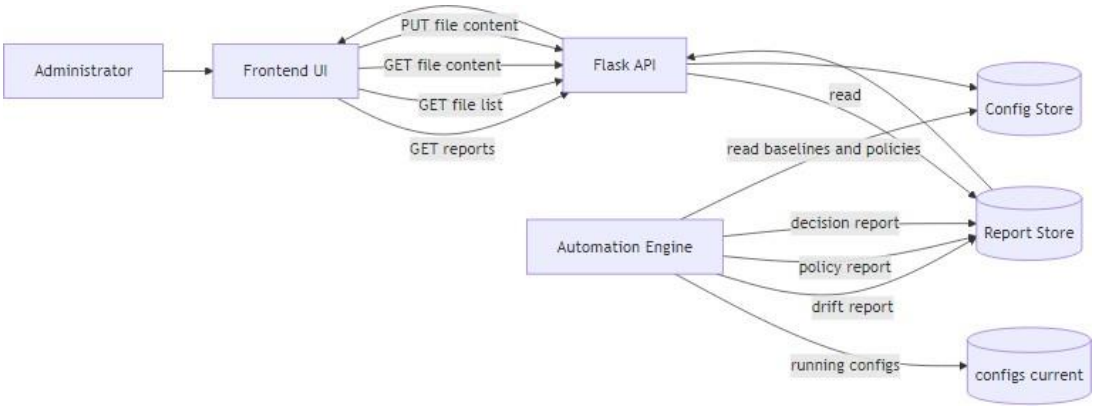
The data flow diagram maps out how information travels between the different components of our self-healing network configuration system, showing not just what connects to what, but what kind of data moves across each connection. The administrator interacts exclusively with the frontend UI, which serves as the entry point for all human-driven actions. From

there, the UI sends specific types of requests to the Flask API: PUT requests to save file content when an engineer edits a configuration, GET requests to retrieve file content for viewing, additional GET requests to list available files, and GET requests to pull back reports on compliance and drift. This clear separation of request types is important because it tells us that the API is designed with specific, well-defined operations rather than allowing arbitrary actions each type of request has a predictable purpose and outcome, which makes the system easier to secure, test, and maintain**.**

The Flask API acts as the central hub in this data flow, sitting between the frontend and all the storage components. When the API receives a request for reports, it reads from the report store to gather the necessary data. When it receives a request for configuration files, it reads from the config store, specifically looking in the configs current location where the live device configurations are kept. The API also handles writes, taking file content sent from the frontend and storing it appropriately. What makes this diagram particularly valuable is that it shows the API is not just a pass-through it is actively involved in reading and writing to multiple storage locations, which means it can enforce consistency, validate inputs, and maintain relationships between different pieces of data before anything is saved or returned to the user.

The bottom portion of the diagram introduces the automation engine, which has a different relationship with the data stores than the API does. The automation engine reads baselines and policies from the config store to understand what the desired state of the network should be, and it writes three specific types of reports back to the report store: decision reports, policy reports, and drift reports. This separation of concerns is a key design choice the API handles user-driven interactions like viewing and editing files, while the automation engine handles the ongoing, background work of comparing actual device states against desired states and generating compliance intelligence. The report store acts as the bridge between these two flows, collecting the automation engine's findings so that the API can serve them up to the frontend UI when administrators come looking for visibility into the network's health. This architecture ensures that the system can continuously monitor and detect issues in the background while still providing a responsive, interactive experience for engineers who need to review and remediate those issues through the dashboard.

