Tampereen ammattikorkeakoulu

Serverless-sovellusten hallinta Terraform-työkalulla

Petri Ranta

OPINNÄYTETYÖ
Helmikuu 2025

Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

RANTA, PETRI:
Serverless-sovellusten hallinta Terraform-työkalulla

Opinnäytetyö 46 sivua, joista liitteitä 0 sivua
Helmikuu 2025

Pilvipalvelut ovat keskeinen osa nykyaikaista ohjelmistokehitystä. Ne tarjoavat yrityksille joustavuutta, kustannustehokkuutta sekä mahdollisuuden hyödyntää erilaisia ohjelmistokehitysresursseja ilman tarvetta omalle infrastruktuurille. Erityisesti serverless-arkkitehtuuri ja infrastruktuuri koodina ovat mullistaneet tapaa, jolla sovelluksia ja infrastruktuuria kehitetään ja hallitaan. Opinnäytetyön taustalla oli toimeksiantajan tarve ratkaisulle, jonka avulla näitä menetelmiä voidaan hyödyntää usealla eri pilvipalvelualustalla.

Opinnäytetyön tarkoituksena oli kehittää Terraform-konfiguraatiot, joiden kautta voidaan määritellä toisiaan vastaavat serverless-sovellukset Amazon Web Services, Microsoft Azure ja Google Cloud -pilvipalvelualustoilla. Opinnäytetyön keskeisenä tavoitteena oli syventää toimeksiantajan ymmärrystä Terraformista ja pilvi-infrastruktuurin hallinnasta sen avulla. Tämä tukee organisaation pilviosaamista, joka taas puolestaan vahvistaa kykyä reagoida liiketoiminnan tarpeisiin.

Työn tuloksena syntyi kolme yhtenäistä ja modulaarista Terraform-konfiguraatiota, jotka koostuvat tietokannan, siihen liitetyn FaaS-palvelun sekä ohjelmointirajapinnan määrittelyistä. Modulaarisuus mahdollistaa komponenttien uudelleenkäytön eri projekteissa, mikä vähentää ylimääräistä työtä ja nopeuttaa uusien sovellusten kehittämistä. Konfiguraatioita voidaan myös käyttää rinnakkain useissa eri kehitysympäristöissä.

Tulevaisuuden jatkokehitys- ja tutkimusmahdollisuuksia löytyy mm. konfiguraatioiden integroinnista CI/CD-automaatioon, lisätyökalujen, kuten Terragruntin käyttömahdollisuuksien selvittämisestä sekä infrastruktuurin tietoturvan parantamisesta.

Asiasanat: infrastruktuuri koodina, terraform, serverless, pilvipalvelut

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Production

RANTA, PETRI:
Serverless Application Management with Terraform

Bachelor's thesis 46 pages, appendices 0 pages
February 2025

Cloud services are a fundamental part of modern software development. They provide businesses with flexibility, cost-efficiency, and access to various development resources without the need for dedicated infrastructure. Serverless architecture and infrastructure as code have greatly changed the way applications and infrastructure are developed and managed. This thesis was initiated by the client's need for a solution that enables the use of these methods across multiple cloud service providers.

The objective of the thesis was to develop Terraform configurations for deploying equivalent serverless applications on Amazon Web Services, Microsoft Azure, and Google Cloud. The main goal of the thesis was to deepen the client's understanding of Terraform and cloud infrastructure management. This strengthens the client's cloud expertise and their ability to respond to business needs effectively.

As a result of the work, three unified and modular Terraform configurations were created. These consist of the definitions of a database, a function as a service component linked to it, and an application programming interface. The configurations can be deployed simultaneously across multiple development environments.

Future development and research opportunities include integrating the configurations with CI/CD automation, exploring additional tools such as Terragrunt, and enhancing infrastructure security.

Key words: infrastructure as code, terraform, serverless, cloud services

SISÄLLYS

LYHENTEET JA TERMIT

AWS	Amazon Web Services, pilvipalveluntarjoaja
Azure	Microsoft Azure, pilvipalveluntarjoaja
API	Application programming interface, ohjelmointirajapinta
CI	Continuous integration, jatkuva integrointi
CD	Continuous delivery/deployment, jatkuva käyttöönotto
DSL	Domain-specific language, täsmäkieli
FaaS	Function as a Service, funktio palveluna
Google Cloud	Pilvipalveluntarjoaja
GPL	General-purpose language, yleiskäyttöinen kieli
HCL	HashiCorp Configuration Language
IaC	Infrastructure as Code, infrastruktuuri koodina
IaaS	Infrastructure as a Service, infrastruktuuri palveluna
PaaS	Platform as a Service, alusta palveluna
SaaS	Software as a Service, ohjelmisto palveluna
Serverless-arkkitehtuuri	Palvelimeton arkkitehtuuri
Terraform	Infrastruktuuri koodina -työkalu

1 JOHDANTO

Nykyaikaisessa ohjelmistokehityksessä pilvipalvelut ovat nousseet keskeiseksi osaksi digitaalisten ratkaisujen toteutusta. Pilvipalvelut tarjoavat yrityksille joustavuutta, kustannustehokkuutta ja mahdollisuuden hyödyntää monipuolisia resursseja ilman merkittäviä investointeja omaan infrastruktuuriin. Erityisesti serverless-arkkitehtuuri on yleistynyt, sillä se mahdollistaa sovellusten suorittamisen ilman tarvetta hallinnoida omia palvelimia. Serverless-arkkitehtuuri tukee nopeaa kehitystä ja automaattista skaalautuvuutta, mikä vastaa modernin liiketoiminnan tarpeita.

Infrastruktuuri koodina (IaC) mahdollistaa infrastruktuurin nopean ja toistettavan hallinnan koodin avulla, mikä tehostaa resurssien käyttöönottoa, vähentää riskejä sekä mahdollistaa kustannustehokkaiden ja luotettavien järjestelmien rakentamisen. Terraform on yksi tunnetuimmista IaC-työkaluista, ja sen vahvuutena on kyky hallita monen eri pilvipalvelualustan resursseja yhtenäisellä tavalla. Tämä tekee Terraformista arvokkaan työkalun erityisesti organisaatioille, jotka toimivat useissa pilviympäristöissä tai suunnittelevat siirtymistä hybridipilviarkkitehtuuriin.

Opinnäytetyön tavoitteena oli syventää toimeksiantajana toimivan Amabit Oy:n ymmärrystä Terraformin mahdollisuuksista ja rajoitteista eri pilviympäristöissä, mikä tukee organisaation infrastruktuuriosaamista kehitysprojekteissa, lisäten samalla mahdollisuuksia uusien asiakassuhteiden luomiseen ja vanhojen kehittämiseen. Lisäksi työ tuo esiin, kuinka Terraform voi yhtenäistää serverless-resurssien hallintaa ottaen huomioon eri resurssien samankaltaisuudet ja erot, ja täten auttaa lukijaa valitsemaan resurssit tarpeidensa mukaan.

Työn tarkoituksena oli luoda Terraform-konfiguraatiot, joiden avulla voidaan luoda toisiaan vastaavat serverless-sovellukset AWS-, Azure-, ja Google Cloud -pilvipalvelualustoille. Toteutukset sisältävät määrittelyt serverless-infrastruktuurille, johon sisältyy tietokanta, siihen liitetty funktio, sekä ohjelmointirajapinta (API). Nämä esimerkkitoteutukset toimivat pohjana, josta toimeksiantaja voi kehittää ja soveltaa serverless-ratkaisujaan Terraformin avulla.

2 PILVIPALVELUT

2.1 Pilvipalveluiden perusteet

Pilvi tarkoittaa erillään olevaa IT-ympäristöä, joka on suunniteltu skaalautuvien ja mitattavien IT-resurssien etähallintaan. Pilvipalvelu taas on mikä tahansa IT-resurssi, joka on tehty etäkäyttöön pilven kautta. Pilvipalvelu voi olla esimerkiksi yksinkertainen verkkopohjainen ohjelma, jota käytetään viestintäprotokollan kautta, tai se voi toimia etähallintapisteenä laajemmille ympäristöille, hallintatyökaluille ja muille IT-resursseille. (Erl & Barcelo 2023.)

Pilvipohjaisia IT-resursseja tarjoavaa organisaatiota kutsutaan pilvipalveluntarjoajaksi. Ottaessaan pilvipalveluntarjoajan roolin organisaatio on vastuussa pilvipalvelujen tarjoamisesta asiakkaiden saataville palvelutasosopimusten mukaisesti. (Erl & Barcelo 2023.)

Pilvipalvelut jaotellaan yleensä julkiseen pilveen, yksityiseen pilveen, monipilveen, sekä hybridipilveen. Julkinen pilvi on julkisesti saatavilla oleva pilviympäristö, jonka omistaa kolmannen osapuolen pilvipalveluntarjoaja – tunnettuja toimijoita ovat esimerkiksi Amazon Web Services, Microsoft Azure ja Google Cloud. Yksityinen pilvi taas on vain yhden organisaation omistuksessa. Yksityiset pilvet mahdollistavat pilvipalveluteknologian hyödyntämisen organisaation sisäisten IT-resurssien keskittämisessä. Monipilvimallissa kuluttajaorganisaatio käyttää usean eri pilvipalveluntarjoajan tarjoamia palveluita ja resursseja. Hybridipilveksi kutsutaan useammasta edellä mainitusta mallista koostuvaa ympäristöä. (Erl & Barcelo 2023.)

2.2 Pilvipalvelumallit

Pilvipalveluilla on useita palvelumalleja, joista tunnetuimmat ovat infrastruktuuri palveluna (Infrastructure as a Service, IaaS), alusta palveluna (Platform as a Service, PaaS), sekä ohjelmisto palveluna (Software as a Service, SaaS). IaaS, PaaS ja SaaS eivät sulje toisiaan pois, ja monet yritykset käyttävät useampaa kuin yhtä. (IBM n.d.) Neljäntenä pilvipalvelumallina käsitellään vielä funktio

palveluna -mallia (Function as a Service, FaaS), joka on avainasemassa serverless-arkkitehtuureissa.

IaaS-malli edustaa itsenäistä IT-ympäristöä, joka koostuu infrastruktuurikeskeisistä resursseista, joita voidaan käyttää ja hallita pilvipalvelupohjaisten rajapintojen ja työkalujen kautta. IaaS-ympäristön tarkoituksena on tarjota pilvipalvelun käyttäjälle vastuu ja hallinta sen konfiguroinnista ja käytöstä. (Erl & Barcelo 2023.) Julkiset pilvipalveluntarjoajat, kuten AWS, Microsoft Azure ja Google Cloud, ovat esimerkkejä IaaS:stä (Red Hat 2022).

