Aalto-yliopisto
Aalto-universitetet
Aalto University

Master’s Programme in Computer, Communication and Information Sciences

Infrastructure as Code for Managed Service Providers: A Case Study

Olli Paloviita

Master’s Thesis
2022

Aalto-yliopisto
Aalto-universitetet
Aalto University

Author Olli Paloviita	
Title of thesis Infrastructure as Code for Managed Service Providers: A Case Study	
Programme Master's Programme in Computer, Communication and Information Sciences	
Major Computer Science	**Code** SCI3042
Thesis supervisor Prof. Mario Di Francesco	
Thesis advisor Jouni Nuutinen, M. Eng	
Collaborative partner Enfo Oyj	
Date 08.07.2022 **Number of pages** 61 + 1	**Language** English

Abstract

This thesis analyses how Managed Service Providers (MSPs) can leverage Infrastructure as Code (IaC) tools. For years now, MSPs have had challenges in managing unique customer solutions. Many MSPs have shifted into offering solutions in cloud environments which means they can potentially use Terraform and other IaC tools to modularize their unique customer solutions.

This thesis is a case study, which develops proposals to improve the collaborating MSP's current IaC practices. The collaborating MSP of this thesis uses Terraform to deliver unique cloud solutions to individual customers. It aims to adopt reusable Terraform modules as it would reduce maintenance and development costs and it would help with creating new services. However, adopting reusable Terraform modules is challenging given the collaborating MPS's requirements related to Intellectual Property Rights (IPR).

This thesis develops and compares three new IaC workflows of which the collaborating MSP can adopt the most suitable one. The developed workflows allow reusing Terraform modules in the collaborating MSP's customers' infrastructures while giving individual customers different levels of access to the Terraform modules' source code. Each new workflow is based on an iterative process including stakeholders' feedback and multiple reference implementations.

Since the developed workflows are entirely Git based, they enable reusing a wide range of versioned files, not just Terraform code, with multiple customers. This means that the outcome of this case study is also applicable to other use cases and organisations.

Keywords Infrastructure as Code, managed services, case study, modularization, version control

Aalto-yliopisto
Aalto-universitetet
Aalto University

Tekijä Olli Paloviita

Työn nimi Infrastruktuuri koodina jatkuvien palveluiden tarjoajille: Tapaustutkimus

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Computer Science **Koodi** SCI3042

Vastuuopettaja/valvoja Prof. Mario Di Francesco

Työn ohjaaja Jouni Nuutinen, M. Eng

Yhteistyötaho Enfo Oyj

Päivämäärä 08.07.2022 **Sivumäärä** 61 + 1 **Kieli** englanti

Tiivistelmä

Tämä diplomityö analysoi, kuinka IT-ulkoistuspalveluiden ja jatkuvien palveluiden tarjoajat voivat hyödyntää infrastruktuuri koodina -työkaluja. Jatkuvien palveluiden tarjoajilla on jo vuosia ollut haasteita ainutlaatuisisten asiakasprojektien hallinnan kanssa. Monet jatkuvien palveluiden tarjoajat ovat siirtyneet tarjoamaan ratkaisuja pilviympäristöissä, mikä mahdollistaa infrastruktuuri koodina -työkalujen kuten Terraformin käytön ainutlaatuisten asiakasratkaisujen modularisointiin.

Tämä diplomityö on tapaustutkimus, joka ehdottaa parannuksia yhteistyössä toimivalle jatkuvien palveluiden tarjoajan infrastruktuuri koodina -työkalujen käyttöön. Jatkuvien palvelujen tarjoaja käyttää Terraformia ainutlaatuisten pilviratkaisujen toimittamiseen yksittäisille asiakkaille. Se haluaa ottaa käyttöön uudelleenkäytettäviä Terraform-moduuleja, koska se vähentäisi ylläpito- ja kehityskustannuksia ja auttaisi uusien palvelujen kehittämisessä. Uudelleenkäytettävien Terraform-moduulien käyttöönotto on kuitenkin haastavaa heidän immateriaalioikeuksiin liittyvien vaatimusten vuoksi.

Tässä opinnäytetyössä ehdotetaan ja verrataan kolmea uutta työnkulkua infrastruktuuri koodina -kehitykseen, joista yhteistyössä toimiva jatkuvien palveluiden tarjoaja voi ottaa käyttöön sopivimman. Ehdotetut työnkulut mahdollistavat Terraform-moduulien uudelleenkäytön asiakkaiden infrastruktuureissa antaen yksittäisille asiakkaille eri määrän pääsyä Terraform-moduulien lähdekoodiin. Työnkulut ovat tulos iteratiivisesta prosessista, joka sisälsi useita referenssitoteutuksia ja palautekierroksia sidosryhmien kanssa.

Koska ehdotetut työnkulut ovat kokonaan Git-pohjaisia, ne mahdollistavat Terraform-kooditiedostojen lisäksi myös muiden tyyppisten versioitujen tiedostojen jakamisen asiakkaille. Tämä tarkoittaa, että esiteltyjä työnkulkuja voidaan soveltaa muihinkin käyttötarkoituksiin ja muissa organisaatioissa kuin mitä tässä diplomityössä esitetään.

Avainsanat Infrastruktuuri koodina, jatkuvat palvelut, tapaustutkimus, modularisaatio, versionhallinta

Contents

Preface

I wish to thank Enfo Oyj for granting me the valuable opportunity to write this thesis. I am especially thankful to Jouni Nuutinen, Marko Ikäheimo, Jani Karhunen and Mika Hartikainen from Enfo Oyj for their guidance and insights which helped me immensely with my writing. I am also grateful for my supervisor, Professor Mario Di Francesco, for his input throughout the research process.

Finally, I would like to give special thanks to my girlfriend, friends, and family for encouraging me in the past six months.

Espoo, July 8, 2022

Olli Paloviita

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CSP	cloud service provider
CI	continuous integration
CLI	command-line interface
DRY	Don't repeat yourself
DSL	Domain Specific Language
GCP	Google Cloud Platform
MSP	managed service provider
IaaS	Infrastructure as a Service
IaC	Infrastructure as code
ICT	Information and Communication Technology
IPR	Intellectual Property Rights
IT	Information Technology
JSON	JavaScript Object Notation
OS	operating system
SaaS	Software as a Service
SLA	service level agreement
SSO	Single Sign-on
VCS	Version Control System
VM	virtual machine
VPN	virtual private network

1 Introduction

Traditionally, using managed services means transferring the responsibility of day-to-day IT management to an external Managed Service Provider (MSP) in order to achieve improved operational and cost effectiveness [1]. However, the MSP business model has evolved in recent years and competition has forced today's MSPs to shift into offering unique capabilities with new value propositions [2]. Nowadays many MSPs are also cloud-enabled and offer a wide range of new solutions built on top of different cloud services [2].

For years now, MSPs have had challenges in packaging their solutions away from one-of-a-kind projects [1]. Compared to the unique projects, the adoption of shared environments, tools, and processes between customers allow MSPs to be more effective at delivering their services [2]. MSPs working in multi-cloud environments can potentially use Infrastructure as Code (IaC) tools such as Terraform for modularizing their solutions for all the main Cloud Service Providers (CSPs).

This thesis is a case study[1], the aim of which is to make proposals to improve the current IaC practices of the collaborating MSP. The collaborating MSP has used Terraform to deliver unique cloud solutions to individual customers. Recently, they have identified an opportunity in adopting modular and reusable Terraform modules which would work with multiple customers. Reusing Terraform modules has several advantages compared to fully unique customer infrastructures. For example, the total amount of managed code would be reduced. Reusing Terraform modules would also cut down maintenance work, as the same technical fixes could be applied to multiple customers. It could also help with delivering managed services faster to new customers, as reusable parts of the solutions have already been developed and tested.

So far, the collaborating MSP has not introduced reusable Terraform modules in its customer solutions. This is partly due to the fact that reusable Terraform modules would require developing a new mechanism to distribute these modules across several customers and their isolated version control systems (VCSs). Another reason is that the questions related to the Intellectual Property Rights (IPR) of these reusable modules must be identified and solved. The IPR clauses defined in a contract between MSP and a customer state which parts of the MSP provided solution the customer has rights to. Different IPR clauses have distinct requirements for the workflow in which

[1] A case study enables a researcher to examine a phenomenon in relation to its actual context in real world. By analyzing one individual, group or event thoroughly, a researcher explores the causes of underlying principles [32].

these reusable modules are developed and shared with the customer. These impacts should be carefully analyzed before certain IPR approach can be chosen.

This case study aims to deepen the understanding on how different IPR approaches affect Terraform development. This information will be utilized by the collaborating MSP when deciding which IPR approach is reasonable in the customer contracts, and how to develop reusable Terraform module code. This thesis contributes to the existing literature by addressing how MSPs could modularize their solutions with IaC tools. To the best of my knowledge, this issue has not been studied very widely so far.

The outcome of this thesis is to propose and compare three new IaC development workflows, which could be used to develop reusable Terraform modules for the collaborating MSP. These new workflows allow reusing Terraform modules with multiple customers, while giving each customer a certain level of access to the Terraform source code. These access levels requirements are based on distinct IPR clauses. The proposed workflows present different solutions how to distribute reusable modules to isolated customer infrastructures in a controlled and secure manner, while allowing centralized development and module versioning. This is not possible with the collaborating MSP's current IaC development workflow. The developed workflows are useful in broader context, as they are applicable to other organizations who want to reuse their IaC solutions with multiple customers.

To evaluate potential benefits of adopting the proposed workflows, the currently used Terraform modules in the collaborating MSP's customer project were analyzed. This analysis was conducted by surveying the module developers and by programmatically analyzing the source code. This analysis helps to assess the potential reusability of the current Terraform modules in the customer infrastructures.

The thesis is structured as follows. Section 2 reviews MSP modularization literature and presents the current IaC tool landscape. Section 3 focuses on Terraform and gathers good practices applicable in the MSP business setting from the literature, official Terraform documentation and blogs. Section 4 explains the benefits and challenges of reusable Terraform modules for MSPs. Section 5 contains the customer infrastructure analysis. Section 6 describes the currently used IaC development workflow. Section 7 provides the requirements for Workflows 1-3. Section 8 proposes and compares Workflows 1-3. Concluding remarks are provided in section 9.

2 MSP modularization and IaC tool landscape

Currently the collaborating MSP of this case study has used IaC tool Terraform to modularize and manage the customer infrastructures in the cloud. The this section and the following section 3 give background information about modularization, IaC and Terraform, which are later used to assess the currently used IaC development workflow and develop Workflows 1-3 which were described in Section 1.

First this section describes in general, how MSP business has transformed towards modularization and cloud platforms. Then it presents how cloud infrastructure can be managed, and how it relates to IaC tools. Finally, the current IaC tool landscape is introduced, and different tools are compared.

2.1 Managed Service Providers and modularization

Managed services are the practice of outsourcing business' IT management and maintenance. Managed services are provided by Managed Service Providers (MSPs). Unlike in traditional break-fix approach, where professionals are approached only when IT problems arise, MSPs take a long term, proactive approach in managing and maintaining the customer's IT infrastructure. For its customers, MSPs can reduce IT expenses and make the costs more predictable, while enabling customer to focus on other areas of their business [3].

Traditionally MSPs offerings were highly customized or even one-of-a-kind projects, but already in 2008 there was research on the benefits of reusable components in providing managed services by researchers Nicholas Leon and Andrew Davies [1]. At that time, they were surprised how labor intensive, and asset-based managed services were compared to other information and communication technology (ICT) sectors.

They proposed a six-step process to transform managed service capabilities into more modular, standardized, and easy-to-configure services that would allow reaching new delivery models and customer sets. They also analyzed three MSPs attempting to move towards more standardized services and found out that the transformation is challenging. Maybe surprisingly, these challenges were not technical in nature, but mostly cultural and organizational issues related to the transformation. To mitigate these issues, they proposed focusing on the customer-experience rather than technical competency when discovering and developing these more modular services.

They also argued that regardless of the service model, the MSPs need to standardize their offerings at least partially in order to stay competitive, even if the services are personalized for each customer. They also proposed tracking the value and proportion of sales that use the standardized modules as one of the key metrics of the company [1].

In Technology-as-a-Service Playbook (2016), it was clear that a transformation had happened in MSP business, and many of them moved to cloud and away from the low-margin outsourcing business [2]. With the new tools, many MSPs could get better economies of scale with multiple customers served with the same set of tools.

Also, the customers and their needs had changed over time: they did not see MSPs just as an outsourcing opportunity to save money, but also as a partner to help with monitoring and optimizing their IT infrastructure. Many customers also used MSPs to help with a transformation into some new technology, or used them for managing individual products, not the whole IT infrastructure. When IT managers were surveyed, more than half expected to use more than one MSPs within next two years [2].

Based on its benchmarking, the book suggests investing in automation early, as the most profitable MSPs have used off-the shelf tools and platforms to automate parts of their operation. It also suggests starting to identify common basic building blocks, that could be combined into services in multiple ways [2].

2.2 Infrastructure Platforms

A while ago, software and IT companies typically managed their own hardware, such as servers, cooling, and power systems by themselves. This work included installing software to run on these systems, and it was often done by manually executing commands on the server [4]. During that time, infrastructure was hardware.

But nowadays infrastructure can mean several things. Virtualization has enabled decoupling the hardware from the systems running on them, and later cloud added Application Programming Interfaces (APIs) to manage those virtualized resources. This type of API or graphical user interface based, on demand resource management is called Infrastructure as a Service (IaaS).

