Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Infrastructure as Code eszközök alkalmazása oktatási környezetben

TDK dolgozat

Készítette:

Karsa Zoltán István

Konzulens:

Dr. Szeberényi Imre

2022

Tartalomjegyzék

Kivonat **i**

Abstract **ii**

Bevezetés **1**

1. Eszközök bemutatása **2**
- 1.1. CIRCLE felhőmenedzser 2
  - 1.1.1. Kialakulás 2
  - 1.1.2. CIRCLE Funkciók 2
  - 1.1.3. Modulok és felépítés 3
  - 1.1.4. Fejlesztések 4
- 1.2. Cloud-init 4
- 1.3. Infrastructure as Code 6
  - 1.3.1. Terraform 7
- 1.4. Kitekintés 12
- 1.5. Lehetőségek 13

2. Megvalósítás **14**
- 2.1. Cloud-init bevezetése 14
  - 2.1.1. Példa konfiguráció 16
- 2.2. REST-API 17
  - 2.2.1. Aszinkron műveletek kezelése 18
- 2.3. Terraform provider 18
  - 2.3.1. Golang kliens 18
  - 2.3.2. Sémák 19
  - 2.3.3. Állapotok kezelése 19
  - 2.3.4. Aszinkron hívások kezelése 20
  - 2.3.5. Az élet-ciklus modell bővítése 22
  - 2.3.6. Távoli állapottárolás 23

3. Használati esetek **24**
- 3.1. Terraform alkalmazási minták 24
- 3.2. Egyszerű VM készítése 24
  - 3.2.1. Tesztelés 26
- 3.3. Többgépes környezet 27
  - 3.3.1. Mérés 28
- 3.4. Dinamikus környezet építése 28
- 3.5. Adaptáció Jenkins-el 32

Összegzés **33**

Rövidítések **34**

Ábrák jegyzéke **35**

Irodalomjegyzék **35**

Kivonat

Dolgozatomban bemutatom az egyre inkább ismertté váló, nagy népszerűségnek örvendő Infrastructure as Code (IaC) technikát, ami a felhő infrastruktúrák, szolgáltatások, adattárházak és különböző szoftvereszközök automatikus, és megismételhető kezelését teszi lehetővé gép-közeli konfigurációs fájlok segítségével. Ahogy a nevéből is sejthető, a különböző szolgáltatásokat programkódhoz hasonló fájlokkal írhatjuk le, amik aztán lefuttathatóak. Így ezen "programkódok" tekinthetők speciális szöveges szakterületi nyelvnek is, amivel ezeket a platformokat tudjuk programozni, sőt léteznek erre célra vizuális eszközök is.

A dolgozat fő fókusza az IaC technika oktatási és tudományos területen való alkalmazhatósága. Ennek megfelelően bemutatom az egyetemen fejlesztett CIRCLE felhőmenedzserhez készített bővítményeket, azok kapcsolatát a rendszerrel. Ilyen például a már virtuális gép példányok automatikus konfigurálást lehetővé tevő, szabványos cloud-init implementálása a CIRCLE rendszerben. Példákkal illusztrálom azok oktatásban és gyakorlatban történő alkalmazhatóságát. Bemutatom továbbá azokat az eszközöket és kutatási irányokat, melyek relevánsak az IaC használatában és fejlesztésében. Valamint bemutatok egy működő megvalósítást is, aminek fejlesztése a Terraform eszköz segítségével történt. A Terraform egy keretet biztosít a saját Infrastructure as a Service (IaaS) környezetünk integrálásához, vagy más szolgáltatások ilyesfajta publikálásához. Elég egy úgynevezett köztes provider modult elkészíteni, ami tartja a kapcsolatot a CIRCLE rendszerrel. Cserébe nem kell foglalkozni alacsony szinten az állapotok nyilvántartásával, szálak kezelésével ...

A dolgozatomban vizsgálom ezen eszközök előnyeit és hátrányait, alkalmazhatóságát oktatási területen, illetve teljesítménybeli szempontokat is figyelembe veszek.

Abstract

In this paper, I will introduce the increasingly popular Infrastructure as Code (IaC) technique, which enables the automatic and repeatable management of cloud infrastructures, services, data warehouses, and various software tools using machine-to-machine configuration files. As the name implies, various services can be described by program-like files that can then be executed. Thus, these 'program codes' can also be considered as a domain specific language for programming these platforms, and there are even visual tools for this purpose.

The main focus of this thesis is the applicability of IaC techniques in education and science. Accordingly, I will present the extensions to the CIRCLE cloud manager developed at the university and their relation to the system. For example, the implementation of a standard cloud-init in the CIRCLE system that allows automatic configuration of already existing virtual machine instances. I will illustrate their applicability in education and practise with examples.

Furthermore, I will demonstrate tools and research directions relevant to the use and development of IaC. I will also present a working implementation developed using the Terraform tool.

Terraform provides a framework to integrate our own Infrastructure as a Service (IaaS) environment or to publish other services in this way. It is sufficient to create a so-called "intermediate provider" module that stays in connection with the CIRCLE system. In return, you don't have to deal with low level statefulness, thread management, etc.

In my thesis, I will examine the advantages and disadvantages of these tools, their applicability in the educational field, and their differences in performance.

Bevezetés

A dolgozat mind elméleti és mind gyakorlati szinten bemutatja az IaC eszközök alkalmazását az oktatási és tudományos/kutatási területeken és a CIRCLE felhőmenedzsert, amihez az IaC modul készült.

A dolgozat 1. fejezete körbejárja a használt technológiákat illetve megemlít több projektet is az IaC eszközök alkalmazásával kapcsolatban. Bemutatja az egyetemen fejlesztett CIRCLE felhőmenedzser rendszert, milyen funkciókkal rendelkezik és milyen modulokból épül fel. A CIRCLE rendszerben implementálásra került cloud-init szabványos konfigurációs lehetőséget is bemutatom ebben a fejezetben: Mire használható, hol érhető el, milyen megoldásokat nyújt. Továbbá bemutatok néhány felhőszolgáltatót, náluk hogy érhető el a cloud-init. A dolgozatban bemutatom az Infrastructure as Code eszközök két csoportját: configuration management (CM) és orchestration eszközöket. Az implementációhoz használt Terraform eszköznél kifejtem az infrastruktúra modellezéséhez szükséges fogalmakat, modulokat és technológiákat (például erőforrás sémák és tervgráf).

A 2. fejezet az implementációt mutatja be: Milyen lehetőségek voltak a rendszer fejlesztésekor és miért úgy került kifejlesztésre az adott funkció. Szó esik a cloud-init CIRCLE-hez tartozó megvalósításáról valamint a Terraform modul implementációjáról is. A Terraform modul fejlesztéséhez szükséges REST-API bővítésről is írok, valamint arról milyen extra funkciókat kellett még bevezetni a praktikus használathoz (pl. perzisztens diszkek) a CIRCLE rendszerben.

Az utolsó fejezet egyszerűbb alkalmazási példákat mutat be: Általános használat például szoftverkomponensek folyamatos integrációjához és teszteléséhez, laboralkalmak és vizsgák automatikus kiszolgálásához tartozó lehetőségek. A CIRCLE rendszert ugyanakkor tudományos és kutatási területeken is alkalmazzák, az ehhez tartozó lehetőségek is bemutatom összegezve az előnyöket és hátrányokat.

A dolgozatban feldolgozom a cloud környezethez kapcsolódó szakirodalmat is, általánosan az egyes technológiákhoz kapcsolódó cikkeket, és az IaC alkalmazásának lehetőségeit is ismertetem más rendszerek esetén.

1. fejezet

Eszközök bemutatása

Ebben a fejezetben röviden bemutatom az egyetemen fejlesztett CIRCLE[1] felhőmenedzsert valamint az IaC eszközök hátterét, és egy konkrét eszközt is (Terraform).

1.1. CIRCLE felhőmenedzser

1.1.1. Kialakulás

A széles körben elterjedt virtualizáció és a különböző laboralkalmakra, gyakorlatokra megfelelő háttérinfrastruktúra követelményeinek folyamatos változása életre hívta a CIRCLE rendszer kifejlesztését. A kezdetleges változatot 2012-ben fejlesztették ki, ami először csak a BME VIK Irányítástechnika és Informatika Tanszéken működött. A cél elsősorban az volt, hogy megfelelő háttérinfrastruktúrát biztosítson a laborokra, illetve az informatikában kevésbé jártas oktatók is könnyen tudják használni, valamint egyszerűen elő tudjanak készíteni egy megfelelő környezetet a hallgatóknak.

A rendszert aktívan fejlesztették 2018-ig, amikor az infrastruktúra további adatközpontokkal bővült (füred, niif): így már elérhetővé vált a teljes hallgatói kar kiszolgálása a VIK-en. Testvérrendszereket is hoztak létre Győrben, Miskolcon és Glasgow-ban. [34]

1.1.2. CIRCLE Funkciók

Mivel elsősorban általános célú felhőmenedzserről beszélünk, alapvető funkciók a virtuális gépek (VM - virtual machine) készítése, indítása, leállítása, annak használata, törlése stb. Valamint elengedhetetlen a működés (ütemező-algoritmusok) és a felhasználók, adminisztrátorok szempontjából az egyes fizikai/host és vendég gépek monitorozása, az erőforrások kihasználásának nyomon követése. Ezeket az információkat a webes felületen nyomon tudjuk követni. [2]

A rendszer lehetőséget nyújt VM sablonok készítésére is: Egy sablon egy már elkészült virtuális gép pillanatképét tárolja. Ebből a képből pedig új virtuális gépek indíthatók a kiinduló virtuális gépre előkészített szoftverkörnyezettel és konfigurációval. Így az oktatók a speciális szoftvereket könnyen meg tudják osztani a hallgatókkal, illetve tanórán használt különböző szoftvercsomagok telepítését (ami akár félévről félévre változhat) nem kell elvégezni a fizikai gépen, így csökken a rendszergazdák, operátorok terhelése is.

A sablonozás a qemu által nyújtott snapshot funkcióval van megoldva, amit aztán az újabb változatok backing fájlként használhatnak (copy-on-write). [16]

A VM-ek egy belső hálózathoz kapcsolódnak. A külső elérés alapvetően ssh/rdp protokollon keresztül portforward technológián biztosított, így más portok kezelése is ilyen módon történik, a rendszer a ”kinyitott” portokat a fizikai gép egy szabad randomizált

[1] Cloud Infrastructure for Research Computing and Laboratory Environment

portjához társítja, ami aztán elérhető a világhálóról. A hálózati konfigurációhoz tartozik még például az ip címek, VLAN-ok (Virtual Local Area Network) és domain-nevek regisztrálása is.

A felhasználók jogaiknak kezelése ACL (Access-control list) alapú: A virtuális gépeknél, sablonoknál megkülönböztetünk tulajdonosokat, operátorokat és felhasználókat. Nyilván ezen csoportok más-más jogosultságokkal rendelkeznek.

Mivel elsősorban oktatási környezethez készült rendszerről van szó, a felhasználók a használat után nem fizetnek, úgy mint például egy üzleti felhős szolgáltatásnál. Ugyanakkor szükséges valamilyen módon ösztönözni az észszerű felhasználást, hiszen az erőforrások sajnos nem végtelenek. Ennek megfelelően egy élet-ciklus modell alapján kezeljük a virtuális gépeket. Így a futó virtuális gép felfüggesztésre (alvó állapotba) kerül, ha lejár az erre vonatkozó időkeret, valamint töröljük, ha a megsemmisítésre vonatkozó határidő elérkezik. A felhasználó bármikor meghosszabíthatja az időkeretet, illetve értesítést is kap, ha az idő intervallum lejárna. Az üzleti célú szolgáltatásoknál is, ahogy a CIRCLE esetén is a felhasznált memória, vCPU, háttértár, egyszerre indítható gépek száma is korlátozható az egyes felhasználóknál.

1.1.3. Modulok és felépítés

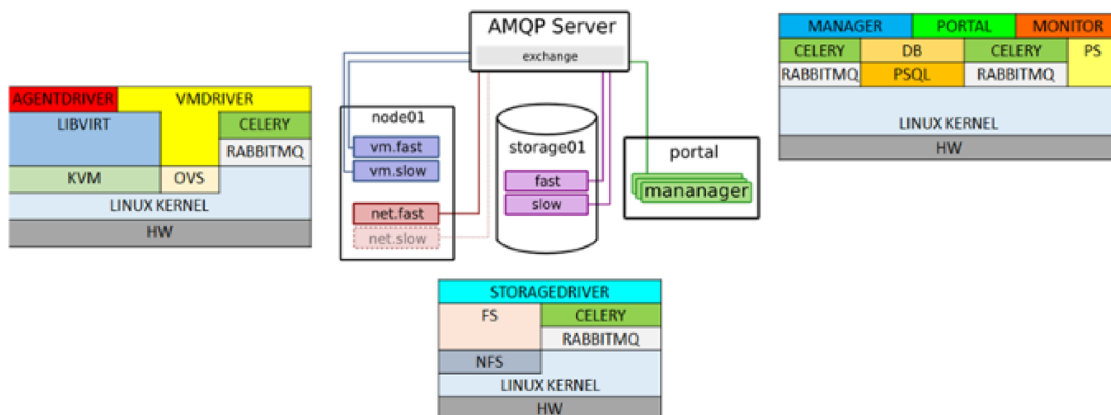
A virtualizációt libvirt API segítségével KVM/QEMU biztosítja, a webes felhasználói felület python nyelven írodott, a Django ([12]) keretrendszer segítségével. Ez a modul kezeli különböző felhasználói jogosultságokat (ACL) is, illetve nyílvántartja az adatközpont paramétereit (felhasznált lemezterület, vm-ek száma, sablonok kezelése stb.). Mivel ezen szerver, illetve a virtuális gépeket futtató szerver akár teljesen egymástól elszeparáltan is megjelenhet, valamint a műveletek viszonylag hosszú lefutása miatt, szükséges egy köztes réteg: Ehhez üzenet-alapú kommunikációs (AMQP-Advanced Message Queuing Protocol [33] - RabbitMQ) protokollt használ celery taszkok segítségével. Ezen kívül még érdemes megemlíteni a lemezképek, virtuális diszkek kezelésére használt strorage drivert is, a tárolót az egyes node-ok NFS (Network File system) segítségével érik el. (1.1. ábra)

A virtuális gépek állapotát a monitor kliens tartja nyilván. Az agent-drivert a virtuális gépek konfigurálását valósítja meg (felhasználók létrehozása, jelszavak beállítása stb.).

