Comparative Performance Analysis of IaC Tools for Deploying Containerized SDN Topologies in Cloud Environments

Vitor Reiel M. Lima[1], **Arthur Callado**[1], **Michel Bonfim**[1], **Enyo Gonçalves**[1]

[1]Programa de Pós-Graduação em Computação (PCOMP)
Universidade Federal do Ceará (UFC) - Quixadá, CE - Brasil

vitorreiel@alu.ufc.br, {arthur, michelsb, enyo}@ufc.br

Abstract. *This paper presents a comparative analysis of Infrastructure as Code (IaC) tools applied to the automation of containerized Software-Defined Networking (SDN) topologies in cloud environments. The study highlights the increasing complexity of modern networks and the necessity for efficient and automated management through NetDevOps principles. It explores the capabilities of popular IaC tools, specifically Ansible and Terraform, in the provisioning and deployment of SDN topologies using Containernet and Docker. The research includes a quantitative analysis of runtime performance across network topologies (Single, Linear, and Tree) and evaluates tool efficiency in creating and destroying network scenarios. As results, we present how the tools perform in terms of execution time, providing insights into advantages and challenges associated with using different IaC tools automating emulated network environments.*

1. Introduction

Modern networks, with growing complexity and increasing dependence on automated deployment, demand study approaches that go beyond isolated models or simulations. While models outline boundaries and simulations offer insights into general behavior, both neglect implementation details [Di Lena et al. 2021]. The concept of NetDevOps stands out by applying DevOps principles to network engineering and operations. NetDevOps promotes a version-controlled, automated network infrastructure, treating network systems and devices as software. By using Infrastructure as Code (IaC), NetDevOps enables network infrastructure efficient management, provisioning, and deployment through machine-readable definition files [Shah and Dubaria 2019]. This work aims to investigate the performance of IaC-based automation mechanisms applied to the provisioning of emulated Software Defined Networking (SDN) topologies in cloud environments.

SDN brings a significant evolution in network infrastructures by separating the network control logic from packet forwarding, enabling greater programmability, centralized management, and reduced operational complexity. This architectural organization enhances management capabilities and supports control function abstraction, handling dynamic heterogeneous network infrastructures more efficiently [Hussain et al. 2022]. This programmability requires automation mechanisms that support the systematic execution of recurring tasks, such as the creation, modification, and destruction of large topologies in experimentation and validation scenarios. This need naturally reinforces the importance of experimental environments in which such automated mechanisms can be evaluated.

Network emulation emerges as a practical alternative by enabling the construction of realistic scenarios for the development and validation of applications with lower cost

and greater flexibility. The literature indicates that simulation and emulation tools play a relevant role in the development and testing of SDN solutions by allowing the evaluation of network behavior prior to deployment in real environments [Hussain et al. 2022]. In addition, SDN emulation provides controlled and repeatable network scenarios, which makes is useful for experimental evaluations of automation mechanisms. Despite this, the efficiency of automation mechanisms applied to emulated environments remains little explored, especially in cloud infrastructure. This gap motivates the present work.

Among SDN emulation platforms, Mininet stands out as one of the most popular solutions due to its ability to create lightweight emulated network topologies [Di Lena et al. 2021]. Building on this foundation, Containernet[1], an extension to Mininet, allows network topologies to be managed using Docker containers as hosts, enabling the creation and operation of complex virtual networks with versatility and the possibility of deploying them in the cloud. This approach offers flexibility and scalability, and is particularly relevant for automatically-provisioned SDN environments.

Network automation arises as a response to operational challenges, reducing manual errors and increasing infrastructure management efficiency. IaC techniques automate the configuration and provisioning of resources, promoting consistency and security [Brikman 2022]. Network as Code (NaC) extends the IaC approach to computer networks, applying it to data centers, local networks, and cloud environments. NaC integrates into the NetDevOps culture, bringing changes in network design and operation, from team collaboration to the choice of tools for managing network configurations [Preston 2024].

Given the lack of systematic evaluations on the efficiency of automation methods applied to emulated SDN networks in cloud environments, this work proposes an experimental comparative analysis between two consolidated IaC tools applied to the provisioning of containerized SDN topologies using Containernet and Docker. The study evaluates the runtime performance of these tools in creating and removing the topologies (Single, Linear, and Tree) in cloud instances. All scripts and applied topologies were made available, including the IaC and NaC provisioning codes, so that they can be adapted to support new experiments and topologies, thereby contributing to transparency and reproducibility.

The remainder of this work is organized as follows. Section 2 presents related work. Section 3 details the theoretical foundation. The methodology is described in Section 4. Section 5 discusses experimental methods and results. Limitations and threats to validity are shown in Section 6. Section 7 presents paper conclusions and future work.

