Università degli Studi di Genova
Scuola Politecnica & Facoltà di Ingegneria

Laurea Magistrale in Ingegneria Strategica

International MSc in Engineering Technology for Strategy and Security of Genoa University

Title: - Challenges in applying cloud cutting edge technologies in air gapped on premises datacenters.

Advisor: - Prof. Marco Gotelli (unige)

Candidate: - Mr. Parag Dubey (s6568223)

Academic Year 2024-26
STRATEGOS

Acknowledgements:

First and foremost, I would like to express my sincere gratitude to the **University of Genoa** and **Hitachi Rail** for providing me with the opportunity to work on this thesis, which has been both a challenging and rewarding journey into the world of cloud technologies and railway automation.

I am deeply grateful to my academic supervisor, **Prof. Marco Gotelli**, whose guidance, expertise, and encouragement have been invaluable throughout the research process. His mentorship helped shape the direction and depth of this thesis, and his continuous support kept me motivated and focused.

A heartfelt thank you to **Mr. Giancarlo Camera**, my company supervisor at Hitachi Rail, for his trust and for giving me the opportunity to explore and contribute to a topic that is both innovative and impactful. His insights and constructive feedback have been crucial to my growth and the success of this work.

I would also like to extend my special thanks to **Mr. Gianmarco Garrisi**, One of my colleagues at Hitachi Rail, whose constant support, technical guidance, and collaborative spirit significantly enriched my experience. Their willingness to share knowledge and their patient assistance helped me overcome many of the practical challenges faced during this project.

A special mention goes to **Mr. Lorenzo Motta**, whose recommendation at the beginning of this journey played a vital role in making this experience possible. I am truly grateful for his support and belief in my potential.

Finally, I wish to thank all the professionals and faculty members who, directly or indirectly, contributed to the development of this thesis. This experience has not only broadened my technical understanding but also strengthened my passion for solving real-world problems at the intersection of innovation and infrastructure.

Abstract:

The digital transformation of industrial sectors like railway systems is driving the adoption of cloud-native technologies to improve efficiency, automation, and maintainability. However, many railway environments operate within strict security and operational constraints, such as air-gapped on-premises datacentres that are physically isolated from the internet. These constraints present significant challenges when trying to leverage the benefits of cutting-edge cloud technologies, which are typically designed for connected and flexible environments.

This thesis explores the complex landscape of adapting cloud-ready tools and practices particularly Infrastructure as Code (IaC) solutions such as Ansible and Terraform to air-gapped deployments. Through a detailed analysis, the work examines the compatibility, portability, and limitations of these technologies when used outside of traditional cloud environments. The research aims to understand what makes certain tools more adaptable to disconnected infrastructures and how deployment workflows can be reimagined to meet both cloud and on-premises requirements.

As part of the study, a practical proof of concept is developed, demonstrating the deployment of virtual infrastructure using Ansible and Terraform in an air-gapped railway automation setting. The goal is to bridge the gap between modern DevOps practices and the real-world constraints of critical infrastructure, offering a pathway to modernize deployment processes without compromising on security or reliability. Ultimately, this thesis contributes to the broader effort of bringing the agility of the cloud to environments where connectivity cannot be assumed, supporting the secure modernization of critical on-premises infrastructure.

Table of Contents

List of figures

List of Tables:

1.Introduction

Modern infrastructure has grown increasingly complex, and nowhere is this more consequential than in critical sectors such as transport and railways, where the way software and hardware are deployed, managed, and maintained has had to change fundamentally. Cloud-native technologies have reshaped IT infrastructure by making automation, scalability, and reproducibility achievable through **Infrastructure as Code** (IaC), orchestration tooling, and DevOps practices. These technologies, however, were conceived for connected environments with unrestricted internet access and elastic resource provisioning. That assumption breaks down in domains such as railway automation, where deployments typically take place in air-gapped, high-security on-premises environments governed by strict performance, safety, and cybersecurity requirements [1].

Railway automation systems run in mission-critical, real-time conditions that cannot tolerate the unpredictability or exposure associated with internet-connected platforms. For this reason, they are usually hosted in air-gapped datacentres, fully isolated from public networks to satisfy security and compliance obligations. Even under this isolation, the benefits that cloud-native tooling offers faster provisioning, standardised environments, and automated workflows remain highly desirable. Closing the distance between modern IT automation and operational isolation is therefore both a genuine engineering challenge and an opportunity for innovation [2].

This thesis examines how far cloud-native technologies, specifically Terraform and Ansible, can be adapted to air-gapped, on-premises infrastructure. Both tools are widely used in cloud settings to manage virtualised and containerised resources: Terraform provisions infrastructure through declarative templates and state-based configuration, while Ansible handles post-provisioning configuration, package management, and orchestration through an agentless, SSH-based model. Applying them in a disconnected setting, however, forces a rethink of conventional workflows. Without access to public registries, cloud APIs, or the open internet, an engineer must resolve module sourcing, dependency management, binary packaging, and artifact mirroring entirely inside a closed network.

At Hitachi Rail I developed a proof of concept (PoC) that demonstrates the feasibility of using these technologies to automate infrastructure provisioning and configuration in an air-gapped setting. The work begins by designing modular, reusable Terraform configurations that provision virtual machines (VMs) on Proxmox VE, the open-source virtualization platform commonly used in Hitachi environments. The challenge is twofold. First, the entire infrastructure must be provisioned without external internet access, which requires every Terraform provider and dependency to be mirrored locally. Second, the deployment must run as a fully automated CI/CD pipeline managed through GitLab, itself hosted inside the air-gapped network. Meeting both conditions calls for a working understanding of how to mirror Terraform providers locally, how to manage state files securely, and how to configure GitLab runners that can execute Terraform and Ansible jobs in a completely self-contained environment.

In parallel, an Ansible runner machine is used to automate post-provisioning tasks package installation, system configuration, and network setup on the newly created VMs. This complements Terraform's declarative provisioning with dynamic configuration management, so that every machine is consistently prepared for operational use. Together, Terraform and Ansible form a hybrid automation model that aligns with modern DevOps practice while respecting the isolation and compliance constraints of air-gapped infrastructure. Two further tools were integrated to support the system: Hauler, for transferring large artifacts and dependencies into the disconnected environment, and RKE2, a lightweight Kubernetes distribution suited to air-gapped container orchestration. Used together, these tools make it possible to build a self-sufficient DevOps ecosystem in which modern automation workflows operate reliably without any internet connection [3].

In short, this thesis is not only a theoretical study but a practical investigation of how cloud-native infrastructure automation can be adapted to industrial, air-gapped environments. It sets out the challenges, dependencies, and design decisions involved in achieving secure, efficient, and reproducible infrastructure management where connectivity simply cannot be assumed. The findings are intended to guide future IaC implementations in critical railway and industrial automation systems, and

in doing so to support the digital transformation of the next generation of on-premises infrastructure[1].

1.1 Introduction to Infrastructure as code -Overview

As IT and software development have evolved, the need for consistent, scalable, and repeatable infrastructure provisioning has become pressing. Infrastructure as Code (IaC) answers that need by allowing computing infrastructure to be managed and provisioned through machine-readable definition files rather than through manual hardware configuration or interactive administration.

At its core, IaC treats infrastructure the way it treats application code. Instead of configuring servers, networks, and storage by hand, engineers write code that describes the desired state of the infrastructure. That code can be versioned, tested, reused, and deployed automatically, giving a standardised, traceable, and auditable way to manage environments.

IaC tools fall into two broad categories: declarative and imperative. Declarative tools such as Terraform describe *what* the final state of the infrastructure should be, while imperative tools including some Ansible use cases specify *how* to reach that state step by step. This distinction shapes how infrastructure is managed, tested, and evolved over time [2], [4].

The advantages of adopting IaC are considerable. Provisioning and configuration that once took days can be completed in minutes. Reusing code keeps environments uniform and reduces human error. Storing infrastructure definitions in repositories enables rollback, auditing, and collaboration. And because IaC integrates naturally with continuous integration and continuous deployment (CI/CD), it forms the backbone of automated delivery pipelines.

In environments with strong security constraints air-gapped systems in defence, critical infrastructure, and transport, for example IaC offers a way to enforce controlled, repeatable processes without weakening operational integrity. Applying cloud-native tools such as Terraform or Ansible in these disconnected settings does require adaptation, including internally hosted

registries and offline workflows, but those same constraints also open room for innovation in secure deployment automation.

1.2 History and Evolution of IaC

Infrastructure as Code began to gain traction in the mid-2000s as IT systems grew rapidly in scale and complexity. Two developments accelerated the shift: the launch of Amazon Web Services' Elastic Compute Cloud (EC2) in 2006 and the rise of dynamic web frameworks such as Ruby on Rails. Together they made applications far easier to build and deploy and far harder to manage, as the number of servers and environments multiplied.

In the early days, system administrators relied on shell scripts and manual processes. As organisations scaled, the resulting inconsistencies and misconfigurations became a serious bottleneck. This gave rise to configuration management tools such as CFEngine, Puppet, and Chef, which allowed infrastructure to be defined and enforced as code, and which made it possible to automate provisioning and configuration across hundreds or thousands of machines.

A later generation of tools Ansible, Salt Stack, and Terraform built on that foundation with more intuitive syntax, better scalability, and stronger support for cloud and containerised environments. Terraform, released by HashiCorp in 2014, was particularly influential: its emphasis on declarative provisioning and provider-agnostic workflows made it a natural fit for multi-cloud and hybrid deployments.

Two architectural models also shaped IaC practice. In the push model, a central system pushes configuration out to servers, as Ansible does. In the pull model, each node retrieves and applies its own configuration, as Puppet and Chef do. Over time, IaC matured well beyond simple scripting, absorbing software-engineering practices such as version control with Git, modular design, testing and CI/CD integration.

IaC has also become tightly bound to DevOps culture. By bringing developers and operations together around shared tools and processes, it speeds up delivery, reduces errors, and improves collaboration. At the same time it introduces new security concerns: the Cloud Threat Report found that misconfigured IaC templates had produced hundreds of thousands of

vulnerabilities, underlining the need for secure-by-design practices and IaC code review within the DevSecOps pipeline. Today IaC is regarded as essential to any modern enterprise IT strategy whether the task is managing cloud infrastructure, automating datacentre provisioning, or orchestrating containers, it lets organisations scale infrastructure with the same discipline and agility they apply to software [5], [6].

1.3 Research Objective and questions.

The primary objective of this research is to investigate the challenges and feasibility of adopting cloud-native, cutting-edge automation technologies in air-gapped, on-premises datacentres. More specifically, the study evaluates how tools such as Terraform and Ansible designed for connected, cloud-based use can be adapted and used effectively in the offline, security-constrained infrastructure found in railway and industrial systems. It also asks whether applying Infrastructure as Code (IaC) principles can deliver a level of automation, reproducibility, and consistency comparable to the cloud, even under air-gapped conditions.

Specific objectives. To pursue this goal, the research sets out to:

- Identify the main technical and operational challenges of deploying cloud-ready tools (Terraform, Ansible, GitLab CI/CD, Helm, and similar) in air-gapped datacentres;
- Analyse the readiness and adaptability of these technologies for offline, on-premises use, with particular attention to dependency management, security, and integration;
- Design and implement a proof-of-concept (PoC) environment that automates infrastructure provisioning and configuration with Terraform and Ansible inside an air-gapped Proxmox setup;
- Evaluate the PoC by comparing manual and fully automated deployment in terms of time efficiency, reproducibility, and error rate;
- Assess how integrating IaC tools can improve security, compliance, and maintainability in railway-grade datacentres.

Research questions. The study is guided by five questions:

- What are the major challenges in applying cloud-native automation tools in air-gapped environments?
- How do different IaC tools compare across deployment scenarios and conditions, and which are best suited to this context?
- How can Terraform and Ansible be adapted to support infrastructure deployment without internet access?
- To what extent can IaC-based automation reduce deployment time and configuration errors compared with manual provisioning?
- What are the security implications and best practices for implementing IaC workflows in air-gapped infrastructure?

Hypothesis. It is hypothesised that implementing Infrastructure as Code with Terraform and Ansible in air-gapped datacentres can significantly reduce deployment time and configuration errors while improving reproducibility and compliance, despite the absence of cloud connectivity.

1.4 Scope and Limitations.

Scope: This research spans several technical and operational dimensions of deploying cloud-native technologies in air-gapped, on-premises environments. It focuses on showing how Infrastructure as Code (Terraform) for provisioning and Ansible for configuration management can improve the automation and scalability of infrastructure inside restricted networks. By automating the creation and management of virtual machine pools, the approach aims to cut deployment and configuration time substantially and so improve the efficiency of the development lifecycle: in large industrial or enterprise settings, where deadlines matter, this kind of automation can accelerate development, testing, and production readiness. The use of IaC also improves consistency, reusability, and flexibility, letting teams reproduce environments quickly and reduce the human error that manual deployment tends to introduce. Configuration changes and updates can be

version-controlled, tested, and applied across environments with minimal downtime, giving a more reliable and maintainable platform.

Limitations: Several constraints temper these benefits. The foremost is dependency management: every provider, package, and module must be mirrored locally, because the system cannot reach online repositories, and maintaining those mirrors adds time and administrative overhead especially when tools release frequent updates. A related limitation is tool compatibility and version control. In an offline setting, each update must first be downloaded and verified on an internet-connected machine and then transferred manually into the air-gapped system, which slows the adoption of new features and security patches. Terraform and Ansible therefore improve deployment efficiency but using them in air-gapped environments demands careful planning, disciplined version management, and continuous synchronisation of local mirrors to remain reliable over the long term.

1.5 Thesis Structure overview.

The thesis is organised into chapters that follow the stages of the research, each contributing to the overall aim of evaluating cloud-native automation in air-gapped environments.

- **Chapter 1 — Introduction** sets out the motivation, problem statement, objectives, scope, and structure of the work.
- **Chapter 2 — State of the Art** reviews existing Infrastructure-as-Code tools and related automation technologies and identifies the research gap.
- **Chapter 3 — Tool Selection and Air-Gapped Strategy** compare tools within the same categories and discusses their applicability and limitations in air-gapped, on-premises infrastructure.
- **Chapter 4 — Methodology** describes the research design and experimental approach, including how the tools were evaluated and why the Proxmox/Telmate platform was selected as the underlying infrastructure provider.

- **Chapter 5 — Implementation** details the technical realisation of the PoC: defining prerequisites, designing reusable VM templates, structuring repositories for the different environments (DEV-S, DEV-M, and others), and managing state. It also presents the Terraform-based pipeline, covering infrastructure validation through state files, the comparison between Terraform outputs and Ansible inputs, automated deployment of application tarballs, and artifact storage for traceability.
- **Chapter 6 — Results and Findings** evaluate the implementation, comparing manual and automated deployment and reporting improvements in provisioning time, reproducibility, and error reduction, together with the operational benefits seen in testing and pre-production.
- **Chapter 7 — Conclusion and Future Work** summarise the findings, sets out the contribution to automation in air-gapped infrastructure, and outlines directions for extending the work toward production-ready environments.
- **REFERENCES**
- **APPENDIX**

Overall, the thesis moves logically from theoretical background and tool comparison through methodological design to practical implementation and empirical evaluation, so that both the conceptual framework and its technical realisation are presented coherently in support of the research objectives.

2 State of the Art

This literature review was conducted using a structured search strategy to ensure comprehensive coverage of relevant work. The primary databases searched were IEEE Xplore, ACM Digital Library, Springer Link, ScienceDirect, and Google Scholar, supplemented by arXiv for preprints and NIST for standards documentation.

Keywords and combinations used:

- "Infrastructure as Code" AND ("air-gapped" OR "offline" OR "disconnected" OR "on-premises")
- "Terraform" OR "Ansible" AND "automation" AND "deployment"
- "DevOps" AND ("critical infrastructure" OR "railway" OR "industrial control systems")
- "Configuration management" AND ("security" OR "compliance" OR "ICS")
- "Software supply chain" AND ("SBOM" OR "reproducible builds")

Inclusion criteria: Peer-reviewed publications (2015–2026), major conference proceedings, recognised standards (NIST, IEC), and highly cited preprints.

Exclusion criteria: Blog posts, vendor marketing materials, non-peer-reviewed grey literature (with exceptions for NIST standards and official framework documentation).

The review identified a substantial body of work on IaC in cloud-connected environments but confirmed a significant gap in research addressing fully air-gapped deployments validating the research problem this thesis addresses.

2.1 The shift to Infrastructure as Code

Infrastructure as Code (IaC) has fundamentally changed how computing environments are built and operated. The transformation from manual server configuration to machine-readable, version-controlled infrastructure

definitions is well documented across the literature, though authors emphasise different aspects of this shift.

Artac et al. [1] frame IaC primarily as a DevOps enabler, emphasising its role in making automation and repeatability systemic properties rather than individual discipline. Guerriero et al. [2] take a more empirical approach, surveying industry practitioners to identify adoption drivers and challenges finding that while IaC improves consistency, teams struggle with testing and debugging infrastructure code. Rahman et al. [6] provide the broadest view through their systematic mapping study, identifying 44 primary studies and noting that most research concentrates on defect detection and configuration testing, with deployment in constrained environments receiving notably less attention.

These perspectives converge on a common conclusion: IaC represents a paradigm shift from imperative manual processes to declarative, auditable automation. However, the literature implicitly assumes connected environments throughout an assumption this thesis challenges directly.

2.2 From configuration management to declarative provisioning

The evolution of IaC tooling has been characterised by increasing abstraction and decreasing operational overhead. Chiari et al. [5] trace this trajectory through their survey of static analysis approaches, noting the progression from early agent-based tools (Puppet, Chef) through agentless configuration (Ansible) to declarative provisioning (Terraform). Özdoğan et al. [7] provide a systematic comparative analysis of IaC technologies, evaluating them across dimensions of language paradigm, execution model, and ecosystem maturity.

The theoretical distinction between declarative and imperative approaches is established by Fahland et al. [8] in the broader context of process modelling, while Howard [9] demonstrates Terraform's specific realisation of the declarative model for infrastructure. Novakova Nedeltcheva et al.[10] extend this discussion by examining model-driven approaches to IaC generation, suggesting that higher-level abstractions may further reduce the gap between intent and implementation.