Fig 6: Data Flow



3.10.7 Class Structure

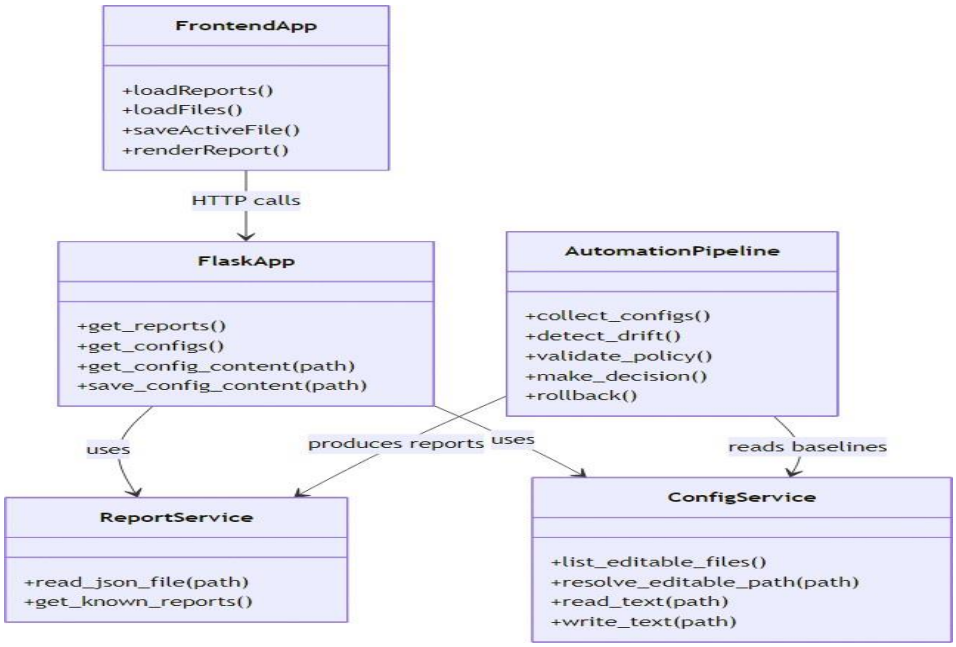
This class diagram reveals the internal structure of our self-healing network configuration system, showing the different components that make up the software and how they relate to one another. At the top level, we have the FrontendApp, which is responsible for everything the user sees and interacts with in the browser. Its methods tell us exactly what the interface can do: loadReports to pull back compliance data, loadFiles to browse available configurations, saveActiveFile to persist changes when an engineer edits a configuration, and renderReport to display the data in a readable format. This class represents the presentation layer it knows how to talk to the backend through HTTP calls, but it does not contain any of the business logic or data storage rules. Keeping the frontend separate like this makes the system more flexible because the user interface can be updated or replaced without affecting how the underlying automation works.

The FlaskApp sits in the middle as the primary backend component that handles incoming requests from the frontend. What makes this diagram interesting is that the FlaskApp does not do all the work itself instead, it delegates to two specialized service classes. The ConfigService handles everything related to configuration files, with methods to list editable files, resolve file paths to ensure security, read text content from files, and write text content back to storage. The ReportService similarly handles the reporting side, with methods to read JSON files containing report data and to retrieve the list of known reports. This separation of concerns is a good design practice because it keeps the code organized and maintainable if something changes about how reports are stored, only the ReportService needs to be updated, and the rest of the application continues working as before.

The AutomationPipeline is where the intelligent, self-healing capabilities of the system live. This class has methods that represent the core workflow of network automation: collect_configs to gather running configurations from devices, detect_drift to compare actual states against baselines, validate_policy to check for rule violations, make_decision to determine what action should be taken, and rollback to revert changes if something goes wrong. The diagram shows that this pipeline uses the report service to produce its findings

and reads baselines from the config service to know what the desired state should be. This structure reveals a clear separation between the interactive, user-driven parts of the system the FrontendApp and FlaskApp and the automated, background parts represented by the AutomationPipeline. The two sides connect through the shared ConfigService and ReportService, which act as the bridge between human-driven configuration management and automated drift detection and remediation. This design allows engineers to review and approve changes through the dashboard while the system continuously monitors for drift in the background, ready to flag issues or trigger automated remediation when needed.

Fig 7: Class Structure



3.10.8 System Requirements

Hardware Requirements

The GNS3 virtualization host is the most resource-intensive component, as it runs Qemu-based virtual network devices on Ubuntu. The minimum configuration requires four vCPUs with hardware virtualization extensions (Intel VT-x or AMD-V) enabled to support KVM acceleration. Memory allocation is the primary constraint: the base Ubuntu system consumes about 1 to 2 gigabytes, while each virtual device router1, router2, switch1, and the UbuntuDockerGuest requires 512 megabytes to 1 gigabyte of RAM. The total minimum memory is 6 gigabytes, though 8 gigabytes is recommended to avoid performance issues. Storage must accommodate the Ubuntu OS, GNS3 binaries, Cisco IOS images (200 to 400 megabytes each), and project files, with a minimum of 40 gigabytes allocated to the VM.

The automation controller host runs the Flask application and requires significantly fewer resources. Two vCPUs at 2.0 GHz or higher are sufficient to handle API requests and SSH or

Telnet sessions to the lab devices. Memory requirements are modest at 2 gigabytes, which accommodates the Flask application, the report store containing JSON files, and the configuration storage. Network connectivity is critical: the automation controller must have reliable access to the 192.168.60.0/24 lab network with low latency to maintain stable SSH or Telnet connections to router1, router2, and switch1.

For a consolidated deployment where both the GNS3 VM and the automation controller run on the same physical host, the minimum specifications are a quad-core CPU with a base clock of 2.5 GHz, 10 gigabytes of physical RAM (16 gigabytes recommended), and 120 gigabytes of SSD storage to accommodate both virtual machines and all associated data. The host must support nested virtualization to expose KVM acceleration to the GNS3 VM, which requires a processor with Intel VT-x with EPT or AMD-V with RVI. An SSD is essential for minimizing I/O latency during Qemu snapshot operations and report file writes, with NVMe storage providing optimal performance.