PaaS-malli tarjoaa pilvipohjaisen alustan sovellusten kehitykseen, suorittamiseen ja hallitsemiseen. Pilvipalveluntarjoaja ylläpitää ja hallitsee kaikkia alustaan kuuluvia laitteistoja ja ohjelmistoja, kuten palvelimia, käyttöjärjestelmiä, tallennustilaa, ja tietokantoja. PaaS-mallin suurin hyöty tulee siitä, että se mahdollistaa sovellusten luomisen, testaamisen, käyttöönoton, päivittämisen, sekä skaalaamisen nopeasti ja kustannustehokkaasti. Tunnettuja PaaS-ratkaisuja ovat esimerkiksi AWS Elastic Beanstalk, Microsoft Azure App Services ja Google App Engine. (IBM n.d.)

SaaS-malli tarjoaa pilvipalvelussa olevan käyttövalmiin ohjelmiston. Käyttäjät maksavat yleensä kuukausi- tai vuosimaksua sovelluksen käyttämisestä. Sovellusta ja kaikkea sen toimittamiseen tarvittavaa infrastruktuuria isännöi ja hallinnoi SaaS-toimittaja. Suosittuihin SaaS-ratkaisuihin kuuluu mm. Trello, Slack ja Canva. (IBM n.d.)

FaaS on pilvipalvelumalli, jossa kehittäjät voivat ajaa sovellustoimintoja ilman tarvetta ylläpitää taustalla olevaa omaa infrastruktuuria, kuten palvelimia tai virtuaalikoneita. FaaS-ympäristössä pilvipalveluntarjoaja hoitaa kaiken palvelinpuolen logiikan ja infrastruktuurin hallinnan, jolloin kehittäjät voivat keskittyä vain koodin kirjoittamiseen ja käyttöönottoon. (DigitalOcean 2024.) Tunnettuja FaaS-palveluita ovat esimerkiksi AWS Lambda, Google Cloud Run functions sekä Azure Functions.

2.3 Serverless-arkkitehtuuri

Serverless-käsite pohjautuu ajatukseen, jossa palvelinpohjaista sovellusta voidaan ajaa ilman tarvetta hallita omaa palvelinta. Terminä serverless saattaa olla harhaanjohtava, sillä palvelin on kuitenkin aina osallisena – se on vain jonkun muun ylläpitämä. (Katzer 2021).

FaaS-malliin ja serverless-käsitteeseen viitataan usein synonyymeinä, vaikka todellisuudessa niillä on toisistaan poikkeavat määritelmät. Serverless viittaa mihin tahansa kategoriaan, jossa palvelin on täysin erotettu loppukäyttäjästä – FaaS taas on palvelimettoman laskennan kategoria, joka perustuu tapahtumapohjaisiin triggereihin, joissa koodi suoritetaan vastauksena tapahtumiin tai pyyntöihin. (DigitalOcean 2024.)

Serverless-arkkitehtuurin vahvuus on palvelimien rakentamisen ja hallinnan tarpeettomuus, joka mahdollistaa mittavat säästöt ajassa ja rahassa. Serverless-arkkitehtuurilla on myös useita muita vahvuuksia kuten esimerkiksi korkea skaalautuvuus sekä luotettavuus. Näiden ominaisuuksien avulla sovelluksen tulevaisuuden käyttökapasiteettia ei käytännössä tarvitse erikseen suunnitella muuten kuin palveluntarjoajan rajoitusten osalta. Oleellisiin vahvuuksiin kuuluu lisäksi maksu per käyttö -malli, joka mahdollistaa sen, ettei ylläpitäjän tarvitse huolehtia ja maksaa turhaan ajasta, jolloin liikenne palvelussa on vähäistä tai sitä ei ole lainkaan. (Katzer 2021.)

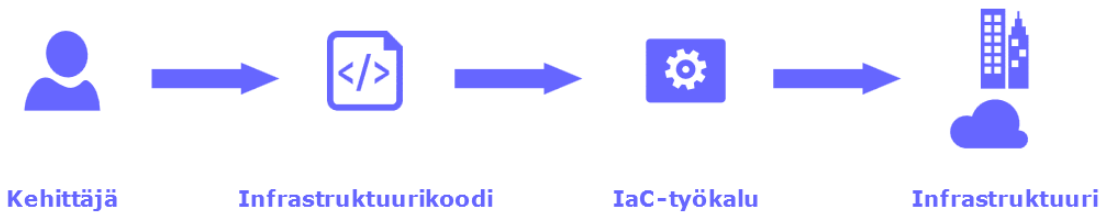
Serverless-arkkitehtuuri ei kuitenkaan ole täydellinen, vaan siihen liittyy myös joitain haasteita. Ehkä suurin suorituskyvyllinen haaste on ns. kylmäkäynnistys (cold start), joka tapahtuu, kun serverless-funktiota kutsutaan sen ollessa poissa käynnistä. Tällöin käyttäjä ei saa vastausta sovellukselta välittömästi, vaan joutuu odottamaan funktion suoritusympäristön käynnistymistä. (Cui, Nair & Sbarski 2022.) Ongelman ilmenemistä voidaan ennaltaehkäistä esimerkiksi lähettämällä heräteviesti funktiolle käyttäjän siirtyessä tietylle sivulle. Kylmäkäynnistyksen ongelmat tulevat luultavasti lieventymään palveluiden kehittyessä ajan saatossa. Haasteensa tuo myös teknologiavalintojen aiheuttama sidonnaisuus tiettyyn palveluntarjoajaan, mikäli sovelluksen tarkoituksena tai vaatimuksena on pysyä pilviagnostisena. (Katzer 2021.)

3 INFRASTRUKTUURI KOODINA

3.1 Infrastruktuuri koodina yleisesti

Infrastruktuuri koodina eli Infrastructure as Code (IaC) on infrastruktuuriautomaation lähestymistapa, jonka tavoitteena on tehdä infrastruktuurimuutoksista nopeita, turvallisia ja toistettavia koodin avulla. Tämä mahdollistaa infrastruktuurin kehittämisen ja ylläpidon samalla tavoin kuin ohjelmiston kehittämisen. (Morris 2021.)

IaC:n hyödyt organisaatioille ovat merkittäviä. Se nopeuttaa arvoa tuottavien IT-resurssien käyttöönottoa, vähentää infrastruktuurimuutoksiin liittyviä riskejä, ja tekee resursseista helposti saatavilla olevia silloin, kun niitä tarvitaan. Lisäksi se tarjoaa yhteiset työkalut kehitystiimeille, luo luotettavia ja kustannustehokkaita järjestelmiä sekä parantaa hallittavuutta ja näkyvyyttä turvallisuus- ja vaatimustenmukaisuuskysymyksissä. IaC nopeuttaa myös ongelmanratkaisua ja edistää infrastruktuurin yksinkertaista ja puhdasta suunnittelua. (Morris 2021.) Kuvassa 1 havainnollistetaan tyypillinen IaC-prosessi, jossa ensimmäisenä kehittäjä määrittelee infrastruktuurin koodina. Tämän jälkeen IaC-työkalu käsittelee ja toteuttaa infrastruktuurikoodin. Lopputuloksena syntyy määritelty ja toistettavissa oleva infrastruktuuri esimerkiksi pilvipalveluun.



KUVA 1. Infrastruktuuri koodina.

3.2 IaC-työkalun valinta

IaC-työkalua valitessaan organisaatioiden tulee arvioida huolellisesti useita tekijöitä, kuten työkalun ominaisuudet, skaalautuvuuden mahdollisuudet sekä sen yhteensopivuus mahdollisen olemassa olevan infrastruktuurin kanssa.

Terraformin lisäksi suosituimpiin IaC-työkaluihin kuuluu mm. Chef, Ansible sekä Pulumi. Chef ja Ansible ovat konfiguraationhallintatyökaluja, mikä tarkoittaa, että ne on suunniteltu asentamaan ja hallitsemaan ohjelmistoja jo olemassa olevilla palvelimilla. Terraform ja Pulumi ovat provisiointityökaluja, jotka vastaavat pääasiassa palvelimien luomisesta. Provisiointityökaluja voidaan käyttää luomaan paitsi palvelimia myös esimerkiksi tietokantoja, välimuisteja, kuormantasaajia ja monitorointiratkaisuja. (Brikman 2022.)

Chef ja Ansible käyttävät proseduraalista ohjelmointityyliä, jossa kirjoitetussa koodissa määritellään askel askeleelta, kuinka haluttu lopputila saavutetaan. Terraform ja Pulumi taas käyttävät deklaratiivista ohjelmointityyliä, jossa koodissa määritellään haluttu lopputila, ja IaC-työkalu itsessään on vastuussa tilan saavuttamisesta. (Brikman 2022.)

Brikmanin (2022) mukaan proseduraalisissa IaC-työkaluissa on kaksi merkittävää ongelmaa. Ensimmäinen ongelma on se, että proseduraalinen koodi ei täysin kuvaa infrastruktuurin tilaa. Koodista itsestään ei suoraan selviä mitä resursseja on otettu käyttöön missäkin järjestyksessä, joten kehittäjän on tiedettävä koko muutoshistoria ymmärtääkseen sen hetkistä infrastruktuuria. Toinen ongelma ilmenee uudelleenkäytettävyydessä, joka on luonnostaan rajoitettua, koska infrastruktuurin tila on otettava aina manuaalisesti huomioon. Tilan muuttuessa vanha koodi ei välttämättä enää toimi, jos se on suunniteltu muokkaamaan infrastruktuurin tilaa, jota ei enää ole olemassa. Tämän seurauksena proseduraaliset koodipohjat kasvavat ajan myötä suuriksi ja monimutkaisiksi. (Brikman 2022.)

Deklaratiivisessa lähestymistavassa koodi edustaa aina infrastruktuurin viimeisintä tilaa. Koodipohjasta voi päätellä mitä on otettu käyttöön ja miten se on konfiguroitu, ilman, että tarvitsee huolehtia historiasta. Uudelleenkäytettävän koodin luominen on myös helppoa, koska nykyistä tilaa ei tarvitse ottaa huomioon. Kehittäjän tarvitsee vain kuvailla haluttu tila, ja työkalu selvittää automaattisesti, miten päästään tilasta toiseen. Tämän ansiosta deklaratiivisen lähestymistavan koodipohjat pysyvät yleensä pieninä ja helposti ymmärrettävinä. (Brikman 2022.)

Chef ja Pulumi sallivat yleiskäyttöisen ohjelmointikielen (general-purpose programming language, GPL) infrastruktuurikoodin hallinnassa. Chef tukee Rubya ja Pulumi useita kieliä kuten JavaScript, TypeScript, Python, Go, C# ja Java. Terraform ja Ansible käyttävät täsmäkieltä (domain-specific language, DSL). Terraform tukee sen omaa HCL-kieltä, kun taas Ansible YAML-kieltä. Täsmäkielillä on lukuisia etuja verrattuna yleiskäyttöisiin kieliin: ne ovat yhtenäisiä, selkeitä ja niiden opettelu on helpompaa. Vastaavasti IaC-työkaluissa yleiskäyttöisten kielten etuna on, että uutta kieltä ei välttämättä tarvitse opetella lainkaan. Lisäksi yleiskäyttöisten kielten laajat ekosysteemit tarjoavat runsaasti valmiita työkaluja, ja niitä voi hyödyntää monipuolisesti erilaisiin ohjelmointitehtäviin. (Brikman 2022.)