A critical observation emerges from comparing these sources: all assume ready access to public registries, plugin repositories, and cloud APIs. The tooling evolution has been driven by and optimised for connected cloud environments a bias that becomes apparent only when one attempts to operate these tools in isolation.

2.3 A cloud-native field with a cloud-native blind spot

The security dimension of IaC in critical infrastructure demands attention to established frameworks that the IaC literature has largely overlooked. NIST SP 800-82 Rev. 3 [11] provides comprehensive guidance for securing operational technology (OT) environments, including railway systems, while NIST SP 800-53 Rev. 5 [12]defines the security controls against which any automation framework in regulated environments must be evaluated.

Ozerov [4]examines cybersecurity specifically in railway command and control systems, identifying the migration to networked technology as a growing threat vector precisely the context in which this thesis operates. Ibadah et al. [3] extend this analysis to modern on-board and track-side infrastructure, proposing a comprehensive cybersecurity strategy that includes network segmentation and access control principles directly applicable to air-gapped deployment architecture.

The ISA/IEC 62443 standard [24] provides the industrial automation security framework most relevant to this work. Notably, none of the surveyed IaC literature systematically addresses compliance with these standards in automated deployment pipelines a gap this thesis begins to fill through its security-conscious design choices (credential isolation, least-privilege access, network segmentation).

The emerging concern of software supply chain security further complicates air-gapped deployments. The NTIA's SBOM framework [25] establishes minimum transparency requirements for software components, while Lamb and Zacchiroli [13] demonstrate that reproducible builds can verify software integrity even without network access a property directly relevant to air-gapped dependency management.

2.4 The air-gapped paradigm conflict

The conflict between cloud-native tooling assumptions and air-gapped operational reality is the central tension this thesis addresses. Standard IaC workflows take internet access for granted: providers are fetched automatically, modules are pulled from public registries, and remote APIs are reached as a matter of course [2], [7]. In an air-gapped system, every one of these assumptions is invalid.

Park et al. [14] address a related but distinct problem bespoke private cloud orchestration demonstrating that custom provisioning solutions are sometimes necessary when standard cloud tooling proves inadequate. Their work reinforces the principle that tool adaptation rather than tool replacement is often the pragmatic path forward.

Shahin et al. [15]identify architectural design challenges in DevOps that become amplified in disconnected environments: dependency management, state synchronisation, and pipeline orchestration all require fundamentally different strategies when external connectivity is unavailable. The result, as Özdoğan et al. [7]observe, is a markedly higher operational complexity that the connected-cloud literature does not address

2.5 The research gap

Two systematic reviews Rahman et al. [6] and their extended mapping study [16] confirm that the IaC research landscape overwhelmingly focuses on cloud-connected scenarios. Of the 44 primary studies identified by Rahman et al., none specifically addresses deployment under complete network isolation. Chiari et al.[5] reach a similar conclusion in their survey of IaC static analysis: all evaluated tools assume access to external registries and APIs.

This gap has concrete consequences for organisations in the railway and defence sectors. Beetz and Harrer [17]describe GitOps as "the evolution of DevOps," but their framework depends on continuous reconciliation agents with network access inapplicable without significant adaptation. Masek et al. [18] demonstrate Ansible's potential for lab automation but operate in a connected university environment.

The present thesis is positioned squarely in this gap. Rather than treat air-gapped operation as an edge case, it takes a single, realistic question can a cloud-grade IaC pipeline run with no internet at all? and answers it with a working implementation. The aim is not only to show feasibility, but to document the challenges and performance trade-offs concretely enough to serve as a reusable reference for future implementations.

selection of tools is the first step in turning the gap identified here into a method[5].

3. Tool Selection and Air-Gapped Strategy

3.1 Tools in practice

In air-gapped infrastructure, the choice of tooling is constrained before it is even made anything that depends on a live connection to a public registry or cloud API is a liability rather than a feature. Terraform and Ansible are often preferred in these conditions precisely because they are flexible about where their inputs come from and because Ansible's agentless model integrates cleanly with on-premises platforms such as Proxmox or VMware. The two divide the work naturally Terraform provisions, Ansible configures and a self-hosted GitLab instance ties them together into a single, auditable pipeline that runs entirely inside the secure perimeter. The table below summarises how each tool is used here and what each one demands of an offline environment.

Table 3.1 — Tools in practice: roles and air-gapped considerations.

Tool	Primary Role	Strengths	Air-Gapped Considerations
Terraform	Provisioning VMs, infrastructure	Declarative syntax, multi-provider support	Providers must be mirrored locally, plugins preloaded
Ansible	Configuration Management	Agentless, flexible, SSH-based execution	Galaxy collections need offline syncing
GitLab CI	Automation Pipelines	Local GitLab instance with runners supports full air-gapped CI/CD	Needs mirrored Docker registry, offline runners

Several wider trends shape how these tools are being used in restricted networks, and they informed the design choices in this work. Zero-trust principles are increasingly applied even inside isolated systems, so that authentication and verification happen at every layer rather than being assumed once a host is inside the perimeter. Immutable-infrastructure thinking is gaining ground too: rather than patching a running machine, the environment is rebuilt from a known definition, which makes deployments more predictable and harder to tamper with. Artifact-management tools such as Nexus, Hauler, have become near-essential, since they hold the offline mirrors of container images, Terraform providers, and Ansible collections that an air-gapped pipeline depends on. Finally, some organisations are experimenting with hybrid air-gap models, in which tightly controlled "sync windows" allow limited, policy-governed updates from outside. Taken together, these developments show that cloud-native ideas continue to drive innovation, but that applying them under isolation always requires real adaptation rather than a straight lift-and-shift [6].

3.1.1Declarative vs Imperative Models

Two paradigms describe how automated systems are specified and executed: the declarative and the imperative. Following Fahland et al. (2009), declarative modelling concerns *what* outcome should be reached, while imperative modelling concerns *how* to reach it through explicit, ordered steps. A declarative specification defines logic and end states without dictating the control flow; an imperative one prescribes the sequence of actions and transitions.

The distinction is easy to see in Infrastructure as Code. Terraform is declarative: the desired state of the infrastructure the virtual machines, networks, and resources is written into configuration files, and the engine reconciles reality with that definition. Ansible is closer to imperative: its playbooks describe tasks to be carried out in a particular order to configure a system. Each style has its strengths. Declarative models offer higher abstraction, repeatability, and ease of reasoning about state, which matters in air-gapped or heavily regulated settings; what they give up is fine-grained procedural control. Imperative models give exactly that control and are often

more intuitive for step-by-step work, but they are easier to get wrong and harder to scale across large, changing infrastructure [2]. Studies of process modelling reinforce the point: imperative representations suit sequential information, while declarative ones better capture circumstantial relationships (Fahland et al. 2009).

This work uses both, deliberately. Terraform handles provisioning, where reproducibility and state management are paramount, and Ansible handles the procedural configuration of software on top. Combining them keeps the abstraction of a declarative core while retaining the control of an imperative layer, which is what makes the resulting pipeline both maintainable and precise in an air-gapped environment[7], [19].

3.1.2 Push vs Pull Architectures

Push and pull describe the two fundamental ways configuration reaches a managed host. In a push architecture, a central controller actively sends configuration or commands out to target nodes. In a pull architecture, each node periodically retrieves its own desired state from a central repository and applies it. The two carry different operational, scaling, and security implications.

The push model, used by Ansible and by traditional SSH-driven scripts, puts the controller in charge: an operator or automation engine initiates the work and pushes updates to every host. This gives strong central control and makes consistency and time-critical rollouts easier to coordinate, but it requires direct network reachability from the controller to every managed host something that can be awkward in heavily segmented networks. The pull model, used by Puppet, Chef, and Kubernetes-with-GitOps, inverts the relationship: nodes fetch the latest configuration, apply it, and report back. It scales well and reduces the need for constant connectivity from a control node, at the cost of new concerns such as configuration drift and delayed convergence when nodes are not synchronised often.

Inside an air-gapped, on-premises environment both models have a place. Push tends to fit best where outbound connectivity from nodes is restricted, because every action originates from a single secured controller within the

same perimeter which is exactly why Ansible's push model suits tightly controlled railway or defence systems. Pull offers more autonomy and resilience, since a node can keep and reapply its own configuration and recover when a connection is briefly lost; in a fully disconnected system it can still operate internally, with nodes synchronising from a local mirror or internal Git repository rather than an external one. In practice, modern IaC blends the two. This research uses the push model (Ansible) for deterministic provisioning and configuration inside the air-gapped datacentre, while reserving pull mechanisms (GitLab CI/CD, and GitOps extensions in future) for synchronising configuration state from internal repositories. The hybrid gives both central control and decentralised resilience, which matches the operational reality of critical, offline environments.

3.2 Project Explanation and tools Used

This research looks at how cloud-native automation Terraform and Ansible in particular can be adapted to air-gapped, on-premises datacentres. Environments like these, cut off from external networks, are common in railway, defence, and other critical sectors where reliability and cybersecurity come first. The difficulty is that most IaC frameworks were built for connected cloud ecosystems and lean on internet-based repositories for modules, providers, and updates. The study therefore asks how these tools can be mirrored, configured, and orchestrated so that the same level of automation and consistency is achievable with no connection at all [14], [20].

3.2.1 Deploying in Air-gapped Environment

Running IaC in an air-gapped setting introduces several distinct categories of difficulty. The most immediate is network and dependency isolation: with no internet, the normal resolution of Terraform providers, Ansible roles, and CI/CD runner images simply stops working. Security and compliance grow more demanding, because secure key management, update validation, and conformance to standards such as IEC 62443 are harder when external verification is impossible. There are scalability questions too cloud-native automation assumes elastic capacity and adapting it to fixed-capacity

offline infrastructure means rethinking state management and resource provisioning. On top of this sit regulatory constraints, since railway systems work under strict certification and audit rules that limit how freely a toolchain can be modified or updated. And finally, there is the ongoing maintenance burden: without online synchronisation, even a minor update has to be mirrored by hand, which adds steady administrative overhead [4], [10][10].

3.2.2 Proposed Architectural Solutions.

To address these issues, the work proposes an end-to-end offline automation pipeline built on a few firm design principles. All required providers, modules, and packages are mirrored inside the air-gapped network, which guarantees reproducibility and removes any external dependency.

Terraform, using the Telmate/Proxmox provider, provisions virtual machines and networks declaratively and with managed state, supporting an immutable-infrastructure approach in which changes are made by redeploying code rather than patching machines by hand. Ansible then takes over after provisioning to enforce configuration, apply security policies, and deploy application packages; its agentless, SSH-based model is well suited to a restricted environment.

A self-hosted GitLab instance acts as the orchestration engine, running the Terraform and Ansible pipelines entirely within the local network, with offline runners and mirrored Docker registries keeping the whole workflow automated and auditable. The combination produces a self-sufficient platform that reproduces cloud-level efficiency and traceability while respecting the isolation and compliance requirements of railway-grade infrastructure [15], [19], [21].

3.2.3 Integrated Framework Components

Component	Tool	Primary Function	Air-Gap Adaptation

Hypervisor	Proxmox VE	Virtualization platform for hosting VMs and containers	Self-hosted cluster; no internet dependency
Provisioning	Terraform	Declarative infrastructure provisioning	Local providers and state backends
Configuration	Ansible	System configuration and application deployment	Offline roles, local package repositories
Automation	GitLab CI/CD	Pipeline execution and orchestration	Self-hosted runners and mirrored registries

Table 3.2 — Integrated framework components

This integration establishes a **closed-loop automation pipeline**: Terraform provisions resources, Ansible configures them, and GitLab coordinates the process. Together, these components form a replicable blueprint for **air-gapped Infrastructure-as-Code environments**.

3.2.4 Terraform, Ansible and Gitlab CI: Comparative analysis and integration

Terraform (provisioning): Terraform is the primary declarative provisioning tool in this work. It describes the desired infrastructure VMs, networks, storage in HCL and keeps a state file that represents what actually exists. For air-gapped, on-premises use its main advantages are its provider-based architecture (which makes Proxmox integration possible), its modular design, and a clear plan/apply lifecycle that supports safe, auditable change [2]. The offline considerations are specific: providers and modules have to be pre-downloaded to a local mirror, and state backends should use internal, locked, and encrypted storage for example a GitLab-backed HTTP endpoint or an on-premises S3-compatible store. Version pinning and the **.terraform.lock.hcl** file are what guarantee reproducibility once there is no remote registry to fall back on [2].

Ansible (configuration management): Ansible is the procedural layer that runs after provisioning. Its playbooks execute tasks against inventory hosts OS hardening, package installation, service configuration, application deployment and because it is agentless and relies on standard SSH, it fits tightly controlled networks and existing operational habits well [12]. Offline, its roles, collections, and any package dependencies must be staged inside the isolated network, sensitive data must be handled through Ansible Vault

with strict inventory separation, and in practice it complements Terraform by carrying out the fine-grained steps that declarative provisioning deliberately leaves alone[7].

GitLab CI (orchestration): A self-hosted GitLab CE instance orchestrates the whole sequence: version control, pipeline definition in .gitlab-ci.yml, and execution through internal runners. It sequences the Terraform plan/apply and the Ansible runs, stores artifacts such as plans and inventory files, and provides the audit trail for every change [8]. To work air-gapped it has to be deployed inside the perimeter, its runners need preloaded binaries and local caches, any container images its pipelines reference must come from an internal registry such as Nexus or Harbor, and its CI variables and secrets must be stored securely under strict access control.

The reason for using all three together is a clean separation of concerns. Terraform answers *what resources exist*, Ansible answers *how they are configured*, and GitLab governs the *lifecycle that connects them*. Terraform's state-and-plan model combined with GitLab's pipeline logs produces a workflow that is reproducible and auditable end to end. Pre-mirroring providers, images, and packages, and serving them from internal registries, is what lets these cloud-grade practices run with no external connectivity at all while Ansible's agentless model keeps integration with self-hosted runners and existing SSH-based operations simple.

3.2.5 Integration model and summary

The combined pipeline follows a straightforward sequence. Terraform provisions the VMs and networks in Proxmox; Ansible installs and configures the components on top; GitLab CI automates and monitors the whole lifecycle; and Proxmox hosts and manages all of the virtualised resources underneath. As established in Section 3.1.1, Terraform operates declaratively while Ansible follows a procedural model the practical implications of this distinction for the present work are summarised in Table 3.3:

Aspect	Terraform	Ansible
Primary Role	Infrastructure provisioning	Configuration management
Language Type	Declarative (HCL)	Procedural (YAML)
Execution Logic	Plans and applies changes to reach desired state	Executes tasks in a specific order
State Management	Maintains state file to track infrastructure	No state by default
Provisioning Support	Excellent (especially with VMs/networks)	Limited (only with some cloud modules)
Configuration Support	Limited (e.g., cloud-init)	Extensive (users, packages, services)
Error Handling	Built-in diff and planning mechanism	More manual (e.g., when, ignore_errors)
Idempotency	Guaranteed through HCL state tracking	Requires careful task definitions
Air-Gapped Usability	High - can use local providers/modules	High - works offline with local roles/scripts

Table 3.3 — Core differences between Terraform and Ansible.

Together these establish a complete Infrastructure-as-Code lifecycle and show that cloud-native automation principles can be reproduced in a fully offline, air-gapped environment. Through careful mirroring, modular design, and disciplined security management, the project reduces manual intervention, speeds up provisioning, improves reproducibility and auditability, and stays within railway-grade operational standards. It also lays the groundwork for later extensions container orchestration with RKE2 or K3s, and GitOps-based management with Flux or Argo CD adapted for secure, disconnected use.

3.3 Design Intent: GitOps and Policy-as-Code

While the current implementation uses GitLab CI/CD to trigger Terraform and Ansible in a push-based model, the architectural design was conceived with GitOps and Policy-as-Code principles in mind. In a GitOps model, Git

serves as the single source of truth and infrastructure changes are reconciled continuously against declared state. Policy-as-Code extends this by encoding compliance rules (e.g., through tools such as Checkov or Sentinel) as pipeline gates. Neither GitOps reconciliation controllers nor Policy-as-Code enforcement tools were deployed in this work; however, the modular, declarative, version-controlled architecture established here is designed to accommodate them as future extensions. The detailed design for these planned capabilities is discussed in Section 7.4.

3.4 Security & Compliance.

Security is one of the most demanding aspects of automation in air-gapped infrastructure, and one that sits at the intersection of several established frameworks relevant to railway and industrial systems. Automation accelerates deployment and improves consistency, but it also concentrates high-value assets SSH credentials, API tokens, Terraform state files any of which, if exposed, could compromise the entire system. For this reason, security was treated as a design input from the outset of this project rather than as a post-deployment audit.

3.4.1 Relevant Standards and Frameworks

Two bodies of standards directly govern the security context of this work. The ISA/IEC 62443 series[25] defines security requirements for industrial automation and control systems (IACS), including railway applications. It establishes security levels (SL 1–4) across zones and conduits, and its zone-and-conduit model maps directly onto the network segmentation and access control strategy adopted in this implementation. NIST SP 800-82 Rev. 3 [11] provides complementary guidance specifically for operational technology (OT) environments, addressing patch management, remote access, and network architecture in systems where availability and safety take precedence over conventional IT security priorities. NIST SP 800-53 Rev. 5 [12] further defines the security and privacy controls against which any automated deployment pipeline in a regulated environment should be evaluated including access control (AC), configuration management (CM), and supply chain risk management (SR) control families directly applicable here.

No formal certification against these standards was pursued within this thesis that falls within the scope of a production deployment rather than a proof of concept. However, the design choices made throughout the implementation were informed by and aligned with their core principles, as described below.

3.4.2 Credential and Secret Management

Sensitive credentials were never embedded in Terraform or Ansible configuration files. API tokens, SSH keys, and GitLab access tokens were supplied at runtime through GitLab CI/CD variables or entered manually where pipeline injection was not possible. This practice directly addresses the NIST SP 800-53 IA (Identification and Authentication) and SC (System and Communications Protection) control families.