2. Related Work

In a rapidly evolving technological landscape, IT infrastructure management automation gains importance. With a simple, agentless architecture, Ansible effectively optimizes repetitive processes. [Elradi 2023] claims that Ansible automation significantly reduced the time required to deploy a web application. On the other hand, for [Noll and Pretto 2020] the implementation of IaC with tools such as Terraform, Ansible, and Kubernetes within the DevOps methodology highlights version control, disaster recovery, and scalability through the automated creation of a cloud environment. These studies reinforce the importance of automation, but focused on infrastructure and application provisioning rather than network emulation.

[1]https://github.com/containernet/containernet

For [Salazar-Chacón and Parra 2023] IaC, implementation in Open Networking using Cumulus Linux, Ansible, and Git for network automation and optimization highlights the importance of digital transformation and agility in infrastructures, especially when faster operational adaptation is needed. Conversely, [Bali and Walia 2023] analyze IaC in modernizing IT management, replacing manual processes with definitions and provisioning via code. They review tools such as Terraform, CloudFormation and Ansible, and propose a methodology to optimize configuration and ensure consistency. The works discuss automation in networking and infrastructure management, but focus on adoption benefits and process improvement rather than controlled comparative evaluation.

Another work [Sicoe et al. 2022] explores network automation deployment in OpenStack using Ansible and Terraform to configure Cisco virtual routers. It demonstrates that automation with infrastructure as code reduces time and complexity in network configuration and suggests testing other cloud providers and using Napalm to support multi-vendor infrastructures. Meanwhile, [Kodolov et al. 2020] addresses the modernization of communication infrastructure management by adopting IaC in lab environments that combine physical and virtual equipment. They show how IaC facilitates the integration and automation of lab resources using Ansible, GitHub, and Visual Studio Code. Although these studies are closer to the networking domain, they remain centered on virtual router configuration or laboratory infrastructure management rather than on automated provisioning of emulated SDN topologies composed of containerized elements.

Previous studies have focused on infrastructure and application deployment, network configuration, or laboratory automation. This study investigates the provisioning of emulated SDN topologies based on containerized elements and deployed in the cloud (see Table 1). In this context, Ansible and Terraform are compared in terms of the time required to create and remove Single, Linear, and Tree topologies under the same experimental conditions. The main contribution of this work, therefore, lies in evaluating IaC performance in cloud-based SDN emulation rather than general automation scenarios.

Table 1. Comparative analysis between related work and the proposed study

Related Work	Full automation	Network emulation SDN	Virtualization tool	Cloud Deployment	Platforms and tools
[Elradi 2023]	✓	×	✓	×	Ansible, VMware Workstation, Nginx, MySQL and Tomcat.
[Noll and Pretto 2020]	✓	×	✓	✓	Terraform, Ansible, Jenkins, Docker, Django, Kubernetes, Postgresql and Google Cloud Platform.
[Salazar-Chacón and Parra 2023]	✓	×	×	×	Ansible and GitLab
[Bali and Walia 2023]	✓	×	✓	×	Ansible, Terraform and CloudFormation.
[Sicoe et al. 2022]	✓	×	✓	✓	Ansible, Terraform, OpenStack, VMware Workstation and Wireshark.
[Kodolov et al. 2020]	✓	×	✓	×	Ansible, Visual Studio Code and VMware vSphere.
This work	✓	✓	✓	✓	Ansible, Terraform, Docker, Containernet and AWS Academy.

Legend: ✓ Yes - × No

3. Theoretical Framework

3.1. NetDevOps

NetDevOps combines DevOps with network management to increase network infrastructure administration efficiency and agility. DevOps unites development and opera-

tions with a focus on automation and agile cycles [Leite et al. 2019]. However, NetDevOps uses IaC to configure and provision network resources consistently and scalably [Shah and Dubaria 2019]. It also integrates tools like Git to manage configurations and code changes, improving traceability, collaboration, and enabling efficient rollbacks.

3.2. IaC - *Infrastructure as Code*

Infrastructure as Code (IaC) is an automation methodology based on software development principles and practices. It manages and provisios infrastructure resources through configuration files rather than relying on manual processes [Morris 2020]. IaC automates the configuration, provisioning, and management of IT environments with scripts and tools. It manages infrastructure similarly to software, enabling process automation, configuration consistency, and allows quick and reliable environments reproduction.

3.3. NaC - *Network as Code*

Network as Code (NaC) applies the principles of IaC to network management. NaC adapts the concept of IaC for managing computer networks, covering traditional data centers, local and metropolitan networks, and cloud environments [Preston 2024]. This allows network assets' configuration and management to be handled in a programmatic and automated way, similar to software. NaC aims to improve efficiency and consistency in network administration, often integrating with the *NetDevOps* culture, which combines *DevOps* practices with network operations to provide agile and automated management.

3.4. *Containernet*

Containernet [Containernet 2026] is a Mininet fork using Docker containers as hosts in emulated networks. It is widely used in cloud/fog/edge computing, and Network Functions Virtualization (NFV) research. It offers features to add or remove containers, execute commands via the Mininet CLI, and control container resources like CPU and memory. It can be installed on bare metal or in a privileged Docker container, providing flexibility to create custom network emulators and testbeds. In this study, it was chosen because it allows the use of containerized hosts within emulated SDN topologies, which is consistent with the cloud-based experimental environment adopted in this work, while preserving the lightweight and programmable approach originally provided by Mininet.