This has led into many kinds of infrastructure platforms, some providing bare-metal non-virtualized hardware like in the old days, and some provide only virtualized resources and infrastructure. Regardless of the platform type, infrastructure platforms provide resources for computing, storage, and networking.

On high level, cloud-based infrastructure platforms can be divided into three groups: public IaaS cloud services, private IaaS cloud products and bare metal cloud tools.

Public IaaS services are provided by Cloud Service Providers (CSPs) such as AWS, Digital Ocean, and Azure. Typically, public IaaS is just one of many services that they provide. On public IaaS, every customer shares the underlying CSPs hardware infrastructure.

Private IaaS products, such as CloudStack, Openstack and VMware vCloud, also use similar virtualization techniques and APIs as the public IaaS services, but this time the hardware is only meant for selected or even one customer.

Bare metal cloud tools, such as Cobbler, FAI, and Foreman, allows the use of hardware through the API but without virtualization. This means that these services do not suffer from the overhead that comes with the use of virtualization.

Many organizations end up hosting their resources in multiple infrastructure platforms. *Hybrid cloud* means hosting a system across private and public cloud services. *Cloud Agnostic* means building the system so that it allows running it on multiple public cloud platforms. *Polycloud* means running different systems on more than one public cloud platform. The reason for these more complicated cloud infrastructure architectures is cost or feature optimizations, legal requirements or avoiding lock-in to one vendor [5].

2.3 Managing cloud infrastructure

Managing cloud infrastructure within these infrastructure platforms is not a simple task. There are many types of technologies to choose from, and many times it is challenging to know where to even start. This section goes through types of technologies and common challenges related to cloud infrastructure management.

The simplest form of infrastructure management is to do everything manually. With cloud resources, this means creating and configuring resources by hand. But as management workload and the number of required resources increases, this manual process inevitably leads into mistakes or unwanted differences between resources. This problem is called *configuration drift*: there is a difference between the wanted and the actual configuration [4].

This drift can cause a lot of problems after a while, so a lot of tools have been created to automate the process of managing infrastructure. By fully automating the configuration modification process, the drift should not occur.

One approach is to use *ad hoc scripts* to automate the management steps. These scripts can be written in a scripting language, such as Python. The challenge is that these scripting languages are general purpose tools, so the same script can be written in many ways. With complex management tasks, purpose-build management tools usually lead into more maintainable tooling [4].

Configuration management tools such as Ansible are purpose-build to install and manage software on existing servers. Usually, they are also operated through scripts/configurations, but unlike ad hoc scripts, they offer features that make infrastructure management easier. These features can include solutions in distribution (how to apply changes to many servers at once), idempotence (applying same script again should not change the server state) and coding conventions (structure is consistent between users, which makes maintaining easier).

Another option for preventing configuration drift on server is to use a *server templating tool*, such as Docker or Vagrant. Unlike configuration management tools, which execute the wanted commands on each server, server templating tools capture the wanted server state into an *image*. This image which represents a state of the whole operating system (OS) can be applied to a server, a virtual machine (VM) or a container directly, depending on the image type and tool. Server templating can be used as a part of *immutable infrastructure* process, where no change is allowed on the server, unless a new image version is deployed [4]. Immutable infrastructure is one strategy to combat configuration drift.

Large amounts of VMs and containers can be managed through *orchestration tools*, such as Kubernetes. They can handle container/VM deployments, updates, monitoring, scaling, load balancing as well as communication. This is a large topic that is beyond the scope of this thesis.

Configuration management, server templating tools and orchestration tools define the code running on servers. *Provisioning tools*, such as Terraform handle the creation of these servers (and cloud resources) themselves [4]. Provisioning tools require a dynamic infrastructure platform that allows provisioning and changing of these resources through an interface [5].

Based on “Terraform: Up & Running” -book, ad hoc scripts, configuration management tools, server templating tools, orchestraion tools and provisioning tools are the main categories of Infrastructure as Code tools.

2.4 Infrastructure as Code

Infrastructure as Code (IaC) means the idea of writing and executing code to manage infrastructure. This code defines the infrastructure, and it can be used to deploy, update, and destroy infrastructure. IaC is not some specific tool, but a broader idea that can be used to make managing infrastructure easier.

Code is not usually the first way for people to learn to manage infrastructure. For example, if you go through CSPs own tutorials on how to use their services, they typically show how to start using their services through their web-based user interface, or through a service provider’s own command-line interface (CLI). An alternative approach is to define the cloud infrastructure in a text format (also known as *infrastructure code*), and then to make these resources defined in the text into reality by using an Infrastructure as Code (IaC) tool. In practice, Web-based approach, CLI approach and IaC approach can lead into same results. However, IaC approach is typically the most preferable option, since it has several good features compared to other options.

The core benefit of IaC tools is in the infrastructure code: when you define the whole cloud infrastructure “as code”, you enable better reusability: after the infrastructure definition has been done, you can create, replicate, repair and rebuild the infrastructure quickly. You also have better consistency: infrastructure build from code is build the same way every time, which makes the system behavior more predictable. You also gain better transparency: everyone can look at the code, and see how the infrastructure has been made, and even suggest improvements. This code also documents the used cloud resources, and it can even be audited for compliance [5].

But the benefits of IaC tools do not end to the infrastructure code itself. The second edition of “Infrastructure as Code: Dynamic Systems for the Cloud Age” (2020) summarised IaC benefits for organisations as follows [5]:

- *Using IT infrastructure as an enabler for rapid delivery of value*
- *Reducing the effort and risk of making changes to infrastructure*
- *Enabling users of infrastructure to get the resources they need, when they need it*
- *Providing common tooling across development, operations, and other stakeholders*
- *Creating systems that are reliable, secure, and cost-effective*

- *Make governance, security, and compliance controls visible*
- *Improving the speed to troubleshoot and resolve failures*

In addition, many IaC tools can be used in multiple cloud environments, which allows migrating from one to another without changing the tooling. Some of the tools are also open-sourced, which has helped creating a community of devoted people around them. This can mean a bigger pool of potential people to hire [4].

2.5 Comparison of IaC tools

There are many IaC tools, but all of them provide different set of features. Let us now categorize and compare tools such as Chef, Puppet, Ansible, SaltStack, CloudFormation, Heat, Terraform, AWS CDK, ARM Templates, Azure Blueprints, Cloud Deployment Manager and Pulumi.

The first difference is in the language used in the infrastructure code. Most of the tools use their own Domain Specific Language (DSL). These tools include Puppet, Chef, Ansible, SaltStack, Terraform and CloudFormation. The upside of DSLs is that they are very convenient for describing the infrastructure, as they have been created for that sole purpose. The downside is that usually the language is tool specific, and it cannot be used for anything else.

Some tools such as Pulumi and AWS CDK support general-purpose programming language such as Python or Java. This means that the same language can be used in the application or service code, which can be a major upside for many organizations already having the expertise in a specific language and the ecosystem of tools with it. Many of these tools have support for multiple general-purpose programming languages [5].

Another way to split the IaC tool languages is to procedural and declarative. Procedural code has a set of instructions, that specify the steps to take in the order to achieve the wanted end state. Most general-purpose languages used with IaC tools are procedural. Declarative code only specifies the end state that is wanted, but not the steps to get there. Most of the DSLs used with IaC tools are declarative. Procedural code does not fully capture the state that the system is in, so it is harder to use in practice as you need more external information to use it effectively [4].

To maintain control and consistency of infrastructure, you must be able to continuously apply the infrastructure code changes as they are made. This is much easier with so called idempotent code [5], which means that the result does not change if the same code will be executed multiple times. Idempotency is easier to accomplish with declarative code than procedural code:

“There should be 5 virtual machines” does not create more virtual machines when rerun unlike “create 5 virtual machines”, where you end up with 10 virtual machines after the second execution.

Most IaC tools can be used for provisioning and configuration management, but they are more suited for one or another. Also, some tools are CSP specific, others work across multiple providers. AWS CDK supports only AWS, ARM Templates and Azure Blueprints support only Azure and Could Deployment Manager support only GCP.

Some IaC tools are more mature than others. Typically, older tools have less bugs and non-backward compatible changes. When picking IaC tool, you also pick the community around it. The community contributes a lot to the project and ecosystem around the tool. It determines the number of available third-party integrations, plug-ins, and extensions. It also determines how easy it is to find help online, and how easy it is to hire an employee or consultant with the relevant skills. In many ways it can have as large impact on the user experience as the technology itself.

It is hard to determine the size and quality of the community, but in 2019 comparison was done [4] based on contributors, stars on GitHub, commits, issues, libraries, jobs, and Stack Overflow posts. It also tracked the change between 2016 and 2019 to see in which direction the trend is going. The result was that only Ansible and Terraform experienced explosive growth in the late 2010s, but many tools had at least some growth. The summarized results about can be seen from the table below at the community column.

Table 1: Comparison of IaC tools based on 2019 data, table adapted from “Terraform: Up & Running, 2nd Edition” [4].

Tool	Cloud	Type	Language	Maturity	Community
Chef	All	Config-mgmt	Procedural	High	Large
Puppet	All	Config-mgmt	Declarative	High	Large
Ansible	All	Config-mgmt	Procedural	Medium	Huge
SaltStack	All	Config-mgmt	Declarative	Medium	Large
Heat	All	Provisioning	Declarative	Low	Small
Terraform	All	Provisioning	Declarative	Low	Huge

In 2019, the biggest weakness of Terraform was Terraforms was its maturity, but the situation has clearly improved since then. The 1.0.0 version of Terraform was released in June 2021. According to HashiCorp, Terraform is currently mature and stable on the technical architecture level. The company promises extended maintenance periods for all future 1.x releases [6].

The collaborating MSP of this case study is currently using Terraform to manage their customer infrastructures. Based on the above comparison, Terraform is still an appropriate IaC tool choice. Since there are no specific general-purpose programming languages, which are currently widely used in the MSP's organization, Terraform's DSL suits well. Based on the literature, declarative code is a good choice for creating reusable modules, as it does not care about the current state of the system. As MSP already has tools for configuration management, only provisioning tools are relevant for them. The collaborating MSP uses multiple CSPs, and has services with multi-cloud and hybrid architectures, which means that provider specific IaC tool does not suit its needs.

Pulumi is an interesting alternative technology to follow in the future. It is the least mature IaC technology, with the oldest public release being from 2018 [7]. While it has a lot of compelling features, the documentation and provider support are behind more established tools, so it is not suited for the MSPs use yet.

Crossplane is also worth mentioning in the IaC tool space. It is an open-source project integrating into Kubernetes as an add-on. It allows describing infrastructure as Kubernetes configurations, and it actively maintains the configured infrastructure state [8]. It could be a good fit for teams where everyone has a deep knowledge of Kubernetes. This is not the case for the collaborating MPS, so Crossplane and its features were not evaluated further.

Based on the arguments above, Terraform is still an appropriate choice for a IaC based provisioning tool for the collaborating MSP. Therefore, it is reasonable to develop the workflows around Terraform further, to enable the reusing modules with multiple customers.

3 Terraform

Terraform is an open-source IaC tool from HashiCorp released in 2014 [9]. HashiCorp is a software company with many known open-source and proprietary tools related to development and deployment of software. It became listed on the stock exchange in 2021 with over 1650 employees [10]. Terraform is licensed under a permissive Mozilla Public License 2.0 [11].

Terraform supports managing cloud infrastructure on public cloud service providers such as AWS, Azure and Google Cloud as well as private cloud and virtualization platforms such as Open Stack and VMWare. Terraform is written in Go language that is compiled into a single binary. The managing is done through provider specific API calls. Terraform decides which API calls to make based on the Terraform configurations and the last known resource state, that is stored into a separate text file.

While Terraform supports all major cloud providers, it does not support transparent portability between providers: this means that you cannot change the infrastructure's cloud provider from one to another and expect the same code to work. This is due to the fact that CSPs are offering different infrastructure resources and Terraform does not attempt to hide it. This is an advantage as you can use all major CSPs and take advantage of each providers unique functionality and pricing, while using the same language, toolset, and practices for all providers.

Terraform is a declarative tool. It understands resource dependencies, so the configuration changes are applied automatically in an order that makes sense. This reduces mistakes, manual work, thinking, and expertise required for making infrastructure changes [4]. Terraform also supports non-declarative configuration through provisioners for behaviors that cannot be represented in Terraforms declarative model [12].

This section introduces the main features of the Terraform configuration language and explains details about Terraforms infrastructure state management. It also describes what Terraform Cloud and Terraform Enterprise are, and what additional features they provide. Lastly some good practices about organising Terraform code and effective using of it in teams are explained.

3.1 Terraform and Git

Similar to many other types of code, it is a best practice to manage Terraform related code with a version control system (VCS) such as Git. VCSs keeps a record of file changes over time and allows recalling old file versions later

[13]. Git enables developing software in teams by allowing non-linear development with fully distributed model [13]. Git stores files in *repositories*. Each set of changes is called a *commit* and a related commit message. A specific moment in the series of commits (known as commit history) can be named with a *tag*.

Terraform understands Git repositories: it knows how to refer code from other repositories through a "source" parameter [12]. It also knows how to index modules from repositories, read and read their versions from tags to public and private module registries. Git tags should use semantic versioning [14] [15] when working with registries.

3.2 Resources, data sources and providers

Terraform configurations are written in Terraform language with a filename extension of ".tf". Declaring resources is the main purpose of the language. Other language features exist only for making the resource definitions more convenient and flexible [12].

