Serverless-prototyyppi lisensointiohjelmistolle

Niklas Ulmanen

Haaga-Helia ammattikorkeakoulu

Tradenomin tutkinto

Opinnäytetyö

2022

Tiivistelmä

Tekijä
Niklas Ulmanen

Tutkinto
Tradenomi

Opinnäytetyön nimi
Serverless-prototyyppi lisensointiohjelmistolle

Sivu- ja liitesivumäärä
47 + 0

Tämän toiminnallisen opinnäytetyön tarkoituksena oli luoda toimeksiantajayrityksen ensimmäinen serverless-teknologiaa hyödyntävä prototyyppi, jolla voidaan julkaista ja ottaa käyttöön yrityksen kehittämä ohjelmisto. Työhön sisältyy myös prototyypin käytettävyyden validointi, jonka tehtävänä on luoda käsitys siitä, voisiko teknologiaa käyttää yrityksen palveluissa.

Prototyyppi on toteutettu pääasiassa Java-ohjelmointikielellä, jolla ympäristön määrittely on tapahtunut. Prototyyppi pohjautuu Amazon Web Services:in tarjoamiin palveluihin, joista työn kannalta tärkein on Fargate. Fargate mahdollistaa ohjelmistojen julkaisemisessa serverless-teknologiaa hyödyntävät ratkaisut käyttämällä virtuaalikontteja. Prototyypin ohelle kehitettiin myös skriptejä, joilla voidaan parantaa ja automatisoida julkaisun ja käyttöönoton kannalta tärkeitä prosesseja.

Osana validointia käytiin läpi prototyypin hyviä ja huonoja puolia. Serverless-teknologiasta löytyi useita hyödyllisiä ominaisuuksia ja se oli selkeästi toimintatavoiltaan tuoreempi ja kehittyneempi kuin aikaisemmin käytetty ratkaisu. Joillakin osa-alueilla serverless-teknologia oli edeltäjäänsä parempi, mutta toisilla osa-alueilla se oli myös heikompi. Ratkaisuiden erot eivät olleet kuitenkaan merkittäviä, joten teknologia todettiin käyttökelpoiseksi yrityksessä.

Työ koostuu prototyypin kehityksestä ja sen vertailemisesta yrityksessä aiemmin käytössä olleeseen ratkaisuun. Työstä on rajattu pois serverless-teknologian käyttämisen kannattavuuden tutkiminen ja se keskittyy prototyypin kehittämisen lisäksi vain sen käytettävyyden validoimiseen. Työ toteutettiin 1.9.2021 – 31.3.2022.

Asiasanat
Serverless, Infrastruktuuri, Java, AWS, Virtualisointi

Sisällys

Käsitteet

API	Application Programming Interface. eli ohjelmointirajapinta, joka mahdollistaa kommunikoinnin ohjelmiston kanssa.
CaaS	CaaS = Containers as a Service, palvelu, joka tarjoaa erilaisia ympäristöjä oman ohjelmiston ajamiseen (Roberts & Chapin 2017, 5.)
FaaS	Functions as a Service, palvelu, jossa keskitytään toiminnallisuuden rakentamiseen yksittäisinä funktioina tai operaatioina (Roberts & Chapin 2017, 7–10.)
HA	Highly Availability, eli korkea saatavuus, on käytäntö, jolla voidaan määritellä, että palvelu tai järjestelmä on aina saatavilla
IaaS	Infrastructure as a Service, palvelu, joka tarjoaa fundamentaalisia resursseja ja mahdollisuudet verkollistamiseen
IaC	Infrastructure as Code, tapa, jossa määritellään palvelun tai sovelluksen infrastruktuuri ohjelmoimalla (Wittig & Wittig 2016, 93)
Instanssi	Instance, yksikkö resurssista, esimerkiksi yksi virtuaalikone tai kontti
PaaS	PaaS = Platform as a Service, palvelu, joka tarjoaa alustan, jolla on mahdollista julkaista käyttöönottoon kustomoituja applikaatioita
REST	Representational State Transfer, ohjelmistoarkkitehtuurin tyyli, jolla voidaan määritellä rajat, joiden puitteissa luodaan verkkopalvelu.
SaaS	Software as a Service, infrastruktuurin ja ohjelmiston yhdistelmä, joka toimii pilvipohjaisesti

1 Johdanto

Ohjelmistoala on jatkuvasti kasvava ja laajentuva ala, jonka vuoksi se myös muuttuu ja kehittyy kovalla vauhdilla. Alan nopeaan kehittymiseen ja kasvuun on jo pidemmän aikaa vaikuttanut digitalisaatio, jonka kautta digitaalitekniikka on yleistynyt jokapäiväisessä elämässämme. Digitalisaatiota on puolestaan kiihdyttänyt viime vuosina Covid-19 pandemia, joka on pakottanut useita yrityksiä nopeuttamaan siirtymää digitaalisiin työtapoihin.