4. Methodology

This section details the methodology, which aims to evaluate the IaC tools with a quantitative and comparative data analysis. The entire process of creating and emulating the network scenarios is automated through scripts and configuration files. Figure 1 depicts the methodology workflow, following a set of steps until the environment is ready and the runtime data is collected. These steps are described in more detail as follows.

4.1. Scenario definition script

4.1.1. Initial execution criteria

The cloud infrastructure provider adopted for the experiments in this study was AWS. Since the provisioning process is defined through IaC tools, the same approach can be

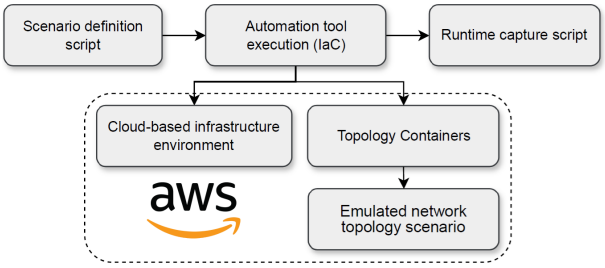

Figure 1. Workflow for automated provisioning of emulated SDN topologies using IaC in cloud environments.

adapted to other providers with corresponding configuration. The AWS access credentials are specified in a configuration file named `aws_access`, which defines the parameters `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`. Once credentials are configured, the script responsible for defining the target topology (Single, Linear, or Tree) enables the selected IaC tool to authenticate with the cloud environment and provision both the infrastructure and the emulated SDN network topology.

4.1.2. Script Execution

The topology implementation begins by selecting the topology type. Three options are supported: Single, Linear, and Tree (detailed in Sections 4.4.1, 4.4.2, and 4.4.3, respectively). Additionally, a script[2] allows the specification of the IaC tool to be employed during the provisioning process, enabling the use of either Ansible or Terraform.

4.2. Automation tools

The selection of Ansible and Terraform was guided by practical and methodological considerations. The literature presents both as well-established and widely adopted IaC. In addition, they present different approaches to infrastructure automation, which makes their comparison relevant in the context of this study. Limiting the analysis to these two tools allows the initial experimental and statistical design to remain controlled and feasible. In this study, the evaluated tools were Ansible v2.16.0 and Terraform v1.14.8.

4.2.1. Ansible

Ansible is a popular open-source automation tool to simplify the configuration, management, and orchestration of IT systems [Ansible 2025]. Tasks such as application deployment, configuration management, and system updates benefit from its agentless architecture (no need to install software on managed devices). Using YAML scripts (*playbooks*), automating tasks in complex and heterogeneous IT environments is simple and efficient.

[2]https://github.com/vitorreiel/cloud-sdn-setup.git

4.2.2. Terraform

Terraform is an open-source tool used to define and provision IT infrastructure in an automated and programmable way [Terraform 2026]. It describes infrastructure resources, such as servers, networks, and cloud services, in HashiCorp Configuration Language (HCL) configuration files that can be versioned and reused. It supports multiple cloud providers (e.g., AWS, Azure, and Google Cloud), and enables consistent and repeatable infrastructure management. It can remotely create, update, and destroy infrastructure resources securely and efficiently, facilitating the management of complex environments.

4.3. Cloud Infrastructure Deployment

The infrastructure in AWS Academy is created and configured through a sequence of processes using the defined automation tool. Regardless of the tool, the sequence of tasks remains consistent. Initially, a Security Group is created to define access control, including port 22 for Secure Shell (SSH) and ports 8181, 6653, and 6633 used by ONOS OpenFlow. Next, an SSH Key Pair is generated to ensure the security of remote connections. Finally, a t2.large EC2 instance is deployed with Ubuntu Server 24.04 LTS and 30 GB of storage, configured with SSH, Docker, and other environment-specific dependencies.

4.4. Implementation of topologies

After the cloud infrastructure is built using the IaC tool, Python scripts are executed within the instance during the final provisioning stage. These scripts generate Docker images and initialize the containers that represent the devices of the emulated topologies defined in the initial script (see Section 4.1.2). The number of containers varies depending on the topology. The Python script applies NaC methods to configure the network elements orchestrated by Containernet. The scenario includes the "onosproject/onos" image for the ONOS controller, "containernet/containernet" for equipment emulation and interconnection, "alpine" for hosts, and Open vSwitch for emulated switches. Each host has its own container, allowing individual access to commands, configurations, and testing. Therefore, the measured runtimes should be understood as referring to the provisioning in the cloud and initialization of emulated and containerized network scenarios.

The topologies defined in this work are used for a comparative analysis of the automation tools runtime, based on a supporting material [Bholebawa and Dalal 2016].