Kun IaC-työkalu valitaan, valitaan samalla myös yhteisö. Monesti ekosysteemillä työkalun ympärillä on suurempi vaikutus kokemukseen työkalusta, kuin itsessään teknologian ominaisuuksilla. Lisäksi yhteisön koko määrittää kuinka paljon lisäosia, integraatioita ja laajennuksia on saatavilla, sekä kuinka helposti tietoa on saatavilla verkosta. (Brikman 2022.)

Teknologian kypsyys on tärkeä huomioitava seikka sen valinnassa. Vakiintuneet teknologiat tarjoavat selkeät käyttömallit, parhaat käytännöt ja tiedon yleisimmistä ongelmista, kun taas uudempien teknologioiden kohdalla asioita saattaa joutua opettelemaan vaikeamman kautta. Pulumi on vertailun nuorin työkalu, mikä näkyy esimerkiksi dokumentaation, parhaiden käytäntöjen ja yhteisön tuottamien moduulien rajallisuutena. (Brikman 2022.)

Terraform on hyvä valinta IaC-työkaluksi sen deklaratiivisuuden, selkeän kielen, kypsyyden, laajan ekosysteemin ja kattavan dokumentaation ansiosta. Pulumi voi olla myös järkevä valinta, jos haluaa hyödyntää yleiskäyttöistä ohjelmointikieltä. On syytä kuitenkin ottaa huomioon Pulumin nuoruus ja pienempi ekosysteemi, jotka saattavat tuottaa haasteita käyttöönotossa ja tuen saamisessa.

Chef ja Ansible puolestaan eivät välttämättä ole yhtä hyviä valintoja infrastruktuurin hallintaan, sillä ne ovat suunniteltu ensisijaisesti konfiguraationhallintaan. Lisäksi niiden käyttämä proseduraalinen

lähestymistapa vaikeuttaa koodin ylläpidettävyyttä ja uudelleenkäytettävyyttä. Brikmanin (2022) mukaan yksi tyypillinen ratkaisu onkin käyttää provisiointityökalua ja konfiguraationhallintatyökalua yhdessä: esimerkiksi Terraformia infrastruktuurin luontiin ja Ansiblea taas asentamaan sovelluksia luoduille infrastruktuurin osille.

3.2.1 OpenTofu

Vuoden 2023 lokakuussa Terraformin kehittäjä HashiCorp siirtyi käyttämään Business Source License (BSL) -lisenssiä, joka rajoittaa avoimen lähdekoodin käyttöä tietyissä kaupallisissa tilanteissa. Vaikka yksityishenkilöt ja integraatiokumppanit voivat käyttää työkalua maksutta, yritykset, jotka kilpailevat HashiCorpin kanssa, joutuvat rajoitusten alaiseksi. Tämä on herättänyt huolta työkalun avoimuudesta ja pitkäaikaisesta yhteensopivuudesta. (Spacelift 2024.)

OpenTofu syntyi yhteisön vastareaktiona näihin rajoituksiin. Se on täysin avoimen lähdekoodin työkalu Mozilla Public License (MPL) -lisenssillä. Tämä takaa, että sen käyttö on vapaata ja avointa sekä henkilökohtaisiin että kaupallisiin tarkoituksiin. OpenTofu on teknisesti Terraformin haarautuma, joka on yhteensopiva Terraform-moduulien ja -tarjoajien kanssa. (Spacelift 2024.)

Terraform säilyttää kuitenkin vahvan asemansa markkinoilla erityisesti pitkäaikaisen tunnettuutensa ja laajan ekosysteeminsä ansiosta, OpenTofun hakiessa paikkaansa. OpenTofun käyttöä kannattaa harkita, jos haluaa suosia avoimen lähdekoodin ratkaisuja, eikä työkalun tunnettuus ole avainasemassa. Yhteensopivuuden ansiosta migraatio Terraformin ja OpenTofun välillä onnistuu tarvittaessa helposti.

4 TERRAFORM

Terraform on HashiCorpin kehittämä infrastruktuurin hallintatyökalu, joka perustuu IaC-periaatteeseen. Terraform tukee laajasti eri pilvipalvelualustoja, mukaan lukien AWS, Azure ja Google Cloud, mikä mahdollistaa yhdenmukaisen ja skaalautuvan infrastruktuurin hallinnan useissa ympäristöissä ilman riippuvuutta yhdestä palveluntarjoajasta. (HashiCorp n.d.a.)

4.1 Terraformin perusteet

Terraform luo ja hallitsee resursseja pilvipalvelualustoilla ja muissa palveluissa niiden ohjelmointirajapintojen kautta. Providerit mahdollistavat Terraformin toimimisen käytännössä minkä tahansa alustan tai palvelun kanssa, jolla on saavutettava ohjelmointirajapinta. (HashiCorp n.d.a.)

Terraformin ydintyönkulku koostuu kolmesta vaiheesta. Ensimmäisenä käyttäjä määrittelee konfiguraatiotiedostoihin resurssit, jotka voivat olla useiden pilvipalveluntarjoajien ja palveluiden välisiä. Seuraavaksi Terraform luo käyttäjän kehotuksesta toteutussuunnitelman, jossa kuvataan määritelty infrastruktuuri. Toteutussuunnitelman perusteella Terraform luo uuden infrastruktuurin, tai päivittää tai tuhoaa olemassa olevan. Viimeisenä, kun käyttäjä on hyväksynyt suunnitelman Terraform suorittaa ehdotetut toiminnot oikeassa järjestyksessä ottaen huomioon mahdolliset resurssiriippuvuudet. (HashiCorp n.d.a.)

Komento `terraform init` alustaa Terraform-konfiguraatioita sisältävän työhakemiston. Tämä komento tulee suorittaa aina kun uusi konfiguraatio kirjoitetaan tai olemassa oleva kloonataan versionhallinnasta. `terraform plan` -komento luo toteutussuunnitelman infrastruktuuriin tehtävistä muutoksista. Se tarkistaa nykyisen tilan, vertaa sitä uuteen konfiguraatioon ja ehdottaa tarvittavia muutoksia. Komento ei vielä toteuta muutoksia, vaan sitä voidaan käyttää muutosten tarkistamiseen ennen `terraform apply` -komennon suorittamista. `terraform apply` -komento toteuttaa Terraformin ehdottamat muutokset käyttäjän hyväksyttyä ne. Komennolla `terraform destroy` tuhotaan kaikki Terraform-konfiguraation hallitsemat resurssit. (HashiCorp n.d.b.)

4.2 Terraform-konfiguraatiot

Terraform-koodi kirjoitetaan deklaratiivisella HashiCorp Configuration Language (HCL) -kielellä konfiguraatiotiedostoihin, joiden pääte on .tf. Konfiguraatiotiedostojen rakenteella tai lohkojen järjestyksellä ei yleisesti ole merkitystä, sillä Terraform huomioi vain resurssien väliset suorat ja epäsuorat riippuvuudet määrittäessään toimintojen suoritusjärjestystä. (Brikman 2022; HashiCorp n.d.a.)

Resurssit (resource) ovat yksi Terraformin keskeisimmistä elementeistä. Resource-lohkoissa määritellään infrastruktuuriobjekteja, kuten esimerkiksi tietokantoja ja tallennustiloja. Resource-lohkossa esitellään resurssityyppi (resource type), jolle annetaan paikallinen nimi. Annettua nimeä voidaan käyttää, kun resurssiin viitataan saman moduulin sisällä. Esimerkissä (kuva 2) esitellään ”aws_s3_bucket”-resurssityyppi, jolle on annettu nimi ”sanko”. Lohkon sisällä asetetaan argumentit, jotka vaihtelevat resurssityypin mukaan. Tässä tapauksessa bucket ja force_destroy ovat erityisiä argumentteja ”aws_s3_bucket”-resurssityypille. (HashiCorp n.d.c.)

```
5  resource "aws_s3_bucket" "sanko" {
6    bucket        = "bucket"
7    force_destroy = true
8  }
```

KUVA 2. Resource-lohko.

Terraform käyttää providereiksi kutsuttuja liitännäisiä kommunikoidakseen pilvipalveluiden, SaaS-palveluiden ja muiden API:en kanssa. Konfiguraatioissa on aina määriteltävä tarvittavat providerit, jotta Terraform voi asentaa ja käyttää niitä. Kaikki resurssityypit ovat providerin toteuttamia, joten ilman providereita Terraform ei pysty hallitsemaan minkäänlaista infrastruktuuria. Tarvittavat providerit määritellään required_providers-lohkossa, joka koostuu nimestä, lähteen sijainnista ja versiorajoituksesta (kuva 3). (HashiCorp n.d.d.)

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "≥ 5.0"
    }
  }
}
```

KUVA 3. Vaadittujen providerien määrittely.

Muuttujat ja arvot ovat oleellinen osa Terraform-konfiguraatioita. Ne tuovat dynaamisuutta infrastruktuurin hallintaan sekä mahdollistavat DRY-periaatteen (Don't Repeat Yourself) soveltamisen moduuleissa. Muuttujia ja arvoja on kolmen tyyppisiä: syötemuuttujat (input variables), ulostuloarvot (output values) ja paikalliset arvot (local values). (HashiCorp n.d.e)

Syötemuuttujat mahdollistavat Terraform-moduulien muokkaamisen ilman niiden lähdekoodin muuttamista, mikä tekee moduuleista uudelleenkäytettäviä. Syötemuuttuja määritellään variable-lohkossa, ja sille voidaan antaa argumentteja – esimerkin (kuva 4) tapauksessa tyyppi ja kuvaus. Muuttujalle annetaan myös yksilöivä nimi variable-avainsanan jälkeen. (HashiCorp n.d.e)

```
variable "bucket_name" {
  type = string
  description = "AWS S3 bucket name"
}
```

KUVA 4. Variable-lohko.

Muuttujan arvoon päästään käsiksi viittaamalla siihen muodossa var.<nimi>, jossa var.<nimi> vastaa muuttujan määrittelyssä annettua nimeä (kuva 5).

```
resource "aws_s3_bucket" "sanko" {
  bucket        = var.bucket_name
  force_destroy = true
}
```

KUVA 5. Viittaaminen muuttujan arvoon.

Muuttujien arvot voidaan asettaa plan- tai apply-vaiheessa joko yksittäin ”-var”-komentorivivalinnan avulla tai muuttujien määritystiedostossa (.tfvars), jossa voidaan asettaa useampia muuttujia. (HashiCorp n.d.e.)

Yksittäisen muuttujan arvo asetetaan komennolla:

```
terraform apply -var=”bucket_name=bucket”.
```

Muuttujien määritystiedosto vastaavasti asetetaan käytettäväksi komennolla:

```
terraform apply -var-file=”terraform.tfvars”.
```

Ulostuloarvoja käytetään, kun halutaan paljastaa tietoa infrastruktuurista komentoriville tai muille Terraform-konfiguraatioille käytettäväksi. Yksi yleinen käyttötapa on paljastaa alatason moduulin resurssien attribuutteja juurimoduulille ulostulon avulla, jolloin attribuutteja voidaan käyttää hyödyksi esimerkiksi muissa alatason moduuleissa. Ulostuloarvot määritellään output-lohkon (kuva 6) avulla. (HashiCorp n.d.e.)

