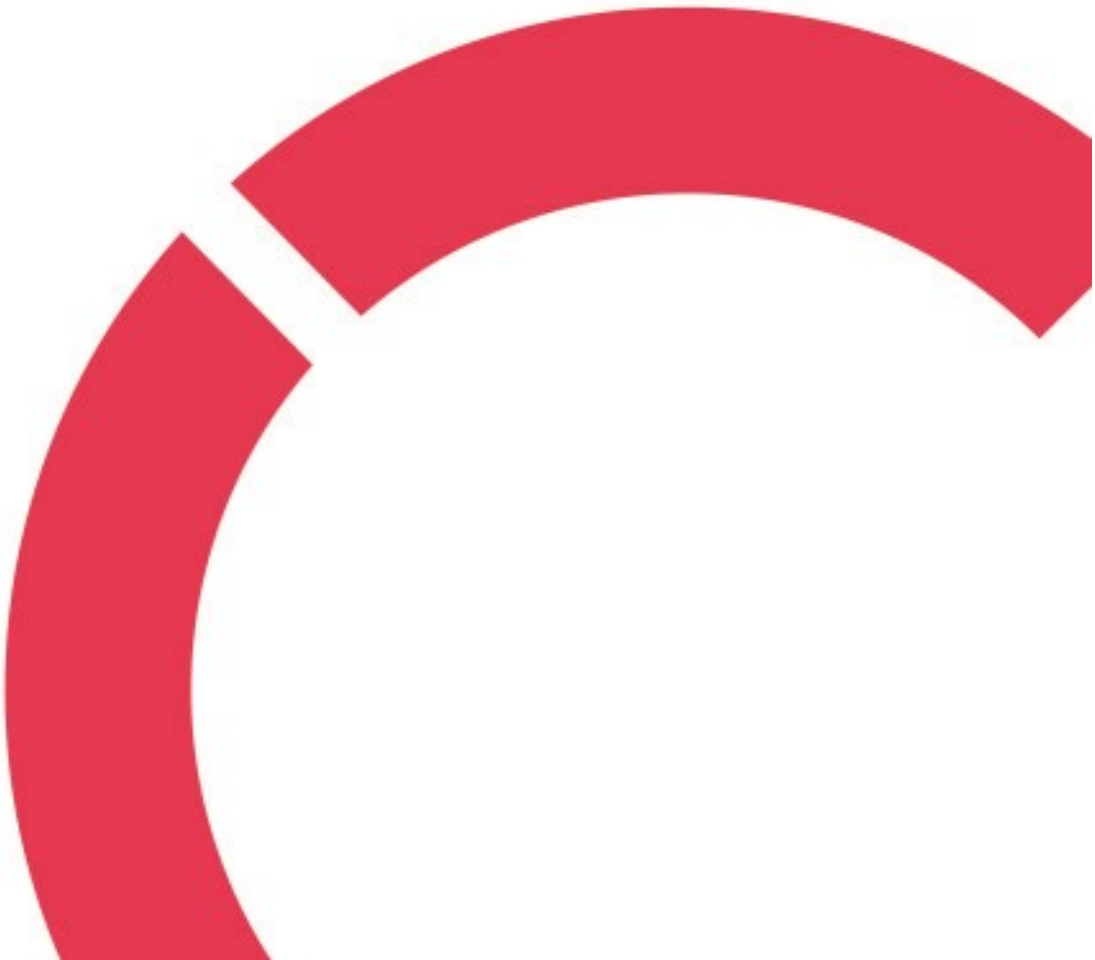
Heini Lehto

SOVELLUKSEN JA INFRASTRUKTUURIN AUTOMAATTINEN PROVISIOINTI AZUREEN

Kattava prototyyppi DevSecOps- ja IaC-käytännöistä

Opinnäytetyö
CENTRIA-AMMATTIKORKEAKOULU
Tieto- ja viestintätekniikan koulutus
Huhtikuu 2024



TIIVISTELMÄ OPINNÄYTETYÖSTÄ

Centria-ammattikorkeakoulu	**Aika** Huhtikuu 2024	**Tekijä/tekijät** Heini Lehto
Koulutus Tieto- ja viestintätekniikka		☒ AMK ☐ YAMK
Työn nimi SOVELLUKSEN JA INFRASTRUKTUURIN AUTOMAATTINEN PROVISIOINTI AZUREEN. Kattava prototyyppi DevSecOps- ja IaC-käytännöistä		
Työn ohjaaja Teppo Pirnes		**Sivumäärä** 39 + 1
Työelämäohjaaja -		

Opinnäytetyössä kuvattiin nykyaikaisen sovelluskehityksen periaatteita ja käytiin läpi sovelluskehityksen kokonaisvaltainen toteutusprosessi painottaen automaatiota, pilvi-infrastruktuurin luomista koodilla sekä jatkuvan integroinnin ja jatkuvan toimituksen vaiheita.

Opinnäytetyön teoriaosuudessa esiteltiin nykyään yleisesti käytettyjä ketteriä sovelluskehitysmenetelmiä sekä siihen liittyvää kulttuurimuutosta organisaatioissa. Lisäksi tarkasteltiin, miksi automaatio ja infrastruktuuri koodina -malli ovat nykyisessä moniulotteisessa sovelluskehityskulttuurissa hyvin tärkeitä, jopa välttämättömiä elementtejä.

Käytännön osuudessa toteutettiin tuotantokelpoinen malliratkaisu sovelluksesta, joka hyödyntää taustalla koneoppimista. Malliratkaisun käyttötapauksena toimi lumenmäärän kehittymistä ennustava sovellus, joka toteutettiin staattisena verkkosovelluksena. Sovellus toimi opinnäytetyössä sovelluskehitysprosessin esittelyn tukena ja sen osalta painotettiin sovelluksen vaatiman pilvi-infrastruktuurin luomista, perusteltiin sen arkkitehtuuria sekä kuvailtiin jatkuvan integroinnin ja jatkuvan toimituksen a prosessia.

Tavoitteena oli havainnollistaa, kuinka alusta alkaen toteutettu ja pilvipalvelualustalle asennettu tuotantokelpoinen sovellus vaatii toimiakseen todella paljon muitakin elementtejä ja toimenpiteitä, kuin vain pelkän ohjelmakoodin kehittämistä. Opinnäytetyössä esitettiin malli, kuinka tämä mahdollistetaan nykyaikaisia menetelmiä hyödyntäen.

Opinnäytetyön kehittämistuotoksena oli toimiva sovellus, jonka kehittämis- ja julkaisuprosessi saatiin kuvattua opinnäytetyössä kokonaisvaltaisesti. Työn kehittämisen yhteydessä saatiin vahvistusta ohjelmistokehityksen nykyaikaisten kehitysmenetelmien hyödyllisyydestä ja tärkeydestä.

Asiasanat
Automaatio, CI/CD-työnkulku, DevOps, DevSecOps, IaC

ABSTRACT

Centria University of Applied Sciences	Date April 2024	Author Heini Lehto
Degree programme Information and communication technology		
Name of thesis AUTOMATIC PROVISIONING OF APPLICATION AND INFRASTRUCTURE IN AZURE. Comprehensive prototype of DevSecOps and IaC practices		
Centria supervisor Teppo Pirnes		**Pages** 39 + 1
Instructor representing commissioning institution or company -		

The thesis described the principles of modern software development and covered the comprehensive implementation process of software development, emphasizing automation, creation of cloud infrastructure with code, and the phases of continuous integration and continuous delivery.

The theoretical part of the work introduced commonly used agile application development methodologies and the associated cultural change within organizations. Additionally, it explained why automation and the infrastructure as code model are crucial elements in the current multidimensional application development culture, and are even deemed necessary.

In the practical part, a production-ready model solution was implemented that showcases an application utilizing machine learning at its core. The use case for the application was a prediction of snow depth, presented as a static web view. The application served as a support for illustrating the application development process in the thesis, with a focus on creating the necessary cloud infrastructure for the application, justifying its architecture, and describing the continuous integration and continuous delivery process.

The goal was to illustrate how a production-ready application, produced and installed on a cloud service platform from the ground up, requires many elements to function properly beyond just the program code.

As a result of the work, a fully functional application was developed, and the development and publishing process were comprehensively described in the thesis. Confirmation regarding the usefulness and importance of modern software development methods was obtained during the development process.

Key words
Automation, CI/CD-pipelines, DevOps, DevSecOps, IaC

KÄSITTEIDEN MÄÄRITTELY

API

API tulee englannin kielen sanoista Application Programming Interface ja sen avulla eri ohjelmistot ja sovellukset voivat kommunikoida keskenään.

AZURE

Microsoftin pilvipalveluita tarjoava alusta.

CI/CD

Continuous Integration/Continuous Delivery - jatkuva integrointi/jatkuva toimitus. Kehityskäytäntö, jossa ohjelmistomuutokset integroidaan koodikantaan ja uudet versiot julkaistaan automaation avulla suoritusympäristöihin.

CLI

Lyhennelmä sanoista Command Line Interface – komentorivikäyttöliittymä. Ohjelma, jossa syötetään tekstimuotoisia komentoja, joiden avulla käyttöjärjestelmä kykenee suorittamaan komennon mukaisia toimintoja.

DevOps

Lyhennelmä sanoista Development ja Operations - kehitys ja operaatiot. Toimintamalli, jolla organisaatiot tehostavat ja parantavat kykyään toimittaa ohjelmistoja sekä sovelluksia.

DevSecOps

Lyhennelmä sanoista Development, Security ja Operations - kehitys, turvallisuus ja operaatiot. Toimintamalli, jossa tietoturva on osana koko ohjelmistokehityksen elinkaarta.

IaC

Infrastructure as code - infrastruktuuri koodina. Tapa tuottaa ja hallita infrastruktuuria koodin avulla ilman laitteistojen ja käyttöjärjestelmien manuaalista konfigurointia.

INFRASTRUKTUURI

Infrastruktuurilla tarkoitetaan pilvessä olevia resursseja, joita tarvitaan sovellusten suoritusympäristön isännöintiin ja kehittämiseen.

ON-PREMISES

Kaikki tarvittavat laitteet, ohjelmistot sekä tietojärjestelmät sijaitsevat ja niitä hallitaan organisaation omassa konesalissa tai datakeskuksessa.

PILVI

Pilvi tai pilviteknologia viittaa palvelimiin, jotka ovat saatavilla internetin kautta. Pilvipalvelimet sijaitsevat datakeskuksissa ympäri maailmaa.

RBAC

Roolipohjainen pääsynhallinta, joka tulee sanoista Role-Based-Access-Control, jossa käyttäjille tai järjestelmille annetaan pääsyoikeudet järjestelmän resursseihin perustuen heidän rooliinsa organisaatiossa.

RESURSSI

Pilvipalvelussa oleva yksittäinen käyttöön otettava komponentti, joka ilmentää kyseiselle resurssityypille ominaisia piirteitä, esimerkiksi suorittaa laskentatoimenpiteitä tai tarjoaa tiedon varastointia.

SERVERLESS-ARKKITEHTUURI

Serverless eli palvelimeton arkkitehtuuri on sovelluksen isännöinnin muoto, jossa kehittäjän ei tarvitse huolehtia palvelinten ylläpidosta tai hallinnasta, sillä pilvipalveluntarjoaja ylläpitää palvelinympäristöä.

SERVICE PRINCIPAL

Sovelluskäyttäjä, joka ei ole sidoksissa tyypillisesti luonnollisten henkilöiden käyttäjätunnukseen, ollen Microsoft Azure-palvelussa oleva koneellinen identiteetti, joka käyttää RBAC-oikeuksia Azure-resurssien hallinnointiin.

KAAVIOT

1 JOHDANTO

Tässä opinnäytetyössä käsitellään nykyaikaista ketteriin toimintatapoihin perustuvaa kokonaisvaltaista sovelluskehitystä. Työssä keskitytään käytäntöihin, jotka edistävät laadukkaan sovelluksen kehittämistä, sen elinkaaren hallintaa ja tietoturvan huomioimista kehityksen eri osa-alueilla. Opinnäytetyössä läpikäytävät käytännöt mahdollistavat myös automaation jatkuvan integroimisen osana sovelluksen kehitys- ja julkaisuprosesseja. Näitä käytäntöjä sovelletaan työssä toteutettavassa kehitystehtävässä, jossa sovellus ja sen pilvi-infrastruktuuri julkaistaan automaation avulla Microsoft Azure -pilvialustalle. Kehitystehtävän lopputulosta voidaan pitää malliratkaisuna, jota voi jatkokehittää ja soveltaa muissa sovellustarpeissa. Työssä korostetaan myös, että yksittäisen sovelluksen tuottaminen ja asentaminen tuotantoympäristöön vaatii paljon monipuolisempiakin toimenpiteitä kuin pelkästään sovelluksen ohjelmointia. Tietojärjestelmiin liittyvien ratkaisujen toteuttajien ja ylläpitäjien on hallittava elinkaaren ja tietoturvan osa-alueet, erityisesti, kun organisaatioissa siirrytään voimakkaasti pilvipalveluihin, sillä se lisää sovellusarkkitehtuurin monimutkaisuutta ja monimuotoisuutta verrattuna perinteisempiin on-premises-palveluihin.

Opinnäytetyössä yhdistetään tietotekniikan opinnoissa käsitellyt yksittäiset sovelluskehitykseen liittyvät aiheet yhdeksi kokonaisuudeksi ja tätä kokonaisuutta syvennetään sekä laajennetaan vastaamaan työelämän korkeita vaatimuksia. Näihin vaatimuksiin pyritään vastaamaan hyödyntämällä automaatiota ja luomaan sovelluksen pilvi-infrastruktuuri täysin koodin avulla. Kokonaisuus esitetään konkreettisesti koneoppimiseen perustuvan lumen määrää ennustavan sovelluksen suunnittelu-, kehitys- ja julkaisuprosessien avulla. Työn taustana on tarve tutkia ja toteuttaa niitä vaiheita, jotka ovat välttämättömiä laadukkaan ja tietoturvallisen sovelluksen onnistuneelle julkaisulle tuotantotasoisessa ympäristössä.

Aiheen laajuutta kompensoidaan keskittämällä tarkempi kehittämistehtävän läpikäynti infrastruktuuri koodina -mallin sekä jatkuvan integroinnin ja jatkuvan toimituksen prosesseihin. Sen sijaan koneoppimissovellusta käsitellään ikään kuin valmiina ohjelmakomponenttina, ja sen eri osa-alueita tarkastellaan vain niiltä osin, kuin ne tukevat opinnäytetyön pääaiheiden läpikäyntiä. Esimerkiksi sovelluksen arkkitehtuurin kuvaaminen tukee työssä esitettävän kokonaisratkaisun hahmottamista. Teoriaosuudessa käytännönosuuteen vahvasti vaikuttavien eri ohjelmointi- ja merkkauskielten syntaksin läpikäynti rajataan teoriaosuudesta pois, eikä myöskään eri työkalujen tai järjestelmien käyttöä käydä tässä opinnäy-

tetyössä läpi. Teoriaosuuden rajauksesta huolimatta opinnäytetyön konkreettisena osana syntyy tuotantotasoinen malliratkaisu, jolla uusi, sovellusta palveleva pilviympäristö ohjelmakomponentteineen provisioituu muutamissa minuuteissa.

Opinnäytetyössä hyödynnetään tieto- ja viestintätekniikan opetussuunnitelmaan sisältyvien opintojaksojen sekä muutamien opetussuunnitelman ulkopuolisten opintojaksojen myötä saatua tietoa. Merkittävimpinä opintojaksoja tai yksittäisiä aiheita ovat ohjelmointiin sekä eri ohjelmointi- ja skriptauskieliin liittyvät opintojaksot, Azureen pohjautuvat opintojaksot sekä ohjelmistoarkkitehtuuria, DevOpsia ja DevSecOpsia käsittelevät opintokokonaisuudet. Lisäksi opinnäytetyön yhteydessä tehdyssä kehittämistehtävässä näkyy vahvasti työelämässä saatu kokemus, joka heijastuu eniten laadukkaan sovelluksen ohjelmalliseen toteutukseen, infrastruktuurin tuottamiseen koodina, sovellukseen liittyvien pilviresurssien yhteen kytkemisessä toimivaksi kokonaisuudeksi sekä valmiin sovelluksen provisioinnissa automatisoiduilla työnkuluilla versionhallintajärjestelmää ja julkaisualustaa hyödyntäen.

Opinnäytetyössä käytettäviä keskeisimpiä käsitteitä ovat DevOps ja DevSecOps, jotka ovat erityisesti ohjelmistokehityksen alalla hyödynnettyjä toimintamalleja. Käytännön osuus perustuu suurelta osin IaC-menetelmään, joka tarkoittaa sovelluksen pilvi-infrastruktuurin luomista koodilla. Lisäksi käytännön osuudessa hyödynnetään CI/CD-prosessia, joka lyhyesti ilmaistuna tarkoittaa sovelluskoodin julkaisemista kehittäjän työasemalta vaiheittaisesti automatisoitujen työnkulkujen kautta suoritusympäristöön. Hyvin yleisesti työssä esiintyvä sana resurssi tai pilviresurssi tarkoittaa Microsoft Azuren pilvialustalla käyttöönotettavaa komponenttia, joka tarjoaa infrastruktuurille ominaisia toimintoja pilvipalvelussa.

2 SOVELLUSKEHITYKSEN ELINKAARIMALLI

Ohjelmistokehityksen elinkaarimalli on olennainen osa kehitysprosessia, ja se auttaa organisaatioita hallitsemaan ja ohjaamaan projektin eri vaiheita tehokkaasti. Hyvä malli edesauttaa kustannustehokkaan, laadukkaan ja laajasti testatun sovelluksen tuottamista ja julkaisemista loppukäyttäjille. Sovelluskehityksen elinkaarimalli kuvaa ohjelmistoprojektin eri vaiheet alkaen ideasta ja vaatimusmäärittelystä ja se päättyy ylläpitoon ja jatkokehitykseen. Laadukkaan elinkaarimallin tunnusmerkkejä ovat ketterien toimintamenetelmän jalkauttaminen kaikkiin projektin vaiheisiin, automaatiota hyödyntävien prosessien ja työkalujen käyttöönotto sekä projektissa olevien henkilöiden hyvä ymmärrys siitä, mitä todella on tarkoitus tehdä ja miksi. (Alexandra 2023.)

2.1 Jatkuvan kehittämisen ja toiminnan DevOps-menetelmä ohjelmistokehityksessä

Ohjelmistokehityksessä nykyään hyvin yleisesti käytetty menetelmä DevOps on laaja käsite, jolla on useita eri tarkoituksia, ja sen tarkka määritelmä vaihtelee hieman lähteen mukaan. Lyhenne DevOps tulee sanoista development ja operations, jotka suomeksi tarkoittavat kehitystä ja toimintaa. Se on itseohjautuvien tiimien muodostamista projektin ympärille siten, että luodaan innovatiivisuutta ruokkivat puitteet projektin kehittämiselle. Se ei ole työkalu sen enempää kuin varsinainen prosessikaan, sillä DevOps voidaan mieltää ennemminkin muutokseksi organisaation kulttuuriin vaikuttavissa ihmisissä, jotka luovat uusia toimintatapoja projektitoiminnan ympärille. (Kivelä 2023.) Lopulta kyse on siitä, ettei yksikään tiimi tai henkilö voi ottaa vastuuta koko projektista. Sen sijaan vastuu jaetaan useamman tiimin kesken, korostaen yhteistyötä näiden tiimien ja yksittäisten henkilöiden välillä.

Useissa eri lähteissä DevOps esitetään eräänlaisena kahdeksikon muotoisena jatkumona, joka kuvaa ohjelmistokehityksen elinkaaren vaiheita käytännössä. Eri vaiheita järjestyksessään ovat suunnittelu, ohjelmointi, ohjelmakoodin koostaminen, testaus, julkaisu, käyttöönotto sekä valvonta. Keskeisintä mallissa on se, ettei ohjelmiston kehitys suinkaan lopu valvontaan, vaan se voi jatkua jälleen suunnitteluvaiheesta, jossa huomioidaan aikaisemman kierroksen testaus- ja valvontahavainnot sekä käyttäjiltä saatu palaute. (Bigelow, Courtemanche & Gillis 2023.)

DevOpsin päätavoite on tuottaa laadukkaampia ohjelmistoja nopeammin ja tehokkaammin, mikä vaatii kehitys- ja muiden tiimien välistä jatkuvaa yhteistyötä. Pääsääntöisesti DevOps-periaatteiden mukaan

projektissa edetään vaiheesta seuraavaan lyhyellä syklillä, sopivan pienien kehitystavoitteiden mukaisesti. Loppupäämäärään ei ole tarkoitus päästä yhdellä kertaa siten, että julkaisuvaiheessa käsillä olisi täydellinen kokonaisuus. DevOps painottaa osiin jakamista, jolloin esimerkiksi voidaan suunnitella ja toteuttaa sovelluksen jokin tietty toiminnallisuus, testata ja julkaista se ja sen jälkeen siirtyä seuraavan toiminnallisuuden kehittämiseen.

2.2 Tietoturva tärkeänä osana ohjelmistokehityksen elinkaarta

Tietoturvan integroiminen ohjelmistokehityksen elinkaaren kehitysvaiheisiin on laadukkaasti toteutetun ohjelmiston yksi merkittävimmistä tekijöistä. Tietoturvaan liittyviä prosesseja ja käytänteitä kuvataan DevSecOps-mallin avulla, jossa tietoturva otetaan osaksi DevOps-käytänteitä. Se integroi tietoturvatoimenpiteet, käytännöt ja vastuut jatkuvasti kehitys- ja toimitusprosessiin. Tietoturvan huomioiminen tarkoittaa erilaisten tietoturvaa parantavien käytäntöjen luomista, työkalujen käyttöönottoa, tietoturvatyövaiheiden automatisointia, tietoturvatoimien vastuuttamista oikeille osapuolille sekä sovellusten valvontaa. Organisaatiotasolla tietoturvan ylläpito ja kehittäminen tarkoittavat koulutusta, tiedon jakamista, turvallisten kehityskäytäntöjen luomista, laadukasta suunnittelua sekä jatkuvaa tietoturvan arviointia. Organisaatioissa tulisi luoda puitteet saumattomalle yhteistyölle kehitys-, ylläpito- ja tietoturvatiimien välille. (What is DevSecOps 2023.)