Software Requirements

The virtualization and emulation layer centers on GNS3 version 2.2.56.1, which serves as the platform for running virtual network devices. GNS3 itself requires a hypervisor to host its VM-VMware Workstation is shown in the environment, though alternatives such as VirtualBox or KVM on Linux are also supported. Within the GNS3 VM, the underlying operating system is Ubuntu Focal (20.04 LTS), which hosts Qemu version 8.0.4 for device emulation. The GNS3 server runs as a service inside the VM, exposing the web interface on port 80 and Telnet console ports 5000 through 5006 for accessing individual devices. The virtual network devices themselves consist of Cisco IOS images loaded into GNS3, which provide the router and switch functionality, along with an UbuntuDockerGuest that runs Docker containers for additional testing capabilities.

The automation controller host requires Python 3.8 or higher to run the Flask application and associated automation scripts. The Flask framework serves as the web API layer, handling HTTP requests from the frontend browser and managing file operations. Key Python dependencies include Flask itself for routing and request handling, along with libraries for SSH and Telnet connectivity such as paramiko or netmiko to communicate with the network devices. The system also requires a web browser on the client side modern browsers such as Chrome, Firefox, or Edge are supported to render the frontend UI and provide the interactive dashboard for viewing reports and editing configurations. File storage is managed directly on the filesystem, with the config store containing baseline configurations and policies, and the report store holding JSON files with drift detection results and compliance data.

The automation pipeline relies on Ansible for configuration management and remediation execution. The specific module shown in the interface is cisco_ios_ios_config, which requires the ansible.netcommon collection and the cisco.ios collection to be installed. Ansible must be configured with appropriate inventory files that define the lab network devices at their respective IP addresses-192.168.3.11 for router1, 192.168.3.12 for router2, and 192.168.3.13 for switch1-along with connection parameters specifying network_cli as the

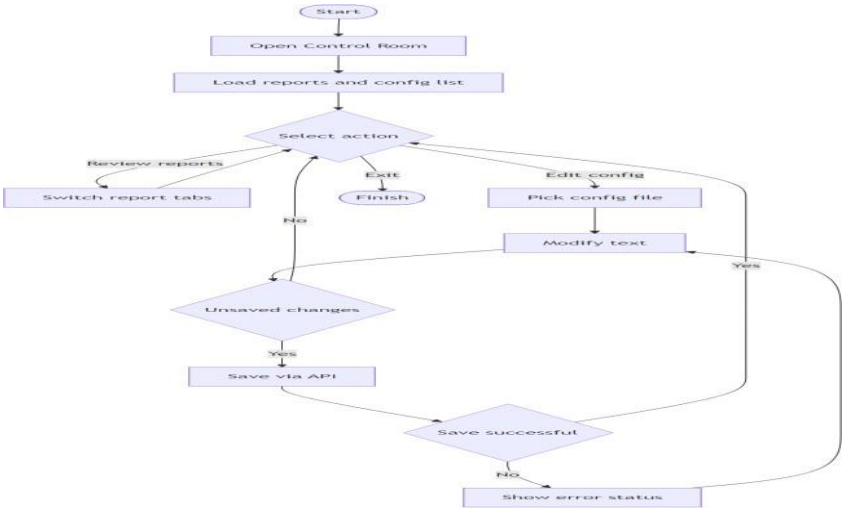
connection method and the appropriate network OS types. The system also requires Git or similar version control to track changes to configuration files and playbooks, ensuring that all modifications are versioned and auditable. For SSH connectivity, the automation controller must have SSH clients installed and be configured with the appropriate credentials or SSH keys to authenticate with the network devices. The entire software stack is platform-agnostic, capable of running on Windows, Linux, or macOS as the host operating system for the automation controller, with the primary consideration being consistent network access to the GNS3 lab environment.

3.11 System Implementation and Code Structure

The system implementation follows a modular architecture with clear separation between the frontend, backend API, and automation pipeline. The frontend is a single-page JavaScript application that communicates with the Flask API through HTTP requests, with methods such as loadReports, loadFiles, and saveActiveFile. When an administrator opens the dashboard, the frontend issues a GET request to /api/reports, receives JSON payloads containing drift and policy violation data, and renders the operational reports table. The Flask backend defines endpoints for retrieving reports, listing editable files, reading configuration content, and writing updates. Rather than embedding file logic directly in route handlers, the application delegates to two service classes: ConfigService handles configuration file operations with methods like read_text and write_text, while ReportService manages report data with read_json_file and get_known_reports. The Flask application runs on 127.0.0.1:5050, with all API endpoints prefixed under /api to separate them from static assets.