Resources are declared in blocks representing one or more infrastructure objects. One resource can represent for example a virtual network, a compute instance, or a DNS record [12]. Terraform is responsible for creating, updating, and destroying these infrastructure objects to match the configuration settings [12]. Therefore, they are sometimes called managed resources in the official documentation [12].

Sometimes Terraform requires information defined outside of the current configuration or some information that can be modified with its built-in functions. This type of information is defined as data sources which are declared in Terraform language as data blocks. Unlike managed resources, data sources are only read by Terraform, not managed by it [12].

Both resources and data sources are added to Terraform by providers. With providers, Terraform can interact with CSPs, Software as a Service providers and other APIs. The used providers must be declared in the configurations using a provider block. Providers are distributed separately from Terraform, and the main source for them is the Terraform Registry. Providers are developed and published by HashiCorp, platform maintainers or users and volunteers. The Terraform Registry splits the providers into tiers based on the source developing and maintaining it [12]. An example use of resource and provider blocks can be seen in Figure 1.

```
# Configure the Azure provider
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0.2"
    }
  }

  required_version = ">= 1.1.0"
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "myTFResourceGroup"
  location = "westus2"
}
```

Figure 1. Official example demonstrating use of providers and resources [12]

3.3 Modules

Using individual resources and data sources with variables and outputs gives the user the maximum freedom to design the cloud infrastructure, even across multiple providers at the same time. But this can quickly become a complex system with lot of duplication, moving parts and code to maintain. Terraform has a solution for this: a module. Modules allow you to contain multiple resources together and abstract away the internal complexity into sets of inputs and outputs. The modules themselves can call modules from other sources, and they can define their own child modules.

A module is declared in a module block in Terraform Language. All modules are required to have a "source" argument, and optionally a "version" argument for constraining which versions are allowed. Modules can be found from the Terraform Registry, they can be defined as local files and directories, or they can be point to a version in a Git repository. Organisations can also have their own set of modules available in a private module registry, if they are users of Terraform Cloud [12]. Figure 1 demonstrates how a specific module version can be fetched from Terraform Registry with a module block.

It is possible to apply more complex constraint rules to the module's version if needed [14] [15].

```
module "consul" {
  source = "hashicorp/consul/aws"
  version = "0.1.0"
}
```

Figure 2. Official example demonstrating fetching a module from Terraform Registry [12]

When module is fetched from another Git repository, the version number is defined in the source parameter value directly [12]. Figure 3 demonstrates, how a specific Git tag or a commit hash can be defined in the source parameter. It is also possible to specify a branch name in the "source" parameter. If a specific module version is not defined, Terraform will use the latest commit from the repository.

```
# select a specific tag
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}

# directly select a commit using its SHA-1 hash
module "storage" {
  source = "git::https://example.com/storage.git?ref=51d462976d84fdea54b47d80dcabbf680badcdb8"
}
```

Figure 3. Official example demonstrating fetching module based on semantic version on a Git tag (top) or a commit hash (below) [12]

3.4 State and backends

Terraform stores managed infrastructure and configuration state either locally into a single file called "terraform.tfstate" or to a remote backend. The Terraform state file is required for mapping real world resources to the configuration resources, keeping track of metadata and improving performance. This file should not be edited manually [12].

On a personal project, storing the state file locally on a computer works well enough for many uses cases. But with a commercial product or a team setting, this approach stops working.

First of all, every team member needs the access to the state file to be able to use Terraform and plan changes in an existing infrastructure. Unlike the code, you should not use version controls systems (VCSs) such as Git for this purpose for three reasons:

1. **Manual errors:** Most likely the state file will get out of date at some point as the latest changes are not pushed to the VCS.
2. **Locking:** VCSs such as Git do not prevent applying changes at the same time from two pcs, which leads to race condition and state file corruption, possibly leading to data loss.
3. **Secrets:** All data stored into the state file is stored in plain text format, including secrets. Production secrets should not be stored on disk anywhere.

The solution for all these challenges is a remote backend, that is a built-in feature on Terraform. Remote backend allows storing and accessing the state file from a remote store. Many remote backends are supported, such as Amazon S3, Azure Storage and Google Cloud Storage. HashiCorp has also official remote backend solution called Terraform Cloud. Remote backends solve these three issues in following ways:

1. **Manual errors:** Terraform will automatically pull the latest state from the remote store, and push the new changes, so there is no chance of manual error.
2. **Locking:** Remote backends have a locking feature, which prevents simultaneous infrastructure changes and state file corruption.
3. **Secrets:** Most remote backends support encryption in transit and at rest of the state file. Most backends also support more complex access control features than VCSs.

Remote backends are configured inside a code block called “backend”. Unfortunately, Terraform does not support regular variables inside “backend” block, but you can use -backend-config parameter to pass partial backend configurations for Terraform when initializing the environment [12]. This can be convenient for example in automation scripts.

3.5 Terraform Cloud and Terraform Enterprise

Terraform Cloud and Terraform Enterprise refer to different distributions of the same proprietary application [16]. Terraform Cloud is a multi-tenant SaaS platform provided by HashiCorp. Terraform Enterprise is self-hosted version of Terraform Cloud with some differences such as no resource limits, additional sign-on features, and more support [17].

Terraform Cloud is Terraform specific continuous integration (CI) system with many additional features compared to a general-purpose CI server [16]. It also acts as a remote backend for Terraform with a special “cloud” block [18], which saves the history of previous states and runs. Terraform Cloud allows running Terraform remotely on virtual machines on top of its cloud infrastructure. The remote execution workflow depends on the workspace type selected on Terraform Cloud. Figure 4 shows how the workflow is chosen when creating a new workspace [16].

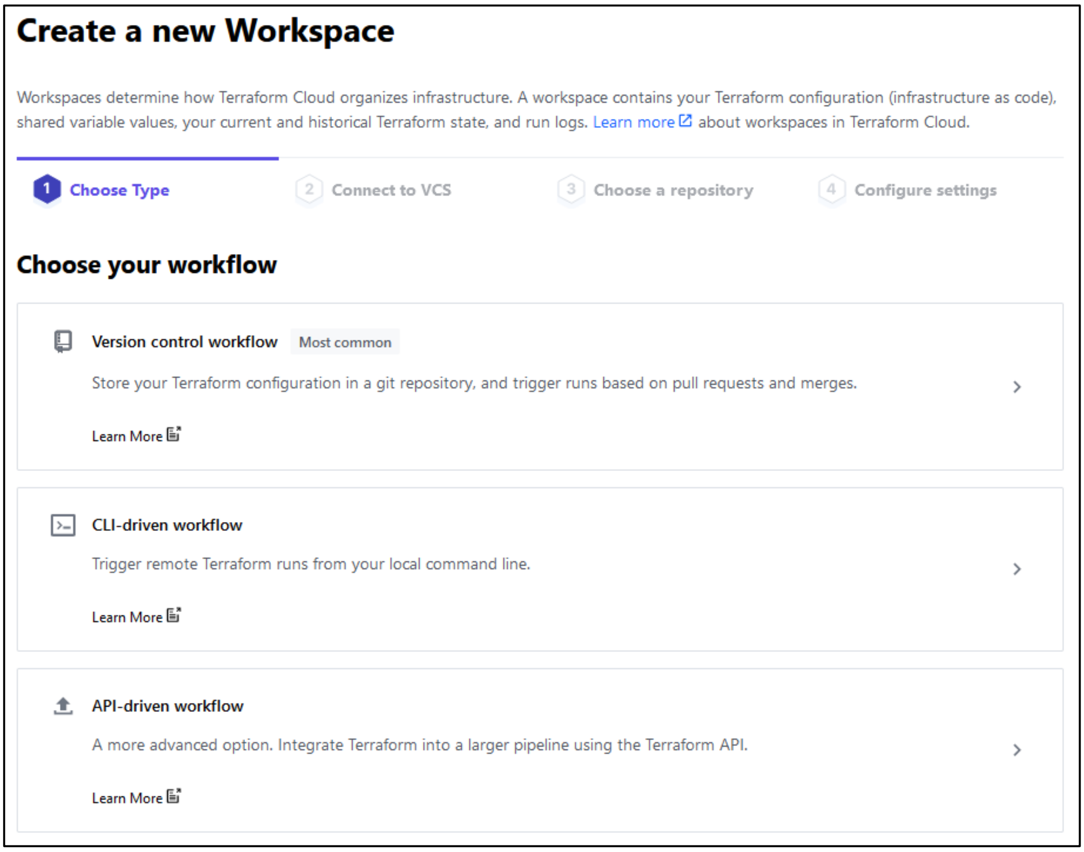

Figure 4. Terraform Cloud workflow decision when creating a new workspace [19]

The primary option is to integrate the remote runs to a version control system (VCS) provider such as GitHub, which is used for storing the Terraform code. You can either trigger the runs manually, or automatically when new code is added to VCS. The second option is to use CLI-driven workflow. This means triggering the run form the command line interface. The commands and the output are very similar when using Terraform locally. The main difference between them is that the code is executed remotely instead of local computer [16]. The third option is to use API-driven workflow. Terraform Cloud exposes an API that allows for example remotely creating workspaces and executing arbitrary Terraform code not tied to VCS [16]. Different workspaces can be connected through Run Triggers, so that successful infrastructure changes trigger events in other workspaces.

Terraform Cloud provides some ready-made integrations to other systems such as ServiceNow and Kubernetes [16]. These allow for example triggering infrastructure provisioning when some conditions are met. The Terraform Cloud API allows building custom integrations and systems that are not available otherwise [16]. Terraform Cloud can also notify other systems through webhooks [16] or execute tasks in external systems through Run Task API [16].

Terraform Cloud adds access to a private registry. This is very similar to the public Terraform Registry, where the public modules and providers are hosted, but in this case only the members of the same organization inside Terraform Cloud can have access to the added modules and providers. Private registry has similar search and discovery features as the public Terraform Registry [16].

Terraform Cloud's access control model allows managing permission based on individual user, team, or whole organization [16]. You can also enforce policies over which types of infrastructures can be created. Policies are set though HashiCorp Sentinel framework. Terraform Cloud also has cost estimation features [16].

Terraform Cloud supports two-factor authentication [16] and Single Sign-on (SSO) through multiple identity providers such as Microsoft Azure AD. Terraform cloud is also ISO 27001 certified and SOC 2 Type I compliant [20]. Terraform Cloud also has some uptime guarantees through service level agreement (SLA) [21]. Terraform Cloud supports storing and injecting secret variables to Terraform [16].

Terraform Cloud has three pricing tiers: Free, Teams & Governance and Business. Free pricing plan is limited to 5 users, and it includes state management, remote workflows, and a private module registry. Teams & Governance pricing plan adds team/policy management features, official support, and SLA. Business tier pricing is not publicly available. It adds SSO features, option to self-host the application, audit logging, reduces resource and performance limits and has more technical support available [22].

3.6 Deployments

Like application code, the best practice for deployments is to use a continuous integration (CI) server instead of developers own computer [4]. The benefits of CI servers are that it helps managing access to production environments, deploys always happen from a consistent environment and it forces automating the whole deployment process.

But unlike with application code CI server, where the server permission can be made explicit and limited to only deploying the application, infrastructure CI server requires arbitrary permissions [4]. The Terraform code can for example destroy databases and create new users. In practice this means giving admin permissions to the CI server.

The consequence is that CI server for IaC code is a larger security risk compared to application code CI servers. To mitigate this challenge, the CI server should not be exposed to the public internet. This means that for example GitHub cannot trigger build with webhooks. One option is to make your CI server poll the VCS for updates by itself. The server should be accessed only through authenticated VPN connection. CI server could also have somehow limited permissions by default, and a way to manually provide temporary full admin privileges [4]. Some of the most popular Terraform specific CI server options include: Terraform Cloud [16], Atlantis [23], Spacelift [24] and setup-terraform (Official GitHub action solution) [25].

Terraform does not provide any deployment strategies out of the box, such as blue-green or rolling deployments or feature toggles [4]. It also does not roll back to previous state in case of errors, as in many cases that is not possible. The deployments also can take quite a while and can occasionally fail even with valid infrastructure code.

This means that more advanced deployment strategies must be built on top of Terraform or included in the provider itself. Also, this means that CI server must handle error states by allowing retrying the deployment [4]. It would be good to handle some more rare Terraform error such as saving the potential "errored.tfstate" file or force unlocking the backend due to connection issues or server crashes.

3.7 Organizing Terraform code

The previous subsections introduced how Terraform works and some of its main features. This subsection gives details on how to start organizing Terraform Code when building infrastructure.

3.7.1 Isolating environments

As seen in 3.4, remote backends allow storing and modifying the state of the infrastructure from multiple sources. But after the infrastructure is live and has users, modifying the production infrastructure directly is very risky and should not be done until the changes have been verified working. Isolating a separate environment for development and testing solves this issue. In this subsection, we will focus on how Terraform allows you to isolate environments.

A simple approach would be to define all environments in one large Terraform configuration. While it may seem to work, this approach has a major drawback: is does not provide a proper isolation between environments. As there is just one shared state file for all the environments, every single “terraform apply” -command risks accidentally breaking the production environment or even corrupting the shared state file. To have proper isolation between environments, you need to have a separate state file for each environment. To achieve that, Terraform has two options.