Käytännön tasolla tietoturvan huomioiminen ohjelmistokehityksessä tarkoittaa tietoturvariskien analysointia jo suunnitteluvaiheessa, kehitysvaiheessa tietoturvatestauksen sisällyttämistä prosessiin sekä tuotannossa olevan ohjelmiston jatkuvaa tietoturvallisuuden uudelleenarviointia. Parhaimpia käytäntöjä tietoturvan huomioimiselle sovelluskehityksessä ovat projektin tietoturvavaatimusten määrittely, potentiaalisten haavoittuvuusriskien tunnistaminen, hyvien kehityskäytäntöjen standardointi, käytettävien koodikirjastojen ja ohjelmistokehysten päivittäminen, tietoturvakoulutukset, tietokantojen suojaaminen sekä ylläpitovaiheessa sovelluksessa tapahtuvien virhetilanteiden monitorointi ja niihin reagoiminen. (Segura 2023.)

Automatisoinnin hyödyntäminen sovelluksen suojaamisessa on erittäin suositeltavaa ja sitä voidaankin hyödyntää staattisessa ja dynaamisessa sovelluksen tietoturvatestaamisessa. Ohjelmiston kehitysvaiheessa tehtävä staattinen sovelluksen tietoturvatestaus analysoi ohjelmiston lähdekoodin ilman koodin suorittamista ja etsii potentiaalisia tietoturvahaavoittuvuuksia, kuten heikosti kirjoitettua koodia, puut-

teellisia tietoturvakonfiguraatioita, vanhentuneita kolmannen osapuolen kirjastoja tai muita turvallisuusriskejä. Dynaaminen sovelluksen tietoturvatestaus puolestaan tehdään ohjelman suorituksen aikana. Dynaaminen testaus simuloi hyökkääjän toimia, kuten syöttää virheellistä tietoa lomakkeisiin, manipuloi URL-parametreja ja skannaa ohjelman käyttöliittymää. (SAST vs. DAST 2023.)

Lisäksi pilvipalveluntarjoajilla on erilaisia maksullisia tietoturvapalveluita, jotka perustuvat automaattiseen infrastruktuurin ja sovelluksen monitorointiin. Esimerkiksi Microsoft Azuren tarjoama Microsoft Defender EASM (External Attack Surface Management) kartoittaa Azuressa olevien sovellusten mahdollisia hyökkäyspintoja ja luo niistä koostenäkymiä, joiden perusteella tietoturvatiimit voivat tehdä kohdistettuja toimenpiteitä sovelluksen tietoturvan parantamiseksi (Defender EASM Overview 2023). Azure tarjoaa myös toisenlaisen tietoturvapalvelun, Defender for Cloud, jota voidaan hyödyntää jo sovelluksen kehitysvaiheessa. Sen avulla voidaan suojata koodin versionhallinta- ja julkaisualustaa. (What is Microsoft Defender for Cloud? 2023.)

2.3 Automatisointi ohjelmistokehityksen eri vaiheissa

Ohjelmistokehityksessä suositaan automatisointiin tarkoitettuja työkaluja ja tekniikoita ohjelmiston elinkaaren jokaisessa vaiheessa niiltä osin, kuin se katsotaan järkeväksi. Eri vaiheita ovat määrittely, suunnittelu, toteutus, testaus, julkaisu ja ylläpito. Määrittelyvaiheen automatisointi ei kaikissa tapauksissa ole kuitenkaan järkevää, sillä määritysten laatimiseen tarvitaan useimmiten ihmisen työpanosta ja tulkintaa. (Science Soft 2023.) Muihin vaiheisiin on tarjolla hyviksi todettuja työkaluja ja menetelmiä ja sovelluksen elinkaaren kannalta automaatiota tulisi hyödyntää etenkin testaus-, toteutus- ja julkaisuvaiheissa.

Suunnitteluvaiheessa voidaan hyödyntää erilaisia automaatioon perustuvia mallintamis- ja analysointityökaluja. Esimerkiksi ohjelmiston suunnitellusta arkkitehtuurista voidaan analyysityökalujen avulla tunnistaa mahdollisia ongelmia, jotka heikentävät laatua. Myös ohjelmiston toteutusvaiheessa käytetään usein erilaisia ohjelmistoja ja työkaluja, jotka automatisoivat esimerkiksi koodin tuottamisen. Useat työkalut ja alustat tarjoavat automaattista koodin täydentämistä, syntaksin korostamista sekä vianjäljitysominaisuuden. Testauksen automatisointi on oleellinen osa ohjelmiston laadun varmistamista ja hyvin tyypillisesti automatisoitavia testaustyyppejä ovat regressio- ja integraatiotestaus, testaus eri selaimilla, suorituskykytestit sekä tietoturvatestaus. Julkaisuvaiheessa automatisoidaan sovel-

luksen julkaisu eri ympäristöihin, kuten kehitys-, testaus- ja tuotantoympäristöön. Osana julkaisuprosessia ohjelmakoodit koostetaan, testataan ja julkaistaan automatisoidusti jatkuvien integrointi- ja toimitustyönkulkujen avulla. (Science Soft 2023.)

2.4 Jatkuva integrointi kokoaa sovellusversion julkaisupaketin

Jatkuva integrointi (Continuous Integration) on osa jatkuvan integroinnin ja jatkuvan toimituksen (Continuous Delivery) prosessia (CI/CD). Jatkuvan integroinnin vaiheessa eri ohjelmistokehittäjien tekemät koodimuutokset integroidaan säännöllisesti ja usein yhteiseen sovelluksen koodikantaan. Kehittäjä julkaisee tekemänsä koodimuutokset versionhallintajärjestelmän avulla versionhallinnan julkaisualustalle, kuten esimerkiksi GitHubiin. CI-prosessissa koodi koostetaan ja sille suoritetaan usein myös automaattiset yksikkö- ja integraatiotestit, joiden avulla varmistetaan, etteivät muutokset ole rikkoneet sovelluksen aiempia ominaisuuksia, käyttäytymistä tai rajapintasopimuksia. Testien täytyy näin ollen kattaa ohjelmakoodi kokonaisuutena aina yksittäisistä toiminnallisuuksista kokonaisiin moduuleihin asti. (RedHat 2023.)

Mikäli versio onnistutaan koostamaan ja kaikki testit ajamaan onnistuneesti, vahvistetaan koodimuutokset luomalla siitä CI-prosessissa uusi versiopaketti julkaisua varten. Prosessissa versiopaketti tallennetaan julkaisualustan sisäiseen pakettivarastoon niin sanottuna jäädytettynä artefaktina, josta se voidaan muuttumattomana noutaa prosessin seuraavassa, jatkuvan toimituksen vaiheessa.

2.5 Sovellusversioiden julkaisumenetelmät

Jatkuvan integroinnin jälkeen sovellus tulee voida julkaista loppukäyttäjille siten, että tehdyt muutokset on testauksen avulla todettu toimiviksi. Sovelluksen julkaisu tapahtuu CD-prosessin avulla, joka voi koostua jatkuvan toimituksen (Continuous Delivery) tai jatkuvan käyttöönoton (Continuous Deployment) menetelmästä tai näiden kahden yhdistelmästä. Menetelmän valintaan vaikuttavat esimerkiksi organisaation tai tiimin DevOps-prosessien maturiteettitaso ja tiimissä kehitetyt parhaat kehitystavat sekä erilaiset lakiin, määräyksiin tai ohjeisiin perustuvat säännökset. Jatkuvan käyttöönoton menetelmässä hyödynnetään enemmän automaatiota jatkuvan toimituksen menetelmään verrattuna, jolloin kriittisemmissä sovelluksissa tulisi pysyttäytyä jatkuvassa toimituksessa sen ennustettavuuden vuoksi, vaikkakin hitaamman julkaisusyklin kustannuksella. (Yazar 2023.)

Tiimit, jotka ovat ottaneet DevOps-periaatteet vahvasti osaksi sovelluskehitystä, valitsevat useimmiten jatkuvan toimituksen menetelmän. Jatkuvassa toimituksessa sovellusversion julkaisu tuotantoympäristöön ei tapahdu automaattisesti, vaan se vaatii aina manuaalisen julkaisun hyväksymisen. Hyväksyntä edellyttää usein myös lisätestausta sekä laadunvarmistusta tuotannon kanssa identtisessä ympäristössä, jossa esimerkiksi suorituskykyä voidaan testata. Onnistuneen testauksen jälkeen sovellukseen kohdistuva liikenne ohjataan tähän uuteen ympäristöön ja loppukäyttäjät saavat uuden version käyttöönsä. (Gillis 2021.)

Jatkuvan käyttöönoton menetelmässä CI-prosessissa luotu versiopaketti julkaistaan haluttuun ympäristöön automaattisesti. Jatkuvassa käyttöönotossa oletetaan, että julkaistava koodi on käännös- ja testaustoimenpiteiden avulla toimivaksi todettu ja että se voidaan julkaista sellaisenaan tuotantoympäristöön eli loppukäyttäjille, jolloin erillistä tuotantoympäristön kaltaiseksi lavastettua ympäristöä ei tarvita. (Yazar 2023.) On kuitenkin hyvin yleinen käytäntö lisätä niin sanottua savutestausta eli sovelluksen kriittisten komponenttien ja toimintojen testausta jatkuvan käyttöönoton vaiheeseen. Sillä varmistetaan, että sovellus tai ohjelmisto toimii myös oikeassa kohdeympäristössä eli loppukäyttäjän laitteella samalla tavalla kuten kehitysympäristössä. (Rivera 2020.)

Versiopaketin julkaisua voidaan myös vaiheistaa julkaisemalla siitä tiettyjä osia eri kohderyhmille tai palvelimille ennen laajamittaisempaa julkaisemista. Tätä Canary Deployment -menetelmäksi kutsuttua prosessia käytetään, kun halutaan varmistaa sovelluksen toimivuus ensin pienellä käyttäjäryhmällä, jotta siinä mahdollisesti ilmenevät ongelmat eivät pilaisi suuren yleisön käyttökokemusta. (Dodd 2023.)

3 PILVIPALVELUT OHJELMISTOKEHITYKSESSÄ

Perinteiseen on-premises-ratkaisuun verrattuna pilvipalvelujen käyttö on yleistynyt sovellusinfrastruktuurin hankinnassa. Pilvipalvelut tarjoavat ostettavia ja etähallittavia sovellusten ja ohjelmistojen isännöintiin tarkoitettuja resursseja, kuten laskentatehoa, tallennustilaa ja erilaisia verkkotason ratkaisuita. Pilvi toimii ikään kuin hajautettuna tietoverkkona, jonka käyttö on nykyään hyvin keskeisessä roolissa ohjelmistokehityksessä. Pilvipalvelut tarjoavat skaalautuvuutta, joustavuutta ja tehokkuutta ohjelmistokehityksen eri vaiheissa. Ne tuovat käyttäjille etuja, kuten kustannustehokkuutta, ketterämpää ja nopeampaa provisiointia sekä ylläpidon helppoutta. Monista eduista huolimatta pilvipalvelujen käyttöönotossa on huomioitava erityisesti tietoturvaan liittyvät näkökulmat. (Sakovich 2023.) Lisäksi pilvipalveluiden käyttö voi tulla kalliiksi, mikäli aiemmin omien konesalien kautta ajettuja sovelluksia siirretään sellaisenaan pilviympäristöön ilman muokkauksia pilven parhaiden käytäntöjen mukaisesti (Microsoft 2023, 353).

Vaikka pilvipalvelujen käyttö on yleistynyt koko 2000-luvun, on edelleen lukuisia sovelluksia, joiden julkaisu pilvessä ei ole mahdollista sovelluksen luonteesta johtuen. Esimerkiksi olemassa olevan sovelluksen siirtäminen pilveen voi olla mahdotonta, mikäli se perustuu vanhentuneisiin alustoihin tai sen rakenne ei ole yhteensopiva pilvialustan kanssa. Sovelluksen vaatima turvallisuustaso voi myös olla niin korkea, että sen tiedot on pakko säilyttää organisaation omilla palvelimilla ja tietovarastoissa. (Sakovich 2023.)

3.1 Pilvipalvelujen tasot

Pilvipalveluiden palvelukattavuudessa on erilaisia tasoja, jotka kuvaavat sitä, kuinka laajasti palvelun hankinnasta vastaava organisaatio on vastuussa pilvialustalta hankittujen palveluiden ja komponenttien käyttöönotosta ja ylläpidosta. Vastuut jaetaan organisaation tai loppukäyttäjän ja pilvialustan tarjoajan välillä. Yleisimmät palvelutyypit ovat Software as a Service (SaaS), Platform as a Service (PaaS) sekä Infrastructure as a Service (IaaS). (Rosencrance 2021).

SaaS-palvelutasolla loppukäyttäjällä on vähäisin vastuu ja pilvipalveluntarjoaja on vastuussa laitteistosta ja tietoliikenneverkoista aina sovelluksen ylläpitoon saakka. Loppukäyttäjä vastaa sisällöstä ja tiedonhallinnasta, käyttöoikeuksien, salasanojen ja käyttäjätunnusten jakamisesta ja ylläpidosta sekä

sovelluksen tietoturvaan liittyvien konfigurointien määrittelystä (Rod 2023). SaaS-palveluja käytetään lähtökohtaisesti aina verkkoselaimella, eikä erillisen työpöytäsovelluksen lataamista tarvita (Watts & Muhammad 2019). Yksi yleisimmistä SaaS-palveluista on esimerkiksi Microsoftin tarjoama Office 365, joka sisältää mm. Outlook-sähköpostipalvelun.

PaaS-palvelutasolla käyttäjän ja palveluntarjoajan väliset vastuuerot hieman kapenevat, kun käyttäjä on vastuussa datan ja pääsynhallinnan lisäksi myös sovelluksista ja niiden ylläpidosta. PaaS-palvelut ovatkin suunnattu etenkin sovelluskehittäjille tarjoten heille sopivan helppokäyttöisen alustan sovelluskehitystä varten. Edelleen palveluntarjoaja huolehtii kaikesta muusta, kuten ohjelman suoritusympäristön toiminnasta, käyttöjärjestelmistä, virtuaalikoneista, tallennustilasta, palvelimista ja tietoliikenneverkoista. (Watts & Muhammad 2019.)

IaaS-palveluissa käyttäjät hallinnoivat koko infrastruktuuria ja he voivat käyttää palvelimia sekä tallennustilaa lähes on-premises-tasolla, mutta virtuaalisen datakeskuksen välityksellä. Pilvipalveluntarjoaja vastaa palvelimien toiminnasta, datan säilyttämisestä sekä tietoliikenneverkoista ja tarjoaa vain alustan infrastruktuurin asennukselle ja käyttöönotolle. (Watts & Muhammad 2019.)

Edellä mainittujen lisäksi neljäs, ehkä hieman vähemmän tunnettu palvelutyyppi on Function as a Service (FaaS), jossa käyttäjän ei tarvitse huolehtia pilveen julkaistujen koodifunktioiden suoritusympäristöstä. FaaS on varteenotettava palvelutyyppi, kun sovellus on mahdollista hajauttaa pienempiin toiminnallisiin kokonaisuuksiin, jotka kuitenkin lopulta kytkeytyvät toisiinsa. (Microsoft 2023, 371.)

3.2 Iac - Infrastruktuuri koodina

Ennen kuin ohjelmistojen vaatimaa pilvi-infrastruktuuria alettiin kehittämään koodin avulla, ohjelmoijat kehittivät vain omaa ohjelmakoodiaan, eikä heidän tarvinnut välittää tietojärjestelmästä kokonaisuutena. Infrastruktuurista vastasivat tiimissä olleet tietojärjestelmäasiantuntijat, jotka huolehtivat järjestelmien infrastruktuurin asentamisesta, ylläpidosta sekä sen dokumentoinnista. DevOps-ajattelun myötä ohjelmistokehittäjien ja tietojärjestelmäasiantuntijoiden välinen kuilu on kaventunut ja nykyään yksittäinen projektihenkilö voikin olla näiden molempien osa-alueiden asiantuntija. Täten kaikki tiimin jäsenet osallistuvat sekä ohjelmakoodin kirjoittamiseen että infrastruktuurin luomiseen ja ylläpitämiseen. Tämä ilmiö on osaltaan ajanut Infrastructure as Code eli IaC-mallin syntymiseen, joka edesauttaa järjestelmäkokonaisuuksien hallintaa. (Falck 2023.)

IaC on eräänlainen automaatiotyökalu, jonka juuret juontuvat 2000-luvun alkupuolelle, kun ensimmäisiä IaC-työkaluja alettiin kehittämään Internetin kasvun ja sovellusympäristöjen ylläpidon vaikeutumisen myötä. Kun vahva pilvipalveluiden käytön kasvu odotteli vielä omaa vuoroaan ja palvelimet olivat palvelintelineisiin asennettavia fyysisiä rautakomponentteja, kattoi automaatio vain sovellusympäristöjen asennuksen ja tuotantoon viennin. Myöhemmin pilvipalveluiden yleistyessä tuli tarpeelliseksi automatisoida myös pilvialustalle konfiguroitavien infrastruktuurikomponenttien asennus. Tämä mahdollisti esimerkiksi virtuaalipalvelimien ja verkkoyhteyksien määrittelyn. (Falck 2023.)

Infrastructure as Code tarkoittaa nimensä mukaisesti infrastruktuurin luomista koodina, jolloin kaikki manuaalinen ja käsin pilvialustan käyttöliittymästä tehtävä toiminta jää pois. Infrastruktuurikoodi voidaan tallentaa versionhallintaan, jolloin sen muutoshistoriaa on helppo seurata ja muutosten aiheuttamia ongelmia voidaan paremmin hallita. Infrastruktuurikoodiin pätevät samat versionhallintajärjestelmien käytännöt kuin perinteiseen ohjelmakoodiin. Se voidaan koostaa, testata ja julkaista CI/CD-prosessien avulla ja näin ollen varmistaa, että kaikki koodissa määritetyt resurssit ovat asennettavissa tai asennettu pilvialustalle ja ne voidaan ottaa käyttöön. (What is infrastructure as code (IaC)? 2022.)

Erityisen hyödyllisenä IaC voidaan nähdä, kun tulee tarpeelliseksi monistaa tuotantoa vastaavia ympäristöjä kehitys- ja testaustarkoituksiin. Tiimit pääsevät testaamaan kehitettyjä sovelluksia varhaisessa vaiheessa ja validoimaan, että ne toimivat myös tuotantoympäristön kaltaisissa suoritusympäristöissä. Tämä on mahdollista deklaratiivisten määrittelytiedostojen avulla. Tiedostoissa on kuvattuna kaikki ympäristön pilvikomponentit ja kokoonpanot sekä niiden asetukset ja keskinäiset riippuvuudet. Tiedostot eivät kuitenkaan sisällä esimerkiksi komponenttien asentamiseen vaadittuja komentosarjoja eli tietoa siitä, miten komponentti asennetaan, jolloin jätetään mahdollisuus käyttää optimoituja asennustekniikoita, tarkoin määriteltyjen ja rajattujen tekniikoiden sijaan. (What is infrastructure as code (IaC)? 2022.)

3.3 Palveluista ja resursseista koostuva infrastruktuuri Microsoft Azuressa

Kaikilla pilvipalveluntarjoajilla pilvi-infrastruktuuri pitää sisällään laitteisto- ja sovelluskomponentteja, jotka yhdessä toimiessaan suorittavat sovellusten vaatimaa laskentaa pilvessä. Infrastruktuuriin kuuluu myös abstraktiokerros, jossa mahdollistetaan resurssien ja palveluiden käyttö erilaisten komentotulkkien tai graafisten käyttöliittymien avulla. Käyttö tapahtuu usein API-rajapintojen kautta. (Montgomery, Marko, Rando 2021). Resurssi- ja palvelutarjonta voivat vaihdella eri pilvipalveluntarjoajien

välillä, mutta niistä löytyy paljon yhtäläisyyksiä. Tässä opinnäytetyössä tarkastellaan Microsoft Azuren resursseja ja palveluita, joista niistäkin käydään läpi vain sellaisia, joita käytetään opinnäytetyön käytännönosuudessa.

3.3.1 Resurssiryhmä

Tiettyyn ratkaisuun sisältyvien Azuren resurssien tulee aina kuulua resurssiryhmään (Resource Group). Yhdessä ratkaisussa voi olla useita resurssiryhmiä, jotka kukin sisältävät samat resurssit, mutta niiden käyttötarkoitus voi vaihdella. Esimerkiksi saman sovellusratkaisun eri ympäristöt voitaisiin jakaa resurssiryhmittäin siten, että kehitys-, testaus- ja tuotantoympäristöjen resurssit olisivat omissa resurssiryhmissään. Resurssiryhmien suunnittelussa tulisi huomioida, kuinka resurssit jaotellaan ryhmiin siten, että niihin kohdistuvat muutokset voidaan tarpeen tullen kohdistaa koko resurssiryhmään. (What is a resource group 2023.)

Resurssiryhmä kuuluu tilaukseen (Subscription). Resurssiryhmää luotaessa määritellään, mihin tilaukseen se kuuluu ja mihin maantieteellisesti sijaitsevaan palvelinkeskukseen se lisätään (What is a resource group 2023). Vaikka resurssiryhmässä olevien resurssien ei tarvitse olla samassa palvelinkeskuksessa, eikä se itsessään suorita mitään tiettyä tehtävää, sijainti on määritettävä, sillä resurssiryhmä sisältää kaikkien siihen liitettyjen resurssien metatiedot ja nämä tiedot tulee säilyttää jonkin alueen palvelinkeskuksen palvelimilla (Brock 2021).

3.3.2 Tiedon varastointi

Sovelluksilla on usein tapauksesta riippumatta tarpeita tiedon varastoimiselle ja siihen tarkoitukseen Azuressa on useampia resurssivaihtoehtoja, mutta yleisesti käyttöön otetaan varastointitili (Storage Account) sekä sen tarjoamat tietojen tallentamisen palvelut, joita ovat säilöt (Container), taulut (Table), verkkolevyn kaltaiset tiedostosäilöt (File Shares) ja viestisäilöt (Queues). Varastointitiliin säilötty tieto on saatavissa HTTP- tai HTTPS-yhteyden avulla. Tieto on aina suojattu salausmenetelmän sekä tietynlaisen avainjärjestelmän avulla, jolloin ulkopuoliset eivät pääse tietoon käsiksi. (Storage account overview 2023.)