The Terraform state file was treated as a sensitive artifact throughout, because terraform.tfstate can contain real resource identifiers, internal IP addresses, and metadata that would assist an attacker in mapping the infrastructure. Access to the state backend was restricted to authorised pipeline processes, and the GitLab HTTP backend provided both persistence and locking to prevent concurrent modification.

Early in the project, a security finding identified that initial versions of the pipeline scripts had embedded sensitive values directly in configuration a risk that persists in Git history even after deletion from the working tree. All such instances were identified, the credentials rotated, and the historical commits sanitised. This experience reinforced the importance of secret scanning as a pipeline gate a capability identified as a future extension in Section 7.4.2.

3.4.3 Access Control and Least Privilege

Access followed the principle of least privilege at every layer. Proxmox API credentials were scoped to the minimum permissions required for VM creation and management. GitLab runner permissions were restricted to the specific projects and environments they served. SSH access to provisioned virtual machines was mediated through the bastion node, reducing the direct attack surface even within the already-isolated internal network.

This approach aligns with the ISA/IEC 62443 requirement for role-based access control within industrial automation zones and with NIST SP 800-82's guidance on limiting administrative access in OT environments.

3.4.4 Software Supply Chain and Dependency Integrity

The air-gapped nature of the environment introduces a supply chain security dimension that connected deployments can partially offload to upstream registries. Here, every provider binary, OS package, container image, and Ansible collection was downloaded on an internet-connected staging machine, scanned for known vulnerabilities, and transferred into the isolated environment through a controlled one-way process. Once inside, dependencies were served from locally hosted repositories (Nexus / Hauler) rather than fetched individually at deployment time.

This mirrors the intent of the NTIA SBOM framework [26], which calls for transparency about the components included in a software system. While a formal SBOM was not produced for this proof of concept, the mirrored artifact inventory maintained in the local repositories provides an equivalent level of traceability every component version is known, pinned, and accessible for audit. Lamb and Zacchiroli's work on reproducible builds [13] further informs this approach: by pinning provider versions in .terraform.lock.hcl and using immutable golden images as VM templates, the implementation ensures that the same inputs always produce the same deployed state, which is a prerequisite for meaningful integrity verification.

3.4.5 Network Segmentation and Isolation

The deployment architecture inherently enforces network isolation: all components operate within a physically air-gapped perimeter, with the single controlled ingress point being the one-way transfer from the external staging machine. Within the perimeter, VM pools are segmented by environment (DEV-S, DEV-M, TEST, UAT) through separate Proxmox pools and VLAN assignments, preventing lateral movement between environments in the event of a compromise.

This segmentation strategy reflects the zone-and-conduit model of ISA/IEC 62443 and the network architecture recommendations of NIST SP 800-82, both of which call for explicit boundary enforcement between zones of differing security levels.

3.4.6 Overall Security Posture

No specialised DevSecOps toolchain such as Checkov for Terraform policy scanning or Vault for dynamic secret management was deployed in this proof of concept. These capabilities are identified as planned extensions in Section 7.4.2. However, the consistent application of the practices described above credential isolation, least-privilege access, dependency scanning, supply chain traceability, and network segmentation ensured that the automation framework strengthened rather than eroded the security posture of the air-gapped infrastructure. Security was not an afterthought appended to a working system; it was a design constraint that shaped every architectural decision from the beginning.

4. Methodology

This chapter sets out how the research was designed and carried out. The work is best described as design-oriented and experimental: rather than testing a fixed hypothesis under controlled laboratory conditions, it builds a working artifact an air-gapped automation pipeline and evaluates it against the manual process it is meant to replace. The chapter first states the contributions the work claims, then describes the experimental environments, the diagnostic baseline established through manual deployment, the variables and metrics used to judge success, the methods by which data were collected, and finally the analytical approach used to interpret that data

4.1 Contributions

This thesis investigates how cloud-native automation technologies, normally designed for connected and elastic environments, can be adapted for air-gapped, on-premises infrastructure of the kind found in railway systems. The contributions are twofold [22].

The first is a **technical adaptation of cloud practices to offline environments.** The work shows that tools usually associated with cloud deployment Terraform, Ansible, and GitLab CI/CD can be repurposed and fully operationalised inside a completely isolated network, using Proxmox VE as the virtualization backend. Crucially, this is achieved with no external connectivity whatsoever, which preserves the security and compliance requirements that critical-infrastructure environments demand.

The second is an **identification of the practical challenges of air-gapped automation.** During implementation, several recurring obstacles were analysed in detail dependency mirroring, credential handling, state persistence, and infrastructure drift. Instead of leaning on external services, the thesis documents offline-compatible strategies for provider installation, package management, pipeline execution, and state synchronisation, so that the result is not just a working demonstration but a reusable account of what makes such a system work.

Tool selection rationale. Each tool earned its place for specific reasons. Terraform was chosen as the primary provisioning tool for its declarative HCL syntax, its predictable plan-and-apply execution model, and the

reproducibility it offers across environments; because its configuration is text-based, infrastructure definitions are easy to track, audit, and modify. Ansible was introduced for post-provisioning configuration and security hardening and was also used through the Ansible runner to install all the component tarballs on top of the provisioned VMs. A self-hosted GitLab CI/CD instance made it possible to run fully automated pipelines inside the air-gapped network, removing manual operator intervention while preserving traceability and repeatability. Proxmox VE, hosted in the local , served as the execution layer: it allowed safe testing, rollback, and iterative refinement of the infrastructure code without financial cost and without exposing sensitive environments to external networks unlike commercial cloud platforms, where a misconfiguration can translate directly into unpredictable operational charges.

4.2 Experimental Design (Environments: DEV-M, UAT ETC.)

The experimental setup was organised around the representative environments defined in Hitachi Rail's official deployment documentation: DEV-S, DEV-M (Development Medium), TEST, and UAT (User Acceptance Testing). The aim of the design was not simply to make automation work in isolation, but to ensure that the result was reusable, scalable, and aligned with existing industrial deployment standards.

Design foundations: The design drew on two sources at once. The first was practical experimentation with Terraform during the early research, in which different state-management models, file hierarchies, and module structures were tried and compared. The second was Hitachi Rail's internal deployment documentation, which prescribes how virtual machine pools, networking layers, and node classifications bastion nodes, control nodes, worker nodes, and other node types must be structured for each environment. Combining the two meant the automation framework mirrored real operational structures rather than an academic simplification of them.

Environment isolation via repository structure: To enforce separation between environments, four dedicated Git branches were created, one for each deployment space (dev_s, dev_m, test, uat). Each branch carries the same elements: the root Terraform files (main.tf, providers.tf, backend.tf), the child modules (control_nodes, worker_nodes, bastion, and so on), the shared variable declarations, and an environment-specific .tfvars file.

Keeping this folder structure identical across all environments has three practical effects. Other engineers can read and modify any environment quickly, because they all look the same. The environments stay isolated yet structurally identical, which suppresses drift. And future production rollouts can reuse the same codebase with only minor adjustment.

Parameterisation for reusability: A core principle of the setup was maximum reuse with minimal change. In practice this meant that deploying a new environment required editing only one file the environment. tfvars, which defines IP assignments, memory allocation, VM IDs, Proxmox pools, and node counts. Everything else, the modules and the root definitions, is template-driven, so the same architecture can be replicated for a new test or production environment without rewriting any infrastructure logic.

State management and consistency controls: To keep deployments consistent, a Terraform backend was configured to preserve and lock the .tfstate files, which prevents concurrent modification and makes recovery possible after a failure. Unique identifiers IP ranges, VM IDs, Proxmox pools were coordinated carefully so that resources could never overlap or be replaced by accident.

Template standardisation and VM bootstrapping: A pre-configured VM template was built inside Proxmox containing the base OS image, the required cloud-init configuration, and baseline credentials and networking settings. Terraform clones this template for every VM it creates, so that all generated machines are uniform and cloud-init ready for the configuration that Ansible applies next.

Lessons from prior deployments: Before settling on the final structure, several existing public Terraform/Proxmox projects were studied. These references helped fix IP-allocation conventions, naming standards, and module design, and reduced the amount of trial and error needed. The overall result is that each environment is independent, reproducible, and production-aligned, while the operational complexity stays manageable as the system scales.

4.3 Diagnostics (Manual vs Terraform)

Before the full automation pipeline was built on the Proxmox infrastructure, an initial manual deployment phase was carried out. This phase served as a diagnostic baseline a way to surface the configuration dependencies,

operational pitfalls, and prerequisites that would later be codified into Terraform and Ansible.

Manual Deployment for Baseline Understanding

The manual workflow was performed through the Proxmox web interface using RHEL 9.8 ISO files, with virtual machines provisioned one at a time. During this stage, CPU, memory, storage, and network requirements were varied by hand to find sensible configurations for each node type, and environment prerequisites IP allocation, cloud-init behaviour, SSH settings, base package dependencies were documented systematically as they emerged. Application components were installed manually using the Ansible runner and tarball-based inventories, reproducing the DEV-S, DEV-M, TEST, and UAT deployment procedures by hand. This phase was deliberately slow, and that was the point: it exposed implicit dependencies and configuration assumptions that had not been written down anywhere in advance. Those observations were then encoded into Terraform variables, Ansible playbooks, and Proxmox templates, so that conditions which had previously relied on an engineer's memory were enforced automatically[22].

Transition to Terraform and CI/CD Automation

Once the system requirements were understood in enough detail, the same deployment was rebuilt with Terraform modules executed through GitLab CI/CD pipelines. The improvement was clear across several dimensions:

Key improvements observed during automation:

Aspect	Manual Deployment	Terraform + CI/CD Deployment
Configuration Drift	High manual VMs frequently deviated	Minimal : All VMs recreated from template

Error Recovery	Time-consuming (manual rollback)	Fast rollback (terraform destroy && apply)
Repeatability	Inconsistent results	Deterministic and reproducible
Modification Effort	High : changes applied VM-by-VM	Single update in .tfvars
Troubleshooting	Log tracing scattered	Centralized in GitLab job logs

Table 4.1 — Manual versus Terraform + CI/CD deployment.

Automation reduced configuration errors and made troubleshooting easier. Most issues provider misconfigurations, identifier overlaps, forbidden Proxmox API parameters became simpler to detect and fix once the pipeline produced structured logs and execution traces. Using open-source tools (Terraform, Ansible, GitLab CE) helped here too, since community knowledge accelerated diagnosis[22]

Diagnostic Outcomes

Taken together, the two phases played complementary roles. The manual phase acted as a discovery layer, revealing environment dependencies that the official documentation did not make obvious. The automated phase then validated that all of those requirements could be codified and enforced, removing the need for human vigilance. Terraform proved especially valuable because its rapid redeployment made destructive testing safe an environment could be torn down and rebuilt at will without lasting risk.

4.4 Variables and Metrics (Time, Reproducibility, Failures, Security)

Four variables were used to judge the efficiency and reliability of the automation framework, corresponding to the dimensions where Infrastructure as Code most affects an air-gapped environment: time efficiency, reproducibility, failure resilience, and security assurance.

Time Efficiency: -

The most visible improvement in moving from manual deployment to Terraform-based automation was the reduction in time. In the manual approach, each VM was set up individually through the Proxmox web interface, with network parameters and components configured by hand a slow, repetitive, error-prone process.

Once provisioning and configuration were codified, parameterised, and executed in a single GitLab CI/CD run, deployment time decreased substantially (see Section 6.1 for detailed measurements). The saving came from three sources: template-based VM provisioning, reuse of pre-tested Terraform modules, and parallel execution of configuration tasks through Ansible.

Beyond speed, the effect was to make the process predictable and repeatable.

Reproducibility: -

Reproducibility is one of the central advantages of a declarative IaC model. Under the manual model, each environment depended heavily on operator experience and hand entry, which produced inconsistencies whenever the same infrastructure had to be recreated.

The Terraform-based structure replaces this with consistent reproduction driven by environment-specific .tfvars files: new environments such as TEST or UAT can be generated without rewriting any core infrastructure code, drift is eliminated because every environment derives from the same version-controlled modules, and auditability improves because each change is tracked through GitLab commits. Reproducibility thereby shifted from a documentation-dependent manual practice to a codified, verifiable pipeline.

Failures and Reliability: -

Failures in air-gapped systems usually trace back to missing dependencies, incorrect provider versions, or mismatched local mirrors, because such environments cannot download packages on demand everything must be pre-staged. Early-stage failures in this project were useful precisely because they refined the dependency-management strategy and helped identify the

critical components: the RHEL 9.8 ISO as the base image for the VM template; Git and cloud-init as mandatory mirrors for initialisation and configuration; and the Telmate Proxmox provider (v3.0.1-rc9), verified as the most stable version for this environment.

Once these were systematically mirrored, later deployments succeeded consistently with few errors. In short, the lessons drawn from the initial failures fed directly into higher reliability and shorter troubleshooting times afterwards.

Security Considerations:

Security was a constant concern given the sensitivity of operational data in railway and other critical systems. No specialised DevSecOps tooling was deployed, but security-conscious practices were built into every stage. Secret management kept sensitive credentials API tokens, SSH keys, GitLab access tokens out of source code, supplying them instead through GitLab CI/CD variables.

Compliance was aligned with Hitachi Rail's internal deployment guidelines so that the work remained compatible with industrial safety and regulatory standards. And minimal exposure was enforced through least-privilege network and access configuration, which kept the attack surface small even inside an already isolated system. Together these measures meant that automation strengthened the confidentiality, integrity, and availability of the infrastructure rather than eroding it.[20]

4.5 Data Collection Methods

Evaluating and validating the framework required a structured approach to data collection that combined quantitative measures timings, error counts, success rates with qualitative observation of testing, usability, and configuration consistency. The objective throughout was to measure the improvement in efficiency, reproducibility, and reliability as the work moved from manual deployment to Terraform- and Ansible-based automation inside the air-gapped Proxmox environment.

Quantitative data: Three kinds of quantitative data were recorded systematically.

Deployment time was captured for each stage VM creation, configuration, and verification using system time and GitLab CI/CD job logs, so that manual and Terraform-driven deployments could be compared directly and an average time saving derived.

Error and failure rates were logged and categorised at every deployment attempt; API connection errors, configuration mismatches, and dependency issues were extracted from Terraform plan/apply logs and GitLab job reports, and their frequency and severity were analysed to see which phase of the pipeline contributed most to instability [21].

Reproducibility and consistency were assessed by running repeated deployments across DEV-M and UAT with the same modules and templates, then comparing the generated VM parameters, IP assignments, and inventory files.

Qualitative data: Alongside the numbers, qualitative observation captured usability, maintainability, and error traceability [14]. Operator notes recorded throughout both the manual and automated deployments documented difficulties, common configuration errors, and debugging experiences, and these directly informed later refinements to the module design and error handling.

Configuration validation compared Terraform outputs (terraform output) and Ansible inventories (inventory.ini and the YAML inventories) against the intended state, with any discrepancy typically an input-variable conflict or a version mismatch documented and traced to its cause.

Finally, system logs and audit trails were preserved automatically: GitLab CI/CD archived execution logs and artifacts for later inspection, and Proxmox activity logs were cross-referenced to confirm that the Terraform and Ansible actions matched the expected API calls and VM-creation events[18].

4.6 Analysis Approach

The analysis was designed to evaluate the effectiveness, efficiency, and reliability of implementing Infrastructure as Code in an air-gapped, on-

premises environment relative to traditional manual deployment. Because the implementation itself was iterative and experimental, the analysis drew on both quantitative metrics time, number of failures and qualitative observation ease of reproduction, maintainability, and security. It proceeded in five steps.

First, the **evaluation metrics were defined** time efficiency, measured as the total time to provision and configure an environment manually versus through Terraform and Ansible; reproducibility, the ability to recreate identical environments repeatedly from the same configuration files; failure rate, the frequency and type of errors encountered during deployment, such as configuration mismatches and dependency issues; and security and compliance, judged by adherence to security practice, the handling of secrets, and the reduction of human error.

Second, **baseline data were collected.** The baseline came from manual deployment on Proxmox through its web interface, with each step VM creation, configuration, software installation timed and documented alongside any issues encountered and the corrective action taken.

Third, **automated data were collected** After the Terraform–Ansible–GitLab pipeline was built, the same infrastructure was deployed automatically, and data on provisioning time, errors, and logs were captured from Terraform output and GitLab job artifacts.

