Infrastructure as Code with Terraform: A Comparative Study of Manual and Automated Infrastructure Provisioning on Amazon Web Services

Omkar Balaso Pawar, Aniket Ganesh Honrao, Azad Bashir Shaikh, Pavan Dnyanoba Magar
Department of Master of Computer Applications, Dr. D. Y. Patil Institute of Master of Computer Applications and Management, Akurdi, Pune, India
Guide: Arati Patil

***Abstract*—**Infrastructure as Code (IaC) describes cloud resources in files that can be reviewed, versioned, and reused. Terraform is a widely used IaC tool that can create Amazon Web Services (AWS) resources from declarative configuration. This paper presents a framework for comparing manual provisioning through the AWS console with provisioning using Terraform. The framework looks at provisioning time, hands-on effort, configuration errors, correctness, repeatability, and maintainability. The proposed experiment builds the same AWS environment with both approaches, using repeated trials and one common validation checklist. To show how the analysis and formatting would work, the results section uses a clearly labelled hypothetical dataset of five trials per approach. In this dataset, the manual process averages 33.8 minutes and the Terraform apply stage averages 12.0 minutes, a difference of 64.5% for that stage. This is not a fair end-to-end comparison, because the Terraform figure leaves out the time spent writing the configuration. The values are examples only and are not real observations. The paper concludes that a valid comparison must measure both initial setup and repeated deployment, use equivalent infrastructure, report errors honestly, and consider security and cloud cost.

Index Terms - Infrastructure as Code, Terraform, AWS, cloud computing, DevOps, provisioning, automation, repeatability.

I. INTRODUCTION

Cloud computing gives on-demand access to computing, storage, and networking resources. These resources can be created through a web console or with automation tools. Manual configuration works well for small learning exercises, but repeating many steps takes time and can lead to inconsistent settings. Infrastructure as Code tackles this by describing the desired infrastructure in configuration files that can be reviewed and reused [1], [2], which also fits the wider aim of reliable and repeatable deployment [12].

Terraform lets users define infrastructure in configuration files and applies changes only after generating a plan. In an AWS environment, it can manage resources such as VPCs, subnets, route tables, security groups, and EC2 instances. Automation does not guarantee correctness, though. Configuration files can contain mistakes, permissions can be too broad, and state files can expose sensitive information if they are handled carelessly [3]–[6]. This paper compares manual provisioning with Terraform-based provisioning for a defined AWS environment. The goal is to judge the practical differences through measurable criteria instead of assuming that automation is always faster or safer.

A. Problem Statement

When cloud infrastructure is created by hand, the operator has to repeat console actions and remember the intended settings. Repetition can lead to missed steps or differences in networking, naming, and security rules. Terraform can store the intended configuration in code, but it adds an upfront cost for writing and validating that code, and it needs careful state management. This study asks how the two approaches compare in time, correctness, repeatability, and maintainability when they create the same AWS resources.

B. Aim and Objectives

The aim is to compare manual and Terraform-based provisioning of a defined AWS environment using measurable operational and quality criteria. The objectives are to:

- measure total elapsed time and active operator effort for both approaches;
- record configuration mistakes and validation results;

- check whether repeated trials reproduce the intended environment;
- compare how easy it is to review, change, and reuse the infrastructure definitions; and
- identify security, cost, and methodological limitations.

C. Research Questions

RQ1: How do provisioning times differ between manual AWS console work and Terraform?

RQ2: Which approach gives more consistent results across repeated trials?

RQ3: What kinds of configuration errors occur?

RQ4: How do initial setup effort and later repeat deployments affect the comparison?

II. LITERATURE REVIEW AND RESEARCH GAP

A. Related Work

Rahman, Mahdavi-Hezaveh, and Williams mapped the research on IaC and found topics such as tools, adoption, empirical studies, testing, defects, and security flaws [1]. Their work supports looking at both the benefits and the risks of IaC. Morris describes IaC as a way to manage infrastructure through repeatable definitions and software engineering practices [2].

Studies on configuration-code quality show why infrastructure definitions should be reviewed like software. Sharma, Fragkoulis, and Spinellis examined configuration-code smells [3], and Schwarz, Steffens, and Lichter studied code smells in IaC [4]. Rahman, Parnin, and Williams looked at security smells in IaC scripts and showed that configuration mistakes can have security consequences [5]. These studies support including correctness and security checks alongside timing.

