CONTINUOUS IMPLEMENTATION THROUGH STANDARDIZED AND COMPLIANT INFRASTRUCTURE AS CODE

S. A. Udara Jayawardana

199379X

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science & Engineering

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

May 2021

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgment any material previously submitted for a degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to the University of Moratuwa the non - exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic, or another medium. I retain the right to use this content in whole or part in future works.

Signature: ..... Date: .....

Name: S.A.U. Jayawardana

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the In19-S3-CS5999 PG Diploma Project.

UOM Verified Signature

Signature: ..... Date: .....

Name: Prof. Gihan Dias

ACKNOWLEDGEMENT

I am extremely grateful to my supervisor, Prof.Gihan Dias, Department of Computer Science and Engineering, University of Moratuwa, for his constant guidance, which played a major part in successfully completing the research conducted and successful thesis. I would like to further extend my gratitude to my examiners Dr.Sunimal Rathnayake and Dr.Lochandaka Ranathunga for their invaluable insights and feedback.

I am fortunate to be a part of the DevOps and Site Reliability Engineering community, which took some valuable time out of their busy schedules to fill out the survey and share their current Infrastructure as Code related practices, which helped in identifying real-world difficulties and roadblocks related to IaC best practices.

Very special thanks to my family's unconditional support and continuous encouragement

ABSTRACT

With cloud computing becoming the new norm and organizations embracing cloud services benefits, the infrastructure provisioning methods of the old are quickly becoming obsolete. Infrastructure as Code concept introduced as an answer for this with provisioning infrastructure in an automated manner, with specifications defined on a machine-readable code. IaC made dynamic provisioning and modification on cloud resources possible, enabling organizations to utilize the full benefits of the cloud.

However, IaC without proper standardization and compliance could result in disastrous outcomes. In order to achieve this, the industry looked into the Software Engineering practices, due to IaC's similarities to coding. Though it may look similar, this proved to be less effective. Therefore, testing, compliance, and standardization methods, specifically tailored for IaC are required. A standardized and compliant IaC will make way to implement *Continuous Implementation*.

Immutable infrastructure is a major roadblock that can inhibit harnessing the benefits of cloud ecosystems. Though many organizations use Continuous Integration (CI) and Continuous Deployments (CD) for code deployments, the infrastructure & configurations mostly remain unchanged. However, infrastructure should follow the same principle of frequent updates, to get the best out of ever-changing cloud infrastructure.

This research focuses on introducing the concept of Continuous Implementation. Continuous Implementation pipelines will be evaluated with the traditional and currently widely-used infrastructure provisioning methods. A standardized IaC framework will be used to support fully automated infrastructure provisioning, modification, and configuration management, on imposing the organizational and security policies. Through the results obtained, a study was conducted on determining the importance of Continuous Implementation for cloud-based infrastructure.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
IAC	Infrastructure as Code
DSL	Domain Specific Language
CI	Continuous Integration
CD	Continuous Deployment
SCM	Source Control Management
CSP	Cloud Service Providers
API	Application Programming Interface
K8s	Kubernetes
OASIS	Organization for the Advancement of Structured Information Standards
TOSCA	Topology and Orchestration Specification for Cloud Applications
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
PCI DSS	Payment Card Industry Data Security Standard
SOC2	Service Organization Control 2
AWS	Amazon Web Services
SRE	Site Reliability Engineer
WAF	Web Application Firewall

1. INTRODUCTION

1.1 Background

Cloud computing has become one of the major forces and significantly altered the entire IT landscape, widely acknowledged by analysts and companies alike. This has been the deciding factor on how data centers are built, how software designed and functions, how upgrades and deployments are handled, to ultimately how services are provided to the end-users.

Given the vital role that "Cloud" lays in the modern industry, it has given birth to the "Cloud Age" mindset. This mindset is fundamentally different from the traditional, "Iron Age" approach we used with static pre-cloud systems [1]. Cloud Age has broadened the horizons on to a dynamic approach as opposed to the static, defined ways. These dynamic applications are made possible due to the infrastructural changes that came along with the cloud age. The technologies make it faster to provision and change infrastructure than traditional, Iron Age technologies.

Iron Age	**Cloud Age**
Physical hardware	Virtualized resources
Provisioning or modifications takes weeks	Provisioning takes minutes
Manual processed	Automated processes

Table 1. Technology changes in the Cloud Age

Source: Kief Morris, Infrastructure as Code, 2nd edition

Regardless of the greener pastures provided by Cloud computing, this does make it easier to manage your infrastructure by moving to the cloud. Migrating a system with large technical debt to unmanaged and ungoverned cloud infrastructure could potentially drive the entire setup to chaos. However, the traditional well-proven

governance models can be applied to overcome this. Thorough up-front design, rigorous change review, and strictly segregated responsibilities are such few practices. However, these models are built with iron age infrastructure in mind, where changes are slow and expensive [1]. Imposing such governance models will inhibit the organization from achieving the full potential of cloud computing.

Cloud computing has fundamental operation differences compared to the iron age.

Iron Age	**Cloud Age**
The cost of change is high	The cost of change is low
Deliver in large batches, test at the end	Deliver small changes, test continuously
Long release cycles	Short release cycles
Favors monolithic architectures (fewer, larger moving parts)	Microservices architectures (more, smaller parts)
GUI-driven or physical configuration	Configuration as Code

Table 2. Design and operational changes in the Cloud Age

Source: Kief Morris, Infrastructure as Code, 2nd edition

Cloud Age technologies make it possible to rapidly provision and change infrastructure resources. “Infrastructure as Code” is the Cloud Age approach to managing systems that embrace continuous change for high reliability and quality [1].

Infrastructure as a Code refers to the practice of provisioning and management of infrastructure (networks, virtual machines, load balancers, and connection topology) through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools, in a descriptive model, based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration.

Benefits of Infrastructure as Code

1. Using IT infrastructure as an enabler for rapid delivery of value
2. Reducing the effort and risk of making changes to infrastructure
3. Enabling users of infrastructure to get the resources they need, when they need it
4. Providing common tooling across development, operations, and other stakeholders
5. Creating systems that are reliable, secure, and cost-effective
6. Easily replicate same configurations across multiple environments
7. Improving the speed to troubleshoot and resolve failures

Applications; which especially on Cloud can be identified as dynamic solutions. Designed to harness the best of Cloud Service Provider has to offer, thus increasing the high availability & performance at a lower cost. The software industry is gradually parting ways with the traditional software models and frequently updating the applications to improve performance, security, and usability. However, these are done through the code level and little attention was given to the underlying infrastructure.

Once the application architecture is finalized, the infrastructure will be provisioned for a newly designed app or a cloud adaptation of the existing application. However, upon provisioned at the beginning, the cloud infrastructure is being treated as the same as the immutable infrastructure on data centers.

The CSPs update and improve their service offerings at a rapid rate, mainly as a step to stay ahead of the competition. Due to the quality of these offerings, there are scenarios that an improvement to the application can be done far more effectively and efficiently through a change of the infrastructure than code.

1.2 Research Problem

As discussed in section 1.1, cloud environments are dynamic ecosystems. Infrastructure resources are rapidly changing to facilitate the requirements. The transition to cloud platforms has given benefits such as agility, scalability, and lower capital costs but the application lifecycle management practices are slow with this disruptive change. DevOps culture extends the agile methodology to rapidly create applications and deliver them across an environment in an automated manner to improve performance and quality assurance. Continuous Integration (CI) and Continuous Delivery (CD) have emerged as a boon for traditional application development and release management practices to provide the capability to release quality artifacts continuously to customers [2].

Through we utilized CI & CD for code deployments to rapidly deliver services to the user through cloud environments, little attention was given and limited research was carried regarding the rapid change of infrastructure needed to support the new code releases on application requirements. Immutable infrastructure is a major roadblock that can inhibit harnessing the benefits of cloud environments. In addition to the infrastructure modifications, configuration management has been a critical area of infrastructure management. Configuration of the server to support the application requirements are no longer done manually thanks to the configuration management tools; Ansible, Chef, and Puppet. Configuration management tools are used for version upgrades, installation of required support applications, setting up the server environment for the application hosting, etc.

Infrastructure should follow the same principle of frequent code updates with CI & CD, to get the best out of ever-changing cloud infrastructure by modifying the infrastructure on small scales use the best resources offered by the CSP, introduce cost-saving methods, and elevating the new changes introduced in a release and keep the installed support applications updated to eliminate any vulnerabilities.

Dynamic infrastructure cannot be maintained through manual configurations. To overcome this, Infrastructure as Code is used therefore, mass infrastructure provisioning and modifications can be done in an automated manner. However, IaC is

required to be properly standardized and provisioning resources must be compliant with the organizational and security policies to use it in a continuous and automated manner.

Standardization of IaC has been mostly achieved via peer reviews, although this can be a very tedious process. The standard process would be to use Git as both version control and the code review for Terraform, and a backend database for state locking to prevent multiple users from modifying the same stack simultaneously. Once the PR is made, the code/security reviewers will be reviewing the code and follow the same methodologies used for software code.

However, this cannot be recognized as a proper methodology to identify and mitigate the unique challenges that arise in the 'Operations domain' as code reviews are largely intended and catered for the 'Development domain'. An analysis of these is stated below.

1.Code review

The most preferred way due to the shallow learning curve. However, in scenarios deploying between tens and hundreds of different infrastructure components with tens or hundreds of servers each with multiple modules, code reviews cannot consider being reliable. Additionally, for compliance, multiple stakeholders with expertise in different areas such as cloud and security are required to go through the same set of code.

2. Pre-Testing in Sandbox

Relatively easier and efficient than code reviews as infrastructure will be created in a sandbox environment, which can contain AWS or 3rd party compliance rules, to conduct alpha testing. However, in small or large-scale automation alike, this adds an unwanted cost. Performing a full integration test within a sandbox implies that you mirror your entire infrastructure (albeit at a smaller scale with smaller compute sizes and dependencies) for a short time. Doing this to confirm simple changes can add up, especially if that mirror is sufficiently large.

3.IaC based Unit Tests

Can be considered as the best way to perform infrastructure-related testing. HashiCorp itself provides some unit testing functionalities and testing patterns [3] providing a reference for its functionality and introducing the basic parts of writing acceptance tests. However, though the unit testing can be used to test the terraform-related infrastructure defects it cannot be used to measure organization or project-specific compliance and security guidelines.

Therefore, we can state that the traditional methods, which have more general characteristics does not capable of providing neither the organizational standardization nor once deployed, the system will reach the desired state. Based on the analysis conducted research conducted by North Carolina State University [4], they have observed the lack of empirical studies that focus on this area. Due to the lack of studies and researches conducted, frequently changing the infrastructure on small scale has flown under the radar.

To summarize, continuous frequent changes to the infrastructure are required, opposed to major infrastructure changes few times per year, to utilize the best of cloud environments to the application. These changes required to be automated, following the same CI & CD theory to implement through lower environments to the production with comprehensive testing. IaC is the proven infrastructure provisioner on the cloud, supporting and adopted by all major CSPs. Furthermore, as IaC follows the software engineering principles, the existing CI & CD practices can be easily adapted for this.

Therefore, the *lack of frequent infrastructure updates, supported by a proper standardizing framework,* can be identified as the research problem.

1.3 Motivation

As explained in section 1.1, the majority of tech companies have made the transition to the cloud from on-prem data centers. The current business demands dynamic solutions, which require cloud technology capabilities. This organizational change has impacted engineering practices and the traditional methodologies are rapidly becoming obsolete.

I have worked with three major technology companies in Sri Lanka, in different business domains such as education, research & publishing, and food distribution. As all three companies made their transition to the cloud, they have faced similar challenges. Infrastructure was pre-designed in the architectural design phase and rarely got changed from the original state. Only infra changes came for a scenario such as a cost-saving initiative where resources were right-sized depends on the usage, but this was almost limited to non-prod environments. Configuration management was done more frequently for pre-planned patching cycles. With regards to IaC, infrastructure provisioning on cloud without compromising the same disciplines and organizational standards kept on the company's data centers was an issue. As infrastructure provisioning on the cloud drastically changed with the introduction of IaC but came with some challenges

However, IaC provided a systematic way to provision the infrastructure with capabilities to review how it will be provisioned, how compliance will be upheld, and track infrastructure drifts from the desired state. As IaC provided a way to build the infrastructure in a code-based way, it was logical to apply the proven and tested software engineering code review practices to IaC. Even though this was successful for small-scale stacks which can be reviewed with human eyes, large sophisticated infrastructure stacks along with constantly updating cloud service resources made it’s nigh impossible to manage via manual methods. Therefore, a dedicated method to standardize the validate the compliance was required.

If the IaC can be validated for organizational standards and resources are compliant to the policy base in an automated manner, this can be used to get infrastructure updates to the same code release cycles as software code. This is rather important because, with an ecosystem such as the cloud, code and the infrastructure cannot be treated as two separate sections. Correct provisioning and updating of the infrastructure required to achieve the intended performance of the application. Thus I believe infrastructure should get evaluated and updated at frequent release cycles to achieve maximum efficiency, reliability and avoid unnecessary costs.

Code releases are done in frequent cycles depends on the organization. A modern app usually follows bi-weekly releases (for bi-weekly sprints) or monthly releases combing two sprints of work. This was achievable due to sophisticated CI/CD pipelines that make the entire deployment and delivery in an automated and downtime-free manner. I believe a continuous pipeline dedicated to infrastructure should be used here, which introduces necessary infrastructure changes to the release through the same automated manner.

In order to conduct this research, I have selected HashiCorp's Terraform as the IaC provisioner, due to the following points

- Support for all major cloud services
- Open-source nature
- A large active community base

The large community will be particularly helpful in evaluating the existing solutions. Furthermore, the outcome solution of this research is largely benefitted from the community with code-level contributions and beta-testing.

As for the CSP, I will be using AWS, as it's the market leader with frequent rapid upgrades to its various cloud resources.

1.4 Purpose of the Research

As explained under section 1.2, in order to introduce automated continuous workflows for IaC, a standardization framework is vital for this as the requirement for standardization becoming a critical requirement for IT organizations.

- Organizational infrastructure standards
- International data & privacy standards
- Security and vulnerability assessments
- IaC code quality
- Infrastructure drifts

The purpose of this research is to explore the studies and researches conducted on workflows and pipelines and IaC based cloud infrastructure provisioning, to identify the progress made on the continuous and/or automated modification of cloud infrastructure and the standardization methodologies.
This will provide an insight into the gaps of the explored areas and findings/drawbacks of the similar studies conducted. The feedback of the questionnaire will grant insights n the current industry practices around the globe. This knowledge will be used to design the baseline of a standardization framework for IaC.

Upon designing the baseline for the proposed solution, the current commercial and open-source tools built with similar purposes and functionalities will be evaluated to find whether they can fulfill the requirements identified in the framework baseline.

Once the work is completed on the research, all the data will be used to design the outcome of the research, an open-source solution that supports automatic and continuous workflows, with a standardization framework to validate the quality of the code, to implement the cloud infrastructure

1.5 The Proposed Solution & the Objectives

This project aims to implement an automated workflow to continuously implement cloud infrastructure, supported by a standardized framework to evaluate the IaC code to validate security and organizational standards, to support application requirements or new cloud service provider resource offering in order to improve the performance, cost-efficiency, or any related non-functional requirement.

With regards to the facts presented in IaC in sections 1.1 to 1.3, we can conclude that infrastructure should frequently have refactored to the application objectives along with the code itself.

Continuous Integration (CI) & Continuous Delivery / Continuous Deployment (CD) is the base for this research's continuous workflow. Though the research has utilized the full idea of CI, traditional CD can not be applied for infrastructure as the product of them are different.

For code-based release, the same code is deployed to all environments (from dev to prod) as the product of the code: the functionality remains the same. But the same IaC code is not applied to all the environments as the product of IaC: infrastructure configurations are different from an environment to environment, only exceptions being UAT, a replica of Production. CD is based on deploying the code from dev to prod in a single pipeline. Thus, using the term CD here could create ambiguity in the definition.

This introduces the concept of *Continuous Implementation*, a term for traditional CI and a tailored version for CD; reserved for continuously updating infrastructure. Implementation refers to infrastructure implementation which covers the cloud resource provisioning, modifications, maintaining and configuration management.

Upon a new release is planned, the infrastructure will be re-evaluated to determine if the current stack is providing the best performance and reliability at the least cost. If changes are identified, infrastructure will be refactored to support the code release starting from lower environments to production. However, as infrastructure changes

will be fully automated and applied through continuous implementation (CIm) without manual intervention, a thorough validation should be conducted.

As unit, acceptance, and perf tests are used to validate the code, the IaC code is required to be evaluated as well. These belong to unit tests, standardization validation & compliance validation. The testing methodology should be fully designed with cloud computing principles in mind, therefore achieving the desired result without compromising any of the benefits an organization can harvest in a cloud-centric infrastructure ecosystem,

The testing and standardization are focused on three key aspects.

1. Security

    Security can be considered as the top priority in operations. Regardless of the size of the infrastructure and the base cloud platform, the tests should be able to measure the security violations specific to the project and the organization. The security should look into infrastructure level violations such as insecure firewall port openings to configuration management violations such as using a flagged version of a component for vulnerabilities such as the NodeJS package.

2. Organizational Compliance

    Each Organization has its internal compliance guidelines for technical operations. These can be a combination of industry best practices, results on expert audits specifically conducted on organization-owned applications or services, and lessons learned through past incidents. These compliance guidelines serve as the organization's best practices on development, infrastructure provisioning, and deploying. The code should be evaluated to measure the adherence to organizational compliance.