Säilöjen määrää varastointitilissä ei ole rajattu ja yksittäinen säilö voi sisältää rajattoman määrän tiedostoja. Yhden säilön sisältämät tiedostot voivat kaikki olla tiedostotyypiltään erilaisia, mutta voi olla

hyvä sisällyttää samaan aiheeseen liittyvät tiedostot yhteen samaan säilöön. (Introduction to Azure Blob Storage 2023.) Varastointitilin taulut tukevat taulukkomuotoisen strukturoidun tiedon säilyttämistä, mutta rakenteellisesta muodosta huolimatta se ei ole SQL-tietokanta, vaan ennemminkin NoSQL-kanta eli se ei noudata relaatiotietomallin rakennetta. Koska taululla ei ole erityistä skeemaa, se voi sisältää erilaisilla ominaisuuksilla varustettuja ilmentymiä. Esimerkiksi sama ominaisuus voi sisältää numero- tai merkkijonotyyppisen arvon. (What is Azure Table storage? 2022.) Kuvasta 1 voidaan havaita, että Value -sarakkeessa on sekä merkkijono- että numeerisia arvoja.

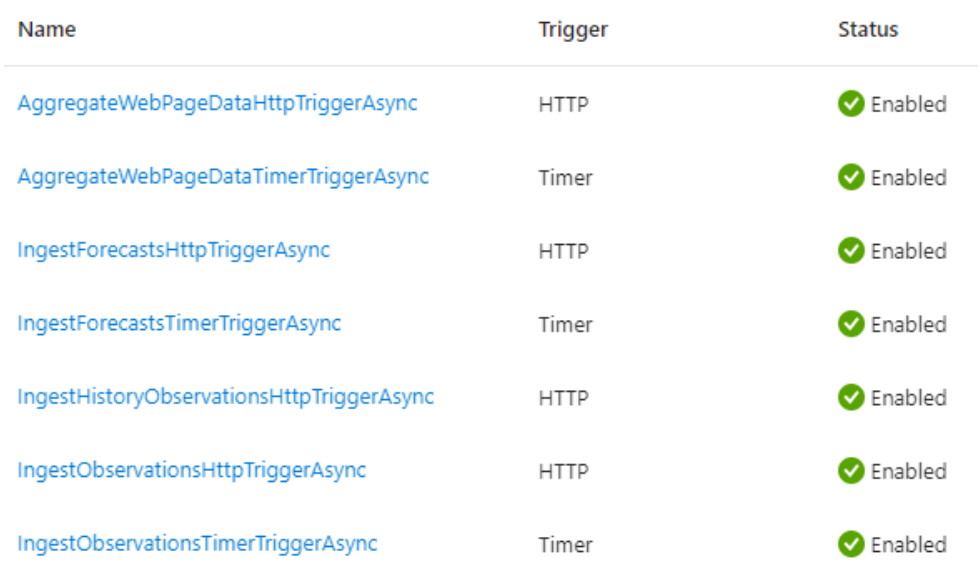
PartitionKey	RowKey	Value	Timestamp
default	fmilocation	Nurmijärvi Röykkä	2023-11-19T20:22:31.564346Z
default	fmisid	101149	2023-11-19T20:22:28.3251806Z
default	slug	fktme	2023-11-19T20:22:24.9660831Z

KUVA 1. Varastointitilin taulun entiteettien ominaisuuksia

3.3.3 Sovelluksen suoritusympäristö

Sovellusten suorittamista varten Azureen luodaan sovelluksen isännöintisuunnitelma (App Service Plan), jonka määrittelyssä huomioidaan siihen asennettavien sovellusten vaatimat laskentaresurssit. Laskentaresurssit kattavat muun muassa prosessointitehon, muistin, tallennustilan ja verkkoresurssit. Azure on laatinut eri tarkoituksiin soveltuvia isännöintisuunnitelman tilaustasoja, joihin on sisällytetty tietty määrä kaikkia edellä mainittuja resursseja. (Azure App Service plan overview 2023.) Consumption- ja Premium-tilaustasot ovat hyvin samankaltaisia, molemmat skaalautuvat dynaamisesti, niistä löytyy sekä Linux- että Windows-käyttöjärjestelmävaihtoehdot ja ne tukevat yleisimpiä kieliä kuten C#, Java, Python (vain Linux), JavaScript ja PowerShell. Eroavaisuudet liittyvät suorituskykyyn, Consumption-tilaustasossa olevat instanssit eivät ole jatkuvasti käynnissä, vaan 10 minuutin käyttökatkon jälkeen instanssi sammuu ja uudelleen käynnistys vie hieman enemmän aikaa. Premiumissa instanssit ovat jatkuvasti käynnissä eikä niin sanottua kylmäkäynnistystä tarvita. Lisäksi Consumptionissa instanssien enimmäissuoritusaika on 10 minuuttia, kun Premiumissa jopa 60 minuutin suoritusaika on mahdollinen. (Gunathilaga 2022.)

Isännöintisuunnitelmaan liitettävä sovellusresurssi voi olla esimerkiksi FaaS-palvelutasoon lukeutuva funktiosovellus (Function App). Yksi isännöintisuunnitelma voi sisältää useita funktiosovelluksia ja yksi funktiosovellus voi sisältää useita erillisiä toimintoja suorittavia funktioita. Nämä funktiot voidaan asettaa suoriutumaan lukuisilla erilaisilla käynnistysmetodeilla. Näitä metodeja ovat esimerkiksi HTTP-kutsu (HTTP-trigger) tai ajastetut suoritukset (Timer Trigger). Kuvassa 2 on funktiosovellus, joka sisältää eri käynnistysmetodeilla olevia yksittäisiä funktioita.



KUVA 2. Malliratkaisun .NET-funktioita

Funktiosovellus on yksi esimerkki palvelimettomasta-arkkitehtuurista (serverless architecture), mikä tarkoittaa, että kehittäjän ei tarvitse huolehtia palvelinten hankinnasta tai niiden ylläpidosta. Palvelinten käyttöä hallitaan dynaamisesti ja palveluntarjoaja skaalaa palvelua automaattisesti käytön mukaan. Tällaiset palvelut ovat hyviä vaihtoehtoja, kun ei olla toteuttamassa kovin kriittisiä ja monimutkaisia sovelluksia, kuten tämän opinnäytetyön yhteydessä toteutettu sovellus. (Microsoft 2023, 736.)

3.3.4 Monitorointi

Pilvialustalla olevien sovellusten ja eri resurssien toimintaa on suositeltavaa seurata ja analysoida erilaisilla mittareilla, jotta niissä ilmeneviin ongelmiin voidaan reagoida ajoissa. Azure-monitori (Azure Monitor) on palvelu, joka valvoo resurssien suorituskykyä keräämällä niiden telemetriatietoja ja analysoimalla niitä. (Azure Monitor overview 2023.) Sovellusseuranta (Application Insights) on osa Azure-

monitori palvelukokonaisuutta ja se tarjoaa kattavaa tietoa sovellusten suorituskyvystä ja käyttäytymisestä reaaliajassa (Application Insights overview 2023). Sovellusseuranta voidaan ottaa käyttöön esimerkiksi funktiosovelluksessa, jolloin se integroidaan automaattisesti sovelluksen koodiin. Tämä instrumentointi mahdollistaa automaattisen telemetriatiedon keräämisen, mukaan lukien suorituskykytiedot, virheet, käyttäjätoiminnot ja muut diagnostiset tiedot. (How to configure monitoring for Azure Functions 2023.)

Sovellusseuranta on integroitu lokityötilaan (Log Analytics Workspace), jonne telemetriatiedot tallentuvat. Kerätty telemetriatieto siirtyy sovellusseurantaan tarkasteltavaksi Azuren portaalin kautta. Sovellusseurannassa voidaan myös mukauttaa, mitä tietoja sovelluksesta halutaan esittää. (Application Insights overview 2023.) Esimerkiksi voi olla hyödyllistä kerätä ja esittää ohjelmakoodin tuottamat lokitiedot, jolloin mahdollisesti ohjelmakoodissa olevan virheen paikannus helpottuu. Tämä edellyttää, että koodiin on lisätty lokitietojen keräys tarvittaviin toimintoihin (KUVA 3).

```
var data = await fmiObservationService.GetDataAsync(fmisid, fromUtc, toUtc);
var count = data.Count;
_logger.LogTrace($"Downloaded {count} entities in {nameof(InternalIngestObservationsAsync)}" +
    $"for location {fmiLocation} {fmisid}. " +
    $"Duration {(DateTime.UtcNow - start).TotalMilliseconds} milliseconds.");
```

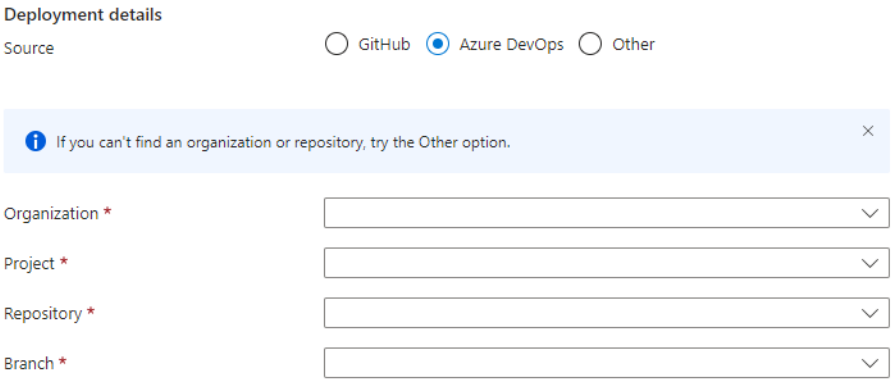
KUVA 3. Malliratkaisuna toteutetun sovelluksen .NET-funktioiden lokitietojen keräys

Azure-monitori kattaa myös hälytysten jakamisen kohderyhmälle. Tällainen häiriöilmoituspalvelu (Alert) otetaan käyttöön luomalla toimintaryhmä (Action Group), jolle määritellään sähköpostiosoite ja/tai puhelinnumero, johon ilmoitukset hälytyksistä lähetetään. Hälytyssääntöön (Alert Rule), määritellään seurattava resurssi, kynnysarvot, tarkastelujakso, toimenpiteet, hälytyksen tärkeysaste sekä ehto, jonka perusteella hälytys laukaistaan. Lisäksi määritellään signaalin lähde, jona voi toimia esimerkiksi lokityötila. (What are Azure Monitor alerts? 2023.)

Azuressa on erilaisia vaihtoehtoja resurssien käytöstä aiheutuvien kustannusten seuraamiseen ja niissä ylitettävien raja-arvojen ilmoittamiseen. Resurssiryhmäkohtainen budjettisääntö (Budget Rule) on yksi näistä vaihtoehdoista. Se seuraa kustannuksia resurssiryhmätasolla, mutta ei esimerkiksi tilauksen tasolla. Budjettisääntöön asetetaan halutut raja-arvot enimmäiskustannukselle, seuranta-ajanjaksolle sekä annetaan prosenttilukuna arvo, jonka ylittyessä ilmoitus kustannusten mahdollisesta ylittymisestä annettuun sähköpostiosoitteeseen lähetetään. (Tutorial: Create and manage budgets 2023.)

3.3.5 Staattinen verkkosovellus

Yksi serverless-arkkitehtuuria edustava palvelu on staattinen verkkosovellus (Static Web App), joka on tarkoitettu nimensä mukaisesti staattisten verkkosovellusten julkaisemiseen. Kun verkkosovellus-resurssi luodaan Azuren portaalissa, tulee määritellä lähde, jossa sen käyttämät lähdekoodit sijaitsevat. Lähteeksi voidaan antaa esimerkiksi versionhallinnan julkaisualustana tunnettu Azure DevOps, jolle määritellään organisaatio, projekti, ohjelmakoodisäilö (repository) sekä haara (branch). Kun määriteltyyn haaraan julkaistaan verkkosovelluksen koodimuutoksia, otetaan ne automaattisesti käyttöön Azuressa. (What is Azure Static Web Apps? 2023.)



KUVA 4. Staattisen verkkosovelluksen luontivaihe Azuren portaalissa

Kuten kuvasta 4 havaitaan, lähdesijaintivaihtoehtona on Azure DevOps-palvelun lisäksi myös GitHub, jonka toimintaperiaate on yhtäläinen DevOpsin kanssa. Vaihtoehto ”Other” valitaan, kun on tarve toteuttaa verkkosovelluksen julkaisu muulla tapaa. Yksi esimerkki on tallentaa verkkosivun muodostamiseen vaaditut HTML-, CSS- ja JavaScript-tiedostot $web-nimiseen säilöön varastointitilissä. Tämä vaatii tiettyjä lisäkonfigurointeja varastointitilissä ja tätä ratkaisua esitelläänkin tarkemmin malliratkaisun läpikäynnin yhteydessä.

3.4 Tietoturva

Tietoturva tulee huomioida sovelluksen elinkaaren jokaisessa vaiheessa ja ottaa käyttöön juuri kyseiselle ratkaisulle hyödyllisiä tietoturvaa lisääviä komponentteja. Microsoft suosittelee erityisesti identi-

teetin hallinnan, infrastruktuurin suojaamisen, sovellusten turvallisuuden, tietojen suvereniteetin ja salauksen sekä muiden turvallisuusresurssien huomioimista Azuren pilvipalvelussa julkaistavissa ratkaisuissa (Microsoft 2023, 4).

Identiteetin tunnistamiseen ja hallintaan suositellaan käytettäväksi Azure Entra -palvelua, jossa voidaan hallita käyttäjien todennusta ja valtuutusta ilman räätälöityjä todennusratkaisuita. Siinä missä on-premises-ympäristöissä luotetaan sisäverkon ja palomuurien tarjoamaan suojaan, pilviympäristössä näiden ei oleteta tarjoavan riittävän hyvää tietoturvatasoa, vaan lisäksi on käytettävä pilvipalveluntarjoajan omia identiteettiratkaisuja. Azuren organisaatiotasolla hyödynnettävä identiteetti ja pääsynhallinta (IAM) -ratkaisu perustuu siihen, että kaikilla käyttäjillä sekä resursseilla on oma identiteetti, jota käytetään todennus- ja valtuutusprosesseissa. Todennusprosessi kohdistuu käyttäjän tiliin ja sillä ohjataan, kuka tai mikä tiliä saa käyttää. Valtuutusprosessissa identiteetille määritetään tarpeelliset oikeudet, joiden mukaisesti identiteetin omistava henkilö tai resurssi saa suorittaa tiettyjä toimenpiteitä sovelluksille. (Identity architecture design 2023.)

Infrastruktuurin suojaamisessa käytetään niin kutsuttua RBAC-mallia (role-based access control), joka on osa kokonaisvaltaisempaa IAM-järjestelmää. Siinä käyttäjälle tai identiteetille myönnetään roolin perusteella oikeuksia resursseihin sekä niihin kohdistuviin toimenpiteisiin. Lisäksi oikeudet voidaan myöntää tilin, resurssiryhmän tai yksittäisen sovelluksen tasolla. (Microsoft 2023, 113.) Käyttäjä voi saada esimerkiksi täydet muokkausoikeudet kaikkiin resurssiryhmään kuuluviin resursseihin, mutta hän ei voi myöntää oikeuksia toisille käyttäjille.

Sovellusten suojaamiseksi Azure suosittelee datan salausta sen siirron yhteydessä. Salauksella tarkoitetaan datan muuttamista sellaiseen muotoon, että vain tietty järjestelmä osaa lukea datan sisällön. Salaus voidaan toteuttaa käyttämällä viimeisintä tuettua TLS-versiota datan salauksessa. (Microsoft 2023, 113.) TLS-versio määritellään uuden varastointilin luonnin yhteydessä.

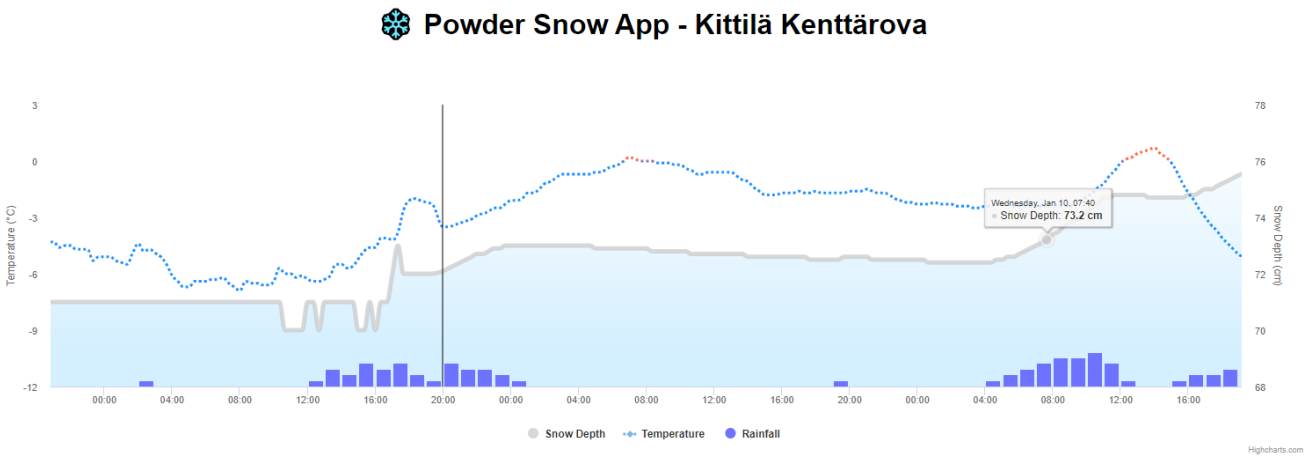
Suositeltavaa on myös suojautua XSS (Cross-Site Scripting) -hyökkäyksiä vastaan, jossa hyökkääjä pyrkii lisäämään käyttäjän selaimessa suoritettavaan verkkosovellukseen haitallista JavaScript -koodia (Synopsys 2023). Toinen yleinen hyökkäysmenetelmä on DDoS (Distributed Denial of Service) eli hajautettu palvelunestohyökkäys, jossa hyökkääjä suorittaa useita samanaikaisia palvelupyyntöjä kohdepalvelimelle saadakseen palvelimen kapasiteetin ylittymään. Tällaisessa tilanteessa loppukäyttäjät eivät voi käyttää palvelua, koska sen kapasiteetti on ylittynyt. (Sullivan 2015.)

Mikäli ohjelmakoodin suorituksessa vaaditaan erilaisia pääsyavaimia, kuten API-avaimia tai Azuren resurssien pääsyavaimia, tulisi ne tallettaa salaisuussäilöön (Azure Key Vault) (Rastogi 2023). Avaimia ei tulisi koskaan kirjoittaa suoraan ohjelmakoodiin, sillä silloin ne voivat päätyä ulkopuolisten luettaviksi versionhallinnan julkaisualustassa. Mikäli hyökkääjä saisi haltuunsa esimerkiksi varastointitilin salaamiseen tarkoitetun yhteysmerkkijonon (Connection String) sekä funktiosovelluksen HTTP-kutsuun tarvittavan osoitteen, olisi hänellä mahdollisuus kirjoittaa ja lukea sensitiivistä tietoa varastointitilin tietovarastoista. Nykyään Microsoft suosittelee käyttämään tietoturvallisempana vaihtoehtona hallitun identiteetin palvelua (Managed Identity) pääsyavainten tai yhteysmerkkijonojen sijasta (Microsoft 2023, 858). Hallitun identiteetin palvelu on osa aiemmin käsiteltyä IAM-järjestelmää.

Kallisarvoinen tieto tulee suojata tahalliselta ja tahattomalta hävittämiseltä. Varastointitilille asetetaan erityinen redundanssitaso sen luonnin yhteydessä. Tasoja on useita ja ne viittaavat siihen, miten tieto on hajautettu eri Azuren palvelinkeskuksiin. Esimerkiksi paikallisesti redundantti varasto (LRS) sisältää perussuojaustason, jossa varastointitilistä muodostetaan kolme kopioita yhden fyysisen palvelinkeskuksen (zone) sisälle yhteen alueeseen (location). Mikäli tähän kyseiseen fyysiseen palvelinkeskukseen kohdistuisi jokin ennalta arvaamaton onnettomuus, kuten tulipalo, kaikki tiedot menetettäisiin. Jos vaaditaan hieman parempaa tiedon säilymiskykyä, voidaan valita maantieteellisesti redundantti varasto (GRS), jossa varastointitili edelleen kopioidaan kolme kertaa saman palvelinkeskuksen sisällä, kuten LRS-redundanssilla, mutta lisäksi sen tiedot replikoidaan kokonaan sekundäärisen alueen palvelinkeskukseen, jossa tiedot tallennetaan jälleen LRS-redundanssilla. Näin samasta tiedosta on yhteensä kuusi kopioita hajautettuna kahteen eri maantieteellisellä alueella sijaitsevaan palvelinkeskukseen. Mikäli jompikumpi näistä palvelinkeskuksista tuhoutuu, tieto säilyy toisessa keskuksessa. (Azure Storage redundancy 2023.)

4 MALLIRATKAISUN SUUNNITTELU JA TOTEUTUS

Malliratkaisun toteutus alkoi tämän opinnäytetyön varsinaisen aiheen ulkopuolelle jätetyn koneoppimissovelluksen suunnittelulla sekä toteutuksella. Jotta pääaihetta voitaisiin konkreettisemmin käsitellä ja havainnollistaa, esitellään itse sovellus pääpiirteittäin. Sovellus muodostaa koneoppimisen avulla Ilmatieteenlaitoksen säätietoaineistoon pohjautuen lumen määrän ennusteen tulevalle kahdelle vuorokaudelle valitulla havaintoasemalla. Havaintoaseman valinta tapahtuu sovelluksen julkaisuvaiheessa valitsemalla ennalta määrittämistäni havaintoasemista yksi. Julkaisuprosessin jälkeen toimivassa sovelluksessa ennuste päivittyy 20 minuutin välein Ilmatieteenlaitoksen sääennusteen sekä havaintohistorian päivitysaikataulun mukaisesti. Ennuste esitetään graafin muodossa verkkosovelluksessa alla olevan esimerkkikuvan 5 mukaisesti.