A rendszerben létrejövő virtuális hálózatot Open vSwitch [20] segítségével építjük fel, melynek konfigurációját folyamatosan frissíti a menedzser modul ahogyan a virtuális gépek létrejönnek, mozognak ill. megszűnnek. Fontos kiemelni, hogy a virtuális gépek között egyedi VLAN-ok is kialakíthatók hallgatói mérések számára, vagy éppen egy virtuális klaszter számára. A virtuális gépek és a felhasználók lokális gépei közötti fájlcserét egyrészt a remote desktop adta lehetőségek segítik ill. minden felhasználónak rendelkezésére áll egy korlátozott méretű perzisztens tároló, ami egyszerűen csatolható a felhasználó virtuális gépéhez ill. drag-and-drop módszerrel kezelhető felhasználó lokális gépéről.

A CIRCLE rendszer authentikációs modulja az egyetemi SSO [2] (eduID) rendszert használja, ami extra adatokat is képes szolgáltatni a tanulmányi rendszerből. Így a felhasználó sikeres azonosítása után a rendszer képes megkülönböztetni az oktatókat és a hallgatókat ill. olyan adat is rendelkezésre áll, hogy az adott hallgató milyen tantárgyakat hallgat az adott félévben. Ennek ismeretében egy adott tárgy oktatója nagyon egyszerűen be tudja állítani, hogy az általa létrehozott sablont azok használhassák, akik a tantárgyat hallgatják. Nagy létszámú (4-500 fő) alaptárgyak esetében ez a beállítás hallgatónként munkaigényes lenne.

[2] SSO: Single Sign On, eduID: `https://eduid.hu/hu`



1.1. ábra. CIRCLE rendszer felépítése

1.1.4. Fejlesztések

Az évek során nagyon sok kissebb-nagyobb fejlesztés történt a rendszerrel kapcsolatban. Ilyen például a szabványos OCCI (Open Cloud Computing Interface) interfész támogatása, vagy a rendszer teljes újragondolása (RECIRCLE), ezek többnyire szakdolgozat vagy diplomaterv szintjén maradtak.

Jelenleg majdnem elkészült a rendszer python3-as klónja (eredetileg python2-es verzióban készült), amelyből egy tesztrendszer működik az Irányítástechnika és Informatika tanszéken, ehhez a tesztrendszerhez készült - a megfelelő bővítésekkel, a dolgozatban később bemutatásra kerülő IaC modul, illetve a cloud-init támogatása.

1.2. Cloud-init

A felhőmenedzserek egyik feladata a virtuális gépek indítása, amikor az új gépeket fel kell paraméterezni, konfigurálni, ilyen például az ip-címek, jelszavak, felhasználói fiókok beállítása. Erre a feladatra széles körben elfogadott módszer a cloud-init. Ezt a módszert a Linux rendszerek kezdeti beállításához fejlesztették ki, de mára kvázi szabvánnyá vált. Ma már a Windows operációs rendszerekhez is kínálnak olyan képfájlokat, amiben a cloud-init kompatibilis megoldás telepítve van. Sőt egyes Linux disztribúciók installló folyamata aktívan használja a cloud-init-et nem felhős környezetben is. Korábban egy új rendszernél telepíteni kellett azt egy megfelelő diszkre, és ott be kellett állítani a különböző felhasználókat, partíciókat és más adminisztratív dolgokat, most szinte rögtön használható egy megfelelő cloud-init konfigurálással.

A működés lényege, hogy a rendszer első induláskor egy külső eszközről (usb, romdrive, más partíció, hálózat) beolvassa a konfigurációt, és lefutnak a konfigurációnak megfelelő szkriptek. A rendszerek különbségeit python fedi el, a cloud-init ebből a python kódból hívja a rendszerspecifikus parancsokat. Ez a konfiguráció így tekinthető speciális IaC kódnak is.

A használathoz így egy cloud-init képfájl, valamint egy megfelelő konfigurációs állomány kell: Egy meta-data, vendor-data, network-config és egy user-data állomány, mindegyik YAML formátumú, a meta-data és vendor-data a cloud-infrastruktúra üzemeltetőhöz szorosan kapcsolódó beállításokat tartalmazza, míg a network-confignál a hálózati beállításokat (ip, vlan, routing stb.), valamint az user-data-ban a VM tulajdonosa tudja a különböző beállításokat elvégezni. [6]

A 1.1. kódrészlet egy user-data állományt mutat: Egy ”cloud” felhasználót konfigurálunk sudo jogosultságokkal, beállítva az alapértelmezett shell-t és jelszót.

```
#cloud-config
users:
  - name: cloud
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    groups: sudo
    shell: /bin/bash
    ssh_pwauth: True
    chpasswd: { expire: False }
    lock-passwd: false
    passwd: "secret"
```

1.1. Kód. Példa cloud-init user-data állomány

A cloud-init fontos a szabványos automatizációhoz, így elengedhetetlen feltétel minden IaaS[3] szolgáltatást megvalósító rendszernél. Valamint nélkülözhetetlen így az IaC eszközök megfelelő alkalmazásához is. Így ezen funkció implementálása elsődleges fontosságú a CIRCLE rendszer esetén is.

Emellett kiváltja a korábban említett agent-driver-t is, hiszen segítségével pontosan az agent feladait tudjuk testre szabni, és mivel nagy múlt és rengeteg extra modul van mögötte, nagyon sok beállítási lehetőség lehetséges.

A cloud-init beállítási lehetőségeit jól tükrözi, hogy rengeteg modul is elérhető hozzá, viszont ezeket a modulokat nem minden operációs rendszer támogatja maximális szinten. Ilyen lehetőség például a `resizefs` modul, ami automatikusan átméretezi a root fájlrendszert, ha érzékeli a virtuális diszk átméretezését. Ez a modul a legtöbb rendszeren elérhető. Vagy ilyen modul a `phone_home`, ami a rendszer boot után egy POST kérést küld a megadott címre az adott virtuális gép adataival. [6]

Boot folyamat cloud-init esetén

A cloud-init szabványos kezeléséhez a boot folyamat során más-más folyamatok futnak le. Az első szakasz (Generator) a cloud-init állapotának ellenőrzésére (engedélyezve van vagy sem) szolgál. Ezután következik a Local szolgáltatás futása, ami felderíti a helyi adatforrásokat a későbbi beállításhoz, például csatolt diszken, vagy valamilyen hálózati elemen keresztül érhető-e el az adatforrás, vagy bizonyos cloud szolgáltatók esetén vannak alapvető konfigurációk, amik betölthetőek. Illetve beállítja a hálózati konfigurációt, ha ez az első tiszta cloud-init boot. A következő Network szakaszban a `cloud_init_modules`-ba tartozó folyamatok futnak le, például diszk felcsatolása és minden bootoláskor lefut a `bootcmd`-hez magadott parancsok. Az utolsó két szakaszban (Config, Final) lefutnak a megfelelő modulokhoz tartozó folyamatok, ilyen lehet például a runcmd parancsok futtatása vagy CM pluginok (chef, ansible, salt - 1.3. szakasz) elindítása. [6]

Cloud-init más felhőszolgáltatóknál

Mivel a network-config, meta-data és vendor-data állományok szorosan kötődnek a felhőszolgáltatókhoz, csak a user-data állomány testreszabása ad lehetőséget a legtöbb szolgáltató.

Azure szolgáltatónál ([7]) például Azure CLI-ből való indítás esetén egy parancssori paraméterben lehet megadni az user-data állományt. A cloud-init képes lemezképeket a szolgáltató folyamatosan frissíti, csak "belsős" lemezkép használható. Továbbá a cloud-init-nek köszönhetően alapértelmezetten a szolgáltató (nem szenzitív) metrikákat gyűjt az indított virtuális gépről.

```
az vm create \
```

[3] Szolgáltatási rétegek szerint: IaaS (Infrastructure), PaaS (Platform), SaaS (Software as a Service), stb.

```
--resource-group myResourceGroup \
--name centos74 \
--image OpenLogic:CentOS-CI:7-CI:latest \
--custom-data cloud-init.txt
```

Vagy például DigitalOcean szolgáltatónál, ha olyan virtuális gépet (droplet) választunk, ami cloud-init képes, akkor egy webes felületen testre tudjuk azt szabni.

1.3. Infrastructure as Code

A folyamatos integráció, tesztelés miatt, valamint hogy mindig a lehető leggyorsabban a legfrissebb szoftvert szállítsuk a megrendelőnek (continuous integration, continuous delivery - CI/CD), előtérbe kerültek az automatizációt szolgáló eszkezök. Illetve mivel az egyes erőforrásköltségek a különböző szolgáltatóknál más és más, megjelentek az egyes cloud-szolgáltatókon átívelő projektek és cloud brókerek is, ezt nevezik multi-cloud-nak. [31] Ez egy olyan közös programozási platformot hívott életre, amivel az automatizáció biztosított, illetve a szolgáltatók közötti átjárhatóság is. Az Infrastructure as Code eszközök elsősorban felhasználási területükön (privát, publikus, hibrid felhő [32]), módjukban és funkcióikban térnek el egymástól.

Léteznek a kifejezetten konfigurációs céllal készített eszközök (Configuration Management - CM), amelyekkel automatikusan konfigurálhatók és telepíthetők távoli példányokon különböző szoftvereszközök. Például egy egyszerű webszervert szeretnénk csinálni, akkor a megfelelő csomagokat kell telepíteni (pl. nginx) és azokat a kívánt módon kell beállítani (.conf fájlok). Ebbe a körbe tartozik például a Chef, SaltStack és Ansible is.

A másik csoport pedig az infrastruktúra-kezelő (orchestration) eszközök, amelyek a kiszolgáló példányokat (virtuális gépek, vagy akár személyre szabott fizikai gépek, tárak stb.) hozzák létre és kezelik. Itt a hangsúly az összetett környezet (VLAN, VPC-Virtual Private Cloud, VM-ek, adattárak) kialakításán van. Bár megfelelő kiegészítésekkel ezek alkalmasak lehetnek CM feladatok ellátására, de teljesen más céljuk van. Ilyen orchestration eszköz a Terraform és a Pulumi.

```
import pulumi
import pulumi_azure as azure

example_resource_group = azure.core.ResourceGroup("example—resources", location="West Europe")
example_virtual_network = azure.network.VirtualNetwork("example—network",
location=example_resource_group.location,
resource_group_name=example_resource_group.name,
address_spaces=["10.0.0.0/16"])
```

1.2. Kód. Pulumi példakód Azure szolgáltatóhoz

Ezen eszközök imperatív vagy dekleratív felépítésűek lehetnek.Terraform esetén dekleratív paradigmáról beszélünk, míg Pulumi és a fenti CM eszközök közül a Chef és Ansible eszközöknél imperatív szemléletről. Dekleratív (pl. SQL, Prolog) szemlélet esetén az eredményt határozzuk meg, míg imperatívnál utasításokat adunk, "parancsolunk", hogy az eredményt hogy kapjuk meg. A kódrészletben (1.2), valamint a következő alfejezetben (1.4. kód) egy Pulumi (Python-kód) és egy Terraformos változatot látunk ugyanazon környezet felépítésére.

Az orchestration eszközökben közös, hogy a létrehozott erőforások, infrastruktúra állapotát tárolják, és a felhasználói igények alapján változtatják ezt az állapotot: létrehoznak erőforrásokat, megszűntetnek, átméreteznek stb.

Az IaC eszközök alkalmazásának előnyei: az infrastruktúra felépítése és "lebontása", skálázása automatizálható, az különösen fontos, ha használat után fizetünk a szolgáltató-

nak. A kód és az infrastruktúra könnyen megoszthatóvá válik, és a kód részei más helyeken is újrafelhasználható.

Az IaC eszközök alkalmazásának célja az oktatási és tudományos környezetben elsősorban a már korábban felvetett automatizáció: Léteznek speciális laborok vagy vizsgák, amikre elő kell készülni a megfelelő eszközökkel, azaz virtuális gépekkel, majd azoknak a labor végeztével meg kell szűnnie.

Ezt a speciális funkciót a CIRCLE rendszer maga is biztosítja: Tömbösített indítással a hallgatók egy csoportjának virtuális gép indítható egy már létező sablonból. Cél ezen funkció kiterjesztése a Terraform segítségével is.

Emellett nem elhanyagolhatók a hagyományos CI/CD funkciók: Így lehetőség lesz a különböző projekteknek "házon belül" tartani a tesztelését. És nem kell külsős szolgáltatóknak fizetni ezért. Valamint az IaC alkalmazása miatt csökken a kiszolgáló szerver terheltsége is, hiszen nem kell ekkor a teljes HTML oldalt legenerálni egy kérés kiszolgálásánál, elég a JSON üzeneteket előállítani.

A felhő infrastruktúra kiépítéséhez léteznek vizuális modellező eszközök. Mivel a szolgáltatók általában heterogén felhő infrastruktúra metamodellel rendelkeznek, nehéz a helyes vizuális modellezés is. Ilyen modellező eszköz például az ARGON projekt (An infRastructure modellinG tool for clOud provisioNing) [25]

1.3.1. Terraform

HCL szintaxis

A HCL egy szakterületi nyelv (domain-specific language - DSL), amivel dekleratívan tudjuk leírni a szükséges környezetet: HashiCorp Configuration Language. Szintaxisa hasonlít a json-ra, előnye hogy verziókezelhető, illetve ember számára is könnyen értelmezhető. Erre egy példa látható a következő alfejezetben (1.4. kód).

A HCL nyelv lexer szabályai a már más nyelvekben megszokott tokenek mellett a különböző blokkokhoz tartozó kulcsszavakból állnak: ilyen például a 'provider' vagy 'resource' szavak.

A parser szabályok a szintaktikai elemzés folyamatát szabályozzák. Ez esetben ez azt jelenti, hogy az egyes konstrukciók, blokkok milyen további alblokkokból és tokenekből épülnek fel. A lenti példában például a Parsing Expression Grammar (PEG, [14]) segítségével megadott szabályok egy részét látjuk. A Noam Chomsky által életre hívott 4 nyelvosztály [21] inspirálta a PEG kialakulását: az elsősorban környezetfüggetlen (CFG) és reguláris kifejezéseket (RE) előszeretettel alkalmazzák programnyelvek szintaxisának leírására, de az ezzel való kifejezés nehéz. Például CFG esetén nem determinisztikus értelmezés előfordulhat, míg ezt prioritással (amelyik szabály előbb szerepel, azt kell alkalmazni) oldja fel a PEG. [14]

```
1 ConfigFile   = Body;
2 Body         = (Attribute | Block | OneLineBlock)*;
3 Attribute    = Identifier "=" Expression Newline;
4 Block        = Identifier (StringLit|Identifier)* "{" Newline Body "}" Newline;
5 OneLineBlock = Identifier (StringLit|Identifier)* "{" (Identifier "=" Expression)? "}"
      Newline;
```

1.3. Kód. PEG parser szabályok a HCL nyelvhez

A fenti példában így egy szöveges fájl törzsében lehet vagy attribútum, többsoros blokk vagy egysoros blokk (|) és ezekből tetszőleges számú (∗).

