

Master's Programme in Computer, Communication and Information Sciences

Mitigating configuration drift in infrastructure-as-code systems

Otso Pohjola

© 2025

This work is licensed under a [Creative Commons](https://creativecommons.org/licenses/by-nc-sa/4.0/) “Attribution-NonCommercial-ShareAlike 4.0 International” license.



Author Otso Pohjola

Title Mitigating configuration drift in infrastructure-as-code systems

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud Computing

Supervisor Lachlan Gunn

Advisor Sami Kairajärvi

Collaborative partner Accenture

Date 27 April 2025

Number of pages 48

Language English

Abstract

Infrastructure-as-code (IaC) is widely utilized in cloud systems, which evolve and expand quickly and are prone to configuration drift. The mismatch between the actual environment state and the environment IaC definition leads to limited visibility in the environment, increased complexity and uncertainty. Some solutions that attempt to fix configuration drift exist already, but they are either platform dependent or utilize third party tools to achieve this.

In order to resolve configuration drift, we developed an application called Driftless, which utilizes and automates IaC framework's own change management capabilities to assemble a complete view of changes in the infrastructure managed by IaC and to import existing resources that are not yet managed by IaC. It also actively searches for new unmanaged resources, as IaC frameworks themselves lack visibility to them.

Driftless produces field-level change data for the resources within our scope, based on which the user can align their infrastructure and their IaC configurations. It also successfully generated import commands for unmanaged resources, which created functional IaC configurations and state entries for the new resources. These two outputs were combined to synchronize the actual infrastructure state with the desired state in both IaC managed and unmanaged parts of the infrastructure.

Keywords Cloud computing, Terraform, configuration drift, change management, infrastructure as code

Tekijä Otso Pohjola

Työn nimi Configuration drift -ilmiön lieventäminen infrastruktuuri koodina -järjestelmissä

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Security and Cloud Computing

Työn valvoja Lachlan Gunn

Työn ohjaaja Sami Kairajärvi

Yhteistyötaho Accenture

Päivämäärä 27.4.2025

Sivumäärä 48

Kieli englanti

Tiivistelmä

Infrastruktuuri koodina (IaC) on laajasti käytetty viitekehys pilviympäristöissä, jotka kehittyvät ja kasvavat nopeasti ja altistuen “configuration drift” -ilmiölle. Eroavuudet ympäristön oikean tilan ja sen IaC määritelmien välillä johtaa rajoitettuun ympäristön näkyvyyteen, kasvaneeseen monimutkaisuuteen ja epävarmuuteen. Olemassaolevia ratkaisuita ilmiön korjaamiseksi on olemassa, mutta ne ovat joko alustasidonnaisia tai hyödyntävät kolmannen osapuolen työkaluja.

Auttaaksemme “Configuration drift” -ongelmien selvittämisessä kehitimme ratkaisun nimeltä Driftless, joka hyödyntää ja automatisoi IaC viitekehyyksen omia muutoksenhallintakyvykkyksiä näkyvyyden luomiseksi IaC:n hallinnoimaan ympäristöön sekä uusien resurssien tuomiseksi osaksi IaC määrittämiä. Se myös etsii uusia entuudelta hallinnoimattomia resursseja, sillä IaC-viitekehyykset eivät itsessään tiedä niiden olemassaolosta.

Driftless tulostaa tietoja muutoksista tutkimuspiiriimme kuuluville resursseille, minkä pohjalta käyttäjä voi sovittaa yhteen infrastruktuurin ja IaC-määrittäykset. Lisäksi, se generoi onnistuneesti tuontikomennot hallinnoimattomille resursseille, ja nämä komennot loivat toimivat IaC- sekä tilamäärittäykset uusille resursseille. Näitä kahta tulostetta yhdistelemällä saatiin yhteensovitettua todellinen ympäristön tila haluttuun tilaan sekä IaC-hallinnoituissa että -hallinnoimattomissa osissa infrastruktuuria.

Avainsanat Pilvilaskenta, Terraform, configuration drift, muutostenhallinta, infrastruktuuri koodina

Preface

I want to thank my supervisor Lachlan Gunn and my advisor Sami Kairajärvi for their guidance, this was a fun journey of uncertainty and adaptation.

Otaniemi, 27 April 2025

Peter Otso Daniel Pohjola

Contents

Abstract	3
Abstract (in Finnish)	4
Preface	5
Contents	6
Symbols and abbreviations	8
1 Introduction	9
2 Background	10
2.1 Cloud Platforms	10
2.1.1 Cloud computing	10
2.1.2 Google Cloud Platform	11
2.1.3 AWS	12
2.1.4 Microsoft Azure	13
2.2 Change in infrastructure as code	13
2.2.1 Terraform	14
2.3 Change Management	15
2.3.1 Evolution of traditional software systems	15
2.3.2 Cloud infrastructure change management	18
2.3.3 Agile change management	19
2.3.4 Configuration drift	20
3 Problem Statement	22
3.1 System Model	22
3.2 Accuracy requirements	22
3.3 Performance requirements	22
3.4 Adaptability requirements	23
4 Design	24
4.1 Detecting drift in managed environment	24
4.1.1 Terraform plan	25
4.1.2 Third party tools	25
4.2 Detecting drift in unmanaged environment	26
4.2.1 Third party tools	26
4.2.2 Terraform import command	27
4.2.3 Terraform import block	27
4.2.4 Result validation	27
4.3 Data retrieval strategies	28

5	Implementation	29
5.1	Mitigating configuration drift in managed infrastructure	29
5.1.1	Extending Terraform plan	30
5.2	Configuration drift in unmanaged infrastructure	31
5.2.1	Querying data	31
5.2.2	Parsing Terraform import parameters	32
5.2.3	Import statements	32
5.3	System design	33
5.3.1	Driftless dependencies	33
5.3.2	Command line arguments	33
5.4	End results	33
6	Evaluation	35
6.1	Accuracy requirements	35
6.2	Adaptability requirements	36
6.3	Performance requirements	36
7	Related work	38
7.1	Terraformer	38
7.2	Cloud Concierge	38
7.3	Terragrunt	38
7.4	GCP Deployment Manager and Config Connector	39
7.5	Comparing Driftless to related tools	39
7.6	Infrastructure as Code research	40
7.7	CSBAuditor	41
8	Discussion	43
8.1	Multiple project support	43
8.2	Further integrations	43
8.3	Orchestration platform support	43
	References	45

Symbols and abbreviations

Abbreviations

IaC	Infrastructure as Code
GCP	Google Cloud Platform
CSP	Cloud Service Provider
AWS	Amazon Web Service
OS	Operating System
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service

1 Introduction

Cloud computing has had a major influence in the shaping of the current era of internet, where low response times, scalability and large amounts of content delivery are expected, offering advantage with reduced and flexible costs, on-demand resources and wide range of tools. Cloud infrastructures expand faster than traditional on-premise infrastructure setups, as the slow and expensive tasks of provisioning and managing physical resources are left to the cloud provider. In the past, expanding on-premise infrastructure might have taken weeks, while in cloud, a new resource is up and running just in minutes. Moreover, managing changes in this type of environment is efficient: the cost is lower, the services are less monolithic and the release cycles tend to be short. [1][2][3]

To keep pace with cloud ecosystems' continuous change, they are often managed using Infrastructure as a code (IaC) [4] [5]. IaC enables managing even large cloud infrastructures, sometimes containing hundreds of thousands of devices [4], by making configurations and deployments both reproducible and centralized. This reduces the likelihood of manual errors and misconfigurations, increasing consistency and repeatability across multiple environments. As Morris states in [4], a growing system "becomes harder to understand, harder to change, and harder to fix. The work involved grows with the number of pieces, and also with the number of different *types* of pieces". Consequently, utilizing IaC streamlines resource orchestration and management and also minimizes variation of the resources in the system. However, utilising IaC introduces the challenge of managing additional configuration drift, where the cloud environment drifts from its IaC configuration. Configuration drift accumulates over time and causes uncertainty and increased complexity in the environment, if left unresolved.

We develop a tool, Driftless, for reducing configuration drift between the cloud environments and their IaC configurations, by increasing visibility to infrastructure changes and importing unmanaged changes to the IaC system. While change management is relevant for every platform, this thesis focuses on controlling change in Terraform managed GCP environments, and utilizing Terraform's native plan and import tools. Currently, using IaC to manage change in larger ecosystems requires manual effort from the user, and Driftless aims to automate these processes. In addition, some solutions to preventing configuration drift exist already, but these do not aim to utilize IaC native functions the same way as our approach.

2 Background

While IaC can be used for various infrastructures, this thesis focuses on cloud platforms, since our target environment has been built in GCP. IaC has had a significant impact on the development and management of modern cloud infrastructure, making these concepts inseparable.

This section explores the relevant topics to this paper, starting with cloud platforms, IaC and Terraform. Lastly, we provide context for change management as a research field at the end of this section.

2.1 Cloud Platforms

Cloud computing is similar to 1950s centralized computing concept, where massive and expensive mainframe systems were used to centralize data processing, instead of distributing the computational power to separate personal computers. However, instead of physical centralization, cloud computing distributes data processing over different data centers globally. The first major cloud service provider was Amazon that launched S3 storage service with the intention of storing customers' data in a secure way, without sacrificing privacy or controllability. Amazon followed up with computational service EC2 later in the same year, allowing their customers to setup IT infrastructures remotely and with lower effort compared to physical infrastructures. This thesis will mostly discuss the three largest cloud providers, AWS, GCP and Microsoft Azure, which take around 70% of the cloud computing market [6].

2.1.1 Cloud computing

There has been a dominant definition of cloud computing over the last decade. In 2011, after cloud computing had started to become mainstream, the National Institute of Standards and Technology (NIST) released the definition in two parts describing cloud computing and its use cases. The first part outlines cloud computing reference architectures, such as actors, activities, methods and standards, and guides cloud architecture development. The key actors mentioned by the definition are the cloud consumer, cloud service provider (CSP), cloud broker, cloud auditor and cloud carrier [7]. The second part is commonly general cloud computing definition, publication 800-145, with the title "The NIST Definition of Cloud Computing", and it has been highly cited in the research field [8]. According to this definition, cloud computing is a model for ubiquitous and on demand network access to shared computing resources, whose provisioning and configuration is easy and require minimal interaction with the provider. It also lists cloud computations essential characteristics to be

- on-demand self-service
- broad network access
- resource pooling

- rapid elasticity
- measured service.