3. Automation

    Automation of recurring activities has become a must-have functionality in today’s IT sector. This provides ways to faster deployments with minimal

errors and downtimes. As code gets tested in Continuous Integration against pre-written test cases, IaC code should be validated before implemented, as a part of the Continuous Implementation pipeline and abort the implementation process if a violation has been detected.

During the performance and user acceptance testing on the code's CD pipeline, the new codebase should be evaluated against both existing and refactored infrastructures to measure the impact due to the changes. This can be used to determine the impact of the infra changes.

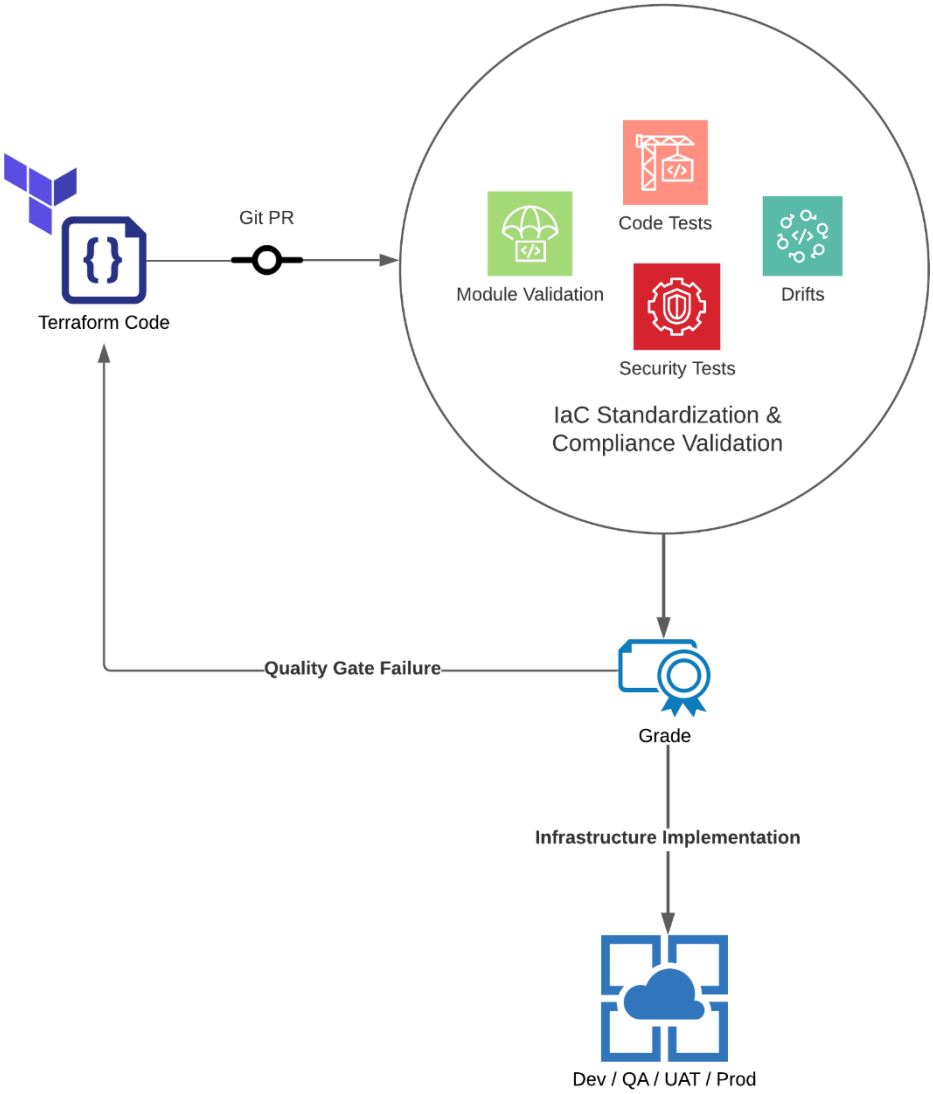


Figure 1. Continuous Implementation – Proposed Solution

2. LITERATURE REVIEW

2.1 Overview

IaC is the practice of automatically provisioning and management of remote or local infrastructure through machine-readable definition files, following declarative programming principles. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration. The world's leading technology companies such as FAANG (Facebook, Amazon, Apple, Netflix & Google) all adopting IaC provides a clear indication that IaC will be the cornerstone in infrastructure provisioning in the cloud computing era.

Following the popularity generated by the concept of IaC, both CSP and software companies have come up with various tools with lots of variety to cater to the demand. CSPs such as Amazon came up with "AWS CloudFormation", Google Cloud's "Cloud Deployment Manager" and Azure with "ARM Templates" as dedicated native resource provisioners of the respective cloud platforms.
Non-CSP companies such as HashiCorp developed "Terraform", an open-source, stateful, multi-vendor solution that codifies CSP API's configuration files. In addition to the cloud, Terraform supports SaaS tools such as monitoring solution StatusCake and container orchestration technologies such as Kubernetes.

Even though native IaC providers hold a distinct advantage over Terraform because new features are released in the respective platforms before similar features appear in Terraform, Terraform's non-vendor locking approach which encourages organizations to implement multi-cloud platforms harnessing best of each CSP and extended support to other services, such as largely popular Kubernetes (both vanilla Kubernetes and K8s based CSP offerings such as EKS), made it's the first IaC choice for most organizations. Terraform's support for the popular open-source configuration management tool, "Ansible" provides an additional advantage in providing both infrastructure and environment configuration management via all-in-one code.

Google Trends search interests indicate a steady growth for Terraform compared to the rest of the IaC offerings which states the dominance of Terraform.

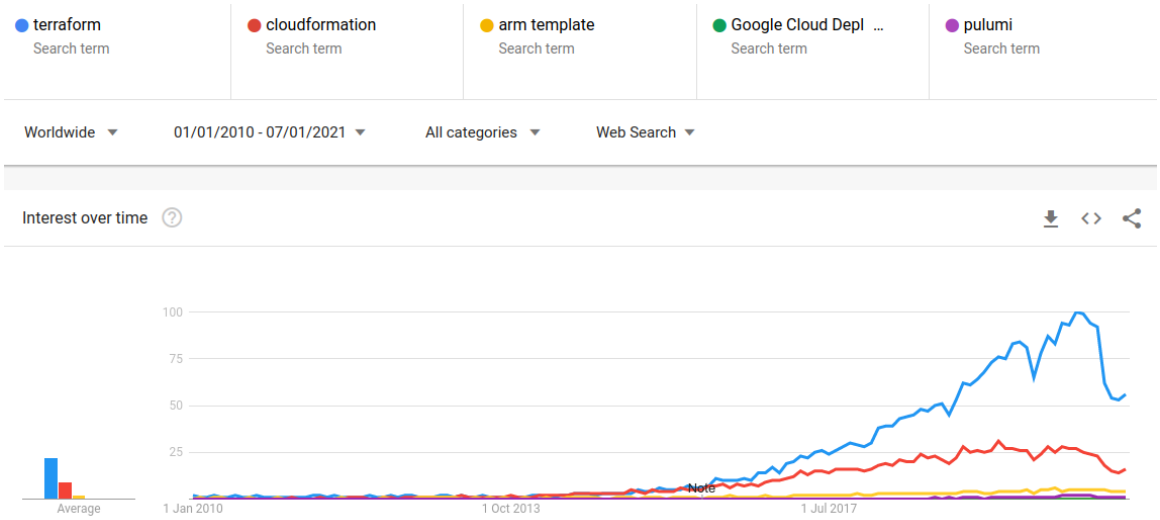


Figure 2. Interests in the IaC technologies in the last 10 years

Source: Google Trends

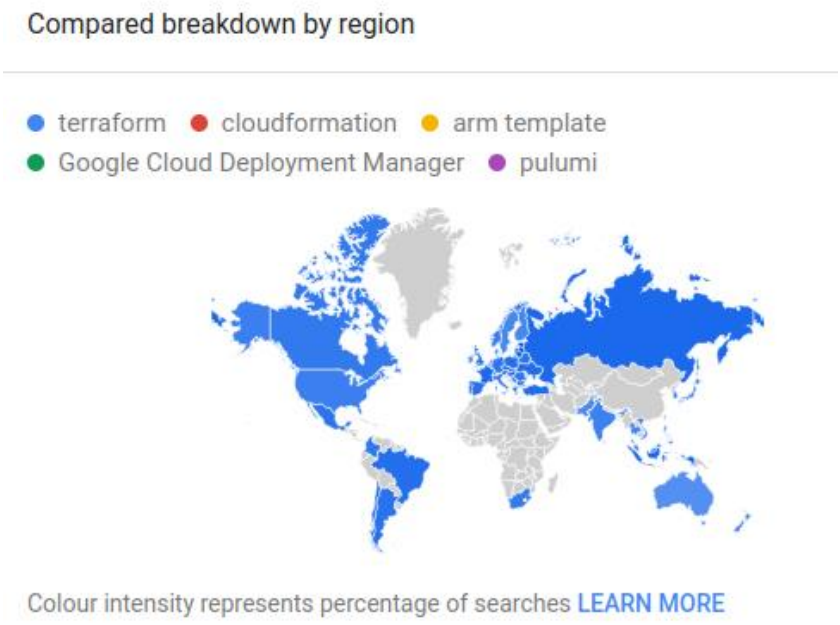


Figure 3. Interests in the IaC technologies in by region

Source: Google Trends

2.2 Prior Research Carried Under IaC

As there are various studies conducted on IaC with a large number of research papers and journal articles, I have focused on the researches that can be categorized into the below four major areas.

- Continuous Implementation of Infrastructure
- Software Engineering practices for quality on IaC
- Testing frameworks for IaC
- Standardization & Compliance

2.2.1 Research on the continuous implementation of infrastructure

A study conducted by Akond Rahman and Laurie Williams explores how leading companies such as Netflix, are using configuration management IaC for their code deploy pipelines to manage their software dependencies. They have used this technique to improve their deployment frequency [5]. Even though this focus on deployment pipelines with IaC, there was no mention of standardizing the IaC to add frequent changes to the cloud environment regularly in an automated manner. However, this can be taken as initial steps on raising the continuous implementation concept as the research further studies how defects of the IaC code can affect the automated pipeline.

Kief Morris in his book Infrastructure as Code discussed the potential of using CI/CD concepts for Infrastructure as Code. He states branching should be avoided as much as possible, as IaC scripts may have dependencies on inter-twined resources. Merging a branch with various changes to the truck may end up creating more merging issues due to IaC's connectivity and resource dependency on other resources.
CI with infrastructure as code involves continuously testing changes made to definition files, scripts, and other tooling and configuration written and maintained for running the infrastructure. Each of these should be managed in a VCS. Teams should avoid

branching, to avoid building up an increasing “debt” of code that will need to be merged and tested. Each commit should trigger some level of testing [1].

With regards to CD, he states the quality of the IaC code is the most important aspect as it can cause disastrous outcomes. Poor-quality infrastructure is difficult to maintain and improve. Choosing to throw something up quickly, knowing it probably has problems, leads to an unstable system, where problems are difficult to find and fix. Adding or improving functionality on a spaghetti-code system is also hard, typically taking surprisingly long to make what should be a simple change, and creating more errors and instability [1].

He further states the issue and impact of implementing a major change at once to the infrastructure. Swapping the old resource to a new one would take weeks of testing and monitoring. Even we identify an issue with this change, it would be rather difficult to pinpoint which caused the issue. However, continuous small changes could be beneficial compared to a major release as it takes little effort for testing as it can be focused on the implemented resource, and troubleshooting or fixing would take minimal effort.

2.3.2 Research on utilizing traditional frameworks for IaC

In a study conducted based on Puppet, Sharma et al. [6] discuss the wide adoption of configuration management and increasing size and complexity of the associated code, prompting the requirement for assessing, maintaining, and improving the configuration code. They have leveraged traditional software engineering knowledge and best practices associated with code quality management to propose 13 implementations and 11 design configuration anti-patterns. This study has shown the possibility of using the existing software engineering quality analysis to identify whether the code is syntactically valid and internally consistent such as misplaced attributes, improper alignment, incomplete conditions, and unguarded values.

Jiang and Adams [7] explored co-evolution of IaC along with other code tightly coupled with it, such as build files and source code, and concluded that IaC scripts should be considered as source code files not just because of the use of a programming language, but also because their characteristics and maintenance needs show the same symptoms as build and source code files. Thus, implicating using the traditional practices comes with programming languages, can be applied to IaC.

Schwarz et al. [8] discusses the code smells to a more extent, categorizing to three major areas naming,

1. Technology Agnostic Smells: ex – Improper alignment, Misplaced attribute, Duplicate block
2. Technology Dependent Smells: ex – Improper quote usage, Unstructured module
3. Technology Specific Smells: ex – Hyphens, Empty default

However, IaC cannot be categorized as a traditional programming language due to its primary design functionalities. Programming code and IaC scripts, although looks quite similar, are fundamentally different.

Terraform can be categorized as a declarative infrastructure as a code tool. Following core declarative principles Terraform is aware of its current state and follows the common syntax model pre-defined for a given resource. Though it has similar control structures such as loops to programming languages, the way logic is implemented to the same defined structure.

Therefore, Terraform engine can validate the smells that can occur and can be utilized for verification of reusable modules, including the correctness of attribute names and value types, misalignments, duplicates, unstructured modules, and unstructured modules [9].

2.2.3 Research on IaC Testing

Hanappi et al. [10] have studied IaC configuration management tools based on declarative resource descriptions. They have introduced a conceptual framework for asserting reliable convergence and utilized state transition graphs to test whether a script makes the system converge to the desired state under different conditions.

The authors have designed a modularized testing stack to carry out the tests.

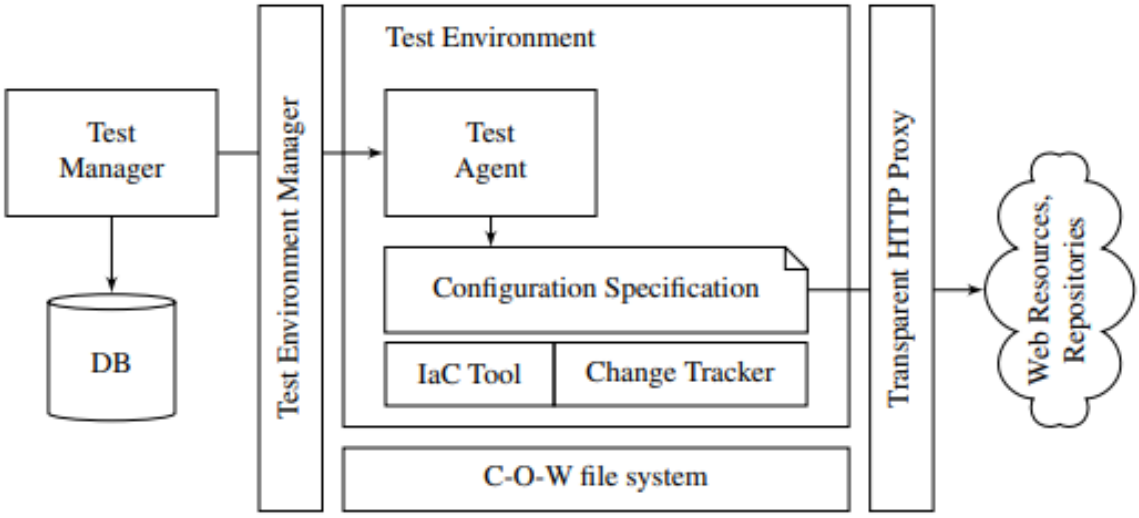


Figure 4. Hanappi Testing framework architecture
Source: O. Hanappi, W. Hummer, S. Dustdar, “Asserting reliable convergence for configuration management scripts”

The framework is made specifically for IaC, moving away from the traditional code testing approaches. It conducts a dedicated environment manager which brings the system to the desired state for every test dedicated module for IaC configuration. Authors conducted evaluation based on real-world Puppet scripts to detect all idempotence and convergence related issues in a set of existing Puppet scripts with known issues as well as some hitherto undiscovered bugs in a large random sample of scripts [10]. They have identified potential causes and issues that can occur in such a scenario.