Gráfok

A Terraform Core csomag az állapotok nyilvántartására és az erőforrások kezelésére egy függőségi gráfot épít és tart fenn. Egy függőségi gráf csomópontokból és azok között futó irányított élekből áll.

A gráf konstruálása a következő módon történik: Adott X és Y erőforrásoknál akkor fut $X \to Y$ irányított él, ha X valamilyen módon függ Y-tól. Például X egyik bemeneti attribútuma Y egy számított értéke lenne.

Legyen a konfiguráció következő: Azure szolgáltatónál kell egy erőforráscsoportot (resource group) létrehozni, és a csoportban egy virtuális hálózatot (VPN - Virtual Private Network) a megfelelő paraméterekkel, ezt a következő módon kell megcsinálni Terraform esetén: Szükséges a provider (Azure) beállítása és konfigurálása, valamint a fenti erőforrások megadása.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.0.0"
    }
  }
}
provider "azurerm" { /* beállítások */ }
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}
resource "azurerm_virtual_network" "example" {
  name                = "example-network"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  address_space       = ["10.0.0.0/16"]
}
```

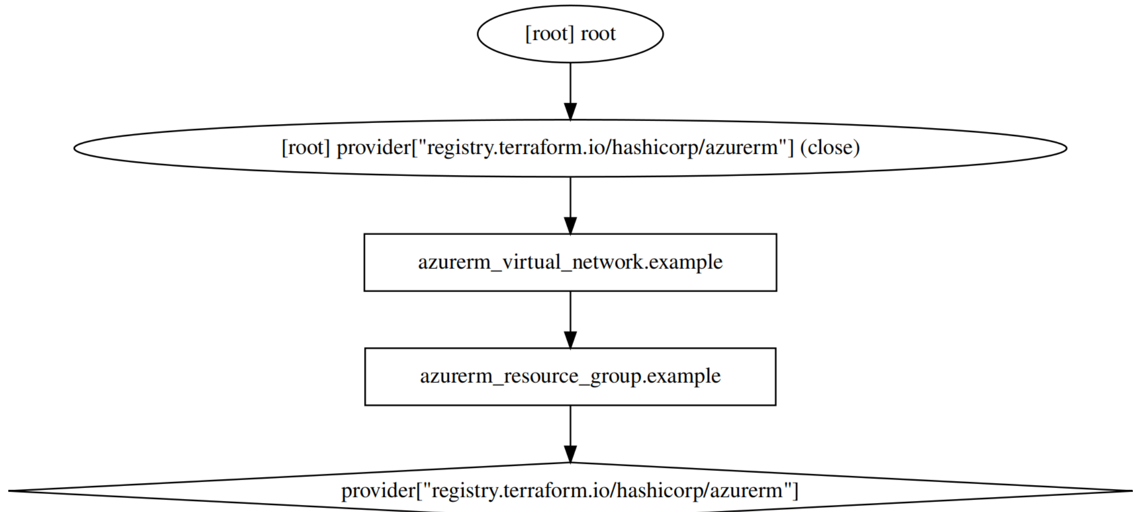
1.4. Kód. Példa HCL kód Azure szolgáltatóhoz

A fenti dekleratív szemléletű Terraform kóddal tudjuk ezeket az erőforrásokat létrehozni. Láthatjuk hogy az egyes erőforrások között különböző függőségek vannak: Ilyen például hogy a terraform core modul valamint provider konfigurálását kell legelőször elvégezni, minden más csak ezek után jöhet. Láthatunk még egy függőséget az "azurerm_resource_group" és a "azurerm_virtual_network" erőforrások között is.

A Terraform segítséget nyújt a végrehajtási terv grafikus megjelenítéséhez, a fenti kód tervét láthatjuk gráf-reprezentációval az ábrán (1.2. ábra - generált). A root speciális gyökér elem, ha a gyökér minden függése feldolgozásra került, akkor a terv végrehajtása véget ért és sikeres volt. A konfigurációs modulokhoz (provider-ek és plugin-ok) létezik a kapcsolatot felépítő, konfigurációs szakasz, és az azt lebontó, felszabadító rész. Az ábrán rombusz (provider azure) és ellipszis (provider azure close) jelöli ezeket. Ezek között láthatóak az erőforrásokat reprezentáló csomópontok. Mivel a VPN -nek szüksége van a resource group két paraméterére (name és location) egy irányított nyíl mutat közöttük.

Észrevehetjük, hogy egy speciális gráfról van szó, méghozzá egy irányított körmentes gráfnak (DAG-directed acyclic graph) kell lennie a végrehajtási tervnek, illetve izolált csomópontok nem fordulhatnak elő (legrosszabb esetben a root elemből közvetlenül is elérhető).

Bár behúzhatnánk egy közvetlen irányított élt a VPN és provider között is, a köztük lévő út már eleve meghatároz egy függést, így ezeket a Terraform elhagyja. Ezt nevezzük a gráf minimális tranzitív redukciójának: Egy $G(V, E)$ gráf minimális tranzitív redukciója azon $G'(V, E')$ gráf, hogy $E' \subseteq E$ és bármely G-beli irányított útra, van megfelelő út G'-

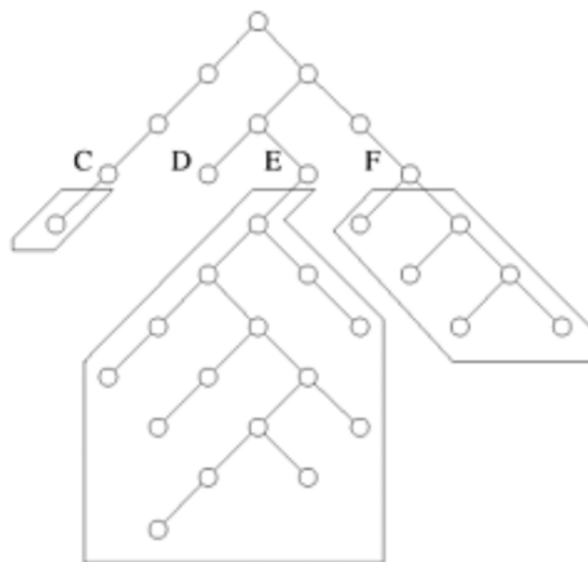


1.2. ábra. Terraform által generált terv-gráf Azure szolgáltatóval

ben (a kezdő és végpontok megegyeznek), hogy a lehető legkevesebb irányított élt vesszük be G-ből. Ez a G' gráf lineáris időben előállítható. [27]

A gráf reprezentáció előnye, hogy az egyes csomópontok, párhuzamosan feldolgozhatóak. Jó módszer lehet, hogy nyelőket (olyan csomópontok amelyekből nem indul irányított él, csak oda érkezik) keresünk minden egyes lehetséges iterációban, és ezeket a feladatokat végrehajtjuk párhuzamosan, aztán töröljük a csomópontokat a gráfból. Ezt addig ismételjük, amíg csak lehetséges. (Így megkapjuk a DAG egy topologikus rendezését is.) Csakhogy ennek kezelése kb. $c \cdot e$ komplexitású, valamint a párhuzamosítás miatt a gráfon való konkurens műveletek is nehézkesek. [1]

Helyette az általános bejárásához használt DFS (Depth First Search - mélységi keresés) algoritmust alkalmazzák párhuzamosan: A mélységi algoritmus azért lesz fontos, hiszen először azon csúcsokat kell meglátogatni (azaz feldolgozni), amik nyelők, vagyis további kimenő él nem indul belőle, különben nem az lenne a ”legmélyebb”. Minden fo-



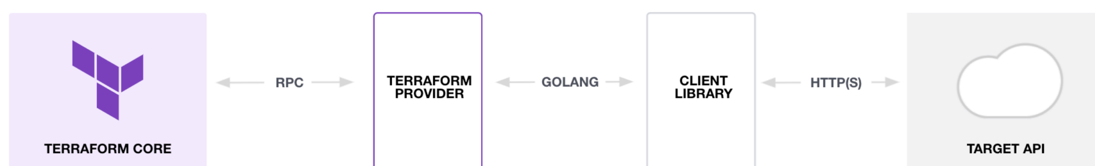
1.3. ábra. Diszjunkt részgráfok párhuzamos DFS-hez (forrás: [15])

lyamat kap egy diszjunkt részgráfot (1.3. ábra) ami elérhető a root csomópontból, és ott

futtatja a DFS algoritmust lokálisan. Persze itt előfordulhatnak olyan esetek, hogy az egyik folyamat részgráfjában sokkal kevesebb csúcs van, mint egy másiknál, így dinamikus terheléselosztást kell alkalmazni: Ilyen megoldás például hogy a túlterhelt folyamat a részgráfot - valamilyen súlyozás vagy más módszerekkel további részgráfokra bontja, és azt elküldi végrehajtásra más szabad folyamatoknak. [15]

Terraform Core és külsős szolgáltatók

A Terraform nagy előnye, hogy teljesen nyílt-forráskódú és bárki nagyon könnyen tud a maga saját szolgáltatásához egy megfelelő modult írni, hogy azt használni lehessen a Terraformon keresztül. Emellett a Terraform az állapotgráf, a párhuzamosítás és a változások kezelését maga a Core modul végzi, a szolgáltató fejlesztőjének nem kell ezeket kezelni, de el tudja érni a megfelelő API-n keresztül.



1.4. ábra. A Terraform működési váza (terraform.io - 2022.10.03.)

Ezt a működési elvet mutatja az ábra (1.4). A külső szolgáltatás fejlesztőjének feladata a Terraform Provider modul megfelelő implementálása Go nyelven: Ez a modul leírja az úgynevezett sémákat, a szolgáltatásnál milyen erőforrások vannak, azok milyen attribútummal rendelkeznek, melyek kötelezőek, mi a számított attribútumok (pl. ID, vagy valami hashelt jelszó stb.) típusa és így tovább. Ezen sémákat a Core modul felhasználja a gráf konstruálására és a mezők változásának detektálására. Maga az állapotváltozást a Core modul deríti fel, de a fejlesztő feladata a provider modulban a megfelelő reagálás rá. Bár nem kötelező, de az áttekinthetőség miatt célszerű egy golang kliens modul elkészítése a szolgáltatás API-jához.

3 fő séma típus van: szolgáltatói, adat és erőforrás. Az adat csak olvasási kontextusban fordulhat elő, míg erőforrást lehet kreálni, az állapotát lekérdezni és frissíteni, valamint megsemmisíteni. A szolgáltatói (schema provider) séma a szolgáltató rendszer (nem feltétlenül csak IaaS-hoz) paramétereit adja meg, mint például adatközpont címe és hitelesítői adatok.

Ennek megfelelően az implementációkor a következő függvényeket kell megvalósítani az RPC (Remote Procedure Call) hívások lebonyolításához:

- Datasource read - A beolvasott séma egy tipikusan szükséges tulajdonságával, pl. egyedi nevének segítségével beolvassuk a távoli objektum további adatait.
- Resource read - Hasonlóan mint az előző, csak erőforráson.
- Resource create - Létrehozzuk a megadott tulajdonságokkal az adott erőforrást, a számított értékeket kiírjuk.
- Resource update - A sémában változás történt, így le kell azokat kezelni.
- Resource destroy - Az erőforrás megsemmisítése.

A sémák paramétereit és azok tulajdonságait meg kell szabni: milyen típusa van, számított elem-e, kötelező vagy sem, stb. Ezeken kívül vannak speciális tulajdonságok:

Például a `ForceNew` bekapcsolása azt jelenti, hogy az erőforrás adott tulajdonságának megváltozásakor a Core modul megsemmisíti a korábbi erőforrást, és egy újat hoz létre.

```
&schema.Resource{
  Schema: map[string]*schema.Schema{
    "uuid": {
      Type:     schema.TypeString,
      Computed: true,
    },
    "name": {
      Type:         schema.TypeString,
      Required:     true,
      ForceNew:     true,
      ValidateFunc: validateName,
    },
    // ... //
  },
}
```

1.5. Kód. Példa sémadefiníció

Koncepció

A Terraform képes a konfigurációs fájlon bekövetkező változások felderítésére és ezen változások adoptálására az éles rendszerben. A következő fogalmakat veszi alapul:

- Konfiguráció: milyen környezetet szeretnénk (cél)
- Állapot: jelenleg milyen környezetünk van
- Változások: a jelenlegi állapotban milyen műveleteket kell végezni, hogy a kívánt eredményt (konfigurációt) kapjuk
- Terv: a változások végrehajtási terve
- Végrehajtás: a változások éles rendszeren való alkalmazása

Azaz ha kezdetben (jelenlegi állapot) nincs semmilyen erőforrásunk létrehozva, akkor a konfiguráció alapján az első fázisban a kívánt erőforrásokat kell létrehozni (cél környezet). Ha módosítjuk a konfigurációt, például felveszünk még egy erőforrást, akkor mivel a jelenlegi állapot, a korábban elvárt cél környezet volt, elégséges csak a változásokat végrehajtani, azaz létrehozni azt az erőforrást.

A Terraform ezek kezeléséhez különböző parancsokat ad: `terraform init` - Inicializálja a környezetet, ellenőrzi és felderíti a $.tf$ kiterjesztésű fájlokat. `terraform plan` - Az aktuális állapotnak és a kívánt környezetnek megfelelően megmutatja milyen változásokat indukálhat a terv végrehajtása. `terraform apply` - Végrehajtja a tervet. `terraform destroy` - Törli az összes létrehozott erőforrást az aktuális állapotnak megfelelően.

További lehetőségek

A Terraform, a HCL nyelv könnyebb kezelhetőség miatt bevezetett változókat, a változók különböző típusúak lehetnek (szám, karakterlánc, számok listája, stb.). A változókhoz ha például érzékeny adatról van szó, lehetőségünk van környezeti változót bekötni, illetve egy külön fájlba kiszervezhetőek (Variable Definition File). Ezek használata elősegíti a biztonsági kockázatok (pl. beégetett jelszavak) mérséklését is, a "büdös kód" (code smells) minimalizálásával [22].

Továbbá hasznos funkció a modulok kezelése: a gyakran használt erőforráscsoportokhoz modulokat rendelhetünk. Ezeket a modulokat pedig később akár egy teljesen más

konfigurációban is újrafelhasználhatjuk. Nagy előny, hogy ezeket a modulokat bárki számára közzétehetjük és egy távoli kiszolgálóról így bárki számára könnyedén hozzáférhető.

1.4. Kitekintés

A fent említett eszközöket széles körben használják a különböző felhőszolgáltatók (Azure, AWS, DigitalOcean, Google Cloud, stb.): a cloud-init minden IaaS szolgáltatást nyújtó rendszer alapja, valamint a különböző automatizációt kínáló rendszerek előszeretettel alkalmaznak valamilyen CM és orchestration eszközt vagy eszközöket is.

A Magyar Tudományos Akadémián fejlesztett Occopus rendszer is használja ezeket a technológiákat. A keretrendszer lehetőséget biztosít komplex rendszerek konfigurálására, például klaszterek létrehozására tudományos célból vagy hagyományos devops[4] feladatok ellátásra és elosztott alkalmazások skálázására is. Ilyen lehet egy Kubernetes[5] vagy SLURM[6] rendszer felállítása is. A rendszerben az operációs rendszerek kezdeti konfigurációjának beállítására cloud-init-et, míg a szoftvercsomagok testreszabására elsősorban a Chef CM eszközt alkalmazzák. A rendszerek leírása egy a HCL-hez nagyon hasonló YAML fájllal történik. (kódrészlet)