KUVA 5. Opinnäytetyössä toteutetun staattisen verkkosovelluksen näkymä

Laadukkaan koneoppimismallin tuottamisessa tuli huomioida useita mallin laatuun vaikuttavia seikkoja. Tietoaineistosta täytyi osata poimia juuri ne tekijät, jotka yleisesti vaikuttavat lumen määrän lisääntymiseen, painumiseen ja sulamiseen. Aiheeseen liittyviä tutkimuksia tai malliesimerkkejä oli verrattain vähän löydettävissä, mutta vuonna 2022 julkaistussa raportissa koneoppimista hyödynnettiin lumenkerääntymisen ennustamisessa Etelä-Koreassa, missä raskaiden lumisateiden oli todettu aiheuttavan mittavia vahinkoja (Song, Yun, Lee & Yum 2022, 1). Raportti toimi tässä koneoppimismallin toteuttamisen tukena. Koska toteutettu sovellus toimii tässä työssä vain välineenä, jolla voin havainnollistaa automaatiota ja IaC-menetelmän toimivuutta sovelluskehityksessä, en käy sovellusta kovinkaan syvällisesti läpi, vaan keskityn pääasiallisesti sovelluksen automaattiseen julkaisuprosessiin.

Tavoitteena oli hyödyntää automaatiota mahdollisimman kattavasti kaikissa sovelluksen kehittämiseen ja julkaisuun liittyvissä vaiheissa. Lisäksi tarkoituksena oli kehittää koneoppimista hyödyntävän verkkosovelluksen vaatima kokonainen pilvi-infrastruktuuri sekä siihen liittyvät ohjelmakoodit tietoturvallisesti ja julkaista ne yhdestä paikasta vähäisillä käyttäjän toimenpiteillä. Julkaisuprosessi perustuu täysin IaC-mallin sekä GitHub-versionhallintapalvelun CI/CD-työnkulkujen kehittämiseen.

IaC-mallin hyödyntäminen on perusteltua, sillä jopa tällaisen kohtuullisen pieneen ja yksinkertaiseen sovellukseen tarvitaan monia erilaisia pilviresursseja, joille kaikille on omat konfiguraatiomäärityksensä. Resurssit ovat useimmiten jollain tapaa kytköksissä toisiinsa ja yhdessä resurssissa tapahtuvat muutokset voivat aiheuttaa muutostarpeita myös toiseen resurssiin. Yhtä sovellusta palveleva pilvi-infrastruktuuri voikin kokonaiskuvassa tarkasteltuna vaikuttaa hyvin moniulotteiselta ja vaikeasti hallittavissa olevalta kokonaisuudelta. Usein kuitenkin sama pilvi-infrastruktuuri voi palvella useita sovelluksia, joilla voi lisäksi olla riippuvuuksia toisistaan. Ilman IaC-mallia tällaisen kokonaisuuden ylläpitäminen ja hallinta voisi olla hyvinkin kompleksista ja virhealtista.

4.1 Teknologia- ja työkaluvalinnat

Valitsin sovelluksen toteuttamiseen nykyaikaisia yleisesti käytössä olevia teknologioita ja työkaluja. Pilvipalveluntarjoajaksi valitsin Microsoft Azuren pilvialustan, jonka vuoksi ratkaisuun valikoitui myös muita Microsoftin kehittämiä tai omistamia teknologioita, työkaluja ja järjestelmiä. Opinnäytetyön rikastamiseksi sekä oman osaamisen syventämiseksi toteutin esimerkiksi koneoppimisosuuden Python-kielellä, minkä vuoksi funktiosovellusten laskentakapasiteetin käyttöjärjestelmäksi valikoitui Linux. Python oli soveltuva koneoppimisosuuteen myös, koska sen laajat koneoppimiskirjastot tekevät koneoppimisen mallien kehittämisestä suhteellisen helppoa.

Versionhallinnan julkaisualustan ja CI/CD-työnkulkujen valinnassa vaihtoehtoina olivat Azuren DevOps ja GitHub, joista kuitenkin Microsoftin vastikään ostama GitHub oli tähän ratkaisuun parempi valinta. Azure DevOps sisältää sellaisiakin ominaisuuksia, joita ei tämän kokoluokan toteutuksen kehityksessä tarvittaisi lainkaan, sillä se on tarkoitettu ennemminkin suurempien organisaatioiden ja projektikokonaisuuksien hallintaan.

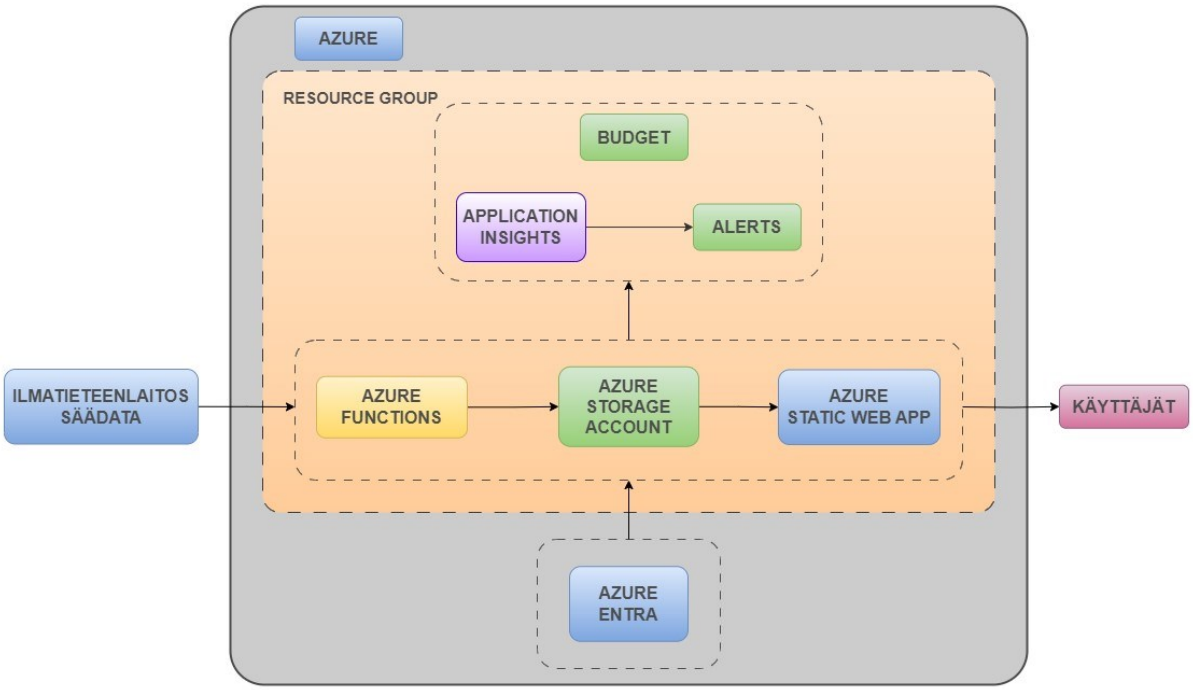
Jotta en olisi täysin ajautunut Microsoft-siiloon, valitsin IaC-työkaluksi HashiCorpin kehittämän Terraformin. Terraform on niin sanottu monipilvi-infrastruktuurityökalu, kun taas esimerkiksi toisena

vaihtoehtona pohtimani Bicep on optimoitu erityisesti vain Azuren infrastruktuurin hallintaan. Lisäksi Terraformilla on laaja käyttäjäkunta, kattava dokumentaatio sekä valmiit moduulit Azuren resursseille, jolloin sen opettelu oli suhteellisen helppoa.

Tietoaineistojen noutoon, esikäsittelyyn, tallentamiseen sekä koostamiseen käytin C#-kielen .NET-funktioita. Ratkaisun näkyvä osuus, eli verkkosovellus koostuu HTML- ja CSS-tiedostoista sekä JavaScript-koodista. GitHubin työnkulkujen kehityksessä käytin YAML-merkkausta. Valittujen kielien ja tekniikoiden ja Microsoft-vaikutusten vuoksi käytettyjä koodinkehitystyökaluja olivat Visual Studio Code ja Visual Studio. Aikaisemman kokemuksen perusteella koin Visual Studion ketterämmäksi .NET-funktioiden koodaamisessa, mutta muiden kielien ja tekniikoiden osalta Visual Studio Code antoi paremman tuen esimerkiksi selkeämmän koodin värivisuallisoinnin ja syntaksin korostuksen myötä. Koodimuutosten siirtämiseen versionhallintaan käytin Visual Studio Coden Git -laajennosta.

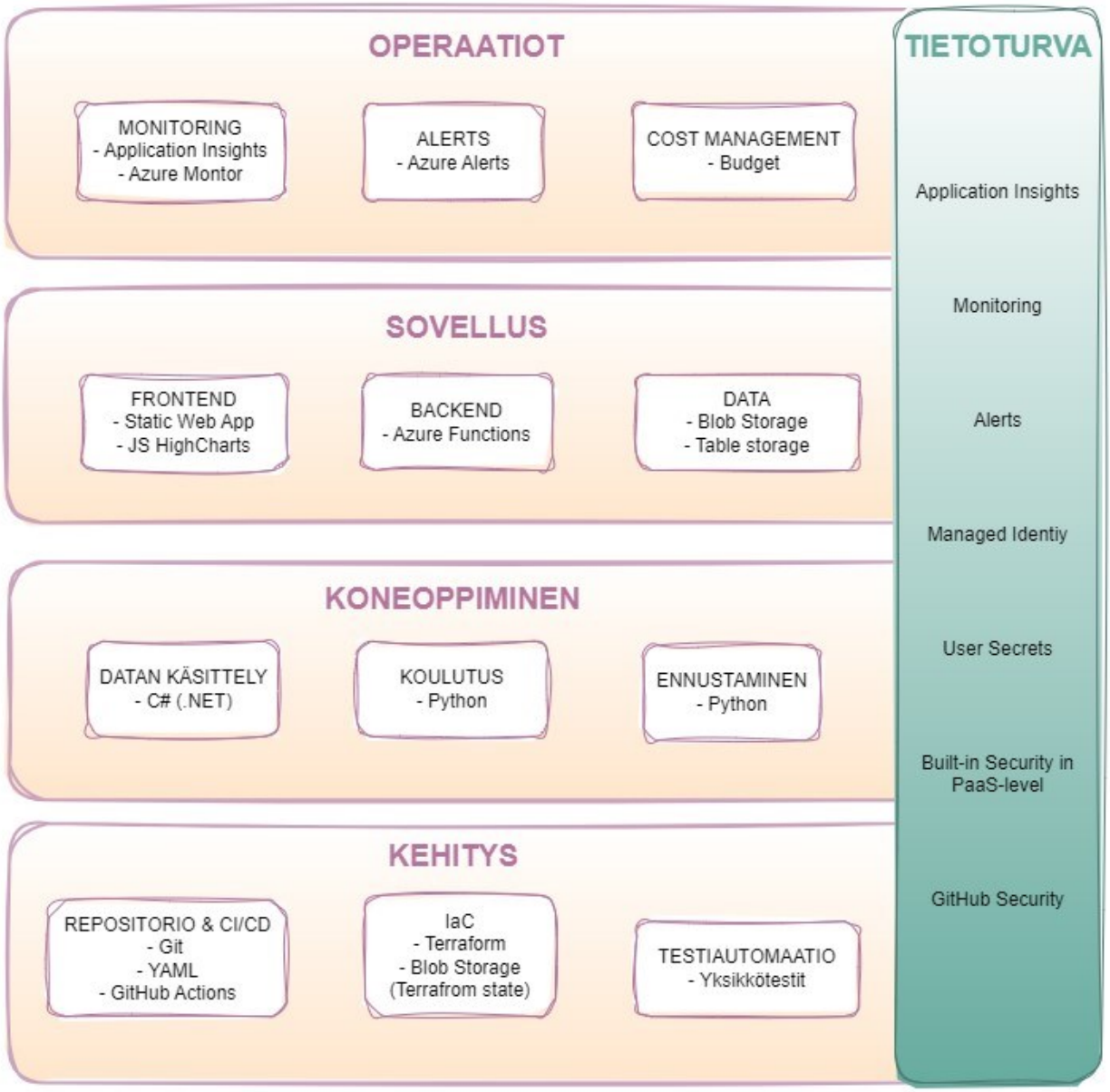
4.2 Ratkaisun kokonaisarkkitehtuuri ja tietovirrat

Sovelluksen suunnittelussa pyrin rakentamaan selkeän arkkitehtuurin sekä minimoimaan eri komponenttityyppien määrän. Komponenttien valintaa ohjasivat sovelluksen tarpeet, budjetti sekä aikaisempi käyttökokemukseni valituista komponenteista. Lisäksi pyrin välttämään sovelluksen kokoon suhteutettuna turhan raskaiden tai kalliiden palvelujen ja resurssien käyttöönottoa. Olisin esimerkiksi voinut ottaa käyttöön Azuren API-hallinta (API Management) -palvelun, jonka avulla olisin voinut vahvistaa sovelluksen tietoturvaa, mutta se olisi kasvattanut sovelluksen kustannuksia tuntuvasti. Kuvassa 6 on esitetty pääpiirteittäin sovellukseen valitut komponentit sekä komponenttien väliset tietovirrat.



KUVA 6. Opinnäytetyössä toteutetun sovelluksen komponenttien väliset tietovirrat

Sovelluksen kokonaisratkaisua suunniteltaessa havaitsin tietynlaisia elinkaarellisia osa-alueita, joihin ratkaisuun valitut teknologiat, työkalut ja komponentit olivat jaoteltavissa (KUVA 7). Ratkaisuun kuuluu operaatiokerros, johon sisältyvät valmiin, pilvialustalle julkaistun sovelluksen monitorointiin osallistuvat komponentit. Sovelluskerroksen komponentit suorittavat tietoaineistojen noudon Ilmatieteenlaitoksen palvelusta, niiden tallentamisen tietovarastoihin, tiedon lukemisen varastoista sekä verkkosovelluksen näkymän muodostamisen. Koneoppimiskerroksen komponentit osallistuvat koneoppimismallin sekä sen avulla muodostettavan ennusteen tuottamiseen. Kehittämiskerrokseen sisällytetyt komponentit ovat työkaluja, joilla sovellusta voidaan kehittää ja testata ja joiden avulla se voidaan julkaista haluttuun ympäristöön CI/CD-prosessin mukaisesti. Tietoturva on huomioitu osa-alueittain muun muassa sisällyttämällä ratkaisuun Azuren valmiita tietoturvallisuutta parantavia komponentteja, hyödyntämällä kehitystyökalujen sisäänrakennettuja turvallisuusmekanismeja sekä huolehtimalla salaisuuksien oikeaoppisesta tallettamisesta.

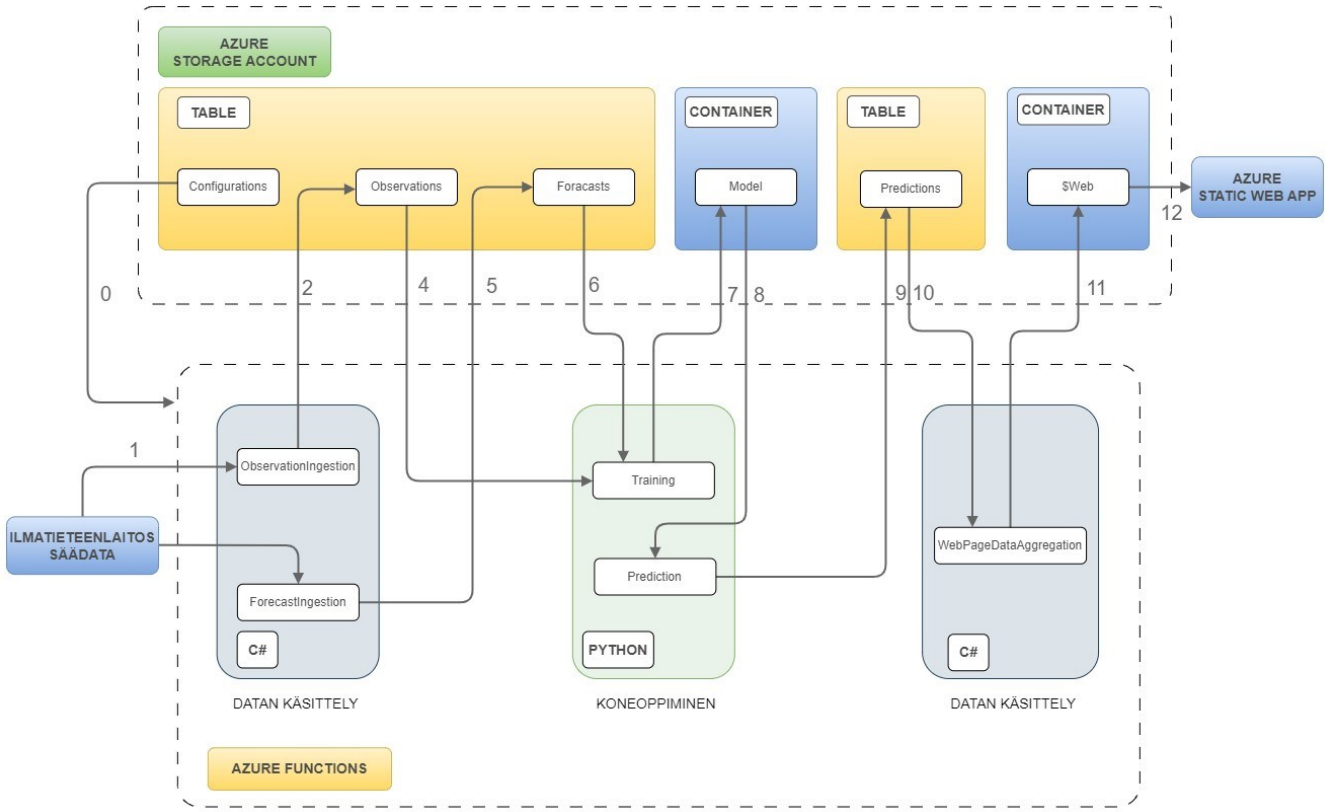


KUVA 7. Opinnäytetyössä toteutetun sovelluksen kokonaisratkaisun osa-alueiden jaottelu

4.3 Tietoaineistojen nouto, tallennus ja käyttö

Julkaistussa toimivassa sovelluksessa funktiot keräävät ja käsittelevät Ilmatieteenlaitoksen tuottamaa sään havainto- ja ennustetietoa jatkuvasti 20 minuutin välein. Tieto on saatavilla koneluettavassa muodossa Ilmatieteenlaitoksen avoimen verkkopalvelun kautta. Molemmille noudettaville tietoaineistotyypille muodostetaan jokaisella funktion suorituskerralla uusi verkko-osoite, jonka pohjaksi asetetaan ”https://opendata.fmi.fi/”, tähän lisätään tarvittavat URL-kyselyparametrit, kuten haluttu aikaväli, jolta dataa halutaan noutaa. Verkkopalvelun palauttama data käsitellään funktiossa sellaiseen muotoon, että

se voidaan tallentaa varastointitilin tauluun, josta työnkulun seuraavat funktiot hakevat datan ja käsittelevät sitä kulloisenkin työnkulun vaiheen mukaisesti.



KUVA 8. Opinnäytetyössä toteutetun sovelluksen säädatan käsittelyjärjestys

Kuvassa 8 on esitetty, kuinka tietoa kuljetetaan Ilmatieteenlaitoksen verkkopalvelusta aina käyttäjälle esitettävään verkkosovellukseen saakka. Tiedon käsittelijöinä toimivien funktioiden suoritusjärjestys on esitetty numeroin ja nuolen avulla kuvataan liikkuvan tiedon suunta. Havaintoaineisto noudetaan *ObservationIngestion*-funktiolla ja sääennusteet puolestaan *ForecastIngestion*-funktiolla. Ne esikäsittelevät aineiston ja tallentavat sen vastaavasti nimettyihin varastointitilin tauluihin. Seuraavassa vaiheessa *Training*-funktio lukee molemmissa tauluissa saatavissa olevan aineiston ja kouluttaa koneoppimismallin datan perusteella, malli tallennetaan *Model*-nimiseen säiliöön. Seuraavaksi *Prediction*-funktio muodostaa ennustedatan ja tallentaa sen tauluun, josta *WebPageDataAggregation*-funktio käy lukemassa datan ja muodostaa niistä tiedoston, jonka sisällöstä verkkosovellus kykenee muodostamaan näkymän sivustolle. Funktioiden suoritusjärjestys sekä niiden riippuvuudet eri resursseista tuli huomioida sekä IaC-mallin toteutuksessa, että sovelluksen julkaisuprosessin CI/CD-työnkulussa, joissa funktiot suoritetaan oikeassa järjestyksessä sen jälkeen, kun ne ovat ensin julkaistu Azureen.