Potential Causes

1. State dependency: a resource depends on the current system state
2. Resource Interactions:
    i. Input dependency: resource requires the output of a previous resource to execute
    ii. Output dependency: if multiple resources configure the same aspect of a system

Potential Issues

1. Single-Resource issues:
    i. State-dependent resources failing if they do not expect the system to be in the desired state
    ii. Resource failing to determine if the system is in the desired state, this reconfiguring it
2. Multi-Resource issues:
    i. Output conflict may occur when two output dependent resources failing to agree on a common desired state and thus conflicting with each other
    ii. Missing successor satisfaction check, thus aborting the workflow with a false-negative state

Ikeshita et al. [11] proposed a model-based testing framework. The model consists of two main parts:

1. A system model of the automation under test and its environment, including the involved

tasks, parameters, system states, and state changes

2. A state transition graph (STG) model that can be directly derived from the system model

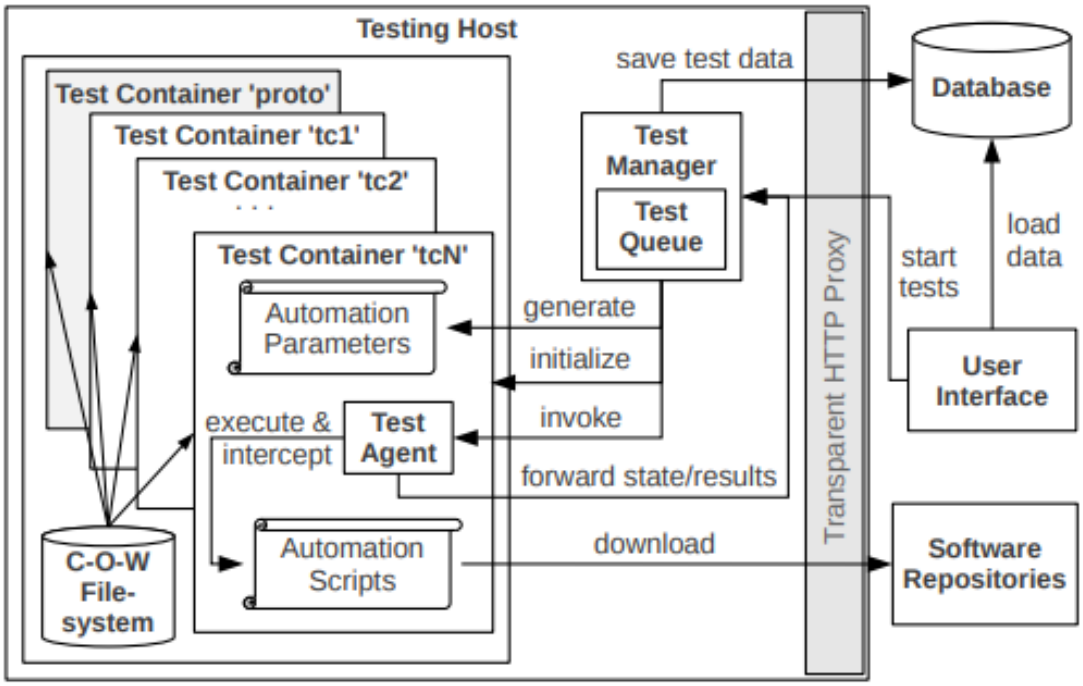


Figure 5. Ikeshita Testing framework architecture

Source: K. Ikeshita, F. Ishikawa, S. Honiden, “Test suite reduction in idempotence testing of infrastructure as code”

The tests are conducted with declarative scripts, thus looks into e idempotence and convergence problems (particularly those involving dependencies among multiple tasks) cannot be avoided solely by providing declarative and idempotence resource implementations [11].

Furthermore, the models designed for this framework differ from the Hanappi architecture due to the usage of highly efficient Linux containers (LXC). This provides a better landscape to test with easily provisioning containers for different OS verities and flavors. Ikeshita model has provided new pathways for conducting testing and designing standardized frameworks due to this.

2.2.4 Research on Standardization & Benchmarking Cloud

Sandobalin ae al. present support for the management of DevOps tools, through the definition of a Domain Specific Language (DSL) based on the concept of Infrastructure as a Code, and a tool that supports this language allowing to model the final state of a provisioning infrastructure in the cloud. They have looked into implementing TOSCA, a standard for Topology and Orchestration Specification for Cloud Application which allows modeling nodes (virtual or physical machines) and orchestrates the deployment of cloud applications [12]. The research conducted on building the model can be utilized to design a standardized framework.

The OASIS TOSCA is a new open cloud standard, supported by a large and growing number of industry leaders. In a nutshell, TOSCA provides the ability to be CSP-independent and move cloud services between different environments with standardized service definitions. With the multi-cloud approach is becoming an industry growing trend, utilizing best of all CSPs, the IaC scripts written adhering to TOSCA can help organizations to avoid vendor locking.

Wettinger, Breitenbücher, Kopp, and Leymann [13] present a systematic classification of DevOps artifacts and show how different kinds of artifacts can be discovered and transformed toward TOSCA. They have presented an integrated modeling and runtime framework to enable the seamless and interoperable integration of different approaches to model and deploy application topologies. The framework is implemented by an open-source, end-to-end toolchain.

Scheuner et al. [14] presented a benchmarking strategy for IaC. Though the study focuses on performance benchmarking on various IaC providers, the model they have used for Cloud WorkBench provides insight into designing a standardized framework. The authors discuss the challenges of designing a framework for a fast-moving cloud environment where IaC provisioner and CSP offering new features, supportability to implement and maintain non-functional requirements continuously.

Cloud WorkBench uses a web interface, which allows running the benchmarks in addition to viewing the results. The CWB server is the main component consisting of a standard three-tiered web application. It provides the web interface, implements the business logic in collaboration with external dependencies, and stores its data (definitions and results of benchmarks) in a relational Database.

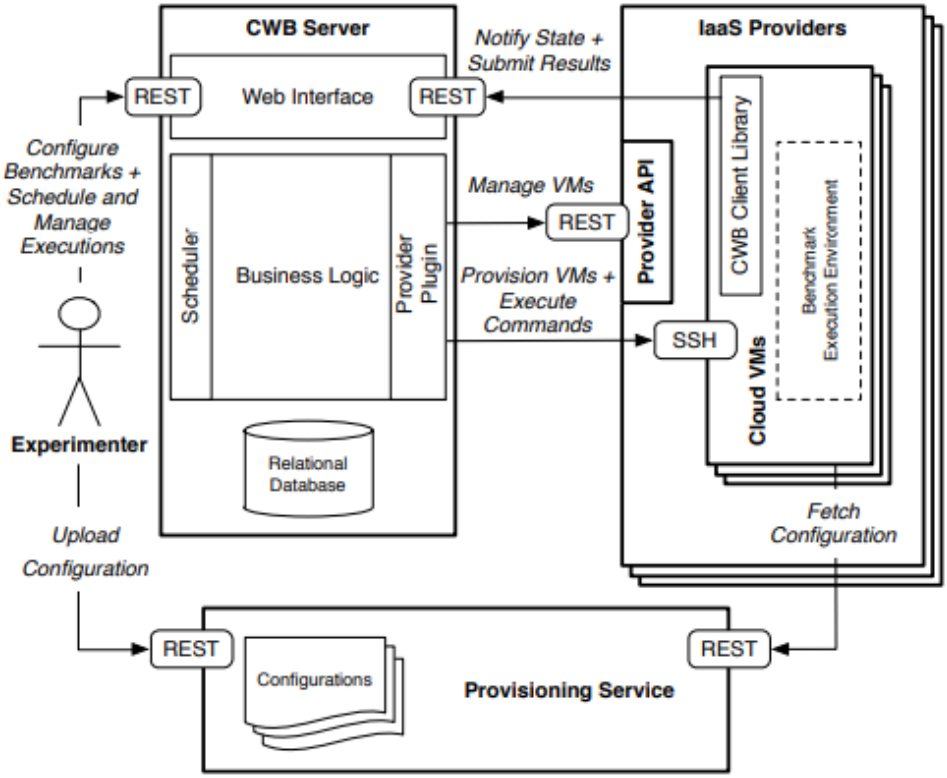


Figure 6. Cloud WorkBench architecture design
Source: J. Scheuner, P. Leitner, J. Cito, and H. Gall, “Cloud work bench - Infrastructure-as-code based cloud benchmarking”

Upon their comprehensive research on IaC which analyses 31 IaC publications, Rahman et al. [4] conclude defects and security flaws in IaC scripts can be one of the major causes which result in disastrous consequences, thus state the requirement for research studies that address code quality issues such as defects, adhering to the organizational and CSP standards and security flaws, along with exploring other research avenues.

2.3. Commercial and Open Source Tools

2.3.1 Acceptance and unit testing

Terraform acceptance tests use real Terraform configurations to exercise the code in the real plan, apply, refresh, and destroy life cycles. When run from the root of a Terraform Provider codebase, Terraform's testing framework compiles the current provider in-memory and executes the provided configuration in developer-defined steps, creating infrastructure along the way. After all the steps, Terraform automatically destroys the infrastructure [15]. Acceptance tests are expressed in terms of Test Cases, each using one or more Terraform configurations designed to create a set of resources under test, and then verify the actual infrastructure created.

Sample Terraform acceptance test below.

```
package example

//example.Widget represent a concrete Go type that represents an API resource
func TestAccExampleWidget_basic(t *testing.T) {
  var widgetBefore, widgetAfter example.Widget
  rName := acctest.RandStringFromCharSet(10, acctest.CharSetAlphaNum)

  resource.Test(t, resource.TestCase{
    PreCheck:     func() { testAccPreCheck(t) },
    Providers:    testAccProviders,
    CheckDestroy: testAccCheckExampleResourceDestroy,
    Steps: []resource.TestStep{
    {
      Config: testAccExampleResource(rName),
      Check: resource.ComposeTestCheckFunc(
        testAccCheckExampleResourceExists("example_widget.foo",&widgetBefore),
      ),
    },
    {
      Config: testAccExampleResource_removedPolicy(rName),
      Check: resource.ComposeTestCheckFunc(
        testAccCheckExampleResourceExists("example_widget.foo",&widgetAfter),
      ),
    },
   },
 })
}
```

Testing plugin code in small, isolated units is distinct from Acceptance Tests and does not require network connections. Unit tests are commonly used for testing helper methods that expand or flatten API responses into data structures for storage into the state by Terraform. The procedure for writing unit tests for Terraform follows the same setup and conventions of writing any Go unit tests [16].

Sample Terraform unit test below.

```
func TestFlattenSecurityGroups(t *testing.T) {
    cases := []struct {
        ownerId  *string
        pairs    []*ec2.UserIdGroupPair
        expected []*GroupIdentifier
    }{
        //simple, no user id included
        {
            ownerId: aws.String("user1234"),
            pairs: []*ec2.UserIdGroupPair{
                &ec2.UserIdGroupPair{
                    GroupId: aws.String("sg-12345"),
                },
            },
            expected: []*GroupIdentifier{
                &GroupIdentifier{
                    GroupId: aws.String("sg-12345"),
                },
            },
        },
        //include the owner id, but keep it consistent with the same account.
        // EC2 classic situation
        {
            ownerId: aws.String("user1234"),
            pairs: []*ec2.UserIdGroupPair{
                &ec2.UserIdGroupPair{
                    GroupId: aws.String("sg-12345"),
                    UserId:  aws.String("user1234"),
                },
            },
            expected: []*GroupIdentifier{
                &GroupIdentifier{
                    GroupId: aws.String("sg-12345"),
                },
            },
        }

    for _, c := range cases {
        out := flattenSecurityGroups(c.pairs, c.ownerId)
        if !reflect.DeepEqual(out, c.expected) {
            t.Fatalf("Error matching output and expected: %#v vs %#v", out,
c.expected)
        }
    }
}
```

These tests provide a critical solution to a major problem that arises with the IaC. Due to the large and complicated infrastructure, and dependencies between the services, failure of a single component could force the entire stack to roll back. Ideally, the only testing scenario to test IaC is to manually spin up the entire cluster. However, unit and acceptance testing provide a base-level testing functionality to avoid this method and assure a certain level of confidence in the modules.

2.3.2 Testing frameworks

Goss

Goss is a YAML-based serverspec alternative tool for validating a server's configuration. It eases the process of writing tests by allowing the user to generate tests from the current system state. Once the test suite is written it can be executed, waited on, or served as a health endpoint [17].

Goss allows defining how the desired state of the system should be and validates against the definition. Goss can be categorized as a wrapper for serverspec, another open-source tool testing. *goss.yml* file may provide the testing criteria and validating the stack.

```
file:
  /etc/httpd/conf/httpd.conf:
    exists: true
  /var/www/html:
    filetype: directory
    exists: true

service:
  httpd:
    enabled: true
    running: true
```

Upon running the test suite against the Terraform code, we can obtain the test results with given success or error messages.

```
# goss validate --format documentation
File: /var/www/html: exists: matches expectation: [true]
File: /var/www/html: filetype: matches expectation: ["directory"]
File: /etc/httpd/conf/httpd.conf: exists: matches expectation: [true]
Service: httpd: enabled: matches expectation: [true]
Service: httpd: running: matches expectation: [true]

Total Duration: 0.014s
Count: 10, Failed: 0, Skipped: 0
```

Kitchen-Terraform

In the current industry testing for configuration management tools has improved compared to IaC solutions. Chef kitchen provides a variety of plugins that support testing for CM and IaC.

Kitchen-Terraform provides a set of Chef's Kitchen plugins that enable the use of Kitchen to converge a Terraform configuration and verify the resulting infrastructure systems with controls. Kitchen-Terraform integrates with the Terraform command-line interface to implement a Test Kitchen workflow for Terraform modules. [18].

Kitchen-Terraform helps testing whether the IaC code provides the desired system. Tests are limited to how infrastructure will be provisioned via cross-checking with certain pre-defined values. The test suit will be provided as a YAML file will be executed against the Terraform script to cross-validate the infrastructure creation.

The below code, snippet contains a sample test written for verifying an AWS EC2 instance for Terraform. It validates whether the server is created with an Ubuntu base image.

```
control 'operating_system' do
  describe command('lsb_release -a') do
    its('stdout') { should match (/Ubuntu/) }
  end
end
```

Upon running the test suite against the Terraform code, we can obtain the test results with given success or error messages.

```
Command: `lsb_release -a`
  stdout
    is expected to match /Ubuntu/

Finished in 0.23381 seconds (files took 2.62 seconds to load)
1 example, 0 failures
```

Terratest

Terratest is a Go library that makes it easier to write automated tests for your infrastructure code. It provides a variety of helper functions and patterns for common infrastructure testing tasks [19].

Terratest provides two distinct advantages over the rest of the testing frameworks. It provides support for all major CSP APIs and extends the testing capability to tools tightly coupled with IaC in modern cloud infrastructure provisionings such as Packer, Docker, Kubernetes, and Helm charts.

Terratest has become the best framework to date to test the code for infrastructure-related testing against a defined desired state document.

Sample terratest code to test a simple web application

```
package test

import (
        "fmt"
        "testing"
        "time"

        http_helper "github.com/gruntwork-io/terratest/modules/http-helper"
        "github.com/gruntwork-io/terratest/modules/terraform"
)

func TestTerraformAwsHelloWorldExample(t *testing.T) {
        t.Parallel()

        terraformOptions := &terraform.Options{
                // The path to where our Terraform code is located
                TerraformDir: "../examples/terraform-aws-hello-world-example",
        }

// At the end of the test, run `terraform destroy` to clean up any resources
that were created.
        defer terraform.Destroy(t, terraformOptions)

// Run `terraform init` and `terraform apply`. Fail if there are any errors.
        terraform.InitAndApply(t, terraformOptions)

// Run `terraform output` to get the IP of the instance
        publicIp := terraform.Output(t, terraformOptions, "private_ip")

// Make an HTTP request to the instance and make sure we get back a 200 OK
with the body "Hello, World!"
        url := fmt.Sprintf("http://%s:8080", privateIp)
        http_helper.HttpGetWithRetry(t, url, nil, 200, "Hello, World!", 30,
        5*time.Second)
}
```

Upon running the test suite against the Terraform code, we can obtain the test results with given success or error messages.

```
$ go test -v -run TestTerraformAwsHelloWorldExample -timeout 30m
=== RUN   TestTerraformAwsHelloWorldExample
Running command terraform with args [init]
Initializing provider plugins...
[...]
Terraform has been successfully initialized!
[...]
Running command terraform with args [apply -auto-approve]
aws_instance.example: Creating...
[...]
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
Outputs:
private_ip = 10.232.14.16
[...]
Making an HTTP GET call to URL http://10.232.14.16:8080
dial tcp 10.232.14.16:8080: getsockopt: connection refused.
Sleeping for 5s and will try again.
Making an HTTP GET call to URL http://10.232.14.16:8080
dial tcp 10.232.14.16:8080: getsockopt: connection refused.
Sleeping for 5s and will try again.
Making an HTTP GET call to URL http://10.232.14.16:8080
Success!
[...]
Running command terraform with args [destroy -force -input=false]
[...]
Destroy complete! Resources: 2 destroyed.
--- PASS: TestTerraformAwsHelloWorldExample (149.36s)
```

2.3.3 Post-provision validation

The current industry standard for infrastructure compliance would post provision validation to test violations in new resource builds and interval validations to identify violations due to drifts or evaluating the existing infrastructure for updated policies.

Initially, this was achieved via custom serverless compute options provided by the CSP triggered with new resource creations. For example, AWS Lambda can be used coupled with AWS CloudWatch to detect, scan, remove and alert the user on a policy violation. However, defining policies in such a way created a lot of maintainability issues and thus resulting in one account made a large amount of repetitive work for organizations with multiple CSP accounts.