```
1  infra_name: slurm-cluster
2  user_id: somebody@somewhere
3  nodes:
4    - &M
5      name: slurm-master
6      type: slurm_master_node
7    - &S
8      name: slurm-worker
9      type: slurm_worker_node
10 scaling:
11   min: 2
12   max: 10
13 variables:
14   mungeversion: 0.5.13-2build1
15   slurmversion: 19.05.5-1
16 dependencies:
17   -
18     connection: [ *S, *M ]
```

1.6. Kód. Occopus példakód

Az egyes gépekhez (node) további leírás tartozik, amivel az adott gép konfigurációját lehet megszabni (hasonló a Terraform moduljaihoz): Például fejgép (slurm-master) esetén nagy erőforrásokat nem érdemes biztosítani, mivel az a node csak hibakeresésre, fájlok és a SLURM ütemező elérésére szolgál. [35]

Tudományos és kutatási területeken különösen fontos a reprodukálhatóság, mivel ezek a területek is egyre jobban építenek valamilyen számítási és adatvezérelt módszerre. Virtualizációval a reprodukálás nehézsége enyhíthető, ugyanakkor ehhez a megfelelő eszközökre van szükség. [17] Ehhez elengedhetetlen az IaC eszközök használata, amikkel akár egy komplex "laboratóriumi" környezet is kiépíthető. [8]

OpenIaC

2022-ben megjelentek törekvések a szolgáltatók egységes, szabványos és nyílt OpenIaC (Open Infrastructure as Code) alkalmazására: A projekt célja, hogy megoldást nyújtson

[4]Development and Operations - a szoftverfejlesztés és üzemeltetés egyesítése
[5]Konténer alapú alkalmazáskezelő szoftver
[6]Simple Linux Utility for Resource Management (SLURM) - feladat-ütemező és erőforrás-kezelő

a felhő alapú számítástechnika egységes integrálására, így kielégítve a modern információs architektúrák igényeit is. A megoldási ötlet az Eduroam[7]-hoz kapcsolódó intézmények ingyenes WiFi eléréséhez hasonlítható: Az egyes intézményi tagok hálózatához bárki csatlakozhat, az azonosítást a szülő intézmény végzi el, és onnan kap megfelelő jogokat más intézményi hálózathoz való csatlakozáshoz. A projekt fő hajtóereje az 5G technológia igényeinek kielégítése is. [24]

1.5. Lehetőségek

A fenti eszközök bevezetésével a CIRCLE rendszer képes a már említett automatizáció mellett különböző orkesztrációs feladatok ellátására is. Így komplex infrastruktúrák felépítésére is alkalmas, ami kutatási és oktatási környezetben előfordulhat: Ilyen például a diákok számára egy virtuális környezet biztosítása, vagy komplex klaszter kialakítása is, mint például az előző szakaszban ismertetett SLURM klaszter tudományos felhasználásra. Az IaC által vezérelt komplex infrastruktúra így könnyen menedzselhetővé is válik.

[7] `https://eduroam.org`

2. fejezet

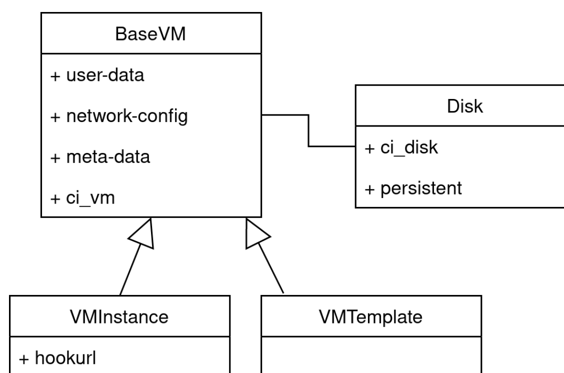
Megvalósítás

2.1. Cloud-init bevezetése

Mivel nagyon fontos, alapvető funkcióról van szó, nemcsak az IaC eszköz számára, de a jelenlegi webes felületre is kivezettem a cloud-init konfigurációs lehetőségét.

A funkció használatához elengedhetetlen a jelenlegi model módosítása a következő adatokkal: Fel kell venni a meta-data, a network-config és az user-data információk tárolására 3 szöveges mezőt, valamint egy logikai mezőt, ami a cloud-init aktivitását jelzi. Ezzel kompatibilis lesz a jelenlegi rendszerrel, nem kell feltétlenül megvalósítani a cloud-init szolgáltatást, illetve ez hasznos lesz a virtuális gép példány kezelésekor is. A vendor-data beállítása minden vm-nél azonos, így ahhoz nem kell külön mező.

Mivel a VM példányok mellett fontos a sablonok támogatása is (1. fejezet) oda is fel kell venni ezeket a mezőket, így a közös ősbe ki lehet ezeket emelni (2.1. ábra).



2.1. ábra. Adatmodell (csak változások)

A konfiguráció minden egyes példánynál más-más lehet, bevezettem a Jinja template motorral ([19]) a cloud-init konfiguráció sablonozását is. Ennek megfelelően a VM indítása előtt a template motor segítségével a cloud-init mezőkben lévő szöveget feldolgozzuk, és behelyettesítjük a megfelelő értékeket. A Jinja motorral komplexebb (ciklusok, elágazások) kifejezések is feldolgozhatóak.

A főbb template motor makrók, amiket használhatunk:

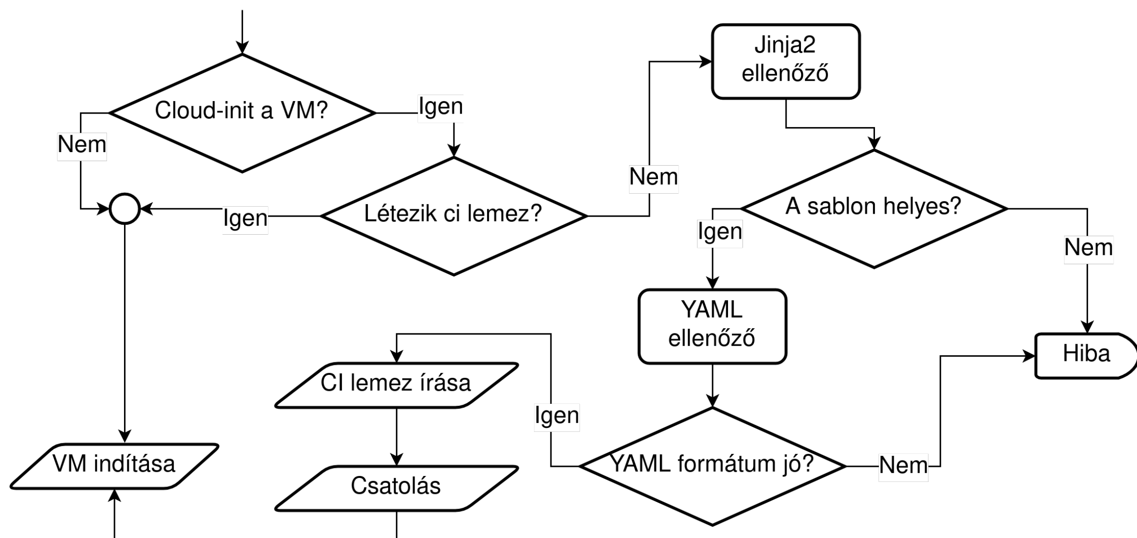
- `sysuser` - felhasználói név, alapértelmezetten a korábban használt 'cloud' értéket adja vissza.
- `password` - a rendszer által generált jelszó az alapértelmezett felhasználóhoz
- `hostname` - a rendszer által kiosztott hosztnév

- `acl.allusers` - az összes vm példányhoz társított felhasználó fióknevei (lista)
- `acl.operators` - az összes vm példányhoz tartozó operátor szintű felhasználó fióknevei (lista)
- `ssh.keys` - a tulajdonos mentett ssh-kulcsai (map, a kulcs az ssh kulcs neve)
- `net.ipv4/ipv6/mac` - alapértelmezett interfész címei
- `variables` - globális kulcs-érték tároló, a felhasználó által is változtatható, elősegíti a security code smell-ek minimalizálást[22]. A REST-API-n keresztül változtatható (2.1.1. szakasz).

A cloud-init használatához egy megfelelő lemezt kell csatolni a virtuális géphez. A lemezen el kell helyezni a meta-data, network-config és a user-data állományokat, ehhez a minden Linux rendszeren alapvető eszközt a `genisoimage` parancsot használtam. A parancs a megfelelő felparaméterezéssel képes optikai meghajtók számára lemezek létrehozására ISO9660 szabvány szerint a Rock Ridge Interchange protokoll (RRIP) alapján. [5]

Bár a cloud-init beállítása történhet hálózaton keresztül is, egy megfelelő kiszolgáló szerver segítségével, végül elvetettem ezt a lehetőséget. Mivel jelenleg is a port-forwarding miatt (1.1.2 szakasz) nagyon sok randomizált port kerül felhasználásra, nem tartottam reálisnak ezt a megvalósítást: a szabad portok folyamatosan változnak még így is, illetve a networkdriver-be is bele kell akkor nyúlni az implementáláshoz, ráadásul, minden indításkor a boot folyamat végéig biztosítani kellene egy szabad portot, amin a kiszolgáló hallgat, ugyanakkor erőforráspazarlás lenne a VM megsemmisitéséig biztosítani ezt (újraindításkor is kellhet). Így folyamatosan változnának a szabad portok.

A működés a folyamatábrán (2.2. ábra) nyomonkövethető: Ha a virtuális gép cloud-init képes, akkor először ellenőrizni kell, hogy létezik-e korábbi cloud-init lemez, ha igen, akkor a VM elindítható. Ha nem akkor először a Jinja2 template-motorral be kell helyettesíteni a makrókat, ha helyes. Majd egy YAML szintaktikai ellenőrző is lefut a feldolgozott konfiguráción. Ha ezek nem adnak semmilyen hibát, akkor elkészülhet a lemez, amit pedig csatolunk a virtuális géphez, végül a gép indítható.



2.2. ábra. Folyamatábra a cloud-init lemez kezeléséhez

A lemez írása az első fejezetben már korábban bemutatott storagedriver segítségével történik: A menedzser gép AMQP (celery taszk) protokollon keresztül kérelmezi a

cloud-init lemez létrehozását. Ami a korábban említett `genisoimage` paranccsal kiírja az információkat egy virtuális lemezre. Végül a lemezt felcsatolja a virtuális géphez, és aztán indítja a gépet.

A virtuális gép sablonként való elmentésénél a cloud-init lemezt nem szabad elmenteni, mivel a konfiguráció a template motor miatt az elmentett sablonból képzett virtuális gépeknél más- és más lesz, ezért ott is le kell futnia a fenti folyamatnak. Így a cloud-init lemezt meg kell különböztetni a többitől (2.1. ábra), és ezeket nem szabad akkor lementeni.

2.1.1. Példa konfiguráció

Bár a meta-data állomány kifejezetten az IaaS szolgáltató számára van, jelenleg úgy tartottam érdemesebbnek, hogy az kivezetésre kerüljön a mostani felületre is a könnyebb változtathatóság miatt, ezeket csak az adminisztrátor szintű felhasználók módosíthatják. A meta-data állomány alapértelmezetten a lenti konfiguráció, a `{{ hostname }}` makró helyére a virtuális gép hosztneve kerül.

```
1 instance-id: {{ hostname }}
2 local-hostname: {{ hostname }}
3 cloud-name: circle3
4 platform: circle3
```

A cloud-name és platform-al meg lehet adni az IaaS szolgáltató nevét, és így be lehetne tölteni alapértelmezett konfigurációkat, mivel a circle3 nincs regisztrálva a cloud-inithez, ez csak formai beállítás.

A következő kódrészletben egy lehetséges user-data konfigurációs példa látható. A sablonozás miatt figyelni kell, hogy milyen konfigurációt adunk meg. Például a lenti állományban direkt azért a `chpasswd` kulcsszónál kell megadni a felhasználó jelszavát, mertha azt a `users` alatt tennénk meg, az csak a felhasználó létrehozáskor fut le (első indításnál), így a sablonnal való elmentésnél, az abból indított vm-nél nem állítódna be a jelszó.

Ez a fajta beállítás elkerülhető lenne a sablon elmentése előtt a `cloud-init clean` parancs kiadásával és a felhasználó törlésével is, hiszen ilyenkor is lefutnak a cloud-init-nek a megfelelő folyamatai.

```
1  #cloud-config
2  ssh_pwauth: 1
3  users:
4    - name: {{ sysuser }}
5      sudo: ['ALL=(ALL) NOPASSWD:ALL']
6      groups: sudo
7      shell: /bin/bash
8      chpasswd: { expire: False }
9      lock-passwd: false
10 chpasswd:
11     list: |
12        {{ sysuser }}:{{ password }}
13 expire: False
```

2.1. Kód. Példa user-data fájl sablonozással

Az alábbi cloud-init sablonból a következő konfigurációs fájl generálódik, ha a virtuális gép operátori szintű felhasználói: alice és joe, és a virtuális gép tulajdonosának van egy ’my-key’ ssh-kulcsa.