Furthermore, [8] divides cloud computing service models to three different categories: Software as a service (SaaS), Platform as a service (PaaS), and Infrastructure as a service (IaaS). These differ in how control and management responsibilities are shared between the CSP and the user. SaaS services are CSP's applications running in the CSP cloud environment, and while the user has minimal control and responsibilities, the major service providers allow configuring them through an IaC framework. PaaS hands over more control to the user, by allowing development and hosting of a software, still without having to mind the underlying infrastructure, such as networks, databases or operating systems. IaaS hands over even more control to the user, allowing them to configure operating systems, storage, deployed applications and selected network components. [8] [9]

The NIST cloud computing definition is still relevant and continuously referred to; however, the definition is fourteen years old, and since its release, cloud technologies have evolved rapidly and vastly. The validity of the old definition was tested by [10] in their article “Game of Definitions — Do the NIST Definitions of Cloud Service Models Need an Update? A Remark”. In the article, Vishal et al. explains how two new cloud services, serverless and containers, do not fit under the NIST's model, and they introduce alternative definitions for cloud computing service models.

According to the article [10], neither the serverless or containers fit under the NIST definition of cloud service models. With serverless, the user has a little more ability to configure applications to be SaaS, and not enough ability to control data or application backend to fit NIST's PaaS definition. Additionally, containers position between IaaS and PaaS: the CSP manages too much of the OS configurations to fit to IaaS definition, and the user has too much control over the middleware and runtime to be PaaS. To add to the confusion, different CSPs cross-categorize serverless and containers inconsistently between different service models [10].

Since cloud technologies seem to be moving away from NIST service model definitions, Vishal et al suggest two alternatives. The first is “anything as a service” (XaaS), which can be any IT service set up in the cloud instead of on-premises [9]. CSPs have already started creating XaaS like services, such as database or storage as a service (DBaaS and StaaS), as this removes the confusion of old NIST defined service models and focuses on the offered functionality.

2.1.2 Google Cloud Platform

A preview version of GCP and Google Apps was released in 2008 and became the first PaaS cloud provider in the industry, offering dynamic webserving, persistent storage, scaling and load balancing [11] [12]. Its full release took place three years later [13]. Google Cloud Platform (GCP) has grown steadily in popularity over the years, and its market share was the third largest (11%) in the global cloud market, in 2023. Some of the GCP interaction channels include:

- **GCP Cloud Console**, a basic web interface that is easy to operate without coding knowledge.
- **CLI tools**, scriptable and direct control over the GCP infrastructure. Uses REST API in the background.
- **REST API**, allows elaborate control over GCP infrastructure and integrating into GCP.
- **GCP Deployment Manager**, a native way to automate GCP resource deployment.
- **SDK libraries**, Google offers libraries for many program languages for integrating GCP operations into custom applications.
- **Infrastructure as code frameworks**, such as Terraform, Ansible, Pulumi.

A recommended change management practice is to choose one of these as the primary change management channel [14][15]. In addition, this thesis only applies to GCP infrastructures that utilize an IaC as the primary change channel. Changing a cloud environment through any secondary channel increases its uncertainty, as these channels are less likely to be monitored and covered by change management processes, leading to drifting configurations.

In 2017, Google introduced the Google Cloud Terraform provider, allowing users to automate the provisioning and management of Google Cloud resources by using an IaC framework. Before this, the only tool for configuration automation and orchestration for GCP resources was Google's own Deployment Manager released in 2015. In GCP, Terraform is used to automate the provisioning and management of virtual machines, serverless and storage, and to manage services, such as Google Kubernetes Engine (GKE) clusters. Terraform is especially useful for managing large-scale deployments, ensuring versioning and configurations across multiple environments.

Using Terraform in GCP has its limitations. To prevent accidental roll-backs or pushing conflicting changes, and to enable collaboration, the Terraform state files have to be stored remotely, usually in a remote backend, such as Google Cloud Storage or Terraform Cloud. Managing the state files securely and granting proper access controls is critical, as a state file may contain sensitive information. Also, Terraform may not always support the latest GCP features immediately, delaying adoption of new services or updates.

2.1.3 AWS

AWS was the first major IaaS cloud service provider, launching in 2006, and it has had the largest market share in the industry since (32% in 2023) [6]. AWS offers similar interaction methods as GCP: a web console, CLI tools, REST API, SDK libraries and similar IaC tools, including, but not limited to, Terraform, Ansible and Pulumi.

In 2011, AWS launched its own IaC framework called CloudFormation. It is a comprehensive interface for configuring and versioning AWS environments and

orchestrating deployments. Both CloudFormation and Terraform support modularity: CloudFormation modules can be saved into AWS registry, allowing reuse of the configurations inside either single or multiple accounts, and similarly HashiCorp offers a Terraform module registry. Terraform offers 123 built-in functions [16], increasing flexibility and allowing writing of complex logic and conditions, while CloudFormation is limited to 16, although it can integrate with AWS's Lambda module, allowing development of more complex logic [17][18]. Similarly to Terraform, CloudFormation gives detailed information of its configuration drift. AWS has an extensive support for their own IaC, but they also cooperate with HashiCorp to maintain and develop a Terraform provider for AWS.

2.1.4 Microsoft Azure

Microsoft Azure is the second largest cloud provider after AWS. While AWS has its CloudFormation and GCP its Deployment Manager, Azure also offers its proprietary IaC system to streamline the provisioning and management of resources directly within the Azure ecosystem. Azure's IaC tools include Azure Resource Manager (ARM) templates and Azure Bicep, a newer IaC language for Azure. ARM templates are a declarative way to define infrastructure in JSON format, while Bicep offers simpler syntax, reusable modules and supports all Azure services and API versions out of the box [19][20]. Neither Azure ARM or Bicep has prevention mechanisms for configuration drift, and Bicep also offers a vast library of built-in functions similar to Terraform [21].

2.2 Change in infrastructure as code

The core idea of Infrastructure as Code (IaC) is to use machine-readable code to manage and provision computing infrastructures. One of the main benefits of this is replicating the agile properties of code and introducing agile software development principles into infrastructure management. One of these principles is to test changes as soon as they are complete, which is especially important in cloud ecosystems, given quickly they are able to change [4]. To integrate agile principles to IaC development, changes to the infrastructure have to be made through an agreed change channel, which is tested and monitored throughout the development life-cycle, reducing the risk of infrastructure breaking down. However, when agile methods and controls are ignored and the infrastructure is modified through an alternative change channel, the change information is not properly communicated to other stakeholders, and their potential impact to dependencies or correctness cannot be assessed or predicted.

Cloud infrastructures can be managed by a number of IaC tools, the most popular being Ansible, Terraform, Pulumi, Crossplane and Chef [22]. Terraform, Pulumi, and Crossplane implement the declarative paradigm, describing what the desired state should be without specifying how to get there, and in contrast, Chef implements the imperative paradigm, describing how the infrastructure or system should be configured, step-by-step. Ansible is a combination of both paradigms.

2.2.1 Terraform

Terraform is an open source tool developed by HashiCorp, and can be used for managing and deploying virtual infrastructures using declarative code that can be versioned and reviewed similarly to software code. Terraform supports various public and private cloud platforms (e.g. GCP, AWS and OpenStack) and has become a key component in the field of DevOps, being the second most popular IaC framework in the industry [22]. HashiCorp maintains a Terraform provider jointly with cloud service providers, and the provider maps Terraform and cloud resources together, allowing Terraform to provision and manage cloud infrastructure through an API. Terraform functions by calling cloud provider APIs according to users instructions in the Terraform configuration. As a result, instead of deploying the changes manually, the Terraform configuration is modified to achieve the same result. This configuration file can define all cloud resources, such as networks, DNS records and databases, and can be stored and versioned in a code repository.

Every time Terraform creates or modifies infrastructure, it records the information about those resources into a state file. This state file is a JSON representation of the real-world resources, which maps to the Terraform configuration. Each time Terraform is executed, it queries a cloud platform and compares the real state with the Terraform state, calculating the necessary changes to synchronize the Terraform state with the actual infrastructure. Terraform allows users to co-operate on a shared state file using remote backends, which store the state in a remote storage, taking care of synchronizing changes, locking the file and encryption. [23]

Terraform is used through a command line interface. Some essential Terraform commands for this thesis are:

- `terraform apply` - Applies the changes defined in the configuration file to the infrastructure, provisioning or updating cloud resources as necessary.
- `terraform plan` - Previews the changes, allowing users to review and verify the planned actions before applying them, preventing unintended modifications. Also, imports state entries for existing resources defined in Terraform configuration import blocks.
- `terraform import` - Creates Terraform state entries for existing resources that are not yet managed by Terraform.
- `terraform state show` - Displays the current state as recorded in the Terraform state file, providing a view of how resources are configured within the infrastructure.

“Terraform plan”’s output displays the user which resources and resource configurations are different in the cloud, and how running “`terraform apply`” would change the environment to match the current terraform state. However, it does not clarify the intention of the changes or whether the changes should be reverted or kept. Furthermore, the software systems complexity or uncertainty remains unchanged, until the user investigates which of the reported changes are unwanted or which changes should be included in Terraform.

From the perspective of change management, Terraform's state management introduces two issues addressed by this paper: Terraform can only operate on one state file at a time, and it only tracks the information of the resources it creates – any cloud resources not present in the state files do not exist to Terraform. This challenges large projects, where Terraform state is often divided into logical sections across multiple state files. Additionally, due to unmonitored resources created by people and processes, the actual environment state can differ significantly from the present IaC definition. This type of contradiction in the state between the environment and IaC configurations is called a configuration drift [4]. This thesis aims to search the entire environment for configuration drift, covering all Terraform state files and unmanaged assets, reducing blind spots and helping to align the environment with its IaC definition.

2.3 Change Management

Cloud technologies have added a new dimension to the field of change management with their fast-paced change ecosystems and emerging cloud migrations of on-premises environments. This section explores whether software evolution models are still relevant with modern challenges, what does change mean in IT infrastructures and how does it occur.

2.3.1 Evolution of traditional software systems

In 1980, Meir M. Lehman introduced a number of principles that are called Lehman laws of software evolution, which originated from his long career in software development and management [24][25]. He noticed that software management process had recurring patterns, which could be used to describe software evolution, complexity and change management over time. The eight laws were:

1. Continuing change: For a software to remain useful and relevant, it must constantly adapt and evolve to match environment's and user's changing requirements. Otherwise, its usefulness decays.
2. Increasing complexity: A software becomes more complex by time as every change increases it, unless work is allocated to decrease it.
3. Self-regulation: Software system development does not happen randomly, and stabilizes over enough time and development.
4. Conservation of organizational stability: Organizational resources define limit the amount and range of changes, and the resources put to each release is about the same.
5. Conservation of familiarity: Large and complex software systems require increasingly extensive knowledge to develop new features. The knowledge of the developers limits development over time, and the amount of developed new content either stays the same or decreases.