One option is to use Terraform CLI Workspaces. These should not be mixed with workspaces inside Terraform Cloud from subsection 3.5, as they are separate features with different functions [16]. Terraform CLI workspaces allow you to create and store multiple state files from the same codebase. A new named workspace can be created from the command line, and corresponding new state file will be created to the remote backend. You can also use conditional logic inside Terraform code to have different infrastructure configurations in different environments.

Workspaces are an easy and officially supported way to test different infrastructure changes with a separate state file, but they some drawbacks. It lacks visibility: there is no visible indicator, what the currently active workspace is. There is a real risk, that you accidentally push changes or run destroy commands to production. The lack of visibility also means that it is hard to say, what infrastructure changes have been pushed and where. Workspaces are also limited to one backend: you must have the production and test environments in the same store. This means that by default you have the same authentication and access controls for all created environments. This authentication and access control could be potentially circumvented with some remote backends with custom access controls [26] [12], but this use case does not seem to be documented. Due to the limited separation between infrastructures in different workspaces, official Terraform documentation discourages their use when used with development stages or different teams [12].

An alternative approach for isolating infrastructure environments is to use the file layout directly to achieve isolation. This is encouraged by the official documentation [12] and the literature [4]. This means that you will have separate folder for each environment. This means that you can now configure each environment backend separately and have full control in authentication and access controls. With one environment per folder, you to have greater visibility: each folder represents the infrastructure deployed to that specific backend.

Compared to the workspace isolation, the file layout isolation technique results in more code to be managed. But this drawback can be minimized by having just the unique backend configuration there, and defining the whole infrastructure as re-usable modules, potentially with different parameters. To reduce the code duplication even further, you can use tools such as Terragrunt.

Terragrunt is an unofficial CLI wrapper for Terraform. It is built by Gruntwork.io and it can help to get rid of duplicated backend code when working with multiple environments [27]. It is an open-source project licensed under a permissive MIT License [28].

The main advantage of Terragrunt is that it allows following "Don't repeat yourself" (DRY) principle in Terraform backend configurations, which results in less IaC code to maintain. Terragrunt does with a base configuration containing all of the generic code from multiple environment configurations, and then allows those environment configurations to inherit from this base configuration [27]. As mentioned in subsection 3.4, Terraform has a limitation that you cannot define regular variables inside backend configuration. Terragrunt works around this issue in a more flexible way than the official “-backend-config” [12] parameter workaround.

Terragrunt has also other features, such as simplifying complicated Terraform CLI commands and adding hooks that execute arbitrary commands on the caller machine before or after Terraform commands [27].

3.7.2 Live code and module code

There are benefits for separating Terraform module code and live infrastructure code and development to their own Git repositories. Module code is not deployed by itself, and it can be used in multiple places. Using modules can happen between different projects, or between different environments (see subsection 3.7.1). Live infrastructure code is currently deployed and available.

While both the literature [4] and HashiCorp [12] agree that live code and reusable module code should be separated into their own repositories, there is a debate on whether you should have all the reusable modules in a single “modules” repository, or have each module in its own repository [29]. Let us now analyze the tradeoffs between the approaches.

The official Terraform documentation recommends having each re-usable Terraform module in its own repository [12]. The main advantage in this approach is that each module has its own commit history and versioning. This means that each module can be access controlled separately. Terraform Cloud's private module registry only supports this module format [16] , which is called "Standard Module Structure" in the documentation [12]. In addition to the main root module, the repository can include nested modules [12].

In contrast, a single "modules" repository has an advantage of being potentially simpler for development. The root of "modules" repository contains all the created modules in their own directories. When all the modules are in the same directory, you can make similar changes to multiple modules in a single commit. Module discoverability might also be easier as you have only one source for all modules. But the modules have only one shared commit history and version and cannot be access controlled separately. Terraform Cloud's private module registry cannot read modules from this format [16]. Single "modules" repository uses more network and space when deployed, as it currently fetches the whole repository for each used sub-module separately, even if the same version is used [30].

4 Assessing reusable Terraform modules for MSPs

This section considers reusable Terraform modules in MSP setting. It explains why the reusable Terraform modules have not been introduced to the collaborating MSP's workflow, even when there are benefits in using them.

First, potential benefits of introducing reusable Terraform modules into MSP's workflows is considered. Then, unique challenges of introducing Terraform modules into MSP's workflows compared to a more typical Software as a Service (SaaS) company, are discussed.

4.1 Potential of reusable Terraform modules for MSPs

Next, how to provide the collaborating MSP with solutions for developing and reusing Terraform modules is discussed. The collaborating MSP is interested in this capability, as it allows more efficient service management by unifying similar Terraform based infrastructure solutions across multiple customers.

Reusable Terraform modules allow sharing tested infrastructure solutions with new and existing customers. Maintenance and testing become easier when the same new module features and fixes can be applied to many customers. If customer infrastructures can be built with reusable Terraform modules at least partially, it has the potential for reducing maintenance/testing work and making the initial delivery of managed services faster for the customer. These outcomes mean cost savings for the collaborating MSP and more value for the customer.

4.2 Challenges of reusable Terraform modules for MSPs

It is challenging for an MSP to introduce reusable Terraform modules in the customer infrastructures. In a typical SaaS company, reusing Terraform modules is simple. This is because only the company employees have access to the Terraform module source code and commit messages, and customers will only indirectly interact with the infrastructure through the service the company provides. The question about the IPR of the Terraform module source code is not relevant. Updates to a reusable Terraform module are instantly available for all the infrastructures in the SaaS company's VCS, and their employees. More fine-grained access control between the repositories can be handled through their own VCS or a Terraform Cloud/Enterprise organization. These statements are not necessarily true for an MSP.

The MSP business model restricts how reusable Terraform modules can be used in their services compared to a typical SaaS company, as in some scenarios there is a requirement for them to share the module source code with their customers. This means that there must be a mechanism to tell what modules should be available to which customers, and how to handle the future updates to these reusable modules.

When providing managed services, MSP can take multiple approaches with its customer contracts and IPR. The contract defines the required customer access to the source codes and what happens if the contract expires. It is possible that the customer has only access to their infrastructure, and not to some MSP's proprietary solutions which makes the infrastructure management simpler. These types of management solutions typically used with multiple customers' infrastructures and make MSP's work more efficient.

The first challenge with Terraform for MSPs comes from the fact that Terraform code is in "a grey area" between the infrastructure and an infrastructure management solution. So, if the contract expires and the infrastructure and its management is handed back to the customer, should the contract state that the used Terraform code must be handed over as well? It is technically feasible to hand back the infrastructure built on top CSP services, without providing access to the Terraform source code, or even to the Terraform state file. Does the interpretation change if the customer infrastructure had been brought up fully by using reusable Terraform code, and not some code developed for this specific customer? And what if the management solution is only partially implemented with reusable Terraform modules? This challenge of which types of contracts to create is a large part why the collaborating MSP of this thesis has so far been hesitant to introduce reusable Terraform modules into their customers' solutions.

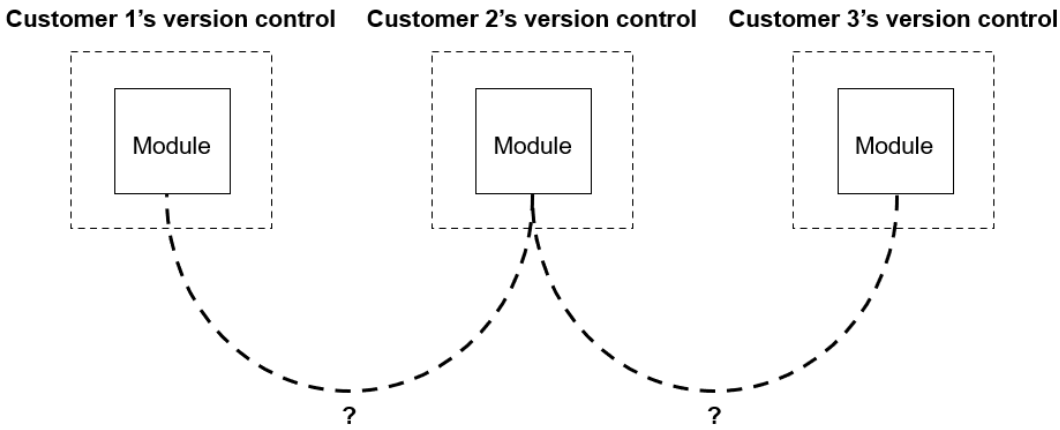

Figure 5. Module distribution mechanism challenge between multiple customer VCSs

Another challenge related to reusable Terraform modules and MSPs is the actual distribution mechanism and access control. This challenge is hardest if the interpretation of these reusable Terraform modules in the contract is that they are part of the infrastructure itself, and the customer must have a constant access to the module source code.

For MSPs it is crucial that the customers are fully isolated from each other, and MSP must take considerable precautions when dealing with sensitive customer information. For example, customer companies cannot be allowed to see each other's names, as this is sensitive information. This means that the earlier SaaS approach of having a single VCS, or a Terraform Cloud/Enterprise organization does not work for MSPs. Giving customers a direct read access to the reusable Terraform code to either one would break the customer isolation[2], and the workarounds[3] are not worth the risks.

Almost all Terraform and IaC related literature, documentation and blogs assume that a single VCS or Terraform Cloud/Enterprise fits the need for Terraform users, who want to reuse modules between the infrastructures. And this is the case for most users such as a typical SaaS company. But as stated earlier, this approach does not work for the collaborating MSP, if the requirement is to give customers access to the reusable modules used in their infrastructures. Some different solution is required.

[2] Git service providers typically shows usernames to everyone in the same organization. It seems that currently Terraform Cloud permissions could potentially be set so that different users do not see each other, but as it is not meant for this type of multi-customer isolation, it should not be relied on.
[3] Anonymized, MSP provided usernames could work with VCS or Terraform Cloud/Enterprise, but this solution was ruled out by the collaborating MSP.

5 Analysis of a customer infrastructure

The potential of reusable Terraform modules and related challenges were discussed in the previous section. In order to get more views of potential and actual reusability of Terraform modules, a survey about a customer infrastructure was conducted for the collaborating MSP's developers. The infrastructure code was also programmatically inspected. This analysis helps to estimate the potential benefits of adopting a workflow, which allows reusing Terraform modules between the customers. First, the analysis methods are introduced, and then, the results are presented and discussed.

5.1 Methods of the analysis

Since the collaborating MSP has observed that multiple customers have at least partially similar needs, sharing modules between customers might be a reasonable way to increase efficiency. To quantify the potential of sharing Terraform modules between the customers, a survey was conducted about a part of MSP customer's Terraform infrastructure. The selected infrastructure, which was chosen by the developers, represents a typical customer infrastructure. The two developers who had designed and implemented this customer infrastructure responded to the following survey questions.

First, individual module names used in the selected infrastructure were gathered, and then the developers were asked to estimate the reusability of individual modules with two criteria:

1. *Potential reusability: is this module reusable with other customers?*

   1 = This module is very customer specific and cannot be reused outside this customer.
   5 = This module is very generic and could be used with other customers with similar needs.

2. *Reusability in practice: how easily this module could be used with another customer?*

   1 = This module would need almost full rewrite or major refactoring for another customer
   5 = This module could be reused with another customer after some minimal changes and testing

When survey participants responded to these questions, they used the scale from 1 to 5. The survey responses were collected and visualized in order find out the reusability of these modules, from two perspectives. The results give a direction whether the current Terraform infrastructures contain potentially reusable modules or not.

As a part of the infrastructure analysis, the modules were also measured based on their size in lines of code. Lines of code in this context means the total lines inside module files, including potentially empty lines. This measurement was done with a Python script. The results of this analysis are visualized in the following subsection. Visualization is done in treemaps with Plotly Python library.

In addition to measuring the amount of potentially reusable modules, the additional module size data allows calculating how large portion of the code is reusable.

5.2 Results of the analysis

The selected customer Terraform infrastructure is relatively small. It consists of 9 modules and total of 27 submodules. The modules and submodules are visualized in Figure 6 below. The real names of the files, modules and submodules have been redacted.