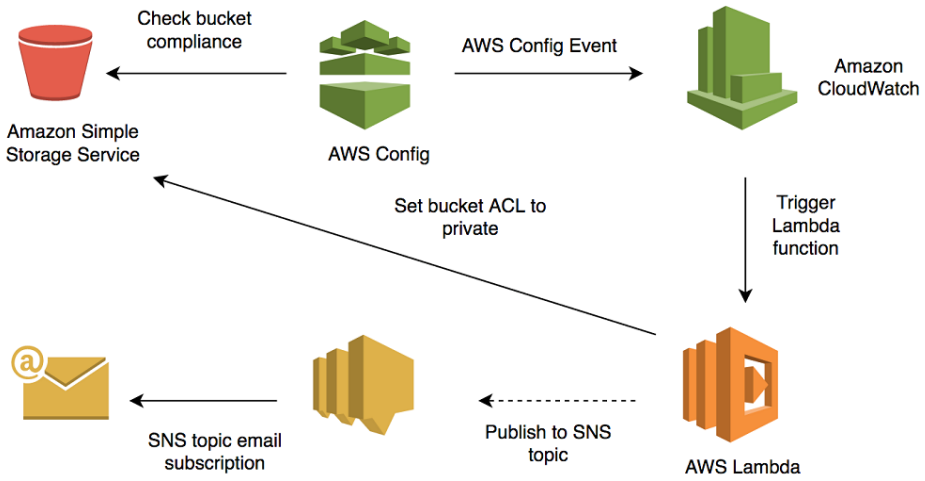


Figure 7. The architecture of AWS Lambda based post validation for S3 ACL violation
Source: AWS Security Blog - How to Use AWS Config to Monitor for and Respond to Amazon S3 Buckets Allowing Public Access

As s solution for this, a more automated and maintainable solution was required. This brought the uprising of Cloud Custodian. Cloud custodian is one of the widely popular and used tools for post validation. The simple YAML DSL allows you to easily define

rules to enable a well-managed cloud infrastructure, that's both secure and cost-optimized. Custodian supports managing AWS, Azure, and GCP public cloud environments [20].

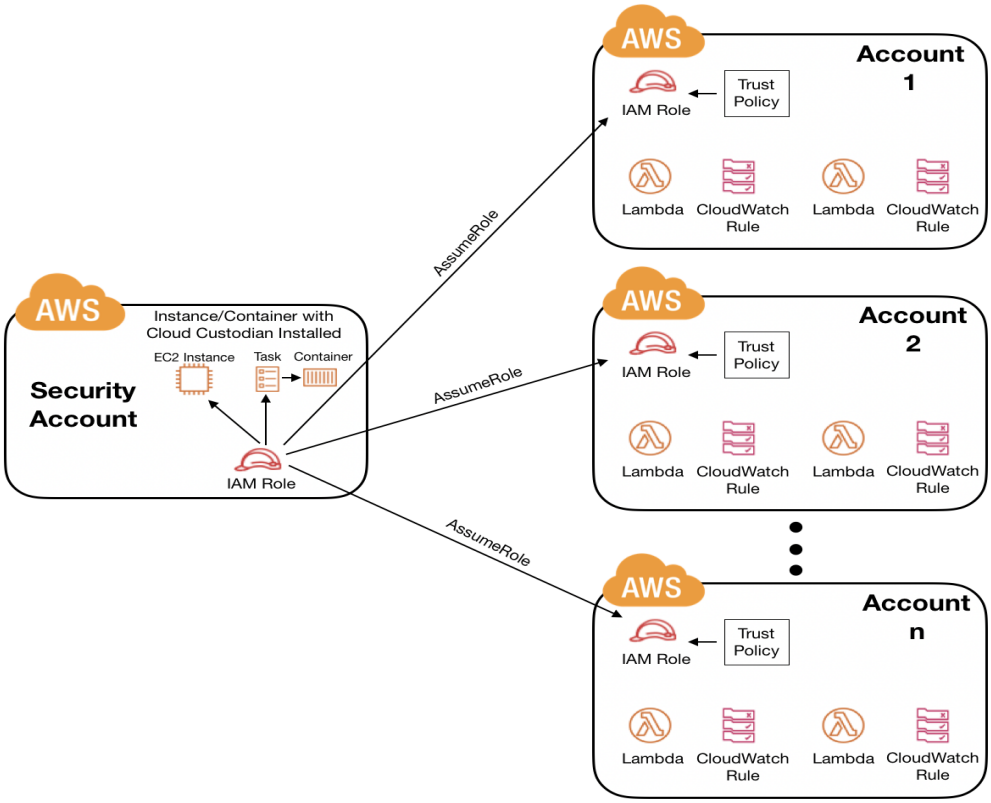


Figure 8. Cloud custodian architecture for multi-account AWS setup Source: AWS Security Blog – Applying IAM policies to multi accounts through Security account

Cloud custodian policies are stored in SCM, such as GitHub. This provides, much easier way to organize, version, and document the organization and compliance policies.

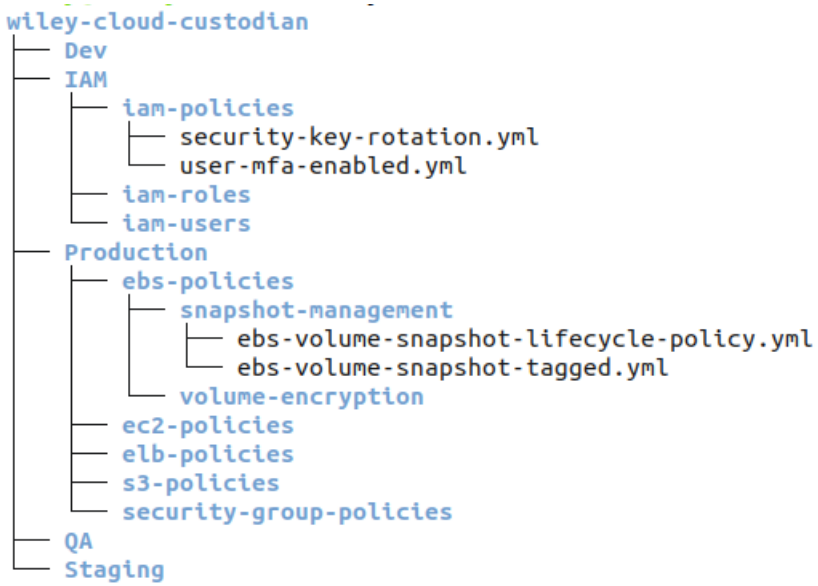


Figure 9. Cloud custodian policy structuring on SCM

Additionally, cloud custodian uses policies written using declarative and universally supported languages such as yml. Below is a sample custodian policy written to identify EC2 instances older than 60 days and AMIs built before 120 days

```
- name: ec2-old-instances
  resource: ec2
  comment: |
    Report running instances older than 60 days
  filters:
    - "State.Name": running
    - type: instance-age
      days: 60

  # Use Case: Report all AMIs that are 120+ days or older

  - name: ancient-images-report
    resource: ami
    comment: |
      Report on all images older than 90 days which should be de-registered.
    filters:
      - type: image-age
        days: 120
```

2.3.4 Continuous integration

As the pipeline plays an important role in continuous integration, continuous delivery, and continuous deployment, there are a lot of tools available for this. The tools can be divided into the below categories.

Commercial	**Open-Source**
CircleCI	Jenkins
Jetbrains TeamCity	TravisCI

Table 3. Commercial and Open-Source CI tools

Apart from this, the CSPs are providing their CI tools

AWS	CodePipeline, CodeDeploy
GCP	CloudBuild
Azure	Azure DevOps

Table 4. CI solutions offered by CSPs

A CSP-specific Continuous Integration pipeline can be built, using their services if a requirement arises.

Ex – AWS

- IaC – CloudFormation
- CI – CodePipeline
- SCM – AWS CodeCommit

As the research is conducted using Terraform, which supports all major CSPs, a CI tool that supports all cloud services should be selected. Therefore, Jenkins is selected due to its support for various resources through Jenkins plugins.

3. METHODOLOGY

3.1 Research Philosophy

The research is looking into the area of frequent and continuous implementation, and modification of cloud-based infrastructure as opposed to major modifications. The literature review explores the prior research conducted on this concept and discusses the finding of them. Further, it reviews the existing commercial and open-source tools related to Infrastructure as Code and standardization of it to design the continuous implementation pipeline.

Additionally, to gain information on how current organizations manage the infrastructure, a questionnaire was presented to the engineers who actively work in the industry with cloud implementation, to add more depth & understanding to the research. The research design was fine-tuned through the insights gathered through the survey. The survey was conducted anonymously as the implementation strategies could be treated as sensitive information by the organization and it could be used for malicious intent.

In order to support future work on the research, technologies were chosen to support a breadth of use cases and remained vendor or CSP neutral as much as possible. Terraform was chosen as the IaC provisioner as this would allow testing the continuous implementation on any major CSP as most organizations are moving towards a multi-cloud approach. Jenkins was selected as the CI pipeline technology due to the variety of technologies supported by it, opposed to most of the other competitive technologies. Further, both technologies are open-source therefore it provides the possibility of further developing the research on the future by contributing directly to both Terraform and Jenkins.

3.2 Research Design

In design, the data was gathered on known IaC defects, that can be caused due to implementations risks and violations of the organizational, CSP, and security policies. Understanding and mitigating IaC defects play a key role in automated infrastructure implementation.

Once the information was gathered on the IaC defects, it was used to design the baseline of the continuous implementation and the pipeline built to conduct the test scenarios.

3.2.1 Infrastructure risks and violations

Understanding these major infrastructure risks on a cloud environment can be used in standardizing the IaC for the Continuous Implementation pipelines, and categorization of impact from IaC defects from severe and minimal.

1. OS & Docker Image vulnerabilities

   Current cloud infrastructure can be categorized into two major sections. Virtual machines are provisioned with OS images (Amazon EC2 AMI) and Containers provisioned with container image files (Docker Images). Most organizations provision their Gold machine images hardened with security measures. However, there are scenarios where open source images are being used such as the Canonical Ubuntu Docker registry. As OS being a critical layer in IaaS stacks, this needs to be validated for outdated images and sources.

```
variable "instance_type" {
  description = "(Required) Type of instance to start”
  type        = string
  default     = "t2.micro"

  validation {
    condition     = substr(var.instance_type, 0, 2) != "i3"
    error_message = "'i3' instance family are not allowed to be provisioned."
  }
}
```

* Above code snippet shows how variables are validated on a Terraform module. Gold modules have validations for critical variables

2. Public or unauthorized core module usage

Usage of publicly available Terraform modules can cause severe issues on both compliance and security to the infrastructure. Terraform module comes with a dedicated variable file that provides default values for all the parameters defined in the Terraform module code. These default values will be used unless a specific value was provided in the control code in the infrastructure creation. As the default values are not governed by the organization, this can leads to a violation of infrastructure creation.

3. Network Exposure

Insecure IaC configurations can expand the attack surface, which enables reconnaissance, enumeration, and sometimes even the delivery of cyberattacks to a cloud environment [21].

- Configuring insecure security groups/firewall policies with unrequired port openings and network ranges
- Publicly accessible cloud storage services and unencrypted storage modules such as AWS S3
- Public SSH or RDP access to the virtual servers
- Databases that are accessible from the internet
- Exposed public application endpoints such as AWS ALB, which does not have a web application firewall (WAF) implemented on top

4. Ghost Resources

Tagging of cloud assets is a critical requirement to ensure compliance and governance. Untagged resources built using IaC result in ghost resources that can cause challenges in detecting, visualizing, and gaining observability within the actual cloud environment. The result is cloud posture drifts that can go undetected for long periods, as well as challenges in remediating risks. Aside

from the security ramifications, untagged resources also make it extremely difficult to detect the impact on operations such as cost, maintenance, and reliability [21].

5. Organizational and International compliance violations

Organizations that infrastructure built on the cloud required to adhere to the organization's policies and mandatory compliance and standards such as GDPR, HIPAA, PCI, and SOC2. If these standard gets compromised, it could result in a disastrous result with data privacy has been one of the key aspects in the modern IT setup.

6. Sensitive Data Exposure

With cloud resources are granular-level resources, it requires constantly communicate with each other. All CSPs have implemented ACLs for this, therefore authorization is required for each service to communicate with each other. This has resulted in credentials and auth keys being hardcoded into the provisioner code. This could be IAM user secrets keys, RDS passwords, SSH keys committed to an SCM, and KMS encryption keys. All these areas need to be scanned and verified.

7. Configurational Drifts

Though all infrastructure is provisioned with best practices, some urgent scenarios occur when developers make changes to the infrastructure outside the IaC provisioner and directly via the console. This will result in losing the immutability of the cloud. Therefore, it is required to either constantly scan the resources against the IaC state files or capture CSP alerts such as AWS CloudWatch events on infrastructure changes and alert developers on a potential configurational drift.

3.2.2 identified requirement baseline

Based on the literature review and the survey conducted I have identified the current state in provisioning cloud resources through IaC, and defects, issues, and risks that have not been covered in the research conducted. Additionally, the study conducted on tools and technologies well-known and widely used a crossed the industry has provided insights on how the current standardization processes and infrastructure updates are implemented.

This can be summarized to identify the baseline on which aspects should be critical in designing the framework.

1. Implement cloud infrastructure and conduct the standardization, such as organization compliance:
    - naming standards
    - mandatory tags
    - restricted resources

    security compliance:

    - IAM resources with over privileges
    - Unnecessary firewall entries

    and IaC based best practices:

    - usage of organizations Terraform gold modules

    and upon passing the quality gates, integrate with the trunk and continuously deploy to environments.

2. Testing frameworks are currently acknowledged and are the recommended testing and standardization framework by HashiCorp, the creators of Terraform [22]. Terratest depends on the creation of infrastructure on some levels to conduct automated testing.

Ex – Terratest was designed on the concept of creation of the test environment, testing, and then destroying the stack.

Regardless of whether it runs on local or virtual environments, this adds an unnecessary cost and compute. The standardization and compliance checks should have the ability to test the infrastructure based on the IaC scripts or state files generated on resource creation planning, thus following a more cost-efficient approach.

3. Supportability to validate both new infrastructure creation and scan the existing infrastructure. However, we are implying that the existing infrastructure created via an IaC as manually created resources does not cover the scope of the research.

Additionally, the supportability to be integrated with a solution that can be leveraged to take remediation action would be beneficial. CSP serverless compute services with an API such as AWS Lambda with AWS API gateway, or more established solutions like Cloud Custodian become candidates for this scenario.

4. Though post validation is a tested and acknowledged method of compliance standardization, this has some major drawbacks on the new infrastructure provisioning. Cloud custodian is effective on existing infrastructure scanning with policies stored in GitHub for version controlling and maintainability, and the ability to take corrective actions.

Ex – Update on the organizational standard to limit all pre-prod environments to internal network can detected be achieved via custodian by leveraging CSP API to modify the firewall allowed CIDR ranges.

However, custodian’s post validation becomes a liability on new infrastructure provisioning. When a new resource is created in violation of the organization's standards and practices, the custodian detects it once the resource is already

provisioned. Depending on the policy on resource violation, the reaction could vary on alerting the user to destroying the resource.

This can be argued as a less effective way for achieving standardization, due to costing on infrastructure such as reserved instances which adds an upfront payment, and engineer's lack of visibility on resource violation till the entire stack creation. Sudden termination of a resource might cause the entire stack to misbehave and failure to terminate a resource with vulnerability may compromise the infrastructure of the entire organization.

5. The maintainability and user supportability. A versioned policy store would be proven beneficial here as this grants the ability to leverage SCM tools organizations already using to easily interact with the framework.

Ex – Cloud custodian uses GitHub as the policy store and git versioning and PRs to keep track of the modifications to it.

Furthermore, considering most IaC tools are written supporting declarative languages, the policy being implemented with the same would decrease the learning curve on onboarding the standardization framework. This provides an additional benefit as most standardization tools such as custodian follows this common principle, thus improve the supportability of the framework by providing the ability to import already written policies.

3.2.3 A Study on current IaC implementation & standardization

To design the architecture, we need to identify how current industry-leading IaC standard validation solutions are implemented. Below architecture shows a solution widely used in the industry.

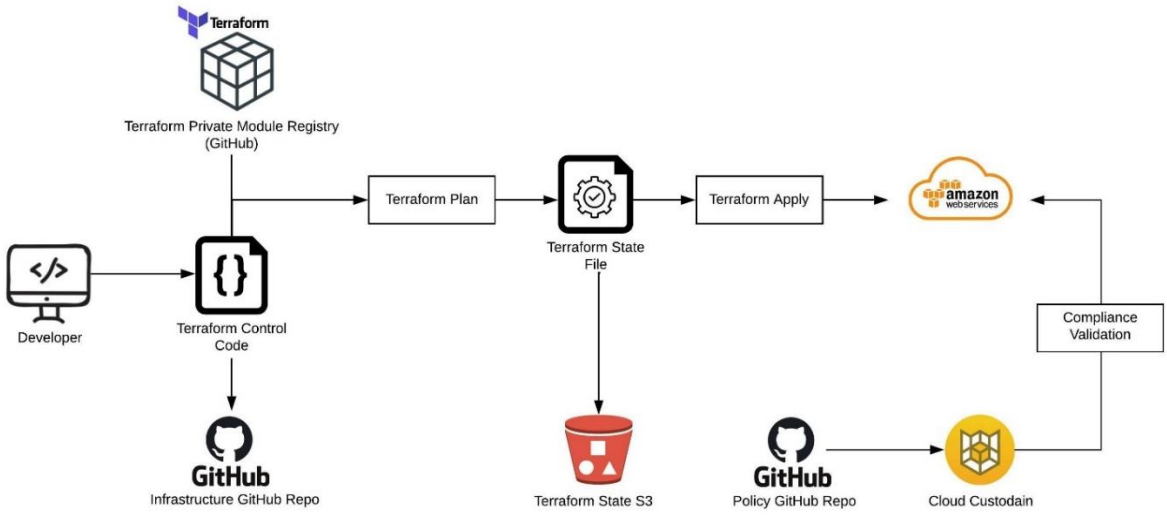


Figure 10. IaC standardization validation model

We can identify the gaps of this solution in the research conducted.

1. Private module registries are used to leverage a common set of modules for all the infrastructure provisions. However, these infrastructure modules do not include in the tests such as Terratest, and new security improvements provided by IaC provisioners are updated.