Siirtymän nopeutuma on puolestaan johtanut uusiin tarpeisiin ohjelmistoalalla. Verkkoliikenteen ja sitä kautta myös liikkuvan datan määrä on kasvanut huimasti maailmalla, jonka vuoksi myös ICT:n aiheuttamat kulut ja vaatimukset ovat nousseet useissa yrityksissä. Uudet tarpeet ovat johtaneet uusien teknologioiden kehittämiseen, jotka tulevat ajan kanssa korvaamaan nykyiset vaihtoehdot. Yritysten onkin pitänyt pystyä ennakoimaan ja kehittymään alan mukana, sillä tänään laajasti käytetty teknologia ei välttämättä ole saatavilla enää lähitulevaisuudessa.

Opinnäytetyön toimeksiantaja on kansainvälisesti toimiva yritys, 10Duke Software Ltd (”yritys”), jolla on toimitilat Iso-Britanniassa ja Suomessa. Yritys on kokoluokaltaan PK-yritys ja se erikoistuu identiteettien hallintaan ja lisensointiin, joihin liittyvä liikevaihto on yhteensä yli miljardi euroa vuodessa.

Opinnäytetyön on tyyliltään toiminnallinen ja sen tavoitteena on toimeksiannon mukaisesti selvittää serverless-teknologian hyödyntämisen mahdollisuutta yrityksen ohjelmistojen julkaisu- ja käyttöönottoprosesseissa. Työssä kehitetään yritykselle ensimmäinen kyseistä teknologiaa käyttävä prototyyppi ja validoidaan teknologian käyttämisen mahdollisuus vertaamalla sitä jo olemassa olevaan ja nykyhetkellä käytössä olevaan ratkaisuun.

Opinnäytetyö pyrkii selvittämään serverless-teknologian käytettävyyttä osana yrityksen tarjoamia palveluita, eli selvittämään teknologian soveltuvuutta yrityksen käytettäväksi. Yrityksen kehitys on viime vuosina ollut vahvaa, mutta sen käyttämät julkaisu- ja käyttöönottoratkaisut ovat pysyneet pitkälti samanlaisina. Opinnäytetyöllä pyritään kokeilemaan vaihtoehtoista ratkaisua ja selvittämään sen käyttömahdollisuutta.

1.1 Opinnäytetyön rajaus

Opinnäytetyö keskittyy 10Duke Software Ltd:lle kehitettävään prototyyppiin ja tapoihin ja välineisiin, joita yrityksessä on aiemmin käytetty. Opinnäytetyössä luodaan yrityksen ensimmäinen prototyyppi serverless-teknologiaa käyttävästä julkaisu- ja käyttöönottoprosessista yrityksen aiempia tapoja mukaillen niin, että käytettäviä ohjelmistoja ei ole tarpeen muuttaa rakenteeltaan.

Työssä selvitetään serverless-teknologian soveltuvuutta yrityksen tarpeisiin prototyyppiä hyödyntämällä. Selvitys rajautuu olemassa olevan järjestelmän ja prototyypin vertailemiseen, eikä sisällä vaihtoehtoisten tapojen huomioonottamista. Vertailemisen tavoitteena on validoida serverless-teknologian käyttämisen mahdollisuus, eikä se esimerkiksi ota kantaa siihen, että onko kyseisen teknologian käyttäminen yritykselle kannattavaa.

2 Ohjelmistojen julkaisu ja käyttöönotto

Ohjelmistojen julkaiseminen (software deployment) voidaan määritellä prosesseina, jotka tapahtuvat ohjelmiston hankinnan ja suorittamisen välillä. Nämä prosessit suorittaa ohjelmistojulkaisija (software deployer), joka valmistelee ohjelmiston julkaisukelpoiseksi sen valmistumisen jälkeen ja aloittaa prosessin, jossa ohjelmisto saatetaan kohdeyleisön saataville käytettäväksi. (Dearle 2007.)