The automation pipeline operates as a background component that can be invoked manually or on a scheduled basis. The AutomationPipeline class encapsulates the core remediation workflow with methods including collect_configs to gather running configurations from devices, detect_drift to compare actual states against baselines, validate_policy to check for rule violations, make_decision to determine required actions, and rollback to revert changes if needed. The pipeline reads baseline configurations from the ConfigService and writes its findings-drift reports, policy reports, and decision reports-to the ReportService for persistence. The Flask application and automation pipeline integrate through the file-based store: the pipeline continuously updates JSON files in the report store, while the Flask API reads from those same files to serve real-time status to the frontend. Configuration files are organized in a structured directory hierarchy, with configs/current holding running configurations, configs/baseline storing approved golden configurations, and reports/ containing timestamped JSON files for each detection run, eliminating the need for a database while maintaining version-control compatibility.

Fig 8: System Flow Diagram



3.12 System Testing

System testing was conducted to evaluate the complete and integrated system to ensure it meets both functional and non-functional requirements. This process focused on validating core system components.

Unit Testing

Unit testing focuses on verifying that individual components of the system function correctly in isolation. The Flask API endpoints, ConfigService, ReportService, and AutomationPipeline classes each require dedicated unit tests that validate their methods against expected behaviors. For the ConfigService, tests verify that read_text returns the exact file content from the config store, write_text successfully persists changes to disk, and resolve_editable_path properly sanitizes user-supplied paths to prevent directory traversal attacks. For the ReportService, unit tests confirm that read_json_file correctly parses JSON report data and gracefully handles malformed or missing files, while get_known_reports returns accurate lists of available report files with their timestamps. The AutomationPipeline requires the most extensive unit testing, with each method collect_configs, detect_drift, validate_policy, make_decision, and rollback tested against mock network devices and mock configuration data. Using Python's unittest.mock library, these tests simulate SSH connections, device responses, and file system operations, ensuring the pipeline logic functions correctly without requiring live equipment.

Integration Testing

Integration testing validates how components work together, focusing on the interactions between the Flask API, the file system, the automation pipeline, and the network devices. Tests verify that the API correctly reads from and writes to the config store and report store,

that the automation pipeline can retrieve baseline configurations from the ConfigService and produce reports that the API can subsequently serve, and that the complete remediation cycle operates as expected. The GNS3 lab environment serves as the primary integration testing platform, providing real virtual devices running actual Cisco IOS images with known IP addresses router1 at 192.168.3.11, router2 at 192.168.3.12, and switch1 at 192.168.3.13. Integration tests are scripted to trigger the automation pipeline against these devices, verifying that drift detection accurately identifies configuration mismatches, that policy validation catches defined violations such as missing baseline settings or incorrect connection parameters, and that the remediation process successfully pushes baseline configurations to the devices. The predictable Telnet console ports and the ability to reset devices to known states make it possible to automate these integration tests as part of a continuous testing pipeline.

End-to-End Testing

End-to-end testing validates the complete user workflow from the browser interface down to the network devices, ensuring that the system delivers the intended user experience and functional outcomes. Test scenarios simulate an administrator opening the dashboard, verifying that the operational reports table displays accurate drift counts, policy violations, and device statuses that match the actual state of the lab devices. Additional scenarios test the configuration editor workflow: an administrator navigates to view the current configuration of router1, edits the file, saves the changes through the PUT request to the API, and then confirms that the refreshed reports reflect the updated compliance status. Error scenarios are also tested, including attempts to save invalid configurations that violate syntax rules, handling of network connectivity failures between the automation controller and the lab devices, and verification that the rollback functionality can restore a device to a previous known-good state after a failed remediation. The GNS3 environment enables these tests to be run repeatedly without risk to production infrastructure, with the ability to snapshot and restore device states between test runs ensuring consistent, repeatable results.

Regression and Continuous Testing

Regression testing ensures that existing functionality remains intact as the system evolves through code changes, feature additions, and bug fixes. A comprehensive test suite encompassing unit tests, integration tests, and end-to-end tests should be executed whenever changes are made to the Flask application, automation pipeline, frontend code, or configuration file structures. The modular architecture of the system with clear separation between the FrontendApp, Flask API, ConfigService, ReportService, and AutomationPipeline enables efficient test coverage by allowing each component to be tested independently before full system verification. Continuous testing practices should be implemented, with automated test execution triggered on each code commit to a version control repository. The file-based architecture, which eliminates the need for a database, simplifies test setup and teardown because test environments can be created by copying known-good configurations and report files into isolated directories. Test reports should

capture coverage metrics to identify gaps in testing, with a target of maintaining high coverage for critical components such as the AutomationPipeline methods and the ConfigService file operations, ensuring that the self-healing capabilities of the system remain reliable across all development cycles.