6. Continuing growth: A software must grow over time to continually provide value to users and to respond to new requirements.
7. Declining quality: A software will deteriorate over time, unless carefully maintained and adapted.
8. Feedback System: Software evolution is an iterative process, and steering it requires constant feedback to improve and adjust to match changing requirements.

In addition, Lehman introduces concepts related to software program life cycles, and different categories of programs, depending on their purpose. All these ideas built a foundation for software change management, but they are very outdated to be applied as such in such a rapidly developing field.

The applicability of Lehman's laws to modern software development has been critiqued by Godfrey [25] in their paper *Journal of Software: "Evolution and Process"*. They compare how software development has evolved over the last decades and highlight how the laws apply less to modern software architectures and characteristics. Next, the paper's observations about modern software evolution and Lehman's laws are explained.

Software architecture interactions have evolved during the last decades. For example, instead of having to learn the entire software code base, it is often enough, for the developers, to focus on learning a specific part of the program. [25] uses Linux kernel and drivers as an example: often the driver developers do not have the knowledge of the kernel operations nor the other drivers, and they bypass the complexity of the kernel code by using kernel interfaces. Consequently, the driver developers only evolve and increase complexity in certain parts of the system.

Godfrey comments this being a reason the complexity of some modern systems can not be judged the way Lehman did it, by measuring simple, absolute and empirical metrics, such as sum of the entire system size and number of changes, as the system components' interaction and modularity have to be also understood. Softwares have also become less monolithic, and systems interact with various dependencies, such as libraries, virtual machines and services. This kind of complexity does not stem from the program's source code, but instead from the environment where the software system is embedded in. Lehman's metrics do not take this into account.

In addition, popularity of open software development have also made it difficult to evaluate how a developer contributes to a project: a large amount of project's code base can be adapted from an open source project, rather than developed from scratch. Modern softwares are collaborative projects of communities, and not tied to one organization or a team, which was assumed by Lehman. Lehman also wrote about software systems feedback loops, and how they receive new requirements and ideas from internal and external sources. However, the essence of internet has been redefined multiple times since 1960, quickening and increasing the volume of feedback software systems are affected by. As a side effect, the amount of interaction has enabled the rise of emergent use cases of many systems, fundamentally changing the way many programs are used, resulting in evolution that Lehman didn't account for. Godfrey's example of this is virtual machines, which were originally created for simulating

Lehman's Law	Critique
L1) Continuing change	The law remains relevant, but the pace and mechanisms of change have shifted significantly due to constant user feedback and agile software development methodologies.
L2) Increasing complexity	Complexity does still increase, but it is more isolated due to program modularity and interfaces.
L3) Self-regulation	Modern development strategies can be more spontaneous, less standardized and include more stakeholders, resulting in less normalized development.
L4) Conservation of organizational stability	Amount of work required for modern software releases based on the stakeholders and organizations involved.
L5) Conservation of familiarity	Although complexity in open source projects also increases over time [26] [27] [28]
L6) Continuing growth	No critique
L7) Declining quality	Software program quality does not necessarily decrease as fast anymore, since there are tools and methods to combat it built into software development methodologies.
L8) Feedback system	Modern feedback systems are faster and emergent use cases shape softwares' life cycle.

Table 1: Godfrey's critique on Lehman's laws.

multiple operating systems on one computer, but are now used as generic deployment platforms, virtual remote workstations and for malware research. Godfrey's critique of Lehman's laws has been further depicted in the Table 1.

In Lehman's time, there were not many tools for managing the pressure of continuous changes and a growing program. While Godfrey does not mention the effects of agile methodologies, such as continuous integration or testing, these methods enable faster changes and managing larger systems. Furthermore, these enable implementation of changes without affecting the rest of the software system or throughout planning. Modern automation and testing catch issues earlier, allowing them to be also fixed earlier, preventing spreading them further, and modern development methods decrease the complexity that was formerly caused by software changes. In addition, open source systems are especially prone to falling into ad hoc development and design practices and lack of formal processes, which may lead to decline in software quality. Consequently, Open source projects are expected to reach a phase where complexity reaches level that is too high for the community to maintain [26].

Evolution of software systems' complexity has been further researched by Alenezi and Almustafa [26]. In their paper, they measure the McCabe's Cyclomatic Complexity (CC) score and Source Lines of Code (SLOC) of five different size and mature open source systems. Larger scores indicate high software complexity, challenges to understand or to change the software, and increased likelihood to containing bugs.

Every source code base grew over ten releases, conforming Lehman's sixth law (Continuing growth), however the complexity stayed the same over time. Alenezi and Almustafa interpret the unchanged complexity score as proof of Lehman's second law (Increasing complexity), as they view it as an indication of "there is no extra precaution measures to decrease the complexity of these systems.". However, in our opinion, this conclusion does not follow Lehman's second law, as a growing systems should also grow in complexity: if the complexity score stays static over time, some work has been done to decrease it.

The most comprehensive definition for "change" has been published by McGee and Greer, who highlight five change areas of change and also classify change causality as either a trigger or uncertainty [29]. The five areas are external market, customer organization, project vision, requirement specification and solution. However, in reality, the cause of change can be difficult to classify, and it can be a mix of uncertainty or a trigger event. The book [30] also divides the change causes further into corrective, adaptive and perfective, based on the purpose of the change. Corrective change, (called corrective maintenance in the book), aims to correct existing errors, adaptive change aims to adapt into new requirements and perfective change aims to either enhance or prepare the software system for future requirements.

2.3.2 Cloud infrastructure change management

Compared to traditional on-premises software architectures, cloud is a multi-stakeholder and heterogeneous environment, with multiple SaaS, Paas and IaaS layers. Pahl et al. extends Lehman's ideas and write in their paper about maintaining sustainability in cloud infrastructures and the strategies for evolving and adapting it [31]. The paper introduces concepts of "patterns", "models" and "rules" that define abstractions utilized by cloud maintenance. Patterns are used to describe how uncertain situations emerge in common recurring situations, which are defined by models, and rules are guidelines to help architects decide how to introduce changes while ensuring the sustainability and continuity of the system. Models should cover the cause, type and impact that a change has on the infrastructure, and what kind of rules it requires to be successful.

They separate the sources of changes into evolution and adaptation based on the predictability of changes. Both evolution and adaptation refer to the ability to modify software system's architecture while still meeting system's goals, evolution referring to unanticipated change and adaptation to anticipated change. Evolution and adaptation arise from two sources: changes in requirements and system goals, or changes in the operational aspects of the architecture [31]. These two categories follow the theme of dividing change origin to conscious decisions, uncertainty or accidents, as highlighted in other research [32]. However, other studies have not distinguished planning and preparing for a change as a separating factor for change. Fundamentally, the main motivator to change results from business drivers; IT market demands adapting, expanding and flexibility - all of them are also reasons why companies migrate to cloud and to IaC.

Where Lehman talks about complexity, Pahl et al. also utilizes uncertainty as a

metric of software system evolution. Managing change involves minimizing uncertainty of infrastructure, which can arise from resources, the presence of multiple shareholders, distribution, and heterogeneity [31]. Additionally, evolving systems grow increasingly complex, as mentioned by Lehman, leading to greater uncertainty unless it is mitigated or intentionally maintained. Key preventive actions for accumulating uncertainty include analyzing, planning adaptation, monitoring, and enacting changes [33].

To prevent the negative sides of change, preparing for evolution and adaptation should guide all stages of infrastructure management, and valid goals, derived from user stories, must be actively enforced [31]. Part of this preparation should be accounting for configuration drift and choosing the infrastructure's source of truth, such as an infrastructure as code configuration that defines the composition of the infrastructure. The paper lists six different sources of uncertainty: monitor, model, environment, adaptation, change enactment, and goal uncertainty. Of these, this thesis aims to reduce monitoring and environment uncertainties by extending the monitoring scope to unmanaged assets, applying identical and centralized configurations, and minimizing configuration drift.

2.3.3 Agile change management

Agile change management has been a rising trend in the 2000s, ever since agile manifesto was released in 2001. Its purpose was to define values and basic principles present in earlier success stories, such as NASA's project Mercury, and apply them to software development, resulting in various frameworks, such as Scrum, Lean and Extreme Programming [34]. Change management is built into agile processes, encouraging change during each iteration. Agile reduces the risk of communication gaps and overly ambiguous requirements by following processes, such as face-to-face communication, review meetings and requirement prioritization [32].

As a software system develops, its requirements may also evolve as stakeholders gain understanding of the use cases and context related to the project. Communicating the changing requirements requires frequent interaction between the shareholder groups and developers and administrators, and the agile processes aim to increase this. Frequent communication and confirmation allows the software stakeholders to steer the application to correct direction quickly, and early changes to the requirements decrease the number of software changes later on. In addition, this way stakeholders may validate the requirements and changes in the earliest phase possible [32]. However, these processes are imperfect: face-to-face communication requires availability and willingness of the stakeholders, customer involvement can be pointless with the wrong representatives and estimating schedules and budgets upfront is challenging or even impossible due to constantly changing requirements, which can result in overruns [32].

Based on the agile activities present in the research field, [32] recognizes three common sequential areas of change management: change identification, change analysis and change effort estimation. These areas are not standalone or isolated from each other, and instead, every area verifies its outcome is in line with previous results and the expectations of stakeholders. An outcome of an area might also provide extra information to its previous steps: e.g. , outcome of change effort estimation could

reveal additional change analysis details that should be re-evaluated.