```
output "bucket" {
  value = aws_s3_bucket.sanko.bucket
}
```

KUVA 6. Output-lohko.

Paikalliset arvot ovat nimettyjä muuttujia, jotka soveltuvat toistuvasti käytettävien vakioarvojen ja laskettujen arvojen tallentamiseen. Toisin kuin syötemuuttujat, paikalliset arvot eivät muutu ajon aikana, vaan säilyvät samoina koko prosessin ajan. Paikalliset arvot määritellään locals-lohkon (kuva 7) avulla. (HashiCorp n.d.e.)

```
locals {
  bucket_name = "bucket"
}
```

KUVA 7. Locals-lohko.

Terraform-moduulit (kuva 8) koostuvat konfiguraatiotiedostoista, jotka sijaitsevat saman kansion sisällä. Jokaisella Terraform-konfiguraatiolla on vähintään yksi moduuli, jota kutsutaan juurimoduuliksi. Minimaalisen moduulin suositellaan sisältävän tiedostot main.tf, variables.tf ja outputs.tf, vaikka ne olisivat tyhjiä. main.tf-tiedosto toimii ensisijaisena tulokohtana, ja yleensä se sisältää käytettävät resurssit. Tiedostoissa variables.tf ja outputs.tf määritellään muuttujat ja ulostulot. (HashiCorp n.d.f.)

KUVA 8. Minimaalinen Terraform-moduuli.

Alatason moduulit (child modules) ovat moduuleja, joita muut moduulit voivat hyödyntää osana konfiguraatiota. Yleinen tapa on kutsua juurimoduulissa alatason moduuleja ja yhdistää ne yhdeksi kokonaisuudeksi module-lohkojen avulla. Source-argumentissa määritellään alatason moduulin sijainti. (HashiCorp n.d.f.)

```
1  module "api" {
2    source      = "./modules/api"
3    project     = "esimerkki"
4  }
5
6  module "function" {
7    source      = "./modules/function"
8    project     = "esimerkki"
9  }
```

KUVA 9. Module-lohkojen kutsuminen.

Terraform tallentaa hallitun infrastruktuurin ja kokoonpanon sen hetkisen tilan (state) aina apply-vaiheen jälkeen. Terraform käyttää tätä tilaa kuvatakseen todellisen maailman resursseja konfiguraatiossa, seuratakseen metatietoja, ja parantaakseen suurten infrastruktuurien suorituskykyä. Tila on oletuksena tallennettu paikalliseen terraform.tfstate-tiedostoon. Ryhmän kanssa työskennellessä paikallisen tiedoston käyttö vaikeuttaa Terraformin käyttöä, sillä

tällöin jokaisen käyttäjän täytyy varmistaa, että heillä on hallussaan viimeisin tiladata, ja että kukaan muu ei aja Terraformia samalla hetkellä. Etätilan (remote state) avulla tilatiedot voidaan kirjoittaa etätietovarastoon, joka taas voidaan jakaa kaikkien ryhmän jäsenten kesken. Terraform tukee tilan tallennusta mm. Amazon S3:een, Azure Blob Storageen, Google Cloud Storageen, ja HCP Terraformiin. (HashiCorp n.d.g.)

5 PROJEKTIN RAKENNE JA YMPÄRISTÖT

5.1 Projektin rakenne

Toteutukset noudattavat yhdenmukaista ja modulaarista rakennetta. Projektin rakenne (kuva 10) koostuu moduuleista, tilan tallennuspaikan luonnin tiedostoista, ympäristökohtaisista tiedostoista, sekä juuritason Terraform-konfiguraatiosta ja määrittelyistä. Tämä rakenne mahdollistaa infrastruktuurin hallinnan eri ympäristöissä ilman tarpeetonta koodin toistamista ja helpottaa muutosten hallintaa.

```
project-root/
├── env/                        # Ympäristökohtaiset konfiguraatiot
│   ├── dev/                      # Kehitysympäristön asetukset
│   │   ├── dev.tfbackend           # Dev-ympäristön backend-konfiguraatio
│   │   └── dev.tfvars              # Dev-ympäristön muuttujien arvot
│   ├── prod/                     # Tuotantoympäristön asetukset
│   │   ├── prod.tfbackend          # Prod-ympäristön backend-konfiguraatio
│   │   └── prod.tfvars             # Prod-ympäristön muuttujien arvot
│   └── staging/                  # Testi-/esituotantoympäristön asetukset
│       ├── staging.tfbackend       # Staging-ympäristön backend-konfiguraatio
│       └── staging.tfvars          # Staging-ympäristön muuttujien arvot
├── init/                       # Terraformin tilatiedoston (state) säilytyspaikan määritykset
│   ├── main.tf                 # Terraform-konfiguraatio tilatiedoston tallennukseen
│   └── variables.tf             # Tallennuspaikkaan liittyvien muuttujien määrittelyt
├── modules/                    # Uudelleenkäytettävät Terraform-moduulit eri komponenteille
│   ├── api/                      # API-infrastruktuurin moduuli
│   │   ├── main.tf                 # API:n Terraform-konfiguraatio
│   │   ├── variables.tf            # API-moduulin muuttujat
│   │   └── outputs.tf              # API-moduulin ulostulot
│   ├── database/                 # Tietokannan moduuli
│   │   ├── main.tf                 # Tietokannan Terraform-konfiguraatio
│   │   ├── variables.tf            # Tietokantamoduulin muuttujat
│   │   └── outputs.tf              # Tietokantamoduulin ulostulot
│   └── function/                 # Funktiopalvelun moduuli
│       ├── function.py             # Funktiokoodi (esim. Python-skripti)
│       ├── main.tf                 # Funktiomoduulin Terraform-konfiguraatio
│       ├── variables.tf            # Funktiomoduulin muuttujat
│       └── outputs.tf              # Funktiomoduulin ulostulot
├── main.tf                     # Pääasiallinen Terraform-konfiguraatio koko infrastruktuurille
├── providers.tf                 # Tarjoajien määrittelyt (esim. AWS, Azure, GCP)
├── terraform.tfvars            # Yleisten muuttujien arvot (esim. projektin nimi)
└── variables.tf                # Yleisten muuttujien määrittelyt (esim. ympäristö, resurssien nimet)
```

KUVA 10. Projektien kansiorakenne.

Juuritason main.tf-tiedostossa (kuva 11) määritellään käytettävät moduulit, sekä muut tarvittavat resurssit.

```
16  module "api" {
17    source      = "./modules/api"
18    project     = var.project
19    environment = var.environment
20  }
21
22  module "database" {
23    source      = "./modules/database"
24    project     = var.project
25    environment = var.environment
26  }
27
28  module "function" {
29    source      = "./modules/function"
30    project     = var.project
31    environment = var.environment
32  }
```

KUVA 11. Esimerkki juuritason main.tf-tiedostosta.

Juuritason variables.tf-tiedostossa (kuva 12) määritellään yleiset muuttujat.

```
1  variable "project" {
2    description = "The project name"
3    type        = string
4  }
5
6  variable "region" {
7    description = "AWS region"
8    type        = string
9  }
10
11 variable "environment" {
12   description = "The environment to deploy (dev, prod, etc.)"
13   type        = string
14 }
```

KUVA 12. Esimerkki variables.tf-tiedostosta.

terraform.tfvars-tiedostossa (kuva 13) määritellään arvot yleisille muuttujille.

```
1  project = "perathesis"
2  region  = "eu-north-1"
```

KUVA 13. Esimerkki terraform.tfvars-tiedostosta.

providers.tf-tiedostossa (kuva 14) määritellään käytettävien providerien ominaisuudet.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "≥ 5.0"
    }
  }
  backend "s3" {}
}

provider "aws" {
  region = var.region
}
```

KUVA 14. Esimerkki providers.tf-tiedostosta.

5.2 Ympäristöjen pystytys ja hallinta

Ennen kuin ympäristöjä voidaan pystyttää ja käyttää, täytyy tilatiedostolle luoda etätallennuspaikka. Tallennuspaikka luodaan suorittamalla komentorivillä projektikansion juuressa seuraavat komennot järjestyksessä:

```
cd init
terraform init
terraform apply -var-file=../terraform.tfvars
```

Ensimmäisellä komennolla siirrytään init-kansioon, joka sisältää tarvittavat tiedostot tilatiedoston tallennuspaikan luontia varten. Toinen komento alustaa kansion Terraformin käytettäväksi. Kolmas komento luo ja näyttää suoritussuunnitelman, kehottaa käyttäjää hyväksymään suunnitelman ja lopulta suorittaa halutut toimet. ”-var-file”-osa komennosta asettaa konfiguraatiossa käytettävät muuttujat terraform.tfvars-tiedostosta.

Kun tallennuspaikka on luotu onnistuneesti, palataan projektikansion juureen. Haluttu ympäristö (dev, prod tai staging) alustetaan käytettäessä komennolla:

```
terraform init -reconfigure
-backend-config=env/<ympäristö>/<ympäristö>.tfbackend.
```

Komennon “reconfigure”-valinnalla varmistetaan, että Terraform konfiguroi backendin uudelleen riippumatta siitä, mitä aiempia asetuksia olemassa. ”-backend-config”-valinnassa taas viitataan tiedostoon, joka sisältää käytettävän backendin asetukset.

Onnistuneen ympäristön alustuksen jälkeen konfiguraatio voidaan suorittaa komennolla:

```
terraform apply
-var-file=env/<ympäristö>/<ympäristö>.tfvars
```

Ympäristöjen tuhoaminen suoritetaan komennolla:

```
terraform destroy
-var-file=env/<ympäristö>/<ympäristö>.tfvars
```

6 SERVERLESS-TOTEUTUKSET ERI PILVIPALVELUISSA

Tässä kappaleessa käsitellään Terraform-konfiguraatioiden toteutuksia AWS-, Azure- ja Google Cloud -ympäristöissä. Toteutukset sisältävät serverless-rakenteen, johon kuuluu tietokanta, siihen liitetty funktio sekä ohjelmointirajapinta. Toteutuksessa hyödynnetään AWS Lambdaa, Azure Functionsia ja Google Cloud Run functionsia funktioiden ajamiseen, sekä tietokantapalveluita, kuten Amazon DynamoDB:tä, Azure Cosmos DB:tä ja Google Firestorea. Ohjelmointirajapinnat käyttävät API-palveluita, kuten AWS API Gatewaytä, Azure API Managementia ja Google Cloud API Gatewaytä.

6.1 AWS

Amazon Web Services (AWS) on pilvipalveluntarjoaja, joka tarjoaa laajan joukon maailmanlaajuisia pilvipohjaisia tuotteita, kuten laskenta-, tallennus, tietokanta, verkko-, ja analytiikkapalveluita. (AWS 2024.)