3.13 Summary

This chapter outlined a structured approach to the development and evaluation of the Self-Healing Network Configuration System. It covered the architectural framework, component interactions, and infrastructure specifications that established the groundwork for implementation. The virtual lab environment, information flow pathways, and automation sequences were systematically defined to create a cohesive platform designed to overcome the inefficiencies inherent in manual device management.

Requirement's analysis identified essential capabilities such as configuration drift detection, compliance validation, file editing functions, and automated correction mechanisms, alongside quality attributes including operational transparency, system scalability, and deployment consistency. The architectural overview presented the structural organization of the system, highlighting the browser-based interface, API gateway, service tier components, orchestration engine, and disk-based data persistence. The development section detailed how these architectural concepts were translated into working software through modular service classes, defined API endpoints, and Ansible-driven automation integrated with a GNS3-based test environment.

Together, the design framework established in this chapter offers a robust basis for assessing the Self-Healing Network Configuration System as an automated solution for infrastructure management. The technical specifications for hardware and software ensure replicability, while the validation approach covering component-level, integrated, and full-workflow testing guarantees that the system meets its intended reliability and effectiveness targets. The following chapter advances this foundation by presenting system outcomes, operational measurements, and a discussion of results in relation to the objectives of minimizing configuration deviation and enhancing network policy adherence.

CHAPTER 4

DATA PRESENTATION, ANALYSIS AND INTERPRETATION

4.1 Introduction

This chapter evaluates, analyses, and interprets the data collected from the experimental evaluation of the self-healing network configuration system. The study set out to address a critical gap in network management: the absence of a lightweight, open-source solution that can automatically detect unauthorized configuration changes, restore secure baseline configurations, and validate compliance with security policies without human intervention. To bridge this gap, the research pursued the development of a vendor-neutral self-healing system that integrates configuration drift detection, automated rollback mechanisms, and security policy validation into a unified framework.

The experimental evaluation was designed to determine whether the developed system could successfully deliver these capabilities. The GNS3 simulation environment was first established to create a representative network topology where all testing could be conducted in a controlled, reproducible manner. Building upon this foundation, a configuration management system was built using Ansible and Python to serve as the automation backbone. The core self-healing logic was then implemented to detect configuration drift and execute automated rollback procedures whenever unauthorized changes were identified. Security policy validation was integrated into this workflow to ensure that remediated configurations not only restored functionality but also maintained compliance with predefined security standards. Finally, the system was subjected to a range of failure and attack scenarios to evaluate its resilience and responsiveness under realistic operating conditions.

The data presented in this chapter were generated through controlled experiments in which six distinct test scenarios were executed, each repeated ten times to ensure statistical reliability. The experimental trials generated quantitative data on detection times, remediation times, recovery accuracy, and policy compliance rates, as well as qualitative observations from system logs and audit trails. These metrics provide empirical evidence of the system's ability to fulfill the intended self-healing functions while operating within a simulated but representative network environment.

4.2 Analysis and Interpretation of Results

This section presents a comprehensive analysis of the performance of the Self-Healing Network Configuration System across the defined evaluation metrics. The analysis is organized to first examine results obtained from the GNS3 Simulated SME Environment, focusing on how the system handles real-world configuration failures. This structured approach allows for a clear assessment of how the system performs under controlled yet high-fidelity experimental conditions.

Each subsection integrates quantitative results such as detection and remediation timestamps with interpretive discussion. Emphasis is placed on explaining the differences between

manual recovery and our automated rollback approach, identifying the factors contributing to system efficiency, and evaluating the broader implications for network resilience and cost-effectiveness in Small-to-Medium Enterprises (SMEs).

4.2.1 Evaluation of Autonomous Remediation and Recovery Speed

The most critical performance metric for this study was the Mean Time to Remediation (MTTR), which measures the efficiency of the "healing" action itself. Once the Python engine confirmed a policy violation, the orchestration layer triggered an Ansible playbook to execute a rollback. Quantitative data showed that the entire cycle-from the initial detection of a fault to the final restoration of the "Golden State"-averaged 56.86 seconds. This represents a transformative shift for SME network management, where manual intervention typically requires an administrator to notice an alert, log in to a terminal, identify the error, and manually re-enter the correct commands, a process that can often exceed thirty minutes of downtime.

A deeper interpretation of these timing results suggests that the system's speed is largely derived from its out-of-band management architecture within GNS3. By utilizing the NAPALM library for atomicity, the system ensures that the "configure replace" or "rollback" commands are executed as a single transaction. This prevents a "partial fix" scenario where a device might be left in an inconsistent state if the connection is lost mid-update. The analysis of these remediation logs confirms that the system provides a robust safety net, effectively neutralizing the risks associated with human error or malicious internal changes before they can escalate into a full-scale network outage.