Change identification creates a base for the overall change process, and it consists of change elicitation and change representation, where change details, such as what needs to be changed and where, are identified, classified and gathered with the stakeholders. Change analysis follows identification, creating understanding on the consequences of the change, such as the impact on the software system and potential side and ripple effects, with the help of user stories and product backlogs. Change analysis helps to estimate what has to be changed to accomplish the requirements, and how much time, money and resources it requires. Its result should always be compared to the result of the change identification and stakeholder feedback. Lastly, change cost and effort are estimated, starting with calculating the size of the change. The result is then confirmed with the previous steps. [32]

2.3.4 Configuration drift

According to [35], configuration drift can occur due to unmonitored changes made by a human or a micro service, creating blind spots and impeding the ability to carry out administrative tasks across the environment, such as monitoring access controls, patching resources, and enforcing IaC configurations and standards. The drift makes fault isolation and error mitigation increasingly difficult, especially when undocumented changes accumulate over time [36] [37]. [4] lists a number of configuration drift prevention methods:

- Minimize automation delays
- Minimize ad hoc changes
- Apply and enforce configurations continuously
- Create immutable infrastructure

The longer the deployment of new infrastructure configurations is delayed, the higher the likelihood of failure due to changes in the system, patches, or transitive changes to dependencies. Additionally, long deployment breaks often lead to larger deployments, which complicates the debugging process, as more things have changed. Unplanned, ad hoc modifications are a prevalent issue that create variance to resources that were initially configured identically. These changes often exist for valid reasons, such as business-critical patches or fixes applied to production resources, but they accumulate over time, further increasing the deviation from the originally intended configuration. They further complicate the problem solving process even further: an ad hoc change in production environment could create a problem that is not present in other, supposedly identical, environments and therefore might remain undetected. [4][37].

One way to prevent configuration drift is continuously enforcing IaC configurations. This reverts any changes that cause configuration drift between the actual infrastructure and the IaC configuration. Some IaC frameworks, such as Chef and Puppet, do this

automatically, but others require a master server that enforces its configuration state to the cloud infrastructure [4][23]. Although Terraform operates without a master server by default, it attempts to synchronize its configuration with the cloud infrastructure by querying the cloud provider's API. Another way to prevent configuration drift is immutable infrastructures. These do not permit changes to the deployed resources, and instead of modifying existing instances, new ones are created. For example, instead of manually patching each server, the general server image is replaced with a patched one. While this approach can create challenges with downtime, it prevents disparities between cloud resources by ensuring uniform changes across all instances. Immutability is further used by some provisioning tools, such as Terraform, which use immutable operating system images to deploy new servers, reducing probability of configuration drift. [23]

Drifted cloud resources can impact security of the whole infrastructure. This can manifest in inconsistent configurations for firewall rules, database credentials, key vault accesses, and orchestration settings. Security issues can also be complex and context-dependent: for instance, an API gateway with a seemingly large request limit may expose a less busy service to denial-of-service or resource-based attacks, while the same limit might be necessary for the operation of a larger application, where applying a smaller limit could cause disruption. Often, a security-sensitive change has a purpose that needs to be understood before addressing the issue, even if it was implemented by bypassing the change management process.

3 Problem Statement

The previous research overviews the causes of increased complexity, uncertainty and configuration drift in software systems. The primary objective of this thesis is to reduce these phenomenon in IaC ecosystems, by developing a tool that identifies drifted configurations in the IaC managed environment and looks for new, unmanaged, resources that can be imported as IaC framework's configuration data. The new resources are changes in respect to the existing state of the framework, as they should not exist, and thus are part of the configuration drift. The tool is called "Driftless", and it is designed to run periodically and automatically as part of a system monitoring process, generating real-time view of the environment drift.

3.1 System Model

Driftless is developed using the C# programming language, but the same result can be achieved using most languages. As inputs, it must receive a IaC project directory and the target cloud environment identifier, since these are needed to identify the project environments. For simplicity, Driftless supports only a single IaC and cloud project per execution, and even though much of commercial IaC usage is based around remote repositories, Driftless only works in local environments. The model scope is further narrowed by limiting the tool to cover only resources present in our environment. To provide insight on all changes in the target environment, Driftless's output must include both managed and unmanaged changes present in the system. A key aspect is also generating resources for importing unmanaged infrastructure into the IaC infrastructure, as no existing solution does this, and this gives the user the opportunity to fine-tune the import resources, before starting the process.

3.2 Accuracy requirements

The imported resource values and reported managed changes must be identical, when compared to the actual values in the cloud infrastructure. The tools coverage must also cover all of the resources in our environment. Driftless must provide accurate data on the unmanaged and managed changes, as inaccurate values can cause disruptions to live systems when applied by the IaC framework.

3.3 Performance requirements

Driftless must be able to run periodically as part of environment maintenance processes, not taking more than two to three hours for large target environments. It should be used frequently, as actively monitoring changes prevent configuration drift from building up [4]. To keep the performance good, especially the code sections that process cloud data must account for larger data set sizes. Its efficiency is important, since it attempts to process every change in a cloud project that may contain large amounts of resources.

3.4 Adaptability requirements

IaC can be utilized with many platforms, and our design must be compatible for as many of these as possible, at the least the three major cloud providers. This is done by utilizing IaC framework's native functions instead of custom or third party solutions.

4 Design

Target resources can be divided into managed and unmanaged resources, based on whether they exist in the IaC state and whether IaC framework is aware of their current state. When they are modified via other channels than IaC, these resources become drifted from their original configurations. In this section, we present a tool design, Driftless, for detecting these drifted changes in both managed and unmanaged resources in IaC-based cloud environments. In addition to detecting changes, one of its key goals is converting unmanaged resources to managed IaC resources, by importing them into the IaC framework, granting future visibility. Driftless will be designed around GCP and Terraform, because of our test environment design, however, the overall concepts, such as change management and data querying, can be extended to other frameworks. Terraform is supported by thousands of platform providers, and each of them that allow multiple change channels can also accumulate configuration drift. Next, we will explain our approach, starting with managing changes in Terraform infrastructure, and continuing to detecting additional unmanaged resources and their import process.

4.1 Detecting drift in managed environment

When IaC managed environment is drifted by a secondary channel change, IaC framework's state and the actual state differ. Manually detecting this can be done by comparing IaC resource states to their corresponding states in the target system, and converting the state data to a matching format. Implementing this comparison logic is complicated, as this requires knowledge of field mappings between IaC objects and their target platform counterparts, which have inherent structure differences, further presented in Figure 1. Formulating IaC and target field associations is manual work, and field pairs have to be matched based on the similarity of their values. In Terraform, these associations are already built-in, considering it can transform Terraform objects into target objects, and they are part of the provider libraries, jointly maintained by HashiCorp and the owners of the target platforms [38]. This mapping process can be facilitated using Terraformer, considering it translates cloud objects into Terraform configurations, unifying the formats. This third party solution makes the manual comparison easier, however, change detection process is already built into Terraform for managed resources and is a fundamental feature for managing infrastructures.

```
Terraform Config:
resource "google_compute_instance" {
  "vm_instance_2"
  boot_disk {
  initialize_params {
  image = "debian-cloud/debian-11"
  } } }
```

(a) A Debian image definition in Terraform.

```
GCP Output:
"disks": [
"licenses": [
"https://www.googleapis.com/compute/v1/
projects/debian-cloud/global/licenses/
debian-11-bullseye"
] ]
```

(b) A Debian image definition in GCP.

Figure 1: Terraform and target platforms have different object structures and field values for the same resources, making manual comparisons of field values challenging.

4.1.1 Terraform plan

Terraform compares environment states automatically when applying configurations, and also offers a tool, `plan`, which triggers the comparison independently from applying. The Terraform `plan` command functions by comparing a single Terraform configuration file to resource data in the target system, and it outputs a simple list of the changes. However, as it is limited to a single configuration file, and as Terraform projects may contain multiple state files, our tool needs to extend the `plan` functionality to cover all of the configuration files.

After forming the comparison, the user must try to align the environments: any desired changes must be implemented in Terraform configuration - any changes that are not, will be removed automatically by Terraform next time configurations are applied. The desirability is affected by context dependent attributes, such as intention, security and business purpose. Our tool leaves this part of the process up to the user, as this is potentially destructive. The end result must be validated based on two factors: how well does Terraform `plan` fit our use case and detect changes to the managed infrastructure, and does our tool extend this functionality to all configurations successfully. All three categories of the infrastructure, database, network and application, will be changed to create informative enough result.

4.1.2 Third party tools

The tools Terragrunt and Cloud Concierge can be used as an alternative solution for extending Terraform `plan`. Terragrunt does this by using “stacks” to treat multiple Terraform directories as one project and relies on Terraform for the actual comparison, but Cloud Concierge has its own internal logic for detecting and comparing the managed environment. While both tools require setting up their own configuration files in the project environment, Cloud Concierge’s requires setting up more detailed data of Terraform and GIT environment. We chose to use the Terraform native `plan` that does not require setting up an additional configurations in the project file system.

4.2 Detecting drift in unmanaged environment

An IaC framework does not control changes in unmanaged resources, and they are unknown and not part of the IaC environment by definition. If done manually, the process of identifying them is similar to identifying managed ones: by comparing the existing cloud infrastructure to the Terraform state. However, identifying them requires additional filtering step, where unknown resources are separated from the existing ones. This is done by cross comparing the cloud infrastructure identifiers with IaC framework's state identifiers, and identifying which resources are not currently present in the IaC state. In GCP, fetching the cloud infrastructure data can happen in multiple ways, such as via a CLI tool "gcloud", WEB API or SDK libraries, the last one being most suitable for our approach because of its higher abstraction level that streamlines our application development.

Importing the new resources to the IaC framework is the next problem after filtering their details. First we will explain the manual way of doing this, and then the existing solutions, including the Terraform native way. Manually importing unmanaged resources requires a similar field mapping between IaC framework and target object formats, but this time, the purpose is converting them to new Terraform configuration and state entries. Creating both entries is vital: if only configuration entries exist, Terraform will try to create new cloud resources as they do not yet exist in its state, and in case of only having state entries, Terraform will destroy any associated cloud resources, as they must not exist according to the configurations. Because Terraform configurations and states are in different formats, two mappings are required. After the field mappings have been used to create new configurations and states for unmanaged resources, Terraform will start tracking changes made to them, making them manageable.

4.2.1 Third party tools

Terraformer can be utilized here similarly to the previous use case to ease the demanding mapping process by unifying the target system's output format. Terraformer is also directly capable of generating the configurations and states, but it does not separate managed cloud infrastructure from unmanaged, and therefore utilizing Terraformer for our use case requires first filtering out the unmanaged resources manually. The only input Terraformer requires, are the target cloud resources to convert, and its results must be verified by the user, as implementing faulty ones may damage the environment. As a critical downside, Terraformer does not support all of the objects in the test environment (google service networking connections). Cloud Concierge is another open source tool capable of generating the configurations and states for unmanaged resources. It utilizes Terraformer for transforming them into Terraform format, leading to similar end result as well, except the tool separates managed changes from unmanaged ones. As a slight downside, Cloud Concierge needs its own configuration setup in a project.