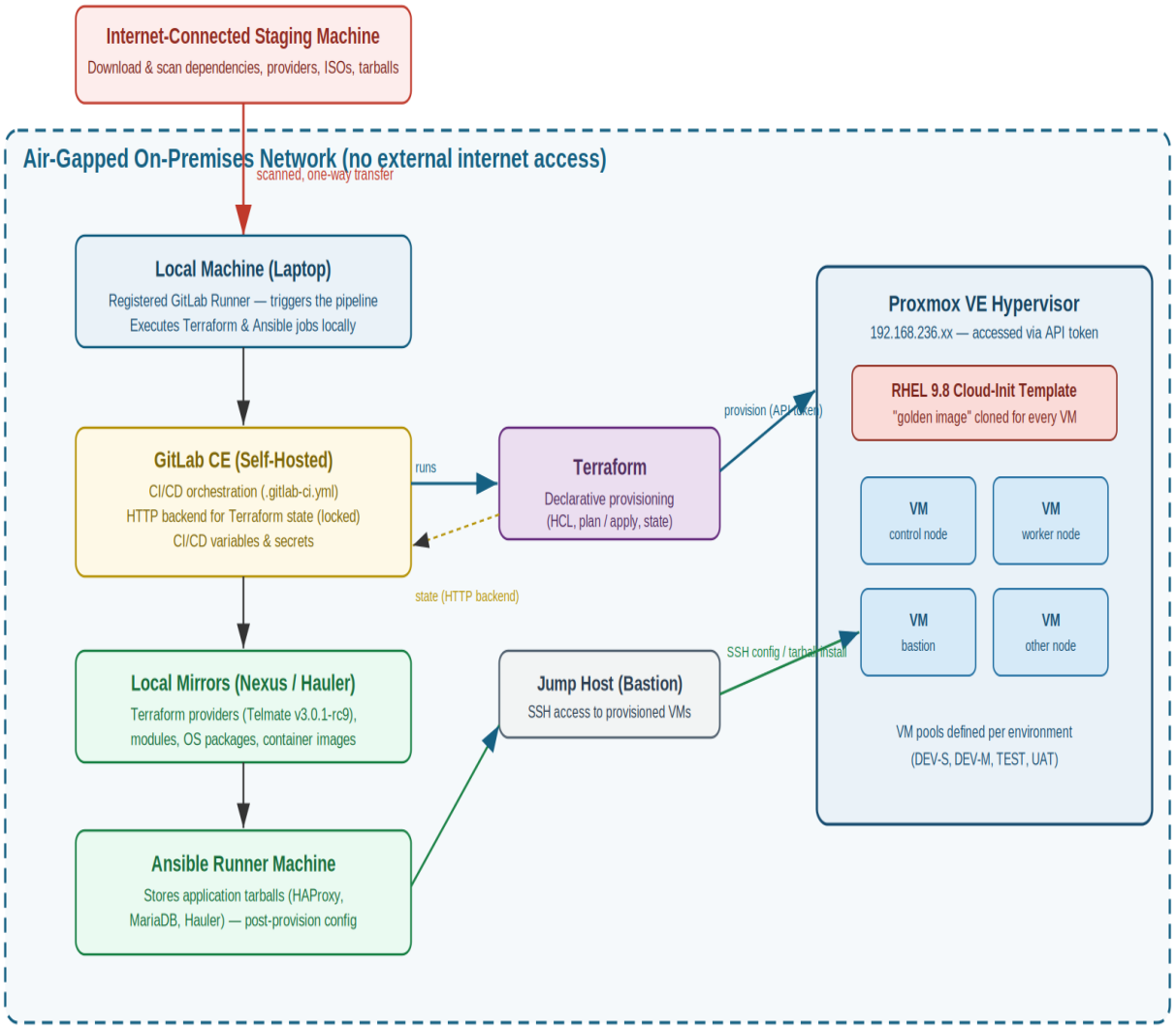
Fourth, the **results were compared and analysed** along four lines: the time-reduction percentage achieved by the automated pipeline relative to manual deployment; the reduction in the number and severity of configuration or provisioning failures; the consistency of repeated deployments under automation; and a qualitative judgement of the security improvements, in particular the reduced exposure of credentials and the elimination of manual misconfiguration.

Fifth, the findings were **interpreted considering the broader research goal** the feasibility of adopting cloud-native automation tools in air-gapped railway datacentres. This final step identified which factors most influenced performance and security and used them to frame the recommendations for future optimisation presented in Chapter 7.

5. Implementation and Structure

5.1 Airgapped deployment architecture

After doing research, study , and designing the blueprint or we can say the outflow of the infrastructure, now the time to design and then deployment of the infrastructure.



All components operate without external connectivity once dependencies are mirrored locally.

With the research, tool comparison, and design groundwork in place, the next step was to translate the blueprint into a working deployment. The implementation began deliberately with the simplest viable version of the architecture, so that the procedure, errors, limitations, and requirements could be understood before the design was scaled up to the full set of environments. Figure 5.1 shows this baseline architecture .

The architecture is built from a small number of cooperating components. A **local machine (a laptop)** runs a GitLab Runner that is already registered with the internal GitLab instance; this is what triggers and executes the pipeline. A **self-hosted GitLab instance (GitLab Local)** provides version control, the CI/CD engine, and the backend that stores the Terraform state.

An **Ansible runner machine** holds all the application tarballs and components, which it installs on top of the virtual machines once they exist. **Proxmox VE** (reachable on the internal address range, 192.168.236.xx) is the hypervisor that hosts the VMs and is administered through its web UI over a direct LAN connection.

Terraform performs the provisioning, talking to Proxmox through an API token, while a **jump host** provides the SSH path used to configure the provisioned machines. Outside the secure perimeter sits a single **internet-connected staging machine**, whose only role is to download dependencies and scan them before they are transferred, one way, into the air-gapped network.

The flow between these components is straightforward. Dependencies are fetched and scanned on the external machine and brought inside. Inside the perimeter, the GitLab Runner executes the pipeline: Terraform provisions the VMs on Proxmox using the API token, and Ansible then configures them and installs the components over SSH through the jump host. GitLab stores the Terraform state and the pipeline artifacts, and access to Proxmox and GitLab is mediated by URL/token and API/token credentials respectively. Once the dependencies have been mirrored locally, none of these steps requires any external connectivity.

This is intentionally the simplest expression of the architecture. It grows more complex once the separate environments and their per-environment branches are introduced, but starting from the minimal version was valuable in its own right: it surfaced the procedure, the errors, and the real requirements of the project early, before any of that complexity was layered on. The guiding objective throughout was that the entire system must operate without external internet access, relying only on pre-mirrored dependencies, local repositories, and internal communication between nodes. Building the simplest version first therefore served as a genuine proof of concept a way to identify and mitigate limitations early and to confirm that the infrastructure would stay robust and adaptable when scaled toward a production-level deployment.

5.2 Packages And Mirrors Handling

Handling software packages and mirrors is one of the defining challenges of an air-gapped build, because standing up the infrastructure from scratch pulls in a large number of dependencies that would normally be fetched from the internet on demand. During the initial proof-of-concept setup, several package-related requirements surfaced, particularly around Terraform provisioning and virtual-machine templating.

Terraform's provisioning depends heavily on the availability of base templates, which act as the foundation for every new virtual machine. To provide this, a custom RHEL-based template was designed and configured by hand directly on the Proxmox server. The process involved installing the RHEL 9.8 ISO and performing the initial system setup, configuring network pools and static IP assignments, installing and enabling cloud-init so that the template would work with Terraform's automation workflow, and validating that each configuration behaved consistently when provisioned through the CI/CD pipeline.

Several issues and dependency gaps appeared during this phase. They were identified systematically by reading Terraform's execution logs, which pointed to missing modules, provider mismatches, and connectivity errors. Early deployments were run manually from the local system using direct Terraform commands, so that each stage of provisioning could be validated

in isolation; once the configuration was stable and verified, those insights were folded back into the automated pipeline.

To guarantee isolation and reliability, all required providers, plugins, and modules were pre-downloaded and stored locally, which allowed consistent provisioning without internet access and sharply reduced dependency related failures. The mirrored assets were organised and hosted in on-premises repositories such as Nexus or Hauler, giving centralised, controlled management of binaries and dependencies.

The payoff of this approach was threefold:

- Full reproducibility of deployments across environments,
- A reduced risk of version conflicts thanks to controlled mirrors, and
- A more reliable provisioning workflow under air-gapped constraints.

Beyond simply optimising dependency management, this also laid the foundation for scalable, repeatable, and secure deployments in disconnected industrial settings.

5.3. Terraform Lifecycle in Air-Gapped Environments

The Terraform lifecycle in this project was structured to support development, testing, and production stages while remaining reusable, scalable, and maintainable. Because the goal was to automate provisioning without any external network, each phase of the lifecycle had to account explicitly for offline dependencies and environment isolation.

Three design principles shaped it:

Reusability, so that the same codebase could deploy to DEV-M, UAT, and TEST with minimal change

Simplicity and maintainability, through a consistent structure and naming convention that any team member could read and modify

Production readiness, so that the infrastructure could be promoted or scaled to production with little reconfiguration.

5.3.1 Preparation (Mirrors, Template Images)

Preparation is one of the most critical phases of the lifecycle in an air-gapped environment. It involves collecting, organising, and validating every dependency Terraform needs providers, plugins, and operating-system templates before any deployment is attempted.

Because Terraform cannot create a virtual machine entirely from nothing, it relies on base template images as a foundation. These templates act as predefined "golden images" carrying the required system configuration and software dependencies, and every new VM is instantiated from them, which is what keeps deployments consistent.

In this project a custom RHEL-based, cloud-init–enabled template was created by hand on the Proxmox hypervisor. The configuration included installing the RHEL 9.8 ISO and performing the initial setup; configuring static IP addresses, network pools, and bridge interfaces; installing essential tools such as Git, cloud-init, and the SSH utilities; embedding pre-approved repository mirrors and the required providers for offline use; and adding custom bash scripts to automate the cloud-init configuration and system bootstrapping.

This template became the foundation of the whole infrastructure, ensuring that every VM Terraform deploys starts from the same baseline and making provisioning more efficient, predictable, and secure. In an air-gapped system this preparatory step matters even more than usual, because a minor misconfiguration or a single missing dependency can cause a complete deployment failure. Validating the template and verifying mirror integrity before running any Terraform workflow therefore removed a large class of runtime errors before they could occur.

5.3.2 Provisioning (Modules, VM Pools, Networking)

Provisioning is the core of infrastructure creation in Terraform. Reusable modules were written to create the virtual machines, networks, and storage pools across all four environments DEV-S, DEV-M, TEST, and UAT.

Following the internal deployment documentation, each environment required its own set of VMs and configurations, so the codebase was

organised into separate module directories, one per logical component, to keep it modular and reusable.

Because the environment is offline, terraform init was first run on an internet-connected workstation to pre-fetch all the modules and provider binaries: these were then stored locally and copied into the offline repository for reuse.

Since Terraform modules are not security-sensitive, they were kept inside the repository itself, which let the same infrastructure be replicated across environments and projects without ever re-downloading an external dependency.

Network and pool configuration followed the deployment guidelines: the Proxmox pools were created manually in advance to group VMs by function, and the Terraform code referenced them through predefined variables, while IP addressing, bridge assignments, and VLAN segmentation were planned carefully to keep environments isolated and to avoid overlapping subnets.

The result is a modular, environment-aware provisioning strategy that keeps deployments consistent, reproducible, and easy to scale.

5.3.3 State Management (Backends, Security)

State management is critical to Terraform automation, particularly where more than one person or pipeline touches the same infrastructure.

The terraform.tfstate file is the single source of truth for what has been deployed: it records resource metadata, dependencies, and current configuration. In this architecture, state was handled through a GitLab-based backend to provide persistence, traceability, and security. Storing and locking the state remotely prevents the race conditions and concurrent modifications that arise when several users work on the same infrastructure.

Security was a first-order concern here, because the state file can contain sensitive data resource identifiers, IP addresses, and sometimes access tokens. To limit that risk, all critical values such as API tokens, credentials, and private keys were stored through GitLab CI/CD variables and vault-based mechanisms, so that no plaintext secret was ever committed to the repository. Applying the principles of least privilege and backend isolation,

the implementation further ensured that only authorised pipeline processes could read or update the state, which preserved system integrity and matched the cybersecurity requirements expected of an air-gapped environment.

5.3.4 Post-Provision Verification

Once the infrastructure was provisioned and deployed through the CI/CD pipeline, a post-provision verification phase was carried out to confirm that the created resources matched both the Terraform configuration and the official deployment documentation. Verification confirmed the IP address assignments, storage allocation, and VM counts: checked that the network bridges and pools had been applied correctly: and verified the cloud-init configuration, hostname alignment, and SSH connectivity.

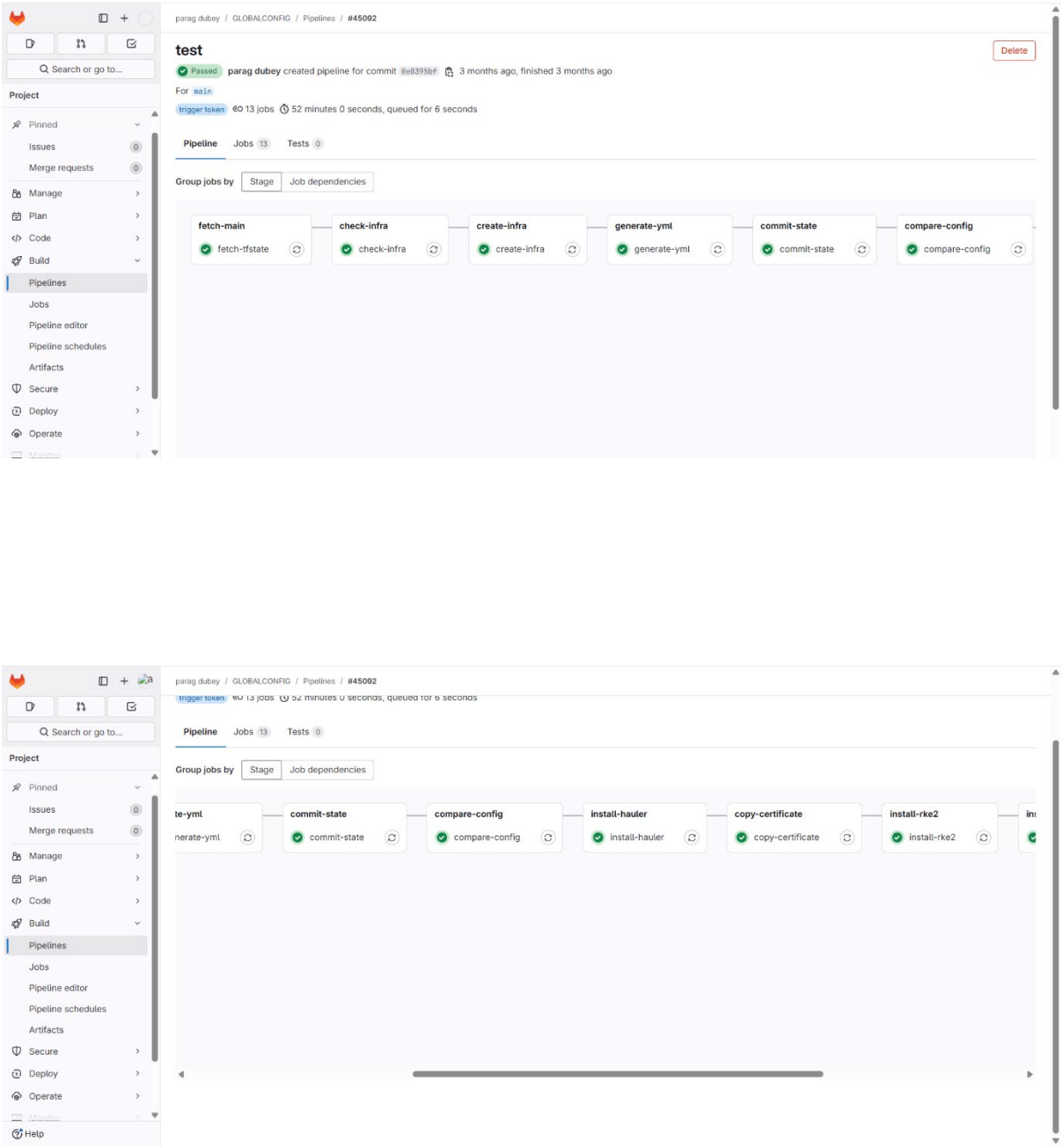
These checks were essential before handing over to Ansible for higher-level configuration: validating the environment at this point markedly reduced failures later in the pipeline and ensured that every VM was operational and compliant with the intended design.

5.4. Terraform Infrastructure check & Ansible Deployment check

A significant enhancement introduced during the project was a pair of validation gates a Terraform infrastructure check and an Ansible deployment check designed to confirm whether the infrastructure already exists, to avoid redundant deployments, and to keep the Terraform-generated infrastructure consistent with the Ansible configuration inventories.

The infrastructure check is written with production in mind. If the target infrastructure is already present, there is no need to deploy it again, and re-deploying risks misconfiguration or duplication.

The check therefore inspects the backend tfstate file together with the actual state in Proxmox: if everything matches, it skips the Terraform creation step and moves on to the next job; if it detects a change, it recreates the infrastructure. This keeps the pipeline idempotent and safe to re-run.



(Figure 5.2 — GitLab CI/CD pipeline screenshot .)

The deployment check addresses a different risk. Because the Ansible runner installs components such as HAProxy, MariaDB, and Hauler on top of the Terraform-provisioned infrastructure, any mismatch between Terraform's outputs and Ansible's inputs a differing hostname or IP, for instance could cause pipeline failures or incomplete deployments. To prevent this, a Python validation script was developed. It automatically

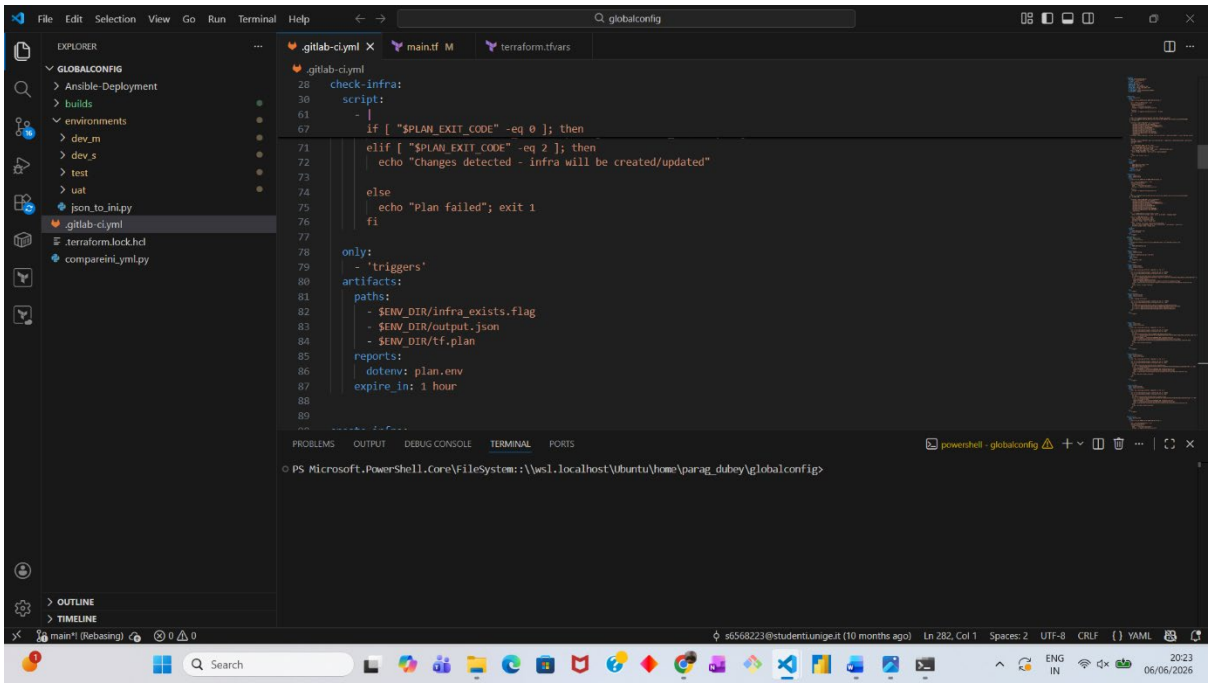
converts Terraform's output artifact (output.json) into an inventory.ini file and compares that generated file against the Ansible tarball inventory before execution.