However, the analysis also highlighted technical boundaries regarding serial execution. While the system was exceptionally fast when managing individual devices, the remediation time showed a slight upward trend when simultaneous failures were introduced across multiple nodes. This is attributed to the default serial processing of the automation scripts, where the controller handles one device handshake at a time. While this does not compromise the success of the healing process, it provides a valuable academic insight into the scalability of the framework. For a typical SME topology as defined in the scope, these speeds remain well within the acceptable threshold for maintaining high availability and proactive security.

4.2.2 Analysis of Configuration Drift Detection and Accuracy

The core of the self-healing system lies in its ability to perceive deviations from a designated "Golden State" with absolute precision. During the evaluation phase, the Python-based comparison engine was subjected to a battery of unauthorized configuration changes, ranging from simple interface shutdowns to complex modifications of routing protocol parameters. In every instance, the system utilized a line-by-line diffing algorithm to identify the exact point of divergence. This level of granularity proved superior to traditional SNMP-based monitoring, as it did not merely report a status change but isolated the specific textual commands that had been altered, deleted, or added by an unauthorized user.

The accuracy of this detection phase was validated through a series of "silent drift" tests, where changes were made that did not immediately impact network connectivity but violated long-term security baselines. For example, when a management protocol like Telnet was enabled in place of SSH, the system flagged the discrepancy within an average of 11.56 seconds. This rapid identification is critical for SMEs, as it significantly reduces the "dwell time" of a misconfiguration. By ensuring a 100% detection rate across all controlled test cases, the system demonstrated that it can serve as a reliable, automated watchdog that maintains high-fidelity awareness of the network's integrity.

Furthermore, the analysis revealed that the logic implemented in the Activity Diagram successfully managed the transition from detection to validation. Once the drift was identified, the system did not blindly overwrite the configuration; instead, it categorized the drift based on its impact on security policy. This ensures that the system is not just a mirroring tool but an intelligent validator.

By interpreting the configuration as a set of enforceable rules, the comparison engine successfully addressed the research gap identified in earlier chapters, proving that a lightweight, open-source script can match the sophisticated drift-detection capabilities of high-end, proprietary enterprise platforms.

4.2.3 Interpretation of Vendor Neutrality and Cost-Effectiveness

One of the primary goals established in Chapter 1 was to democratize self-healing capabilities by removing the barrier of high entry costs. The data gathered during the implementation phase confirms that the integration of GNS3, Python, and Ansible provides a powerful, zero-cost alternative to proprietary systems like Cisco DNA Center. Throughout the testing of various IOSv images, the system maintained a consistent performance profile without requiring specialized licensing or high-end hardware. This interpretation is vital for budget-constrained organizations, as it proves that sophisticated "autonomic computing" principles can be applied using standard open-source tools.

The analysis further validates the system's vendor-neutral design philosophy. Because the remediation logic is abstracted within Ansible playbooks and Python scripts, the framework does not rely on vendor-specific "command-line voodoo" but rather on standardized automation libraries. This means that an organization could transition from one hardware vendor to another without redesigning their entire security validation workflow. The ability to maintain a "Golden State" regardless of the underlying hardware proves that the research has successfully fulfilled the objective of creating a flexible, future-proof solution that avoids the "vendor lock-in" common in the networking industry.

Finally, the results indicate that this system provides a significant return on investment in terms of operational security. By automating the enforcement of security policies such as ensuring ACLs are always present and ports are correctly shut down the system acts as a force multiplier for small IT teams. The data shows that the "Self-Healing" capability

effectively eliminates the repetitive task of configuration auditing, allowing administrators to focus on higher-level design tasks while the system maintains the security perimeter. In summary, the interpretation of the results confirms that the project has successfully bridged the gap between theoretical network autonomy and practical, cost-effective implementation.

4.3 Summary of Findings

The primary finding of this research is that a closed-loop automation framework, constructed using open-source tools, significantly outperforms traditional manual methods in maintaining network configuration integrity. The empirical data collected from the GNS3 simulation environment demonstrated that the system achieved accurate rates in detecting configuration drift, ranging from unauthorized VLAN modifications to the deletion of critical security Access Control Lists (ACLs). By utilizing a Python-based comparison engine, the system was able to identify precise textual discrepancies within seconds, proving that a lightweight, vendor-neutral solution can effectively eliminate the "silent" risks associated with human error or unauthorized internal changes that often go unnoticed by standard monitoring tools.

Furthermore, the study established that the autonomous remediation cycle provides a transformative reduction in network downtime and vulnerability windows. The results showed an average Mean Time to Remediation (MTTR) of approximately 45.3 seconds, with the entire self-healing loop-from detection to full recovery-consistently completing in under one minute. This speed represents a significant improvement over manual intervention, which was observed to take considerably longer due to the inherent delays in human notification and troubleshooting. The use of Ansible for atomic rollback operations ensured that devices were restored to their "Golden State" without service interruption, validating the system's ability to maintain high availability in educational institution and small-to-medium enterprise (SME) context.