4.4 Infrastruktuuri koodina toteutus

Konfiguroin kaikki luotavat resurssit IaC-mallin mukaisesti Terraform -tiedostoissa, joissa annoin niille resurssikohtaiset määritykset sekä riippuvuudet muista resursseista jonkin tietyn ominaisuuden perusteella. Esimerkiksi kaikki resurssit ovat riippuvaisia resurssiryhmästä ja niiden määrittelyissä resurssiryhmään viitataan ryhmän nimen perusteella. Jaottelin resurssit niiden toiminnallisuuden perusteella moduuleihin, jolloin muodostui moduulit *storage, function, monitor* ja *web*. Moduulit ovat Terraformissa tapa pakata ja käyttää resursseja uudelleen ja tässä työssä pyrinkin rakentamaan moduulit siten, että ne olisivat helposti uudelleenkäytettäviä myös muissakin projektissa. Terraformissa hakemistorakenne koostuu päätasosta *Infra* sekä sen alla olevasta *modules*-hakemistosta, joka sisältää omat hakemistonsa ratkaisukohtaisille moduuleille. Kaaviossa 1 on esitetty toteuttamaani ratkaisuun sopiva hakemistomalli.

```
Infra
|
|-- modules
|    |
|    |-- function
|    |    |-- main.tf
|    |    |-- variables.tf
|    |    |-- outputs.tf
|    |
|    |-- monitor
|    |    |-- main.tf
|    |    |-- variables.tf
|    |    |-- outputs.tf
|    |
|    |-- storage
|    |    |-- main.tf
|    |    |-- variables.tf
|    |    |-- outputs.tf
|    |
|    |-- web
|    |    |-- main.tf
|    |    |-- variables.tf
|    |    |-- outputs.tf
|
|-- tests
|    |-- function_module_validation.tftest.hcl
|    |-- main_modules_validation.tftest.hcl
|    |-- monitor_module_validation.tftest.hcl
|    |-- storage_module_validation.tftest.hcl
|    |-- web_module_validation.tftest.hcl
|
|-- main.tf
```

```
|-- outputs.tf
|-- providers.tf
|-- variables.tf
```

KAAVIO 1. Terraform koodien hakemistorakenne

Kuten kaaviosta havaitaan, jokainen moduuli sisältää *main.tf*, *variables.tf* ja *outputs.tf* -tiedostot, päätasolla näiden lisäksi on myös *providers.tf*-tiedosto. Nimensä mukaisesti *main.tf* sisältää Terraformin pääohjelman, joka määrittää pilvi-infrastruktuurin resurssit sekä niiden riippuvuudet toisistaan. Mikäli resurssi on riippuvainen toisessa moduulissa esitellystä resurssista, voidaan liitettävän resurssin tunnisteet jakaa kyseisen moduulin *outputs.tf*-tiedostossa, jolloin resurssi muuttuu ikään kuin julkiseksi.

Kaikki muuttujina käytettävät parametrit lisätään *variables.tf*-tiedostoihin. Päätason *variables.tf*-tiedosto sisältää tässä malliratkaisun tapauksessa muuttujia, jotka alustetaan Terraformin ulkopuolella GitHubin työnkuluissa ja siirretään sieltä Terraformin käytettäviksi. Moduulien *variables.tf*-tiedostot sisältävät muuttujina luettavien parametrien esittelyn, jolloin niiden arvot tulee antaa joko moduulitason tai päätason *main.tf*-tiedostossa. Moduulitasolla kovakoodatut muuttujien arvot eivät tukisi uudelleenkäytettävyyttä, joten rakensin Terraform-koodin siten, että arvot määritettiin pääosin päätasolla. Kuitenkin joissain tapauksissa halusin muuttujan arvon pysyvän aina samana, esimerkiksi määrittelemässäni *function*-moduulissa luodaan vain Linux-pohjainen isännöintisuunnitelma, jolloin asetin käyttöjärjestelmäksi Linuxin kovakoodaamalla sen suoraan moduulitason *main.tf*-tiedostoon.

4.4.1 Storage-moduuli

Storage-moduuli sisältää tiedon varastointiin liittyvien Azure-resurssien määrittelyt. Se sisältää varastointitilin resurssin sekä siinä käyttöönotettavat säilö- ja taulu. Koska ratkaisussa oli tarkoituksena luoda vain yksi varastointitili, sen tuli palvella myös staattisen verkkosovelluksen vaatimuksia. Asetin varastointitilille Terraformissa säännöt, joilla mahdollistin staattisen verkkosovelluksen käyttää varastointitilissä olevan *$web*-nimisen säilön sisältämiä HTML- ja CSS-tiedostoja sekä JavaScript-koodia verkkosovelluksen näkymän muodostamiseksi. Näitä turvallisuusteenkin liittyviä sääntöjä kutsutaan nimellä CORS-säännöt (Cross-Origin Resource Sharing) ja ne määrittävät verkkoselaimelle, mitkä verkkotunnukset ovat sallittu käyttämään varastointitilin tiedostoja. Toinen tärkeä turvallisuuteen liittyvä konfiguraatio varastointilin osalta on sen käyttöoikeuksien rajoittaminen. Tässä ratkaisussa olen

asettanut käyttöoikeudeksi yksityinen, mikä tarkoittaa, että kaikille resursseille, jotka käyttävät varastointitilin tietovarastoja, on erikseen annettava luku- ja kirjoitusoikeudet varastointiliin.

Koska ratkaisussa tarvitaan useita varastointitilin tauluja, olen käyttänyt Terraformissa *count*-muuttujaa, jolla voidaan välttää toisteisuutta koodissa. *Count*-muuttuja määrittää, kuinka monta kertaa resurssi luodaan. Arvo perustuu pääkonfiguraatiossa määriteltyyn taulu-nimilistan pituuteen. Jokainen listan alkio vastaa yhtä taulua, ja Terraform luo näin ollen dynaamisesti tarvittavan määrän tauluja halutuilla nimillä.

```
resource "azurerm_storage_table" "st_table" {
  count                = length(var.st_table_names)
  name                 = var.st_table_names[count.index]
  storage_account_name = azurerm_storage_account.st.name
}
```

KUVA 9. Count-muuttujan käyttö Terraformissa

4.4.2 Function-moduuli

Function-moduulissa määritellään sovelluksen isännöintisuunnitelma sekä siihen lisättävät funktiosovellukset. Kun isännöintisuunnitelma luodaan, sille tulee määritellä käyttöjärjestelmä, jossa funktiot isännöidään. Ratkaisussa määräävänä tekijänä olivat Python-funktiot, joita Microsoft suosittelee isännöitäväksi vain Linux-käyttöjärjestelmässä. Tällä perusteella valitsin käyttöjärjestelmäksi jo moduulitasolla Linuxin, jolloin sitä ei voi enää päätason *main.tf*-tiedostossa määrittää ja käyttöjärjestelmä on ikään kuin aina oletusarvoisesti asetettu. Isännöintisuunnitelmassa tulee määritellä kyseisen suunnitelman tilaustaso ja koska olin liittämässä siihen vain funktiosovelluksia, asetin *function*-moduulitasolla tilaustasoon viittavan parametrin *sku_name* arvoon *Y1*, jolla tarkoitetaan funktiosovelluksille soveltuvaa Consumption-tilaustasoa.
Kun olin määritellyt isännöintisuunnitelman, pystyin viittaamaan siihen funktiosovelluksen määrittelyssä. Ratkaisussa on kaksi funktiosovellusta, omansa .NET- ja Python-funktioille, joten minun tuli mahdollistaa useiden funktiosovellusten luonti yhtä moduulia käyttäen. Käytin tämän vaatimuksen toteutumiseen Terraformin metodia *for_each*, joka käy läpi kaikki päätason *main.tf*-tiedostossa luetellut funktiosovellus-esiintymät ja asettaa muuttujien arvot esiintymäkohtaisesti. Kuvassa 10 vasemmalla puolella on esitetty kahden funktiosovelluksen konfiguraatiot päätasolla ja oikealla puolella näiden resurssien läpikäynti *for_each*-metodin avulla *function*-moduulissa.

```hcl
function_apps = [
  {
    func_name                  = "${var.slug}-dotnet-func"
    FUNCTIONS_WORKER_RUNTIME   = "dotnet-isolated"
    FUNCTIONS_EXTENSION_VERSION = "~4"
    dotnet_version             = "8.0"
    python_version             = null
  },
  {
    func_name                  = "${var.slug}-python-func"
    FUNCTIONS_WORKER_RUNTIME   = "python"
    FUNCTIONS_EXTENSION_VERSION = "~4"
    dotnet_version             = null
    python_version             = "3.11"
  }
]
```

```hcl
resource "azurerm_linux_function_app" "func" {
  resource_group_name        = var.rg_name
  location                   = var.rg_location
  storage_account_name       = var.st_name
  storage_account_access_key = var.st_access_key
  service_plan_id            = azurerm_service_plan.asp.id

  for_each = { for idx, func in var.function_apps : idx => func }
  name     = each.value.func_name
```

KUVA 10. For Each-rakenne Terraformissa

Funktiosovellukselle tulee määritellä myös muita konfiguraatioita, jotka tarkemmin ottaen ovat sovelluksen asetuksia. Azuren portaalissa funktiosovelluksen Configuration-valikossa nämä asetukset ovat nähtävillä ja muokattavissa käsin ja niitä voi tarpeen tullen lisätä tai päivittää. On kuitenkin yleisesti ottaen erittäin suositeltavaa pitää konfiguraatioiden määrittelyt IaC-mallin mukaisesti versionhallinnassa koodissa, sillä Azuren portaalissa käsin tehdyt muutokset ylikirjoittavat aiemman asetuksen eikä asetuksen muutos jää mihinkään talteen. Mikäli konfiguraatiota ei muista ulkoa, voi virhetilanteen sattuessa olla vaikeaa saada palautettua edellistä toimivaa versiota. Kuvassa 11 on esimerkki funktiosovelluksen vaatimista konfiguraatiosta Azuren portaalissa.

Name	Value
APPINSIGHTS_INSTRUMENTATIONKEY	08efbc24-4380-40a
AzureWebJobsDashboard	DefaultEndpointsPr
AzureWebJobsStorage	DefaultEndpointsPr
DATAST_BLOB_CONNECTION_STRING	https://qwijydatast.
DATAST_TABLE_CONNECTION_STRING	https://qwijydatast.
FUNCTIONS_EXTENSION_VERSION	~4
FUNCTIONS_WORKER_RUNTIME	dotnet-isolated
WEBSITE_CONTENTAZUREFILECONNECTION!	DefaultEndpointsPr
WEBSITE_CONTENTSHARE	qwijy-dotnet-func-
WEBSITE_USE_PLACEHOLDER_DOTNETISOLA	1

KUVA 11. Funktiosovelluksen konfiguraatiot Azuren portaalissa

IaC-mallin periaatteiden mukaisesti määrittelin nämä samat konfiguraatiot Terraformissa funktiosovellukselle kuten kuvassa 12 on esitetty.

```
app_settings = {
  "FUNCTIONS_EXTENSION_VERSION"            = each.value.FUNCTIONS_EXTENSION_VERSION
  "FUNCTIONS_WORKER_RUNTIME"               = each.value.FUNCTIONS_WORKER_RUNTIME
  "WEBSITE_USE_PLACEHOLDER_DOTNETISOLATED" = each.value.FUNCTIONS_WORKER_RUNTIME == "dotnet-isolated" ? "1" : null
  "APPINSIGHTS_INSTRUMENTATIONKEY"         = var.APPINSIGHTS_INSTRUMENTATIONKEY
  "DATAST_BLOB_CONNECTION_STRING"          = "https://${var.st_name}.blob.core.windows.net"
  "DATAST_TABLE_CONNECTION_STRING"         = "https://${var.st_name}.table.core.windows.net"
}
```

KUVA 12. Funktiosovelluksen konfiguraatiot Terraformissa

Konfiguraatioiden lisäksi funktiosovellus tarvitsee asetuksia liittyen sovelluksen kokoonpanoon. Terraformissa ne määritellään *application_stack*-lohkossa. Kuvasta 13 huomataan, että parametri *use_dotnet_isolated_runtime* asetetaan vain siinä tapauksessa, mikäli kuvassa 12 esiintyvä *FUNCTIONS_WORKER_RUNTIME*-parametrin arvoksi on asetettu *dotnet-isolated*. Parametrit *dotnet_version* ja *python_version* määritellään päätason *main.tf*-tiedostossa sen mukaan, kumpaa versiota on tarkoitus käyttää. Lisäksi nykyaikaisiin tietoturvamenetelmiin perustuen funktiosovellus asetetaan käyttämään kappaleessa 4.5.4 läpikäytävää hallittua identiteettiä (Managed Identity) *identity*-lohkon *type*-parametrin avulla.

```
site_config {
  application_stack {
    dotnet_version              = each.value.dotnet_version
    use_dotnet_isolated_runtime = each.value.FUNCTIONS_WORKER_RUNTIME == "dotnet-isolated" ? true : null
    python_version              = each.value.python_version
  }
}

identity {
  type = "SystemAssigned"
}

}
```

KUVA 13. Funktiosovelluksen määrityksiä Terraformissa

4.4.3 Web-moduuli

Muista moduuleista poiketen *web*-moduulissa määritellään vain yksi resurssi. Tämä ei ole suositeltava tapa Terraformissa, mutta tein tässä poikkeuksen, jotta toteutustyyli säilyi samankaltaisena Terraform -koodissa. Lisäsin *web*-moduuliin vain staattisen verkkosovelluksen resurssin. Kun kyseinen resurssi luodaan, sille muodostuu verkkosivun osoite, joka tulee asettaa varastointitilin CORS-sääntöihin sallituksi verkkotunnukseksi (KUVA 14). Ilman tätä konfiguraatiota, verkkosivun esittäminen ei olisi malliratkaisussa mahdollista. Staattiseen verkkosovellukseen liittyen tulee määritellä myös muita CORS-

sääntöjä, kuten HTTP-kutsun sallitut otsikkotiedot, sallitut kutsumetodit, HTTP-paluusanoman sallitut otsikkotiedot sekä määritellään, kuinka kauan esitarkistuksen OPTIONS-kutsun CORS-asetuksia pidetään välimuistissa.

```
 8    cors_allowed_headers     = ["*"]
 9    cors_allowed_methods     = ["GET"]
10    cors_allowed_origins     = ["https://${module.StaticWebApp.stapp_endpoint}"]
11    cors_exposed_headers     = ["*"]
12    cors_max_age_seconds     = 3600
```

KUVA 14. Varastointitilin CORS-sääntöjen asetus Terraformissa

4.4.4 Monitor-moduuli

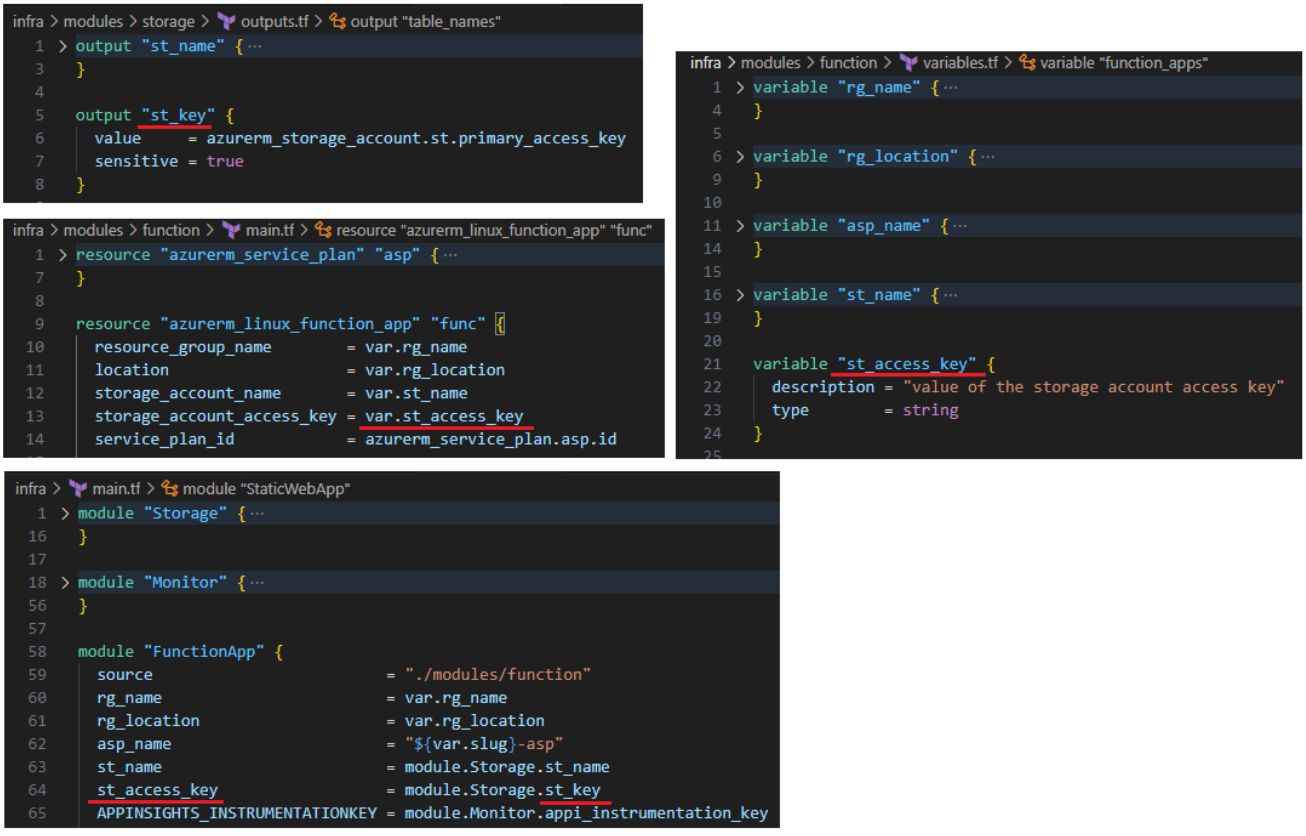
Monitor-moduulissa määritellään kaikki sovelluksen valvontaan ja kustannusten seuraamiseen liittyvät resurssit. Ensimmäiseksi määrittelin lokityötilan, jolle annoin jo moduulitasolla *sku*- (Stock Keeping Unit) sekä *retention_in_days*-parametrien arvot. Koska sovellusseuranta-resurssin *Classic Resource*-tila poistettiin käytöstä helmikuussa 2024, käytin resurssin määrittelyssä *Workspace-based*-tilaa. Tämä tarkoittaa, että sovellusseuranta kerää metriikkatietoja lokityötilaan.

Monitorointiresurssien havaitsemista poikkeuksista on hyvä kyetä ilmoittamaan taholle, jolle poikkeamien käsittely valtuutetaan. Tätä varten loin toimintaryhmän (Action Group), jonka tiedot lisätään metriikkahälytys-resurssille (Metric Alerts). Hälytystyyppejä voi olla useita erilaisia ja tässä ratkaisussa mahdollistin usean metriikkahälytystyypin luomisen yhdellä moduulilla. Tässä käytin jälleen Terraformin metodia *for_each*, joka käy päätasolla määritetyt metriikkahälytys-esiintymät ja niiden muuttujat yksitellen läpi.

Resurssiryhmäkohtaisten kustannusten seurantaan (Budgets) lisäsin kustannusten rajauksen, joka poikkeaa metriikkahälytyksistä. Toimintaryhmän määrityksen sijaan sille määritellään muuttuja *contact_emails*, johon asetetaan kustannusten seuraamisesta vastaavan tahon sähköpostiosoite. Malliratkaisun tapauksessa sähköpostiosoite on dynaaminen ja muuttuu sen perusteella, kenen käyttäjän Azure tiliin resursseja ollaan perustamassa. Käyttäjän sähköpostiosoite noudetaan Azuresta GitHubin työnkuluissa ja luetaan Terraformiin muuttujana.

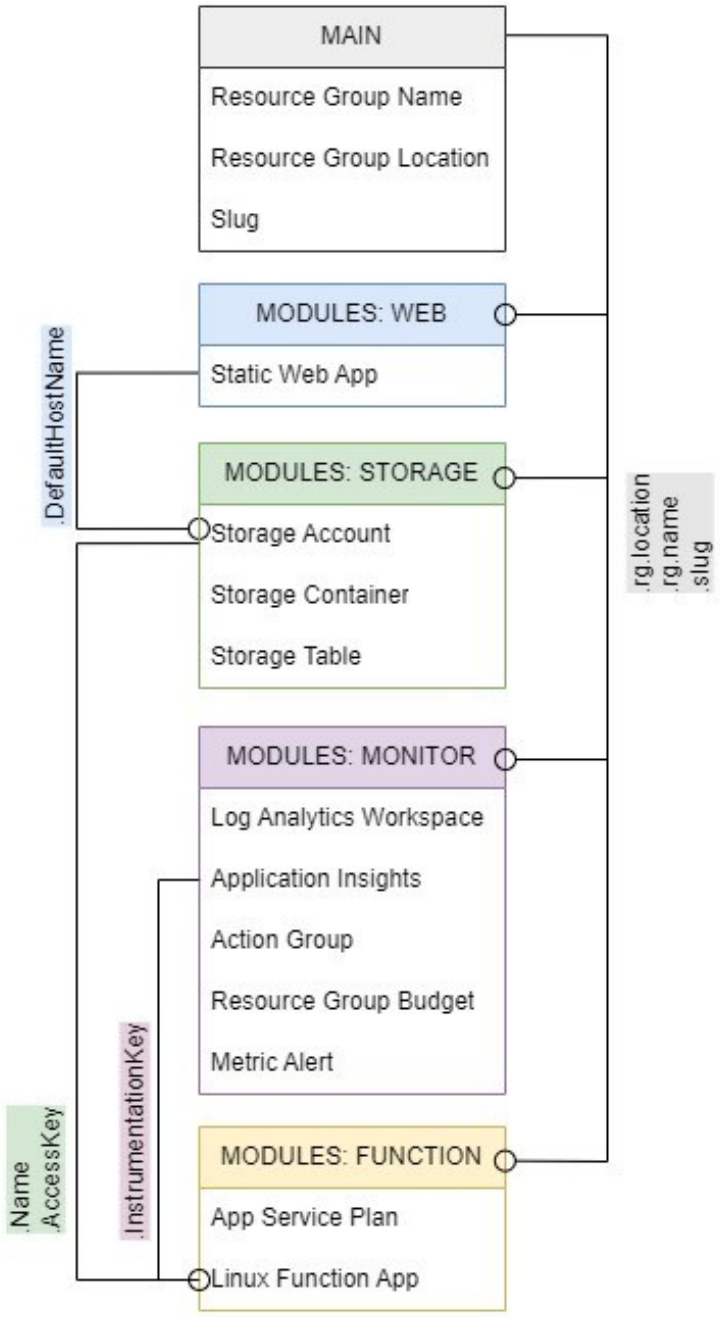
4.4.5 Resurssien keskinäiset riippuvuudet

Azuressa resurssit ovat usein jollain tapaa kytköksissä toisiinsa ja tämä täytyi huomioida myös Terraformissa resurssien määrittelyissä sekä resurssien luontijärjestyksessä. Selkeimmin tätä voidaan havainnollistaa resurssiryhmän avulla, yhtäkään resurssia ei voida luoda Azureen ennen kuin niitä varten on perustettu resurssiryhmä ja kaikkien resurssien määrityksistä täytyy löytyä yhteys resurssiryhmään. Terraformissa on useita eri tapoja linkittää resursseja toisiinsa. Moduulitasolla voidaan viitata toisen moduulin resurssiin, päätasolla voidaan viitata moduulien resursseihin ja yksittäisen moduulin sisällä voidaan viitata kyseiseen moduuliin kuuluviin resursseihin. Toiseen resurssiin voidaan viitata usealla eri tavalla ja tavan valinta on riippuvainen siitä, miten resurssin attribuutit on eri tiedostoissa esitelty. Kuvassa 15 on malli, kuinka toteutin funktiosovelluksen riippuvuuden varastointitilistä. Funktiosovellukselle tulee aina määritellä varastointitilin nimi sekä sen pääsyavain. Kuviin on korostettu selkeyden vuoksi pääsyavaimen *st_access_key* käyttölogiikka *storage*- ja *function*-moduuleissa sekä päätason *main.tf*-tiedostossa.