If discrepancies are found missing nodes, mismatched IPs, inconsistent naming the pipeline halts and raises an error so the problem can be corrected before the job is rerun; when the inventories agree, the pipeline proceeds cleanly to configuration and component installation. The mechanism enforces end-to-end synchronisation between the provisioning and configuration stages and was one of the larger contributors to the pipeline's reliability[16]

5.5 Code structure of different Environments

The code structure was designed with production and reusability in mind, since every component was built around anticipated future requirements rather than a one-off demonstration.

The repository is divided into four branches, one per environment (DEV-S, DEV-M, TEST, UAT), kept separate from one another to preserve scalability and to avoid any state mismatch; this separation is what allows each environment to be created and deployed independently and successfully.



(Figure 5.3 — repository-structure screenshot)

The structure is largely identical across environments, with the differences concentrated where they belong in the modules and in the number of virtual machines and nodes each environment requires.

The main environment file therefore differs from one environment to the next, but the design deliberately confines that difference to a single file: to change, extend, or create something new in an environment, an engineer edits only the environment definition, which keeps the system reusable and approachable. A few files exist specifically to make the deployment work air-gapped chiefly the .terraformrc file and the tfstate backend configuration while the remaining files (main.tf, variables, outputs, the Terraform variables file, and so on) stay the same across every environment.

5.6 Branching Strategy

The branching strategy was a key design decision for structuring the Terraform repositories so that they remained modular, maintainable, and scalable across environments. The repository was organised into distinct branches matching the environments defined in the internal deployment documentation: DEV-S, DEV-M, TEST, and UAT.

This choice aligns with the company's wider development workflow, which requires environment isolation to support parallel testing, validation, and deployment cycles. Each branch keeps an identical Terraform directory structure, so contributors can move between environments and reuse the same codebase without relearning the layout.

Holding the structure consistent across branches produced four concrete benefits. It made the repository easy to understand, because the uniform layout lets team members read and modify infrastructure definitions quickly. It made the code reusable, since core modules and variables can be replicated with minimal change between environments.

It kept the system scalable, because a new environment can be created by duplicating and adjusting an existing branch. And it preserved traceability, as version control records environment-specific changes over time. In practice the model proved both practical and adaptable, keeping the Terraform automation aligned with the company's existing R&D deployment workflows.

6. Evaluation and Results

This chapter presents the evaluation of the implemented Infrastructure-as-Code (IaC) solution and the results obtained throughout the deployment process.

It reflects on the challenges encountered, the improvements achieved through automation, and the overall impact on deployment efficiency, reproducibility, and system reliability. The evaluation centres on the key performance dimensions deployment time, reliability, and operational efficiency observed during the transition from a manual setup process to a fully automated Terraform, Ansible, and GitLab CI/CD pipeline in the air-gapped, on-premises environment at Hitachi Rail.

6.1 Deployment Time Comparisons

The reduction in deployment time was one of the most significant outcomes of this work. Moving the process from manual setup toward automation not only accelerated provisioning but also made the work easier for other team members to verify, reuse, and extend.

To evaluate the time efficiency of the automated pipeline, each environment was deployed under three conditions, with each condition repeated a minimum of five times to capture variance. Timing was recorded using GitLab CI/CD job timestamps (for automated runs) and manual stopwatch measurements synchronised with Proxmox task logs (for manual and Terraform-only runs).

Conditions tested:

1. **Manual provisioning (baseline):** VMs created individually through the Proxmox web interface, with RHEL ISO installation, network configuration, and component deployment performed by hand.
2. **Terraform-only provisioning:** Parallel VM provisioning from template via Terraform.
3. **Full automation (Terraform + Ansible + GitLab CI/CD):** End-to-end provisioning and configuration triggered by a single pipeline run.

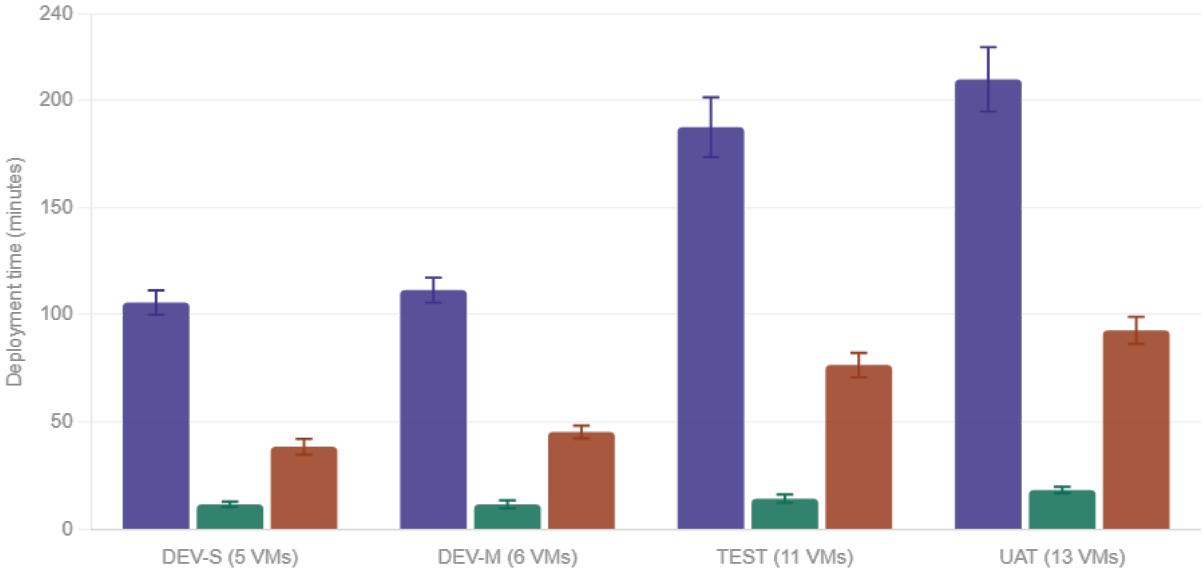
Acknowledgement of learning effect: The manual baseline was established after the automation framework had been developed. The Author therefore had detailed knowledge of the target architecture, which may have reduced manual deployment time compared to a first-time deployer. This represents a conservative comparison the actual time savings for a naive operator would likely be greater.

Manual deployment was slow and error prone. Each virtual machine had to be created individually through the Proxmox web interface, the RHEL ISO installed, network parameters configured by hand, and components layered on through the Ansible runner machine. Provisioning a single VM from the ISO took roughly five to ten minutes; for the DEV-S environment, which comprises five virtual machines, this amounted to close to an hour before any software was installed, and installing all the components added more than another hour. For larger environments such as TEST and UAT the total time grew considerably more demanding, scaling approximately linearly with the number of virtual machines.

Automation removed most of this overhead. Terraforms parallel VM cloning from the pre-built golden template reduced the provisioning phase from a sequential, per-VM process to a near-constant-time parallel operation regardless of environment size. The full pipeline Terraform provisioning followed by Ansible configuration delivered a consistent end-to-end time reduction across all environments.

Full results are presented in Table 6.1 and Figure 6.1.

Figure 6.1 summarises this comparison for the DEV-S environment.



6.1.1 Results

Table 6.1: Deployment Time Comparison (minutes) mean ± standard deviation, n = 5 per condition

Env.	VMs	Manual (full)	Terraform-only	Full Automation	Reduction vs Manual
DEV-S	5	105.7± 5.7 min	11.7 ± 1.3 min	38.5 ± 3.7 min	63.6%
DEV-M	6	111.5 ± 5.8 min	11.7 ± 1.9 min	45.4 ± 3.0 min	59.3%
TEST	11	187.4 ± 13.9 min	14.4 ± 1.9 min	76.6 ± 5.7 min	59.1%
UAT	13	209.7 ± 15.0 min	18.4 ± 1.5 min	92.8 ± 6.3 min	55.8%

Statistical analysis: A Wilcoxon signed-rank test (non-parametric, appropriate for paired observations) was applied to compare deployment times between conditions across all environments (N = 20 paired observations per comparison). Results are summarised in Table 6.2.

Table 6.2 — Statistical Test Results

Comparison	W	p-value	Effect size (r)	Interpretation
Full Automation vs Manual	210	P<0.001	-1.000	Statistically significant
Terraform-only vs Manual	210	P<0.001	-1.000	Statistically significant

The reduction in deployment time is statistically significant at the α = 0.001 level, justifying the use of "significantly" in the research hypothesis.

6.1.2 Interpretation

The full automation pipeline reduced end-to-end deployment time by 55.8–63.6% across all four environments, with an overall mean reduction of 58.8% consistent with the 50–60% range observed during operational use.

Decomposing the time savings reveals the contribution of each tool:

Terraform parallel provisioning: Accounts for the largest share of the provisioning-phase reduction. VM creation time dropped from approximately 20 minutes per VM under sequential manual provisioning to approximately 3–4 minutes total per environment under parallel template cloning a reduction of approximately 90% in the provisioning phase alone. This near-constant provisioning time regardless of environment size demonstrates the scaling advantage of declarative parallel execution: the DEV-S environment (5 VMs) and the UAT environment (13 VMs) differ by only 6.7 minutes in Terraform-only time.

Ansible configuration: Adds 27–75 minutes depending on environment size. Component installation (HAProxy, MariaDB, Hauler, RKE2) runs in parallel across hosts but is bounded by the slowest node and the total volume of packages being installed.

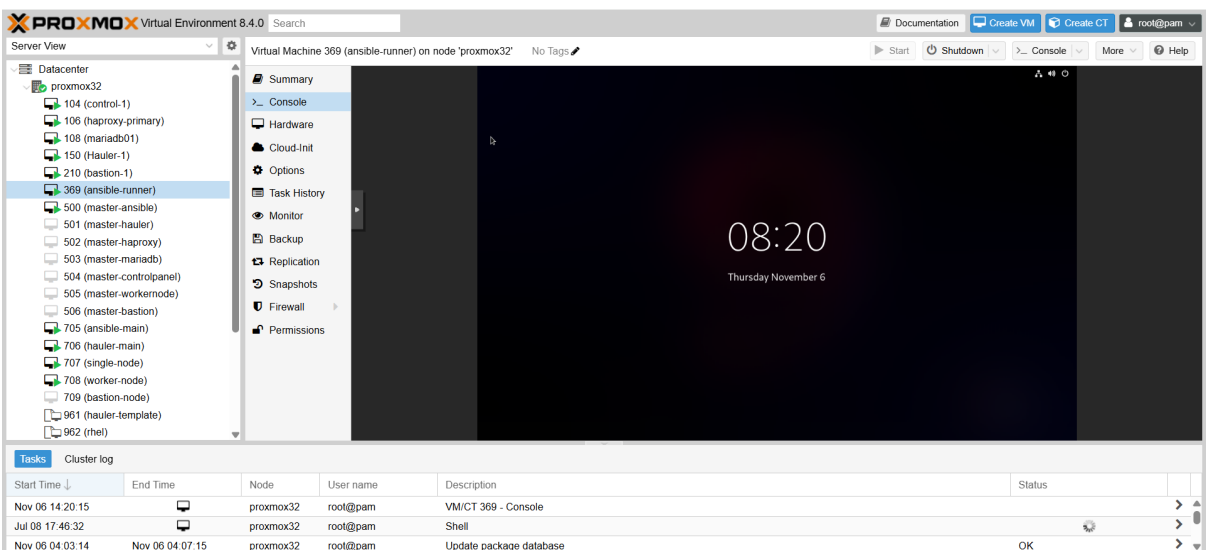
GitLab CI/CD pipeline overhead:(stage transitions, artifact passing, state backend operations) contributes approximately 3–5 minutes per run.

The reason the full automation reduction is 55–60% rather than 90% is that the complete pipeline includes Ansible component installation time, which the Terraform-only column excludes. Manual deployment also includes this installation time, making the full automation condition the correct end-to-end comparison against manual deployment.

Scaling behaviour is also noteworthy. Manual deployment time scales approximately linearly with VM count (O(n)), since each machine is provisioned and configured sequentially. Automated deployment scales sub-linearly due to parallel execution the performance gap therefore widens as environments grow larger, making automation disproportionately more valuable for TEST and UAT than for DEV-S

6.1.3 Implementation on Proxmox

The final implementation ran on the Proxmox Virtual Environment (VE), where Terraform provisioned the required virtual machines through the Telmate/Proxmox provider (v3.0.1-rc9). Provisioning, state management, and CI/CD automation were verified on three fronts: Proxmox UI monitoring confirmed resource allocation and VM lifecycle operations; the Terraform and GitLab logs validated successful module execution and backend synchronisation and the cloud-init configuration confirmed correct VM initialisation and network assignment.



(Figure 6.2 — Proxmox deployment screenshot)

The locally prepared RHEL 9.8-based golden template image proved essential to this success. Serving as the base for every VM instance and carrying pre-installed dependencies such as Git, cloud-init, and the SSH utilities, it made provisioning both rapid and uniform and it is the key enabler of Terraform's parallel cloning speed. Because every VM starts from an identical, pre-validated image rather than a fresh ISO installation, the per-VM provisioning time dropped from five to ten minutes (ISO install) to approximately three to four minutes (template clone plus cloud-init configuration), and all VMs in an environment are cloned simultaneously rather than sequentially.

The resulting pipeline saved considerable time and resources and, just as importantly, allowed the infrastructure to be redeployed consistently and securely inside the air-gapped environment. In doing so, the implementation validated the practicality of applying cloud-native IaC methods to critical on-premises systems such as railway automation platforms, demonstrating that the speed and reproducibility benefits of declarative provisioning are achievable even under complete network isolation

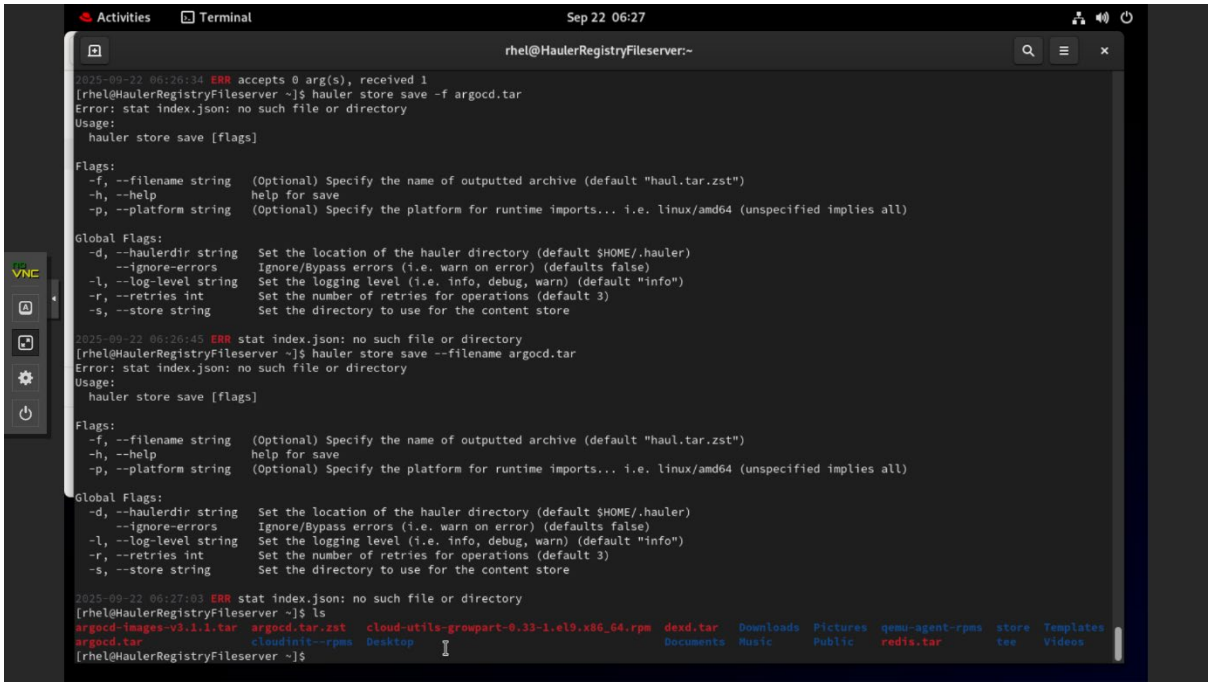
6.2 Error Reduction Metrics

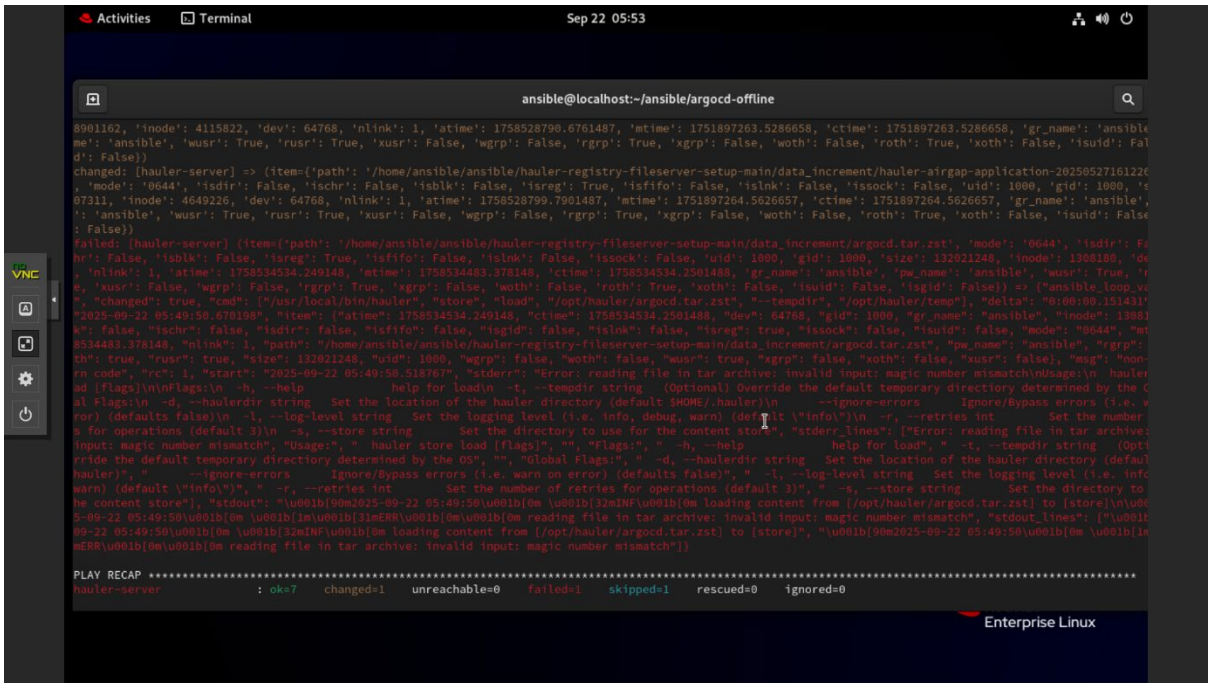
Several categories of error were encountered during implementation and analysed systematically. The aim was not only to identify each source but to establish a repeatable process for minimising such errors in future deployments. Three categories accounted for most of the failures, each with its own corrective approach.