The official Terraform documentation describes the workflow of initialisation, validation, planning, applying, and state management [6]. AWS documentation gives guidance on cloud resources and infrastructure management [7], and AWS CloudFormation offers a native alternative for declarative provisioning [11]. The NIST definition of cloud computing gives a basic description of cloud service characteristics [8]. These technical sources guide the implementation design, but they cannot replace independent experimental measurement.

Recent comparisons also show the value of controlled testing. Regvart, Vlahović, and Balković compared Terraform, Pulumi, and CloudFormation using a standardised AWS architecture with repeated deployments [9]. A 2026 applied paper indexed by SCITEPRESS compares Terraform with manual AWS console provisioning and discusses provisioning time, accuracy, and security validation [10]. These results are useful context, but their numbers cannot be carried over to this study, because the architecture, timing definitions, and experimental conditions may differ. Table I summarises how each source is used.

TABLE I. LITERATURE SYNTHESIS

Source	Contribution	Use in this paper
Rahman et al. [1]	Maps research themes in IaC	Frames the research gap and need for empirical evaluation
Morris [2]	IaC concepts and practice	Defines repeatability and infrastructure management
Sharma et al. [3]	Configuration code quality	Motivates maintainability checks
Schwarz et al. [4]	IaC code smells	Highlights code structure and quality concerns
Rahman et al. [5]	Security smells in IaC	Supports security validation
HashiCorp [6], AWS [7]	Technical documentation	Guides workflow and AWS implementation
Regvart et al. [9]	Controlled IaC tool comparison	Illustrates repeated-trial evaluation

B. Research Gap

Existing literature explains IaC concepts and points out code-quality and security concerns. Published tool comparisons also use different architectures and measurement rules, so their results do not automatically apply to a small AWS lab. This study addresses a practical gap by describing a repeatable comparison of manual and Terraform provisioning. It defines the timing boundaries, uses the same validation checklist, and keeps first-time configuration effort separate from later deployments. The gap is specific to this context; it is not a claim that no earlier comparison exists.

III. RESEARCH METHODOLOGY

A. Research Design

The proposed design is a quantitative comparative experiment. Both approaches must create the same target infrastructure in the same AWS region, with the same permissions and resource specifications. At least five measured trials per approach should be run where possible, after separate practice runs. All trials, including failed ones, are recorded.

B. Proposed AWS Environment

A suitable lab environment contains one VPC, two public subnets, two private subnets, an Internet Gateway, route tables, security groups, and a small EC2 instance. A NAT Gateway should be included only if it is required, because it adds cost. The target configuration, resource names, CIDR ranges, region, and security rules must be identical for both methods. Fig. 1 shows the proposed layout.

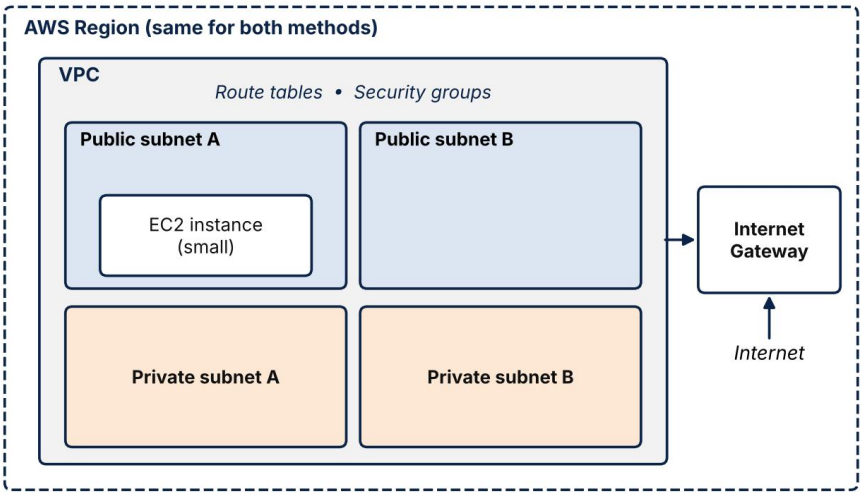


Fig. 1. Proposed AWS lab environment, identical for both provisioning methods.

C. Variables and Measurements

The variables and their definitions are listed in Table II.