KUVA 15. Azuren resurssien keskinäiset riippuvuudet Terraformissa

Kaaviossa 2 on esitetty ratkaisun resurssien väliset riippuvuudet ja millä attribuutilla yhteys luodaan. Kaaviosta huomataan, että kaikki moduulit ja sitä kautta resurssit ovat riippuvaisia resurssiryhmästä ja erityisestä *slug*-attribuutista. *Slug* on malliratkaisussa käytetty viisikirjaiminen uniikki koodi, joka esiintyy kaikkien resurssien nimissä resurssin ja koko ratkaisussa luotavan sovellusinstanssin yksilöivänä tunnisteena. Resurssiryhmää tai *slug*-arvoa ei luoda Terraformilla, sillä niitä tarvitaan jo GitHubin julkaisuprosessin käynnistysvaiheessa, jossa ne myös luodaan.



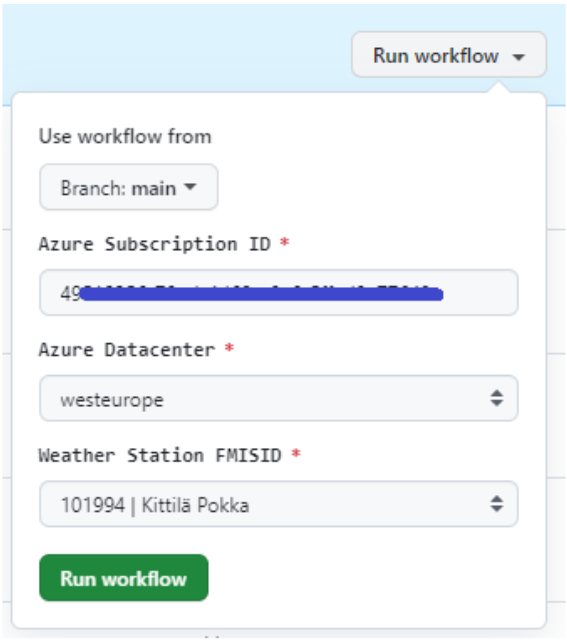
KAAVIO 2. Opinnäytetyössä toteutetun sovelluksen Azure-resurssien riippuvuudet toisistaan Terraform-koodin tasolla

4.5 GitHub Actions -työnkulut

Sovelluksen julkaisu tehdään GitHubin Actions -osion sisältävillä työnkuluilla (Workflow), jotka toteuttavat CI/CD-prosessin kokonaisuuden. Työnkulut on toteutettu YAML-merkkauksella. Kehitin ratkaisuun kolme työnkulkua, jotka suoritetaan tietyssä järjestyksessä (LIITE 1.) ja jotka yhdessä IaC-koodin sekä ohjelmakoodien kanssa luovat uuden sovellusinstanssin. Aina kun sovelluksesta halutaan julkaista uusi instanssi, kaikkien työnkulkujen suorittaminen oikeassa järjestyksessä on pakollista. Työnkulut yhdessä suorittavat käyttäjän tunnistautumisen GitHubiin ja Azureen, luovat Azureen resurssiryhmän, sovelluskäyttäjän (Service Principal) sekä varastointitilin ja koostavat ohjelmakoodin sekä testaavat ja julkaisevat sen Azureen. Lopuksi työnkuluissa kutsutaan HTTP-triggerillä varustettuja funktioita oikeassa järjestyksessä tietoaineistojen latausta sekä koneoppimismallin koulutusta varten.

4.5.1 Azure-ympäristön valmistelu

Kun GitHubin ohjelmakoodisäilöön on lisätty tässä työssä kehitetyt ohjelmakoodit, voidaan käynnistää työnkulku *1. Initialize Connectivity* manuaalisesti. Ennen käynnistystä käyttäjää pyydetään syöttämään käyttäjän oman Azure-tilin tilauksen tunnus. Lisäksi tulee valita Azuren palvelinkeskus sekä Ilmatieteelaitoksen havaintoasema kuvan 16 mukaisesti.



KUVA 16. Käyttäjältä pyydetyt syötteet ennen GitHub Actions -työnkulun suorittamista

Käyttäjältä vaaditaan tiettyjä toimia työnkulun ensimmäisessä vaiheessa, jossa hänen tulee kirjautua ohjeiden mukaisesti sekä GitHubiin että Azureen. Kirjautumisella mahdollistetaan ensimmäisten resurssien sekä sovelluskäyttäjän luominen. Työnkulussa suoritetaan tietyt GitHub CLI - ja Azure CLI -komennot, joiden avulla käyttäjälle voidaan esittää näytöllä komentojen generoimat kertakäyttökirjautumiskoodit sekä kirjautumislinkit – kumpaankin palveluun omansa. Onnistuneen kirjautumisen jälkeen GitHubille on myönnetty lupa suorittaa käyttäjän Azure-tiliin kohdistuvia toimenpiteitä, joista ensimmäinen on resurssiryhmän luonti.

Resurssiryhmän luonti on suoraviivaista eikä se vaadi erityisten oikeuksien lisäämistä Azuressa. Kuitenkin, jotta käyttäjän Azure-tilin muiden tilausten resurssit olisi suojattu tahattomilta tapahtumilta kuten resurssin poistamiselta, luodaan resurssiryhmäkohtainen sovelluskäyttäjä, joka saa oman identiteetin. Sovelluskäyttäjälle myönnetään *Contributor*-oikeudet ainoastaan juuri luotuun resurssiryhmään. *Contributor*-oikeudet tarkoittavat, että sovelluskäyttäjällä on lupa luoda, päivittää ja poistaa kaikkia resurssiryhmän resursseja. Näiden oikeuksien lisäksi sovelluskäyttäjä tarvitsee *Role Based Access Control Administrator*-oikeudet resurssiryhmään, joita se tarvitsee sovelluksen julkaisun myöhemmässä vaiheessa. Seuraavissa työnkuluissa Azureen kirjaudutaan sovelluskäyttäjän tunnuksilla, eikä käyttäjän syötettä tai kirjautumista enää tarvita.

Jotta sovelluskäyttäjän tunnuksia voidaan hyödyntää, tulee sen asiakastunnus, salasana sekä tilin tunnus tallentaa GitHub-projektin muuttujiin. Nämä noudetaan työnkulun suorituksen aikana Azuresta siihen tarkoitetulla Bash-skriptillä, jonka jälkeen ne voidaan asettaa muuttujiin. Kuvassa 17 on esitetty sovelluskäyttäjän tietojen nouto Azuresta sekä muuttujiin asetus. Kuvasta huomataan, että julkaisuprosessi hyödyntää useita muitakin muuttujia ja salaisuuksia. Muuttujien käyttö kovakoodattujen arvojen sijasta on yksi tärkeimmistä toimintatavoista koko prosessissa.

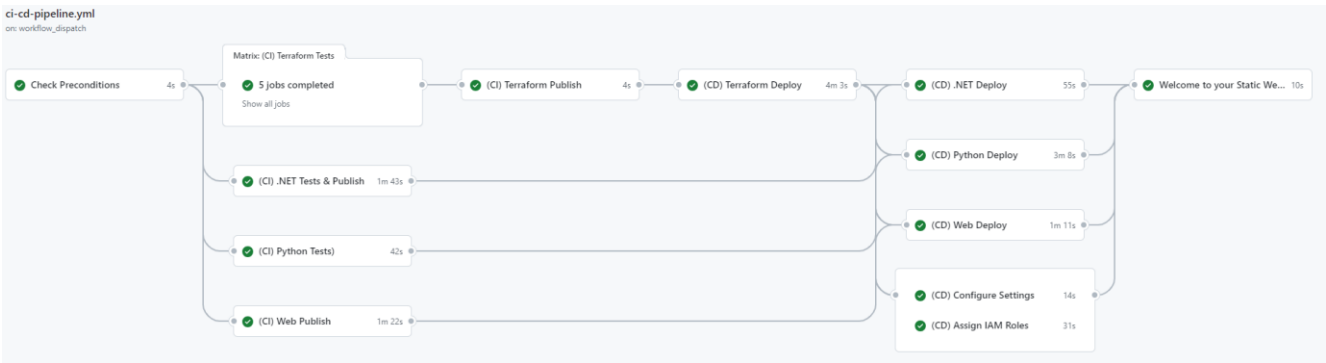
```yaml
- name: Creating new Service Principal with access limited to the Resource Group
  run: |
    sp=$(az ad sp create-for-rbac --name "terraform-deployment-${{ env.TF_VAR_slug }}-sp" --role "Contributor"
    --scopes "/subscriptions/${{ github.event.inputs.subscriptionId }}/resourceGroups/${{ env.TF_VAR_rg_name }}")
    sp=$(az ad sp create-for-rbac --name "terraform-deployment-${{ env.TF_VAR_slug }}-sp" --role "Role Based Access
    Control Administrator" --scopes "/subscriptions/${{ github.event.inputs.subscriptionId }}/resourceGroups/${{
    env.TF_VAR_rg_name }}")
    echo "TF_VAR_client_id=$(echo $sp | jq -r '.appId')" >> $GITHUB_ENV
    echo "TF_VAR_client_secret=$(echo $sp | jq -r '.password')" >> $GITHUB_ENV
    echo "TF_VAR_tenant_id=$(echo $sp | jq -r '.tenant')" >> $GITHUB_ENV

- name: Store the Secrets and Variables to GitHub Repository
  run: |
    gh secret set TF_VAR_client_id --body "$TF_VAR_client_id"
    gh secret set TF_VAR_client_secret --body "$TF_VAR_client_secret"
    gh secret set TF_VAR_tenant_id --body "$TF_VAR_tenant_id"
    gh variable set TF_VAR_subscription_id --body "$TF_VAR_subscription_id"
    gh variable set TF_VAR_slug --body "$TF_VAR_slug"
    gh variable set TF_VAR_rg_name --body "$TF_VAR_rg_name"
    gh variable set TF_VAR_rg_location --body "$TF_VAR_rg_location"
    gh variable set TF_VAR_email --body "$TF_VAR_email"
    gh variable set TF_VAR_fmisid --body "$TF_VAR_fmisid"
    gh variable set TF_VAR_fmilocation --body "$TF_VAR_fmilocation"
```

KUVA 17. Sovelluskäyttäjän tietojen nouto Azuresta Bash-skriptillä YML-merkkauksessa

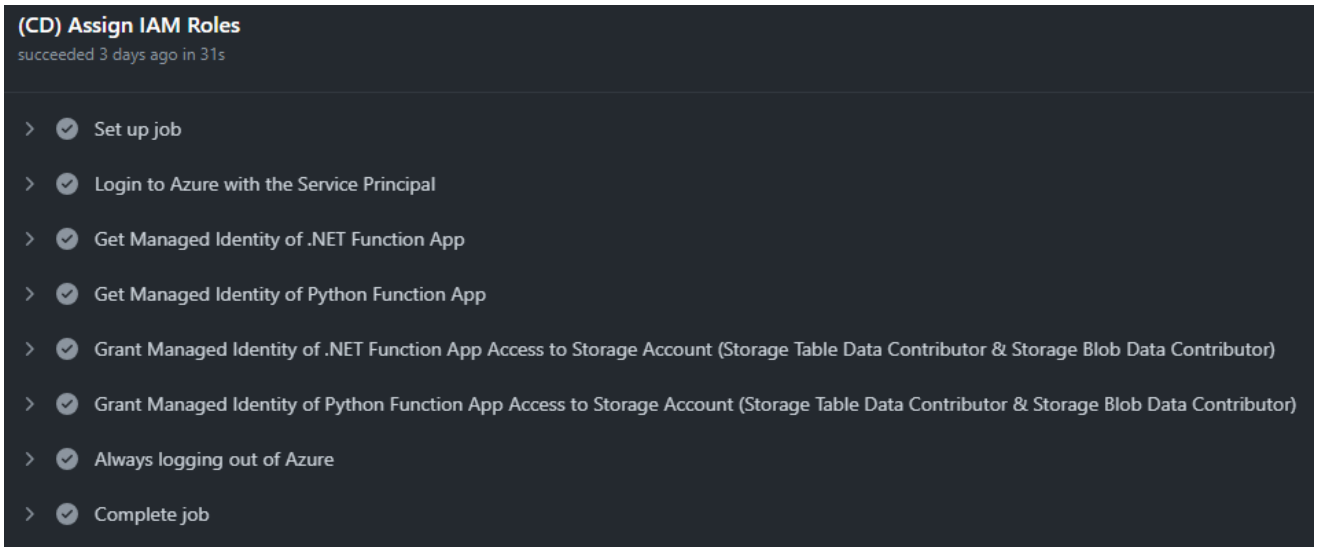
4.5.2 Sovelluksen julkaisu Azureen

Seuraavaksi suoritettava GitHub työnkulku on nimeltään *2. Trigger CI/CD* ja nimensä mukaisesti siinä suoritetaan CI/CD-prosessille ominaiset vaiheet, joita ovat järjestyksessään koodien koostaminen, testaus ja julkaisu. Työnkulun ensimmäisenä vaiheena suoritetaan tarkastus, mitkä kooditiedostot ovat muuttuneet, jotta voidaan tarvittaessa ohittaa tarpeettomien vaiheiden suoritus ja näin ollen nopeuttaa koko CI/CD-työnkulun suoritusaikaa. Mikäli on tarpeen suorittaa kaikki vaiheet, suoritusjärjestys etenee CI-vaiheessa koodien koostamisesta testaamiseen ja CD-vaiheessa koodien julkaisuun. Osa vaiheista voidaan suorittaa rinnakkain eli yhtäaikaisesti ja osa vaiheista odottaa jonkin edeltävän vaiheen valmistumista ennen kuin sen suoritus voi alkaa, kuten kuvasta 18 voidaan havaita. Tällä nopeutetaan muutoin melko hidasta työnkulun suoriutumista.



KUVA 18. GitHub Actions -työnkulku, jossa useita rinnakkaisia vaiheita.

Karkeasti kuvailtuna, Terraform-koodien testausvaiheen kanssa samanaikaisesti voidaan suorittaa .NET-, Python- ja Web App -koodien koostaminen sekä testaus. Kuitenkin valmistumisen jälkeen nämä odottavat seuraavaa vaihetta, kunnes Terraform-koodien julkaisu eli Azuren infrastruktuurin luonti on kokonaisuudessaan valmis. Terraformin valmistuttua voidaan jälleen suorittaa rinnakkaisia vaiheita. Tässä vaiheessa suoritetaan CD-vaiheet jäljellä oleville .NET-, Python- ja Web App-koodeille. Lisäksi yhtä aikaa käynnistyy funktioiden suorituksessa tarvittavien konfiguraatioasetusten tallennus varastointilin *Configurations*-nimiseen tauluun. Funktiot tarvitsevat suoriutuakseen myös luku- ja kirjoitusoikeudet varastointitilin resursseihin ja oikeuksien lisäys suoritetaan osana CD-vaihetta. Kuvassa 19 esitetään oikeuksien lisäämiseen liittyvät vaiheet työnkulun vaiheessa *Assign IAM Roles*.



KUVA 19. Varastointitilin oikeuksien lisääminen funktiosovelluksille

Kun nämä edellä mainitut vaiheet ovat onnistuneesti suoriutuneet, voidaan noutaa verkkosivuston osoite Azuresta ja esittää se käyttäjälle. Tässä vaiheessa verkkosovelluksen näkymä on vielä melko tyhjä, sillä vain osa funktioista on käynnistynyt julkaisun jälkeen automaattisesti, eikä verkkosovellus ole saanut graafien ja lukemien esittämiseksi tarvitsemiaan tiedostoja.

4.5.3 Funktioiden käynnistys

Viimeisessä *3. Initialize ML Model* -vaiheessa käynnistetään funktiot, jotta saadaan haettua Ilmatieteenlaitokselta havaintohistoria- ja ennusteaineistot sekä lopulta muodostettua verkkosovelluksen tarvitsemat tiedostot. Kaikki funktiot on ohjelmoitu käynnistymään ajastetusti timer-triggerin avulla,

mutta niihin on lisätty myös HTTP-trigger, joka mahdollistaa funktioiden käynnistämisen HTTP-kutsun avulla. HTTP-kutsuja käytetään vain työnkulkujen prosessissa sekä funktioiden paikallisessa testauksessa. Muutoin funktiot toimivat ajastetusti käynnistyen aina 20 minuutin välein. Työnkulun alussa noudetaan .NET- ja Python-funktiosovellusten pääsyavaimet Azuresta, joita tarvitaan funktioiden HTTP-kutsuissa. Avaimet tallennetaan GitHubin ympäristömuuttujiin, jolloin seuraavassa vaiheessa ne asetetaan funktioiden HTTP-kutsun otsikkotietoihin. Funktioita kutsuttaessa noudetaan vuosi kerrallaan historia-aineistoja vuodesta 2015 lähtien, jolloin koneoppimismallin koulutusta varten saadaan kattavasti mittauksia usealta talvelta. Kuvassa 20 on esitetty funktioiden kutsujärjestys *while*-silmukassa ja funktioiden nimet ovat alleviivattuna valkoisella.

```
- name: Improve ML Model with old Observations Since 2015
  run: |
    # Get old observations year by year since 2015 with a while loop. Improve the model with more data, and
    refresh the predictions for the UI.
    this_year=$(date +%Y)
    year=2015
    while [ $year -le $this_year ]
    do

      echo "Fetching old observations for year $year"
      body=$(echo -n "{\"FromUtc\":\"${year}-01-01T00:00:00Z\",\"ToUtc\":\"${year}-12-31T23:59:59Z\"}")
      curl -f -X POST -H "Content-Type: application/json" -H "x-functions-key: ${{ env.DOTNET_FUNCTION_KEY }}
      " --data $body https://${{ vars.TF_VAR_slug }}-dotnet-func.azurewebsites.net/api/
      IngestHistoryObservationsHttpTriggerAsync

      echo "Re-training ML Model including the observation data from year $year"
      curl -f -X POST -H "Content-Type: application/json" -H "x-functions-key: ${{ env.PYTHON_FUNCTION_KEY }}
      " https://${{ vars.TF_VAR_slug }}-python-func.azurewebsites.net/api/model_training_http_trigger

      echo "Predict Future Snow Delta based on the model re-trained in the previous step"
      curl -f -X POST -H "Content-Type: application/json" -H "x-functions-key: ${{ env.PYTHON_FUNCTION_KEY }}
      " https://${{ vars.TF_VAR_slug }}-python-func.azurewebsites.net/api/save_predictions_http_trigger

      echo "Aggregating more accurate data for UI"
      curl -f -X POST -H "Content-Type: application/json" -H "x-functions-key: ${{ env.DOTNET_FUNCTION_KEY }}
      " https://${{ vars.TF_VAR_slug }}-dotnet-func.azurewebsites.net/api/AggregateWebPageDataHttpTriggerAsync

      year=$((year+1))
    done
```

KUVA 20 Azure-funktioiden HTTP-kutsut Bash-skriptillä YML-merkkauksella

Yksinkertaistettuna *while*-silmukassa haetaan aineistoja vuosi kerrallaan, koulutetaan koneoppimismalli, luodaan ennuste ja muodostetaan staattisen verkkosovelluksen luettavissa oleva ennustetiedosto. Tätä prosessia toistetaan, kunnes päästään nykyhetkeen. Tämän vaiheen kokonaissuoritukseen voi kulua jopa 15 minuuttia, vaikka funktioiden sisäiseen suoritukseen onkin ohjelmoitu rinnakkaisuutta. Kun viimeinenkin työnkulku on suoritettu onnistuneesti loppuun, on verkkosovelluksen näkymässä ajantasainen lumen määrän muutosta ennustava graafi.

4.5.4 Autentikaatioprosessi

Työnkulkujen suorittaminen ja infrastruktuurin luonti edellyttävät tiettyjä autentikointiin liittyviä vaiheita ja toimivien autentikointiprosessien löytäminen olikin yksi työn kulmakivistä. Näistä yksi tärkeimmistä oli ensimmäisen työnkulun ensimmäinen vaihe, kun GitHub CLI komentoa *gh auth login* käytetään kirjautumisessa GitHubiin. Ajettaessa komento GitHubissa, käyttäjältä ei vaadita muita toimia kuin työnkulun tarjoaman koodin lisäämistä linkistä avautuvaan kenttään ja automaatio kykenee toimimaan minimaalisella käyttäjän interaktiolla, sillä komento tunnistaa GitHubin suoritusympäristökseen. Esimerkiksi samaa komentoa paikallisesti komentokehotteesta käytettäessä, tulisi käyttäjän määritellä muutamia asetuksia syöttämällä ne komentotulkille interaktiivisesti ennen kuin se tarjoaisi kirjautumiskoodin sekä linkin. Tällainen toimintamalli myös GitHubin automaatiotyönkulussa olisi estänyt kirjautumisprosessin ja automaation sijasta olisi täytynyt ainakin osittain täytynyt siirtyä manuaalisiin työvaiheisiin GitHub-julkaisualustan konfiguroinnissa. Azure CLI komento *az login --use-device-code* sen sijaan tarjoaa laitekoodin suoraan myös paikallisesti komentokehotteessa ajettuna ilman käyttäjän syötteitä.