2. Inability to detect configurational drifts on the infrastructure. Custodian can provide an effective service in standardization violations, however as validation and provisioning are not coupled, infrastructure immutability cannot be validated

3. Issues occur with the post validation. There can be two outcomes from the post validation which could conflict with the organizational standardizations. In a case where violation, for example, security group which allows SSH to 0.0.0.0/0 (public internet), the remedial actions would be,

i. Terminate the resource. However, as cloud resources are tightly coupled with other cloud resources, in this case, security groups with EC2, removal of the resource may impact the functionality and behavior of the system.

ii. Alert the user instead of the termination. This approach grants the user to take the corrective action preserving the infrastructure functionality, however leaving the system vulnerable for a certain period till the corrective action is taken.

4. Violations to severity cannot be measured. Scenarios may occur where certain violations can be ignored. For example, a dev environment may be built on private subnets, completely isolated from the public internet. In such a scenario a publicly open security group does not create a violation impact compared to the same scenario on a publicly accessible production environment. Therefore, identifying the violation depends on the circumstances and requirements.

With gaps of the current implementations gathered and requirements that need to be validated through an IaC standardization framework identified through the literature reviews, we can propose the architecture for the continuous implementation.

3.2.4 Limitations

Configuration management tools can be coupled with IaC tools, to manage server configurations. Ansible is popularly used with Terraform, due to its open-source nature and being one of the top CM tools. CM tools can be userd for,

- User creations
- File system modifications
- System variable and service modifications
- Software Installations & Upgrades
- Setup server configurations in AutoScaling

The standardization framework, however, cannot detect the violations or negligence of organization policies which is handled through configuration management tools. These violations may include,

- Changing security policies (allow password-based SSH)
- Installation of outdated & vulnerable software versions
- Changes to the filesystem permissions

The core reason for not being able to detect these changes due to the flexibility of Ansible's configuration, as the same outcome can be achieved through multiple configurations and some can be hard to detect to an automated tool.

As an example, we can consider the installation of Node.js.

Method #1 – Ansible code

```
- name: Download NodeJs Package
  get_url:
    url: "rpm.nodesource.com/pub_12.x/nodejs-12.13.0-1nodesource.rpm"
    dest: /tmp
    mode: '0644'

- name: Install the nodejs Package
  yum:
    name: /tmp/ nodejs-12.13.0-1nodesource.rpm
    state: present
```

This can be scanned and validated through a standardization framework as it follows a systematic approach.

Method #2 – Shell script

```
- name: Install NodeJS
  shell: "wget –o rpm.nodesource.com/pub_12.x/nodejs-12.13.0-
1nodesource.rpm && rpm –i nodejs-12.13.0-1nodesource.rpm"
```

As this is handled through a shell script, it's near impossible to scan the code to identify a violation. However, an advanced scanning tool can theoretically identify it.

Method #3 – Remote script execution

```
- name: Get a specific version of an object.
  amazon.aws.aws_s3:
    bucket: server_configurations_ldpp
    object: /config/system/nodejs_installer.sh
    version: 48c9ee5131af7a716edc22df9772aa6f
    dest: /usr/local/config/installer.sh
    mode: get

- name: Install NodeJS
  shell: "./usr/local/config/installer.sh"
```

This installation is handled through a downloadable shell script, in which the content is unknown to the pre-scanning tool. Therefore it not possible to identify a violation in the installation.

3.2.5 Architecture

With the identified risks and violations, discussed in section 3.2.1, requirement baseline on 3.2.2, the current IaC implementations on 3.2.3, and limitations on section 3.2.4 we can design the architecture as below.

1. Continuously implement (provision or modify) infrastructure like a CI pipeline

- CI Tool – *Jenkins* (Open-source and most used CI application)
- Source Code Management – As Terraform code will be stored in an SCM, any of the popular code repositories could be used. *GitHub* will be used for the solution
- Trigger – A Pull-Request

2. Check for infrastructure drifts

- Before the new infrastructure provisioned, it will be checked for any configurational drifts (modifications done outside the IaC, typically a manual change through the web console or CLI)
- If a drift is identified, the pipeline should abort and inform the user to fix the drift as the IaC code will try to revert the changes to the desired state on its code. This could break the functionality of the infrastructure if the manual change is a critical one

3. Test the infrastructure based on the IaC scripts or state files

- Ability to test the infrastructure on based on the IaC scripts or state files generated on the resource creation planning stage, instead of the create-test-destroy or post-implementation methods
- Testing using the state file at planning state will prevent the potential issues discussed in the post-validation (in section 2.3.3)

4. Evaluate the infrastructure for code, compliance, and security

- Code-level policies will validate for standard Terraform configurations, usage of organization gold modules, and proper syntaxes
- Compliance policies will validate for organization's and international data and privacy standards.
- Security policies will validate for cloud and organization best practices on infrastructure implementation
- A versioned ruleset store would be proven beneficial here as this grants the ability to leverage SCM tools organizations already using to version and seamlessly interact with the framework

5. Weight and measure compliance violations

- Measure compliance violations on severity, environment, and the impact through weighted matrices; acting as a quality gate (as discussed in section 3.2.3).
- Quality gate failure should result in aborting the implementation process and notify the developer

6. Automated and continuous workflow

- The continuous implementation workflow should work automatically with zero human interactions.
- Continuous feedback to the developer who made the PR, and in case of a standardization failure, the violation identified should be passed. Communications can be done through email and company chat applications such as MS Teams or Slack

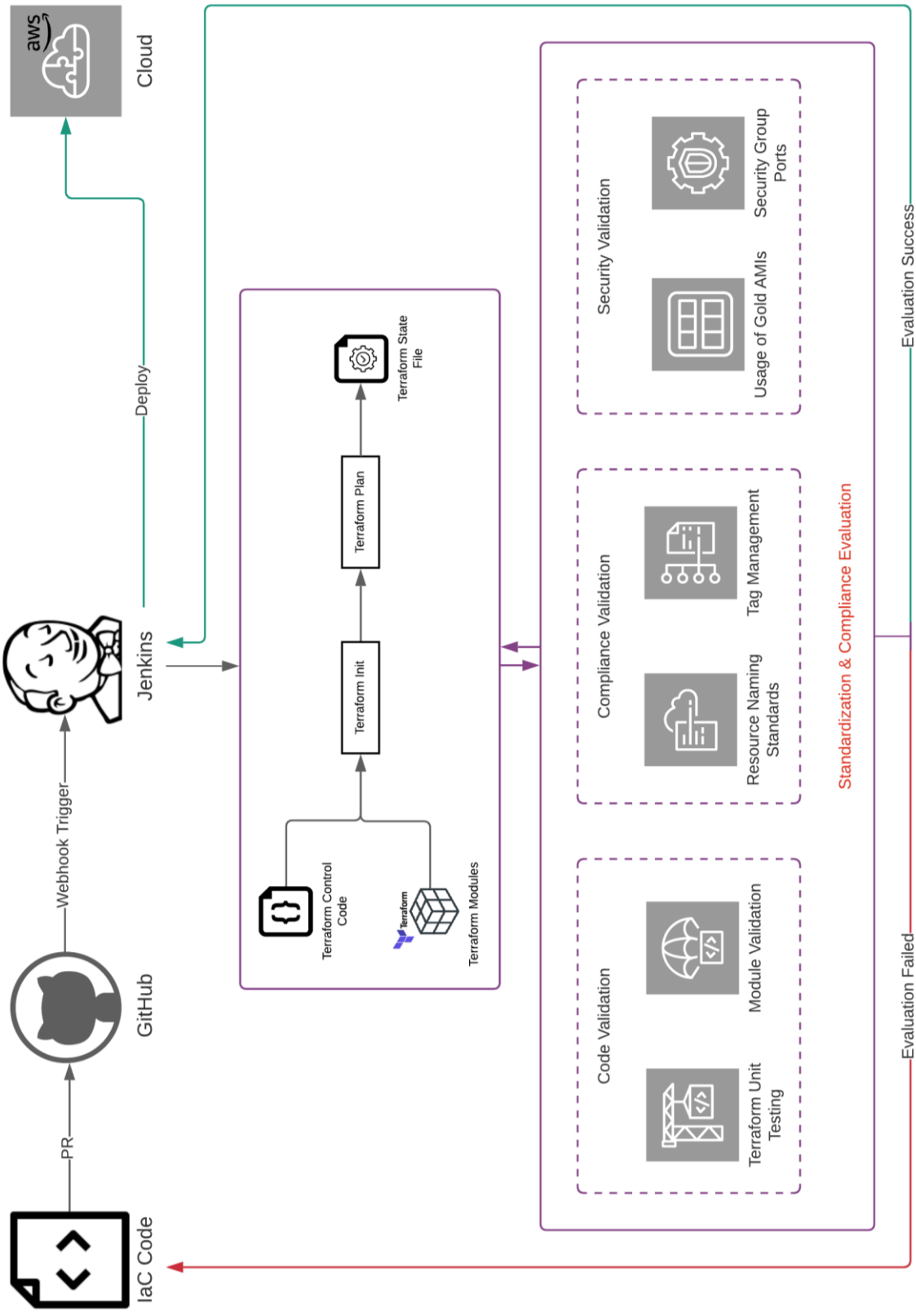


Figure 11. Continuous Implementation Framework

3.3 Test Scenarios

3.3.1 Scenario 01 – Infrastructure drifts

This test scenario will be used to test infrastructure implementation, with an infrastructure drift.

Requirement – Provision an EC2 instance and a security group

Specifications -

Instance Type	m4.large
Image	Organization's hardened Gold AMI – AWS Linux
Security Group Rules	Port 22 for 10.0.0.0/8, Port 80 for 10.0.0.0/8

Added IaC defect - None

Policy –

```
package sinasco.aws.drift
import input as tfplan

// Total score for the validation
quality_gate = 5

// Marks assigned for validations
quality_values = {
    "aws_s3_bucket": {"violation": 10},
    "aws_security_group": {"violation": 10},
    "aws_security_group_rule": {"violation": 10},
    "aws_instance": {"violation": 10}
}

// Cloud resources measured in the validation
resource_types = {"aws_s3_bucket","aws_security_group","aws_security_group_rule","aws_instance"}

// Quality Gate Evaluation
default quality_gate_passed = false
quality_gate_passed {
    score < quality_gate
}

// Compute the score for encryption
score = eval {
    all := [ res |
            some resource_type
            crud := quality_values[resource_type];
            violation := crud["violation"] * evaluate_drift[resource_type];
            res := violation
    ]
    eval := sum(all)
}
```

```
34
35  // List all resources json objects
36  resources[resource_type] = all {
37      some resource_type
38      resource_types[resource_type]
39      all := [name |
40          name:= tfplan.resource_changes[_]
41          name.type == resource_type
42      ]
43  }
44
45  // Error message to display on a violation
46  violation["Drift detected!"] {
47      evaluate_drift[resource_types[_]] > 0
48  }
49
50  // Validate for infrastructure drifts
51  evaluate_drift[resource_type] = num {
52      some resource_type
53      resource_types[resource_type]
54      all := resources[resource_type]
55      modifies := [res |  res:= all[_]; res.change.actions[_] == "update"];
56      num := count(modifies)
57  }
58
```

Expected Outcome:

Figure 12. Test 01: Infrastructure drift

3.3.2 Scenario 02 – Code level violation

This test scenario will be used to test infrastructure implementation, however, the Terraform module for the EC2 instance creation was used outside the organization's gold Terraform module registry

Requirement – Provision an EC2 instance and a security group

Specifications -

Instance Type	m4.large
Image	3rd Party AMI – AWS Linux
Security Group Rules	Port 22 for 10.0.0.0/8, Port 80 for 10.0.0.0/8

Added IaC defect:

- Unauthorized Terraform module

Policy:

```
package sinasco.aws.code.module
import input as tfplan

// Total score for the validation
quality_gate = 5

// Marks assigned for validationss
quality_values = {
    "module_source": {"gold_modules":10}
}

// Cloud resources measured in the validation
resource_types = {"module_source"}

// Quality Gate Evaluationn
default quality_gate_passed = false
quality_gate_passed {
    score < quality_gate
}

// Compute the score for the terraform gold module usage
score = eval {
    all := [ res |
            some resource_type
            crud := quality_values[resource_type];
            gold_modules := crud["gold_modules"] * validate_modules[resource_type];
```

```
27          res := gold_modules
28      ]
29      eval := sum(all)
30  }
31
32  // List all module source json objects
33  modules[resource_type] = all {
34      some resource_type
35      resource_types[resource_type]
36      all := [name |
37          name:= tfplan.configuration[_]
38      ]
39  }
40
41  // Error message to display on a violation
42  violation["Unauthorized Terraform module(s) detected. Please use the Gold modules"] {
43     validate_modules[resource_types[_]] > 0
44  }
45
46  // Validating the modules
47  validate_modules[resource_type] = num {
48      some resource_type
49      resource_types[resource_type]
50      all := modules[resource_type]
51      val = true
52      creates := [res | res:= all[_]; not val = (glob.match("**Udaara/rp-code-modules.git**",
53          [], res.module_calls[_].source))];
54      num := count(creates)
55  }
56
```

Expected Outcome:

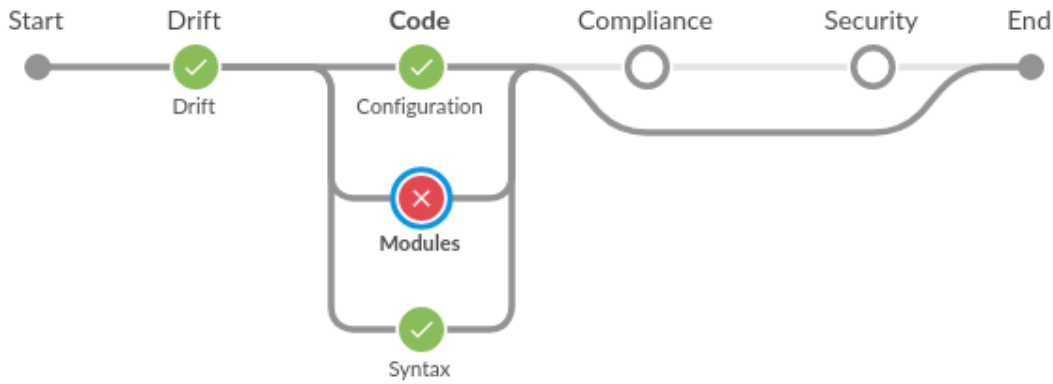


Figure 13. Test 02: Gold modules were not used

3.3.3 Scenario 03 – Organization compliance violation

This test scenario will be used to test infrastructure implementation, however, the EC2 instance to be created are not tagged with the mandatory tags required by the organization's cloud infrastructure best practices. This is a severe violation as this can leads to ghost resources, in addition to the cloud bill with no-ties to any of the projects

Requirement – Provision an EC2 instance and a security group

Specifications -

Instance Type	m4.large
Image	Organization's hardened Gold AMI – AWS Linux
Security Group Rules	Port 22 for 10.0.0.0/8, Port 80 for 10.0.0.0/8

Added IaC defect:

- Mandatory tags are not added to the EC2 instance

Policy -

```
package sinasco.aws.code.module
import input as tfplan

// Total score for the validation
quality_gate = 5

// Marks assigned for validations
quality_values = {
    "module_source": {"gold_modules":10}
}

// Cloud resources measured in the validation
resource_types = {"module_source"}

// Quality Gate Evaluation
default quality_gate_passed = false
quality_gate_passed {
    score < quality_gate
}
```

```
20
21  // Compute the score for the terraform gold module usage
22  score = eval {
23      all := [ res |
24              some resource_type
25              crud := quality_values[resource_type];
26              gold_modules := crud["gold_modules"] * validate_modules[resource_type];
27              res := gold_modules
28      ]
29      eval := sum(all)
30  }
31
32  // List all module source json objects
33  modules[resource_type] = all {
34      some resource_type
35      resource_types[resource_type]
36      all := [name |
37          name:= tfplan.configuration[_]
38      ]
39  }
40
41  // Error message to display on a violation
42  violation["Unauthorized Terraform module(s) detected. Please use the Gold modules"] {
43     validate_modules[resource_types[_]] > 0
44  }
45
46  // Validating the modules
47  validate_modules[resource_type] = num {
48      some resource_type
49      resource_types[resource_type]
50      all := modules[resource_type]
51      val = true
52      creates := [res | res:= all[_]; not val = (glob.match("**Udaara/rp-code-modules.git**",
53          [], res.module_calls[_].source))];
54      num := count(creates)
55  }
56
```

Outcome:

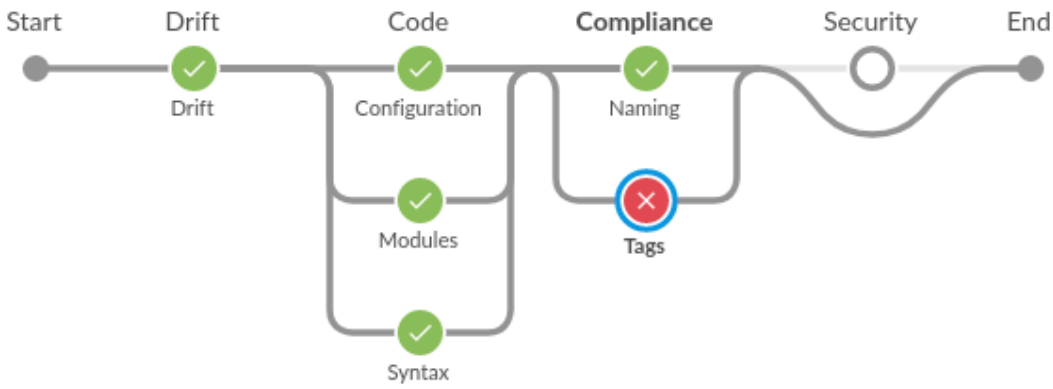


Figure 14. Test 03: Mandatory tags were missing

3.3.4 Scenario 04 – Security violation

This test scenario will be used to test infrastructure implementation, however, the security group to which the EC2 instance belongs has opened the SSH port for public internet.