6.1.1 DynamoDB

Amazon DynamoDB on täysin hallinnoitu ja palvelimeton NoSQL-tietokantapalvelu, joka on suunniteltu skaalautumaan joustavasti ilman käyttäjän tarvetta hallita monimutkaista infrastruktuuria. DynamoDB:ssä maksetaan vain käytetystä tallennustilasta ja suoritetuista luku- ja kirjoitustoiminnoista. DynamoDB:n tietorakenne perustuu tauluihin, jotka sisältävät kohteita (items). Jokainen kohde yksilöidään ensisijaisen avaimen avulla ja voi sisältää joukon avain-arvopareja, joita kutsutaan attribuuteiksi. (AWS 2022.)

6.1.2 Lambda

AWS Lambda on FaaS-palvelu, joka mahdollistaa koodin suorittamisen ilman palvelimien hallintaa. Lambda huolehtii automaattisesti resursseista, kuten palvelinten ja käyttöjärjestelmien ylläpidosta, kapasiteetin provisioinnista, automaattisesta skaalauksesta ja lokien hallinnasta. (AWS n.d.)

6.1.3 API Gateway

API Gateway on palvelu, jonka avulla voidaan luoda API-kerros käyttöliittymän ja palvelinpuolen välille. API Gatewayn elinkaaren hallinta mahdollistaa useiden API-versioiden ajamisen samanaikaisesti, ja se tukee useita julkaisuvaiheita. (Cui, Nair & Sbarski 2022.)

6.1.4 Terraform-konfiguraation toteutus

Terraform-konfiguraation toteutus AWS:lle aloitetaan DynamoDB-moduulin rakentamisella. Tarvittavan tietokantataulun luominen onnistuu ”aws_dynamodb_table”-resurssilla (kuva 15). Pakollisia argumentteja resurssille ovat name, hash_key, sekä attribute. Taulun nimi määritellään name-argumenttiin yleisten muuttujien var.environment ja var.project, sekä loppuosan ”table” avulla. Tämä tapa mahdollistaa taulujen erottamisen ympäristöjen ja projektien välillä. Vastaavaa nimeämistapaa käytetään myös muissa jatkossa esiintyvissä resursseissa ja toteutuksissa. Argumentissa billing_mode määritellään arvoksi ”PAY_PER_REQUEST”, jotta veloitus tapahtuu käytön mukaan. Argumentissa hash_key taas määritellään taulun pääavain, joka toimii yksilöivänä tunnisteena taulun tietueille, joka on tässä tapauksessa ”id”.
Sisäkkäisessä attribute-lohkossa määritellään taulun hajautusavaimeksi ”id” ja taulun tietotyypiksi merkkijono (string, S). ttl-lohkossa määritellään, että taulun tietueilla on elinikä (Time to Live, TTL). Tunnisteet, jotka kuvaavat ympäristöä ja projektia pystytään määrittelemään tags-argumentilla.

```
resource "aws_dynamodb_table" "table" {
  name         = "${var.environment}-${var.project}-table"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "id"

  attribute {
    name = "id"
    type = "S" # String type
  }

  ttl {
    attribute_name = "ttl"
    enabled        = true
  }

  tags = {
    "Environment" = "${var.environment}-${var.project}"
  }
}
```

KUVA 15. DynamoDB-konfiguraatio.

Konfiguraation ulostuloksi (kuva 16) täytyy määritellä DynamoDB-taulun Amazon Resource Name (ARN), jotta taulu voidaan integroida myöhemmin Lambda-moduuliin.

```
# Output the table ARN for IAM role policy in the Lambda module
output "dynamodb_table_arn" {
  value = aws_dynamodb_table.table.arn
}
```

KUVA 16. DynamoDB-moduulin ulostulo.

Seuraavaksi luodaan konfiguraatio Lambda-moduulille. Ensimmäiseksi määritellään IAM-rooli (Identity and Access Management) ”aws_iam_role”-resurssilla (kuva 17), jota Lambda-funktio käyttää. Argumentissa assume_role_policy määritellään, että kyseistä roolia voivat käyttää vain AWS Lambda -palvelut.

```
# Define IAM role for Lambda to allow access to DynamoDB
resource "aws_iam_role" "lambda_role" {
  name = "${var.environment}-${var.project}-lambda-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          Service = "lambda.amazonaws.com"
        }
      }
    ]
  })
}
```

KUVA 17. IAM-roolin määrittely.

Tämän jälkeen luodaan IAM-politiikka Lambda-funktiolle ”aws_iam_role_policy”-resurssilla, jossa määritellään oikeudet, joita Lambda-funktio tarvitsee DynamoDB:n käyttöön. Politiikassa sallitaan Lambda-funktiolle seuraavat DynamoDB-toiminnot: PutItem, GetItem, Query, UpdateItem, sekä Scan. Resource-muuttujassa määritellään, että politiikka koskee vain taulua, jonka ARN saatiin ulostulona DynamoDB-konfiguraatiossa. Politiikka liitetään vielä aikaisemmin luotuun IAM-rooliin (kuva 18).

```
# Define IAM policy for Lambda to access DynamoDB
resource "aws_iam_policy" "lambda_dynamodb_policy" {
  name        = "${var.environment}-${var.project}-lambda-dynamodb-policy"
  description = "Policy to allow Lambda function to access DynamoDB"

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = [
          "dynamodb:PutItem",
          "dynamodb:GetItem",
          "dynamodb:Query",
          "dynamodb:UpdateItem",
          "dynamodb:Scan"
        ]
        Effect   = "Allow"
        Resource = "${var.dynamodb_table_arn}"
      }
    ]
  })
}

# Attach policy to Lambda IAM role
resource "aws_iam_role_policy_attachment" "lambda_dynamodb_policy_attachment" {
  role       = aws_iam_role.lambda_role.name
  policy_arn = aws_iam_policy.lambda_dynamodb_policy.arn
}
```

KUVA 18. IAM-politiikan määrittely ja sen liittäminen IAM-rooliin.

Funktiokoodi pakataan ZIP-tiedostoksi ”archive-file”-resurssilla. Viimeisenä luodaan vielä itse Lambda-funktio ”aws_lambda_function”-resurssilla. Funktiolle määritellään suoritettava käsittelijä (handler), sekä ajoympäristö (runtime), joka on tässä tapauksessa Python 3.11. IAM-rooli, joka antaa Lambda-funktiolle oikeudet käyttää muita AWS-palveluita, määritellään viittaamalla aiemmin luotuun IAM-rooliin. ZIP-tiedosto, joka sisältää Lambda-funktion koodin, osoitetaan filename-parametrilla. Jotta Terraform havaitsisi mahdolliset muutokset Lambda-funktion koodissa ja päivittäisi sen automaattisesti, hyödynnetään source_code_hash-parametria, joka generoi koodista tarkistussumman. Ympäristömuuttujat asetetaan environment-lohkossa, jossa määritellään käytettävä ympäristö ja projekti (kuva 19).

```
data "archive_file" "function_code" {
  type        = "zip"
  source_dir  = "${path.module}/lambdas/"
  output_path = "${path.module}/lambdas/lambda_function.zip"
  excludes    = ["**/*.zip"]
}

resource "aws_lambda_function" "lambda_function" {
  function_name = "${var.environment}-${var.project}-lambda-function"
  handler       = "lambda_function.lambda_handler"
  runtime       = "python3.11"
  role          = aws_iam_role.lambda_role.arn
  filename      = data.archive_file.function_code.output_path

  # Ensure Terraform redeploys on code changes
  source_code_hash = filebase64sha256(data.archive_file.function_code.output_path)

  environment {
    variables = {
      ENVIRONMENT = var.environment
      PROJECT = var.project
    }
  }
}
```

KUVA 19. Funktiokoodin pakkaus ja Lambda-funktion määrittely.

Lambda-funktion koodissa määritellään aluksi ympäristö- sekä projektimuuttuja ja muodostetaan yhteys DynamoDB-resurssiin boto3-kirjaston avulla. Taulu nimetään ympäristön ja projektin mukaisesti, mikä mahdollistaa dynaamisen konfiguroinnin eri ympäristöille. Funktio lambda_handler sisältää yksinkertaisen toimintaperiaatteen, joka lisää tauluun item-olion ja palauttaa statuskoodin, viestin sekä lisätyn olion sisältävän JSON-vastauksen funktiokutsun onnistuessa (kuva 20).

```
import json
import os
import boto3

environment = os.getenv("ENVIRONMENT")
project = os.getenv("PROJECT")
dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table(f"{environment}-{project}-table")

def lambda_handler(event, context):
    item = {
        "id": "123",
        "message": "Hello from Lambda to DynamoDB"
    }
    try:
        table.put_item(Item=item)
        return {
            "statusCode": 200,
            "body": json.dumps({"message": "Item inserted into DynamoDB", "data": item})
        }
    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"message": "Failed to insert item", "error": str(e)})
        }
```

KUVA 20. Lambda-funktion koodi.

Vastaavasti kuin DynamoDB-konfiguraatiossa, jotta Lambda-funktio voidaan integroida API Gateway-moduuliin, pitää ulostuloksi määritellä Lambda-funktion ARN (kuva 21).

```
output "lambda_arn" {
  value = aws_lambda_function.lambda_function.arn
}
```

KUVA 21. Lambda-moduulin ulostulo.

API Gateway -konfiguraatiossa ensimmäiseksi luodaan REST API käyttäen ”aws_api_gateway_rest_api”-resurssia, jossa määritellään API:n nimi ja kuvaus. ”aws_api_gateway_resource”-resurssilla määritellään API:n polku /lambda, ja polulle määritellään GET-metodi ”aws_api_gateway_method”-resurssilla (kuva 22).

```
# API Gateway REST API
resource "aws_api_gateway_rest_api" "apigw" {
  name        = "${var.environment}-${var.project}-api"
  description = "API for Lambda Integration"
}

# API Gateway Resource (path part '/lambda')
resource "aws_api_gateway_resource" "root" {
  rest_api_id = aws_api_gateway_rest_api.apigw.id
  parent_id   = aws_api_gateway_rest_api.apigw.root_resource_id
  path_part   = "lambda"
}

# Define GET method for /lambda
resource "aws_api_gateway_method" "get_lambda" {
  rest_api_id   = aws_api_gateway_rest_api.apigw.id
  resource_id   = aws_api_gateway_resource.root.id
  http_method   = "GET"
  authorization = "NONE"
}
```

KUVA 22. REST API -määrittely.