```
#cloud-config
users:
  - name: {{sysuser}}
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    groups: sudo
    shell: /bin/bash
    ssh_authorized_keys:
      - {{ ssh.keys['my-key'] }}
{% for u in acl.operators %}
  - name: {{u}}
    shell: /bin/bash
{% endfor %}
```

2.2. Kód. Kezdeti sablon

```
#cloud-config
users:
  - name: cloud
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    groups: sudo
    shell: /bin/bash
    ssh_authorized_keys:
      - ssh-rsa skahjd...
  - name: alice
    shell: /bin/bash
  - name: joe
    shell: /bin/bash
```

2.3. Kód. Generált fájl

2.2. REST-API

A Terraform kezeléséhez elengedhetetlen egy megfelelő REST API az erőforrások kezeléséhez és az adatok eléréséhez. Az API-t a Django ([12]) keretrendszerhez adott django-rest-framework ([11]) modullal valósítottam meg. A modul nagyon gyors fejlesztést tesz lehetővé, bár több megközelítést is támogat, én az implementált megoldáshoz tartozó elemeket szeretném áttekinteni a dolgozatban.

A modul rendelkezik úgynevezett serializer osztályokkal, amikkel megadható, hogy az adott objektumnál milyen mezőket kell kezelni, melyik a csak olvasható mező (ilyen például az ip címek, jelszavak, amiket a rendszer maga generál). A json sorosítást ezen osztályok végzik, gyakorlatilag egy lehetséges nézetről van szó, ezeken belül megkülönböztethetünk a a django Model-osztályok számára készült és saját serializer osztályokat is.

Ezen sorosító osztályok segítségével az egyes végpontokhoz a megfelelő http függvényekkel (get, put, post, delete) viselkedés társítható: Például az alábbi lista a virtuális gépek kezeléséhez tartozó alapszintű végpontokat adja meg, az adat mindig a kérés törzsében van:

- `POST acpi/vm/` - Létrehoz egy virtuális gépet.
- `GET acpi/vm/` - Visszaadja a virtuális gépek listáját.
- `GET acpi/vm/<id>/` - Lekérdezi az 'id' azonosítójú virtuális gépet.
- `DELETE acpi/vm/<id>/` - Törli az 'id' azonosítójú virtuális gépet.
- `PUT acpi/vm/<id>/` - Módosítja az 'id' azonosítójú virtuális gépet.
- `POST acpi/vm/<id>/op/deploy/` - Elindítja az 'id' azonosítójú virtuális gépet, ha nem aktív.

Például a `GET acpi/vm/12` visszaadja a 12 azonosítójú virtuális gép json leírását.

Az authentikációhoz minden kérésnél egy felhasználói token-t kell elküldeni a fejlécben. A fenti REST hívásokat jelenleg csak admin jogosultságú felhasználók használhatják, így ennek megfelelően a Terraform funkcionalitását csak ők tudják jelenleg kihasználni.

Sajnos a volt rendszer nem támogatta az url címről való lemezkép letöltése közbeni módosítások végrehajtását a virtuális gépen, ami bizonyos esetekben indokolható volt, de többnyire teljesen felesleges, ami szükségtelenül akadályozta más, független operációk végrehajtását. Így ennek megfelelően ezzel a funkcióval bővítettem a menedzser modult a CIRCLE rendszerben.

Valamint a másik hátrány volt, hogy virtuális diszket, önmagában nem lehetett létrehozni, csak egy virtuális gépnél, ha azt rögtön az adott vm-hez társítjuk. Így perzisztens

tárolókat csak manuális megoldásokkal lehetett kezelni, ha kézzel beletúrunk a django-ba, és úgy társitjuk más virtuális géphez az adott diszket. Ennek megfeleően a következőkkel bővült a funkcionalitás: VM nélkül is lehet létrehozni diszket (url címről, és teljesen üreset is) valamint az egyes diszkeket le lehet másolni, hogy aztán a másolatot felhasználhatjuk egy másik virtuális gépnél. Perzisztens tárolónál lehetőség van letöltés után automatikus átméretezésre is.

Emellett érdemes lenne bevezetni egy tárolót (akár minden adatközponthoz egy közöset), ahonnan az egyes képfájlokat lehet letölteni. Ez egységes kezelést tesz lehetővé, valamint így nem kell a távoli letöltési megoldást választani, amivel a felhasználók kevésbé megbízható forrásból is beszerezhetnek ilyen képfájlokat. Viszont ennek kezelése és karbantartása jelentős humánerőforrás bevonásával járna.

2.2.1. Aszinkron műveletek kezelése

A tipikusan hosszú lefutású műveletekhez, ilyen például egy lemezkép letöltése egy url címről, vagy egy virtuális gép sablonként való elmentése, a végpont egy referencia (activity) objektumot ad vissza, amin keresztül ellenőrizhető a folyamat állapota: Például a letöltés hány százalékon áll, és ha a folyamat befejeződött, milyen azonosítóval, és hol érhető el az adott erőforrás.

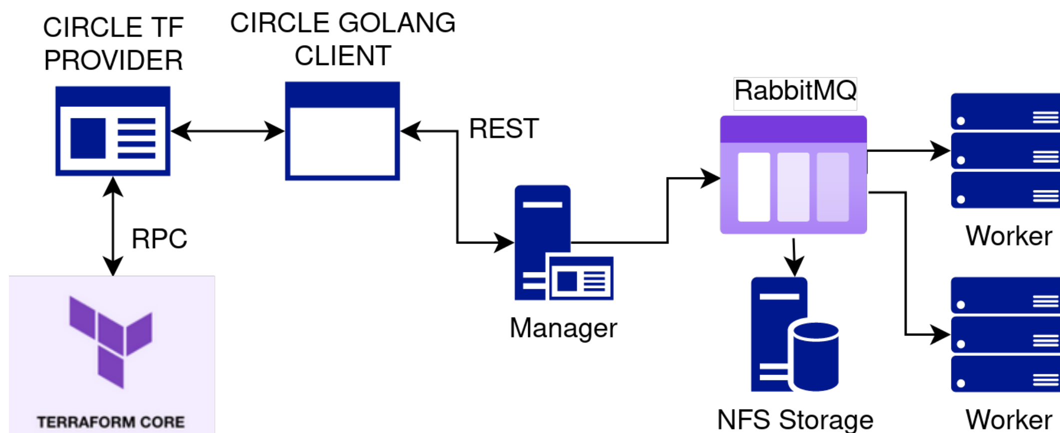
A használathoz egy szekvencia diagram megtalálható a következő szekcióban (2.3.3).

2.3. Terraform provider

2.3.1. Golang kliens

A provider létrehozásához elengedhetetlen egy megfelelő kliens alkalmazás (könyvtár), amivel el lehet érni a REST API-t Go nyelvből. Ez a kliens alkalmazás felépíti a kapcsolatot, nyilvántartja a hitelesítő adatokat és a golang számára a kapott json objektumokat sorosítja a megfelelő go struktúrába. Ezen kívül az elküldött kéréseket, válaszokat naplózza a jobb nyomon követhetőség miatt.

Ezen kliens könyvtáron keresztül érem el a CIRCLE rendszert a Terraform provider moduljából, a működést 2.3. ábra mutatja.



2.3. ábra. A Terraform provider és a CIRCLE rendszer működése

2.3.2. Sémák

A szolgáltatás kezeléséhez szükséges a megfelelő sémák felvétele. A táblázat (2.1.) összefoglalja milyen adat- és erőforrássémákat (1. fejezet) hoztam létre:

Séma	Típus	Feladat
CIRCLE	szolgáltatói	adatközpont tulajdonságainak megadása
VM	adat/erőforrás	virtuális gépek kezelése
Template	adat/erőforrás	sablonok kezeléséhez
Disk	adat/erőforrás	lemezek (távoli url, üres, perzisztens)
Port	adat/erőforrás	virtuális gépek port forwarding megoldásához
VMPool	erőforrás	tömbösített indítás több felhasználónak
VLAN	adat	VLAN adatok lekérdezésére
Lease	adat	lejárati csoportok lekérdezésére
User	adat	Felhasználói adatok lekérdezésére
Group	adat	Felhasználói csoportok lekérdezésére

2.1. táblázat. Jelenleg létrehozott sémák a modulhoz

A fenti sémák közül kiemelem a Disk és a VMPool sémákat: Ahogy azt a REST API-nál említettem (2.1.1), bevezettem további lemeztípusokat: megjelentek a perzisztens tárolók is mind az üres, mind az url-ről letöltött változatnál. A perzisztens tárolók nagy előnye, hogy független a csatolt virtuális géptől, pontosabban ha T_p egy perzisztens tároló M pedig egy virtuális gép, akkor csak $T_p \leftarrow M$ függés áll fenn, míg nem perzisztens lemeznél függ az eredeti virtuális géptől, így nem mozgatható át (a virtuális gép megsemmisítése a lemezt is magával vonja). Így a perzisztens lemezek kezelése sokkal rugalmasabb és újrafelhasználhatók. Az implementáció, az hogy éppen perzisztens vagy nem perzisztens tárolót hozunk-e létre, attól függ, hogy a disk erőforrásnál megadtuk-e a *vm* tulajdonságot, ami a virtuális gép azonosítóját várja, hiszen ennek megadásakor már egyértelmű függés van a lemez és a gép között.

A VMPool sémát a tömbösített indítás során használhatjuk: A felhasználók egy részének így virtuális gép indítható egy megadott sablonból. Ezen virtuális gépek együtt kezelhetőek, emiatt kevésbé testre szabható, minthogyha a Terraform által nyújtott szolgáltatásokat (`for_each` meta-argumentum vagy `for` nyelvi elem) használtuk volna. Így ez a séma csak erőforrás típusú lehet.

2.3.3. Állapotok kezelése

A Terraform Core által felderített változásokat fel kell oldani a megfelelő módon. Ezekre attól függően hogy milyen erőforrásról van szó és hogy az erőforrás milyen állapotban van, másként kell reagálni.

Ha egy virtuális gép szimbolikus neve megváltozik, akkor annak kezelése egyszerű: Mivel magát a működést nem érinti (nem kell például a virtuális gépet megállítani), illetve más erőforrásokat sem érint a változás, mert azok az elsődleges azonosítóval (id) hivatkoznak a virtuális gépre, csak egy megfelelő http kérést kell elküldeni. Ugyanakkor a helyzet többnyire nem ilyen egyszerű: Például a virtuális gép memóriáját, CPU-k számát vagy azok prioritását csak leállított (STOPPED állapot) virtuális gépnél lehet megváltoztatni. Azt is fontos figyelembe venni, hogy a változtatásokat milyen sorrendben küldjük el. Ha az előző példában kérjük a virtuális gép megállítását, akkor a többit végre tudjuk hajtani, de fordítva nem. Így fontos, hogy mely változásokat oldjuk fel először, mert ezek is egymástól függhetnek. Így a virtuális gép állapotát (futó, alvó ...) kell elsősorban lekezelni.

Korábbi állapot	Cél	Operáció
Megállított Függőben	Futó	Indítás
Futó	Leállított	Megállítás
	Alvó	Altatás
Alvó	Futó	Ébresztés

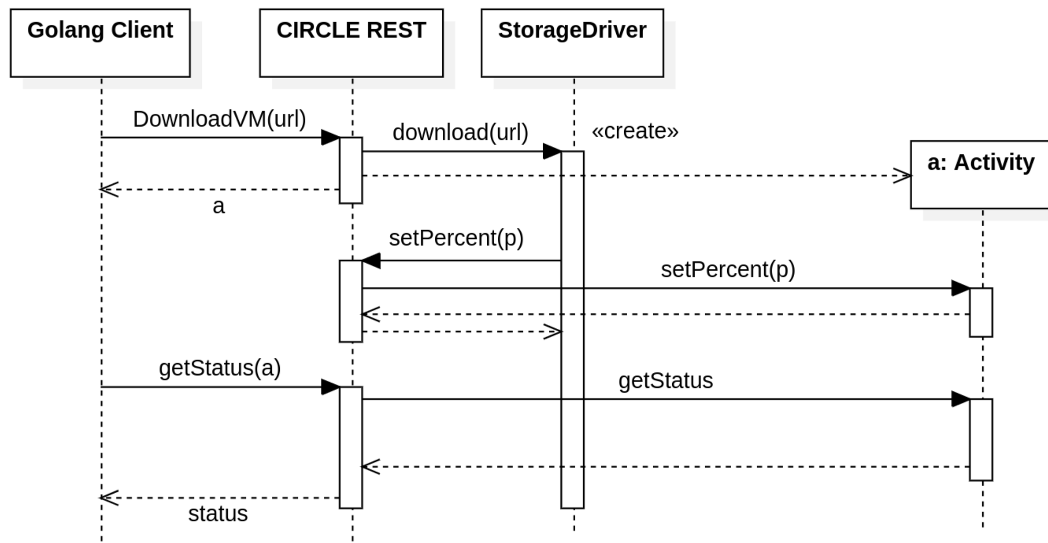
2.2. táblázat. Állapotmátrix

A virtuális gép állapotának kezeléséhez megfelelő átmeneti mátrixra van szükség, milyen operációkat kell végezni a megfelelő eredményhez. Ez a mátrix látható a táblázatban (2.2. táblázat).

2.3.4. Aszinkron hívások kezelése

Mivel a Terraform alapvetően blokkoló megoldást használ az erőforrások közötti függések miatt, az aszinkron műveleteket be kell várni. A Terraform Core gondoskodik arról, hogy ahol lehet és a függőségek nem akadályozzák, párhuzamosan hajtsa végre az infrastruktúra felépítését.

Így például a lemezek letöltésénél először a kérést kell elküldeni, és a visszaadott activity objektumon (Asynchronous Completion Token - ACT [26], a kliens nézőpontjából egy Future object) keresztül tudjuk az állapotát lekérdezni (2.4. ábra). Ennek megfelelően amíg az erőforrás nem készül el vagy hibaüzenetet nem kapunk, lekérdezzük az állapotot.



2.4. ábra. A távoli hívás szemléltetése (egyszerűsített szekvencia diagram)

Implementációs példa

Mivel kevés rendelkezésre álló publikus cím van, a CIRCLE rendszerben eltér a címek kezelése: A virtuális gépek, hogy tudjanak szolgáltatásokat biztosítani, ideiglenes portokat kapnak a publikus címhez kapcsolva (bővebben az első fejezetben).

A címek kezeléséhez hoztam létre a port sémát. A séma mezőit a táblázat mutatja. Ahogy a táblázatban látható, két számított (computed) elem van a sémában: egress portot a rendszer osztja ki, így kivűlről az adott szolgáltatás azon a porton keresztül érhető el.

Mező	Típus	Tulajdonság	Jelleg
type	string	tcp/udp	Szükséges
vm	int	a virtuális gép azonosítója	Szükséges
vlan	int	vlan (interfész) azonosítója	Szükséges
port	int	belső port	Szükséges
egress port	int	külső port	Számított
forwarding	bool	átirányított-e	Számított

2.3. táblázat. A port séma attribútumainak tulajdonságai

Míg a fowarding a használt VLAN beállításaitól függ. Egy port nyitásához a kapcsolat típusán felül meg kell adni, hogy melyik vm-hez, illetve melyik VLAN-hoz kapcsolódva kell az adott portot megnyitni. Ezen erőforrásokra az elsődleges azonosítójukon keresztül lehet hivatkozni, így ez egy potenciális függés lesz a gráfban lefutáskor.

Az implementációhoz el kell készíteni a megfelelő függvényeket az erőforrás kezeléséhez (1. fejezet). A létrehozás során be kell olvasni az attribútumokat (`CreateContext`), és el kell küldeni a kérést a megfelelő API végpontnak. A végponttól kapott választ pedig be kell tölteni az állapotba, ezek a számított értékek beállítását jelenti.