The three network topologies (Single, Linear, and Tree) represent increasing levels of structural complexity and dependency amongst network elements. This progression imposes different orchestration and provisioning demands on the automation tools, enabling the evaluation of their behavior under different conditions. The use of multiple topologies also avoids conclusions based on a single experimental configuration.

4.4.1. Single Topology Definition

The Single topology (Figure 2) consists of 1 controller, 1 switch, and 16 hosts in the IP range 10.0.10.0/24. The controller (C1) is directly connected to the switch (S1), which distributes connectivity to all 16 hosts (H1 to H16). This results in a simple and efficient structure suitable for environments where hierarchical segmentation is not required.

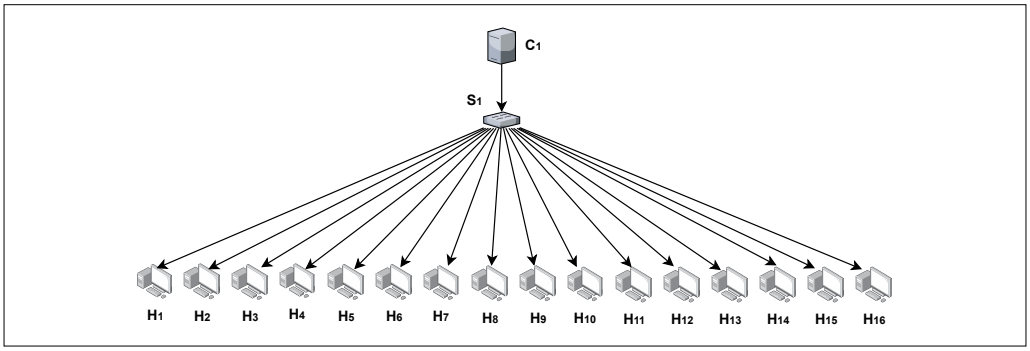

Figure 2. Single Topology

4.4.2. Linear Topology Definition

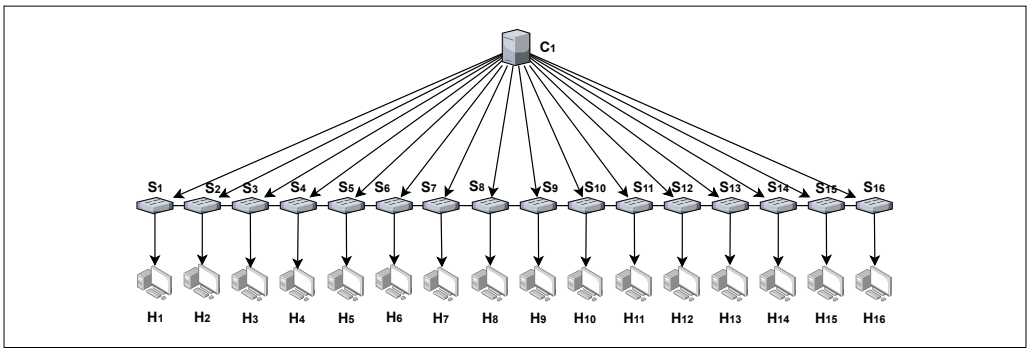

Figure 3. Linear Topology

The Linear topology (Figure 3) is composed of one controller, 16 switches, and 16 hosts, organized sequentially and using the IP addressing range 10.0.10.0/24. The controller (C1) is connected to all switches (S1 to S16), and the switches are also interconnected with their neighboring switches linearly. In addition, each switch is connected to a single host, forming a linear chain from controller to host. In this configuration, traffic between hosts traverses multiple intermediate switches, characterizing a gradual chaining of the network suitable for scenarios with simple structure and easy progressive expansion.

4.4.3. Tree Topology Definition

The Tree topology (Figure 4) consists of 1 controller, 5 switches, and 16 hosts, organized hierarchically and using the IP range 10.0.10.0/24. The controller (C1), at level 0, connects to the root switch (S1). This switch (S1) is then connected to four level 2 switches (S2, S3, S4, and S5). Each level 2 switch is connected to four hosts, totaling 16 hosts in the network. This arrangement provides a hierarchical and scalable structure, ideal for environments that require efficient network segmentation and organization.

4.5. Runtime capture script

The runtime script creates a dataset that stores the time, in seconds, required for each IaC tool to build the cloud infrastructure and implement the emulated SDN network en-