”aws_api_gateway_integration”-resurssi yhdistää API Gatewayn ja Lambda-funktion käyttäen AWS_PROXY-integraatiotyyppiä, jolloin HTTP-pyynnöt ohjataan suoraan Lambdaan. Toinen resurssi ”aws_lambda_permission” myöntää API Gatewaylle oikeuden kutsua Lambda-funktiota, varmistaen, että vain määritelty API Gateway voi tehdä kutsuja Lambdaan (kuva 23).

```
# Integration with Lambda
resource "aws_api_gateway_integration" "lambda_integration" {
  rest_api_id             = aws_api_gateway_rest_api.apigw.id
  resource_id             = aws_api_gateway_resource.root.id
  http_method             = aws_api_gateway_method.get_lambda.http_method
  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = "arn:aws:apigateway:${var.region}:lambda:path/   2015-03-31/
functions/${var.lambda_arn}/invocations"
}

# Grant API Gateway permission to invoke Lambda
resource "aws_lambda_permission" "api_gateway_invoke" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambda:InvokeFunction"
  function_name = var.lambda_arn
  principal     = "apigateway.amazonaws.com"
  source_arn    = "${aws_api_gateway_rest_api.apigw.execution_arn}/*/*"
}
```

KUVA 23. Lambda-integraatio ja -oikeudet.

”aws_api_gateway_deployment”-resurssi ottaa API:n käyttöön ja ulostulo ”api_url” asettaa API Gatewaylle julkisen URL-osoitteen, josta API on käytettävissä (kuva 24).

```
# Deployment resource to create a new deployment
resource "aws_api_gateway_deployment" "apigw_deployment" {
  depends_on  = [aws_api_gateway_integration.lambda_integration]
  rest_api_id = aws_api_gateway_rest_api.apigw.id
}

output "api_url" {
  value       = "https://${aws_api_gateway_rest_api.apigw.id}.execute-api.${var.region}.
amazonaws.com/${var.project}-${var.environment}"
  description = "Base URL of the API Gateway endpoint"
}
```

KUVA 24. API Gatewayn käyttöönotto ja URL-ulostulo

6.2 Azure

Microsoft Azure on pilvipalveluntarjoaja, joka tarjoaa laajan valikoiman pilvipohjaisia ratkaisuja ja palveluita, kuten laskenta-, tallennus-, tietokanta-, analytiikka, ja tekoälypalveluita. Se tukee useita ohjelmointikieliä, työkaluja ja kehyksiä, mikä tekee siitä joustavan vaihtoehdon monenlaisille tarpeille.

6.2.1 Azure Cosmos DB

Azure Cosmos DB on tietokantapalvelu, joka on suunniteltu korkean suorituskyvyn, saatavuuden ja skaalautuvuuden tarpeisiin. Se tukee useita

NoSQL-malleja ja tarjoaa myös relaatiotietokannan ominaisuuksia. (Microsoft 2024a.)

6.2.2 Azure Functions

Azure Functions on FaaS-palvelu, joka mahdollistaa koodin suorittamisen ilman palvelimien hallintaa Lambdan tavoin. Käyttäjät maksavat vain suoritetusta koodista, mikä tekee palvelusta kustannustehokkaan. (Microsoft 2024b.)

6.2.3 API Management

Azure API Management on monipilvialusta sovellusliittymille kaikissa ympäristöissä. PaaS-palveluna API Management tukee koko API:n elinkaarta. (Microsoft 2024c.)

6.2.4 Terraform-konfiguraation toteutus

Terraform-konfiguraation toteutus aloitetaan määrittelemällä Cosmos DB -tietokantamoduuli. Cosmos DB -tilin luominen tapahtuu resurssilla "azurerm_cosmosdb_account" (kuva 25). Pakollisia argumentit tälle resurssille ovat name, location ja resource_group_name, jotka määritetään Terraformin muuttujien avulla. Tilin nimeämisessä käytetään ympäristön ja projektin muuttujia sekä loppuosaa "admin". Toteutuksessa halutaan käyttää NoSQL-tietokantatyyppiä, joten kind-argumentin arvoksi asetetaan ”GlobalDocumentDB”. Konsistenssipolitiikka on asetettu arvoksi "BoundedStaleness", jonka avulla määritellään sallittu viive ja staleness-rajoitukset.

```
resource "azurerm_cosmosdb_account" "account" {
  name                = "${var.environment}-${var.project}-admin"
  location            = var.location
  resource_group_name = var.resource_group_name
  offer_type          = "Standard"
  kind                = "GlobalDocumentDB" # Define NoSQL database type

  geo_location {
    location          = var.location
    failover_priority = 0
  }

  consistency_policy {
    consistency_level       = "BoundedStaleness"
    max_interval_in_seconds = 300
    max_staleness_prefix    = 100000
  }
}

```

KUVA 25. Azure Cosmos DB -tilin luominen.

Kun Cosmos DB -tili on luotu, voidaan luoda tietokanta. NoSQL-tietokanta luodaan Cosmos DB -tilin sisälle resurssilla ”azurerm_cosmosdb_sql_database”. Tietokannan nimi ja kapasiteetti määritellään argumenteissa. Kapasiteetin arvoksi on asetettu pienin mahdollinen 400 RU/s (request unit/second). Viimeiseksi luodaan vielä tietokantaan liittyvä kontti resurssilla ”azurerm_cosmosdb_sql_container” (kuva 26). Kontille asetetaan nimi, konttiin liittyvän tietokannan ja Cosmos DB -tilin tiedot, sekä partition_key_paths, joka on tässä tapauksessa "/id". Myös kontin kapasiteetin arvoksi on määritelty 400 RU/s.

```
resource "azurerm_cosmosdb_sql_database" "main" {
  name                = "${var.environment}-nosqldb"
  resource_group_name = var.resource_group_name
  account_name        = azurerm_cosmosdb_account.account.name
  throughput          = 400
}

resource "azurerm_cosmosdb_sql_container" "container" {
  name                = "${var.environment}-container"
  resource_group_name = var.resource_group_name
  account_name        = azurerm_cosmosdb_account.account.name
  database_name       = azurerm_cosmosdb_sql_database.main.name
  partition_key_paths = ["/id"]
  throughput          = 400
}
```

KUVA 26. Cosmos DB -tietokannan ja kontin luominen.

Moduulille on määriteltävä neljä ulostuloa, jotta funktio voidaan integroida tietokantaan. Ulostulo ”cosmosdb_endpoint” antaa Cosmos DB -instanssin URL-osoitteen, jonka kautta tietokantaan voidaan yhdistää. Ulostulo ”cosmosdb_key” tarjoaa instanssin pääavaimen (primary key), jota myöhemmin luotava funktio

tarvitsee tietokannan käyttöä varten. Tietokannan nimi palautetaan ulostulossa ”cosmosdb_database_name” ja kontin nimi (container) taas ulostulossa ”cosmosdb_container_name” (kuva 27).

```
output "cosmosdb_endpoint" {
  value = azurerm_cosmosdb_account.account.endpoint
}

output "cosmosdb_key" {
  value = azurerm_cosmosdb_account.account.primary_key
}

output "cosmosdb_database_name" {
  value = azurerm_cosmosdb_sql_database.main.name
}

output "cosmosdb_container_name" {
  value = azurerm_cosmosdb_sql_container.container.name
}
```

KUVA 27. Cosmos DB -moduulin ulostulot.

Azure Functions -moduulin rakentaminen aloitetaan määrittelemällä tarvittava tallennustili resurssilla ”azurerm_storage_account”. Tallennustyypiksi asetetaan "Standard" sekä replikaatiotyypiksi "LRS" (Local Redundancy Storage). Tallennustilille luodaan yksityinen kontti nimeltä ”function-code”, johon Function App -kooditiedosto tallennetaan (kuva 28).

```
# Storage account (required by the Function App)
resource "azurerm_storage_account" "storage" {
  name                     = "${var.environment}${var.project}storage"
  resource_group_name      = var.resource_group_name
  location                 = var.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "function_container" {
  name                  = "function-code"
  storage_account_name  = azurerm_storage_account.storage.name
  container_access_type = "private"
}
```

KUVA 28. Tallennustilin ja -kontin luominen.

Seuraavaksi käytetään resurssia ”archive_file”, jolla pakataan Function App -koodi ZIP-tiedostoksi. Pakkausprosessin aikana voidaan sulkea pois esimerkiksi kehitysympäristöön liittyvät tiedostot, kuten __pycache__ ja local.settings.json. Sovelluksen käyttöön luodaan Azure Functions -kulutusmallin mukainen palvelusuunnitelma resurssilla ”azurerm_service_plan” (kuva 29). Palvelusuunnitelma käyttää Linux-käyttöjärjestelmää ja SKU-tasoksi

määritellään "Y1", joka viittaa kulutusmalliin, jossa laskutus perustuu resurssien todelliseen käyttöön.

```
# Package the function into a zip file
data "archive_file" "function" {
  type        = "zip"
  source_dir  = "${path.module}/functionapp/"
  output_path = "${path.module}/functionapp/function_app.zip"
  excludes = [
    "**/__pycache__",
    "**/.venv",
    "**/function_app.zip",
    "**/local.settings.json"
  ]
}

# Service plan
resource "azurerm_service_plan" "service_plan" {
  name                = "${var.environment}-${var.project}-service-plan"
  resource_group_name = var.resource_group_name
  location            = var.location
  os_type             = "Linux"
  sku_name            = "Y1" # Consumption plan for Azure Functions
}
```

KUVA 29. Funktiokoodin pakkaus ja palvelusuunnitelman luominen.

Lopuksi määritellään varsinainen Function App -sovellus resurssilla ”azurerm_linux_function_app”. Sovelluksen nimi muodostetaan ympäristön ja projektin tunnisteiden perusteella. Resurssi käyttää tallennustilin pääsyavainta ja aikaisemmin pakattua ZIP-tiedostoa sovelluksen koodin julkaisemiseen. Konfiguraatiossa määritellään myös sovelluskohtaiset muuttujat, kuten Cosmos DB -tiedot, ja käytettävä Python-versio 3.11 (kuva 30).

```
# Function app
resource "azurerm_linux_function_app" "function_app" {
  name                       = "${var.environment}-${var.project}-function-app"
  resource_group_name        = var.resource_group_name
  location                   = var.location
  service_plan_id            = azurerm_service_plan.service_plan.id
  storage_account_name       = azurerm_storage_account.storage.name
  storage_account_access_key = azurerm_storage_account.storage.primary_access_key

  app_settings = {
    "PACKAGE_HASH"                  = data.archive_file.function.output_base64sha256
    "FUNCTIONS_WORKER_RUNTIME"      = "python"
    "ENABLE_ORYX_BUILD"             = "true"
    "SCM_DO_BUILD_DURING_DEPLOYMENT" = "true"
    "COSMOSDB_URL"                  = var.cosmosdb_endpoint
    "COSMOSDB_KEY"                  = var.cosmosdb_key
    "COSMOSDB_DATABASE"             = var.cosmosdb_database_name
    "COSMOSDB_CONTAINER"            = var.cosmosdb_container_name
  }

  site_config {
    application_stack {
      python_version = "3.11"
    }
  }

  zip_deploy_file = data.archive_file.function.output_path # Function zip path
}
```

KUVA 30. Function App -sovelluksen määrittely.