TABLE II. VARIABLES AND MEASURES

Type	Measure	Definition
Independent	Provisioning method	Manual console or Terraform
Dependent	Elapsed time	Start event to successful validation; specify whether code-writing is included
Dependent	Hands-on effort	Active operator minutes, recorded separately from cloud waiting time
Dependent	Error count	Distinct incorrect settings or omissions
Dependent	Correctness	Checklist items passed divided by total checklist items
Controlled	Region, resources, permissions	Keep the same or document unavoidable differences

D. Procedure

1) Define the target architecture and the checklist before timing begins.
2) Create the resources manually and record the start and end times, effort, errors, and validation results.
3) Prepare the Terraform files, validate the syntax, and review the execution plan. Record the code-writing time separately.
4) Run Terraform init, validate, plan, and apply, and record the apply time and the total elapsed time separately.
5) Verify both environments using the same checklist.
6) Repeat the trials, record any failures, and summarise the measurements.
7) Delete the resources safely and confirm that no chargeable resources remain.

The procedure is summarised in Fig. 2.

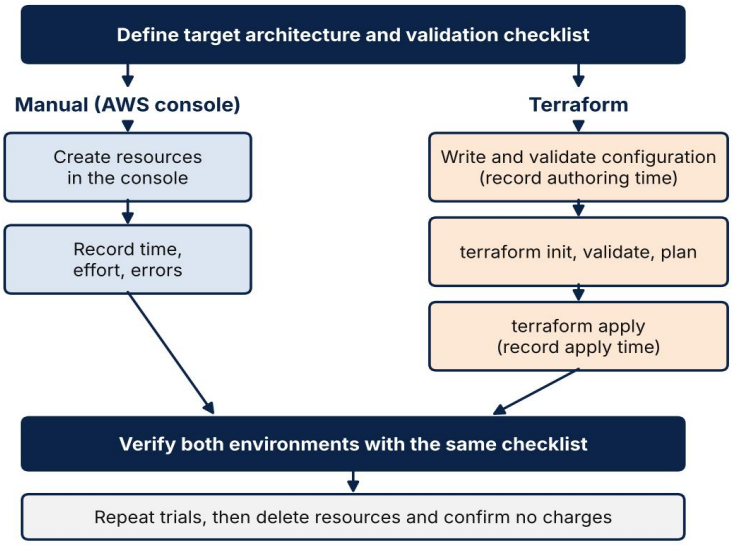


Fig. 2. Experimental procedure for the manual and Terraform provisioning methods.

E. Data Analysis

The analysis calculates the mean and median for each time measure. It also reports individual trial values and summarises error counts and correctness percentages. The percentage difference in measured time is given by:

$$Difference\ (\%) = ((T_{manual} - T_{Terraform}) \div T_{manual}) \times 100 \quad (1)$$

where T denotes the mean elapsed time of each method. Equation (1) is meaningful only when the time boundaries are comparable. If Terraform authoring time is excluded, the result must be labelled as apply-stage time rather than total setup time.

IV. ILLUSTRATIVE EXAMPLE: HYPOTHETICAL DATA

To show the analysis, this section uses a set of five trials for each approach. The values were chosen for demonstration and are not real results; they should not be read as proof of real performance. Table III lists the trials, and Table IV gives a summary.

A. Example Trial Data

TABLE III. HYPOTHETICAL TRIAL DATA (NOT MEASURED)

Trial	Manual (min)	Terraform apply (min)	Manual errors	Terraform errors
1	34	12	2	1
2	31	11	1	0
3	36	13	2	1
4	33	12	1	0
5	35	12	2	1

B. Summary of Hypothetical Results

TABLE IV. SUMMARY OF HYPOTHETICAL RESULTS

Metric	Manual example	Terraform example	Interpretation
Mean time	33.8 min	12.0 min	64.5% lower for apply stage only; not end-to-end
Total errors	8	3	Simulated counts only
Checklist correctness	94%	100%	Based on a hypothetical 10-item checklist
Mean hands-on effort	33.8 min	18.0 min	Terraform example excludes code-authoring time

C. Interpretation

In this example, the Terraform apply stage takes less time than the manual setup process. It also shows fewer configuration errors and a higher checklist score. However, these results were deliberately chosen as example values. Since the time taken to write the Terraform code is not included, the example does not prove that Terraform is faster for the first deployment. A fair final experiment should report (a) the time spent writing and validating Terraform, (b) the apply-stage time, (c) the effort for the first-time deployment, and (d) the time needed for later repeated deployments. Hypotheses should be accepted or rejected only after collecting real data.