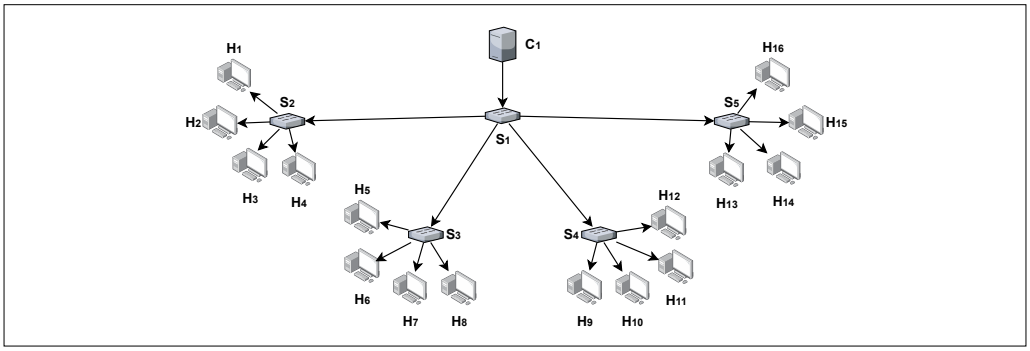

Figure 4. Tree Topology

vironment. With 100 replications for each topology in each automation tool, it totals 300 measurements per IaC tool. It counts the seconds between the IaC tool starts, and the infrastructure is fully provisioned and the emulated network topology is implemented. The script is also configurable, allowing for its use in other experimental settings.

5. Experiments and Results

To validate our solution, we conducted an experiment consisting of a quantitative analysis (section 5.1). In the following sections, we will describe how the experiment was guided.

5.1. Quantitative Analysis

In this experiment, we analyzed the runtime of different IaC tools for implementing three distinct emulated network topologies, ensuring that these topologies become operational for the end user with implementation on AWS. For this, we used the t2.large EC2 instance.

The data capture process for analysis was done by executing the capture script (section 4.5), which runs each automation tool individually and records the time taken by each tool to create the necessary Cloud infrastructure, initialize the EC2 instance, and perform the required installations and configurations of the scenario. This process is repeated 100 times for each network topology defined in the study to generate samples for statistically significant results. Additionally, once the script captures the time each IaC tool takes to set up the scenario, it also destroys the previously created infrastructure. The obtained data is stored in a dataset[3]. The following are the results:

Figure 5 presents the boxplot graphs comparing the runtimes of the Single, Linear, and Tree topologies using Ansible and Terraform. In the left graph, referring to Ansible, the medians are close to 260 seconds for all topologies. Outliers above 300 seconds are observed in all topologies, with more significant outliers in the Tree topology above the upper limit of the boxplot, indicating considerable latency in creating this scenario. In the right graph, referring to Terraform, the median times are below 200 seconds for all topologies. However, there are many outliers, especially in the Single and Linear topologies, while the Tree topology shows less variability and fewer outliers. Overall, Terraform exhibits more consistent and often shorter runtimes than Ansible, suggesting potentially superior efficiency in implementing these topologies.