4.2.2 Terraform import command

For simplicity, we prioritize utilizing Terraform native tools to achieve our goal. Similar to `plan` Terraform has an import functionality `import`, which imports unmanaged resources by mapping user defined the cloud infrastructure to Terraform's state file and can be done either by executing Terraform import command or by inserting special "import blocks" in the configurations. As the functionality does not import configurations, the import command method requires the user to predefine general configurations for the imported resources, the structure varying based on resource type. This is a difficult precondition to scale, as the exact details of all of the unmanaged resources have to be known. Terraform import command not synchronizing configurations with the state creates a risk of the applying the state before its integration into configuration has been fully completed. Consequently, this may lead to destruction of the assets, as Terraform state assumes its resources as destroyed, if they are not present in the Terraform configuration as well [39]. To prevent accidentally applying the import changes early, it is recommended to notify anyone involved in the deployment of the import activities taking place and to configure pipelines not to apply Terraform automatically. Both Terraform cloud and HashiCorp Sentinel allow disabling automatic applies in the CI/CD pipelines.

4.2.3 Terraform import block

Utilizing import blocks generates a state as well, however instead of exact configurations, it requires only two details; which cloud resource is to be imported and where [40]. As mentioned previously, a matching configuration is still needed to prevent Terraform from deleting the imported environment, and the manual approach is not scalable. Terraform has an experimental workaround for this when using import blocks without configurations, and it can generate them from the imported states with the `generate-config-out` parameter. Users are strongly recommended to inspect the generated configuration file manually, as it may have mistakes in complex cases and affects the environment after apply [39][40]. Other benefits of import blocks are predictability, compatibility with CI/CD pipelines and being able to preview the import operation before any modifications are done to the local Terraform state. For us, import blocks are preferable, as they do not require manually configuring the resources, and thus can be used without much knowledge of the cloud resources.

4.2.4 Result validation

To utilize Terraform's import solutions, our tool must generate commands or import blocks based on the unmanaged environment, which can be either executed automatically, or manually by the user. When validating this output, these generated import statements must be executable without additional changes and contain all the unmanaged resources in the environment. The result of Terraform import process must also be validated, and the configurations and states must match the previously unmanaged cloud environment. Terraform must also be able to monitor further changes to the imported objects, confirming that the unmanaged resources are now managed.

Lastly, the result must be compared to Terraformer and Cloud Concierge to gain insight on correctness of each solution.

4.3 Data retrieval strategies

Depending on the target platform, various methods can be used to fetch and process data, such as streaming, batching, and event-based processing. Conventionally, streaming is continuous flow of smaller data sets, can be real-time if the stream connection is kept awake, batching is intended for larger datasets, which are divided into smaller sections, “batches” or pages of data, which are processed separately and with longer latency, and event-based processing triggers real-time, but only for current objects. Based on this, batching is the best approach due to the large sizes of cloud infrastructures, at least to older environments with accumulated configuration drift. However, newer projects might benefit more from event-based processing, as this catches unmanaged resources once they occur and in addition, some platforms support streaming resource data on demand while it is being processed.

In the end, supported processing methods of the target system limits our choice. The way GCP sends query responses, when using the C# library, is by using an asynchronous aggregated streams, which return objects on demand every time Driftless requests for one, e.g. in a for loop. The benefit of this approach, is not loading all of the resources in the memory at once and not having to implement manual batch handling, such as paging, as the GCP API handles the batching process internally. However, as the cloud provider handles the batch process optimization, our ability to tweak the process is limited.

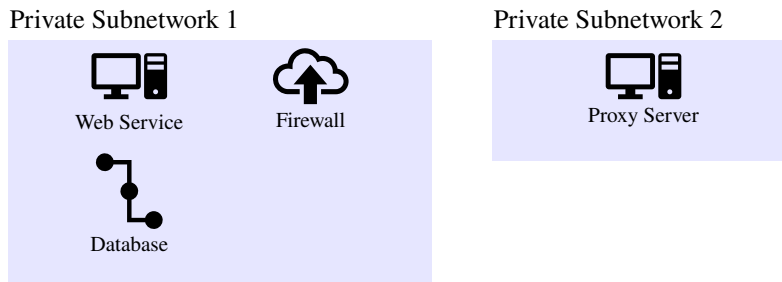


Figure 2: Access to the web service is limited to the two private networks, defined by firewall rules.

5 Implementation

Previously, we explored various approaches for detecting and mitigating infrastructure drift and synchronizing Terraform and target platform states. This section will introduce an application, Driftless, whose purpose is detecting changes in managed and unmanaged resources and generating resources that help resolving the differences in configuration states. It covers scaling Terraform plan command for projects with isolated configurations, processing of infrastructure data, generation of Terraform import statements and the system design of Driftless.

In our scenario, Driftless will operate against a target cloud infrastructure, presenting a private network segment with a private database and a web service only accessible through a proxy server. The infrastructure consists of virtual private networks, computational instances (virtual machines), a firewall and a database. These components form the managed environment, and they will be modified to create configuration drift. The modifications are described in the next section and the system architecture is depicted in the Figure 2, which will be referenced throughout this section to provide context for the implementation. Details regarding different infrastructure components are not relevant for understanding the implementation, hence most of them are omitted.

5.1 Mitigating configuration drift in managed infrastructure

As explained in the Section 4.1, IaC frameworks contain essential operations for managing IT infrastructures. Our IaC framework of choice, Terraform, an essential operation is `plan`, which compares the actual state to the desired state, and creates a plan for aligning former to latter, listing each change and the action required to reverse it. Terraform compares its configuration file to the actual state via the target platform's API, utilizing resource identifiers it stores. The list of changes and actions outputted by Terraform plan is detailed enough for handling drift in the managed infrastructure, however, the command covers only one Terraform configuration file at a time. Terraform projects can contain configurations in several directories, which is why Driftless has to run the Terraform plan command several times to cover all of the project configurations.

We exemplify configuration drift in managed assets by changing the firewall access rules to allow the private web service from public networks (0.0.0.0/0) and by enabling

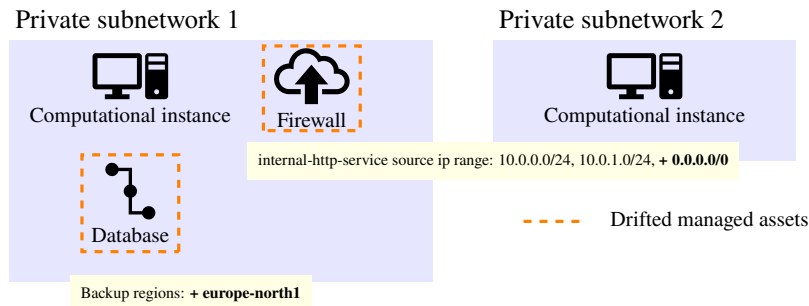


Figure 3: An internal http-service firewall rule was modified to allow traffic from any address, and a backup configuration was added to the database. These changes create configuration drift in resources already present in the infrastructure’s Terraform configuration.

the database backup configurations (Figure 3). The changes are deployed via the GCP cloud console, which creates a configuration drift between the GCP configurations and Terraform configurations. In addition, while both the firewall and the database are managed in Terraform, they are defined in separate Terraform configurations, meaning a single Terraform plan command execution will not produce configuration drift status for both of them.

5.1.1 Extending Terraform plan

We extend the Terraform plan command by repeating it once for every Terraform state file found in the project. An equally good alternative solution could have been utilizing and modifying OpenTofu, an opensource fork of Terraform, to accept multiple configuration directories during planning, however this project is specifically focused on Terraform. Repeating the plan command begins with Driftless receiving the project root directory as a parameter, which is used to search every subdirectory for a file with extension `.tfstate`, as Terraform allows renaming of the state files. Driftless then executes the plan command for each discovered state directory and composes a summary of the Terraform output, which displays the changes made to the firewall and the database and the action Terraform takes to revert the changes. Driftless executes Terraform with its default command arguments and trims off unnecessary and repetitive information, such as command instructions, from the Terraform plan output, any errors encountered during Terraform’s execution are displayed separately.

It is worth noting that this implementation only works for local Terraform states on the user’s hard drive, and the local state must be synchronized with any remote state repositories before running the program. Otherwise, Terraform will reference the old local version when identifying changes, potentially reverting the environment to an earlier state to match the outdated state. Supporting only local environments was a choice made out of convenience; however, it limits Driftless’s applicability, as larger projects often utilize remote repositories. Next, we will manage configuration drift in the unmanaged part of the infrastructure.

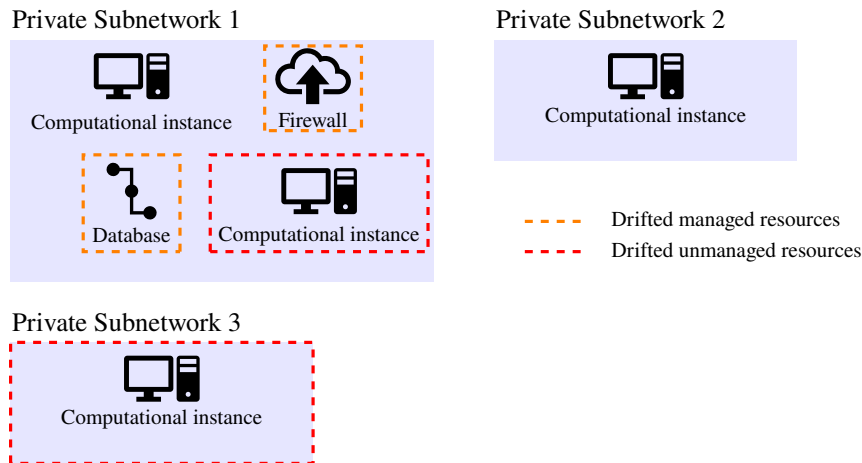


Figure 4: Two new virtual machines and a subnetwork were created through the GCP cloud console. These resources should not exist in the infrastructure according to its Terraform configuration.

5.2 Configuration drift in unmanaged infrastructure

To create resources not managed by Terraform, a third subnetwork and two additional computational instances were added using the GCP cloud console (Figure 4). These changes are not visible when executing Terraform plan: it is only able to manage the parts of the target infrastructure that are defined in its configuration; according to it, anything else must not exist. As a result, Driftless has to distinguish the added resources to determine which parts of the infrastructure are not managed by Terraform yet. This requires querying objects from the target system and comparing them with Terraform state, which is covered in detail in the next section.