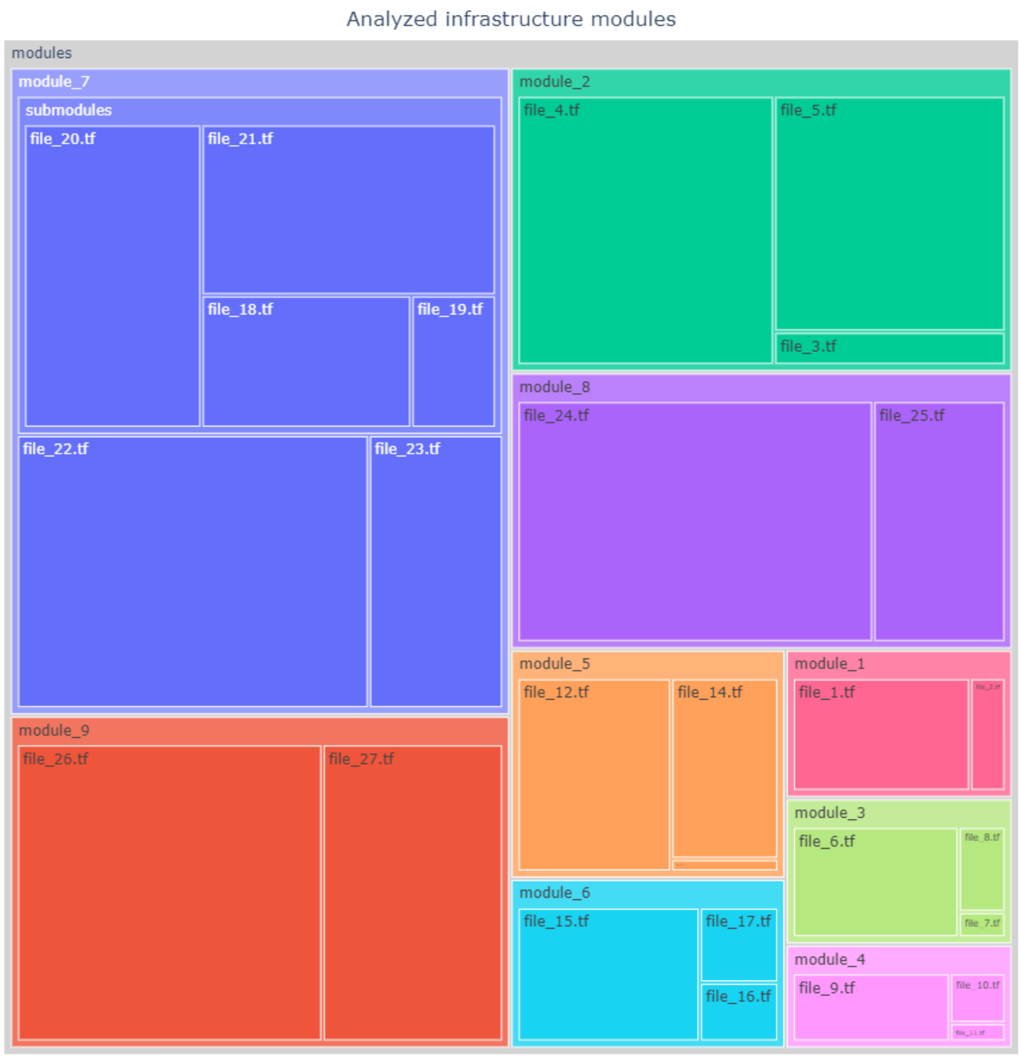

Figure 6. Analyzed infrastructure modules

In Figure 6, modules are visualized with colors, one color corresponding to one module. The size of a colored rectangle corresponds to the size of module or submodule in lines of code. The smallest module is 42 lines of code, and largest 576 lines of code. The total size of this infrastructure is 1747 lines of code.

The first question asked to rate the reusability potential of each module from one to five. The average responses from survey participants are visualized in Figure 7 below.

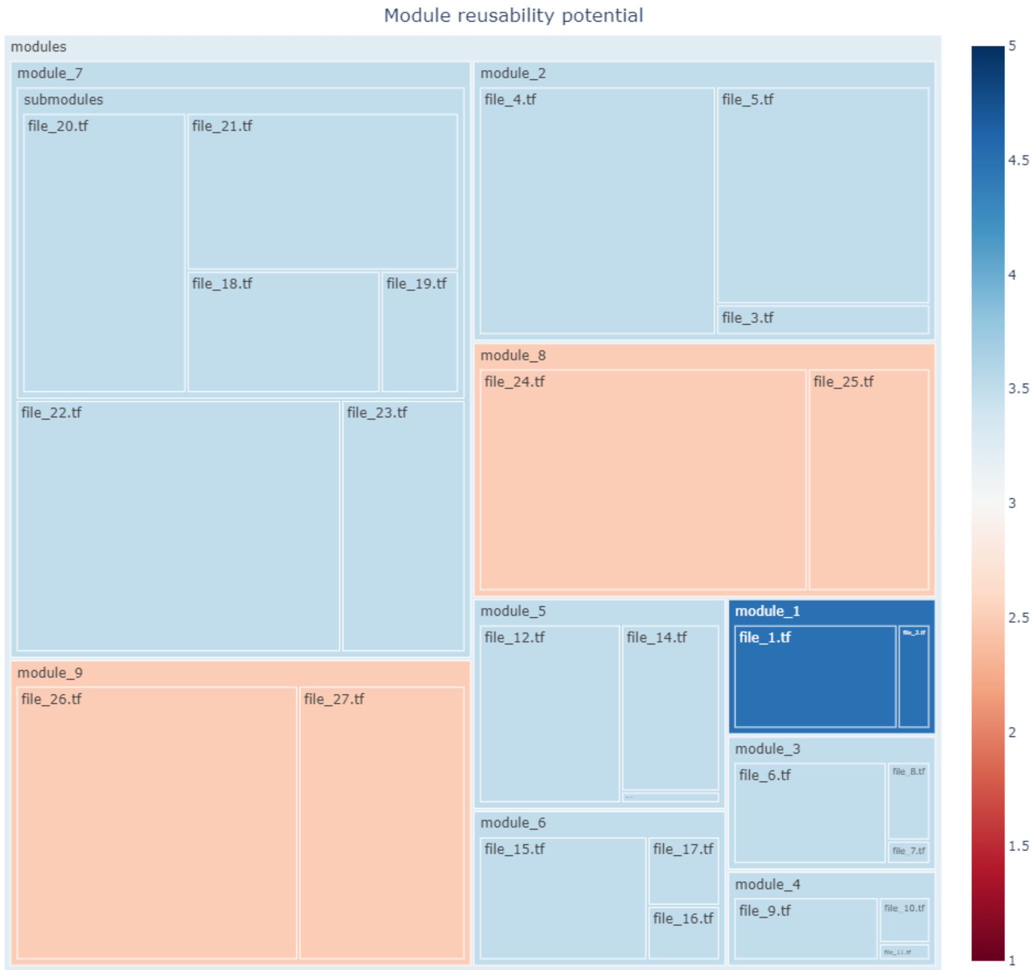

Figure 7. Module reusability potential

In Figure 7, varying colors illustrate varying reusability potential of modules in the selected customer infrastructure. More specifically, dark red refers to poor potential module reusability and dark blue very high potential module reusability. White color represents a neutral average response.

Figure 7 shows that one relatively small module with 60 lines of Terraform code was rated higher than the rest with a potential average reusability rating of 4.5. 6 other modules were rated to have some reusability potential with a rating of 3.5. Two medium sized modules were rated to have only limited potential reusability with a rating of 2.5. Individual responses varied from 2 to 5.

The other question was about reusability of these modules in practice. The survey scale was again from 1 to 5, and the responses were visualized in a similar way to Figure 7. The results can be seen in Figure 8 below.

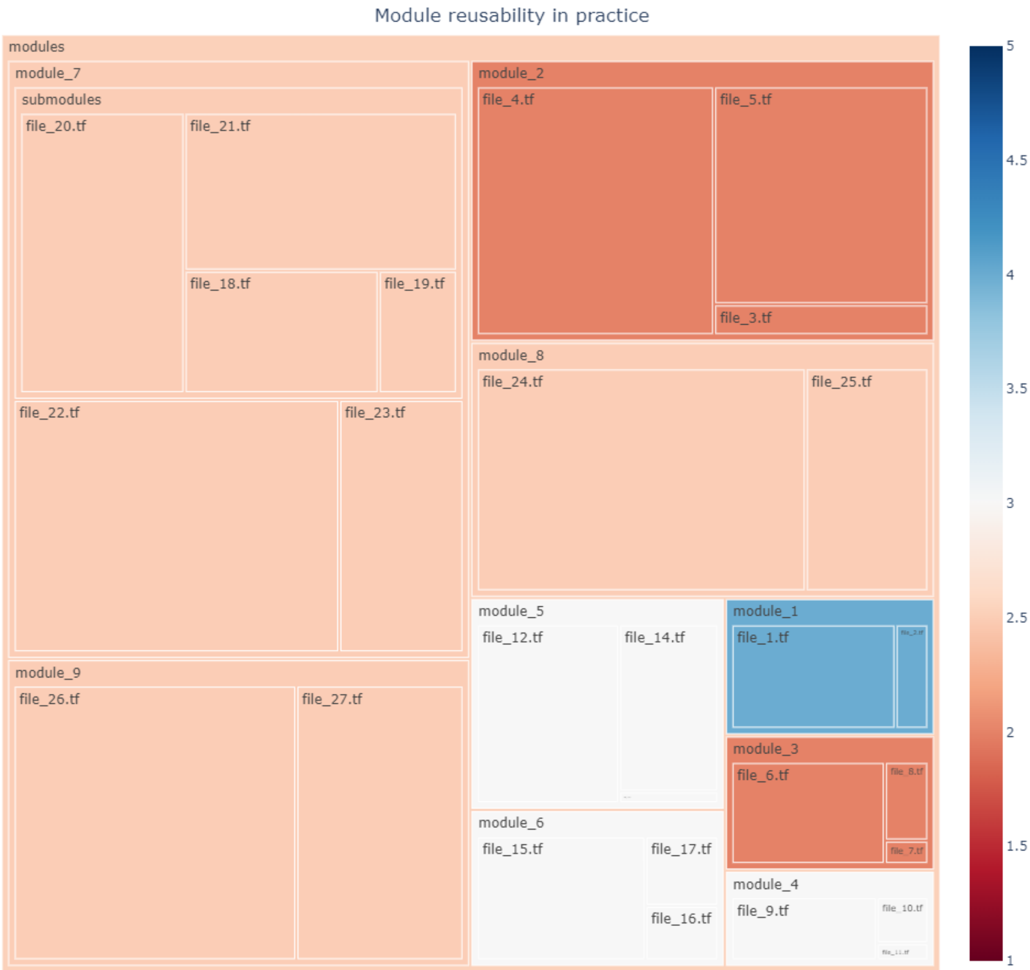

Figure 8. Module reusability in practice

Interestingly, the module with the highest reusability potential (Figure 7) ranks the highest also when asked about reusability in practice (Figure 8). Based on this survey, this module has a high potential for reuse, and it should be quite straightforward to be reused in practice, as it has a reusability in practice score 4 out of 5.

The rest of the modules did not get practical reusability scores over 3. Two modules got a neutral score of 3, three modules got an average score of 2.5 and two of the modules got a score of 2. Individual responses varied from 2 to 4.

It is worth noting that there is a noticeable difference between the potential reusability and reusability in practice: the overall average module reusability potential was approximately 3.89 while the reusability in practice was on average somewhat lower, approximately 2.72. If these scores are weighed by the module sizes, these average scores are approximately 3.22 and 2.52.

5.3 Discussion

Overall, the analysis results reveal that the customer infrastructure includes some reusable parts. The more reusable modules provide an opportunity to increase the operational effectiveness and reduce costs if a workflow allows reuse of these modules between multiple customers.

It seems that there is a large difference between the reusability potential and reusability in practice. This result is probably expected, as these modules were developed for this specific customer infrastructure, and not more general reusability in mind. If module reusability had been one of the infrastructure design criteria, it would have been possible to divide this infrastructure into modules in a different way.

The results show that according to its developers, there are clear differences in the module reusability, and there exists a small number of modules that have significantly higher reusability potential. It should be noted that one responder was more optimistic on the reusability of these modules than the other, approximate average scores were 3.89 and 2.22, respectively.

Summing up, the analysis reveals potential for reusing Terraform modules between customers, if the current workflow could be modified to allow this. Module reuse could lead to considerable cost savings, especially if the reusability of a module would be a requirement during the initial development.

6 The current IaC workflow

This section describes the current IaC development workflow by paying special attention to its strengths and weaknesses based on existing literature presented in sections 2 and 3. The current workflow was researched by interviewing its developers and investigating how VCS is used. The workflow description focuses on how the Git is used, and how modules and infrastructures are split into repositories.

Currently, the partner MSP's customer infrastructures managed with Terraform are individual repositories which do not reuse modules among themselves. Typically, the Terraform related IaC code is in customer's Git VCS, where some MSP employees and the customer has the access into.

The Git repository structure for a Terraform project is not standardized, and it depends on the customer needs. A typical customer infrastructure managed with Terraform is in a single Git repository which includes customer specific modules, which are referenced directly to a client infrastructure through Terraforms "source" parameter. The repository structure is shown in Figure 9.

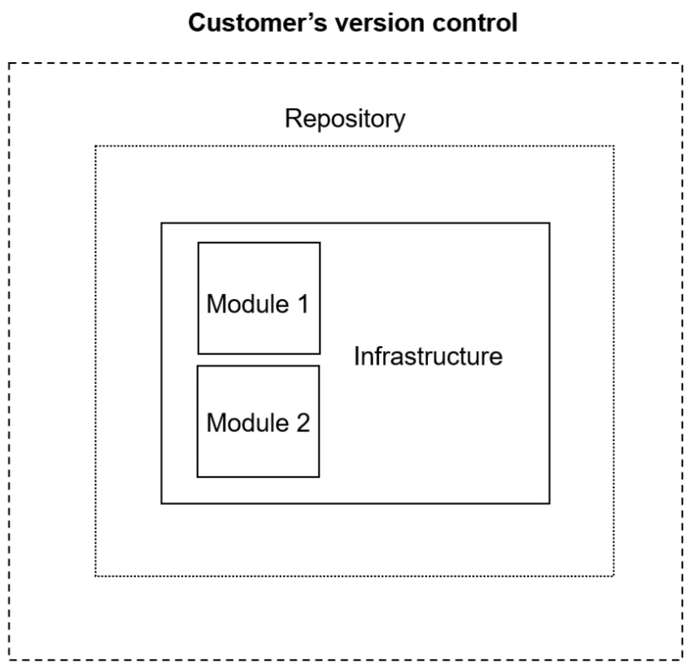

Figure 9. The current workflow with a customer that has one infrastructure managed with Terraform

The strength of this workflow is in its simplicity: as there are just individual infrastructure repositories to manage at a time, committing new changes is simple, and there has been no need to version these customer specific modules separately from the infrastructure. This workflow has worked well with customers who require simple unique infrastructures. But this workflow does have scaling challenges with larger infrastructures and larger customers.