[3]https://github.com/vitorreiel/dataset-sdn-setup

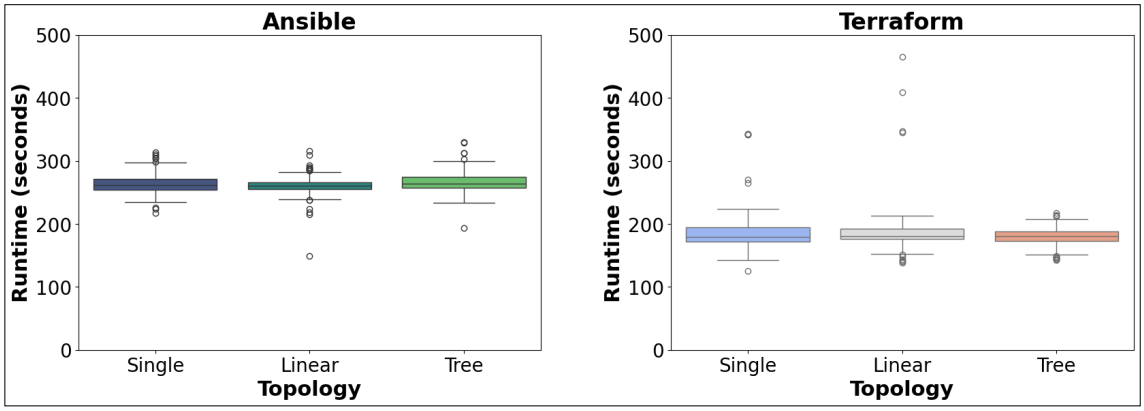

Figure 5. Boxplots graphs of runtimes for scenario implementations

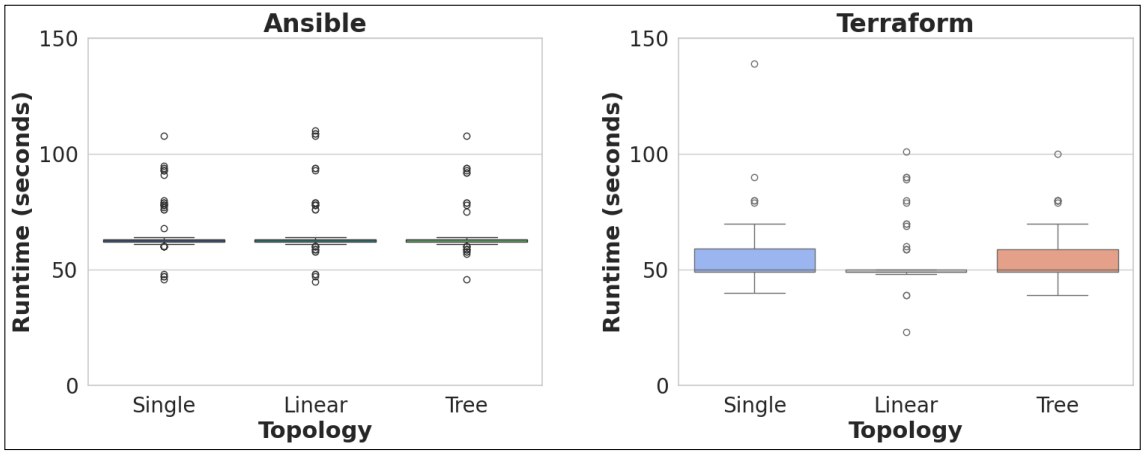

Figure 6. Boxplots of runtimes for scenario destruction

Figure 6 presents the boxplot graphs comparing the runtimes for removing the Single, Linear, and Tree topologies using Ansible and Terraform. For Ansible, the runtimes are generally low across all topologies, with medians around 60 seconds. Outliers above and below the boxplot limits are present in all topologies, indicating higher variability in destruction times. For Terraform, the medians are even lower, taking around 50 seconds in nearly all topologies. However, there is considerable data dispersion, especially in the Single topology, which shows a single outlier of almost 142 seconds. Overall, Terraform tends to show slightly lower and more consistent destruction times compared to Ansible, suggesting better efficiency for removing the analyzed topologies.

5.2. Descriptive Statistics

To determine which IaC tool performs better in creating and destroying network scenarios, we applied a hypothesis test. In this study, we used the Shapiro-Wilk (SW) normality test to analyze the data distribution and guide the decision on the most appropriate statistical approach. The Shapiro-Wilk test was chosen due to its sensitivity in detecting deviations from normality in samples. Its application aims to verify whether the data can be considered parametric if they follow a normal distribution or non-parametric otherwise.

5.2.1. Adhesion Test

Table 2. Shapiro-Wilk (SW) normality test for scenario creation

Tool (IaC)	Topology	Sample	Average	Median	Mode	W (SW)	p-value (SW)
Ansible	Single	100	264,8	262,0	258,0	0,963	0,006
Ansible	Linear	100	259,9	260,0	259,0	0,848	0,000
Ansible	Tree	100	267,3	263,5	257,0	0,909	0,000
Terraform	Single	100	189,9	179,5	177,0	0,706	0,000
Terraform	Linear	100	188,9	180,0	179,0	0,513	0,000
Terraform	Tree	100	179,5	180,0	175,0	0,959	0,003

Table 2 presents the results of the SW normality test applied to the creation runtimes of network scenarios using IaC tools. For each defined network topology (Single, Linear, and Tree), descriptive statistics are provided, including the mean, median, and mode of the execution time in seconds, as well as the SW test's W statistic and the respective p-value. The results indicate that all combinations of tools and topologies showed p-values below 0,05, suggesting that the data do not follow a normal distribution. It implies that the runtimes for creating network scenarios cannot be analyzed using parametric statistical methods, so a non-parametric approaches was needed for subsequent analysis.

Table 3. Shapiro-Wilk (SW) normality test of scenario destruction

Tool (IaC)	Topology	Sample	Average	Median	Mode	W (SW)	p-value (SW)
Ansible	Single	100	66,2	63,0	63,0	0,707	0,000
Ansible	Linear	100	65,9	63,0	63,0	0,626	0,000
Ansible	Tree	100	64,9	63,0	63,0	0,531	0,000
Terraform	Single	100	55,2	50,0	49,0	0,578	0,000
Terraform	Linear	100	52,1	49,0	49,0	0,506	0,000
Terraform	Tree	100	53,9	50,0	49,0	0,664	0,000

Table 3 presents the results of the SW normality test applied to the destruction runtimes of network scenarios using IaC tools. As in Table 2, descriptive statistics are provided, including the mean, median, and mode of the execution time in seconds for each network topology (Single, Linear, and Tree). The table also includes the W statistic of the SW test and the corresponding p-value. The results reveal that all p-values are below 0,05, indicating that the population of execution time data for destroying the scenarios does not follow a normal distribution. This evidence reinforces the need to use non-parametric methods for statistical analysis of the data, as the normality assumption is not met, which may affect the interpretation of the results and comparison between the IaC tools.

Table 4 presents the overall median of each tool in the creation and destruction scenarios. We calculated each median combining results obtained from tests conducted on the Single, Linear, and Tree topologies, considering the scenario in execution and the IaC tool used. This procedure facilitates the formulation and validation of our hypotheses.

5.2.2. Hypothesis Testing

The non-parametric Mann-Whitney (MW) test, called Mann-Whitney U test, was selected for the analysis because it is suitable for comparing two independent samples and because

Table 4. Overall median of IaC tools

Scenario	Median in Ansible	Median in Terraform
Creation	261,0	180,0
Destruction	63,0	49,0

the collected data do not follow a normal distribution. The MW test compares the medians of the samples without assuming normality and can be applied to both large and small samples, offering flexibility and accuracy in data analysis under these conditions.