Configuration drift caused by unmanaged resources can be negated by either removing the unmanaged resources or by importing them into the IaC framework, allowing it to oversee them in the future. To support the import process, Driftless will create import statements for our IaC framework, Terraform, by prefilling executable commands and command blocks for the unmanaged resources. Driftless queries this resource data from the target system and parses the values that Terraform import process requires.

5.2.1 Querying data

Driftless retrieves data from the GCP environment through its SDK libraries, each of the resource type requiring its unique processing logic. Reusing processing logic between the types makes Driftless more efficient, however, SDK class abstractions lack critical functions that allows modularity for the process, such as functions related to authentication or data querying. This is not an issue for an implementation that only covers a small environment, but because Terraform supports over a hundred GCP resources, this makes modularity and streamlining the interactions with GCP and response processing challenging. Moreover, Driftless functions by querying all available resources in the target environment, putting much emphasis on the

efficiency of the data handling. Our approach of creating for each resource its own data querying and processing logic is not unique, as a similar solution can be seen used in e.g. Terraformer [41].

Regardless of these class structure differences, GCP's responses always contain the needed data in similar formats. GCP returns data as aggregated and asynchronous lists, and takes care of sending it efficiently, making additional batching logic redundant. The response is parsed for two fields to construct Terraform import statements; this is explained in more detail next. The other fields would be utilized, if we were manually creating the Terraform configurations and states, and needed to create a GCP to Terraform format mapping.

5.2.2 Parsing Terraform import parameters

Before processing the received data for importing, Driftless compares its identifiers with Terraform state to determine which response objects are part of unmanaged infrastructure. Afterwards, the parsing process is only applied to unmanaged objects.

Driftless creates Terraform import statements for each individual unmanaged object, and these statements require three parameters: the Terraform resource type, the cloud resource name and cloud resource's identifier. For GCP, the identifier and name can be parsed from response fields `self_link` and `name`. `self_link` is in an URI format and contains a GCP identifier, which is parsed by following a regex pattern. The second field, `name`, corresponds to the resource name in GCP, and does not require additional processing: this name is used to name the Terraform resource during importing. Terraform import also requires the resource type of the new resource. Unfortunately, this can not be derived from the GCP response, and mappings between GCP and Terraform resource types are not readily available online either. As a result, this mapping has to be done manually, to insert each resource type string to its logic in Driftless. This project used the list of `google` Terraform provider resource types found in HashiCorp documentation to discover which GCP resource each relates to.

5.2.3 Import statements

Driftless creates two files when given the command `import`, one for import commands and one for import blocks. The files also include instructions for a controlled and safe import process, because the process can cause unplanned changes and deletions in the current environment, if the user or Terraform generates its configurations faultily. The first file, `import_configurations.txt`, includes Terraform import blocks, and the second file, `import_commands.txt`, Terraform import commands. Both import blocks and commands can be used to import cloud resources, and they consist of the same parsed cloud provider values, their differences are described in the Section 4 in more detail.

Driftless does not execute any import procedures, and leaves this to the user. Import blocks created by Driftless can be utilized by copying them into Terraform configuration and running Terraform's `plan` command. Terraform then fetches the GCP resources based on the identifier and transforms them into Terraform state objects.

Furthermore, if the plan command is ran using an optional `-generate-config-out` parameter, Terraform attempts to automatically generate configurations of the resources as well, although the result must always be validated by the user. Alternatively, using the created import commands can be executed to create Terraform state entries, however, the user has to create configurations for these manually.

5.3 System design

Driftless is a command line interface tool, implemented using C#, the .NET framework. Its supported functionality is limited to our GCP test environment, which affects the number of cloud resource types that Driftless can detect and interact with, and can be expanded in the future by implementing the GCP SDK further. In this section, we will explore the other systems Driftless depends on and how its users interact with it.

5.3.1 Driftless dependencies

Driftless depends on both the local IaC installations and remote API definitions of any supported target platforms. The IaC framework determines the methods of drift detection or importing existing resources, and in our case, a local Terraform installation is needed for executing Terraform plan and import commands and for utilizing its configuration generation feature. The target platform API defines the rules of interaction with their services, and to interact with our test environment in GCP, Driftless utilizes the GCP software development kit, which allows authenticating with the cloud environment and retrieving and discovering infrastructure data.

5.3.2 Command line arguments

Driftless accepts two commands named after the Terraform functions they are related to; `plan` and `import`. Driftless's `plan` is intended to be a less resource demanding operation, which interacts with just the managed environment. It creates a summary of configuration drift in a single Terraform based GCP project, by executing Terraform plan command for local Terraform configuration files. To gain control over the resources not managed by Terraform, Driftless's `import` command scans the target project environment and compares it to a local Terraform state to detect unmanaged resources, and finally it generates Terraform import statements in the project root folder. Alternatively, to sweep the entire target infrastructure for configuration drift, the import command can be executed using a `-all` parameter, which executes the Driftless plan command within the same execution. Both commands must be provided with a cloud project identifier and a source directory.

5.4 End results

Driftless is expected to discover and list all of the components with configuration drift. This includes modified resources managed by IaC, a firewall and a database in our case, and resources that must not exist according to the IaC definition, the new

computation instances and the new subnetwork. The changed components are either listed in the `plan` summary or in the import statements, and those already aligned with IaC configurations are ignored. These outputs provide the the user should be able to begin the process for mitigating the drift.

Out of 12 resource types in the test environment, there was a single resource type that Driftless could not cover: service networking connections. It differs from the other types by requiring a service identifier string as a parameter, however, using all of the documented formats resulted in unexplainable errors [42]. As a side note, Out of all of the test environment resource types, service networking connections is the only resource type that Terraformer neither supports.

Driftless output covers all the other resource types related to the test infrastructure, consisting of computation instances, a database and its private address, networks and firewall rules. Its plan command displays the modification of the firewall rule and the database backup setting, and how they do not exist in Terraform configurations. This offers enough information for the user to know what has been changed, however the source and the owner of the change remains unknown.

Driftless's import command produces Terraform import commands and import blocks for the new resources: both computation instances and the subnetwork. Debug logs also show Driftless querying all of the other resources, however these exist already and are thus ignored. Executing either the import commands or the blocks creates Terraform state entries for the unmanaged instances and the subnetwork. Matching Terraform configurations are created either by the user, when using import commands, or generated via Terraform, when using import blocks. The success of the process can be measured by running the Terraform plan afterwards. In our case, Terraform does not detect changes in the new imported resources, which confirms the generated states and configurations are identical copies of the target resources. Importantly, this does not indicate correctness or security of the imported resources and their Terraform configurations must be reviewed.

While Driftless's summary of Terraform plan lists the differences between the Terraform state and the target environment, using it for mitigating configuration drift can be challenging. Since the plan output does not categorize whether a change is warranted or necessary for the system to function, the user have to investigate each change extensively before a safe decision can be made. Automating the decision making process, or even giving the user recommended actions, is also difficult, as changes are often context dependent, and can be critical for business and system security. Same consideration must be applied to import statements, even more carefully, as they might not have a purpose to exist in the first place.

6 Evaluation

This section will consider the requirements set in the problem statement chapter and reflect on how Driftless fulfills them. We will evaluate requirements related to accuracy, adaptability and performance.

6.1 Accuracy requirements

The accuracy of Driftless was measured after first creating changes in our environment, and verifying whether they are correctly listed in the Driftless output. The changes created were selected by hand to resemble the real world, where some of the changes must be reverted and some retained.

Based on this, two changes were made to the IaC managed environment: the firewall access rules were changed from the GCP console to allow public traffic to an internal application, which must be reverted as a security risk, and enabled the database backups, a change that cannot be reverted without risking data integrity. Driftless's `plan` output accurately displays the changed configuration fields and the original values for both objects.

Simulating configuration drift by creating unmanaged infrastructure follows the previous logic as well: two new computational instances were added to the environment from the GCP console, one of which should be removed as a security risk, and one that is a legitimate development instance and should remain untouched. Driftless lists these two computational instances in its `import` output, in addition to a new VPC segment where one of the instances resides, covering all of the unmanaged resources in the environment.

Driftless's accuracy is also measured by how identical the IaC configurations and states created when executing Driftless `import` are to their existing counterparts. For the three reported unmanaged resources, the created configurations and states completely match the existing states. However, some edge cases exist, where the new resources are changed before the import process has been finalized. In Terraform, these new unimported changes are displayed to the user, before the now outdated resources overwrite anything.

While Driftless is accurate with our hand selected configuration drift changes, we recognize that this is a limited indicator of accuracy when compared to, for example, programmatically creating unmanaged deployments and changes in the managed environment for each resource type. However, different resource types share the same SDK parent classes and are all supported by Terraform, behaving in a similar way. Lastly, while Terraform provider limitations did not affect the accuracy of the project, Terraform provides support a limited number of resource types. This might affect Driftless's accuracy especially during new releases, when Terraform providers add support to new resource types with delay.

Project IaC configuration directories	Average plan execution time
4	90 s
3	99 s
2	60 s

Table 2: The performance of the Driftless plan command varies non-linearly based on the amount of IaC configuration directories the project has.

6.2 Adaptability requirements

Driftless adaptability can be measured by the extent it supports different target platforms, as mentioned in 3. Driftless’s functionality is mainly based on Terraform’s native configuration drift management capabilities, the plan and import functions, and thus is easily adaptable to other major cloud provider supported by Terraform, providing a good amount of adaptability. Terraform’s import function has varying levels of support for less known providers, but it is extendable if an implementation does not yet exist [43]. Driftless also utilizes application programming interfaces to communicate with target platforms, and while every platform might not offer it in a SDK format, not having any sort of programmatic interface is rare.

Driftless is not fully adaptable to other IaC frameworks, as the success of applying its design to them depends on import capabilities for unmanaged resources. For example, IaC frameworks Pulumi and CloudFormation have a similar import functionalities compared to Terraform, while Ansible requires creating the existing resources manually, and thus not allowing Driftless to utilize Ansible’s own functionalities for the import process.

6.3 Performance requirements

To ensure Driftless’s performance is suitable for integration into environmental maintenance processes, Driftless must not take more than two to three hours to produce its results in large cloud environments. When evaluating the performance, outputs of Driftless import and plan commands must be reviewed separately, as their execution logic differ significantly: `import` interacts with the target platform by using its interface while `plan` starts multiple new Terraform processes. In our environment, on average, Driftless plan finishes in 99 seconds and Driftless import in seven seconds.