```
1  func resourcePortCreate(ctx context.Context, d *schema.ResourceData, m interface{}) diag.
       Diagnostics {
2    c := m.(*circleclient.Client)
3    interf := circleclient.PortsReq{
4      Vlan:     d.Get("vlan").(int),
5      Instance: d.Get("vm").(int),
6    }
7    port := circleclient.OpenPort{
8      DestinationPort: d.Get("port").(int),
9      Type:            d.Get("type").(string),
10   }
11   portres, err := c.CreatePort(interf, port)
12   if err != nil {
13     return diag.FromErr(err)
14   }
15   d.SetId(fmt.Sprintf("%v/%v/%v", interf.Instance, interf.Vlan, portres.Egressport))
16   d.Set("egress_port", portres.EgressPort)
17   d.Set("forwarding", portres.Forwarding)
18   return diag.Diagnostics
19 }
```

2.4. Kód. Port létrehozását lekezelő függvény

Az `UpdateContext` esetén fel kell oldani a változásokat: A portok kezelésénél ez azt jelenti, hogy bármelyik attribútum megváltozásánál a korábbi portot törölni kell, és egy új portot kell nyitni az új beállításokkal.

A `ReadContext` beolvassa az összes attribútumot a távoli szolgáltatótól, míg a `DeleteContext` törli a portot az elsődleges azonosító alapján, ami minden erőforrásnál kötelező tulajdonság.

A lenti kód mutat egy példát a használatra.

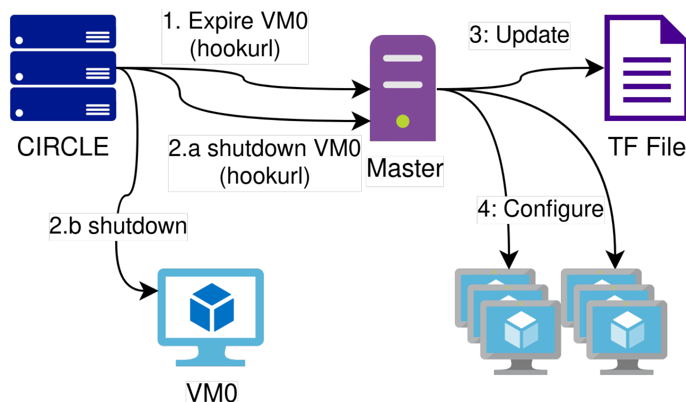
```
1  terraform {
2    required_providers {
3      circle3 = {
4        version = "0.1"
5        source  = "bmeik/tf/circle3"
6      }
7    }
8  }
9  provider "circle3" {
```

```
  address = "https://cloud3.fured.cloud.bme.hu"
  port    = 443
  // export CIRCLE3_TOKEN="secret-key"
}
data "circle3_template" "basetemplate" {
  name = "ubuntu v1"
}
resource "circle3_vm" "from_template_tf" {
  name = "from template"
  from_template = data.circle3_template.basetemplate.id
}
resource "circle3_port" "openport8080" {
  port = 8080
  vlan = circle3_vm.from_template_tf.vlans[0]
  vm = circle3_vm.from_template_tf.id
  type = "tcp"
}
```

A "basetemplate" azonosítójú adatforrásnál a megadott név alapján megkeressük az adott sablont, majd beolvassuk azt. Az olvasott értékeket pedig felhasználjuk a "from_template_tf" virtuális gép készítésekor. A port nyitásához szükség van a VLAN-ra és a virtuális gép azonosítójára, ezeket a virtuális gép készítése után szintén megkapjuk, és hivatkozhatunk rájuk a fenti módon a HCL nyelvben.

2.3.5. Az élet-ciklus modell bővítése

Mivel a virtuális gépek az élet-ciklus modell (1.1.2. szakasz) miatt a CIRCLE menedzsere által felfüggesztésre kerülhetnek, fontos hogy erről időben értesüljön a virtuális gépet kezelő automatizáló eszköz, ami a Terraform modult használja, akár emberi beavatkozás nélkül is. Ehhez a rendszer egy esemény bekövetkezte előtt, illetve az esemény bekövetkeztekor



2.5. ábra. Példa a webhook alkalmazására

vagy a virtuális gép állapotának megváltozáskor egy webhook (2.1. ábra) címre egy HTTP kérést küld a virtuális gép azonosítójával és a konkrét esemény eseményleírójával együtt. Így lehetőséget ad a CIRCLE rendszer a beavatkozásra vagy a távoli állapot automatikus frissítésére.

A 2.5. ábra egy példakörnyezetet mutat a webhook funkció használatára: A 'VM0' virtuális gép, ami egy virtuális klaszter egyik tagja, éppen lejárna, ekkor a virtuális géphez beállított hookurl-en keresztül értesítést küldünk a klaszter master csomópontjának (ami szintén lehet akár egy virtuális gép). Mivel a jelenlegi terhelés mellett nem szükséges a futó virtuális gép, a mester nem reagál rá, így megkapja a CIRCLE menedzser által szintén a hookurl-re küldött értesítést, hogy a VM0 le lett állítva. Ennek megfelelően a

helyi Terraform állapotokat frissíteni kell, illetve a klaszterhez tartozó további vm-eket is értesíteni kell, hogy 'VM0' már nem elérhető.

2.3.6. Távoli állapottárolás

Az infrastruktúra közös fejlesztésének megkönnyítésére szükséges a távoli állapottárolás: Több felhőszolgáltató eltérő megoldásokat nyújt az állapotok tárolására: többnyire valamilyen Object Storage-en tárolhatók ezek a fájlok, vagy Amazon esetén egy speciális bucket tárolón DynamoDB adatbázis motorral. [29]

A legegyszerűbb implementáció erre, ha a Postgres adatbázissal alakítjuk ki a táblákat az állapotok tárolására. A megvalósításban ez annyit jelent hogy létrehozzuk a felhasználóknak megfelelő fiókokat az adatbázismotornál is, és aztán a különböző Terraform projekteknek külön-külön adatbázis táblákat csinálunk. Az éles megvalósításban ezt egy virtuális gépen készítettem el (rendszerkomponens VM) biztonsági okokból, valamint a virtuális gép így könnyen mozgatható, ha el kellene költöztetni máshova.

A Postgres alkalmazásának előnye, hogy támogatja az állapot zárolását minden olyan művelethez, ami írhatja azt. Így elkerülhető az, hogy inkonzisztens állapotok szerepeljenek.

3. fejezet

Használati esetek

A fejezetben alapvető alkalmazási módokat mutatok be az IaC eszköz használatára oktatási és kutatási környezethez.

3.1. Terraform alkalmazási minták

A hivatkozott forrás [13] alapján a következő alkalmazási módokban fordulhat elő a Terraform által felépített infrastruktúra:

Terralith: egyszerű konfiguráció egy infrastruktúra kezeléséhez, általában egy tf fájllal. Így az egyes, szeparálható komponensek nincsenek különválasztva, általában a DRY (Don't Repeat Yourself[1]) elv sérül. A különböző konfigurációs paraméterek a fájlba vannak égetve. Az infrastruktúra kód nehezen átlátható, kevésbé automatizálhatók a változtatások.

Multi-Terralith: Több terraform definíciós fájlt használunk (pl. logikai vagy funkcióbeli szétválasztás), többnyire alkalmazzuk a változókat a konfigurációban, elválasztjuk egymástól a nem összetartozó állapotokat a projekt szétválasztásával, így azok egymásra nincsenek hatással.

Terramod: A terraform projekt modulokba szervezése, a modulok segítségével az azonos jellegű infrastruktúrák készítése könnyen újrafelhasználható.

Power Terramod: Az előző minta használatára épít: Definiáljuk az alacsony szintű alapmodulokat és a rendszer központi elemeit meghatározó modulokat. Így a kódduplikáció előfordulása szinte megszűnt.

Terraservices: Komplex kódnál elengedhetetlen a csapatmunka, ugyanakkor ez problémát is felvet: Ha az egyik fejlesztő változtat az infrastruktúrán akkor az csak a lokális állapotfájlban kerül frissítésre, másoknál nem, így inkonzisztencia fordulhat elő. Megoldás: Áttérés távoli állapottárolásra (remote state). Egy publikus verziókezelőn való tárolás a HCL kód szempontjából biztonságos, de a belső állapotot reprezentáló tfstate fájl tartalmazhat nem publikus adatokat is. Így ezeket célszerű egy belső tárolón elhelyezni.

A következőkben bemutatok néhány alkalmazási módot, ugyanakkor a használati esetek megfelelő bemutatása miatt többnyire csak 'Terralith' szinten. Továbbá a helytakarékosság miatt, csak a lényegi részeket emelem be a kódrészletekbe.

3.2. Egyszerű VM készítése

A lenti példában egy új virtuális gép indítására használható HCL kód látható. Ez segítséget nyújthat új laboros környezet kialakítására, az automatizáció miatt nem kell köz-

[1] Tervezési alapelv, lényege hogy kerüljük a kódduplikációt

beavatkozni, a konfiguráció pedig később újrafelhasználható a megfelelő paraméterek módosíthatóak. Például a virtuális gépek sablonjait automatikusan előállíthatjuk különböző operációs rendszerek és verziók esetén is könnyedén, beavatkozás nélkül.

```
// [a terraform és a szolgáltató konfigurálása]
data "circle3_lease" "labor_lease" {
  name = "lab"
}
data "circle3_vlan" "default_vlan" {
  name = "vm"
}
resource "circle3_disk" "ubuntu18" {
  name = "ubuntu18.04"
  url = "http://cloud-images.ubuntu.com/bionic/current/bionic-server-cloudimg-amd64.img"
  resize = "10G" //letöltés után a cloud-init image átméretezése
}
resource "circle3_vm" "basic" {
  name          = "terraform"
  // [további tulajdonságok]
  lease         = data.circle3_lease.labor_lease.id
  cloud_init    = true
  ci_user_data  = file("${path.module}/user-data.yaml") // egyéni user-data konfiguráció
  num_cores     = 2
  ram_size      = 256
  priority      = 80
  arch          = "x86_64"
  disks = [circle3_disk.ubuntu18.id]
  vlans = [data.circle3_vlan.default_vlan.vid]
}
```

3.1. Kód. Egyszerű HCL kód egy infrastruktúra felépítéséhez

A konfigurációt testre szabhatjuk a `remote-exec` szolgáltatás használatával, ami ssh kapcsolatot épít fel az erőforráshoz, és végrehajtja a megadott parancsokat. (Pl. egy repository letöltése). Ebben az esetben az erőforrás egészen addig nem készül el, amíg nem sikerült csatlakozni hozzá, és lefutottak a kiadott parancsok. Így a tőle függő elemek nem is hajtódhatnak végre addig. A parancsokat bizonyos eseményekhez is tudjuk társítani: ilyen lehet például az erőforrások frissítésekor és azok megszűnésekor lefutó parancsok.

Vagy hasonló eredményt tudunk elérni a cloud-init `runcmd/bootcmd` eszközökkel is. Ennek megfelelően, ha a laboranyaga megváltozik, akkor nem kell egy új virtuális gépben frissíteni a szükséges fájlokat, könyvtárakat, majd aztán egy új sablonban elmenteni azt, hanem pl. egy megfelelő url-címről a változások letölthetőek automatikusan, vagy scriptekkel tovább konfigurálhatóak.

```
resource "circle3_vm" "from_template_tf" {
  // [beállítások]
  connection {
    user = "cloud"
    password = self.pw
    host = self.hostipv4
    port = self.sshportipv4
  }
  provisioner "remote-exec" {
    inline = [ "git clone https://... && echo 'hello world' > hello.txt" ]
  }
}
```

3.2. Kód. remote-exec működése

3.2.1. Tesztelés

A létrehozott infrastruktúra teszteléséhez, attól függően hogy milyen tesztelési módszert (funkcionális, nem funkcionális) választunk, különböző megoldások vannak (Bővebben: [4]).

A Terraform is rendelkezik alapvető teszteszközökkel, ugyanakkor nem praktikus (elsősorban csak az infrastruktúra létezését lehet ellenőrizni). Egy egyszerű és alapvetően funkcionális tesztekhez készített eszköz a Terratest ([30]). Segítségével egy infrastruktúrát leíró terraform kód könnyen tesztelhető, nem csak a létrehozott infrastruktúra, hanem az azon futó szolgáltatások is.

A CIRCLE3 folyamatos teszteléséhez például felhasználjuk ezt a technológiát is. A 3.1 és 3.2 kód felhasználásával mindig új környezetet építünk fel: A remote-exec letölti a legújabb verziót, és telepíti lokálisan. Majd a Terratest a lokális tesztek lefutása után a külső interfészt is teszteli, azaz hogy kívülről is elérhető a honlap.

Nagy előny, hogy komplex infrastruktúra felépítése is könnyen automatizálható, azaz az is egyszerűen tesztelhető, hogyha az egyes komponensek külön gépeken futnak (más-más VM-ben).

```
package test
import (
  "testing"
  "github.com/gruntwork-io/terratest/modules/terraform"
  "github.com/stretchr/testify/assert"
)
func TestTerraformExample(t *testing.T) {
  terraformOptions := terraform.WithDefaultRetryableErrors(t, &terraform.Options{
    TerraformDir: "../examples/circle3", })

  defer terraform.Destroy(t, terraformOptions)

  terraform.InitAndApply(t, terraformOptions)
  output := terraform.Output(t, terraformOptions, "vm_name")
  assert.Equal(t, "circle3", output)
}
```

3.3. Kód. Példakód Terratest alkalmazásához

Importálás

A Terraform lehetőséget nyújt nem Terraform által menedzselt erőforrás betöltésére/importálására is. Ezáltal a külső erőforrás állapota is eltárolásra kerül, így azután az is kezelhetővé válik. Egy példát látunk a lenti HCL kódrészletben, ezután a `terraform import circle3_vm.basic <id>`-val betölthető az erőforrás.