This test is essential to determine if there is a significant difference in performance, in terms of runtime, between the IaC tools Ansible and Terraform in the analyzed scenarios. To conduct it, the following hypotheses were formulated, as shown in Table 5:

Table 5. Hypotheses about the efficiency of Ansible and Terraform in creating and destroying scenarios

Hypothesis	Description
H_0	Ansible and Terraform have similar efficiency in creating the scenarios.
H_1	Terraform is more efficient in creating the scenarios than Ansible.
H_2	Ansible is more efficient in creating the scenarios than Terraform.
H'_0	Ansible and Terraform have similar efficiency in destroying the scenarios.
H'_1	Terraform is more efficient in destroying the scenarios than Ansible.
H'_2	Ansible is more efficient in destroying the scenarios than Terraform.

5.2.3. Validation of Hypothesis Tests

Table 6 presents the results of the MW hypothesis test for the creation of the scenarios. The Mann-Whitney U test produced a statistical value of 86.855,5 and a p-value of 1,6511e-85, indicating a significant difference between the performance of the tools. Based on these results, we reject the null hypothesis, as the p-value is much smaller than 0,05. Next, we compared the other two hypotheses to identify which tool offers superior performance, measured by the shorter runtime in the proposed scenario. Thus, we accept H_1, which states that Terraform is more efficient than Ansible in creating the scenarios, as it shows a statistical value compared to H_2, which does not. This indicates that H_1 is true, reflecting Terraform's superior performance, with a shorter median runtime.

Table 6. Hypotheses about the efficiency of Ansible and Terraform in creating scenarios

Hypothesis	Statistic	p-value	Conclusion
H_0	N/A	1,6511e-85	Rejected
H_1	86855,5	1,6511e-85	Accepted
H_2	N/A	1,6511e-85	Rejected

Table 7 shows the hypothesis test results for the destruction of the scenarios. The Mann-Whitney U test resulted in a statistical value of 78.028,5 and a p-value of 2,6834e-55, again indicating a significant difference between the performance of the tools. In this

case, we reject the null hypothesis, as the p-value is much smaller than 0,05. Then, we analyzed the other two hypotheses to identify which tool performs better, requiring less time in the proposed destruction scenario. Thus, we accept H_1, which states that Terraform is more efficient than Ansible in destroying the scenarios, as it shows a significant statistical value while H_2 does not. This reflects that H_1 is true, demonstrating that Terraform also shows superior performance, with a shorter median runtime compared to Ansible.

Table 7. Hypotheses about the efficiency of Ansible and Terraform in destroying the scenarios

Hypothesis	Statistic	p-value	Conclusion
H'_0	N/A	2,6834e-55	Rejected
H'_1	78028,5	2,6834e-55	Accepted
H'_2	N/A	2,6834e-55	Rejected

5.3. Discussion of the Results

Based on results, Terraform presented lower median runtimes than Ansible in both the creation and destruction of the evaluated topologies, and the statistical tests indicated that these differences were significant under the experimental conditions adopted. However, these findings should be interpreted with caution, since the observed behavior is associated with the specific environment, topology set, execution workflow, and provisioning model used. Although the provisioning scripts were designed to be as similar as possible in terms of infrastructure objectives and overall workflow, differences inherent to each tool's execution architecture may still influence runtime behavior.

The differences identified may results not only from a tool's characteristics, but also to the structure of the topologies and to the way provisioning tasks were modeled. The measured runtime includes not only the provisioning of cloud resources, but also the preparation of the EC2 instance and the initialization of the emulated SDN scenario with Docker, Containernet, and the topology scripts. Therefore, part of the variation may reflect the interaction between tool behavior, topology organization, and the sequence of steps required to make the scenario fully operational, rather than absolute tool advantage.

Terraform's favorable behavior is related not only to its declarative syntax, but also to how it organizes execution. Terraform relies on persisted state to map managed resources to the configuration and uses dependency-aware planning to execute operations, including destruction. By contrast, Ansible follows an agentless, task-oriented model coordinated from a control node, typically through SSH. In the workflow adopted here, where the scenario depends on preparation steps inside the provisioned EC2 instance, part of Terraform's lower runtime may be associated with its state-based execution model, while Ansible may incur overhead from the ordered execution of playbook tasks.

This interpretation is especially relevant for the destruction phase. According to Terraform documentation, destroy mode operates against the remote objects recorded in the Terraform state, leaving an empty state after execution. This gives Terraform a direct view of which managed resources exist and how they relate, whereas Ansible does not maintain an equivalent built-in state structure for infrastructure management. Thus, the lower destruction runtimes observed for Terraform can be interpreted as consistent with its execution model, although this result remains bounded to the adopted design.

6. Limitations and Threats to Validity

With respect to *internal validity*, although experiments were conducted in an automated manner and under controlled conditions, execution times vary due to transient factors inherent to cloud computing environments, such as resource contention and scheduling.