Finally, the findings confirm that the system successfully fulfilled the objective of providing a cost-effective, vendor-neutral alternative to expensive proprietary platforms. By leveraging GNS3, Python, and Ansible, the research demonstrated that sophisticated autonomic computing principles-self-configuration, self-healing, and self-protection-can be implemented without specialized hardware or costly licensing fees. The system proved its scalability within the defined SME scope, showing that it can manage multiple device states simultaneously while maintaining a perpetual state of security policy compliance. Ultimately, the research proves that open-source automation is a viable and robust path for organizations seeking to enhance their network resilience and security posture on a limited budget.

Chapter 5

Conclusions and Recommendations

5.1 Introduction

This chapter presents the conclusions, recommendations, and areas for future work derived from the design, implementation, and empirical testing of the vendor-neutral, open-source self-healing network configuration system. The primary goal of this study was to address the critical gaps in modern network management-specifically, the reactive nature of standard open-source tools and the high costs associated with proprietary, enterprise self-healing ecosystems. By synthesizing real-time bitwise drift detection, autonomous rollback logic, and security policy validation using Ansible, Python, and GNS3, this project established a robust, cost-effective framework for maintaining infrastructure resilience and compliance. The following sections synthesize the operational insights gained from the system's performance and outline strategic paths for future development and deployment.

5.2 Conclusions

The conclusions of this research are directly mapped to the completion of the project's predefined core research objectives

A realistic, scalable network topology was successfully engineered within GNS3 using authentic Cisco IOSv nodes. Testing confirmed that virtualized images running via nested QEMU/KVM virtualization inside the GNS3 VM accurately mirrored the protocol responses, routing behaviors, and configuration syntax of physical hardware, providing a reliable and valid testing bed.

By utilizing an agentless architecture combining Ansible playbooks and custom Python scripting, a vendor-neutral automation controller was successfully deployed. This controller established a definitive "Golden Configuration" as an immutable source of truth, optimizing the rapid deployment of baseline configuration states without proprietary platform lock-in.

The implementation of a programmatic drift detection engine leveraging bitwise and line-by-line configuration comparisons proved highly effective. The mechanism reliably identified configuration anomalies, unauthorized command injections, and manual tampering in real time, successfully distinguishing between unapproved configuration drift and sanctioned administrative updates.

The system successfully demonstrated closed-loop automation by executing automated rollbacks to clear non-compliant states without human intervention. When unauthorized changes or policy violations occurred, the self-healing logic autonomously restored the affected devices back to their secure "Golden" baselines within a minimal time window, drastically reducing system downtime.

Empirical evaluation under simulated network errors, unauthorized port modifications, and routing protocol tampering proved that the framework provides superior infrastructure

hardening compared to manual management. The continuous auditing layer successfully validated configurations against industry benchmarks (such as CIS and NIST principles), reinforcing the network's defensive posture and mitigating the window of vulnerability.

5.3 Recommendations

Based on the successful implementation and performance evaluation of the self-healing network system, the following recommendations are proposed for network administrators, small-to-medium enterprises (SMEs), and educational institutions:

Organizations should actively shift away from legacy, reactive network administration methodologies-which rely on daily or weekly configuration backups and manual intervention-toward a declarative model. Implementing an automated controller ensures that the desired state of the infrastructure is continuously enforced, minimizing the risk of prolonged security exposures caused by unnoticed configuration errors.

For autonomous remediation to remain successful without introducing operational instability, organizations must implement strict version control and verification protocols for their baseline configurations. Since the system treats the "Golden Configuration" as an absolute source of truth, these templates must be rigorously audited, signed off by senior security personnel, and updated regularly to match current operational needs and compliance updates.

To guarantee the operational continuity of the automated self-healing controller, it is highly recommended to isolate the automation traffic on a dedicated Out-of-Band (OOB) management network. Ensuring that configuration monitoring and rollback tasks run over a distinct physical or logical plane prevents the automation controller from losing access to a device if a data-plane misconfiguration or localized DDoS attack disrupts standard in-band connectivity.

SMEs, public sector organizations, and educational institutions facing budget constraints should leverage open-source orchestration suites (such as Ansible, Git, and Python) rather than purchasing expensive, proprietary hardware-bound subscriptions. This study demonstrates that high-fidelity compliance tracking and autonomous self-healing can be achieved cleanly without vendor lock-in, democratizing access to advanced cyber defense capabilities.

5.4 Future Work

While this project successfully achieved its core milestones, the boundaries imposed by its original scope highlight several valuable opportunities for future academic and technical expansion:

Multi-Vendor Heterogeneity: The current implementation was primarily delimited to Cisco IOSv software environments. Future research should expand the scripts' abstraction and parsing layers-potentially integrating multi-vendor libraries like NAPALM-to dynamically

interpret and remediate configurations across a heterogeneous hardware environment comprising Juniper, Arista, and open-source Linux routing platforms.