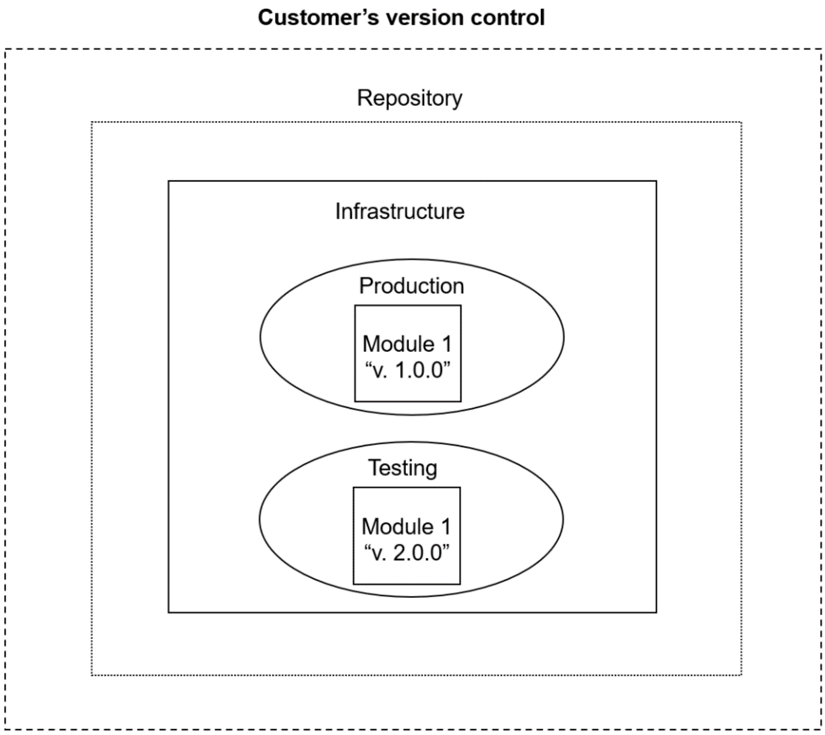

Figure 10. The current workflow with multiple environments

If a customer has a large infrastructure which requires separate testing and production environments as seen in Figure 10, different module versions[4] codes are duplicated between environments during the development. This duplication is necessary as it prevents configuration drift [4] in production during the new module version development. When the changes are ready, the code of the new module version is manually copied over to production directory and applied.

While this approach is manageable, when there are only few modules, it has shortcomings. Manual copying becomes a large and error prone task if there are many modules under development at the same time in these environments. This workflow also does not force to design the modules in a way that

[4] The modules are not currently versioned, and the version numbers in this figure just show that there are two separate versions of the same module in the repository at the same time.

they can be used in different environments, so in addition of the manual copying, there is also an error prone checking that the module can be directly copied over to the next environment.

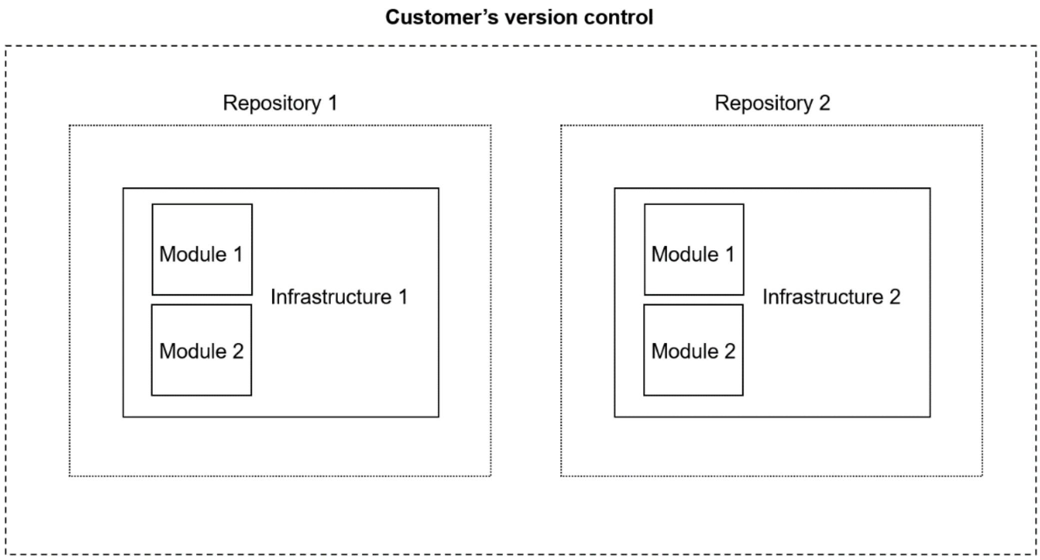

Figure 11. The current workflow with multiple customer infrastructures

If a customer has multiple infrastructures requiring similar Terraform modules as seen in Figure 11, there are similar challenges to multiple the situation with multiple environments. The module is developed and tested in one of the infrastructures, and if the changes were successful, the changes are copied manually over to other infrastructures in the current workflow.

For the development, it would be convenient if the similar modules in separate repositories would have similar interface and functionality. But as the IaC code of individual infrastructures and their modules are stored in separate repositories, the similar modules must be manually kept in sync when changes and fixes are introduced, or they will stop being similar. There is also no easy way to see, what are the differences between the similar modules in separate repositories, so the developers must assume that they might be different.

In practice this has led to developing multiple similar but not identical modules with larger customers, which results in more work in maintenance and testing. This is the currently used workflow with Terraform and VCS in the collaborating MSP at least with some customers.

7 Development process for Workflows 1-3

The collaborating MSP wants to explore adopting reusable Terraform modules in three scenarios which have distinct IPR requirements. In order to proceed systematically, three separate workflows, i.e. *Workflow 1, Workflow2* and *Workflow 3*, are determined. Table 2. Customer code access requirements for Workflows 1-3 shows how IPR based customer code assess requirements and Workflows 1-3 are related.

Table 2. Customer code access requirements for Workflows 1-3

	Customer has access to customer specific code	Customer has access to reusable module code
Workflow 1		
Workflow 2	X	
Workflow 3	X	X

The development process of Workflows 1-3 proceeded iteratively. Several workflow approaches were tested to meet the above requirements. The existing literature was reviewed, Terraform features were examined, and the current workflow was assessed. In practise, reference implementations for each workflow were thoroughly experimented and presented to many stakeholders in the collaborating MSP's organisation. The tests and feedback from stakeholders clearly revealed that the initial requirements in Table 2 are not sufficient in describing what features are required for Workflows 1-3.

7.1 Additional requirements for Workflows 1-3

The additional requirements beyond the customer code access requirements and views of several stakeholders are presented in this subsection. The reference implementations for the new workflows were developed step by step until they met these requirements of several stakeholders. This was a very iterative and time-consuming process, as new requirements were recognized several times only after the workflow reference implementation was tested in practise or when feedback was asked from new stakeholders.

According to the assessment in section 6, the current workflow does not allow module reuse even when working with just one customer. The current workflow does not allow module reusability between the customer environments and infrastructures without duplicating the source code. This adds error prone manual steps that the MSP would want to avoid. The Workflows 1-3 should address this.

The current workflow gives Terraform code access to the customers in their own VCS, which brings value for them. They can see the Terraform source code in the same location with their own internal repositories. It also makes giving automation such as CI/CD pipelines access to the code simple. The current approach makes the process of handing back the infrastructure management to the customer when a contract expires simple, which means that it is easier to sell these contracts in the first place. Overall, it might be preferable, if Workflows 1-3 continue distributing the code to the customer VCS, while allowing centralized development for reusable Terraform modules.

As explained in subsection 4.2, giving customer a direct access to MSP's VCS or Terraform Cloud/Enterprise has challenges as it is easy to leak sensitive information between customers. In practice this means that the collaborating MSP wants a solution where only the MSP employees have a direct access to the reusable Terraform module code and if a customer requires a source code access, it is done indirectly. The upside of this approach is that it cannot leak information about other customers, and a theoretical malicious customer has very limited means of trying to affect other customers.

Several aspects about module updates, distribution details and sensitive data were discovered while working and developing the reference implementations for the workflows. The new workflows' full requirements are the following:

1. Customer should have appropriate level of access to the Terraform code, as described in Table 2.

2. Enable reusing Terraform modules between environments, infrastructures, and customers.

3. If customer has rights to access some reusable Terraform modules, they should gain access only to the specific modules their infrastructure requires, and they should never get access to MSP's VCS directly.

4. Distributing module updates should allow incremental, gradual adoption of module updates as an opt-in system, without forcing all infrastructures to update to the latest version.

5. No sensitive customer information should be leaked through the distribution of reusable Terraform modules to customers.

6. Reusable module development should happen in one centralized place, MSP's own VCS.

8 Proposals for new IaC workflows

This section presents the outcome of this case study, i.e., it makes proposals how to develop the current IaC workflow by making only minor changes to the current workflow or by replacing the current workflow with entirely new one. These proposals are results of the iterative process based on researching the current workflow, exploring best practices from the literature and working closely with the stakeholders as explained in section 7.

First, before proposing Workflows 1-3, subsection 8.1 shows how flexibility and module reusability of the current IaC workflow can be improved without making major changes. This suggestion does not meet the requirements from section 7, as it does not allow reusing modules between multiple customers.

Then, subsections 8.2-8.4 presents three proposals for new IaC workflows: Workflow 1, Workflow 2, and Workflow 3. These proposals are based on customer code access requirements in Table 2 and they also meet the additional workflow requirements from subsection 7.1. The workflows are presented from the simplest to the most complex one. Workflows 1-3 meet the requirements by utilizing Git VCS in a different way compared to the current workflow investigated in section 6.

Lastly, the workflows are compared based on their features and technical details discussed in subsection 8.5. This comparison provides the collaborating MSP information what the technical trade-offs related to each IPR approaches are. This information is based on testing module sharing with the reference workflow implementations, and the literature review done in earlier sections of this thesis.

8.1 Increasing module reusability of the current IaC workflow

Based on the assessment of the current workflow in subsection 6, we can argue that many of the current module development challenges could be mitigated by organizing code into repositories in a different way, using proper module versioning and references between repositories.

As seen in 3.7.2, both the literature [4] and HashiCorp [12] argue that reusable module code should be separated and developed in their own module repository (or multiple module repositories). Maybe the biggest benefit of separated module repositories is related to the fact that they allow referring different versions of the same module in multiple environments (see Figure 12) and in multiple infrastructures (see Figure 13). This referring is done through a “source” parameter that can read the semantic version from a Git tag (see subsection 3.1 for more information).

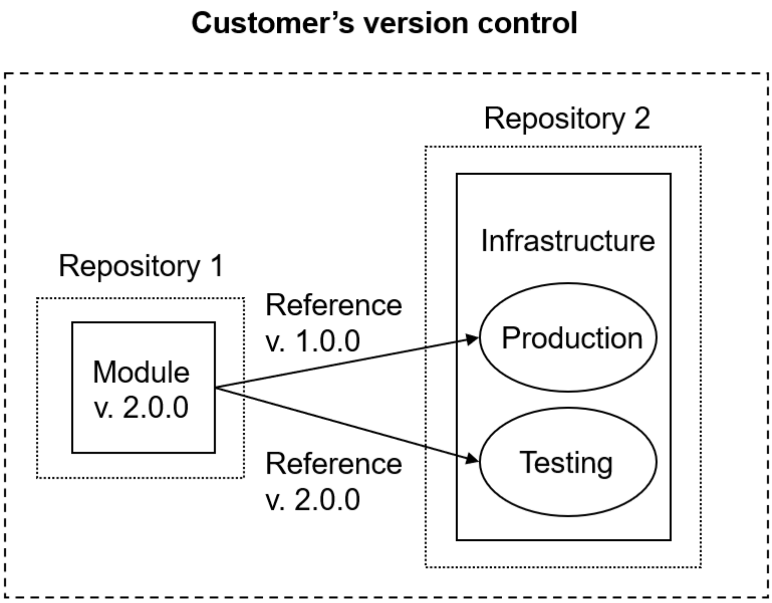

Figure 12. Module development separated into its own repository, and two module versions referenced in environments

Figure 12 shows how this process works if the production and testing environments are separated as explained in 3.7.1. In this case, the module is in a separate repository in customers VCS and is properly versioned. Both the production and the testing environment can independently refer to a version of this module. This means that when developing version 2.0.0 of the module, the production environment can continue to refer to the earlier version without any changes in its code or requirement of duplicating the module code into a separate directory.

Testing environment will start referring to the new 2.0.0 version, and the development and testing is still fully isolated form the production environment. When the new version of the module is ready for production, instead of copying new module version code manually to a different directory as in the currently used workflow from Figure 10, the production infrastructure can simply update the reference from version 1.0.0 to version 2.0.0.