Requirement – Provision an EC2 instance and a security group

Specifications -

Instance Type	m4.large
Image	Organization's hardened Gold AMI – AWS Linux
Security Group Rules	Port 22 for 0.0.0.0/0, Port 80 for 10.0.0.0/8

Added IaC defect:

- Port 22 opened for 0.0.0.0/0 (internet)

Policy –

```
package sinasco.aws.security.sg
import input as tfplan

// Total score for the validation
quality_gate = 5

// Marks assigned for validations
quality_values = {
    "aws_security_group_rule": {"ingress":10}
}

// Cloud resources measured in the validation
resource_types = {"aws_security_group_rule"}

// Quality Gate Evaluation
default quality_gate_passed = false
quality_gate_passed {
    score < quality_gate
}

// Compute the score for Security Group Ingress Rules
score = eval {
    all := [ res |
```

```
            some resource_type
            crud := quality_values[resource_type];
            ingress_source := crud["ingress"] * validate_sg_ingress[resource_type];
            res := ingress_source
    ]
    eval := sum(all)
}

// List all resources json objects
resources[resource_type] = all {
    some resource_type
    resource_types[resource_type]
    all := [name |
        name:= tfplan.resource_changes[_]
        name.type == resource_type
    ]
}

// Error message to display on a violation
violation["One or more security group ingress rules are exposed to internet.
    Please change the source to 10.0.0.0/8"] {
    validate_sg_ingress[resource_types[_]] > 0
}

// Enforce ingress sources to organization intranet
validate_sg_ingress[resource_type] = num {
    some resource_type
    resource_types[resource_type]
    all := resources[resource_type]
    creates := [res | res:= all[_]; res.change.after.type == "ingress";
        res.change.after.cidr_blocks[_] == "0.0.0.0/0"];
    num := count(creates)
}

```

Outcome:

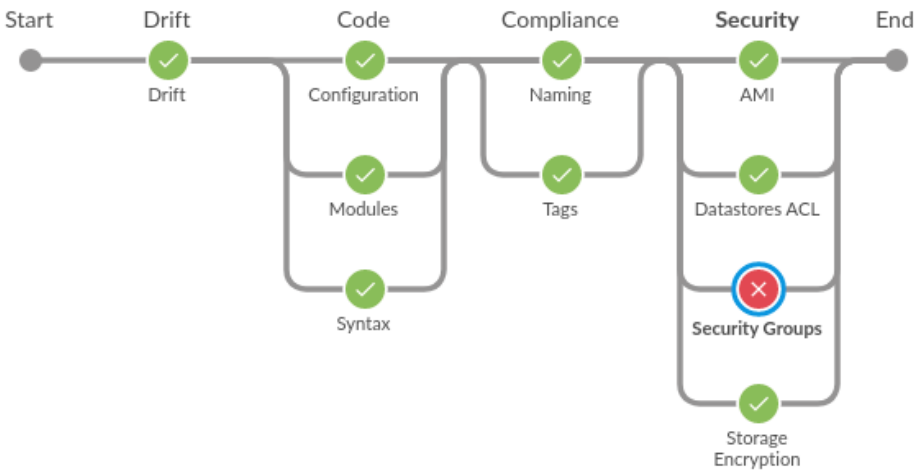


Figure 15. Test 04: Security group with insecure rule

3.3.5 Scenario 05 – Custom quality gates

This test scenario will be used to test infrastructure implementation, however, the EBS volumes that are used to store application data are not encrypted.

The dev environment does not contain any customer data, therefore failure to encrypt the EBS volumes would not be a critical violation. However, not encrypting the Prod EBS volumes would be a severe violation.

Specifications -

Instance Type	m4.large
Image	Organization's hardened Gold AMI – AWS Linux
EBS volumes	Unencrypted
Security Group Rules	Port 22 for 10.0.0.0/8, Port 80 for 10.0.0.0/8

Added IaC defect:

- None

Dev policy –

```
1   package sinasco.aws.security.datastore
2   import input as tfplan
3
4   // Total score for the validation
5   quality_gate = 20
6
7   // Marks assigned for validations
8   quality_values = {
9       "aws_ebs_volume": {"encryption": 10}
10  }
11
```

* Violation grants a score of 10, while the quality gate is 20

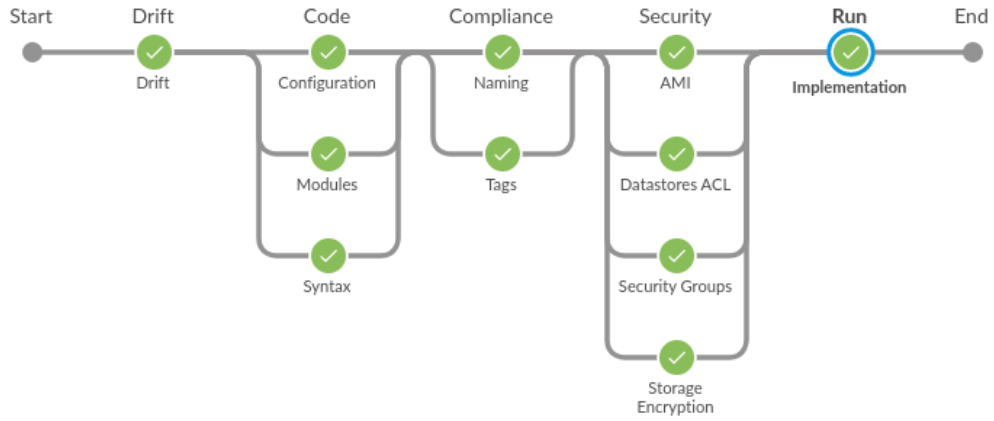
Prod policy –

```
1   package sinasco.aws.security.datastore
2   import input as tfplan
3
4   // Total score for the validation
5   quality_gate = 20
6
7   // Marks assigned for validations
8   quality_values = {
9       "aws_ebs_volume": {"encryption": 30}
10  }
11
```

* Violation grants a score of 30, while the quality gate is 20

Outcome:

Dev –



Prod –

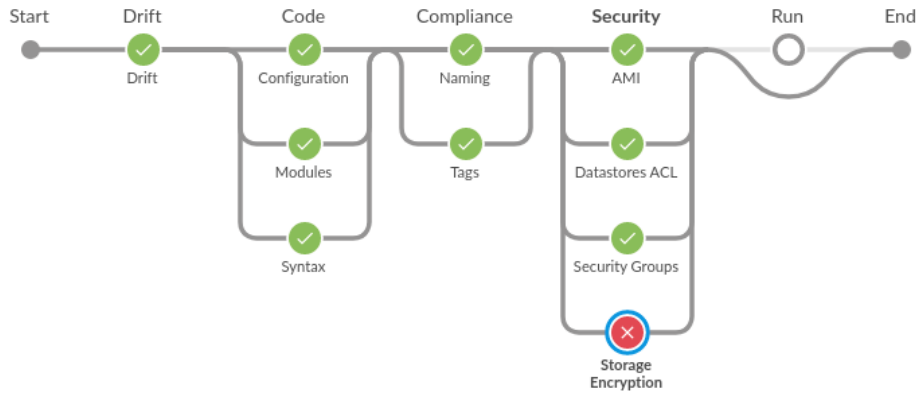


Figure 16. Test 05: A violation with different impact based on the environment

3.3.6 Scenario 06 – Successful implementation

This test scenario will be used to test infrastructure implementation. There are no code-level, compliance, or security violations added to the Terraform code. Therefore the standardization should succeed and infrastructure should be implemented.

Requirement – Provision an EC2 instance and a security group

Specifications -

Instance Type	m4.large
Image	Organization's hardened Gold AMI – AWS Linux
Security Group Rules	Port 22 for 10.0.0.0/8, Port 80 for 10.0.0.0/8

Added IaC defect:

- None

Outcome:

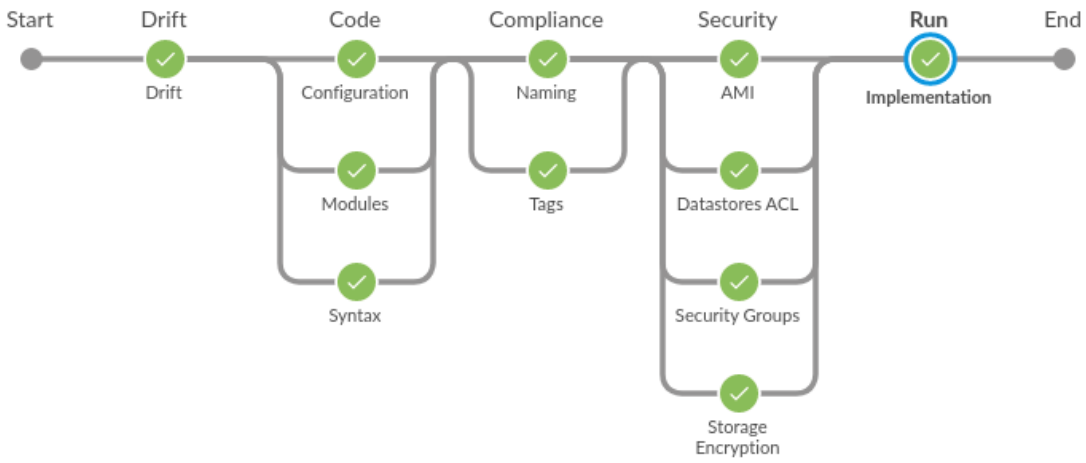


Figure 17. Test 06: Successful implementation

4. ANALYSIS AND FINDINGS

4.1 Survey Results Findings

The survey questionnaire was designed to gather information on IaC across the industry. To ensure the maximum user engagement, it was designed to be completed in under 10 minutes with the most popular answers (such as AWS, Azure & GCP for the cloud service provider) were presented as radio buttons with an additional text-box if users want to add more CSPs.

The questionnaire was shared among the DevOps Engineering, Site Reliability Engineering, and Systems Engineering communities across the globe.

The survey questions can be categorized into the below sections.

- User
    - User's role
    - The company's country of origin
    - User's experience in industry & cloud computing
- Cloud
    - Preferred CSP
    - Provisioning method
- Experience on IaC
    - Learning curve
    - Efficiency
    - Challenges
- IaC practices
    - Testing
    - Infrastructure provisioning
    - Standardization

User

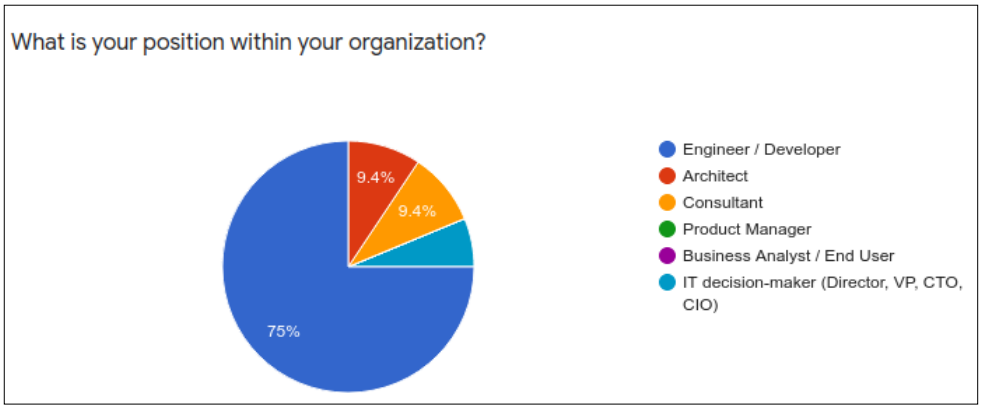


Figure 18. Survey – User designation

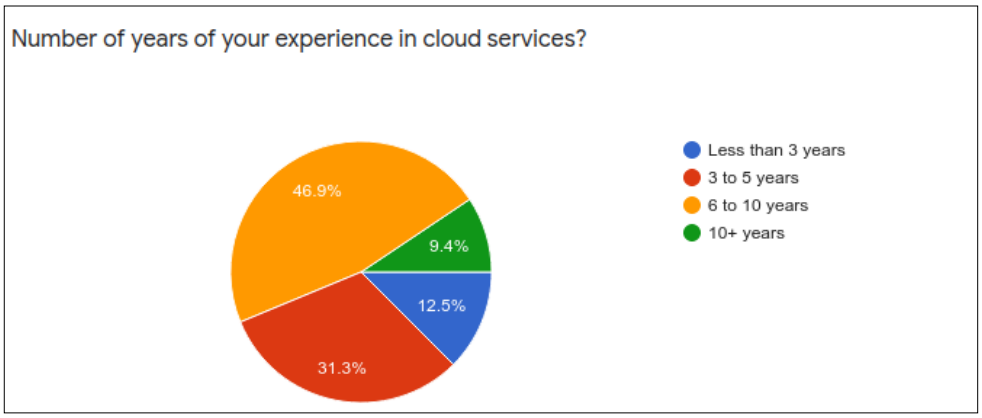


Figure 19. Survey – User experience on cloud computing

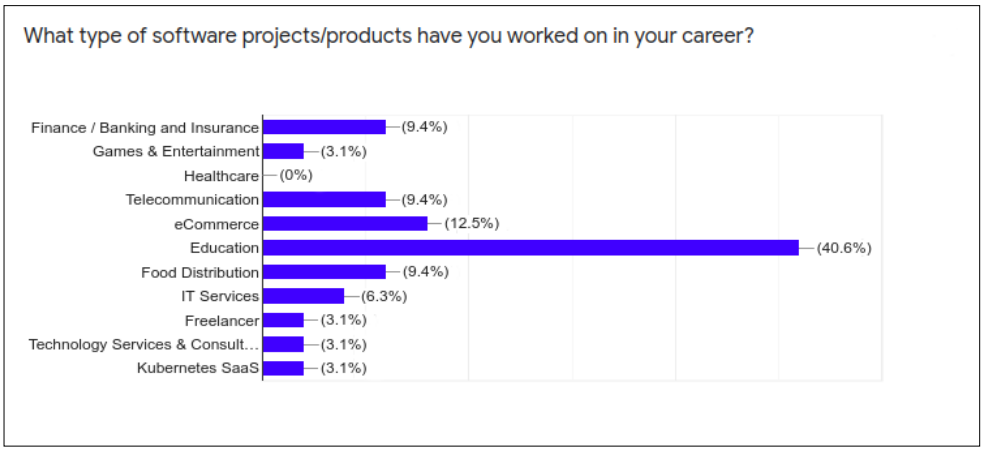


Figure 20. Survey – User experience on industry domains

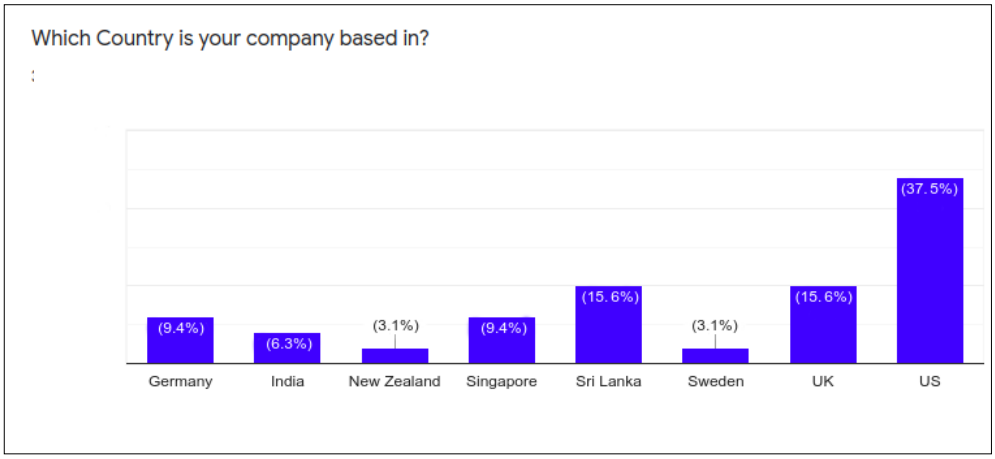


Figure 21. Survey – Company's country of origin

75% of the users were engineers and developers and 20% of the users consisted of Architects and Consultants. The domain knowledge of the userbase covered most of the popular industries, with the majority of 40.6% belonged to the education domain. Their organizations were based on multiple regions, thus providing a diverse result set.

Diversity of the userbase ensured that the IaC and IaC standardization practices of various organizations were captured through the survey. Results of this survey played a vital role in finetuning the research problem and proposed solution & objectives.