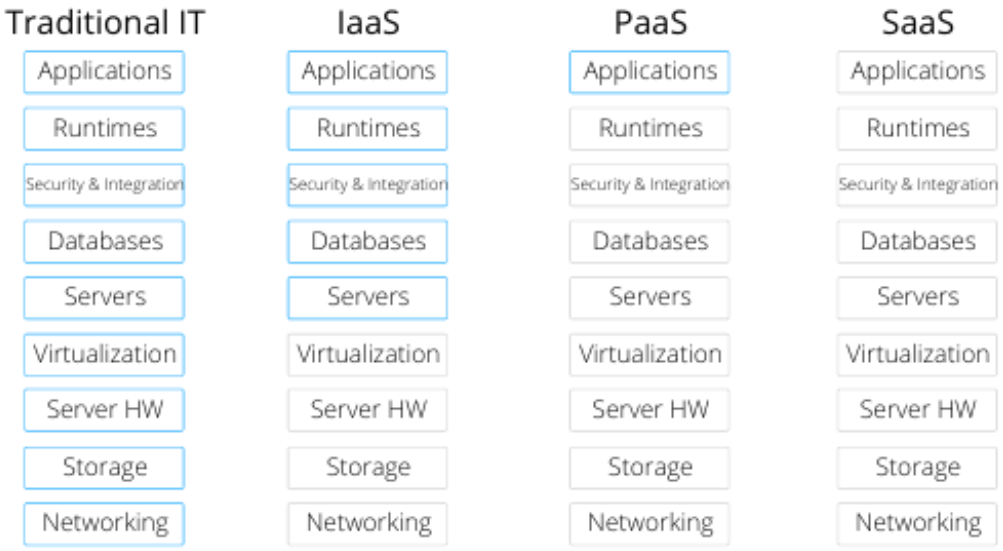
Tätä tukee myös Coupayen ja Estublierin (2000) määritelmä, jonka mukaan julkaisuprosessissa on kolme erillistä, toisiinsa sidonnaista prosessia. Prosessit puolestaan jakautuvat pienempiin aktiviteetteihin. Julkaiseminen jaetaan kolmeen prosessiin sen vuoksi, että niihin liittyvät aktiviteetit suoritetaan eri tahojen toimesta, eikä kyseisillä tahoilla ole yhteneväisiä tavoitteita, eikä vastuita. Prosessien kolme osapuolta ovat tuottaja, yritys ja käyttäjä. (Coupaye & Estublier 2000.)

Ohjelmiston julkaisemisen voi suorittaa monella tavalla ja se voi olla esimerkiksi automatisoitu prosessi, jossa luodaan tuore ympäristö ja johon siirretään kopio julkaistavasta ohjelmistosta niin, että se voidaan ottaa käyttöön. Vaihtoehtoisesti julkaisemisen voi tehdä myös täysin manuaalisesti esimerkiksi jo olemassa olevalle palvelimelle. Julkaiseminen voi olla myös yhdistelmä manuaalisia ja automatisoituja vaiheita.

2.1 Infrastruktuuri

Tietotekniikassa infrastruktuuri koostuu kolmesta eri komponentista. Infrastruktuuriin kuuluu laitteisto (hardware), ohjelmisto (software) ja verkko (network). (Redhat 2019a.) Jotta ohjelmisto voidaan tuoda kohdeyleisölle käytettäväksi, tarvitsee se tuekseen laitteiston ja verkon, joilla voidaan mahdollistaa jakelu. Hyvin suunniteltu infrastruktuuri on tietoturvallinen ja ohjelmiston vaatimusten mukaisesti optimoitu. Tietotekniikassa infrastruktuureja on olemassa kolme erilaista.