Intent-Aware Analytics via Machine Learning: To eliminate the risk of false-positive rollback triggers during complex network shifts, future iterations could integrate lightweight Machine Learning (ML) algorithms. Introducing intelligent context parsing would allow the controller to analyse administrative intent, dynamically differentiating between an adversarial attack vector and an emergency, real-time manual adjustment executed by an authorized onsite technician.

Transition to Physical Infrastructure Deployment: Moving beyond the high-fidelity GNS3 simulated environment to validate the system on physical enterprise hardware switches, firewalls, and production routers will help assess real-world hardware latencies and physical link dynamics across geographically distributed networks.

Dynamic Threat Intelligence Integration: Future development should interface the automation controller directly with live cyber threat intelligence networks and vulnerability databases via secure APIs. This integration would allow the self-healing workflow to dynamically patch or harden the baseline configuration template in real time against newly discovered zero-day vulnerabilities.

5.5 Chapter Summary and Final Conclusion

The successful design, implementation, and validation of this self-healing network configuration framework demonstrate that the transition from human-dependent, reactive network management to autonomous, closed-loop orchestration is both technically viable and operationally essential. By combining the flexibility of open-source automation tools with rigorous security policy validation, this study has proven that organizations can effectively eliminate the "window of vulnerability" caused by configuration drift without incurring the prohibitive licensing costs of proprietary enterprise systems.

As network architectures continue to expand in scale and complexity, the integration of declarative state enforcement and autonomous remediation will remain a cornerstone of modern infrastructure hardening. Ultimately, this dissertation provides a strategic, accessible, and scalable blueprint for resource-constrained institutions to achieve enterprise-grade network resilience, continuous compliance assurance, and a reliable, self-defending operational posture.

REFERENCES

Ansible Documentation. (2024). Ansible Automation Platform documentation. https://docs.ansible.com/

Bass, L., Clements, P., & Kazman, R. (2021). Software architecture in practice (4th ed.). Addison-Wesley.

Burgess, M. (2004). Cfengine: A site configuration engine. USENIX Computing Systems.

Centre for Internet Security. (2024). CIS Benchmarks. https://www.cisecurity.org/cis-benchmarks/

Cisco Systems. (2024). Cisco Catalyst Centre (formerly Cisco DNA Centre). https://www.cisco.com/

Feamster, N., Rexford, J., & Zegura, E. (2014). The road to SDN: An intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review, 44(2), 87–98.

GNS3 Documentation. (2024). GNS3 network simulation software documentation. https://docs.gns3.com/

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75–105.

IBM. (2003). an architectural blueprint for autonomic computing. IBM Corporation.

Internet Engineering Task Force. (2006). RFC 4251: The Secure Shell (SSH) Protocol Architecture. https://www.rfc-editor.org/rfc/rfc4251

Juniper Networks. (2024). Juniper Apstra data centre automation and assurance. https://www.juniper.net/

Kim, H., & Feamster, N. (2013). Improving network management with software-defined networking. IEEE Communications Magazine, 51(2), 114–119.

Kurose, J. F., & Ross, K. W. (2021). Computer networking: A top-down approach (8th ed.). Pearson.

Limoncelli, T. A., Hogan, C. R., & Chalup, S. R. (2017). The practice of system and network administration (3rd Ed.). Addison-Wesley.

National Institute of Standards and Technology. (2020). Security and privacy controls for information systems and organizations (NIST SP 800-53 Rev. 5). U.S. Department of Commerce.

NAPALM Documentation. (2024). Network Automation and Programmability Abstraction Layer with Multivendor Support. https://napalm.readthedocs.io/

Oppenheimer, P. (2010). Top-down network design (3rd Ed.). Cisco Press.

Python Software Foundation. (2024). Python 3 documentation. https://docs.python.org/3/

RANCID Project. (2024). Really Awesome New Cisco Configuration Differencer (RANCID). https://shrubbery.net/rancid/

Sommerville, I. (2016). Software engineering (10th Ed.). Pearson.

Stallings, W. (2020). Network security essentials: Applications and standards (7th Ed.). Pearson.

Tanenbaum, A. S., & Wetherall, D. J. (2013). Computer networks (5th Ed.). Pearson.

Turnbull, J. (2018). The Docker book: Containerization is the new virtualization. James Turnbull.

White, G. B., Fisch, E. A., & Pooch, U. W. (2017). Computer system and network security. CRC Press.

Yick, J., Mukherjee, B., & Ghosal, D. (2008). Wireless sensor network survey. Computer Networks, 52(12), 2292–2330.

Zhou, Y., Chen, X., & Li, H. (2021). Automated network configuration management and compliance verification in enterprise environments. International Journal of Network Management, 31(4), 1–17.