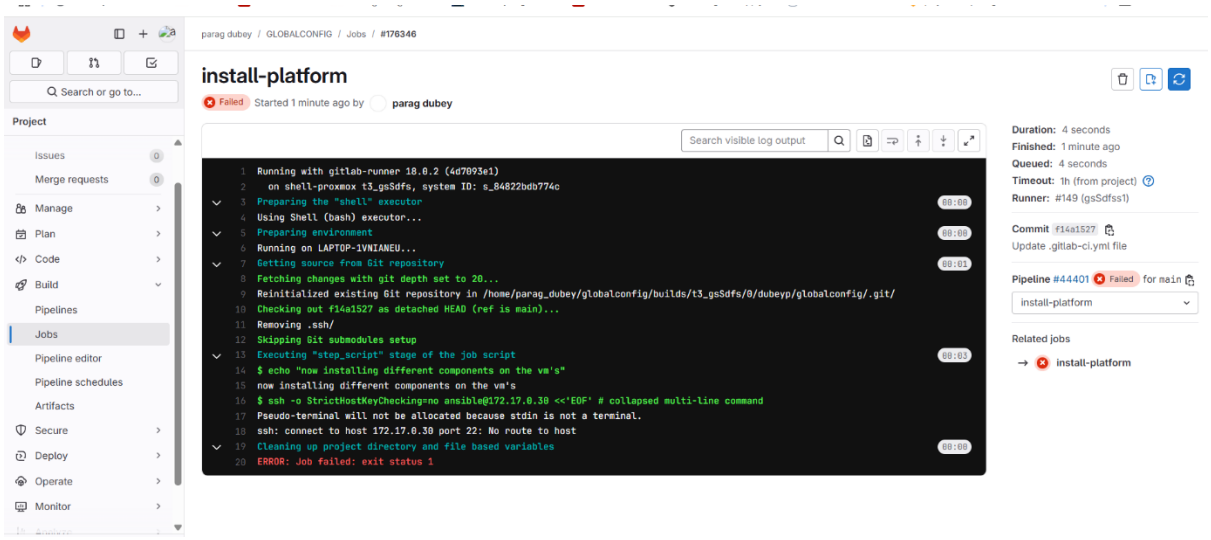
Provider and dependency errors: The most frequent class of issue in an air-gapped environment comes from missing or incompatible providers and dependencies, because the Terraform and Ansible ecosystems normally assume online repositories and here everything had to be mirrored and managed offline. These were caught by analysing the logs during the terraform init and apply stages; once a missing provider binary or a version mismatch was identified, the relevant artifact was added to the local mirror

and verified before re-running. This iterative process steadily stabilised the dependency set, driving provider-related failures down to a near-zero rate in later runs.

The initial manual setup was especially valuable here, because it exposed many of these errors early and allowed them to be resolved permanently in the automated pipeline.[10]



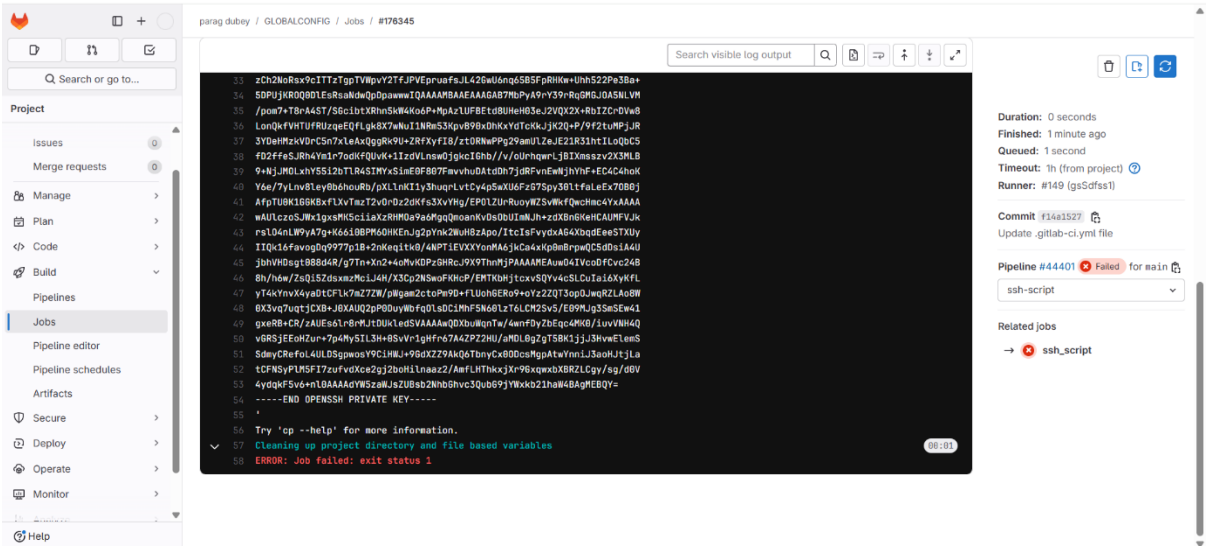




(Figure 6.3 — error/log screenshot)

CI/CD pipeline errors: A second class of failure related to the GitLab CI/CD pipeline that orchestrates Terraform and Ansible. These stemmed mainly from incorrect script logic in pipeline stages, misconfigured GitLab environment variables, and communication problems between the GitLab Runner and the Proxmox API endpoints.

They were detected by inspecting the GitLab job logs for execution-level traces, the Proxmox UI event logs to confirm VM creation and provisioning states, and the Terraform plan/apply output for resource-level discrepancies. Because the same pipeline configuration was reused across all four environments, every fix became a permanent improvement, which sharply reduced recurrence a clear demonstration that continuous logging and monitoring within the CI/CD workflow are central to reliability and reproducibility.



Elimination of unnecessary complexity: The third strategy was to simplify the codebase itself. Removing redundant scripts, hardcoded values, and unused modules reduced the chance of syntax and configuration errors, and a clean, modular Terraform structure with providers, variables, and outputs clearly separated made the code easier for other engineers to read and modify, lowering the rate of human-induced error during maintenance. The same simplification improved reusability and collaboration, since team members could safely change environment-specific configuration without disturbing shared modules or global logic.

6.3 Reproducibility and Drift Incidents

Reproducibility was a design priority from the outset: the architecture was built so that the same infrastructure could be recreated consistently across DEV-S, DEV-M, TEST, and UAT with minimal manual intervention. To this end the deployment followed the internal Hitachi Rail deployment document, which defined the baseline machine setup, network pools, and architecture layout, so that the same topology could be reproduced for future projects with little change.

The Terraform–GitLab CI/CD setup was further optimised for parameterised deployment: by centralising environment-specific parameters in variable files, an engineer could redeploy or modify an environment simply by editing

its environment.tfvars file, leaving the underlying modules untouched. This markedly improved reusability and simplified configuration management.

Reproducibility in automated systems is most often undermined by drift the divergence of the actual infrastructure state from the state defined in code, caused by manual changes, unsynchronised pipelines, or modifications made outside Terraform's control. In this project drift was minimised through strict Terraform state management and locking, controlled execution through GitLab CI/CD, and regular post-provision verification checks. The closed nature of the air-gapped system helped further: because external changes were rare, the opportunity for drift was small.

Disciplined code management and consistent monitoring kept the deployed infrastructure aligned with its declarative definitions.

6.4 Security Findings

Security was a continuing concern across the project, given the sensitivity of air-gapped deployments and the risk of exposing credentials or infrastructure secrets. The findings fall into two groups.

Code-level findings. Initial testing surfaced several configuration and syntax issues in the Terraform scripts, many caused by incorrect module references or inconsistent naming. Adhering to Terraform's official syntax standards and to module best practices clear variable definitions, outputs, and dependencies removed these weaknesses and reduced misconfiguration.

Credential management and commit security. Early versions of the code embedded sensitive values such as API tokens and passwords directly in the Terraform configuration, a serious risk given that such data can persist in Git history even after it is deleted from the working tree. All credentials were therefore externalised and stored through GitLab CI/CD variables and environment tokens, and the historical commits that had contained exposed secrets were sanitised so that no residual traces remained. Using variable substitution in the pipelines improved not only security but also maintainability and auditability.

6.5 Challenges In Air-gapped Environments

A central goal of this research was to identify and overcome the challenges of adopting cloud-native technology in air-gapped, on-premises environments. The principal challenges, and how each was resolved, are summarised below.

Designing the code structure: Building a Terraform and CI/CD architecture that matched the internal deployment document was one of the most demanding tasks. Air-gapped implementations are uncommon, so reference material was scarce, and most open-source Terraform examples assume internet-connected cloud environments adapting them for on-premises Proxmox required substantial customisation and experimentation.

Providers and plugins: Terraform's providers and plugins are normally fetched from the internet; here each had to be downloaded, mirrored, and validated for offline use. Finding the Telmate/Proxmox provider version that matched the Proxmox API (v3.0.1-rc9) was critical for stability.

Mirror management: Maintaining offline mirrors for dependencies, modules, and OS packages was essential. Hauler and Nexus repositories held the required binaries so that Terraform and its supporting tools could operate fully offline.

Configuration consistency check: A notable feature of the system was the configuration check that validated Terraform's output against the Ansible inventory before component installation, preventing mismatched IPs or hostnames from breaking the pipeline.

A custom Python script converted Terraform's output.json into an .ini file for comparison and halted the pipeline automatically on any discrepancy. Extending this idea to Terraform itself was both an achievement and a challenge, because Terraform is not designed to act as a configuration check. The difficulty is that installing components changes a VM's memory footprint and other attributes, so that on a subsequent run Terraform would detect the change and rebuild the machine causing repeated installations even though the infrastructure already existed.

This was solved with a lifecycle block instructing Terraform to ignore such changes (for example disk-space or tag/name changes), which removed one of the most troublesome problems encountered during the project.

Backend setup: Configuring Terraform's HTTP backend for GitLab while preventing outbound network requests was a further challenge, since terraform init attempts to reach online registries by default.

A .terraformrc configuration file was used to redirect provider lookups to the local mirrors and to disable the internet access checks entirely.

Template creation: Building the base RHEL 9 template was complex but critical, requiring precise configuration of cloud-init, network parameters, and user permissions; transferring the image from an internet-connected machine into the air-gapped Proxmox environment also had to be handled securely to avoid corruption or version mismatch.

6.5.1 Things to keep in mind while creation and deployment

- The lessons learned point to a set of best practices for future air-gapped deployments of this kind. Mirrors and providers should be managed carefully and kept current.
- The base template is the heart of the creation-and-deployment process and deserves the most attention.
- The whole setup should be designed for reusability. Hardcoded values should be avoided entirely, and a remote backend should be used for state.
- A lifecycle block is necessary to support the configuration-check feature reliably.
- Defining infrastructure through variables keeps it easy to modify and free from avoidable errors.

Finally, only stable, well-tested provider versions should be used, to ensure long-term reliability.

7.Conclusion and Future work

This chapter draws together the outcomes of the research, practice, deployment, and implementation described across the preceding chapters, and sets out what the work contributes, where its limits lie, how industry might adopt it, and which directions future research could take:

7.1 Summary of the contribution

This thesis examined the challenges and the potential of applying cloud-native technologies Terraform, Ansible, and GitLab CI/CD within air-gapped, on-premises environments, with a particular focus on railway automation systems.

It set out a coherent framework showing how the principles of Infrastructure as Code can be adapted to disconnected environments that have traditionally depended on manual processes. The main contributions are the following:

- The design and implementation of a complete IaC-based automation pipeline Terraform for provisioning, Ansible for configuration, and GitLab CI/CD for orchestration operating entirely without internet access.
- A reusable, modular, environment-specific architecture covering several deployment stages (DEV-S, DEV-M, TEST, UAT), which delivers consistent and repeatable provisioning across all of them.
- The integration of offline mirrors and provider repositories to overcome the constraints of air-gapped deployment, allowing Terraform and Ansible to run reliably in fully isolated networks.
- A configuration-validation mechanism that automatically compares Terraform's output with Ansible's input inventories, preventing deployment mismatches and improving reliability.
- An empirical evaluation (Section 6.1) demonstrating measurable reductions in infrastructure setup time relative to manual provisioning, alongside reduced human error and improved reproducibility.

Taken together, the work validates a single, practical claim: cloud-ready IaC practices can be re-engineered to operate efficiently inside on-premises, critical, isolated infrastructure.

In doing so it helps close the gap between modern DevOps methodology and the high-security industrial environments where those methods have so far seen little use.

7.2 Limitations

Three limitations should be read alongside these results.

- The first is manual dependency management: all provider plugins, modules, and mirrors were handled by hand, which adds operational overhead, and automating the mirroring process with tools such as Hauler or Nexus would improve efficiency further.
- The second is limited template flexibility: the RHEL-based golden image required manual preparation and updates, and automating its lifecycle would standardise image management and remove a source of manual error.
- The third is the restricted testing scope: the research was conducted in a controlled on-premises lab, and real-world railway or defence environments may introduce additional factors latency, stricter security, heavier traffic, and formal compliance checks that this setting did not fully reproduce.

7.3 Recommendation for industry Adoption

The research outcomes suggest several recommendations for organisations seeking to adopt IaC-driven automation within air-gapped, on-premises infrastructure. Organisations should **adopt a modular design**, separating Terraform modules from environment-specific configuration to simplify maintenance and improve reuse. They should **establish secure internal mirrors**, deploying artifact repositories such as Nexus and Terraform provider mirrors inside the corporate network to remove any external dependency. They should **integrate DevSecOps principles**, embedding security validation, policy-as-code, and secret management through Vault or GitLab CI variables at every stage of the pipeline. And they should **promote reusability**, standardising the Terraform,Ansible & GitLab

framework across internal projects to shorten delivery cycles and reduce onboarding time. By following these practices, sectors such as railways, defence, and energy can adopt cloud-native automation within their secure, isolated infrastructure while preserving both compliance and operational efficiency[23].

7.4 Future research

Several directions extend naturally from this work. The most immediate is **automated drift detection and self-healing**, using machine-learning models or configuration-validation scripts to identify inconsistencies proactively rather than at the next pipeline run.

Second is the **integration of MLOps workflows** with the air-gapped system, so that model training and deployment can benefit from the same reproducible pipeline.

Third is **stronger security**, through advanced secret-management systems and stricter automated security checks.

Fourth is **automated cluster monitoring** to keep distributed nodes synchronised.

Finally, several capabilities familiar from public cloud platforms such as AWS and Azure VPC management, object storage equivalents, autoscaling, and deeper Kubernetes integration for containerised workloads could be reproduced inside this isolated environment, bringing more of the cloud's operational convenience into air-gapped infrastructure without compromising its isolation.

7.4.1 GitOps Integration for Air-Gapped Environments

A natural evolution of the current push-based pipeline model is the adoption of a full GitOps workflow. In connected environments, tools such as Flux or Argo CD continuously reconcile live infrastructure state against the desired state stored in Git. In an air-gapped setting, the standard pull model must be re-engineered into a controlled push-based workflow or adapted using locally hosted reconciliation agents operating against mirrored repositories.

The four core GitOps principles remain achievable offline:

1. **Declarative:** Configuration is already expressed entirely as Terraform and Ansible definitions.
2. **Versioned:** Git already serves as the single, versioned source of truth.
3. **Automated:** Self-hosted GitLab pipelines already handle provisioning and testing without external connectivity.
4. **Reconciled:** This is the primary gap: future work should implement scheduled or triggered CI/CD runs that systematically compare deployed infrastructure against committed state, flagging and optionally remediating drift automatically.

Implementing an offline GitOps controller would strengthen auditability and provide continuous assurance that deployed infrastructure matches its declared specification a property particularly valuable in safety-critical railway environments.

7.4.2 Policy-as-Code Enforcement

Policy-as-Code (PaC) extends the GitOps model by encoding compliance and governance rules directly into the deployment pipeline. Rather than relying on post-deployment manual audits, policies would be codified, version-controlled, and automatically enforced as mandatory pipeline stages.

Three enforcement mechanisms are envisioned for future implementation:

- **Terraform policy scanning:** Tools such as Checkov or HashiCorp Sentinel would validate Terraform plans against organisational rules (e.g., permitted resource types, naming conventions, network segmentation requirements) before any infrastructure change is applied.

- **Ansible configuration linting:** Extended use of ansible-lint with custom rules would enforce password policies, SSH hardening, and service configuration standards.
- **Pipeline gate enforcement:** GitLab CI/CD stages would block non-compliant changes automatically, preventing insecure configurations from reaching production environments.