Infrastruktuuri vaikuttaa pitkälti myös siihen, mitä vastuualueita käyttäjälle jää (Mikovic, Lolić, Stefanović & Sladojevic 2017). Kuvassa 1 vertaillaan perinteisen infrastruktuurin mahdollistamaa mallia pilvi-infrastruktuurin tarjoamiin palvelumalleihin. Kuvassa sinisellä merkityt osa-alueet ovat käyttäjän vastuulla ja käyttäjän hallinnoimia ja harmaalla merkityt puolestaan palveluntarjoajan vastuulla ja palveluntarjoajan hallinnoimia. Infrastruktuurilla on siis suuri vaikutus siihen, mitä julkaisun jälkeiset toimenpiteet voivat vaatia.



Kuva 1. Perinteisen infrastruktuurin ja pilvi-infrastruktuurin palvelumallien vertailua (mukaillen Mikovic, Lolić, Stefanović & Sladojevic 2017)

2.1.1 Perinteinen infrastruktuuri

Perinteisessä infrastruktuurissa vastuu on pääasiassa käyttäjällä. Kaikki arkkitehtuurin eri osa-alueisiin liittyvät laitteistot ja komponentit ovat käyttäjän itse hallinnoimia ja omistamia. Perinteinen infrastruktuuri mielletään usein kalliiksi vaihtoehdoksi, sillä se vaatii suuren määrän laitteistoa, energiaa ja tilaa. (Red Hat 2019a.)

Perinteisen infrastruktuurin suurimmat haasteet ovatkin laitteiston omistamisessa. Mikäli esimerkiksi käyttäjien määrä kasvaa nopeasti, pitää olla jo etukäteen hankittuna laitteistot vanhojen tueksi, jotta palvelimia voidaan skaalata suuremmaksi (Pramanik 2021). Myös laitteistojen rikkoutumiseen tulee varautua etukäteen, sillä laitteiston hajoamisen kohdalla oleva palvelukatko saattaa venyä pitkäksikin, mikäli laitteistoja ei sillä hetkellä ole saatavilla.

Tämän lisäksi on hyvä ottaa huomioon, että kuten useimmissa käyttötavaroissa, myös palvelimissa sekä muissa tarvittavissa laitteistoissa arvo laskee ajan kuluessa ja käytön kasvaessa (Pramanik 2021). Tämä toimii myös merkittävänä erontekijänä pilvi-infrastruktuuriin nähden, sillä alaspäin skaalaaminen ei tuo käyttäjälle juurikaan säästöjä.

2.1.2 Pilvi-infrastruktuuri

Pilvipohjainen infrastruktuuri (cloud infrastructure) koostuu kaikista komponenteista, jotka vaaditaan siihen, että pilvipalvelut on mahdollista tuoda jakeluun kohdeyleisölle (Sumo Logic s.a.). Pilvi-infrastruktuurissa ei tarvitse omistaa käytettävää laitteistoa, vaan ne vuokrataan kolmannen

osapuolen pilvipalveluiden tuottajalta tarpeen vaatiessa (Red Hat 2019b). Tämä malli on usein käyttäjille huomattavasti kustannustehokkaampi, sillä resurssien käyttämisen voi optimoida tarpeiden mukaisiksi.

Pilvi-infrastruktuuri on mahdollistanut useiden eri käyttömallien kehittämisen, joissa käyttäjän vastuu ja tarve hallinnoida resursseja pienenee. Kuten kuvasta 1 voi nähdä, niin esimerkiksi IaaS (Infrastructure as a Service), jossa käyttäjä voi käytännössä vuokrata kokonaisen infrastruktuurin tai osan sitä, vähentää käyttäjän suorittaman hallinnoinnin tarvetta ja siirtää useiden palveluiden vastuun palveluntarjoajalle. Nykyään malleja on tarjolla useita, joista jokainen mahdollistaa käyttäjän fokuksen keskittämisen niille osa-alueille, joihin käyttäjä itse haluaa keskittyä. Tämä ei kuitenkaan sulje pois osa-alueisiin vaikuttamista halutessaan, vaan sen sijaan käyttäjälle tarjotaan valmiita malleja, joita voi käyttää, jos ei itse halua määritellä jotakin palvelua.

2.1.3 Hyperyhdennetty infrastruktuuri

Hyperyhdennetty infrastruktuuri (hyperconverged infrastructure) on ohjelmistoon keskittyvä arkkitehtuurityyli, jossa kaikki infrastruktuuriin kuuluvat komponentit on integroitu yhteen pakettiin hyödyntämällä virtualisointia. Koska komponentit on integroitu yhtenäiseksi paketiksi, ei niitä voi irrottaa toisistaan toisin kuin muissa infrastruktuureissa. Yhteen integroiminen kuitenkin mahdollistaa paremman skaalautuvuuden ja kyvyn selvitä suuremmistakin työmääristä (Red Hat 2019a).