Kun käyttäjä on kirjautuneena omaan GitHub- ja Azure-tileihinsä, on mahdollista luoda sovelluskäyttäjä ja myöntää sille tarvittavia oikeuksia. On hyvä käytäntö käyttää identiteettejä, jotka eivät liity luonnolliseen henkilöön. Näin varmistetaan, että useat projektissa olevat eri kehittäjät voivat suorittaa automaattisia työnkulkuja eikä tunnistautumista vaadita. Kehittäjälle tulee tässä tapauksessa myöntää oikeudet vain GitHub-projektiin.

Riittävillä oikeuksilla varustetun sovelluskäyttäjän avulla pystyin antamaan funktiosovellukselle luku- ja kirjoitusoikeudet niiden tarvitsemaan varastointiliin. Tämä edellytti, että määrittelin Terraformissa funktiosovelluksen käyttämään hallittua identiteettiä, johon liittyvät tiedot noudetaan työnkulun ajon aika Azuresta. Tämä olisi ollut vältettävissä, mikäli olisin käyttänyt varastointitilin yhteysmerkkijonoa funktioiden suorituksissa, mutta tämä tapa ei nykyään ole enää turvallisuussyistä suositeltu, joten halusin välttää sen käyttöä. Normaalitilanteessa tämän kaltainen oikeuksien lisäysprosessi hoidettaisiin riittävillä valtuuksilla jonkun henkilön toimesta joko suoraan Azuren portaalissa tai Powershell tai Azure CLI-komennoilla. Tässä en kuitenkaan halunnut, että työnkulkujen suoritusten välissä käyttäjältä vaadittaisiin minkäänlaisia lisätoimia, joten oikeuksien lisääminen tehtiin automaatiolla osana työnkulkua Azure CLI-komentojen avulla.

5 YHTEENVETO JA POHDINTA

Opinnäytetyön aihe oli hyvin laaja, mutta mielestäni äärimmäisen tärkeä osa ohjelmistokehityksen kokonaisuuden ja ennen kaikkea työelämän vaatimusten ymmärtämistä. Tarkoituksena oli yhdistää opintojen yhteydessä opitut yksittäiset osa-alueet yhteen sekä laajentaa että syventää kokonaisuutta työelämästä saaduilla opeilla ja kokemuksilla. Työhön sisältyi paljon jo entuudestaan tuttuja aiheita ja menetelmiä, mutta halusin käyttää tilaisuuden hyväksi ottamalla mukaan paljon uusia teknologioita ja työkaluja mm. Terraform ja GitHub Actions. Lopulta kävikin niin, että juuri nämä elementit päätyivät koko työn ydinasiaksi ja osoittautuivat varsin oivallisiksi työkaluiksi, joita toivoisinkin voivani käyttää myös työssäni tulevaisuudessa.

Vaikka toteutin työn yksilötyöskentelynä ilman työelämäriippuvuutta koin, että DevOps-menetelmien ominaispiirteet toteutuivat tietyin osin. Käytännössä tein osakokonaisuuksien testijulkaisuita hyvin tiheästi ja sillä periaatteella, ettei lopputuloksen tarvinnut olla lähellekään täydellinen, kunhan jotain pientä kehitystä tapahtuisi jokaisella julkaisukerralla. Hyvin usein kaikki DevOpsin tunnusomaiset toiminnot, eli suunnittelu, ohjelmointi, koodin koostaminen, testaus, julkaisu, käyttöönotto, operointi ja monitorointi toteutuivat järjestyksessään iteratiivisen kehitystyön aikana. Tästä käytännön esimerkkinä mainittakoon tapaukset, joissa ohjelmakoodin virheet tulivat esille vasta Azureen julkaisun jälkeen, kun funktiosovelluksia suoritettiinkin pilviympäristössä oman työaseman sijasta. Asettamieni hälytysten mukaisesti, sain sähköpostin suorituksessa tapahtuneesta virheestä ja tämän jälkeen selvitystyö tapahtui Azuren portaalissa sovellusseurannan lokiviestejä tutkimalla.

Työn myötä sain lisävahvistusta siitä, että DevOps-periaatteiden mukainen automaatio ja jatkuvat integrointi- ja julkaisuprosessit mahdollistavat tehokkaan ohjelmakoodin muutosten hallinnan ja nopean käyttöönoton pilviympäristössä. Tämä vähentää käyttökatkoja ja mahdollisia virheitä, parantaen samalla sovelluksen luotettavuutta. Jos työssä kehittämäni malliratkaisu olisi toteutettava laajemmassa mittakaavassa jonkin organisaation tilauksesta, ratkaisun ympärille muodostuisi monialainen toteutustiimi, jossa kaikilla olisi omat vastuualueensa. Itse sovelluksen toteuttamiseen voitaisiin tarvita useamman eri osaajan työpanosta, datatieteilijät vastaisivat tiedon keräyksestä ja sen analysoinnista, ohjelmistokehittäjät vastaisivat sovelluksen toteuttamisesta ja käyttöliittymäsuunnittelijat suunnittelisivat sovellukselle käyttöliittymän. Näiden lisäksi tulisi olla tiimi, joka vastaisi uuden sovelluksen elinkaaresta ja ylläpitämisestä yhteistyössä toteutustiimin kanssa.

Aiheen laajuuden vuoksi jouduin joitain yksittäisiä osa-alueita jättämään käsittelystä pois. Alkuperäinen aihekattaus olisi sisältänyt enemmän myös itse sovelluksen ohjelmointiosuuden läpikäyntiä, mutta se olisi paisuttanut työtä turhan paljon. Lopulta kohtuullinen tasapaino löytyi, kun päätin jättää sovelluksen koneoppimisosuuden sekä tiedon keräysmetodit työstä vähemmälle huomiolle ja keskittyä arkkitehtuuriin, IaC-malliin sekä automaatioon. Sovelluksen työstö oli kuitenkin välttämätöntä, jotta pääsin mallintamaan ja esittämään käytetyt prosessit todellisuutta vastaavassa tilanteessa. Toki sovellus olisi voinut olla todella paljon kevyempi, mutta kyseinen sovellus tulee ainakin minulle itselleni käyttöön, eikä tehty kehitystyö siten ollut laisinkaan turhaa.

LÄHTEET

Alexandra. 2023. *What Is SDLC? Understand the Software Development Life Cycle.* Stackify 10.3.2023. Saatavissa: https://stackify.com/what-is-sdlc/. Viitattu 20.11.2023

Application Insights overview. 2023. Microsoft 28.12.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview. Viitattu: 10.12.2024

Azure App Service plan overview. 2023. Microsoft 26.5.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/app-service/overview-hosting-plans. Viitattu: 10.12.2023

Azure Monitor overview. 2023. Microsoft 7.12.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-monitor/overview. Viitattu 10.12.2023

Azure Storage redundancy. 2023. Microsoft 8.1.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy. Viitattu 3.12.2023

Bigelow, S., Courtemanche, M. & Gillis, A. 2023. *What is DevOps? The ultimate guide.* Saatavissa: https://www.techtarget.com/searchitoperations/definition/DevOps. Viitattu: 30.1.2024

Brock, D. 2021. *Ask About Azure: Why do resource groups need a location?* Saatavissa: https://www.daveabrock.com/2021/03/08/ask-azure-resource-group-locations
Viitattu: 25.11.2023

Defender EASM Overview. Microsoft 14.7.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/external-attack-surface-management/. Viitattu 6.12.2023

Dodd, R. 2023. *What's a Canary Deployment?* Saatavissa: https://launchdarkly.com/blog/four-common-deployment-strategies/. Viitattu 6.12.2023

Falck, K. 2023. *Pilvi-infra automatisoituu koodilla – näin toimii iac.* Tivi 10.6.2023. Saatavissa: https://www.tivi.fi/uutiset/pilvi-infra-automatisoituu-koodilla-nain-toimii-iac/72901252-382e-4efb-b425-06f8d18dbf3f?fbclid=IwAR3FQiCUs5iJyq_t_QI56eG_4mCAn2UuvDRQxle5PgwnEHYm-QLvL5uKR75I. Viitattu 25.11.2023

Gillis, A. 2021. *continuous delivery (CD).* Saatavissa: https://www.techtarget.com/searchitoperations/definition/continuous-delivery-CD. Viitattu: 4.2.2024

Gunathilaga, S. 2022. *How to select the most suitable Hosting Plan for Azure Function App deployment?* Saatavissa: https://sahangunathilaka.medium.com/how-to-select-the-most-suitable-hosting-plan-for-azure-function-app-deployment-8896f0a63a6a. Viitattu: 30.11.2023

How to configure monitoring for Azure Functions. Microsoft 29.11.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-functions/configure-monitoring?tabs=v2. Viitattu 10.12.2023

Identity architecture design. 2023. Microsoft. Saatavissa: https://learn.microsoft.com/en-us/azure/architecture/identity/identity-start-here. Viitattu: 28.1.2024

Introduction to Azure Blob Storage. Microsoft 11.10.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction. Viitattu 24.2.2024

Kivelä, N. 2023. *DevOps-Team is a Myth.* Saatavissa: https://devops-latam.cioreview.com/cxoinsight/devopsteam-is-a-myth-nid-37184-cid-354.html. Viitattu 6.12.2023

Microsoft 2023. *Azure-well-architected Framework.* Saatavissa: https://learn.microsoft.com/pdf?url=https%3A%2F%2Flearn.microsoft.com%2Fen-us%2Fazure%2Fwell-architected%2Ftoc.json. Viitattu 5.12.2023

Montgomery, J., Marko, K., Rando, N. 2021. *cloud infrastructure*. Saatavissa: https://www.techtarget.com/searchcloudcomputing/definition/cloud-infrastructure. Viitattu 25.11.2023

Rastogi, S. 2023. *Storing and Retrieving Secrets in Azure Key Vault for Azure App Service.* Saatavissa: https://medium.com/@smritirastogi33/storing-and-retrieving-secrets-in-azure-key-vault-for-azure-app-service-4074ff7ac4bb. Viitattu: 10.12.2023

RedHat. 2023. *What is CI/CD?* Saatavissa: https://www.redhat.com/en/topics/devops/what-is-ci-cd. Viitattu 20.11.2023

Rivera, A. 2020. *Smoke testing in CI/CD pipelines*. Saatavissa: https://circleci.com/blog/smoke-tests-in-cicd-pipelines/. Viitattu 6.12.2023

Rod, B. 2023. *Shared Responsibility Model in SaaS | Definition, Benefits, Trends.* Saatavissa: https://acsense.com/blog/shared-responsibility-model-in-saas/. Viitattu: 4.2.2024

Rosencrance, L. 2021. *SaaS vs. IaaS vs. PaaS: Differences, Pros, Cons and Examples.* Saatavissa: https://www.techtarget.com/whatis/SaaS-IaaS-PaaS-Comparing-Cloud-Service-Models. Viitattu: 4.2.2024

Sakovich, N. 2023. *Cloud Software Development: All You Need to Know*. Saatavissa: https://www.sam-solutions.com/blog/cloud-software-development/. Viitattu 21.11.2023

SAST vs. DAST. 2023. GitLab. Saatavissa: https://about.gitlab.com/topics/devsecops/sast-vs-dast/. Viitattu 20.11.2023

Science Soft. 2023. *Software Development Automation at Every Step of SDLC*. Saatavissa: https://www.scnsoft.com/software-development/automation#automation. Viitattu 20.11.2023

Segura, T. 2023. *Top 10 Practices for Secure Software Development.* Saatavissa: https://blog.gitguardian.com/top-10-practices-for-secure-software-development/. Viitattu 6.12.2023

Song, M., Yun, H., Lee, J., Yum, S. 2022. *A Comparative Analysis of Machine Learning Algorithms for Snowfall Prediction Models in South Korea. 1* Saatavissa: https://nhess.copernicus.org/preprints/nhess-2022-118/nhess-2022-118.pdf. Viitatttu: 8.12.2023

Storage account overview. Microsoft 6.12.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview. Viitattu: 25.11.2023

Sullivan, N. 2015. *An introduction to JavaScript-based DdoS.* Saatavissa: https://blog.cloudflare.com/an-introduction-to-javascript-based-ddos. Viitattu: 3.12.2023

Synopsys. 2023. *Cross-Site Scripting (XSS).* Saatavissa: https://www.synopsys.com/glossary/what-is-cross-site-scripting.html. Viitattu: 3.12.2023

Tutorial: Createa and manage budgets. 2023. Microsoft 2.11.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/cost-management-billing/costs/tutorial-acm-create-budgets. Viitattu: 10.12.2023

Yazar, K. 2023. *continuous deployment.* Saatavissa: https://www.techtarget.com/searchitoperations/definition/continuous-deployment. Viitattu: 4.2.2024

What are Azure Monitor alerts? 2023. Microsoft 6.12.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-overview. Viitattu: 10.12.2023

What is Azure Static Web Apps? Microsoft 24.4.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/static-web-apps/overview. Viitattu 8.12.2023

What is Azure Table storage? Microsoft 29.11.2022. Saatavissa: https://learn.microsoft.com/en-us/azure/storage/tables/table-storage-overview. Viitattu: 29.11.2023

What is DevSecOps?. 2023. GitLab. Saatavissa: https://about.gitlab.com/topics/devsecops/. Viitattu 20.11.2023

What is infrastructure as code (IaC)? Microsoft 28.11.2022. Saatavissa: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code. Viitattu 25.11.2023

What is Microsoft Defender for Cloud?. 2023. Microsoft 16.11.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/defender-for-cloud/defender-for-cloud-introduction. Viitattu 6.12.2023

What is a resource group. Microsoft 16.8.2023. Saatavissa: https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal. Viitattu: 25.11.2023

Watts, S, Muhammad, R. 2019. *SaaS vs PaaS vs IaaS: What's The Difference & How To Choose* Saatavissa: https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/. Viitattu 6.12.2023

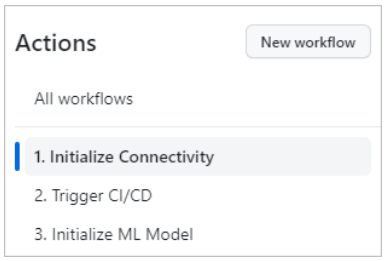
Powder Snow projektin GitHub työnkulkujen suoritusohje sovelluksen ensimmäisen julkaisun yhteydessä.

Esivalmistelut ja ehdot:

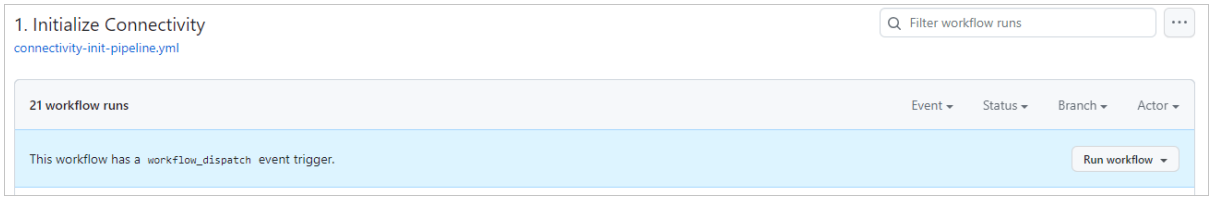
- Azure-tili.
- GitHub-tili.
- Kloonattu repositorio julkaistaan sellaisenaan GitHubissa.
- Varmistetaan, että tilin tallennustila ei ole täynnä, sillä projektin työnkulut luovat tallennustilaa vaativia artifacteja.
- HUOM! Ohjeet englanniksi myös repositorion github-hakemiston README.md-tiedostossa, joka sisältää työnkulkujen kuvaukset sekä yleisohjeet niiden suorittamiseen.

Työnkulkujen suoritus (sovelluksen julkaisu):

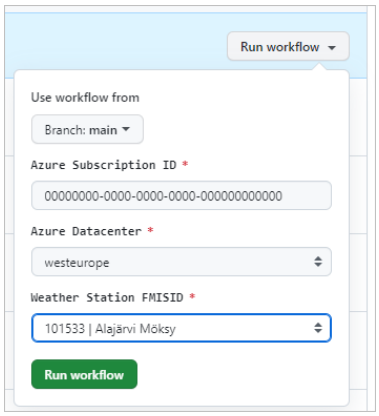
- Klikataan repositorion etusivun yläpalkista löytyvää ”Actions” valikkoa.
- Avautuvan sivun vasemmassa laidassa on listattuna kaikki repositorioon liittyvät työnkulut (workflows). Työnkulkuja on 3 kpl ja ne on numeroitu suoritusjärjestyksessä.
- Työnkulkujen suorittaminen aloitetaan valitsemalla ”1. Initialize Connectivity”.



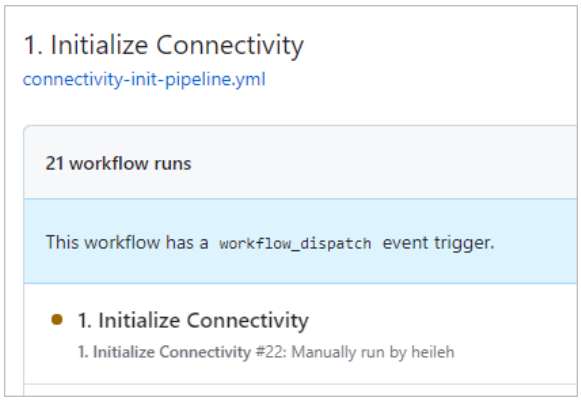
- Avautuvasta näkymästä valitaan oikean laidan ”Run worklow”.



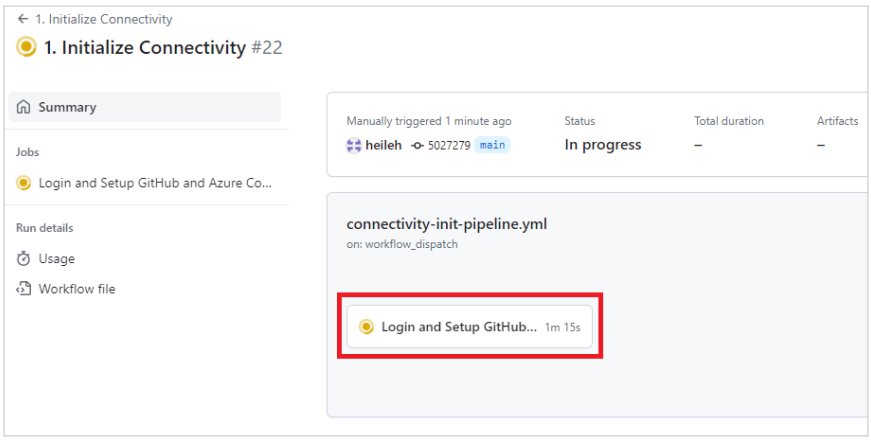
- Täytetään vaaditut kentät avautuvasta valikosta.
    - HUOM! Azure Subscription ID-kentän tulee olla täytetty täsmällisesti ilman tyhjiä välilyöntejä.
- Valitaan ”Run workflow”



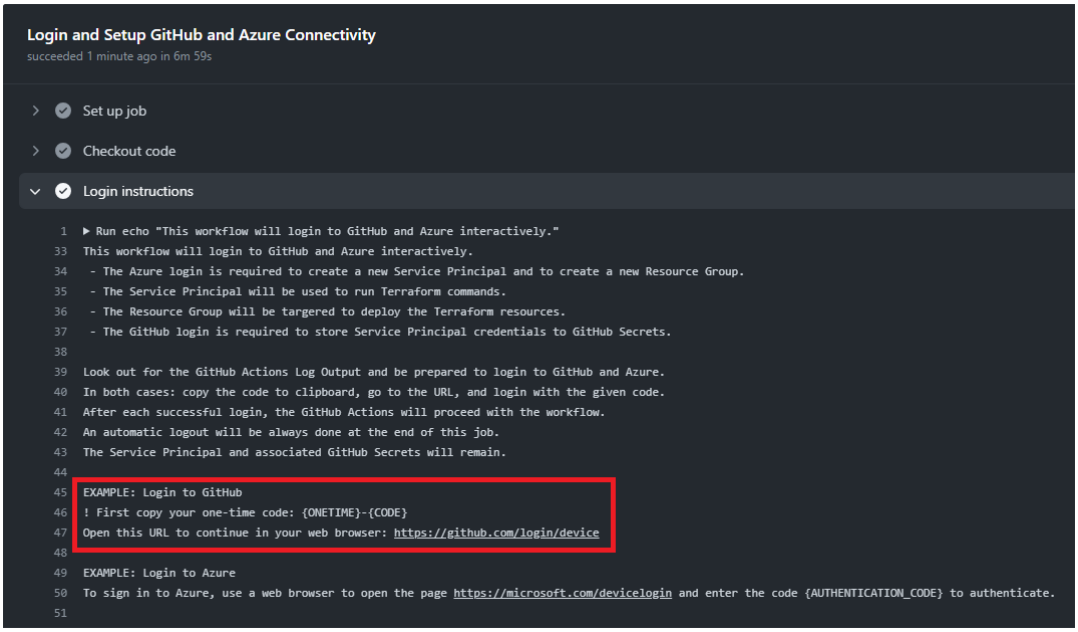
- Kun työnkulku on käynnistynyt, voidaan suoritusta tarkastella klikkaamalla työnkulun nimeä.



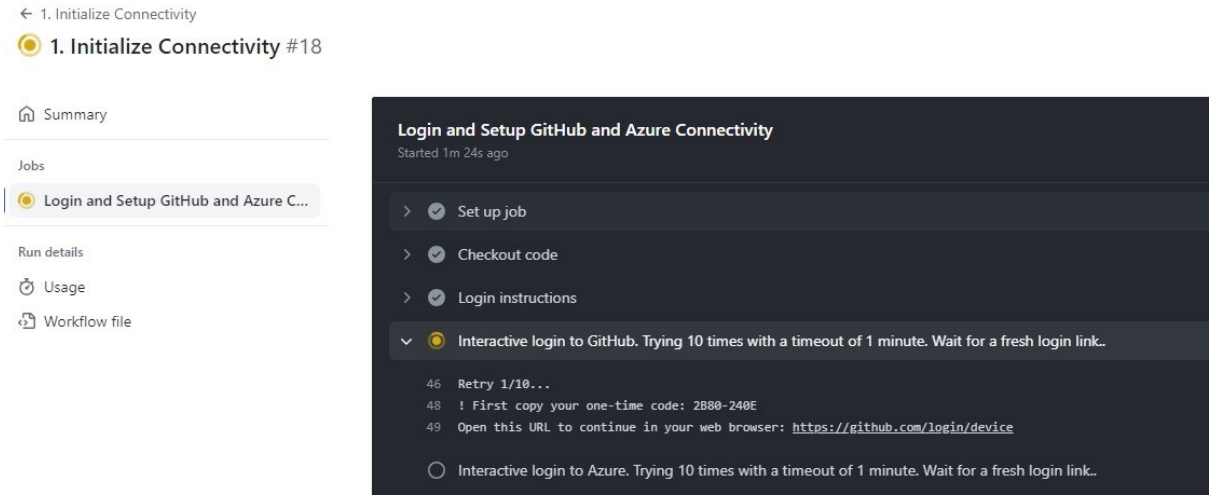
- Valitaan ”Login and Setup GitHub…”.