Funktiokoodissa määritellään ensin yhteys Cosmos DB -tietokantaan hyödyntäen tietokannan URL-osoitetta, pääavainta, tietokannan nimeä ja kontin nimeä. Yhteys muodostetaan käyttämällä Azure Cosmos -kirjastoa. Funktio http_trigger vastaa HTTP-pyyntöihin ja suorittaa datan tallennuksen Cosmos DB:hen. Funktio luo olion, jossa on esimerkkiarvot kentille id ja message, ja lisää tietueen tietokantaan metodilla upsert_item. Kutsun onnistuessa funktio palauttaa statuskoodin, viestin, sekä lisätyn olion (kuva 30).

```python
import os
import azure.functions as func
import json
from azure.cosmos import CosmosClient

# Cosmos DB connection setup
cosmos_url = os.environ["COSMOSDB_URL"]
cosmos_key = os.environ["COSMOSDB_KEY"]
client = CosmosClient(cosmos_url, cosmos_key)

database_name = os.environ["COSMOSDB_DATABASE"]
container_name = os.environ["COSMOSDB_CONTAINER"]
database = client.get_database_client(database_name)
container = database.get_container_client(container_name)

app = func.FunctionApp(http_auth_level=func.AuthLevel.ANONYMOUS)

@app.route(route="http_trigger")
def http_trigger(req: func.HttpRequest) → func.HttpResponse:
    item = {
        "id": "123",
        "message": "Hello from Azure Function to CosmosDB"
    }
    try:
        container.upsert_item(body=item)
        response_body = json.dumps({"message": "Item inserted into CosmosDB", "data":  item})
        return func.HttpResponse(response_body, status_code=200)
    except Exception as e:
        error_body = json.dumps({"message": "Failed to insert item", "error": str(e)})
        return func.HttpResponse(error_body, status_code=500)

```

KUVA 31. Function App -koodi.

Moduulin ulostuloksi määritellään Function App:n default_hostname, jotta Function App voidaan integroida API Management -moduuliin. (kuva 32).

```
output "function_app_hostname" {
  value = azurerm_linux_function_app.function_app.default_hostname
}
```

KUVA 32. Functions-moduulin ulostulo.

Azure API Management -konfiguraatiossa luodaan ensin API Management -palvelu käyttäen ”azurerm_api_management”-resurssia. Tämän resurssin avulla määritellään palvelun nimi, sijainti, resurssiryhmä sekä julkaisijan tiedot, kuten

nimi ja sähköpostiosoite. Lisäksi määritellään käytettävä maksumalli, tässä tapauksessa "Consumption"-malli. Seuraavaksi määritellään uusi API ”azurerm_api_management_api”-resurssilla. Resurssissa asetetaan API:n nimi, näyttönimi, revisio, URL-polku sekä sallitut protokollat (kuva 33).

```
resource "azurerm_api_management" "api_management" {
  name                = "${var.environment}-${var.project}-apim"
  location            = var.location
  resource_group_name = var.resource_group_name
  publisher_name      = "ExamplePublisher"
  publisher_email     = "publisher@example.com"
  sku_name            = "Consumption_0"
}

resource "azurerm_api_management_api" "api" {
  name                = "${var.environment}-${var.project}-api"
  resource_group_name = var.resource_group_name
  api_management_name = azurerm_api_management.api_management.name
  revision            = "1"
  display_name        = "${var.project} API"
  path                = "${var.environment}-api"
  protocols           = ["https"]
}
```

KUVA 33. API Management -palvelun ja API:n luominen

API-operaatio lisätään ”azurerm_api_management_api_operation”-resurssilla. Tässä määritellään operaatioon liittyvät yksityiskohdat, kuten tunniste, metodityyppi, ja URL-malli. Lisäksi asetetaan palautuskoodi 200, joka kuvaa onnistunutta vastausta. Lopuksi määritellään API:n politiikka ”azurerm_api_management_api_operation_policy”-resurssilla (kuva 34). Tämä politiikka ohjaa saapuvat pyynnöt Azure Functionsiin, jonka URL annetaan XML-pohjaisessa politiikkasisällössä. Politiikka sisältää asetukset pyynnön välittämiseksi eteenpäin ja takaa yhteensopivuuden API Managementin ja taustapalvelun välillä.

```
resource "azurerm_api_management_api_operation" "operation" {
  operation_id        = "${var.project}-operation"
  api_name            = azurerm_api_management_api.api.name
  api_management_name = azurerm_api_management.api_management.name
  resource_group_name = var.resource_group_name
  display_name        = "${var.project} Operation"
  method              = "POST"
  url_template        = "/"
  response {
    status_code = 200
  }
}

resource "azurerm_api_management_api_operation_policy" "operation_policy" {
  operation_id        = azurerm_api_management_api_operation.operation.operation_id
  api_name            = azurerm_api_management_api.api.name
  api_management_name = azurerm_api_management.api_management.name
  resource_group_name = var.resource_group_name
  xml_content         = <<XML
    <policies>
      <inbound>
        <set-backend-service base-url="https://${var.function_app_hostname}/api/http_trigger" />
      </inbound>
      <backend>
        <forward-request />
      </backend>
      <outbound />
    </policies>
  XML
}
```

KUVA 34. API-operaation ja API-operaatiopolitiikan määrittely

6.3 Google Cloud

Google Cloud, entiseltä nimeltään Google Cloud Platform, on pilvipalveluntarjoaja, joka tarjoaa kattauksen erilaisia tuotteita kuten laskenta-, tallennus-, tietokanta-, ja tekoälypalveluita. (Google Cloud n.d.a)

6.3.1 Cloud Firestore

Cloud Firestore on joustava, skaalautuva NoSQL-tietokanta mobiili-, verkko- ja palvelinkehitykseen. (Google Firebase n.d.)

6.3.2 Cloud Run functions

Cloud Run functions on FaaS-palvelu pilvipalvelujen rakentamiseen ja yhdistämiseen. Cloud Run -toimintojen avulla voi rakentaa yksikäyttöisiä toimintoja, jotka on liitetty pilvi-infrastruktuurin ja -palveluiden lähettämiin tapahtumiin. (Google Cloud n.d.b.)

6.3.3 API Gateway

API Gateway mahdollistaa suojatun pääsyn palveluihin tarkasti määrittelyn REST-ohjelmointirajapinnan kautta, joka on yhdenmukainen kaikissa palveluissa. (Google Cloud n.d.c.)

6.3.4 Terraform-konfiguraation toteutus

Firestore-tietokannan konfiguraatiossa (kuva 35) ensimmäiseksi otetaan käyttöön Firestore API ”google_project_service”-resurssilla. Sen jälkeen luodaan itse Firestore-tietokanta ”google_firestore_database-”resurssilla, jossa määritellään tietokannan nimi, sijainti, projekti ja tyyppi. Argumentti depends_on varmistaa, että Firestore API on otettu käyttöön ennen tietokannan luomista.

```
# Enable required APIs
resource "google_project_service" "firestore" {
  service            = "firestore.googleapis.com"
  disable_on_destroy = false
}

# Deploy the Firestore database
resource "google_firestore_database" "database" {
  name             = "${var.environment}-${var.project}-db"
  location_id      = var.region #
  project          = var.project_id
  type             = "FIRESTORE_NATIVE"
  deletion_policy = "DELETE"

  depends_on = [google_project_service.firestore] # Ensure the API is enabled before creation
}
```

KUVA 35. Firestore-tietokannan konfiguraatio

Cloud Run functions -konfiguraatiossa määritellään ensin tarvittavat API-palvelut resurssilla ”google_project_service”, tässä tapauksessa Cloud Functions, Cloud Build ja Cloud Run. Tämän jälkeen luodaan resurssilla ”google_storage_bucket” tallennuspaikka (bucket), joka toimii funktiokoodin säilytyspaikkana. Tallennuspaikan nimi muodostetaan satunnaisen id:n avulla, mikä varmistaa sen ainutlaatuisuuden (kuva 36).

```
# Enable required APIs
resource "google_project_service" "cloudfunctions" {
  service            = "cloudfunctions.googleapis.com"
  disable_on_destroy = false
}

resource "google_project_service" "cloudbuild" {
  service            = "cloudbuild.googleapis.com"
  disable_on_destroy = false
}

resource "google_project_service" "run" {
  service            = "run.googleapis.com"
  disable_on_destroy = false
}

resource "random_id" "default" {
  byte_length = 8
}

# Unique storage bucket for the function's source code
resource "google_storage_bucket" "function_bucket" {
  name                        = "${random_id.default.hex}-function-bucket"
  location                    = var.region
  uniform_bucket_level_access = true
}
```

KUVA 36. Tarvittavien API-palveluiden ja tallennustilan määrittely.

Funktiokoodi pakataan ZIP-tiedostoksi ”archive_file”-resurssilla. Paikallisessa arvossa locals generoidaan ZIP-tiedoston tarkistussumma, jonka avulla Terraform osaa päivittää muutokset palveluun koodin muuttuessa. Resurssi ”google_storage_bucket_object” tallentaa ZIP-tiedoston tallennuspaikkaan käytettäväksi (kuva 37).

```
# Zip the function code and exclude unnecessary files
data "archive_file" "function_code" {
  type        = "zip"
  source_dir  = "${path.module}/function_code/"
  output_path = "${path.module}/function_code/function_code.zip"
  excludes    = ["**/*.zip"]
}

# Generate hash to track function code changes
locals {
  function_code_hash = filebase64sha256(data.archive_file.function_code.output_path)
}

# Upload the function code to the storage bucket
resource "google_storage_bucket_object" "function_archive" {
  name   = "function_code.zip"
  bucket = google_storage_bucket.function_bucket.name
  source = data.archive_file.function_code.output_path
}
```

KUVA 37. Funktiokoodin pakkaus ja tallennus.

Funktiokoodi otetaan käyttöön ”google_cloudfunction2_function”-resurssilla. Funktiosovellukselle määritellään ajoympäristöksi Python 3.11 ja tulokohta eli käytettävä funktio. Argumentissa source määritellään tallennustilan ja

funktiokoodin sijainti, jotta tarvittava ZIP-tiedosto voidaan ladata. Ympäristömuuttujiin määritellään Firestore-tietokannan project id, sekä funktion tarkistussumma koodin muuttumisen tarkistusta varten (kuva 38).

```
# Deploy the Cloud function
resource "google_cloudfunctions2_function" "function" {
  name        = "${var.environment}-${var.project}-function"
  location    = var.region
  description = "Google Cloud Function with HTTP trigger"
  depends_on  = [google_project_service.cloudfunctions, google_project_service.run]

  build_config {
    runtime     = "python311"
    entry_point = "main"
    source {
      storage_source {
        bucket = google_storage_bucket.function_bucket.name
        object = google_storage_bucket_object.function_archive.name
      }
    }
  }

  service_config {
    max_instance_count = 1
    available_memory   = "256M"
    timeout_seconds    = 120
    environment_variables = {
      FIRESTORE_PROJECT  = var.project_id
      FUNCTION_CODE_HASH = local.function_code_hash
    }
  }
}

# Make the function publicly accessible
resource "google_cloud_run_service_iam_member" "public_invoker" {
  location = google_cloudfunctions2_function.function.location
  service  = google_cloudfunctions2_function.function.name
  role     = "roles/run.invoker"
  member   = "allUsers"
}
```

KUVA 38. Funktiokoodin käyttöönotto ja julkaisu.