Cloud

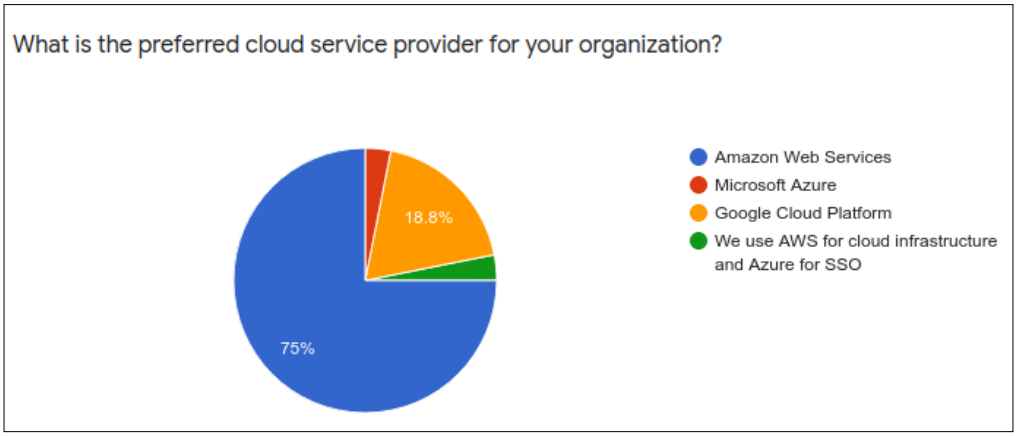


Figure 22. Survey – Cloud service providers

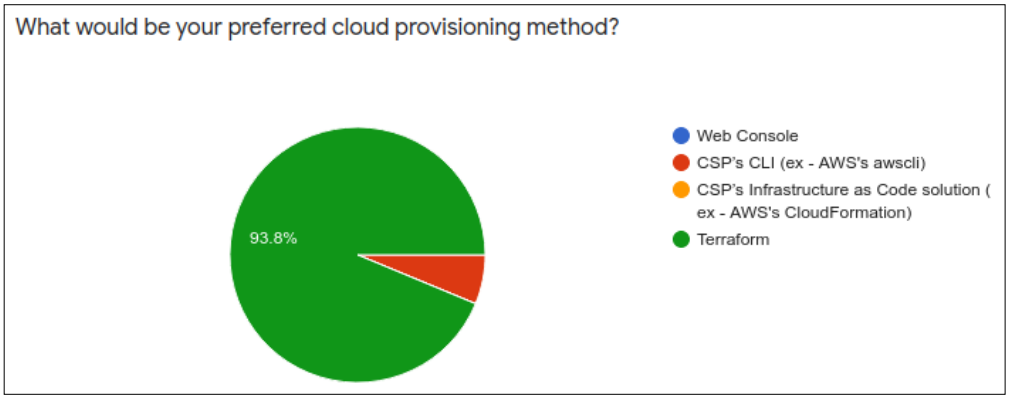


Figure 23. Survey – Infrastructure provisioning in Cloud

The results on cloud technologies were favoring AWS as the CSP with 75% and Terraform as the IaC with over 90 percent. This gives a clear indication of the industrys' preference on Terraform with current trends such as multi-cloud and non-vendor-locking approaches.

Experience on IaC

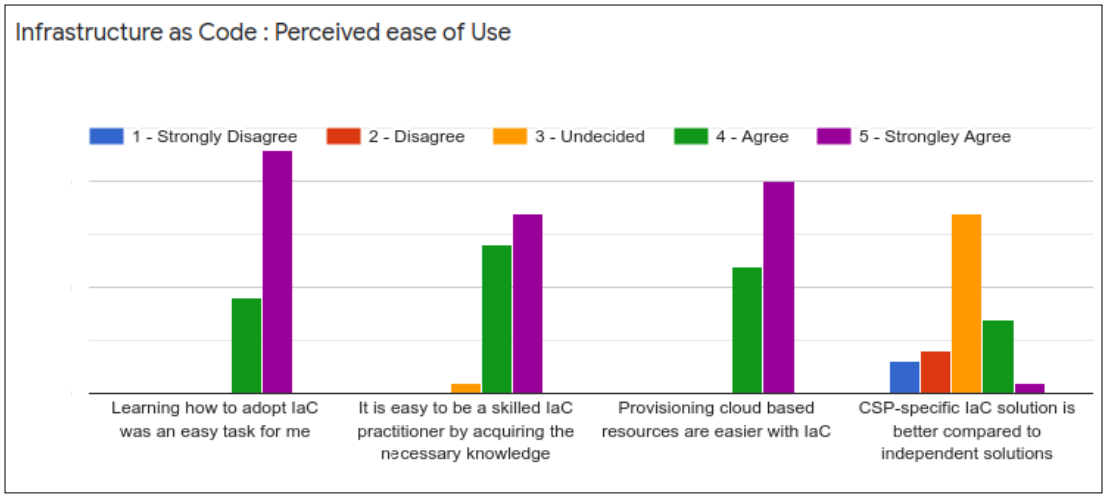


Figure 24. Survey – IaC: Ease of use

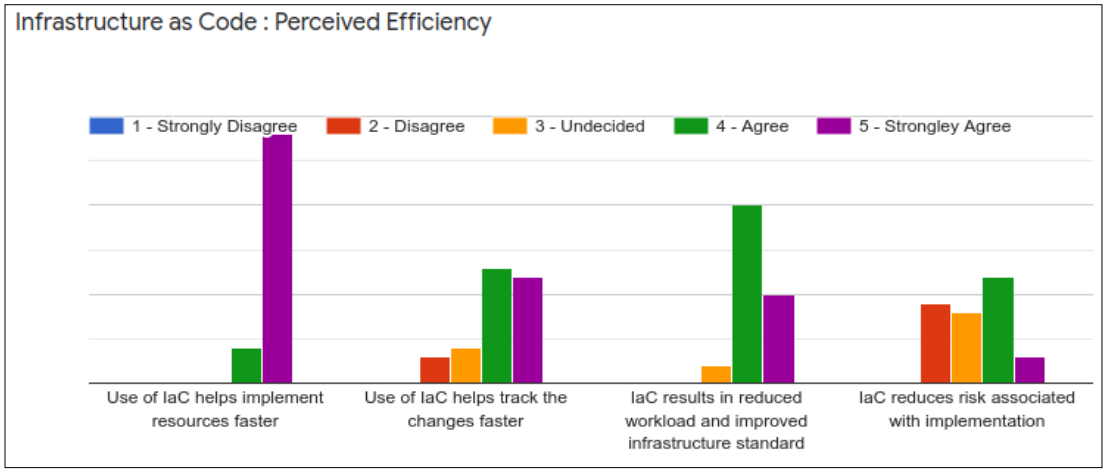


Figure 25. Survey – IaC: Efficiency

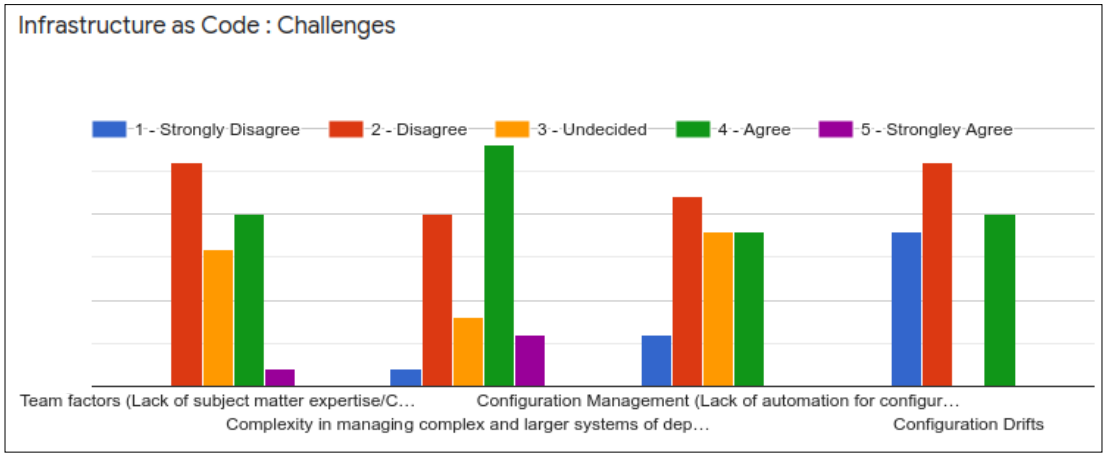


Figure 26. Survey – IaC: Challenges 1

IaC is widely adopted through all organizations and this small learning curve of IaC technologies may have contributed to this. The majority agrees on the fact that IaC helps to implement the resources faster, and 50% stated that IaC can contribute to introducing risk and standardization violations.

Figure 27, shows the concerns users have noticed with IaC. All opinions weigh into the issues with IaC standardization and code reviewing as a misconfiguration can be applied to an entire cloud environment.

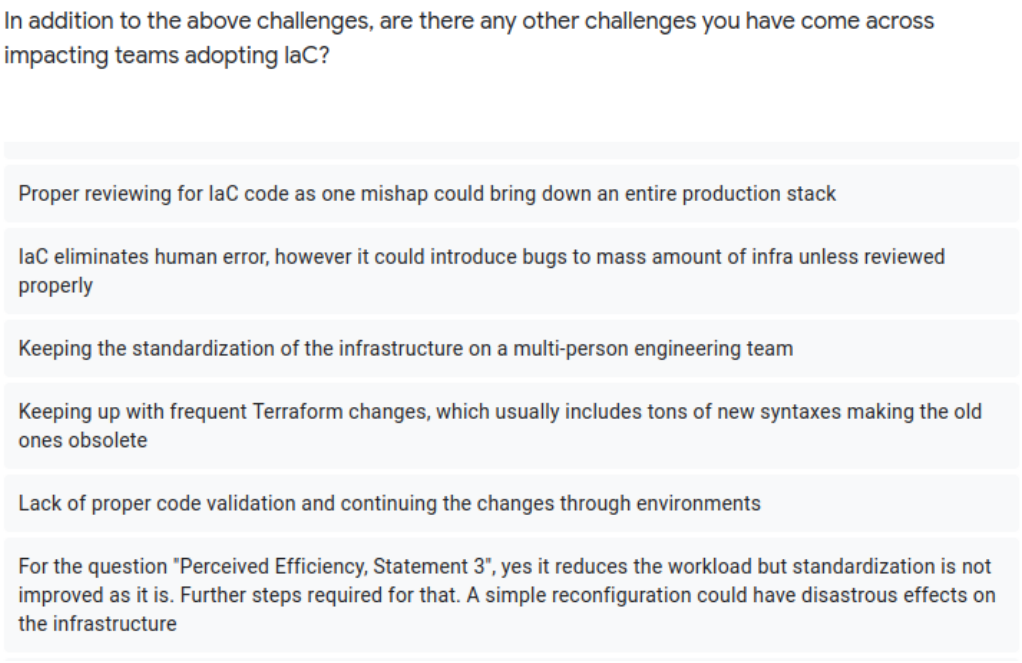


Figure 27. Survey – IaC: Challenges 2

IaC practices

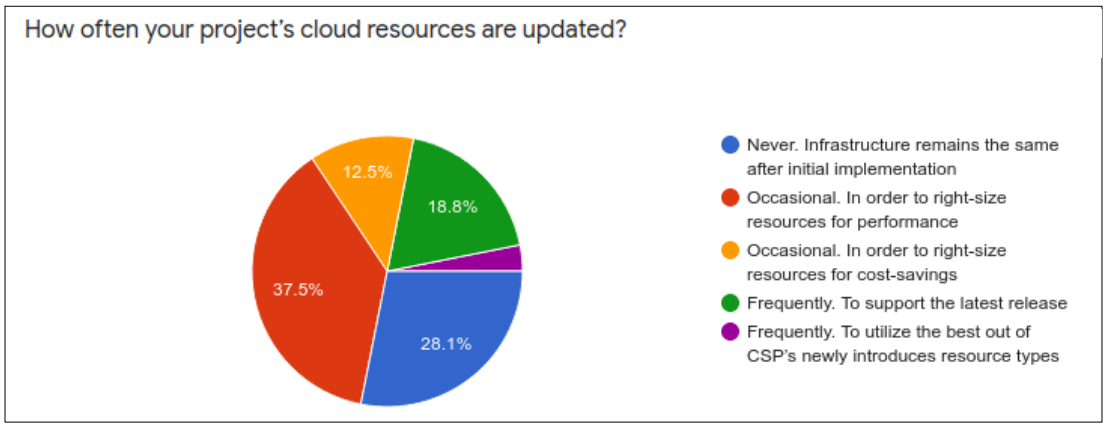


Figure 28. Survey – Cloud resource update frequency

More than 80% of the organizations do not update the infrastructure to support a new code release and nearly 30% never updates the infrastructure once provisioned. This highlights the concern that has been discussed in the *Research Problem*.

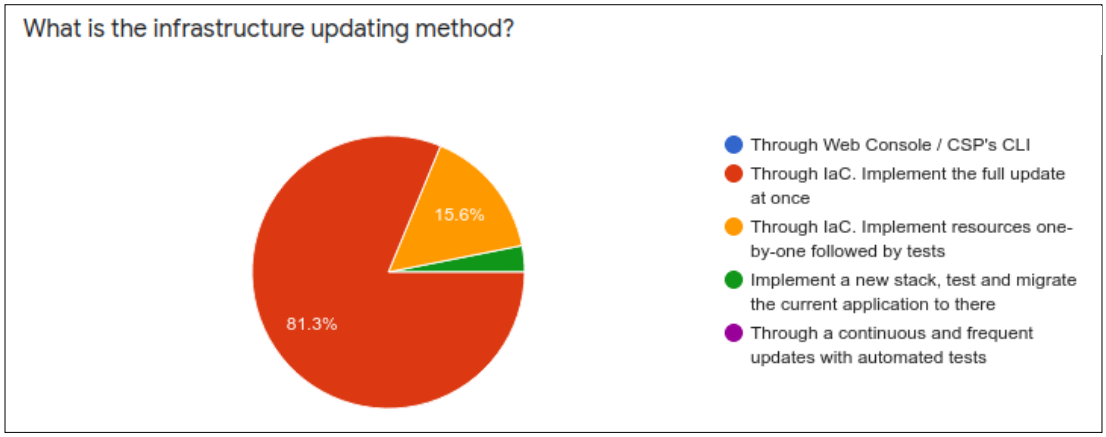


Figure 29. Survey – Cloud resource updating methods

As expected, the infrastructure was updated through IaC with zero web console updates. However, more than 80% update the infrastructure as a bulk update through IaC. This can cause issues as if the code is not properly reviewed and standardized, the violation could impact the full stack of infrastructure and due to cloud resources are tightly coupled to each other to perform the desired task, it would be very time-consuming and complicated to troubleshoot in the issue. This is issue has been discussed thoroughly by Keif Morris[1] and identified as an inadequate cloud engineering practice.

None of the users are currently following continuous implementation workflows or automated IaC standardized tests for the infrastructure provisioned through IaC.

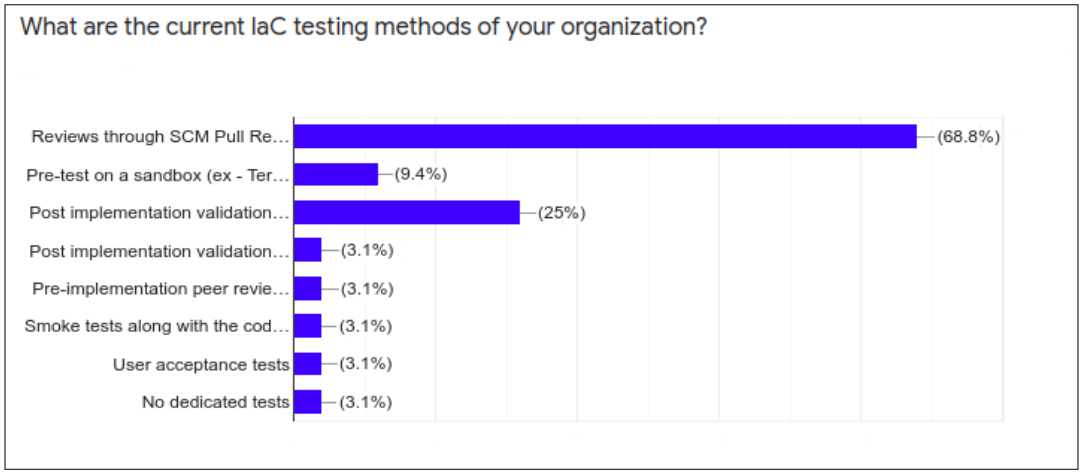


Figure 30. Survey – IaC testing

Figure 30 visualizes the current testing methods for IaC. Nearly 70% of the users conduct peer reviews as the IaC validation method. As explained in the *Literature review* human review of IaC code proved to be less effective on large and complex infrastructure stacks.

The rest of the reviewing methods weighs more towards post-validation methods, popularized with Cloud Custodian and pre-testing on a sandbox method. Security concerns and potential cost-additions of these methods were explained in the Literature review in section 2.3.

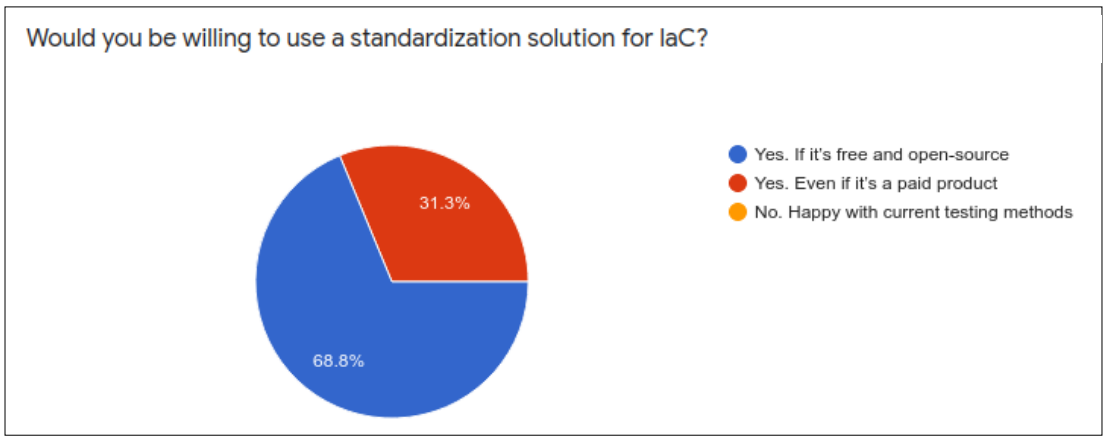


Figure 31. Survey - IaC standardization tool

Finally, we can analyze the current market demand for proper standardization methods for IaC. All answers were divided between open-source solutions and commercialized solutions with zero inputs on satisfaction with the current standardization methods.