The combined GitOps + Policy-as-Code workflow would ensure that every infrastructure change is traceable to a specific commit, every environment can be recreated identically, and non-compliant configurations are stopped before deployment. This is particularly relevant for compliance with standards such as ISA/IEC 62443 and NIST SP 800-82 in railway and critical infrastructure contexts.

Glossary

IaC: Infrastructure as Code.

Air-Gapped: An environment (network or device) that is physically and logically isolated from unsecured networks such as the public internet.

Configuration Management (CM): The process of maintaining computer systems, servers, and software in a desired, consistent state.

Provisioning: The process of setting up IT infrastructure, including servers, virtual machines (VMs), and network resources.

DevOps: A set of practices combining software development (Dev) and IT operations (Ops) to shorten the development life cycle and enable continuous delivery.

Cloud-Native: An approach to building and running applications that exploits the advantages of the cloud computing delivery model.

On-Premises: Software and computing infrastructure hosted and managed internally at an organisation's site rather than in a remote cloud facility.

Ansible: An open-source, imperative-style IaC tool for configuration management, application deployment, and task automation.

Terraform: An open-source, declarative-style IaC tool used to provision and manage infrastructure using HCL (HashiCorp Configuration Language).

Declarative Definition: A configuration style in which the user defines the desired final state and the tool determines the steps to achieve it.

Imperative Definition: A configuration style in which the user specifies the commands or steps the system must execute to reach a desired state.

Dependency Management: The process of identifying, resolving, and retrieving the external libraries, packages, and modules a software project or IaC deployment requires.

CI/CD: Continuous Integration / Continuous Delivery (or Deployment): methods to frequently merge code changes and automatically prepare and deploy them.

Version Control: A system that records changes to a file or set of files over time so that specific versions can be recalled later (e.g. Git).

PoC: Proof of Concept: a realisation of a method or idea to demonstrate its feasibility; often the deliverable of a thesis.

Reproducibility: The ability to execute a process multiple times under the same conditions and achieve the same result.

Proxmox VE: Proxmox Virtual Environment: an open-source server virtualization platform.

ICS: Industrial Control Systems: systems used to control industrial processes (relevant to the railway and critical-infrastructure context).

VLAN: Virtual Local Area Network.

State File: A file that tracks the real-world state of the infrastructure managed by an IaC tool (relevant to Terraform[21].

References

[1] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, “DevOps: Introducing Infrastructure-as-Code,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires: IEEE, May 2017, pp. 497–498. doi: 10.1109/ICSE-C.2017.162.

[2] Mi. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 580–589. doi: 10.1109/ICSME.2019.00092.

[3] N. Ibadah, C. Benavente-Peces, and M.-O. Pahl, “Securing the Future of Railway Systems: A Comprehensive Cybersecurity Strategy for Critical On-Board and Track-Side Infrastructure,” *Sensors*, vol. 24, no. 24, p. 8218, Dec. 2024, doi: 10.3390/s24248218.

[4] A. Ozerov, “Cybersecurity of Railway Command and Control Systems,” *JITA - J. Inf. Technol. Appl. Banja Luka - APEIRON*, vol. 18, no. 2, Mar. 2020, doi: 10.7251/JIT1902053O.

[5] M. Chiari, M. De Pascalis, and M. Pradella, “Static Analysis of Infrastructure as Code: a Survey,” in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 218–225. doi: 10.1109/ICSA-C54293.2022.00049.

[6] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, “A systematic mapping study of infrastructure as code research,” *Inf. Softw. Technol.*, vol. 108, pp. 65–77, Apr. 2019, doi: 10.1016/j.infsof.2018.12.004.

[7] E. Özdoğan, O. Ceran, and M. T. Üstündağ, “Systematic Analysis of Infrastructure as Code Technologies,” *Gazi Univ. J. Sci. Part Eng. Innov.*, vol. 10, no. 4, pp. 452–471, Dec. 2023, doi: 10.54287/gujsa.1373305.

[8] D. Fahland *et al.*, “Declarative versus Imperative Process Modeling Languages: The Issue of Understandability,” in *Enterprise, Business-Process and Information Systems Modeling*, vol. 29, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, Eds., in Lecture Notes in Business Information Processing, vol. 29. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 353–366. doi: 10.1007/978-3-642-01862-6_29.

[9] M. Howard, “Terraform -- Automating Infrastructure as a Service,” May 21, 2022, *arXiv*: arXiv:2205.10676. doi: 10.48550/arXiv.2205.10676.

[10] G. Novakova Nedeltcheva, A. De La Fuente Ruiz, L. Orue-Echevarria Arrieta, N. Bat, and L. Blasi, “Towards Supporting the Generation of Infrastructure as Code Through Modelling Approaches - Systematic Literature Review,” in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, Honolulu, HI, USA: IEEE, Mar. 2022, pp. 210–217. doi: 10.1109/ICSA-C54293.2022.00048.

[11] K. Stouffer *et al.*, “Guide to Operational Technology (OT) security,” National Institute of Standards and Technology (U.S.), Gaithersburg, MD, NIST SP 800-82r3, Sep. 2023. doi: 10.6028/NIST.SP.800-82r3.

[12] Joint Task Force Interagency Working Group, “Security and Privacy Controls for Information Systems and Organizations,” National Institute of Standards and Technology, Sep. 2020. doi: 10.6028/NIST.SP.800-53r5.

[13] C. Lamb and S. Zacchiroli, “Reproducible Builds: Increasing the Integrity of Software Supply Chains,” *IEEE Softw.*, vol. 39, no. 2, pp. 62–70, Mar. 2022, doi: 10.1109/MS.2021.3073045.

[14] J. Park, S. Jeong, and K. Yeom, “Private cloud bespoke orchestrator: techniques for constructing and operating bespoke-private cloud virtual machine environments for cloud users,” *J. Cloud Comput.*, vol. 14, no. 1, p. 34, Jul. 2025, doi: 10.1186/s13677-025-00760-x.
[15] M. Shahin, A. R. Nasab, and M. A. Babar, “A Qualitative Study of Architectural Design Issues in DevOps,” Nov. 13, 2021, *arXiv*: arXiv:2108.06705. doi: 10.48550/arXiv.2108.06705.
[16] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, “Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research”.
[17] F. Beetz and S. Harrer, “GitOps: The Evolution of DevOps?,” *IEEE Softw.*, vol. 39, no. 4, pp. 70–75, Jul. 2022, doi: 10.1109/MS.2021.3119106.
[18] P. Masek, M. Stusek, J. Krejci, K. Zeman, J. Pokorny, and M. Kudlacek, “Unleashing Full Potential of Ansible Framework: University Labs Administration,” in *2018 22nd Conference of Open Innovations Association (FRUCT)*, Jyvaskyla: IEEE, May 2018, pp. 144–150. doi: 10.23919/FRUCT.2018.8468270.
[19] “Compliance and Audit Challenges in DevOps: A Security Perspective,” *Int. Res. J. Mod. Eng. Technol. Sci.*, Oct. 2023, doi: 10.56726/IRJMETS45309.
[20] H. Arif, A. Haikal, and D. Putra, “Implementation of Server Provisioning and Hardening Automation using Terraform and Ansible with NIST SP,” vol. 02, 2025.
[21] S. Almalki, “Integrating Quantitative and Qualitative Data in Mixed Methods Research—Challenges and Benefits,” *J. Educ. Learn.*, vol. 5, no. 3, p. 288, Jul. 2016, doi: 10.5539/jel.v5n3p288.
[22] P. R. D. Achanta, “Strategic Cloud Engineering with Terraform Using Patterns for High-Impact Infrastructure Automation,” vol. 6, no. 12, 2023.
[23] Oluwatosin Oluwatimileyin Abiona, Oluwatayo Jacob Oladapo, Oluwole Temidayo Modupe, Oyekunle Claudius Oyeniran, Adebunmi Okechukwu Adewusi, and Abiola Moshood Komolafe, “The emergence and importance of DevSecOps: Integrating and reviewing security practices within the DevOps pipeline,” *World J. Adv. Eng. Technol. Sci.*, vol. 11, no. 2, pp. 127–133, Mar. 2024, doi: 10.30574/wjaets.2024.11.2.0093.
[24] ISA/IEC 62443, Security for Industrial Automation and Control Systems (series), International Electrotechnical Commission / International Society of Automation.

[25] NTIA, "The Minimum Elements for a Software Bill of Materials (SBOM)," National Telecommunications and Information Administration, Jul. 2021. [Online]. Available: https://www.ntia.gov/sbom.

Appendices

Appendix A — Terraform Root Configuration

The following listings show the root Terraform configuration used to provision a single environment, as described in Section 5.3.2. The provider, backend, and variable definitions are kept in separate files so that each environment differs only in its variable values (Section 5.5).

Listing A.1 — providers.tf (the Telmate/Proxmox provider, pinned to v3.0.1-rc9; see Section 6.5)

```
terraform {
  required_providers {
    proxmox = {
      source = "Telmate/proxmox"
      version = "3.0.1-rc9"
    }
```

Listing A.2 — backend.tf (GitLab HTTP backend with state locking; see Section 5.3.3)

```
backend "http" {}
```

Listing A.3 — main.tf (root module wiring; see Section 5.3.2)

```
provider "proxmox" {
  pm_api_url          = var.pm_api_url
  pm_api_token_id     = var.pm_api_token_id
pm_api_token_secret  = "REDACTED_USE_ENV"
  pm_tls_insecure     = true
  pm_debug = true
}

module "control_nodes" {
  source = "./modules/vm"
  for_each = var.control_nodes
  vm_config = each.value
}

module "worker_nodes" {
  source = "./modules/vm"
```

```
  for_each = var.worker_nodes
  vm_config = each.value
}

module "bastion_nodes" {
  source = "./modules/vm"
  for_each = var.bastion_nodes
  vm_config = each.value
}

module "galera_cluster_nocc" {
  source = "./modules/vm"
  for_each = var.galera_cluster_nocc
  vm_config = each.value
}

module "maxscale_servers" {
  source = "./modules/vm"
  for_each = var.maxscale_servers
  vm_config = each.value
}

module "haproxy_nodes" {
  source = "./modules/vm"
  for_each = var.haproxy_nodes
  vm_config = each.value
}

module "hauler_vm" {
  source = "./modules/vm"
  for_each = var.hauler_vm
  vm_config = each.value
}
```

Listing A.4 — variables.tf and environment.tfvars (the single per-environment file an engineer edits; see Section 5.5)

DEVS.tfvars

```
control_nodes = {
  node1 = {
    name        = "control-1"
    ip          = "priv-172-17-0-100.local"
    gateway     = "priv-172-17-0-1.local"
    dns_servers = ["1.1.1.1", "8.8.8.8"]
    vm_id       = 104
    memory      = 8192
    cores       = 8
    disk_size   = "40G"
    template    = "rhel"
    bridge      = "vmbr1"
    storage     = "local-lvm"
    node        = "proxmox32"
  }
}
galera_cluster_nocc = {
  nocc_node1 = {
    name        = "mariadb01"
    ip          = "priv-172-17-0-104.local"
    server_id   = 1
    gtid_domain_id = 104
    gateway     = "priv-172-17-0-1.local"
    dns_servers = ["1.1.1.1", "8.8.8.8"]
    vm_id       = 108
    memory      = 16384
    cores       = 8
    disk_size   = "60G"
    template    = "rhel"
```

```
    bridge      = "vmbr1"
    storage     = "local-lvm"
    node        = "proxmox32"
  }
}

haproxy_nodes = {
  primary_group = {
    name        = "haproxy-primary"
    ip          = "priv-172-17-0-106.local"
    primary_ha =  "haproxy1"
    gateway     = "priv-172-17-0-1.local"
    dns_servers = ["1.1.1.1", "8.8.8.8"]
    vm_id       = 106
    memory      = 4096
    cores       = 2
    disk_size   = "40G"
    template    = "rhel"
    bridge      = "vmbr1"
    storage     = "local-lvm"
    node        = "proxmox32"
  }
}


hauler_vm = {
  hauler = {
    name = "Hauler-1"
    ip = "priv-172-17-0-20.local"
    gateway = "priv-172-17-0-1.local"
    dns_servers = ["1.1.1.1"]
    vm_id = 150
    memory = 16384
    cores = 4
    disk_size= "200G"
```

```
    template = "hauler-template"
    bridge = "vmbr1"
    storage = "local-lvm"
    node = "proxmox32"
}
}
bastion_nodes = {
    node1 = {
    name = "bastion-1"
    ip = "priv-172-17-0-110.local"
    gateway = "priv-172-17-0-1.local"
    dns_servers = ["1.1.1.1"]
    vm_id = 210
    memory = 4096
    cores = 8
    disk_size= "40G"
    template = "rhel"
    bridge = "vmbr1"
    storage = "local-lvm"
    node = "proxmox32"
}
}

Variable.tfvars
variable "control_nodes" {
  type = map(any)

}

# variable "worker_nodes" {
#   type = map(any)

# }

variable "bastion_nodes" {
```

```
  type = map(any)

}



variable "pm_api_url" {
  type = string
}
variable "pm_api_token_id" {
  type = string
}
variable "pm_api_token_secret" {
  type = string
}




variable "galera_cluster_nocc" {
  type = map(any)

}




variable "haproxy_nodes" {
  type = map(any)

}

variable "hauler_vm" {
  type = map(any)

}
```

Listing A.5 — .terraformrc (redirects provider lookups to local mirrors and disables online checks; see Section 6.5)

```
- export TF_CLI_CONFIG_FILE="$ENV_DIR/terraform.rc"
    - |
      cat > "$TF_CLI_CONFIG_FILE" <<'RC'
      provider_installation {
       filesystem_mirror {
        path    = "terraform-providers"
        include = ["registry.terraform.io/*/*"]
       }
       direct {
        exclude = ["registry.terraform.io/*/*"]   # block internet
       }
      }
      RC
```

Appendix B — Terraform Modules

These are the reusable modules referenced in Section 5.3.2. Each is parameterised so the same code serves DEV-S, DEV-M, TEST, and UAT.

Listing B.1 — modules/control_nodes/main.tf

```
}
  backend "http" {}
}
provider "proxmox" {                                    # access
  pm_api_url          = var.pm_api_url
  pm_api_token_id     = var.pm_api_token_id
pm_api_token_secret  = "REDACTED_USE_ENV"
  pm_tls_insecure     = true
  pm_debug = true
}

module "control_nodes" {
```

```
  source = "./modules/vm"
  for_each = var.control_nodes
  vm_config = each.value
}

# module "worker_nodes" {
#   source = "./modules/vm"
#   for_each = var.worker_nodes
#   vm_config = each.value
# }

module "bastion_nodes" {
  source = "./modules/vm"
  for_each = var.bastion_nodes
  vm_config = each.value
}

module "galera_cluster_nocc" {
  source = "./modules/vm"
  for_each = var.galera_cluster_nocc
  vm_config = each.value
}



# module "maxscale_servers" {
#   source = "./modules/vm"
#   for_each = var.maxscale_servers
#   vm_config = each.value
# }


module "haproxy_nodes" {
  source = "./modules/vm"
  for_each = var.haproxy_nodes
```

```
  vm_config = each.value
}


module "hauler_vm" {
  source = "./modules/vm"
  for_each = var.hauler_vm
  vm_config = each.value
}
```

Appendix C — CI/CD Pipeline

The pipeline definition shown in Figure 5.2: validation, the infrastructure check, Terraform plan/apply, the deployment check, and the Ansible run (see Section 5.4).