There is a significant gap between the two commands, and performance of Terraform’s internal logic needs further investigation, as it currently seems unpredictable. As seen in the table 2, adding a new state directory resulted in slightly decreased execution time, and removing one of the three original directories halved the execution time.

On the other hand, Driftless import command’s performance is much faster at seven seconds, and much of it depends on the API optimization of the target platform, in our case, GCP. Although this was not tested, `import`’s performance may vary for each target platform, and some may even shift the responsibility of optimization to Driftless.

Performance was only measured in a small environment of ten cloud resources, consisting of a database, networks, firewall rules, and computational instances. When extrapolating the measurements into a larger environment of a thousand resources, Driftless plan takes 2.75 hours, close to our performance target upper limit of three hours, and the execution of Driftless import is completed in just under 12 minutes, making it suitable for our use case. However, when cloud environments contain hundreds of thousands of resources, which can occur according to [4], Driftless's performance fails to meet our criteria: in such environments, Driftless takes 275 hours to plan 20 hours to import.

Predicting Driftless performance in large environments based on our small sample set is not very accurate, but it offers some sense of the challenges Driftless would face in larger environments. Especially `import` is impacted by the infrastructure size, as its design demands fetching data for all resources, and the real results could be worse than our predictions. It is also important to note that the current version of Driftless does not support parallelization, which would be necessary in larger environments, and would largely reduce its execution times.

7 Related work

This chapter reviews configuration drift mitigation capabilities of similar tools and discusses related research.

7.1 Terraformer

Terraformer is an actively maintained open source tool that works in reverse order of Terraform, generating Terraform configuration and state files based on existing cloud infrastructures and deployed resources. Its creator is listed as Waze SRE, and while the company was acquired by Google, Terraformer itself has claimed not to be affiliated with Google[44][45].

Terraformer supports 17 cloud platforms and various other infrastructure, including network, monitoring and orchestration solutions [46]. Terraformer utilizes Terraform provider plugins to interact with cloud platform APIs; however, its support for cloud platform services is limited. For example, Terraformer is unable to recognize a service networking connection object, as it does not support the GCP service “google_service_networking_connection” [45]. Additionally, Terraformer sometimes assigns resource attributes and configurations incorrectly, requiring the user to fix the generated Terraform configuration and state files before they are usable [47]. The quality of Terraformer’s result depends on the maturity and completeness of each platform’s Terraform providers.

7.2 Cloud Concierge

Cloud Concierge is an open source tool created by a company “dragondrop.cloud”. It focuses on “cloud codification”, meaning identifying cloud resources and configuration drift, and generating corresponding Terraform code alongside Terraform import statements. Based on its source code, central to its functionality is the use of Terraformer, which allows Cloud Concierge to generate Terraform configurations and states. Cloud Concierge compares the generated Terraform configuration files and import blocks against an existing state files to detect configuration drift and identify resources that are not currently managed by Terraform. Cloud Concierge utilizes an NLP algorithm as well, although it is unclear whether this is related to detection or resource importing.

Cloud Concierge flags accounts making changes outside the Terraform workflow, ensuring that all future infrastructure modifications are tracked and managed centrally. The tool also performs static security scans and cost estimations and compiles the final output into a pull request, making the import process for the new Terraform resources more seamless. This tool currently supports only AWS, Azure and GCP. [48]

7.3 Terragrunt

Terragrunt is an open source Terraform orchestration tool and a wrapper, maintained by Gruntwork.io. Unlike the other tools mentioned in this section, Terragrunt does

not aim to prevent configuration drift, nor does it generate IaC code based on existing cloud infrastructures. Instead, it eases managing and scaling Terraform development, by enforcing DRY (Don't Repeat Yourself) principles, simplifying the development process and making the Terraform configurations immutable [49]. Terragrunt also streamlines the managing of complex remote states by file locking and enforcing DRY to backend configurations, and acts as a Terraform wrapper, providing additional functionalities, such as running the command “terraform plan” for multiple file directories, provider caching, and automatic initialization. [50]

7.4 GCP Deployment Manager and Config Connector

Google provides native ways to manage and automate GCP deployments and for exporting its infrastructure configurations as Terraform code. These can be done using Deployment Manager and Config Connector.

Deployment Manager allows users to define infrastructure and resources using declarative YAML (or Python/Jinja) templates. Deployment Manager simplifies resource orchestration, allowing users can automate deployment and management of GCP resources, such as virtual machines, networks, and storage in a repeatable and scalable way. Deployment Manager is specifically designed for Google Cloud and manages its configuration state file internally in the GCP platform, while Terraform requires a plan and process for managing its state file. [51]

Config Connector is a beta stage CLI tool for resource bulk importing and exporting module for GCP, and in addition, the bulk exported Terraform configurations can be imported into Terraform state afterwards. It allows the exported configurations to be in Terraform's format, but this does not work on the Windows operating system and has limited support for number of GCP resources. Even if Terraform GCP provider supported defining a resource configuration, this tool might not, and it must be noted that this tool does not have extensive popularity yet, possibly because its still in an early development phase. [52]

7.5 Comparing Driftless to related tools

We compare three of the previously mentioned 3rd party solutions, Terraformer, Cloud Concierge, and Terragrunt which can be utilized similarly to Driftless, however, only the last one is intended for detecting configuration drift or for change management. Terraformer is strictly made for converting cloud resources into Terraform configurations and states, equating to the import command in Terraform, as it does not compare the cloud environment to the existing Terraform configurations, and is thus unaware of any changes.

We tested importing the unmanaged resources using Terraformer, inputting manually the information of what was changed, and its outcome matched almost identically Terraform import's, the only mistake being a broken object reference that was quickly fixed. Importing resources using Terraformer is challenging in larger cloud environments, as much preparation has to be used to distinguishing the unmanaged resources,

e.g., tagging resources created by Terraform. Terraform's ability to generate configurations while importing is still an experimental feature, and Terraformer might be a more stable alternative. Also, this project can be used with Terraformer to provide the list of unmanaged resources that are then ran through Terraformer instead of terraform import.

Cloud Concierge discovers configuration drift in managed environment and looks for any infrastructure that exists outside of Terraform definition. It supports isolated Terraform configurations and produces a git pull request as an output, which is comparable to Driftless output when running the "import" command with "-all" flag. When executed against our test environment, it detects the drift in managed and unmanaged resources accurately, and enriches its report with additional details, such as security and cost impact of each drifted resource utilizing open source tools tfsec and Infracost. Similar to Driftless, it creates Terraform import blocks for the unmanaged computation instances and private network, but also goes further and creates Terraform configuration files for these resources. Cloud Concierge also creates import blocks for the drifted managed infrastructure, which seem to be redundant, as Terraform ignores import blocks for already existing resources. To conclude, in our test environment, Driftless discovers configuration drift as adeptly as Cloud Concierge, but its provided additional security and cost information increase the usefulness of the produced output, as this makes the mitigation decision making easier for the user. However, Driftless is a fully Terraform native implementation, while Cloud Concierge benefits from utilizing other open source tools.

Lastly, Terragrunt is capable of detecting the changes in the database and firewall instances, by executing `terragrunt stack plan`. The output includes the same details as Driftless, as both tools simply execute Terraform plan multiple times. Unlike Driftless, Terragrunt does not detect the execution directories based on Terraform's `.tfstate` files, and searches instead its own `.hcl` configurations, which have to be set up for each state directory.

Compared to the existing tools, Driftless is the only approach that manages Terraform's configuration drift using its native functionalities, making it more lightweight, as complex infrastructure or change scanning logic is not needed. This eliminates any discrepancies that can appear between Terraform state and external drift detection tool interpretations. In addition, this approach is easy to expand across different IaC frameworks, as it does not require implementing custom detection logic for managed and unmanaged parts of the infrastructure, as long as the target IaC platform includes these methods.

7.6 Infrastructure as Code research

Infrastructure as code is a broadly covered topic and configuration drift is also covered in the publications. Two examples of this are the books "Infrastructure as Code" by Kief Morris and "Infrastructure as Code, Patterns and Practices" by Rosemary Wang. [4] [35]

Morris introduces multiple patterns that cause or manage configuration drift. As an example, the "Apply on change" anti-pattern is about creating drift by making

individual changes to individual systems, either by a human or a script. This is not a sustainable way for managing larger groups of resources: some resources may have longer update intervals, causing them to become unlike the other resources belonging to the same group. Changing resources through IaC does not completely resolve this issue, as its own code can also be carelessly changed when done without proper software development practices. Understanding anti-patterns helps building change models for different change scenarios and acts as a preventive measure.

In contrast, the “continuous change synchronization” management pattern decreases the motivation to create ad hoc changes or using alternative change channels. It enforces a single channel for changes and frequently deploys the newest environment state on all managed systems, overwriting any configuration drift. Having proper monitoring in place is important when adapting this pattern, so that any emerging issues caused by overwriting states can be caught.

The patterns introduced by Morris aim to prevent configuration drift, but they do not introduce methods for detecting or mitigating existing configuration drift. Wang suggests detecting configuration drift by running frequent regression tests for each project environment, as discrepancies between test results can reveal configuration drift. It should be noted that regression testing managed infrastructure alone would not reveal configuration drift caused by unmanaged infrastructure: this would require separate infrastructure composition analysis. Driftless could also be used similarly by comparing its outputs for different project environments.

7.7 CSBAuditor

Both Morris and Wang focus on preventing configuration drift from accumulating in IaC systems, but do not discuss monitoring or mitigation methods. However, Torkura et al. introduces a cloud security posture management (CSPM) solution similar to Driftless, CSBAuditor. It monitors cloud resources in the entire cloud environment, by comparing each resources actual state to the expected state defined as the secure state baseline. Driftless depends on state baselines maintained by the IaC frameworks, but it is unclear whether CSBAuditor follows the same approach or creates its own configuration baselines. In addition to monitoring existing states, CSBAuditor also detects creation of new resources without proper authorization. [53]

Because of cloud environments actively changing, Torkura et al. decides to monitor the environment every two minutes, which is much more frequent compared to Driftless, and results in configuration drift issues being fixed more quickly from their creation. Another reason for frequent execution rate is CSBAuditor’s ability to actively enforce the state baseline, behaving similarly to the “continuous change synchronization” scheme mentioned earlier, although it can also instead notify administrators of found configuration drift. However, unlike Driftless, it does not try to create state baselines for unmanaged resources, leaving this to the user.