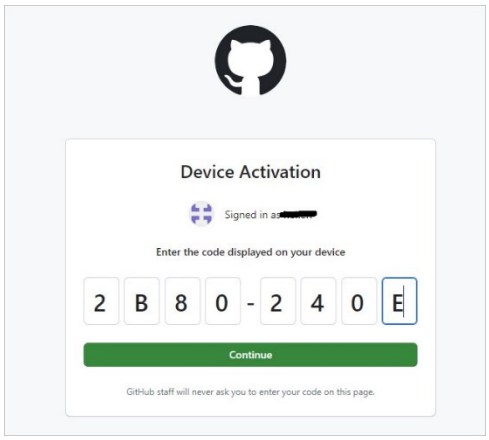
- Avautuvassa näkymässä voidaan avata ”Login instructions”-osa, jossa esitetään esimerkki seuraavassa vaiheessa tapahtuvasta GitHub-kirjautumisesta.



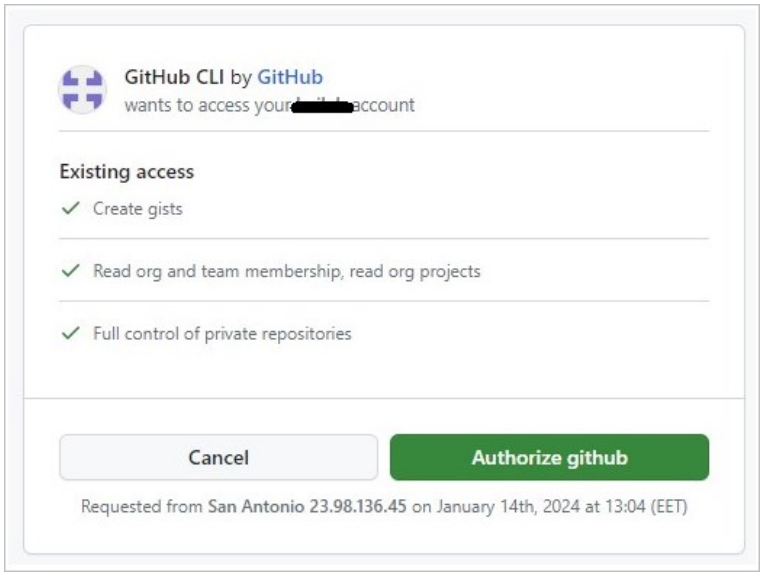
- Odotetaan, kunnes vaihe “Interactive login to GitHub…” aktivoituu ja tarjoaa kertakäyttöisen kirjautumiskoodin sekä linkin.
    - HUOM. Kirjautumiseen on annettu aikaa 1 minuutti, jos aikaa kuluu enemmän, tarjoaa työnkulku uuden koodin ja linkin.



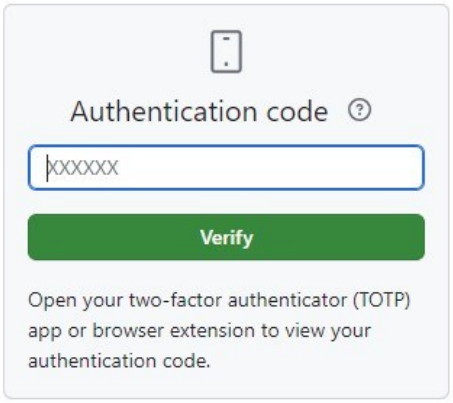
- Kopioidaan koodi leikepöydälle ja avataan linkki uuteen välilehteen.
- Välilehdellä liitetään kopioitu koodi sille varattuun tilaan ja valitaan ”continue”.



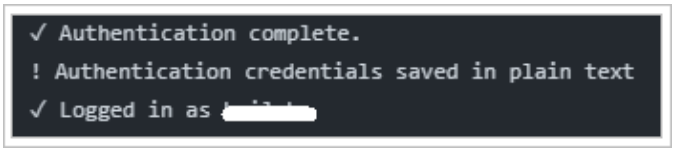
- Seuraavaksi avautuvassa ikkunassa valtiaan ”Authorize github”.



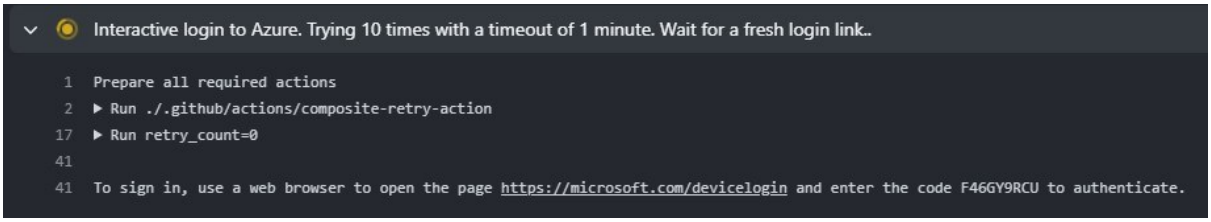
- Jos GitHub-tilillä on otettu käyttöön kaksivaiheisen tunnistautumisen, pyydetään seuraavaksi antamaan autentikointisovelluksesta löytyvä tunnistautumiskoodi.
- Syötetään koodi ja valitaan ”verify”.



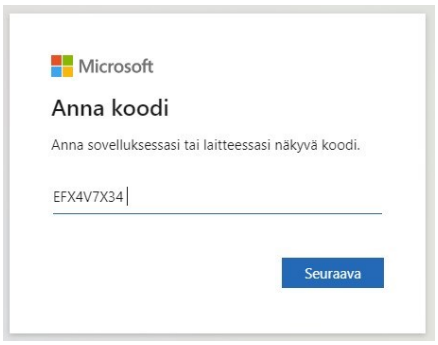
- Onnistuneen GitHub -kirjautumisen jälkeen työnkulkuun tulostuu seuraavat viestit.



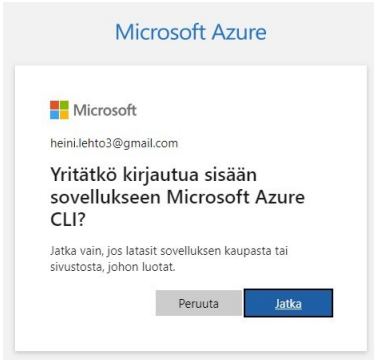
- Seuraava vaihe ”Interactive login to Azure” käynnistyy ja noudattelee pitkälti samoja vaiheita kuin GitHub-kirjautuminen



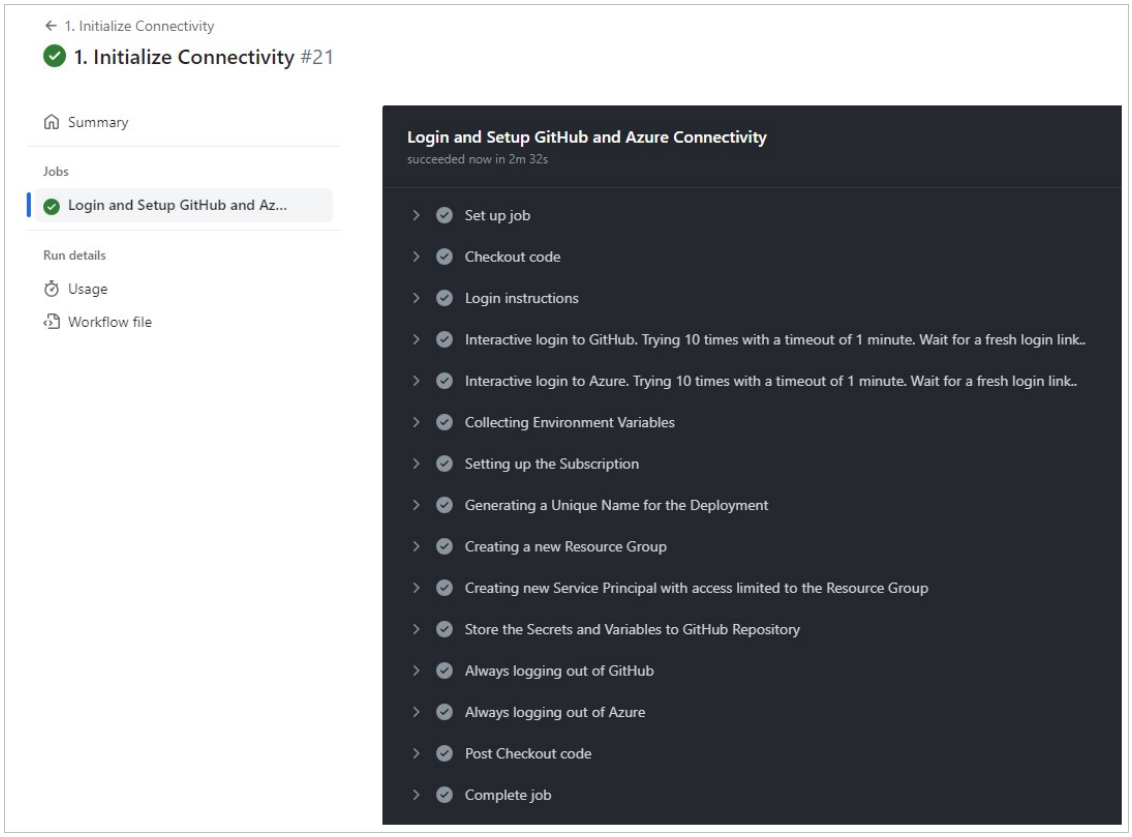
- Kopioidaan jälleen koodi leikepöydälle ja avataan linkki uuteen välilehteen.
- Syötetään koodi sille tarkoitettuun tilaan välilehdellä ja valitaan ”seuraava”.



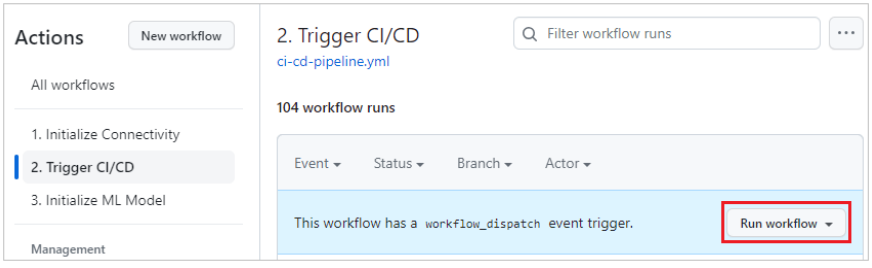
- Hyväksytään painamalla “Jatka”.



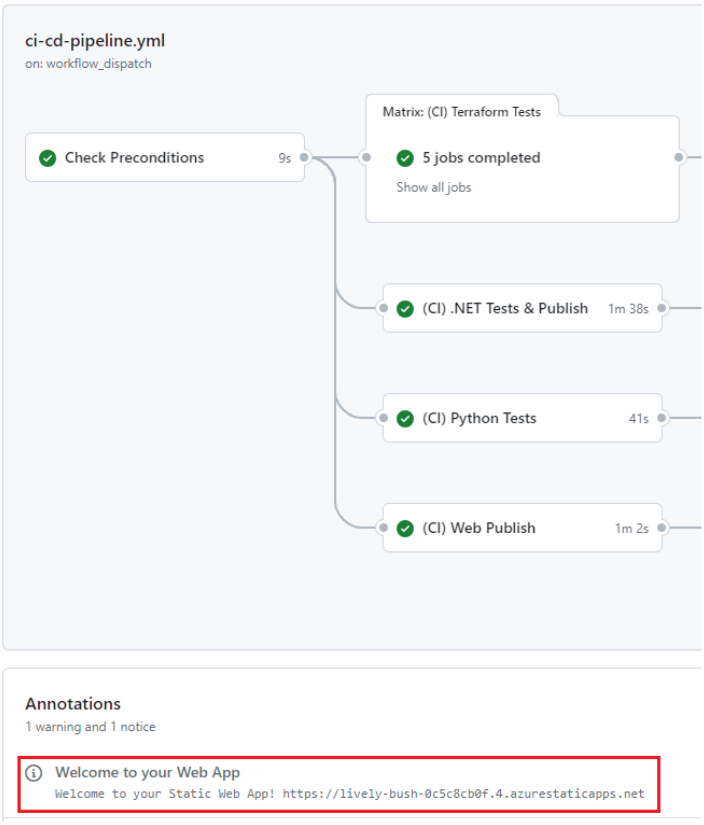
- Palataan takaisin työnkulun suoritukseen ja odotetaan, kunnes kaikki vaiheet on suoritettu loppuun.
- Onnistunut työnkulun suoritus näyttää tältä.



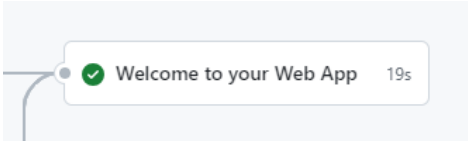
- Palataan takaisin työnkulkujen koostenäkymään ja valitaan nyt työnkulku ” 2. Trigger CI/CD”.
- Valitaan jälleen ”Run workflow”

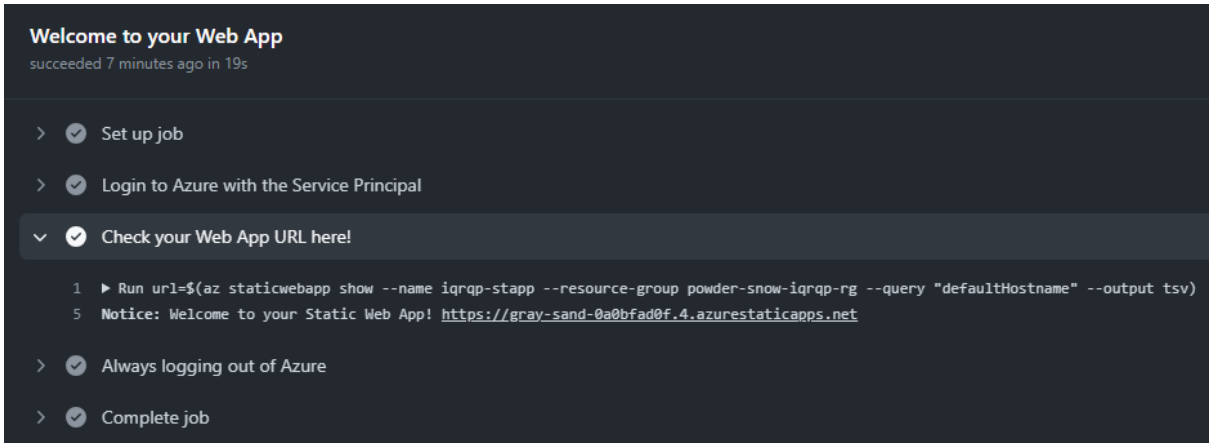


- Tässä vaiheessa ei tarvitse kuin odottaa työnkulun valmistumista, jonka jälkeen näytölle tulostetaan juuri luodun verkkosivun osoite.



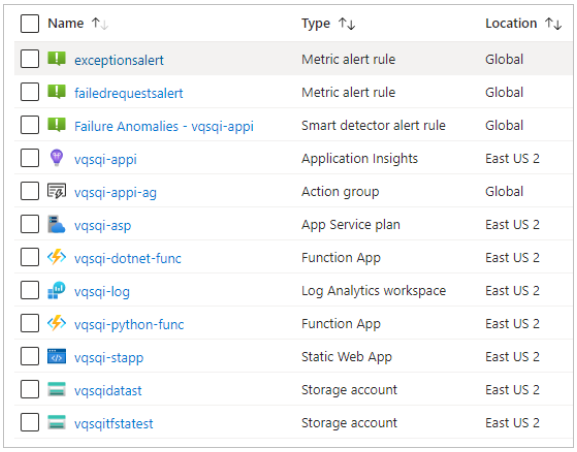
- Suora linkki voidaan löytää myös klikkaamalla työnkulun viimeistä vaihetta ”Welcome to your Web App” Tai vaihtoehtoisesti Azuren portaaliin luodusta Static Web App -resurssista.





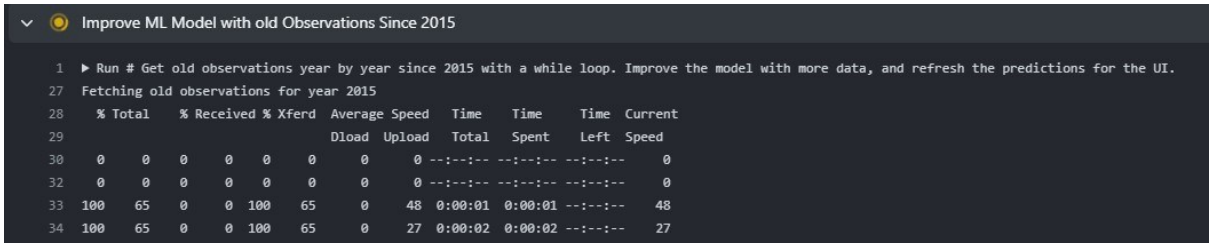
- Tässä vaiheessa verkkosivulla ei ole näkyvillä kuin otsikko, sillä viimeinen työnkulku on suorittamatta.

- Kaikki Azuren resurssit on nyt luotu ja ne ovat toiminnassa.



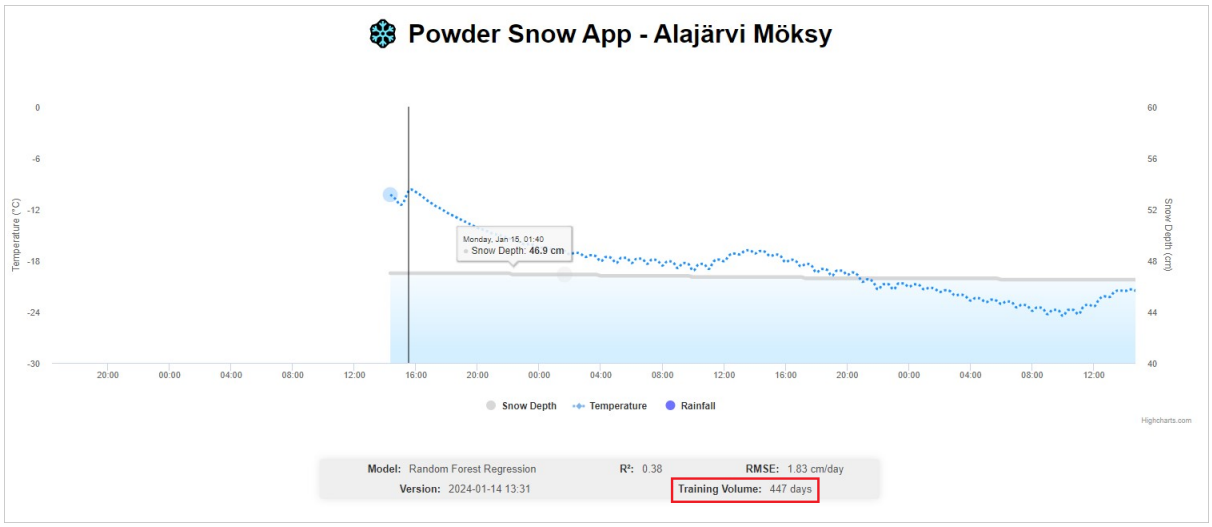
Name	Type	Location
exceptionsalert	Metric alert rule	Global
failedrequestsalert	Metric alert rule	Global
Failure Anomalies - vqsqi-appi	Smart detector alert rule	Global
vqsqi-appi	Application Insights	East US 2
vqsqi-appi-ag	Action group	Global
vqsqi-asp	App Service plan	East US 2
vqsqi-dotnet-func	Function App	East US 2
vqsqi-log	Log Analytics workspace	East US 2
vqsqi-python-func	Function App	East US 2
vqsqi-stapp	Static Web App	East US 2
vqsqidatast	Storage account	East US 2
vqsqitfstatest	Storage account	East US 2

- Palataan takaisin työnkulkujen koostenäkymään ja valitaan nyt työnkulku ” 3. Initialize ML Model”.
- Valitaan ”Run workflow”.
- Tämän työnkulun suorittamiseen kuluu arviolta 15–20 minuuttia aikaa, sillä tässä noudetaan koneoppimismallin kouluttamista varten Ilmatieteenlaitoksen dataa vuodesta 2015 lähtien.



- HUOM! Mikäli työnkulku päätyy virheeseen, voi sen käynnistämistä kokeilla uudelleen.
- Ajon aikana verkkosivua voi päivittää ja huomata, kuinka koneoppimismallin tehokkuudesta kertovassa metriikkataulussa oleva ”Training Volume” arvo kasvaa sitä mukaa, kun saadaan vuosi kerrallaan dataa noudettua.
- Uusi esiintymä sovelluksesta on nyt luotu!

- Verkkonäkymä päivittyy tästä eteenpäin 20 minuutin välein, kun Ilmatieteenlaitokselta on haettu uusi sääennustedata ja funktiot ovat prosessoineet datan.



- Poistetaan resurssit Azuresta poistamalla koko resurssiryhmä.
- Poistetaan Service Principal sekä Application Azure Entrasta. Ne löytyvät helpoiten kirjoittamalla hakukenttään: terraform-deployment-{slug}-sp.
    - Microsoft Entra ID-otsikon alla.
- slug-arvo löytyy helpoiten resurssiryhmän nimestä powder-snow-{slug}-rg.