Regarding *external validity*, the experimental evaluation was carried out using a single cloud provider and a specific instance type. Therefore, the results are not directly generalizable to other platforms, infrastructure configurations, or execution environments. In addition, the study considers only three predefined SDN topologies, which do not cover the full diversity of network scenarios observed in real-world environments.

Regarding *construct validity*, execution time was adopted as the metric. Although this metric is suitable for comparing provisioning efficiency, it does not capture other relevant aspects, such as resource utilization, fault tolerance, or operational complexity.

Finally, to mitigate these limitations, the experiments were repeated under equivalent conditions and all scripts, configurations, and artifacts used were made publicly available, allowing reproducibility and independent validation of the results.

Additionally, communication between the local execution environment and the cloud infrastructure can introduce minor latencies during the initial control and orchestration phases, without affecting the internal execution of the provisioning process.

7. Conclusions

In this work, a comparative analysis was conducted between the IaC tools Ansible and Terraform in the context of implementing emulated network topologies in cloud environments. The results indicated that, within the experimental conditions adopted in this study, Terraform achieved lower execution times than Ansible for both the creation and destruction of the evaluated topologies. In addition, the statistical analysis confirmed that these differences were significant for the analyzed scenarios. Nevertheless, these findings should be interpreted according to the specific topologies, cloud configuration, execution workflow, and provisioning model adopted in the experiments, rather than as evidence of a universal advantage of one tool over the other.

The study also highlighted the effectiveness of integrating NetDevOps practices with tools such as Containernet and Docker, which proved suitable for building flexible and scalable experimental network scenarios. The results obtained provide relevant insights for professionals and researchers in networking and cloud computing, particularly regarding the selection of IaC tools for automation in SDN-based environments.

This work also opens several directions for future research. Potential extensions include the evaluation of additional IaC tools, such as CloudFormation, OpenTofu, Puppet, Pulumi, and Chef, as well as the exploration of more complex and large-scale network topologies. Furthermore, future studies may investigate the adoption of more realistic network emulation platforms, such as Containerlab, to increase the fidelity of experimental environments and better approximate real-world deployment conditions. These extensions may contribute to a deeper understanding of automation efficiency in SDN and cloud-based infrastructures.

References

Ansible (2025). Documentation. https://docs.ansible.com/ansible/latest/. ac. 29 dec. 2025.

Bali, M. and Walia, R. (2023). Enhancing efficiency through infrastructure automation: An in-depth analysis of infrastructure as code (iac) tools. In *2023 Intl. Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 857–863.

Bholebawa, I. and Dalal, U. (2016). Design and performance analysis of openflow-enabled network topologies using mininet. *International Journal of Computer and Communication Engineering*, 5:419–429.

Brikman, Y. (2022). *Terraform: Up and Running*. " O'Reilly Media, Inc.".

Containernet (2026). Containernet use docker containers as hosts in mininet emulations. https://containernet.github.io/. accessed 02 jan. 2026.

Di Lena, G., Tomassilli, A., Saucez, D., Giroire, F., Turletti, T., and Lac, C. (2021). Distrinet: a mininet implementation for the cloud. In *SIGCOMM Comput. Commun. Rev.*, 51(1):251–258.

Elradi, M. D. (2023). Ansible: A reliable tool for automation. *Electrical and Computer Engineering Studies*, 2(4).

Hussain, M., Shah, N., Amin, R., Alshamrani, Alotaibi, A., and Raza, S. (2022). Software -defined networking: Categories, analysis, and future directions. *Sensors*, 22(15):5551.

Kodolov, S., Klimova, A., Aksyonov, K., and Filimonov, A. (2020). Using the iac approach for building a distributed laboratory complex for modern communication infrastructures investigation. In *2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, pages 0440–0443.

Leite, L., Rocha, C., Kon, F., Milojicic, D., and Meirelles, P. (2019). A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6).

Morris, K. (2020). *Infrastructure as code*. O'Reilly Media.

Noll, J. L. and Pretto, F. (2020). ImplementaÇÃo de infraestrutura como cÓdigo para provisionamento e deploy de aplicaÇÕes. *Revista Destaques Acadêmicos*, 12(4).

Preston, H. (2024). What does "network as code" mean?. https://blogs.cisco.com/developer/what-does-network-as-code-mean.

Salazar-Chacón, G. and Parra, D. M. (2023). Infrastructure-as-code in open-networking: Git, ansible, and cumulus-linux case study. In *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1126–1131.

Shah, J. A. and Dubaria, D. (2019). Netdevops: A new era towards networking devops. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 0775–0779.

Sicoe, A.-F., Botez, R., Ivanciu, I.-A., and Dobrota, V. (2022). Fully automated testbed of cisco virtual routers in cloud based environments. In *2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 49–53.

Terraform (2026). What is terraform? https://developer.hashicorp.com/terraform/intro. ac. 02 jan. 2026.