Listing C.1 — .gitlab-ci.yml

```
stages:
  - check-infra
  - create-infra
  - generate-yml
  - compare-config
  - install-hauler
  - copy-certificate
  - install-rke2
  - install-haproxy
  - install-mariadb
  - destroy-infra


variables:
  ENV_DIR: "environments/dev_s"
  TF_APPROVE: "auto-approve"
  TF_INPUT: "false"
  TF_env: "dev_s.tfvars"
  COMPARE_ENV: dev_s
  ANSIBLE_ENV: "dev-s.yml"
  STATE_NAME: "infra-$COMPARE_ENV"
  GITLAB_HOST: "http://github.com"
  CI_PROJECT_ID: "590"
```

```
  CI_JOB_TOKEN: "$JOBTOKEN"
  CI_USERNAME: "dubeyp"




check-infra:
  stage: check-infra
  script:
    - export TF_CLI_CONFIG_FILE="$ENV_DIR/terraform.rc"
    - |
      cat > "$TF_CLI_CONFIG_FILE" <<'RC'
      provider_installation {
       filesystem_mirror {
        path    = "terraform-providers"
        include = ["registry.terraform.io/*/*"]
       }
       direct {
        exclude = ["registry.terraform.io/*/*"]   # block
       }
      }
      RC


    - echo "Init remote backend (GitLab) and check changes via plan…"
    -                                                                    export
TF_ADDRESS="$GITLAB_HOST/api/v4/projects/$CI_PROJECT_ID/terraform/state/$STATE_NAME
"
    # - cd $ENV_DIR
    - |
      terraform -chdir="$ENV_DIR" init -reconfigure \
        -backend-config=address="$TF_ADDRESS" \
        -backend-config=lock_address="$TF_ADDRESS/lock" \
        -backend-config=unlock_address="$TF_ADDRESS/lock" \
        -backend-config=lock_method=POST \
        -backend-config=unlock_method=DELETE \
        -backend-config=retry_wait_min=5 \
        -backend-config=username="$CI_USERNAME" \
```

```
        -backend-config=password="$CI_JOB_TOKEN" \
        -input=false
    - terraform -chdir="$ENV_DIR" refresh -var-file="$TF_env" -input="$TF_INPUT" ||
echo "Refresh failed"


    - |
      set +e
      terraform -chdir="$ENV_DIR" plan -var-file="$TF_env" -input=false -detailed-
exitcode -out=tf.plan
      PLAN_EXIT_CODE=$?
      set -e


      if [ "$PLAN_EXIT_CODE" -eq 0 ]; then
        echo "No changes needed - infra up to date"
        touch "$ENV_DIR/infra_exists.flag"
        terraform -chdir="$ENV_DIR" output -json > "$ENV_DIR/output.json"
      elif [ "$PLAN_EXIT_CODE" -eq 2 ]; then
        echo "Changes detected - infra will be created/updated"


      else
        echo "Plan failed"; exit 1
      fi


  only:
    - 'triggers'
  artifacts:
    paths:
      - $ENV_DIR/infra_exists.flag
      - $ENV_DIR/output.json
      - $ENV_DIR/tf.plan
    reports:
      dotenv: plan.env
    expire_in: 1 hour



create-infra:
```

```
  stage: create-infra
  needs: [check-infra]
  script:
    - export TF_CLI_CONFIG_FILE="$ENV_DIR/terraform.rc"
    - |
      cat > "$TF_CLI_CONFIG_FILE" <<'RC'
      provider_installation {
       filesystem_mirror {
        path    = "terraform-providers"
        include = ["registry.terraform.io/*/*"]
       }
       direct {
        exclude = ["registry.terraform.io/*/*"]
       }
      }
      RC
    - export TF_ADDRESS="$GITLAB_HOST/api/v4/projects/$CI_PROJECT_ID/terraform/state/$STATE_NAME"
    # - cd $ENV_DIR
    - |
      terraform -chdir="$ENV_DIR" init -reconfigure \
        -backend-config=address="$TF_ADDRESS" \
        -backend-config=lock_address="$TF_ADDRESS/lock" \
        -backend-config=unlock_address="$TF_ADDRESS/lock" \
        -backend-config=lock_method=POST \
        -backend-config=unlock_method=DELETE \
        -backend-config=retry_wait_min=5 \
        -backend-config=username="$CI_USERNAME" \
        -backend-config=password="$CI_JOB_TOKEN" \
        -input=false
    - |
      if [ -f "$ENV_DIR/infra_exists.flag" ]; then
        echo "Infrastructure already exists and is up to date - skipping apply"

      elif [ -f "$ENV_DIR/tfplan" ]; then
```

```
        echo "Applying pre-generated plan..."
        terraform apply -input=false tfplan
        terraform  output -json > output.json
      else
        echo " No plan file found, generating and applying..."
        terraform  -chdir="$ENV_DIR"  apply  -var-file="$TF_env"  -auto-approve  -
input=false
        terraform output -json > output.json
      fi
  artifacts:
    paths:
      - $ENV_DIR/output.json
    expire_in: 2 hours
  only:
    - 'triggers'


generate-yml:
  stage: generate-yml
  # needs: [create-infra]
  script:
    -        python3        environments/json_to_ini.py        $ENV_DIR/output.json
$ENV_DIR/inventory.yml
  artifacts:
    paths:
      - $ENV_DIR/inventory.yml
  only:
    - 'triggers'


compare-config:
  stage: compare-config
  script:
    - python3 compareini_yml.py --env dev_s
  artifacts:
    when: always
    paths:
      - comparison_logs/
```

```
  only:
    - 'triggers'


install-hauler:
  stage: install-hauler
  needs: [compare-config]
  script:
    - echo "now installing different components on the vm's"
    - |
      ssh -o StrictHostKeyChecking=no root@Proxmox <<EOSSH
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<EOF
        set -ex
        cd /home/ansible/ansible/hauler-registry-fileserver-setup-main
        if [[ ! -f "/home/ansible/ansible/hauler-registry-fileserver-setup-main/flags/hauler_installed.flag" ]]; then
          echo "installing hauler"
          ansible-playbook playbooks/site.yml
          mkdir -p /home/ansible/ansible/hauler-registry-fileserver-setup-main/flags/
          touch /home/ansible/ansible/hauler-registry-fileserver-setup-main/flags/hauler_installed.flag
        else
          echo "hauler already installed"
        fi
      EOF


  only:
    - 'triggers'


copy-certificate:
  stage: copy-certificate
  needs: [install-hauler]
  script:
    - echo "copying certificate"
    - |
      ssh -o StrictHostKeyChecking=no root@proxmox <<'EOSSH'
```

```
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<'EOF'
        set -ex
        cd /home/ansible/ansible/hauler-registry-fileserver-setup-main/CA-Certs
        cp    -f    ca.crt    /home/ansible/ansible/rke2-airgapped-deployment-hauler-
main/cert
        cp -f ca.crt /home/ansible/ansible/haproxy-hauler-playbook-main/cert
        cp -f ca.crt /home/ansible/ansible/mariadb-hauler-community-main/cert
      EOF


  only:
    - 'triggers'









install-rke2:
  stage: install-rke2
  needs: [copy-certificate]
  script:
    - echo "now installing different components on the vm's"
    - |
      ssh -o StrictHostKeyChecking=no root@proxmox <<EOSSH
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<EOF
        set -ex
        cd /home/ansible/ansible/rke2-airgapped-deployment-hauler-main
        if   [[   !   -f   "/home/ansible/ansible/rke2-airgapped-deployment-hauler-
main/flags/rke2_installed.flag" ]]; then
          echo "installing rke2"
          ansible-playbook -i inventories/$ANSIBLE_ENV playbooks/site.yml
          mkdir      -p      /home/ansible/ansible/rke2-airgapped-deployment-hauler-
main/flags/
          touch              /home/ansible/ansible/rke2-airgapped-deployment-hauler-
main/flags/rke2_installed.flag
        else
          echo "rke2 already installed"
```

```
        fi
      EOF


  only:
    - 'triggers'




install-haproxy:
  stage: install-haproxy
  needs: [copy-certificate]
  script:
    - echo "now installing different components on the vm's"
    - |
      ssh -o StrictHostKeyChecking=no root@proxmox <<EOSSH
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<EOF
        set -ex
        cd /home/ansible/ansible/haproxy-hauler-playbook-main
        if [[ ! -f "/home/ansible/ansible/haproxy-hauler-playbook-
main/flags/haproxy_installed.flag" ]]; then
          echo "installing haproxy"
          ansible-playbook -i inventories/$ANSIBLE_ENV playbooks/haproxy.yml
          mkdir -p /home/ansible/ansible/haproxy-hauler-playbook-main/flags/
          touch /home/ansible/ansible/haproxy-hauler-playbook-
main/flags/haproxy_installed.flag
        else
          echo "haproxy already installed"
        fi
      EOF


  only:
    - 'triggers'




install-mariadb:
```

```
  stage: install-mariadb
  needs: [copy-certificate]
  script:
    - echo "now installing different components on the vm's"
    - |
      ssh -o StrictHostKeyChecking=no root@proxmox <<EOSSH
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<EOF
        set -ex
        cd /home/ansible/ansible/mariadb-hauler-community-main
        if [[ ! -f "/home/ansible/ansible/mariadb-hauler-community-main/flags/mariadb_installed.flag" ]]; then
          echo "installing mariadb"
          ansible-playbook -i inventories/$ANSIBLE_ENV playbooks/site.yml
          mkdir -p /home/ansible/ansible/mariadb-hauler-community-main/flags/
          touch /home/ansible/ansible/mariadb-hauler-community-main/flags/mariadb_installed.flag
        else
          echo "mariadb already installed"
        fi
      EOF

  only:
    - 'triggers'




destroy-infra:
  stage: destroy-infra
  script:
    - export TF_CLI_CONFIG_FILE="$ENV_DIR/terraform.rc"
    - |
      cat > "$TF_CLI_CONFIG_FILE" <<'RC'
      provider_installation {
       filesystem_mirror {
        path    = "terraform-providers"
        include = ["registry.terraform.io/*/*"]
```

```
       }
       direct {
        exclude = ["registry.terraform.io/*/*"]
       }
      }
      RC
    - export TF_ADDRESS="$GITLAB_HOST/api/v4/projects/$CI_PROJECT_ID/terraform/state/$STATE_NAME"
    - |
      terraform -chdir="$ENV_DIR" init -reconfigure \
        -backend-config=address="$TF_ADDRESS" \
        -backend-config=lock_address="$TF_ADDRESS/lock" \
        -backend-config=unlock_address="$TF_ADDRESS/lock" \
        -backend-config=lock_method=POST \
        -backend-config=unlock_method=DELETE \
        -backend-config=retry_wait_min=5 \
        -backend-config=username="dubeyp" \
        -backend-config=password="$CI_JOB_TOKEN" \
        -input=false
    - terraform -chdir="$ENV_DIR" destroy -$TF_APPROVE -input=$TF_INPUT -var-file=$TF_env
    - |
      ssh -o StrictHostKeyChecking=no root@proxmox <<'EOSSH'
      ssh -o StrictHostKeyChecking=no ansible@priv-172-19-0-30.local <<'EOF'
        rm -f /home/ansible/ansible/hauler-registry-fileserver-setup-main/flags/hauler_installed.flag
        rm -f /home/ansible/ansible/rke2-airgapped-deployment-hauler-main/flags/rke2_installed.flag
        rm -f /home/ansible/ansible/haproxy-hauler-playbook-main/flags/haproxy_installed.flag
        rm -f /home/ansible/ansible/mariadb-hauler-community-main/flags/mariadb_installed.flag
      EOF
  when: manual
  only:
    - 'triggers'
```

Appendix D — Configuration Consistency Check

The validation step that keeps Terraform output and the Ansible inventory in sync before component installation (Section 5.4, Section 6.5).

Listing D.1 — Python validation script (converts Terraform output.json into inventory.ini and compares it with the Ansible inventory, halting the pipeline on any mismatch)

```
import yaml

import os

import sys

import argparse


parser = argparse.ArgumentParser()

parser.add_argument("--env", required=True)

args = parser.parse_args()

terraform_yml_path = f'environments/{args.env}/inventory.yml'

ansible_root_dir = 'Ansible-Deployment'




components = {

    'rke2': {

        'file': f'{args.env}.yml',

        'prefixes': ['control', 'worker', 'bastion'],

        'name_keys': ['node_name']

    },

    'haproxy': {

        'file': f'{args.env}.yml',

        'prefixes': ['haproxy'],

        'name_keys': ['primary_ha','secondary_ha']

    },

    'mariadb': {

        'file': f'{args.env}.yml',

        'prefixes': ['mariadb', 'galera'],

        'name_keys': ['wsrep_node_name']

    }

}
```

```
def load_tf_inventory(path):
    with open(path, 'r') as f:
        data = yaml.safe_load(f)

    hosts = {}
    for group, entries in data.items():
        if isinstance(entries, dict):
            for hostname, conf in entries.items():
                if isinstance(conf, dict) and 'ansible_host' in conf:
                    hosts[hostname] = str(conf['ansible_host'])
    return hosts


def load_ansible_component_hosts(path, name_keys):
    with open(path, 'r') as f:
        data = yaml.safe_load(f)

    hosts = {}
    if not data or 'all' not in data or 'children' not in data['all']:
        return hosts

    children = data['all']['children']
    for group_name, group_data in children.items():
        if not isinstance(group_data, dict):
            continue


        if 'children' in group_data:
            for nested_group, nested_data in group_data['children'].items():
                if 'hosts' in nested_data:
                    for ansible_name, meta in nested_data['hosts'].items():
                        if not isinstance(meta, dict):
                            continue
```

```
                        if 'ansible_host' in meta:
                            ip = str(meta['ansible_host'])
                            for key in name_keys:
                                if key in meta:
                                    logical_name = str(meta[key])
                                    hosts[logical_name] = ip
        elif 'hosts' in group_data:
            for ansible_name, meta in group_data['hosts'].items():
                if not isinstance(meta, dict):
                    continue
                if 'ansible_host' in meta:
                    ip = str(meta['ansible_host'])
                    for key in name_keys:
                        if key in meta:
                            logical_name = str(meta[key])
                            hosts[logical_name] = ip

    return hosts


def compare_hosts(component, tf_hosts, ans_hosts, prefixes):
    print(f"\n Comparing for component: \033[1;36m{component}\033[0m")


    tf_filtered = {
        h: ip for h, ip in tf_hosts.items()
        if any(h.startswith(prefix) for prefix in prefixes)
    }

    all_ok = True
    for host, ip in tf_filtered.items():
        if host not in ans_hosts:
            print(f"\033[1;31m        Host        '{host}'        missing        in
{component}/f'{args.env}.yml'\033[0m")
            all_ok = False
```

```
        elif ans_hosts[host] != ip:
            print(f"\033[1;31m    IP    mismatch    for    '{host}':    TF={ip},
YAML={ans_hosts[host]}\033[0m")
            all_ok = False
        else:
            print(f"\033[1;32m {host} matches ({ip})\033[0m")


    for host in ans_hosts:
        if host not in tf_filtered:
            print(f"\033[1;33m⚠ Host '{host}' in YAML but not expected for this
component\033[0m")


    return all_ok


# === Main ===
def main():
    if not os.path.exists(terraform_yml_path):
        print(f"\033[1;31m      Missing      Terraform      inventory      YAML:
{terraform_yml_path}\033[0m")
        sys.exit(1)


    tf_hosts = load_tf_inventory(terraform_yml_path)


    all_matched = True
    for component, info in components.items():
        yaml_path = os.path.join(ansible_root_dir, component, info['file'])
        if not os.path.exists(yaml_path):
            print(f"\033[1;31m Missing YAML file: {yaml_path}\033[0m")
            all_matched = False
            continue


        ans_hosts = load_ansible_component_hosts(yaml_path, info['name_keys'])
        matched = compare_hosts(component, tf_hosts, ans_hosts, info['prefixes'])
        if not matched:
            all_matched = False
```

```
    if all_matched:
        print("\n\033[1;32m All inventory entries match!\033[0m")
        sys.exit(0)
    else:
        print("\n\033[1;31m Some mismatches found. Please review above.\033[0m")
        sys.exit(1)


if __name__ == "__main__":
    main()
```

Appendix E — Ansible Configuration

Post-provision configuration and component installation applied on top of the Terraform-provisioned VMs (Section 5.4).

Listing E.1 — Sample playbook (HAProxy, MariaDB, Hauler)

```
all:
  vars:
    environment_type: "DEV-S"
    auto20_remote_user: rhel
  children:
    galera_cluster_nocc:
      hosts:
        nocc-node1:
          ansible_host: priv-172-17-0-104.local
          wsrep_node_name: mariadb01
          server_id: 1
          gtid_domain_id: 104
```

Listing E.2 — Generated inventory.ini (example output of the check in Appendix D)

```
bastion_nodes:
  bastion-1:
    ansible_host: priv-172-17-0-110.local
control_nodes:
  control-1:
    ansible_host: priv-172-17-0-100.local
galera_cluster_nocc:
  mariadb01:
```

```
    ansible_host: priv-172-17-0-104.local
haproxy_nodes:
  haproxy-primary:
    ansible_host: priv-172-17-0-106.local
```

Appendix F — Base Template Preparation

A step-by-step record of building the RHEL 9.8 cloud-init golden image on Proxmox (Section 5.3.1): ISO install, static IP and bridge configuration, installation of Git / cloud-init / SSH utilities, embedded mirrors, and the bootstrap scripts.

Listing F.1 — cloud-init configuration and bootstrap commands

```
Step 1 — Select and upload the ISO. Place the RHEL 9.8 installation ISO in Proxmox storage so it can be used as installation media. In the web UI, open the storage that holds ISO images (for example local), select ISO Images → Upload, and upload the RHEL 9.8 ISO. In an air-gapped environment the ISO is transferred from the scanned staging machine rather than downloaded directly.

bash

# Equivalent CLI: copy the ISO into the Proxmox ISO store

cp RHEL-9.8-x86_64-dvd.iso /var/lib/vz/template/iso/

Step 2 — Create the virtual machine and apply the base configuration. In the web UI choose Create VM and configure it as the template baseline:

General: set a VM ID (<VMID>) and a clear name such as rhel98-template.

OS: select the uploaded RHEL 9.8 ISO as the installation medium; guest OS type Linux.

System: keep the default SCSI controller (VirtIO SCSI), enable the QEMU Guest Agent, and select the BIOS/UEFI used by your environment.

Disk: create the system disk on the target storage (<storage>) with the agreed size; use the VirtIO/SCSI bus.

CPU / Memory: assign the baseline cores and RAM defined for the template (these can be overridden per environment later through Terraform variables).

Network: attach the VM to the correct bridge (vmbr0) and VLAN, matching the IP/pool plan for the environments.

bash

# Equivalent CLI

qm create <VMID> --name rhel98-template --memory 4096 --cores 2 \

  --net0 virtio,bridge=vmbr0 --scsihw virtio-scsi-pci \
```

```
  --scsi0 <storage>:32 --ide2 <storage>:iso/RHEL-9.8-x86_64-dvd.iso,media=cdrom \
  --ostype l26 --agent enabled=1
```

Step 3 — Install RHEL 9.8 and prepare it for automation. Start the VM, open the console, and complete a standard RHEL 9.8 installation (minimal/server base, root account, and initial network settings). Once the OS is installed, install and enable the packages the automation workflow depends on — cloud-init (so Terraform/Proxmox can inject configuration at first boot), the QEMU guest agent, Git, and the SSH client utilities — and point the package manager at the local mirrors so no external repository is contacted.

```
bash

# Inside the guest, using the air-gapped local mirror

dnf install -y cloud-init qemu-guest-agent git openssh-clients

systemctl enable cloud-init qemu-guest-agent sshd

# (configure /etc/yum.repos.d/*.repo to point at the local Nexus/mirror)
```

Step 4 — Generalise the image. Before the VM is turned into a template it must be "cleaned" so that every clone boots with a unique identity rather than inheriting the original's machine ID, SSH host keys, and logs. This is what lets cloud-init reconfigure each clone correctly.

```
bash

# Inside the guest, run as root, then power off

cloud-init clean --logs

truncate -s 0 /etc/machine-id

rm -f /etc/ssh/ssh_host_*

rm -f /var/log/wtmp /var/log/btmp

history -c

poweroff
```

Step 5 — Attach the cloud-init drive and convert to a template. With the VM powered off, add a cloud-init drive so Proxmox can pass per-VM settings (user, SSH key, IP configuration) at deployment time, set the boot disk, and confirm the guest agent is enabled. Finally, convert the VM to a template. In the web UI this is right-click the VM → Convert to template; from the CLI:

```
bash

qm set <VMID> --ide0 <storage>:cloudinit

qm set <VMID> --boot order=scsi0

qm set <VMID> --serial0 socket --agent enabled=1

qm template <VMID>
```