```
resource "circle3_vm" "basic" {
}
```

3.4. Kód. Import funkció

Az importálás nagy hátránya, hogy csak a távoli erőforás állapotát "importálja" be, az később nem módosítható a HCL konfigurációs fájlból, mivel azt nem generálja le. Ennek megoldására Microsoft Azure esetén használható az "aztfy"[3] vagy Google Cloud esetén a "terraformer"[2] (3. fél által) csomagok, amik egy külső erőforrásból vagy erőforráscsoportból képesek legenerálni a HCL/Terraform konfigurációs fájlokat, amivel később az adott erőforrások reprodukálhatóak. A modulok használatánál a cél nem a teljes reprodukálhatóság, az nem is érhető el, mivel például a virtuális gépen utólag telepített szoftvereket

[2] `https://github.com/GoogleCloudPlatform/terraformer`

nehéz összegyűjteni. Ezt a funkciót érdemes lenne implementálni a CIRCLE esetén is, illetve hasznos funkció lenne még erőforráscsoportokat létrehozni az egyes virtuális gépek, diszkek, VLAN-ok közös kezeléséhez.

3.3. Többgépes környezet

Bizonyos laboralkalmakra vagy vizsgákra szükséges a hallgatók egy csoportjának virtuális gépeket indítani. Így előfordulhat az is, hogy egyszerre akár több 100 virtuális gépet használnak ilyen célból!

Nagy hallgatói létszámmal rendelkező tárgyaknál egy adatközpont általában nem elég az igények kiszolgálására és a központok közötti automatikus fájlátvitel nem megoldott. Ekkor, mivel bizonyos CIRCLE verziókban a virtuális gépek kiexportálása sem teljesen kiforrott funkció, vagy nem könnyen megoldható a webes felületen keresztül, általában helyileg ”kézzel” kell az egyes fájlokat lementeni és átvinni egy másik adatközpontba. (Közben figyelni kell a sablonozásra és a backing fájlra az átvitel során)

A cloud-init és Terraform konfiguráció segítségével ezen problémák könnyen megoldhatóak: A cloud-init biztosít egy egységes kezelést a virtuális gép konfigurálásához és alapvető CM feladatok is elvégezhetőek vele. A Terraform pedig megoldást nyújt alternatív adatközpontok hozzáadására is.

```
1  // [terraform konfiguráció]
2  provider ”circle3” { //alapértelmezett
3    address = ”https://cloud3.fured.cloud.bme.hu”
4    port    = 443
5    // token
6  }
7  provider ”circle3” {
8    alias   = ”pumi”
9    address = ”https://pumi.niif.cloud.bme.hu”
10   port    = 443
11   // token
12 }
13 resource ”circle3_disk” ”ubuntu18” {
14   // cloud3-t használja
15   name = ”ubuntu18.04”
16   url = ”http://cloud−images.ubuntu.com/bionic/current/bionic−server−cloudimg−amd64.img”
17 }
18 resource ”circle3_vm” ”basic” {
19   // cloud3-t használja
20   name          = ”terraform”
21   ci_user_data  = file(”${path.module}/user−data.yaml”)
22   disks         = [circle3_disk.ubuntu18_pulumi.id]
23 }
24 resource ”circle3_disk” ”ubuntu18_pumi” {
25   provider = circle3.pumi
26   name = ”ubuntu18.04−pumi”
27   url = ”http://cloud−images.ubuntu.com/bionic/current/bionic−server−cloudimg−amd64.img”
28 }
29 resource ”circle3_vm” ”basic_pumi” {
30   provider      = circle3.pumi
31   name          = ”terraform”
32   ci_user_data  = file(”${path.module}/user−data.yaml”)
33   disks         = [circle3_disk.ubuntu18_pumi.id]
34 }
```

3.5. Kód. Több adatközpont használata

A Terraform lehetőséget nyújt speciális meta-argumentumokkal több erőforrás létrehozására egyetlen erőforrásblokkból. Például a lenti példában felhasználók egy csoportjának (Programozás alapjai 2 hallgatói) indítunk virtuális gépet egy adott sablonból.

```
data "circle3_group" "prog2" {
  name = "Prog2"
}
data "circle3_template" "basetemplate" {
  name = "ubuntu v1"
}
resource "circle3_vm" "each_users" {
  for_each = toset(data.circle3_group.prog2.users)
  name = "vm-${each.key}" // a felhasználó azonosítója
  owner = each.key
  from_template = data.circle3_template.basetemplate.id
}
```

3.6. Kód. For-each használata

A for-each hátránya, hogy a felhasználók számával megegyező REST kérés kerül elküldésre, így terheli a kiszolgáló szervert, de cserébe mivel minden létrehozott erőforrás külön állapotban kódolható, az állapotváltozások kezelése egyszerűbb. Ellenben a tömbösített (vmpool) megoldással, ahol a sok virtuális gép készítését a kiszolgálóra bízzuk és csak egy kérést küldünk el, valamint ez a megoldás csak olyan esetben használható, ahol sablonból szeretnénk a virtuális gépet létrehozni.

```
data "circle3_group" "prog2" {
  name = "Prog2"
}
data "circle3_template" "basetemplate" {
  name = "ubuntu v1"
}
resource "circle3_vmpool" "pool_users" {
  name = "vm pool"
  from_template = data.circle3_template.basetemplate.id
  users = data.circle3_group.prog2.users
}
```

3.7. Kód. VMPool használata

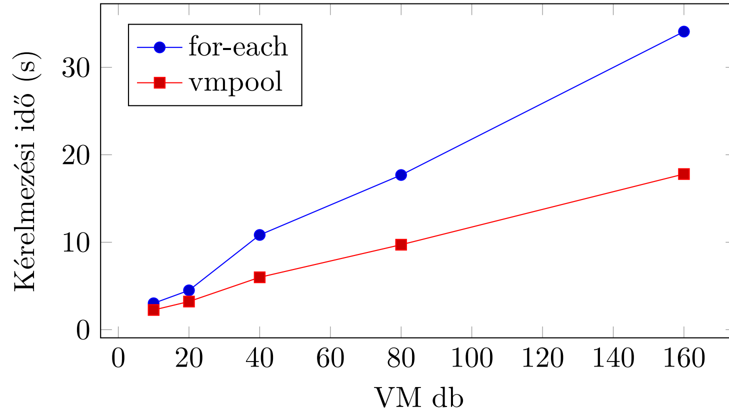
A fenti megoldásokkal könnyen kezelhetők a nagy hallgató létszámmal működő laborok vagy vizsgák automatizációja: Egy sablonból a virtuális gépek könnyen indíthatóak akár több adatközpontnál elosztva is a kihasználtság alapján. Majd hozzátársíthatóak a felhasználókhoz a virtuális gépek labor kezdetén. Akár komplexebb környezet kialakítása is lehetséges: például különböző VLAN-okhoz társított vm-ek csapatonként.

3.3.1. Mérés

A 3.1 ábra több virtuális gép konfigurálásának időigényét mutatja, a mért idő csak a virtuális gép készítésének idejét veszi figyelembe, annak felbootolását már nem. Látható hogy a sok REST kérés jelentősen lassítja az infrastruktúra készítési idejét. A tesztkörnyezetben 32GB RAM és 40CPU mag állt rendelkezésre.

3.4. Dinamikus környezet építése

Kutatási és tudományos céllal számos virtuális gépet használnak a CIRCLE menedzserben. Ezeken a VM-eken tipikusan erősen erőforásszükséges feladatokat futtatnak, így ezek jelentősen befolyásolják más virtuális gépek felhasználói élményét. Ezért elsősorban az



3.1. ábra. A for-each meta-argumentum és a pool erőforrás összehasonlítása.

ilyen projekteket aktívan az esti órákban és a hétvégén futtatják, illetve kellene futtatni az egyenletesebb erőforráskihasználás miatt.

Nagy hátrány a jelenlegi rendszerben, hogyha komplex infrastruktúra szükséges egy kutatási projekthez, akkor annak konfigurálása jelentős többletterhet okoz. Az IaC alkalmazásának nagy előnye, hogy ezek ilyenkor is könnyen felépíthetőek.

Munkák ütemezése

A minél jobb erőforráskihasználás miatt szükséges elméleti szinten megvizsgálni az erőforrások használatát. Így a feladat modellezésével olyan absztrakciókat vehetünk be, amivel sikerül az egyes feladatokat általánosan megfogalmazni, hogy azokra már adott legyen valamilyen tudományos megoldás. Legyen egy adatközpontunk és tekintsünk el az eltérő konfigurációktól, azaz minden VM azonos erőforrásokkal rendelkezzen. Ekkor az adatközponthoz maximálisan N db virtuális gép indítható úgy, hogy azoknak elfogadható legyen a teljesítményük, de aktív használatnál minél jobban használják ki az erőforrásokat. Legyen továbbá J_i munkák, amiket úgy kell az N db virtuális gépre ütemezni, hogy valamilyen célfüggvény szerint optimális eredményt kapjunk (pl. átlagos legrövidebb idő) és egyszerre csak egy munkát tudnak "végrehajtani" és egy munka tovább nem bontható. Pontosabban az N darab vm nem örökéletű gépek, mert ezek közben megszűnnek, újraindulnak, leállnak, de az aktív virtuális gépekből kb. N darab van mindig.

A fenti megszorítások bevezetésével eltekinthetünk attól is, hogy az egyes virtuális gépeket melyik "worker" gépekre ütemezzük, mivel feltéteztük a homogenitásukat.

Ekkor pl. egy vizsga vagy laborfoglalkozás lebonyolításakor a hozzá taratozó J_k (k - a hallgatók száma) munkák kezdési és befelyezési ideje előre meghatározott, azokat nem lehet átütemezni. Ugyanakkor a tudományos és kutatási céllal indított virtuális gépek munkái többnyire szabadon ütemezhetőek, de itt is lehetnek megszorítások, például egy adott munkának el kell készülnie határidőre, vagy bizonyos munkákat nem lehet párhuzamosan végezni, esetleg valamilyen precedencia áll fenn közöttük. Míg a laboros feladatoknál a szükséges "végrehajtási idő" előre ismert, addig a tudományos területeken ez többnyire csak becsülhető, így ez is megnehezíti ezen munkák ütemezését. Vannak továbbá olyan rendszerkomponensek amik fixen lefoglalnak bizonyos erőforrásokat, továbbá olyan előre nem látható feladatok beérkezésére is számítani kell, amiket szintén megfelelő minőségben kell kiszolgálni.

Elsősorban egy aktív (futó) virtuális gépnél a hozzá csatolt memória a legfőbb akadálya más VM gép használatának. Hiszen a memóriát nem lehet olyan könnyen dinamikusan

változtatni, ellenben, ha a virtuális gép aktív, és CPU kihasználtsága alacsony, más virtuális gépek teljesen függetlenül használhatják a CPU erőforrásokat. Nagyobb felhőszolgáltatók ugyanakkor többnyire alkalmazzák a ”ballooning” technikát: Ha nagyobb memóriára van szükség (pl. egy új virtuális gépnek), akkor a hypervisor egy virtuális gépen speciális programmal elkezd memóriát foglalni és kiszorítja így az ott futó folyamatokat is (ez a ballonozás - felfújja a foglalt memóriát). Majd aztán az így megszerzett memóriát a másik virtuális géphez társítja. [10]

Az ütemezéselmélet aktívan kutatott terület, míg egy feldolgozó esetén ismertek hatékony algoritmusok, de 2 vagy több feldolgozónál a probléma NP-nehéz, csak hatékony közelítő algoritmusok vannak. Mivel itt IaaS rendszerről van szó, az ütemezés csak a virtuális gépekre terjed ki, és más megoldást kell alkalmazni az eltérő típusú felhőszolgáltatásoknál. [28] [9]

A dolgozat nem tér ki a feladatok ütemezésére, ugyanakkor az egyes munkák életciklusát a Terraform eszközzel könnyen tudjuk menedzselni. Az előző szakaszban bemutatott megoldás alkalmas a laborok és vizsgák során használt virtuális gépek kezelésére. Akár lehet egy órarend alapján működő ütemezőt is felállítani. A kutatási és tudományos céllal használt infrastruktúrák ugyanakkor komplexebbek és dinamikusan változhatnak a számításhoz használt virtuális gépek száma.

Megvalósítás

A megvalósításra több lehetőségünk is van: Használhatjuk a korábban már bemutatott for-each meta-argumentumot, illetve a *count* argumentumot is, ami a megadott számú erőforrást hozza létre. Utóbbinál a testre szabhatóság korlátozott, viszont a for-each-nél egy megfelelő kulcs-érték tárolóval változó konfiguráció is építhető.

```
1 resource ”circle3_vm” ”cluster” {
2    count = 4
3    name = ”node−${count.index}”
4    from_template = data.circle3_template.basetemplate.id
5 }
```

3.8. Kód. Count használata

A for-each összetett alkalmazására látunk egy példát a következő példakódban. Mivel célunk az, hogy dinamikusan változtassuk az éppen futó virtuális gépeket, a virtuális gépek állapotát egy külső fájlból vagy környezeti változóból is beolvashatjuk. Fontos, hogy többnyire nem kell ilyenkor az éppen felesleges vm-eket megsemmisíteni, elég megállítani őket, ekkor bár tárhelyet foglalnak, de más erőforrásokat nem igényelnek. Ezért is fontos a virtuális gépek állapotának kezelése és az állapotváltozásra való reagálás.

Ha a Terraform például a klaszter infrastruktúra központi gépén fut, akkor a `local-exec` funkcióval frissíthetjük a konfigurációs paramétereket: Például egy SLURM klaszter esetén az aktív node-okat frissíthetjük, vagyha megváltozik a RAM mérete, akkor azt is be tudjuk állítani a központi gépen.

```
1 variable ”vms” {
2   type = map(object({
3     status = string
4     ram = number
5   }))
6 }
7 resource ”circle3_vm” ”cluster” {
8   for_each = var.vms
9   name = ”${each.key}”
10   ram_size = each.value.ram
11   status   = each.value.status
12   from_template = data.circle3_template.basetemplate.id
```

```
  provisioner "local-exec" {
    command = "echo ${self.ipv4} > ip.txt"
  }
}
```

3.9. Kód. For-each használata

A távoli állapotok támogatására az előző fejezetben bemutatott Postgres adatbázismotort érdemes használni. Ennek az is előnye, hogy bárhonnan hozzáférünk a tábla tartalmához, azaz az állapothoz. A lenti példában egy admin nevű felhasználó használja a távoli állapotot a "demo" projekthez.

```
terraform {
  required_providers {
    circle3 = {
      version = "0.1"
      source  = "bmeik/tf/circle3"
    }
  }
  backend "pg" {
    conn_str = "postgres://admin:<pass>@pg.cloud3.cloud.bme.hu/demo"
  }
}
// [infrastruktúra kód]
```

3.10. Kód. Remote state

A dinamikus viselkedéshez nagy segítség, hogy lehetőség van külső program meghívására is, és aztán a program kimenetét felhasználhatjuk a konfigurációban. Ez jól jöhet olyan helyzetekben, amikor a Terraform lehetőségei kimerülnek. Ilyen lehet például az aktuális felhőközpontok terheltsége alapján elosztani a virtuális gépeket, vagy beolvasni egy fájlból, kik azok a hallgatók akik jelentkeztek egy vizsgára és akár más-más feladathoz is elő lehet készíteni a környezetet a számukra.