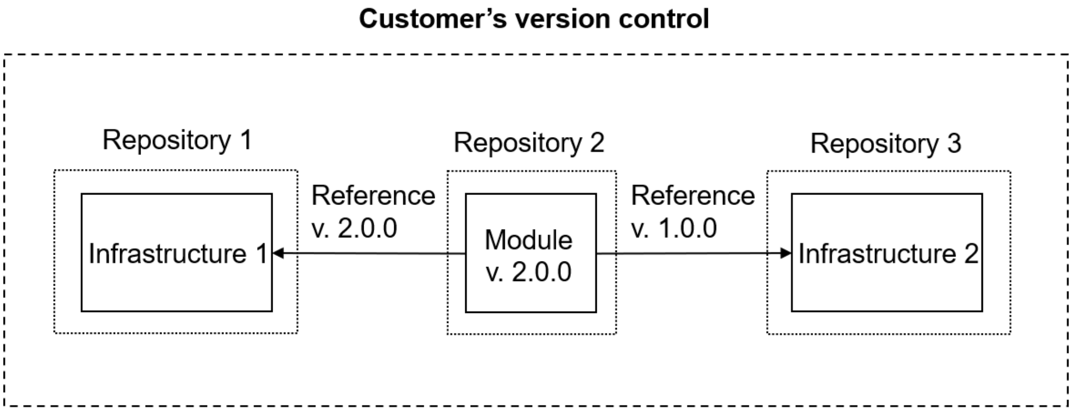

Figure 13. Module development separated into its own repository, and two module versions referenced in infrastructures

In the same way, Figure 13 shows how having a properly versioned module in a separate repository allows two infrastructures to refer to the same module with two versions, 1.0.0 and 2.0.0. This means that when the module was updated to meet the requirements of Infrastructure 1 and version changed from 1.0.0 to 2.0.0, the other infrastructure keeps referring to the previous version and running "terraform apply" does not change the other infrastructure. Only when the referenced version is intentionally changed to Infrastructure 2, the new module version is used. This way of separating modules from infrastructures does not require duplicating module code in multiple infrastructures unlike in the current workflow, which can be seen in Figure 11.

This system has an additional benefit of forcing to design the module parameters so that they can be reused in multiple environments and infrastructures. Even if the module is originally meant for only one infrastructure, this opens potential for wider use in the related customer infrastructures. In the current workflow, some modules might have some small differences depending on in which environment or infrastructure they are part of, which means that copying the source code from one place to another must be done carefully, which is time-consuming.

Table 3. The current IaC workflow and improved workflow module reusability comparison

	Modules reusable between environments	Modules reusable between customer infrastructures	Modules reusable between customers
The current workflow			
Improved workflow	X	X	

Overall, adopting an improved workflow where the module development is done in separate repositories would allow sharing modules between customer environments and infrastructures efficiently, as seen in Table 3. This is a clear benefit compared to the currently used workflow.

If module versioning is done properly, the improved workflow also increases transparency and control to where each module version is used, and an easy way to roll back to an earlier version. The amount of duplicated code in the customer VCS can be dramatically reduced with this workflow, which should make the infrastructure management and modifications easier.

It is worth pointing out that the last column of Table 3 reveals that neither the current nor the improved workflow allow reusing modules with multiple customers, which was the main improvement this thesis aims to add to the current workflow. Also, as seen in Table 4 below, both the current and this improved workflow gives a full access to the Terraform module source code for the customer, and this is not possible to change, if all of the development takes place in the customer's own VCS. This is challenging from the point of view of the different customer code access requirements from Table 2.

Table 4. The current IaC workflow and the improved workflow customer code access comparison

	Customer has access to customer specific code	Customer has access to reusable module code
The current workflow	X	No reusable module code
Improved workflow	X	No reusable module code

8.2 Workflow 1: No customer code access

Workflow 1 allows reusing Terraform modules between customers, where a customer does not have access to the Terraform code used to manage their infrastructure.

This is very similar workflow to the improved workflow in the previous sub-section. The only difference is that the code is moved from all of customers' VCSs to MSP's VCS, which can be seen in Figure 14.

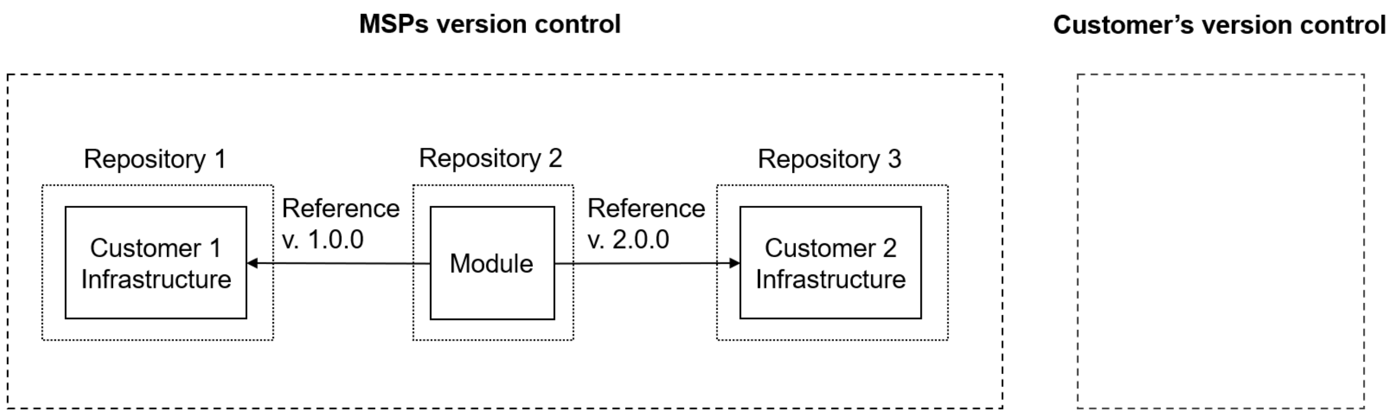

Figure 14. Both module and infrastructure code in MSPs VCS. No customer code access.

This change to the VCS system in use has some appealing properties. This Workflow makes Terraform development simpler compared to the current workflow, as there is only one VCS to work with instead of multiple customer VCSs. This change makes it very practical to reuse Terraform modules with many customers: as all the customer infrastructures are in the same VCS with the Terraform modules, they can be referenced directly with the “source” parameter.

Unlike the current workflow, this VCS strategy is very comparable to a typical SaaS company (see subsection 4.2), and most of the best practices from the literary could be applied directly. In this process there is no need to duplicate infrastructure or module code, even if the same module is in use with multiple customers.

Customers get indirectly access to reusable Terraform modules, which means that the services provided by the MSP are well tested and potentially served faster. But it hard to identify any direct benefits for the customer compared to the current workflow. Previously the customer oversaw the access controls for the IaC code in their VCS, which is no longer the case in this workflow. This way of working reduces the transparency: the customers are not able to monitor who has access to the IaC code, and they cannot see who has made changes and when, as they cannot access the commit messages from the repository. This might mean that they pay for some services, but the results of the payment are hidden from them.

Another downside for the customers in this Workflow 1 is that they do not get access to IaC solution used to manage their infrastructure. This might make the service harder to sell to the customers, and they might value the provided service less.

In Workflow 1, creating automation such as deployment and testing pipelines is very similar to the current workflow: as earlier, those automations require an access to the VCS, which is now managed by the MSP instead of the customer.

8.3 Workflow 2: Partial customer code access

Workflow 2 allows reusing Terraform modules between customers, where a customer has access to customer specific Terraform code but not to reusable Terraform module code.

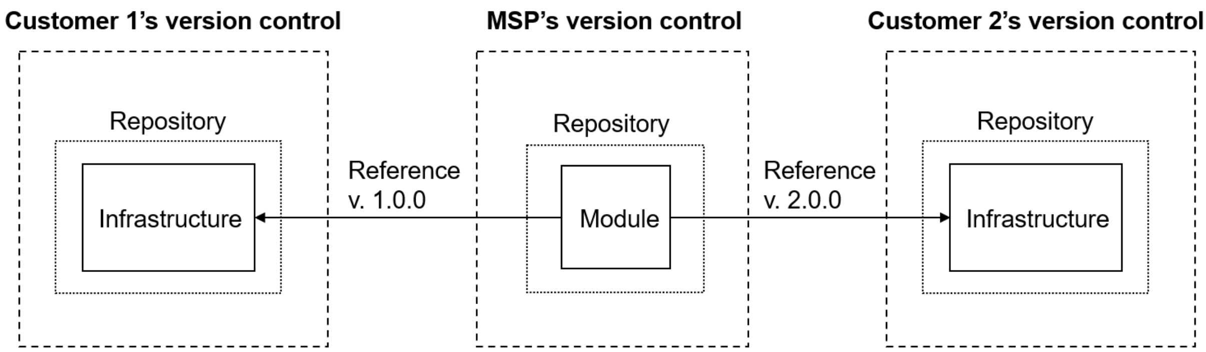

Figure 15. Referencing module directly from MSP VCS to a customer VCS. Partial customer code access.

Contrary to Workflow 1, the customer specific solutions are developed in their own VCSs in Workflow 2. The reusable module development is kept in the MSP's version control, and the customer modules can reference the modules from there.

The customer never gets an access to the MSP's VCS, which means that only the MSP employees can run "terraform apply" command, which clones the module repositories as a part of the process.

Workflow 2 is more transparent than Workflow 1: customers get to see what development has been required for their infrastructures. This might be valuable and increase the customer satisfaction, especially if they are being billed for the required custom work. In this workflow, the customer can take part in access control, and see which MSP developer accounts have access to their infrastructures, but they do not get this information about the reusable modules used in their infrastructures.

The downside for the MSP is that this adds some complexity, as there are now multiple VCSs to work with. Also, splitting IaC code in many repositories makes adding automated testing and deployments trickier, as the automation requires access to both VCSs.

8.4 Workflow 3: Full customer code access

Workflow 3 allows reusing Terraform modules between customers, where a customer has access to customer specific Terraform code and reusable Terraform module code used in their infrastructure.

According to the additional workflow requirements (see subsection 7.1), the reusable module development should happen in the MSP's VCS, and the customer should not have access to it. Also, it would be desirable the reusable modules that the customer has access to, should be accessible in their own VCS.

This means, that from the customer's perspective, Workflow 3 is very similar to the current workflow, as both give a full access to the Terraform code in their own VCS. However, some of the modules they have access to might be reused with multiple customers, which means that they are potentially better tested, and services could have been delivered faster.

Workflow 3 accomplishes this by setting up Git repositories in a different way compared to all the other workflows. The customers should have the reusable modules they use in their own VCS, but in a way that the MSP employees can pull module updates from the original source repository in the MSPs VCS. This setup is presented in Figure 16.

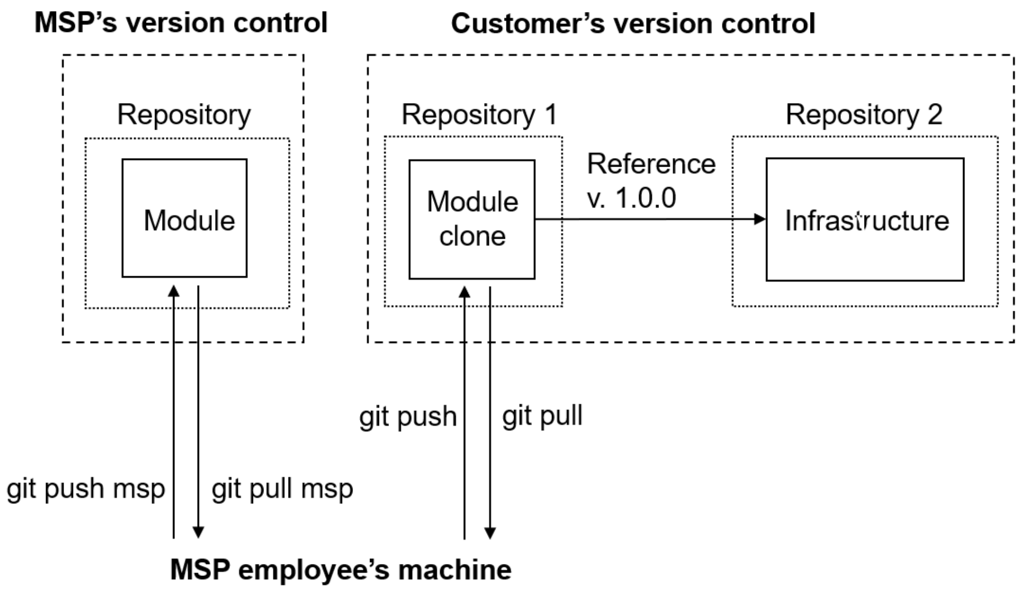

Figure 16. A setup where MSP employee can update module code between MSP and customer VCSs. Indirect module code access.

To get this process working, the customer's repository should be made to track both the MSP's original module repository, and the customers cloned module repository. This can be accomplished by Git's "remote" feature [31]. Let us say that the original module repository is at "*<module_url>*" with a branch called "main". Then a new empty repository has been created in the customer VCS at "*<module_clone_url>*". Then the process is the following:

git clone <module_url>
git remote rename origin msp
git remote add origin <module_clone_url>
git push -u origin main

After these steps, a new clone of the module has been pushed to the customer VCS. A command “git remote -v” lists both remotes, one called “msp” and another called “origin”, which is the default one.

Now the customer employees can use this cloned module like any other Git repository in their VCS. Commands “git push” and “git pull” point to their own VCS repository (origin), which most likely is the behavior they expect. But they do not have a direct access to the MSP’s VCS. Only the MSP employees can successfully run commands such as “git pull msp” and “git push msp”. For example, to get the latest module updates to the customer module clone, an MSP employee can run commands: “git pull msp” and “git push”.