V. DISCUSSION

IaC can make infrastructure easier to review and recreate, because the intended resources are shown in version-controlled files [1], [2], [6]. At the same time, research on code smells and security smells shows that automation can repeat incorrect settings if the configuration is wrong [3]–[5]. As a result, deployment speed alone is not enough to determine success. A good comparison should look at correctness, security, repeatability, change review, and the time needed to maintain the configuration. A final discussion should compare the results with the research questions and the literature. It should say where the data supports the hypotheses and where it does not. It should also identify the conditions that may explain the outcome.

VI. LIMITATIONS AND VALIDITY

Operator experience can influence the results. The order in which trials are run may also affect measurements. Delays in the AWS API can add variability, and resource quotas might limit what can be tested. The choice of AWS region and the exact architecture used in the experiment can also change the outcomes.

Terraform requires time to learn and to write code. If only the apply time is measured, the initial learning cost may be overlooked. Five trials per approach give a small sample, so the results should be seen as descriptive rather than generalisable.

The test setup uses one AWS design. That design cannot represent every cloud provider or every production workload. This experiment is not a security audit, and passing the checklist does not prove that the setup is completely secure.

VII. SECURITY AND COST CONSIDERATIONS

A separate test environment should be set up. Only the minimum required permissions should be granted, using IAM roles and policies. Never store passwords, private keys, or any sensitive data in version control systems. Avoid exposing SSH or RDP ports to the internet. Always review the Terraform plan output before applying it, to make sure no unexpected changes will happen. Keep track of AWS costs for NAT Gateway usage, compute resources, storage, and public IPv4 addresses. After testing is finished, delete all resources and confirm that they are gone, to avoid unnecessary charges.

VIII. CONCLUSION

This study offers a way to compare manual AWS provisioning with Terraform-based Infrastructure as Code. It draws on established literature and a practical method that looks at timing, effort, errors, correctness, and repeatability for both approaches, together with security. The hypothetical results show how measurements might be presented, but they do not show real-world performance. A final research conclusion must come from trials in which both methods create the same resources and the timing boundaries of each method are reported openly. Future work might compare Terraform with AWS CloudFormation [11], include more trials, test more architectures, and add automated security checks.

REFERENCES

[1] A. Rahman, R. Mahdavi-Hezaveh, and L. A. Williams, “A systematic mapping study of infrastructure as code research,” Information and Software Technology, vol. 108, pp. 65–77, 2019, doi: 10.1016/j.infsof.2018.12.004.

[2] K. Morris, Infrastructure as Code: Managing Servers in the Cloud, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021.

[3] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?,” in Proc. 13th Working Conf. Mining Software Repositories (MSR), 2016, pp. 189–200, doi: 10.1145/2901739.2901761.

[4] J. Schwarz, A. Steffens, and H. Lichter, “Code smells in infrastructure as code,” in Proc. 12th Int. Conf. Quality of Information and Communications Technology (QUATIC), 2018, doi: 10.1109/QUATIC.2018.00040.

[5] A. Rahman, C. Parnin, and L. Williams, “The seven sins: Security smells in infrastructure as code scripts,” in Proc. IEEE/ACM 41st Int. Conf. Software Engineering (ICSE), 2019, doi: 10.1109/ICSE.2019.00083.

[6] HashiCorp, “Terraform documentation.” [Online]. Available: https://developer.hashicorp.com/terraform/docs (accessed Oct. 12, 2026).

[7] Amazon Web Services, “AWS documentation.” [Online]. Available: https://docs.aws.amazon.com/ (accessed Oct. 12, 2026).

[8] P. Mell and T. Grance, The NIST Definition of Cloud Computing, NIST Special Publication 800-145, 2011, doi: 10.6028/NIST.SP.800-145.

[9] D. Regvart, I. Vlahović, and M. Balković, “A controlled comparative evaluation of infrastructure as code tools: Deployment performance and maintainability across Terraform, Pulumi, and AWS CloudFormation,” Applied Sciences, vol. 16, no. 6, Art. no. 2971, 2026, doi: 10.3390/app16062971.

[10] [Authors to be confirmed], “Comparative study of Terraform-based Infrastructure-as-Code and manual AWS console provisioning,” SCITEPRESS, 2026. [Online]. Available: https://www.scitepress.org/publishedPapers/2026/148730/pdf/index.html

[11] Amazon Web Services, “AWS CloudFormation User Guide.” [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html (accessed Oct. 12, 2026).

[12] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Boston, MA, USA: Addison-Wesley, 2010.