2.2 Cloud computing

Cloud computing, eli pilvipohjainen tietojenkäsittely, tarkoittaa tietokoneresurssien ja – palveluiden jakelua käyttäjälle hyödyntämällä pilveä (Mikovic, Lolić, Stefanović & Sladojevic 2017). Pilvipohjainen tietojenkäsittely on mahdollistanut pilvipalveluiden tuomisen kaikkien saataville pilvipalveluntarjoajien kautta. Cloud computing on myös mahdollistanut pilvi-infrastruktuurien kehittämisen, sekä niihin liittyvien kehitysmallien luomisen.

Suurimpia etuja, mitä cloud computing on tuonut käyttäjille, ovat jatkuva ja laaja saatavuus. Pilvipohjaisissa ratkaisuissa resurssit ovat saatavilla aina tarpeen vaatiessa, eivätkä ne yleensä vaadi suurta erillistä pääomaa (Pramanik 2021). Pilvessä muutoksia voi tehdä tarpeen vaatiessa paikasta riippumatta, ja monia prosesseja on myös mahdollista automatisoida tarpeiden mukaisesti. Käyttäjille tarjotut resurssit ovat lähestulkoon rajattomat, eikä niiden käyttäminen vaadi palveluntarjoajan tukea, vaan niitä voi ottaa itse käyttöön.

2.3 Pilvisovellukset

Pilvisovellukset (cloud-native applications) ovat pienemmistä palveluista koostuvia kokonaisuuksia. Nämä palvelut ovat pääasiassa toisistaan erillisiä, mutta niiden välille on luotu jonkinlainen kytkös, jotta ne toimivat yhtenäisesti sovelluksessa. Pilvisovelluksien tuoman rakenteen avulla voidaan nopeuttaa julkaisu- ja käyttöönottoprosesseja, joka puolestaan mahdollistaa myös serverless-arkkitehtuurin. (Red Hat 2018.)

Pilvisovellusten suurin potentiaali onkin niiden itsenäisyydessä. Palveluita voidaan kehittää ja julkaista omina osinaan, jolloin ne eivät ole toisistaan riippuvaisia. (Klein 2019.) Näin voidaan tuoda esimerkiksi jokin yksittäinen ominaisuus asiakkaan käyttöön jo ennen kuin palvelu tai sovellus olisi valmis. Tätä kautta myös eri ominaisuuksien päivittäminen voisi vaikuttaa vain päivitettävään ominaisuuteen, eikä se välttämättä aiheuttaisi palvelukatkoa muiden ominaisuuksien osalta.

Ketterän kehitettävyytensä lisäksi pilvisovellukset ovat myös usein nopeasti skaalattavia, joka mahdollistaa niiden tarkemman optimoinnin. Skaalautuvuuden voi myös automatisoida, jolloin ei tarvitse huolehtia sovelluksen kaatumisesta liian suuren verkkoliikenteen vuoksi, eikä myöskään ennakoida tarvetta yhtä tarkasti etukäteen. Automatisointia kannattaa hyödyntää skaalauksen lisäksi myös palvelun terveydentilan seurannassa ja tarvittaessa myös terveydentilan korjaamisessa (Grey, 2019).

2.4 Serverless

Serverless on pilvipohjainen teknologia, jossa mahdollistetaan sovellusten kehittäminen, julkaiseminen ja ylläpito ilman, että kehittäjien tarvitsee hallinnoida palvelimia (Cloudflare s.a.). Tällöin voidaan tarkemmin keskittyä sovelluksen kehitystyöhön jo heti projektin alkuvaiheessa, eivätkä ylläpitoon liittyvät asiat myöskään rasita myöhemmissä vaiheissa (Tozzi 2021).