As a CSPM tool, CSBAuditor also analyzes configuration drift for threats by using its own common vulnerability scoring system (CVSS) based risk model, aiming to find suspicious changes that indicate of security compromise. While the security impact of configuration drift is not a well researched topic, configuration drift creates blind

spots in security monitoring and makes the infrastructure less stable [\[31\]](#) [\[4\]](#) [\[35\]](#).

8 Discussion

This thesis examines mitigating configuration drift in managed and unmanaged IaC environments, approaching the problem using native Terraform methods. While the produced tool gives promising results in our test cases, there are areas that have to be matured further to apply this design in real-life.

8.1 Multiple project support

This project was developed in a cloud environment containing a single Terraform project, which is not the norm in real-world, as cloud environments often contain resources belonging to multiple different IaC projects. Our current approach results in configuration drift findings, which are not classified for any certain IaC project, and while this does not harm the infrastructure, users have to determine correct IaC projects for new resources themselves, requiring extra effort. Preventing this is challenging, unless the unmanaged resources contain clear project identifiers, as it first requires identifying respective IaC projects for all unmanaged resources, which is part of the problem we aim to solve in the first place.

As there is not yet a method to classify cloud resources to corresponding IaC projects, searching the entire cloud environment for configuration drift is not practical in larger environments. An alternative approach could be to limit our scope to an infrastructure of a single development team instead of the whole infrastructure, increasing the likelihood that its results will be handled by the relevant infrastructure management team, making the configuration drift mitigation process easier.

8.2 Further integrations

More comprehensive integration with IaC frameworks allows detecting discrepancies between the local and remote IaC states automatically before execution. This allows always working with the most current version of the infrastructure definition and alerting on occurring drift in real-time. In addition, inspired by cloud-concierge, integrating with third-party tools that provide extra details on the detected drift helps prioritizing and addressing drift based on each change's cost and security impact. Prospective tools, in addition to tfsec and Infracost used by cloud-concierge, are Checkov, Regula or native CSP tools, which give insight to the security and regulatory compliance of IaC configurations.

8.3 Orchestration platform support

This thesis deals with configuration drift within the main environment, such as different cloud services and configurations. However, the environment may also include separate orchestration platforms, such as Kubernetes, which define and manage additional infrastructure resources independently from the primary IaC configurations. Detecting this drift requires taking the orchestration configurations into account, as a separate, nested, infrastructure. In Terraform, it is possible to use additional providers to build

Terraform configurations for the infrastructure differing from the main one. Some orchestration platforms may include their own drift mitigation tool sets, but accessing them might be challenging from outside of the target environment.

References

- [1] Shane Greenstein. The basic economics of internet infrastructure. *Journal of Economic Perspectives*, 34, 2020.
- [2] Jayachander Surbiryala and Chunming Rong. Cloud computing: History and overview. 2019.
- [3] S Bharath Bhushan, Pradeep Reddy, Dhenesh V Subramanian, and XZ Gao. Systematic survey on evolution of cloud architectures. *International Journal of Autonomous and Adaptive Communications Systems*, 11, 2018.
- [4] Kief Morris. *Infrastructure as Code: Dynamic Systems for the Cloud Age*. 2021.
- [5] Fatemeh Khoda Parast, Chandni Sindhav, Seema Nikam, Hadiseh Izadi Yekta, Kenneth B. Kent, and Saqib Hakak. Cloud computing security: A survey of service-based models. *Computers & Security*, 114, 2022.
- [6] Praveen Borra. Comparison and analysis of leading cloud service providers (AWS, Azure and GCP). *International Journal of Advanced Research in Engineering and Technology (IJARET)*, 15, 2024.
- [7] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, Dawn Leaf, et al. NIST cloud computing reference architecture. *NIST special publication*, 500, 2011.
- [8] Peter Mell. The NIST definition of cloud computing. *NIST Special Publication*, 2011.
- [9] Joel Gibson, Robin Rondeau, Darren Eveleigh, and Qing Tan. Benefits and challenges of three cloud computing service models. 2012.
- [10] Vishal Kaushik, Prajwal Bhardwaj, and Kaustubh Lohani. Do the NIST definitions of cloud service models need an update? 2022.
- [11] Paul McDonald. Introducing Google App Engine + our new blog, 2008. <https://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html>.
- [12] Janakiram MSV. A look back at ten years of Microsoft Azure, 2020. <https://www.forbes.com/sites/janakirammsv/2020/02/03/a-look-back-at-ten-years-of-microsoft-azure/>.
- [13] Google. App Engine 1.6.0 out of preview release, 2011. <http://googleappengine.blogspot.com/2011/11/app-engine-160-out-of-preview-release.html>.
- [14] Dean Leffingwell and Don Widrig. *Managing software requirements: a unified approach*. 2000.

- [15] Khaled El Emam, Dirk Holtje, and Nazim H Madhavji. Causal analysis of the requirements change process for a large system. 1997.
- [16] HashiCorp. Built-in functions, 2025. <https://developer.hashicorp.com/terraform/language/functions>.
- [17] Ernesto Marquez. AWS CloudFormation vs. Terraform: How to choose?, 2024. <https://www.techtarget.com/searchcloudcomputing/tip/AWS-CloudFormation-vs-Terraform-How-to-choose>.
- [18] Amazon. Intrinsic function reference, 2025. <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference.html>.
- [19] Microsoft. ARM template documentation, 2025. <https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/>.
- [20] Mumian, Tfitzmac, LiSeda, Naveedkharadi, Tejas Nagchandi, Alex Frankel, UcheNkadiCode, Daphnemamsft, and Davidsmatlak. What is Bicep?, 2025. <https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview>.
- [21] Mumian, LiSeda, Polatengin, Danay1999, and Tfitzmac. Bicep functions overview, 2025. <https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/bicep-functions>.
- [22] Niko Kalliomaa. Choosing the right IaC tool for building reusable cloud infrastructure. 2024.
- [23] Yevgeniy Brikman. *Terraform: Up and Running: Writing Infrastructure as Code*. 2022.
- [24] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 1980.
- [25] Michael W Godfrey and Daniel M German. On the evolution of Lehman's Laws. *Journal of Software: Evolution and Process*, 2014.
- [26] Mamdouh Alenezi and Khaled Almustafa. Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 2015.
- [27] Gregorio Robles, Juan Jose Amor, Jesus M Gonzalez-Barahona, and Israel Herraiz. Evolution and growth in large libre software projects. 2005.
- [28] Qiang Tu et al. Evolution in open source software: A case study. 2000.
- [29] Sharon McGee and Des Greer. Software requirements change taxonomy: Evaluation by case study. 2011.

- [30] Rajib Mall. *Fundamentals of software engineering*. 2018.
- [31] Claus Pahl, Pooyan Jamshidi, and Danny Weyns. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *Journal of Software: Evolution and Process*, 29, 2017.
- [32] Shalinka Jayatilleke and Richard Lai. A systematic review of requirements change management. *Information and Software Technology*, 2018.
- [33] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. 2008.
- [34] Philipp Hohl, Jil Klünder, Arie van Bennekum, Ryan Lockard, James Gifford, Jürgen Münch, Michael Stupperich, and Kurt Schneider. Back to the future: origins and directions of the “agile manifesto”—views of the originators. *Journal of Software Engineering Research and Development*, 2018.
- [35] Rosemary Wang. *Infrastructure as Code, Patterns and Practices: With Examples in Python and Terraform*. 2022.
- [36] Ravikanth Chaganti and Shay Levy. *Windows PowerShell Desired State Configuration Revealed*. 2014.
- [37] Blessing Bwalya and Aaron Zimba. An SDN approach to mitigating network management challenges in traditional networks. 2021.
- [38] HashiCorp. Terraform repository, 2025. https://github.com/hashicorp/terraform-provider-google/blob/70262675cbe9c1c599c4131f8869d0936e4d88e6/google/provider/provider_dcl_resources.go.
- [39] Samuel Baena Hayas. Importing Terraform resources the safe way, 2022. https://www.youtube.com/watch?v=VNBi-HhVX_Q.
- [40] HashiCorp. Terraform import block documentation, 2024. <https://developer.hashicorp.com/terraform/language/import>.
- [41] Google Cloud Platform. GCP repository, 2025. <https://github.com/GoogleCloudPlatform/terraformer/tree/a5b4b69b1cd36942ed85c1c6f8ba81e16b8b8a2b/providers/gcp>.
- [42] Google. GCP repository, 2025. <https://googleapis.dev/dotnet/Google.Apis.ServiceNetworking.v1/latest/api/Google.Apis.ServiceNetworking.v1.ServicesResource.ConnectionsResource.html>.
- [43] HashiCorp. HashiCorp documentation, 2025. <https://developer.hashicorp.com/terraform/plugin/sdkv2/resources/import>.

- [44] Uri Levine. Was selling Waze to Google a good decision? founder of Waze reflects on the deal, 2023. <https://www.forbes.com/sites/urilevine/2023/06/09/was-selling-waze-to-google-a-good-decision-founder-of-waze-reflects-on-the-deal/>.
- [45] Waze SRE. Use with GCP, 2023. <https://github.com/GoogleCloudPlatform/terraformer/blob/master/docs/gcp.md>.
- [46] Waze SRE. Terraformer GitHub repository, 2024. <https://github.com/GoogleCloudPlatform/terraformer>.
- [47] Ned in the cloud. Investigating Terraformer, 2021. <https://www.youtube.com/watch?v=GpjCF4yZU9A>.
- [48] Dragondrop.cloud. Cloud Concierge repository, 2023. <https://github.com/dragondrop-cloud/cloud-concierge>.
- [49] Gruntworks.io. Keep your Terraform code dry, 2024. <https://terragrunt.gruntwork.io/docs/features/keep-your-terraform-code-dry/>.
- [50] Gruntworks.io. Terragrunt documentation, 2024. <https://terragrunt.gruntwork.io/docs/>.
- [51] Google. Google Cloud deployment manager documentation, 2024. <https://cloud.google.com/deployment-manager/docs>.
- [52] Google Cloud. Export your Google Cloud resources to Terraform format, 2024. <https://cloud.google.com/docs/terraform/resource-management/export>.
- [53] Kennedy A Torkura, Muhammad IH Sukmana, Feng Cheng, and Christoph Meinel. Continuous auditing and threat detection in multi-cloud infrastructure. *Computers & Security*, 102, 2021.