4.2 Analysis

During the testing phase of the solution, over 20 IaC defects and violations were tried out against the standardization framework. These IaC defects belong to the seven categories discussed in section 3.2.1 *Infrastructure risks and violations*

Defect	Category	Env	Impact & Severity	Action
Code misconfigurations	Code	Any	Work collaboration is impacted as standard configurations were not used	Reformat the IaC code & initiate the pipeline
Invalid code	Code	Any	IaC code does not function as intended	Alert the developer of the invalid code block
3rd party Terraform module	Code	Dev, QA, UAT	- Low - Unintended and unauthorized values added to code	Abort the implementation and alert the developer
3rd party Terraform module	Code	Prod	- High - Unintended and unauthorized values added to code	Abort the implementation, alert the developer and escalate the issue

Invalid EC2 instance name	Organization standards	Any	Organization naming standards are vital to keeping continuity across cloud eco-system	Abort the implementation and alert the developer
EC2 instance provisioned without mandatory tags	Organization standards	Any	Resources are bound to projects and costs are calculated through tags. Missing tags will cause ghost resources	Abort the implementation, alert the developer and escalate the issue
RDS provisioned in invalid region/VPC	Organization standards	Dev, QA, UAT	- Medium - Inter-resource communications are impacted unless the VPCs have peered	Abort the implementation and alert the developer
Resource created in invalid region/VPC	International standards	Prod	- Critical - Provisioning the resources in an invalid VPC might cause the customer data to be stored in a different geo-region which	Abort the implementation, alert the developer and escalate the issue

			will violate regulations such as GDPR	
Provisioning a Classic ELB	Organization standards	Any	An organization may decide to move away from sunset-phase resources such as classic ELBs	Abort the implementation and alert the developer
S3 bucket provisioned without versioning	Organization standards	Dev, QA, UAT	Versioning for pre-prod may not be a significant violation as the data is not important	Alert the developer but implement the resource
S3 bucket provisioned without versioning	Organization standards	Prod	Production data version may require for a project. A bucket without versioning could violate the requirement	Abort the implementation, alert the developer and escalate the issue
Route53 entry is created which points to ALB DNS without an Alias	Organization standards	Any	Alias records can be implemented free of charge compared to CNames.	Alert the developer but implement the resource

S3 bucket created without a bucket policy	Organization standards	Any	Bucket policy is not mandatory for S3 functionality unless there's a specific purpose. However, as a standard, it's encouraged to use bucket policies	Alert the developer but implement the resource
Create an EC2 instance with a public AMI	Security	Any	Public AMI could have os level vulnerabilities that were not patched yet and it won't have the organization security tools installed	Abort the implementation and alert the developer
Provisioning a security group with SSH access for 0.0.0.0/0	Security	Any	Opening an SSH port to the public internet could make the infrastructure vulnerable to intruder attack	Abort the implementation and alert the developer
Provisioning an EC2 instance	Security	Dev, QA, UAT	As pre-prod EBS volumes	Alert the developer but

with unencrypted EBS volumes			won't be storing sensitive or customer data, this defect won't be a significant violation	implement the resource
Provisioning an EC2 instance with unencrypted EBS volumes	Security	Prod	As prod resource may contain customer data, this violation on Prod will be critical	Abort the implementation and alert the developer
An internet-facing ALB is provisioned without a WAF	Security	Dev, QA, UAT	A downtime that may be caused by a DDoS like an outage can be tolerated by pre-prod. It could be a cost-saving practice as well	Alert the developer but implement the resource
An internet-facing ALB is provisioned without a WAF	Security	Prod	Any sort of downtime on prod won't be tolerated	Abort the implementation and alert the developer
Creating an EC2 instance while there's an	Drift	Any	Manual updates of a cloud resource may cause an	Abort the implementation and alert the developer

infrastructure drift			infrastructure drift. These manual changes will get updated through IaC and might break the functionality expected through the manual change	

Table 5. IaC defects analysis

As the tests were designed to cover most common cloud infrastructure related violations on areas,

- Terraform code, syntax, configuration, and module violations
- Organizational standard violations
- Internation data and privacy violations
- Security violations

All test scenarios provided the expected outcome of,

- Alert the developer
- Alert the develop+ Implement the infrastructure
- Alert the developer + Abort the implementation process
- Alert the developer + Abort the implementation process + Escalate the violation

which were based on the severity of the violation and the environment: the violation occurred

Scenarios designed for success ran out as expected with infrastructure changes applied to the respective environments.

Through implementing CloudCustodian, a popular cloud infrastructure standardization tool, and using it as the compliance validator had the same defects identified however corrective actions had several issues.

For example, test scenario 01's defect was flagged by the CloudCustodian but as it follows the post validation, terminating the defected instance was not an option as other cloud resources were dependent on it. To negate this, email alerts can be introduced stating manual correction required but this leaves them vulnerable infrastructure up and running for a considerable amount of time on a security scale. The other widely-adopted testing tool, Terraform unit tests were not able to determine organization compliance violations or defects related to configuration management.

A scenario was tested on applying the infrastructure changes to 3 pre-prod environments and testing the entire environments for the new changes. The traditional Terraform deployment through the local machine and using the automated test suite took nearly 300% of the time compared to the continuous implementation pipeline designed.

Terraform 14's infamous issue of installing configuration management packages every time a terraform init command was executed, further contributed to the slowness as deployment time had another factor to depend on; the quality of internet connectivity. As the Jenkins build server was implemented on AWS and the same network as the resources that required to be modified, eliminated the time delay caused due to re-downloading the packages.

5. CONCLUSION

The research was focused on introducing continuous and automated cloud-based infrastructure implementation through Infrastructure as Code, backed by an IaC standardization framework to review and validate the IaC code-built infrastructure. This process has been named *Continuous Implementation*.

The traditional infrastructure implementation focuses on minimizing the changes, as any change could have the potential to break the system, changes were thoroughly examined and minimized as much as possible. Cloud infrastructure obstructs this method as it's designed to offer dynamic infrastructure, which can be rapidly provisioned, modified, and terminated. Furthermore, CSPs are offering new and improved offerings at a faster rate compared to the past due to the completion of becoming the dominant cloud provider. However traditional provisioning methods with a large number of manual tasks are not suitable for rapidly provisioning the cloud infrastructure. Infrastructure as a Code refers to the practice of provisioning and management of infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. Through IaC infrastructure can be rapidly implemented.

Modern applications; which especially on Cloud can be identified as dynamic solutions. New features and functionality are offered to the end-users at a rapid rate due to the new agile software development methodologies. However, little attention was given, and limited research was carried under the rapid change of infrastructure to support the new code releases on application requirements. Cloud infrastructure could be frequently updated aligning with the code release to further improve the new features, functionalities, and performance or provide the same functionality at a reduced cost.

Continuous implementation proposes a method to design a continuous workflow using the CI/CD principles to IaC and cloud resource provisioning. As this requires to be automated, comprehensive testing methods are required to validate and evaluate the IaC code and make sure the international, organizational, and security standards and

regulations are met. This was achieved through an IaC standardization framework with custom and version policies, which IaC code will be cross-checked with to identify any violations and take remedial actions.

The literature review explores the research studies conducted on IaC. Researches that carried under IaC states the importance of small and frequent changes to the cloud infrastructure as opposed to huge changes applied. IaC testing was more explored and there were several implementations on standardization areas such as multi-cloud supportability (TOSCA compliance), unit testing, and acceptance testing. However, policy or compliance-based standardization method was not explored there.
Then we look into commercial tools and technologies, which were gathered through the survey conducted on IaC, on their capabilities and drawbacks. Though PR peer-reviews and post-validation seem to be the popularly used methods, integrating these into a continuous pipeline could not be done.

In the methodology, I have built a baseline for the continuous implementation architecture on the usage of CI tools to run the continuous. Testing and standardization framework was built to cover the most common defects and risks associated and identified with IaC cloud resource provisioning. IaC validation methods are designed to cover all these areas to minimize the defects which can cause disastrous outcomes. The entire design was based on the open-source tools and the test scenarios were implemented through resources at AWS free-tier, therefore it would help the further development and engagement of the IaC community.

As explained in the analysis and findings, IaC defects were successfully identified by the framework and once they were fixed changes were seamlessly integrated and into the existing infrastructure. The continuous implementation approach plays a significant role here, as opposed to manually executing the IaC scripts, upon verifying through PR reviews, environment by environment could be a tedious process. Further, this has the potential to add human error, as any manual recurring task. Infrastructure

implementation, modification, and configuration times were cut down significantly as well.

Continuous implementation with standardized IaC could open the doors to utilize the full benefits of cloud computing and finally unify the code and infrastructure changes. Further, with fully automated review and standardization process ensures the infrastructure is compliant even in most complex environments provisioning, which is nigh impossible to validate through human PR and code reviews. The custom and versioned ruleset for validations play an important role, as organizations would be able to shape the standardization framework to the best-suited way for their infrastructure. Finally, the custom quality gates, which grades the impact of a violation on environment factor, and remediate actions for violations make the framework more user-friendly and adaptable for any scenario.

Future research may need to focus on continuous implementation on multi-cloud platforms as the current trends in the industry indicates that future infrastructure may consist of resources from two or more CSPs. To face such a scenario, usage of CSP-independent stack (IaC and CI) should be encouraged as combining CSP-specific continuous implementation workflows could be a tedious process. As the same principles can be applied to containerization and container orchestration technologies such as Kubernetes, this would be another potential area to further conduct more researches on.

The Continuous Implementation workflow supported by the standardization framework is an Open-Source project licensed under MIT License.

6. APPENDIX

6.1 Survey Questions

1. What is your position within your organization?

   _____

2. Which country is your company based in?

   _____

3. What type of software projects/products have you worked on in your career? (you can choose more than one)
   a. Finance/Banking and Insurance
   b. Games & Entertainment
   c. Healthcare
   d. Telecommunication
   e. E-commerce
   f. Education
   g. Other
      i. _____

4. The number of years of your experience in the IT industry?
   a. Less than 3 years
   b. 3 to 5 years
   c. 6 to 10 years
   d. 10+ years

5. Do you have experience with Cloud Services?
   a. Yes
   b. No

6. What is the preferred cloud service provider for your organization?
    a. Amazon Web Services
    b. Microsoft Azure
    c. Google Cloud Platform
    d. Other

7. The number of years of your experience in cloud services?
    a. Less than 3 years
    b. 3 to 5 years
    c. 6 to 10 years
    d. 10+ years

8. What would be your preferred cloud provisioning method?
    a. Web Console
    b. CSP's CLI
    c. CSP's Infrastructure as Code solution
    d. Terraform

9. If you are not using Infrastructure as Code for cloud-based resource provisioning, please state the reason
    _____

10. Perceived Ease of Use of IaC

	1 - Strongly Disagree	2 - Disagree	3 - Undecided	4 - Agree	5 - Strongly Agree
Learning how to adopt IaC was an easy task for me					

It is easy to be a skilled IaC practitioner by acquiring the necessary knowledge					
Provisioning cloud-based resources are easier with IaC					
CSP-specific IaC solution is better compared to independent solutions					

11. Perceived Usefulness of IaC

	1 - Strongly Disagree	2 - Disagree	3 - Undecided	4 - Agree	5 - Strongly Agree
The use of IaC helps implement resources faster					
The use of IaC helps track the changes faster					
IaC results in reduced workload and improved infrastructure standard					
IaC reduces the risk associated with the implementation					

12. IaC Challenges

	1 - Strongly Disagree	2 - Disagree	3 - Undecided	4 - Agree	5 - Strongly Agree
Team factors (Lack of subject matter expertise/Communication and Collaboration/ Discipline/ Resistance to change/ Pressure management/Focus etc.)					
Complexity in managing complex and larger systems of dependencies and compliance violations					
Configuration Management (Lack of automation for configuration management)					
Configuration Drifts					

In addition to the above challenges, are there any other challenges you have come across impacting teams adopting IaC?

_____

13. What are the current IaC testing methods of your organization?
    a. Reviews through SCM Pull Requests (ex – GitHub PR reviews)
    b. Pre-test on a sandbox (ex - Terragrunt)
    c. Post-implementation validation (ex – Cloud Custodian)
    d. Other
        i. _____

14. Would you be willing to use a standardization solution for IaC?

a. Yes. If it's free and open-source
b. Yes. Even if it's a paid product
c. No. Happy with current testing methods
d. Other
    i. _____

15. How often your project's cloud resources are updated?
    a. Never. Infrastructure remains the same after initial implementation
    b. Occasional. To right-size resources for performance
    c. Occasional. To right-size resources for cost-savings
    d. Frequently. To support the latest release
    e. Frequently. To utilize the best out of CSP's newly introduces resource types

16. What is the infrastructure updating method?
    a. Through IaC. Implement the full update at once
    b. Through IaC. Implement resources one-by-one followed by tests
    c. Implement a new stack, test and migrate the current application to there
    d. Through continuous and frequent updates with automated tests

17. How does your company enforce security patches or version upgrades for a vulnerability?
    a. Through monthly patch cycle
    b. Email updates from CISO/Security teams to the SRE/DevOps team
    c. The server scans through agents and flagging the affected resources
    d. Other
        i. _____

7. REFERENCES

[1] Kief Morris, Infrastructure as Code, 2nd edition. O'Reilly Media, Inc., 2020.

[2] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery," 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Bangalore, India, 2015, pp. 85-89, DOI: 10.1109/CCEM.2015.29.

[3] Unit Testing – HashiCorp. [Online] Available: www.terraform.io/docs/extend/testing/unit-testing.html [Accessed Nov 17, 2019]

[4] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams, “Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research”, Journal of Information and Software Technology. Jul 2018.

[5] Akond Rahman and Laurie Williams, “Source code properties of defective infrastructure as code scripts”, Information and Software Technology Volume 112, August 2019, Pages 148-163

[6] T. Sharma, M. Fragkoulis, D. Spinellis, Does your configuration code smell?, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR ’16, ACM, New York, NY, USA, 2016, pp.

[7] Y. Jiang and B. Adams. Co-evolution of Infrastructure and Source Code: An Empirical Study. In Proceedings of the 12th Working Conference on Mining Software Repositories, MSR ’15, pages 45–55, Piscataway, NJ, USA, 2015. IEEE Press.

[8] J. Schwarz, A. Steffens, and H. Lichter, "Code Smells in Infrastructure as Code," 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), Coimbra, 2018, pp. 220-228, DOI: 10.1109/QUATIC.2018.00040.

[9] Terraform Docs [Online] Available: https://www.terraform.io/docs/commands/validate.html [Accessed March 21, 2020]

[10] O. Hanappi, W. Hummer, S. Dustdar, Asserting reliable convergence for configuration management scripts, SIGPLAN Not. 51 (10) (2016) 328–343.

[11] K. Ikeshita, F. Ishikawa, S. Honiden, Test suite reduction in idempotence testing of infrastructure as code, in S. Gabmeyer, E. B. Johnsen (Eds.), Tests and Proofs, Springer International Publishing, Cham, 2017, pp. 98–115.

[12] Sandobalín, Julio & Insfran, Emilio & Abrahão, Silvia. (2017). An Infrastructure Modelling Tool for Cloud Provisioning. 10.1109/SCC.2017.52.

[13] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as a standardized metamodel," in Future Generation Computer Systems, 2015, vol. 56, pp. 317–332

[14] J. Scheuner, P. Leitner, J. Cito, and H. Gall, "Cloud workbench - Infrastructure-as-code based cloud benchmarking," in Cloud Computing Technology and Science, CloudCom, 2014, pp. 246–253.

[15] Acceptance Tests [Online] Available: www.terraform.io/docs/extend/testing/acceptance-tests/index.html [Accessed March 27, 2020]

[16] Testing Patterns [Online] Available: https://www.terraform.io/docs/extend/best-practices/testing.html [Accessed March 27, 2020]

[17] aelsabbahy/goss [Online] Available: https://github.com/aelsabbahy/goss [Accessed March 30, 2020]

[18] newcontext-oss/kitchen-terraform [Online] Available: https://github.com/newcontext-oss/kitchen-terraform [Accessed March 30, 2020]

[19] Terratest [Online] Available: https://terratest.gruntwork.io/ [Accessed March 30, 2020]

[20] Cloud Custodian Documentation [Online] Available: wwe.cloudcustodian.io/docs/index.html [Accessed March 30, 2020]

[21] Top 5 Security Risks for Infrastructure-as-Code [Online] https://thenewstack.io/top-5-security-risks-for-infrastructure-as-code/ [Accessed May 30, 2020]

[22] End To End Testing On Terraform With Terratest [Online] Available: https://www.hashicorp.com/resources/end-to-end-testing-on-terraform-with-terratest/ [Accessed May 25, 2020]