Funktiokoodissa määritellään ympäristömuuttujaa hyödyntäen project_id, jonka avulla funktio voidaan yhdistää Firestore-tietokantaan. Funktio main lisää tietokantaan item-olion ja palauttaa statuskoodin, viestin, sekä lisätyn olion sisältävän JSON-vastauksen funktiokutsun onnistuessa (kuva 39).

```python
import os
import json
from google.cloud import firestore

# Set up Firestore client
project_id = os.environ["FIRESTORE_PROJECT"]
client = firestore.Client(project=project_id)

def main(request):
    data = {
        "id": "123",
        "message": "Hello from Google Cloud Function v2 to Firestore"
    }
    try:
        client.collection("messages").add(data)
        return json.dumps({"message": "Data inserted into Firestore", "data": data}), 200
    except Exception as e:
        return json.dumps({"message": "Failed to insert data", "error": str(e)}), 500
```

Kuva 39. Cloud Run functions -koodi.

Ulostulona API Gatewayta varten asetetaan julkaistun Cloud Functionin URL-osoite (kuva 40).

```
output "function_url" {
  description = "The URL of the deployed Cloud Function"
  value       = google_cloudfunctions2_function.function.service_config[0].uri
}

```

KUVA 40. Functions-moduulin ulostulo.

Google Cloud -konfiguraatioille tyypillisesti, myös API Gateway -konfiguraation rakentaminen aloitetaan määrittelemällä tarvittavat API-palvelut API Gateway, Service Control, ja Service Management resurssilla ”google_project_service”. Seuraavaksi ”google_api_gateway_api”-resurssissa määritellään uusi API Gateway -instanssi. Sille asetetaan tarjoaja, id, näyttönimi, sekä riippuvuus API-palveluun (kuva 41). Huomionarvoista on, että työtä tehdessä kaikki API Gateway-resurssit kuuluivat vielä ”google-beta”-tarjoajan alle, joka sisältää ”ennakkovaiheessa” olevat resurssit.

```
# Enable required APIs
resource "google_project_service" "apigateway" {
  service            = "apigateway.googleapis.com"
  disable_on_destroy = false
}

resource "google_project_service" "service_control" {
  service            = "servicecontrol.googleapis.com"
  disable_on_destroy = false
}

resource "google_project_service" "service_management" {
  service            = "servicemanagement.googleapis.com"
  disable_on_destroy = false
}

# Define the Google API Gateway
resource "google_api_gateway_api" "api" {
  provider     = google-beta
  api_id       = "${var.environment}-${var.project}-api"
  display_name = "${var.environment}-${var.project} API"

  depends_on = [google_project_service.apigateway]
}
```

KUVA 41. Tarvittavien API-palveluiden ja API Gateway -instanssin määrittely.

”google_api_gateway_api_config”-resurssilla määritellään OpenAPI-spesifikaation avulla reitityssäännöt (kuva 42).

```
# API Config that defines the endpoint routing
resource "google_api_gateway_api_config" "api_config" {
  provider      = google-beta
  api           = google_api_gateway_api.api.api_id
  api_config_id = "${var.environment}-${var.project}-api-config"
  display_name  = "API Config for ${var.environment}-${var.project}"

  openapi_documents {
    document {
      path = "api_gateway_openapi.yaml"
      contents = base64encode(<<EOF
        swagger: "2.0"
        info:
          title: ${var.environment}-${var.project}-api
          version: "1.0"
        paths:
          /:
            post:
              operationId: postId
              x-google-backend:
                address: ${var.function_url}
              responses:
                '200':
                  description: A successful response
      EOF
      )
    }
  }
  depends_on = [google_api_gateway_api.api]
}
```

KUVA 42. API-konfiguraation määrittely.

API Gateway otetaan käyttöön ”google_api_gateway_gateway”-resurssilla. Resurssissa määritellään alue, API-konfiguraatio, sekä gateway id. Lopuksi

määritellään vielä ulostuloksi ”api_gateway_url”, joka luo API:lle käytettävän URL-osoitteen (kuva 43).

```
# Deploy the API Gateway
resource "google_api_gateway_gateway" "gateway" {
  provider   = google-beta
  region     = "europe-west1" # europe-north1 used elsewhere is not available in API GW
  api_config = google_api_gateway_api_config.api_config.id
  gateway_id = "${var.environment}-${var.project}-gateway"
}

# Output the API Gateway URL
output "api_gateway_url" {
  value = "https://${google_api_gateway_gateway.gateway.default_hostname}"
}
```

KUVA 43. API Gatewayn käyttöönotto ja URL-ulostulo.

7 POHDINTA

Opinnäytetyössä kehitetyt Terraform-konfiguraatiot tarjoavat selkeän pohjan serverless-sovellusten määrittelyyn ja hallintaan AWS-, Azure-, ja Google Cloud -pilvipalvelualustoilla. Konfiguraatiot eivät ole päässeet vielä varsinaiseen käyttöön, mutta ne antavat tilaajalle pohjan, jota voidaan hyödyntää sopivan projektin ilmetessä. Opinnäytetyön tavoitteena ollut ymmärryksen syventäminen on mielestäni onnistunut sen osalta, mitä opinnäytetyö on käsitellyt – reaalimaailma tuo varmasti esiin uusia näkökulmia ja haasteita, joita ei välttämättä tässä laajuudessa ole osattu ottaa huomioon.

Työn tuloksena syntyneet Terraform-konfiguraatiot sisältävät yhtenäisen ja modulaarisen rakenteen, johon kuuluu tietokanta, siihen liitetty FaaS-palvelu, sekä integroitu ohjelmointirajapinta. Modulaarisuuden ansiosta samoja komponentteja voidaan käyttää eri projekteissa ilman merkittäviä muutoksia, mikä vähentää ylimääräistä työtä ja nopeuttaa uusien sovellusten kehitysprosessia. Konfiguraatioita on myös mahdollista käyttää useammassa kehitysympäristössä samanaikaisesti.

Tulevaisuuden jatkokehitys- ja tutkimusmahdollisuuksia löytyy esimerkiksi konfiguraatioiden yhdistämisestä CI/CD-automaatioon (CI, continuous integration/ CD, continuous delivery) mahdollisten lisätyökalujen, kuten Terragrunt käytöstä, sekä infrastruktuurin tietoturvan parantamisesta.

Konfiguraatioilla on niiden dynaamisen rakenteen vuoksi tällä hetkellä valmius helppoon käyttöönottoon CI/CD-palveluissa kuten Bitbucket Pipelines ja GitHub Actions. Terragruntin tai vastaavien työkalujen avulla on mahdollista muun muassa automatisoida etätilan tallennuspaikan luonti, sekä vähentää syntyvää boilerplate-koodia, jota ei voida puhtaasti Terraformia käyttämällä välttää.

LÄHTEET

AWS. n.d. What is AWS Lambda? Verkkosivu. Viitattu 25.11.2024. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
,

AWS. 2022. Overview of Amazon DynamoDB. Verkkosivu. Viitattu 22.11.2024. https://docs.aws.amazon.com/whitepapers/latest/best-practices-for-migrating-from-rdbms-to-dynamodb/overview-of-amazon-dynamodb.html

AWS. 2024. Overview of Amazon Web Services. Verkkosivu. Viitattu 22.11.2024. https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html

Brikman, Y. 2022. Terraform: Up and Running. 3. painos. O'Reilly Media, Inc.

Cui, Y., Nair, A., Sbarski, P. 2022. Serverless Architectures on AWS. 2. painos. Manning Publications.

DigitalOcean. 2024. What is FaaS? Function as a Service Explained. Verkkosivu. Viitattu 8.12.2024. https://www.digitalocean.com/resources/articles/function-as-service-faas

Erl T, Barcelo E. 2023. Cloud computing: concepts, technology, security & architecture. 2. painos. New York: Pearson.

Google Cloud. n.d.a. Google Cloud overview. Verkkosivu. Viitattu 27.11.2024. https://cloud.google.com/docs/overview

Google Cloud. n.d.b. Cloud Run functions overview. Verkkosivu. Viitattu 27.11.2024. https://cloud.google.com/functions/docs/concepts/overview

Google Cloud. n.d.c. About API Gateway. Verkkosivu. Viitattu 28.11.2024. https://cloud.google.com/api-gateway/docs/about-api-gateway

Google Firebase. n.d. Cloud Firestore. Verkkosivu. Viitattu 27.11.2024. https://firebase.google.com/docs/firestore

HashiCorp. n.d.a. What is Terraform? Verkkosivu. Viitattu 5.11.2024. https://developer.hashicorp.com/terraform/intro

HashiCorp. n.d.b. Terraform CLI Documentation. Verkkosivu. Viitattu 31.1.2025. https://developer.hashicorp.com/terraform/cli/

HashiCorp. n.d.c. Resources. Verkkosivu. Viitattu 31.1.2025. https://developer.hashicorp.com/terraform/language/resources

HashiCorp. n.d.d. Providers. Verkkosivu. Viitattu 2.2.2025. https://developer.hashicorp.com/terraform/language/providers

HashiCorp. n.d.e. Variables and Outputs. Verkkosivu. Viitattu 3.2.2025. https://developer.hashicorp.com/terraform/language/values

HashiCorp. n.d.f. Modules. Verkkosivu. Viitattu 4.2.2025 https://developer.hashicorp.com/terraform/tutorials/modules/module

HashiCorp. n.d.g. State. Verkkosivu. Viitattu 8.12.2024. https://developer.hashicorp.com/terraform/language/state

IBM. n.d. What are Iaas, Paas and Saas? Verkkosivu. Viitattu 8.12.2024. https://www.ibm.com/topics/iaas-paas-saas

Katzer, J. 2021. Learning serverless: design, develop, and deploy with confidence. Sebastopol: O'Reilly Media, Inc.

Microsoft. 2024a. Welcome to Azure Cosmos DB. Verkkosivu. Viitattu 25.11.2024. https://learn.microsoft.com/en-us/azure/cosmos-db/introduction

Microsoft. 2024b. Azure Functions overview. Verkkosivu. Viitattu 25.11.2024. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview

Microsoft. 2024c. What is Azure API Management? Verkkosivu. Viitattu 25.11.2024. https://learn.microsoft.com/en-us/azure/api-management/api-management-key-concepts

Morris, K. 2021. Infrastructure As Code. 2. painos. Sebastopol: O'Reilly Media, Inc.

Red Hat. 2022. IaaS vs. PaaS vs. SaaS. Verkkosivu. Viitattu 9.1.2025. https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas

Spacelift. 2024. OpenTofu: The Open-Source Alternative to Terraform. Verkkosivu. Viitattu 20.11.2024. https://spacelift.io/blog/what-is-opentofu