```
import json, time
import circle3

max_vm = 100
pumi = circle3.connect("pumi.cloud.bme.hu")
wombat = circle3.connect("wombat.cloud.bme.hu")
pulumi_vm, wombat_vm = circle3.balance(pumi, wombat, max_vm)
result = {
  "pumi": pumi_vm,
  "wombat": wombat_vm
}
print(json.dumps(result))
```

3.11. Kód. "balancer.py"

Ezután a fenti kódot a lenti terraform konfigurációval tudjuk használni: A program bekötésére az external adatforrás szolgál ami lefuttatja a programot és a standard kimenetet feldolgozza (szabályos json kimenet).

```
data "external" "cnt" {
  program = ["python3", "${path.module}/balancer.py"]
}
resource "circle3_vm" "vms_pumi" {
  provider = circle3.pumi
  count = data.external.cnt.result.pumi
  name = "node-${count.index}"
  from_template = data.circle3_template.basetemplate.id
}
resource "circle3_vm" "vms_wombat" {
  provider = circle3.wombat
```

```
12   count = data.external.cnt.result.wombat
13   name = "node—${count.index}"
14   from_template = data.circle3_template.basetemplate.id
15 }
```

3.12. Kód. External adatforrás használata

3.5. Adaptáció Jenkins-el

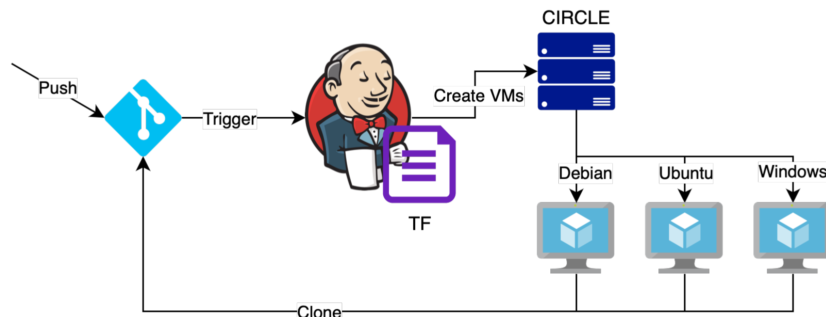
A Jenkins egy népszerű, nyílt forráskódú automatizálási kiszolgáló, elsősorban a szoftverprojektek folyamatos integrációjának megvalósítására. [18]

A Jenkins-ben megadhatunk különböző projekteket, amiket aztán tudunk majd ütemezni. Ilyen lehet egy freestyle project vagy pipeline, amivel komplexebb folyamatok is testre szabhatóak. Támogatja a verziókezelést, illetve a megfelelő plugin-okkal képes egy git tárolóból letölteni az éppen aktuális szoftververziót és annak a tesztelését elvégezni.

A Jenkins-t alkalmazhatjuk a CI/CD folyamatok végrehajtására is, anélkül, hogy például bizonyos szenzitív adatokat valamelyik verziókezelőn kellene tárolni: A verziókezelő értesítést küld a Jenkins kiszolgálónak hogy pl. új release érhető el, ekkor a szerver lehúzza a módosításokat, megsemmisíti a korábbi infrastruktúra kiszolgálót, és egy újat épít fel a legújjabb verzióval. (Automated Continuous Integration/Delivery, [23]) Erre láthatunk egy példa környezetet a 3.2. ábrán.

A Jenkins-et triggereken keresztül is tudjuk irányítani, ilyen lehet a circle webhook-ra való beregisztrálás: Például ha a virtuális gép lejárt és ezért a CIRCLE leállítja, akkor a visszahívással automatikusan frissíthető az állapota.

Egy központi Jenkins szerver használatával az ütemezett laborokhoz szükséges virtuális gépek automatikus indítása és megsemmisítése is megoldott, ha szükséges.



3.2. ábra. Jenkins és Terraform hagyományos devops feladatokra

Összegzés

A dolgozatban röviden bemutattam a cloud technológiák főbb összetevőit, elsősorban az Infrasturcture as Code technológiát ismertettem, ami a dolgozat fő fókusza volt. Az ehhez kapcsolódó cloud-init konfigurációs lehetőséget is bemutattam.

A fenti eszközöket implementáltam az egyetemi felhőinfrastruktúra (CIRCLE) klónjához. Az implementáció során elkészült a cloud-init támogatása a rendszerhez és bővítettem a rendszert a megfelelő végpontokkal a Terraform IaC eszköz használatához. A Terraformhoz szükséges go nyelven írt kliens modult is elkészítettem a legfontosabb funkciókhoz illetve a provider is elkészült, amivel a Terraform-on keresztül tudjuk "programozni", menedzselni az infrastruktúrákat a rendszerben.

Bemutattam az IaC technológia alkalmazásának lehetőségeit oktatási, kutatási és tudományos környezetben egyaránt. Vizsgáltam a telejesítménybeli szempontokat is. Kijelenthető, hogy ezen eszközök alkalmazása elősegíti az oktató gárdára nehezedő adminisztratív jellegű feladatok csökkentését, a használt infrastruktúra automatizázióját és menedzselését. A technológia bizonyos megkötésekkel alkalmazható dinamikus környezet kialakítására is.

A 22/23 tavaszi félév elejére munkám folytatásaként a rendszer teljes funkcionalitását a hallgatók és az oktatók rendelkezésére szeretném bocsájtani kezdetben a Smallville adatközpont egy részének bevonásával, majd később az egész csomópont használatával. A cloud-init konfigurációs lehetőséget korábban oktatók is hiányolták, valamint a rendszer nem volt automatizálható, nem volt egységes interfész a virtuális gépek nem grafikus/webes kezelésére, ami az éles rendszerekben is alkalmazható lett volna, amit pedig a kutatási oldal hiányolt elsősorban. Így a Terraform modulhoz is a rendszer teljes funkcionalitását (VLAN-ok kialakítása, domain-nevek regisztrálása) szeretném elérhetővé tenni IaC által felügyelt kódból, valamint a provider modult publikálni szeretném a Terraform Registry tárolóba, hogy hivatalosan is elérhető legyen, bármilyen Terraform projekt esetén.

Rövidítések

ACI	Automated Continuous Integration
ACL	Access-control list
ACT	Asynchronous Completion Token
AMQP	Advanced Message Queuing Protocol
ARGON	An infRastructure modellinG tool for clOud provisioNing
CFG	Context Free Grammar
CI/CD	Continuous Integration/Delivery
CIRCLE	Cloud Infrastructure for Research Computing and Laboratory Environment
CM	Configuration Management
DAG	Directed Acyclic Graph
DFS	Depth First Search
Devops	Development and operations
HCL	HashiCorp Configuration language
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
NFS	Network File System
OCCI	Open Cloud Computing Interface
OpenIaC	Open Infrastructure as Code
PEG	Parsing Expession Grammar
PaaS	Platform as a Service
RE	Regular expression
RPC	Remote Procedure Call
RRIP	Rock Ridge Interchange Protocol
SLURM	Simple Linux Utility for Resource Management
SSO	Single Sign On
SaaS	Software as a Service
TF	Terraform
VLAN	Virtual Local Area Network
VM	Virtual machine
VPC	Virtual Private Cloud
VPN	Virtual Personal Network

Ábrák jegyzéke

1.1. CIRCLE rendszer felépítése 4
1.2. Terraform által generált terv-gráf Azure szolgáltatóval 9
1.3. Diszjunkt részgráfok párhuzamos DFS-hez (forrás: [15]) 9
1.4. A Terraform működési váza (terraform.io - 2022.10.03.) 10

2.1. Adatmodell (csak változások) 14
2.2. Folyamatábra a cloud-init lemez kezeléséhez 15
2.3. A Terraform provider és a CIRCLE rendszer működése 18
2.4. A távoli hívás szemléltetése (egyszerűsített szekvencia diagram) 20
2.5. Példa a webhook alkalmazására 22

3.1. A for-each meta-argumentum és a pool erőforrás összehasonlítása. 29
3.2. Jenkins és Terraform hagyományos devops feladatokra 32

Irodalomjegyzék

[1] Deepak Ajwani – Adan Cosgaya-Lozano – Norbert Zeh: A topological sorting algorithm for large graphs. *ACM J. Exp. Algorithmics*, 17. évf. (2012. sep). ISSN 1084-6654. URL https://doi.org/10.1145/2133803.2330083. 21 p.

[2] Michael Armbrust – Armando Fox – Rean Griffith – Anthony D. Joseph – Randy Katz – Andy Konwinski – Gunho Lee – David Patterson – Ariel Rabkin – Ion Stoica – Matei Zaharia: A view of cloud computing. *Commun. ACM*, 53. évf. (2010. apr) 4. sz., 50–58. p. ISSN 0001-0782. URL https://doi.org/10.1145/1721654.1721672. 9 p.

[3] Azure aztfy tool. https://github.com/Azure/aztfy. 2022.10.28.

[4] Antonia Bertolino – Guglielmo De Angelis – Micael Gallego – Boni García – Francisco Gortázar – Francesca Lonetti – Eda Marchetti: A systematic review on cloud testing. *ACM Comput. Surv.*, 52. évf. (2019. sep) 5. sz. ISSN 0360-0300. URL https://doi.org/10.1145/3331447. 42 p.

[5] Brian D. Carrier: Different interpretations of iso9660 file systems. *Digital Investigation*, 7. évf. (2010), S129–S134. p. ISSN 1742-2876. URL https://www.sciencedirect.com/science/article/pii/S1742287610000435. The Proceedings of the Tenth Annual DFRWS Conference.

[6] Cloud-init documentation. https://cloudinit.readthedocs.io/. 2022.10.11.

[7] cloud-init support for virtual machines in azure. https://learn.microsoft.com/en-us/azure/virtual-machines/linux/using-cloud-init. 2022.10.24.

[8] Carlos Serodio Daniel Gomes, Pedro Mestre: Infrastructure-as-code for scientific computing environments. In *CENTRIC 2019, The Twelfth International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services* (konferenciaanyag). Valencia, Spain, 2019, CENTRIC 2019, 7–10. p. ISBN 978-1-61208-754-2. 4 p.

[9] Yash P. Dave – Avani S. Shelat – Dhara S. Patel – Rutvij H. Jhaveri: Various job scheduling algorithms in cloud computing: A survey. In *International Conference on Information Communication and Embedded Systems (ICICES2014)* (konferenciaanyag). 2014, 1–5. p.

[10] Ye Ding – He Xu – Peng Li – Jie Ding: Review of virtual memory optimization in cloud environment. In *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)* (konferenciaanyag). 2016, 53–58. p.

[11] Django rest framework. https://www.django-rest-framework.org/. 2022.10.11.

[12] Django web framework. https://www.djangoproject.com. 2022.10.21.

[13] Evolving your infrastructure with terraform: Opencredo's 5 common terraform patterns (konferenciaanyag). https://www.hashicorp.com/resources/evolving-infrastructure-terraform-opencredo. 2022.10.11.

[14] Bryan Ford: Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39. évf. (2004. jan) 1. sz., 111–122. p. ISSN 0362-1340. URL https://doi.org/10.1145/982962.964011. 12 p.

[15] Ananth Grama – George Karypis – Vipin Kumar – Anshul Gupta: *Introduction to Parallel Computing*. Second. kiad. 2003, Addison-Wesley. ISBN 0201648652 9780201648652.

[16] Irfan Habib: Virtualization with kvm. *Linux J.*, 2008. évf. (2008. feb) 166. sz. ISSN 1075-3583.

[17] Bill Howe: Virtual appliances, cloud computing, and reproducible research. *Computing in Science and Engineering*, 14. évf. (2012) 4. sz., 36–41. p.

[18] Jenkins. https://www.jenkins.io/. 2022.10.21.

[19] Jinja template engine. https://jinja.palletsprojects.com/. 2022.10.11.

[20] Open vswitch. https://www.openvswitch.org/. 2022.10.21.

[21] Alan P. Parkes: *Elements of Formal Languages*. London, 2002, Springer London, 11–36. p. ISBN 978-1-4471-0143-7. URL https://doi.org/10.1007/978-1-4471-0143-7_2.

[22] Akond Rahman – Chris Parnin – Laurie Williams: The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (konferenciaanyag). 2019, 164–175. p.

[23] Syeda Gazala Rizvi – G. S. Mamatha: Automated continuous integration (aci) scheme based on jenkins. In S. Smys – Ram Palanisamy – Álvaro Rocha – Grigorios N. Beligiannis (szerk.): *Computer Networks and Inventive Communication Technologies* (konferenciaanyag). Singapore, 2021, Springer Nature Singapore, 915–925. p. ISBN 978-981-15-9647-6.

[24] Chunming Rong – Jiahui Geng – Thomas J. Hacker – Haakon Bryhni – Martin G. Jaatun: Openiac: open infrastructure as code - the network is my computer. *Journal of Cloud Computing*, 11. évf. (2022) 1. sz., 12. p. URL https://doi.org/10.1186/s13677-022-00285-7.

[25] Julio Sandobalin – Emilio Insfran – Silvia Abrahao: An infrastructure modelling tool for cloud provisioning. In *2017 IEEE International Conference on Services Computing (SCC)* (konferenciaanyag). 2017, 354–361. p.

[26] Douglas C. Schmidt: Asynchronous completion token. 1998.

[27] Klaus Simon: Finding a minimal transitive reduction in a strongly connected digraph within linear time. In Manfred Nagl (szerk.): *Graph-Theoretic Concepts in Computer Science* (konferenciaanyag). Berlin, Heidelberg, 1990, Springer Berlin Heidelberg, 245–259. p. ISBN 978-3-540-46950-6.

[28] E. Taillard: Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64. évf. (1993) 2. sz., 278–285. p. ISSN 0377-2217. URL `https://www.sciencedirect.com/science/article/pii/037722179390182M`. Project Management anf Scheduling.

[29] Terraform backends. `https://developer.hashicorp.com/terraform/language/settings/backends/remote`. 2022.10.21.

[30] Terratest. `https://terratest.gruntwork.io/`. 2022.10.15.

[31] Orazio Tomarchio – Domenico Calcaterra – Giuseppe Di Modica: Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, 9. évf. (2020) 1. sz., 49. p. URL `https://doi.org/10.1186/s13677-020-00194-7`.

[32] Types of cloud computing. `https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud`. 2022.10.10.

[33] Steve Vinoski: Advanced message queuing protocol. *IEEE Internet Computing*, 10. évf. (2006) 6. sz., 87–89. p.

[34] Karsa Zoltán István és Dr. Szeberényi Imre: A circle felhő elmúlt évtizede. In *NetworkShop 2022 Konferencia* (konferenciaanyag). Debrecen, 2022.

[35] Kovács József és Kacsuk Péter: Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures. 2018. URL `https://doi.org/10.1007/s10723-017-9421-3`.