Comparison of Git commit histories between the original repository and the customer clone reveals if there are updates since the last time updates were pulled. After this, the module can be used in the same way as in the improved workflow in subsection 8.1.

From the customers perspective, Workflow 3 is more valuable and more transparent than Workflow 1 and Workflow 2. It gives them access to all the Terraform based IaC tools MSP has developed to manage their infrastructure. They can also see all the development done in their infrastructure and take part in access controlling their IaC code. These features potentially have a positive impact on the price the customer is willing to pay for the service, or the service might be easier to sell in the first place.

Unlike in workflow 2, in this case it is simple to leverage automated testing and deployment pipelines, as the automations need to access the IaC code is in one VCS instead of multiple. As Workflow 3 is enabled through Git, this system works with all kinds of files, not just Terraform modules.

Workflow 3 is the most complex one introduced in this thesis. It requires as many VCSs as Workflow 2. In addition to that, it requires the mechanism of syncing customer clones of the modules. Workflow 3 has some requirements and limitations that are not present in Workflow 1 or Workflow 2. They are discussed in the next chapter.

8.5 Comparison of Workflows 1-3

The previous subsections described unique features of Workflow 1, Workflow 2, and Workflow 3. Next, we compare more technical details and features of these three workflows.

Table 5. Comparison of the proposed workflows' technical details

	Requires splitting modules into multiple repositories	Requires a process to keep sensitive information out of generic repositories
Workflow 1		
Workflow 2		
Workflow 3	x	x

Table 5 presents some technical details that are relevant when choosing the workflow. The first column indicates that only Workflow 3 requires splitting reusable modules into multiple repositories. This is due to the fact that one of the requirements in subsection 7.1 is that the customer should get access to only the reusable modules their infrastructures require, not to all of them. And if you put all the modules in one repository, you cannot clone just part of it to a customer.

As mentioned in subsection 3.7.2, there is a tradeoff between having all reusable modules in the same repository and having them in separate repositories. A major question is whether it is better to have the modules share a single version (a Git tag) or whether it is better to version them individually. Only Workflow 3 requires having all modules split up into individual repositories and versions, while Workflows 1 and 2 can function either way.

The second column in Table 5 reveals that only Workflow 3 requires a process to keep sensitive information out of the reusable module repositories. Similar to the first column of Table 5, this is related to the fact that the actual module code is shared with the customer. This means, that there cannot be any sensitive information about any customer in commit messages, file names, code, or comments, as this information would be shared among the customers, which would be against the additional requirements from subsection 7.1[5].

Enforcing this rule in practice might be hard, as automation tools cannot reliably distinguish sensitive information from non-sensitive one. A code review process for each change or additional training could be a solution for this. Workflow 1 and Workflow 2 do not have this requirement, as the code is never shared with customers. In practice this can make development easier as the commit message can state for example which customer the change is related to.

Workflows 1-3 utilize Git VCS in a new way to distribute reusable modules between customers. Another option would be to use Terraform Cloud/Enterprise (see subsection 3.5) and its private module registry. This could be convenient especially if other features from the Terraform Cloud/Enterprise will be utilized. In Workflow 3, there would be similar risks in giving access to Terraform Cloud organization as there is to give a direct customer access to MSPs Git VCS related to leaking the information about other customers. This option is not explored in this thesis further.

[5] It should be noted that technically it is possible to share modules without Git commit messages, but this complicates the sharing process, and does not remove sensitive information that is potentially in file or variable names.

Table 6. New proposed workflows, module reusability and customer code access compared

	Modules reusable between environments	Modules reusable between customer infrastructures	Modules reusable between customers	Customer has access to customer specific code	Customer has access to reusable module code
Workflow 1	x	x	x		
Workflow 2	x	x	x	x	
Workflow 3	x	x	x	x	x

Finally, Table 6 compares the Workflows 1-3 based on their module reusability, and how they meet the requirements presented in Table 2. As the three first columns of Table 6 show, all of the workflows allow reusing Terraform modules between environments, customer infrastructures and customers. This is not the case with the current workflow or the improved workflow (see Table 3).

The last two columns of Table 6 indicate that Workflows 1-3 meet the access control criteria presented in Table 2, which was the goal of this case study. This customer access control correlates with the customer experience of how transparently the service is operated.

As shown in Table 6, giving customers more access to the Terraform source code means more valuable and transparent services to customers. However, more access to Terraform source leads into several restrictions and more complicated process. A full combined comparison of all the workflows and their technical details is reported in Appendix A.

9 Conclusions

This thesis analyses how MSPs can leverage IaC tools. It reviews MSP modularization and cloud infrastructure management literature and gathers the best IaC practices and Terraform features applicable to MSP settings. The literature review clearly indicates that MSPs operating in cloud environments can utilize IaC tools. However, leveraging IaC tools is more challenging for MSPs than for typical SaaS companies due to some specific challenges related to the required isolation between customers and the use of multiple VCSs. To the best of my knowledge, the question of how MSPs can benefit from IaC has not been studied very widely so far.

In addition to the general IaC analysis, the aim of this case study is to investigate how to benefit from reusable Terraform modules across multiple customers. The analysis of the collaborating MSP's current Terraform modules suggests that module reusability has room for improvement. Utilizing reusable Terraform modules results in less code to maintain and less code duplication across MSP's repositories. Reusable modules may also lead to more efficient service delivery if a catalogue of tested modules is available for customers.

The new workflow proposals, i.e. Workflows 1-3, are consistent with the collaborating MSP's IPR requirements and they allow reusing Terraform modules with multiple customers. Each workflow has specific IPR requirements that dictate how much Terraform code access the customer gets. Each workflow has a reference implementation which was tested in practice and used for gathering feedback from stakeholders. The reference workflow implementations were improved iteratively until they met all the requirements discovered through the stakeholders' feedback.

The key feature of Workflows 1-3 is the way that VCS is leveraged. Individual module versioning is not possible in the current workflow. However, Workflows 1-3 allow individual versioning of the modules and referring to a specific module version in different customer infrastructures. In Workflows 1-3, separating the live code and the module code into separate repositories opens up potential for improving development practices.

Comparison of the proposed workflows reveals that giving customers more access to the Terraform modules leads to several restrictions and more complicated workflows. Because of this tradeoff, from a technical point of view, it is not straightforward for the collaborating MSP to select the most suitable workflow from the three proposals. Thus, the decision could be based on the customers' expectations on the amount of Terraform code access they should have. Some new practices based on Workflows 1-3 have already been adopted by the collaborating MSP. The new practices have reduced the amount of duplicated code in the customer repositories which has simplified infrastructure maintenance.

Later, after a new workflow has been selected and a catalogue of reusable modules has been established, it will be worth exploring whether the reusable modules can be used as building blocks for a totally new set of managed services. Moreover, it might be beneficial to introduce new metrics quantifying the benefits of reusable modules. One such metric is to track the percentage of value and sales based on reusable modules, as proposed by the existing research [1].

This case study focuses on the specific requirements of the collaborating MSP. However, similar solutions can potentially be applied in other organisations, if IaC is used with multiple VCSs and especially if access control requirements are relatively complex. Workflows 1-3 are entirely VCS based, which means that they enable sharing a wide range of versioned files, not just Terraform code, with the customers.

References

[1] N. Leon and A. Davies, "Managed service paradox," *IBM Systems Journal,* vol. 47, no. 1, pp. 153-166, 2008.

[2] J. Wood and T. Lah, "The Case for Managed Services," in *Technology-as-a-Service Playbook*, Technology Services Industry Association, 2016, pp. 283-306.

[3] OryxAlign, *An Insight Into IT Service Delivery: Traditional Break-Fix vs A Managed Service,* OryxAlign, 2017.

[4] Y. Brikman, Terraform: Up & Running, 2nd Edition: Wrtiting Inrastructure as Code, O'Reilly Media, Inc., 2019.

[5] K. Morris, Infrastructure as Code: Dynamic Systems for the Cloud Age, 2nd edition, O'Reilly Media Inc., 2020.

[6] K. Ruddy, "Announcing HashiCorp Terraform 1.0 General Availability," HashiCorp, 8 June 2021. [Online]. Available: https://www.hashicorp.com/blog/announcing-hashicorp-terraform-1-0-general-availability. [Accessed 11 March 2022].

[7] Pulumi, "Available Versions," [Online]. Available: https://www.pulumi.com/docs/get-started/install/versions/. [Accessed 2022 March 11].

[8] Upbound, "https://crossplane.io/," [Online]. Available: https://crossplane.io/. [Accessed 29 March 2022].

[9] M. Hashimoto, "Terraform announcement," HashiCorp, 28 July 2014. [Online]. Available: https://www.hashicorp.com/blog/terraform-announcement. [Accessed 2022 March 3].

[10] M. Hashimoto and A. Dadgar, "A New Chapter for HashiCorp," HashiCorp, 09 December 2021. [Online]. Available: https://www.hashicorp.com/blog/a-new-chapter-for-hashicorp. [Accessed 2022 March 11].

[11] GitHub, "Terraform license page on GitHub source code," HashiCorp, [Online]. Available: https://github.com/hashicorp/terraform/blob/main/LICENSE. [Accessed 14 March 2022].

[12] HashiCorp, "Terraform Language Documentation," [Online]. Available: https://www.terraform.io/language. [Accessed 11 March 2022].

[13] S. Chacon and B. Straub, "Pro Git book," git-scm, 2014. [Online]. Available: https://git-scm.com/book/en/v2. [Accessed 20 June 2022].

[14] HashiCorp, "Manage Terraform Versions," [Online]. Available: https://learn.hashicorp.com/tutorials/terraform/versions. [Accessed 17 March 2022].

[15] HashiCorp, "Terraform plugin best practices: Versioning and Changelog," [Online]. Available: https://www.terraform.io/plugin/sdkv2/best-practices/versioning. [Accessed 17 March 2022].

[16] HashiCorp, "Terraform Cloud and Terraform Enterprise documentation," [Online]. Available: https://www.terraform.io/cloud-docs. [Accessed 11 March 2022].

[17] HashiCorp, "Terraform Enterprise," [Online]. Available: https://www.terraform.io/enterprise. [Accessed 11 March 2022].

[18] HashiCorp, "Terraform CLI: Terraform Cloud Settings," [Online]. Available: https://www.terraform.io/cli/cloud/settings. [Accessed 11 March 2022].

[19] HashiCorp, "Terraform Cloud web application," [Online]. Available: https://app.terraform.io/app/. [Accessed 11 March 2022].

[20] A. Sydykov, "HashiCorp Compliance Program Updates," HashiCorp, 14 December 2020. [Online]. Available: https://www.hashicorp.com/blog/the-hashicorp-compliance-program-updates. [Accessed 11 March 2022].

[21] HashiCorp, "HashiCorp Cloud SLA," July 2021. [Online]. Available: https://cloud.hashicorp.com/sla. [Accessed 11 March 2022].

[22] HashiCorp, "Terraform Pricing," [Online]. Available: https://www.hashicorp.com/products/terraform/pricing. [Accessed 11 March 2022].

[23] Atlantis, "Atlantis home page," Atlantis, [Online]. Available: https://www.runatlantis.io/. [Accessed 1 July 2022].

[24] Spacelift, "Spacelift home page," Spacelift, [Online]. Available: https://spacelift.io/. [Accessed 1 July 2022].

[25] HashiCorp, "setup-terraform GitHub page," HashiCorp, [Online]. Available: https://github.com/hashicorp/setup-terraform. [Accessed 1 July 2022].

[26] Amazon.com, Inc., "Managing access controls: Creating an S3 bucket access policy," [Online]. Available:

https://docs.aws.amazon.com/transfer/latest/userguide/users-policies.html. [Accessed 11 March 2022].

[27] "Terragrunt home page," Gruntwork.io, [Online]. Available: https://terragrunt.gruntwork.io/. [Accessed 14 March 2022].

[28] GitHub, "Terragrunt license page," Gruntwork.io, [Online]. Available: https://github.com/gruntwork-io/terragrunt/blob/master/LICENSE.txt. [Accessed 14 March 2022].

[29] T. Holmes, "Terraform Mono Repo vs. Multi Repo: The Great Debate," HashiCorp, 2021 January 28. [Online]. Available: https://www.hashicorp.com/blog/terraform-mono-repo-vs-multi-repo-the-great-debate. [Accessed 16 March 2022].

[30] "Issue: Fetches same module multiple times," Github, 30 August 2016. [Online]. Available: https://github.com/hashicorp/terraform/issues/8555. [Accessed 23 March 2022].

[31] git-scm, "Git remote documentation reference," [Online]. Available: https://git-scm.com/docs/git-remote. [Accessed 6 5 2022].

[32] J. R. Feagin, A. M. Orum and G. Sjoberg, A Case for the Case Study, Chapel Hill, NC: University of North Carolina Press, 1991.

A. A full workflow comparison table

Table 7. All workflows compared by module reusability, customer code access and technical details

	Modules reusable between environments	Modules reusable between customer infrastructures	Modules reusable between customers	Customer has access to customer specific code	Customer has access to reusable module code	Requires splitting modules into multiple repositories	Requires a process to keep sensitive information out of generic repositories
The current workflow				x	No reusable module code	No module repositories	
Improved workflow	x	x		x	No reusable module code		
Workflow 1	x	x	x				
Workflow 2	x	x	x	x			
Workflow 3	x	x	x	x	x	x	x