Vaikka teknologian suora suomennos tarkoittaakin palvelimetonta teknologiaa, ei tämä kuitenkaan tarkoita sitä, että serverless-mallissa ei käytettäisi lainkaan palvelimia. Käytännössä serverless voidaan mieltää vähemmän palvelimia käyttäväksi teknologiaksi. Serverless-mallissa vastuu ja työ palvelimien osalta on yleisimmin jätetty palveluntarjoajalle, kuten Amazon Web Services:ille, joka sitten vastaa ympäristön hallinnoinnista, ylläpidosta ja skaalauksesta (Red Hat 2018).

Ero siinä, että onko palvelu serverless-teknologiaa käyttävä vai ei, ei ole aina selkeä. Monet palvelut saattavat sisältää serverless-teknologiaa joissakin ominaisuuksissaan, mutta eivät välttämättä kokonaisuuksina toimi serverless-palveluina. Robertsin ja Chapinin (2017, 39–42.) mukaan serverless palvelun tulee täyttää seuraavat ehdot:

- Palvelu ei vaadi sovellusinstanssin tai pitkään elävän isäntäinstanssin hallinnointia

- Palvelu sisältää automaattisen skaalauksen kuormituksen mukaisesti
- Palvelun kustannukset perustuvat käyttöön ja kustannukset voivat olla olemattomat, mikäli käyttöä ei ole
- Palvelussa voidaan määritellä käytettävä kapasiteetti muuten kuin määrittelemällä isäntäinstanssien määrä
- Palvelulla on korkea saatavuus (HA)

Näitä ehtoja hyödyntämällä voidaan määritellä, mikä on serverless-teknologiaa käyttävä palvelu. On olemassa useita palveluita, joita voidaan kutsua serverless-palveluiksi, vaikka ne ovat vanhempia kuin itse termi serverless. Osa kehitysmalleista täyttää myös nämä ehdot poikkeuksetta, kuten FaaS, jossa käytetään funktioita eri asioiden tekemiseen. (Roberts & Chapin 2017, 42–44.) FaaS toimii periaatteella, jossa kutsumisvaiheessa luodaan kontti, jossa funktio ajaa ja ajamisen jälkeen kyseinen kontti tuhotaan. Näin palvelu ei kustanna mitään, mikäli funktioita ei käynnistä, kapasiteettia ei tarvitse määritellä instanssien määrässä ja palvelu skaalautuu kutsujen mukaisesti. Sen lisäksi palvelu on jatkuvasti saatavilla eikä se vaadi hallinnointia käyttäjältä.

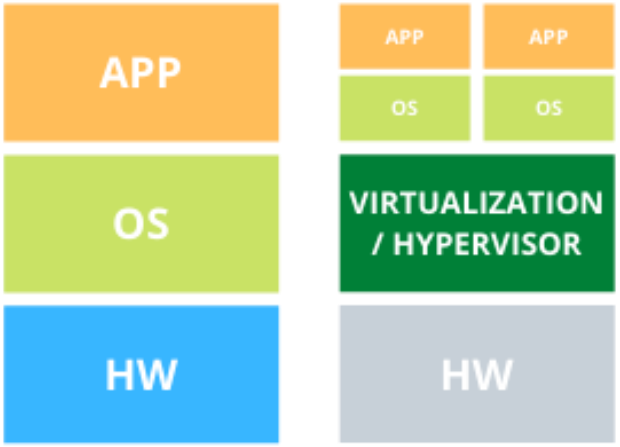
Useimmiten serverless-teknologian käyttäminen on kalliimpaa, kuin vastaavien resurssien käyttäminen perinteistä infrastruktuuria hyödyntäen pilvessä. Tämä siis tarkoittaa sitä, että yksittäinen resurssi, jota käytetään saman koodin ajamiseen yhtä kauan, on kalliimpi käyttää serverless-mallissa, kuin mitä se olisi perinteisessä mallissa. (Tozzi 2021.) Tämän vuoksi ympärivuorokautista käyttöä varten serverless ei välttämättä ole paras vaihtoehto.

3 Virtualisointi

Virtualisointi tarkoittaa keinotekoisen asian luomista sellaisella tavalla, että se toimii ja näyttää ulkopuolisesti luonnolliselta, niin että alusta, jolla virtualisointi on tapahtunut, ei näy ulospäin, eivätkä sen ominaisuudet ole havaittavissa virtualisoidun instanssin näkökulmasta. Virtualisointia hyödyntämällä voidaan jakaa yksi fyysinen resurssi useiksi pienemmiksi virtualisoiduiksi resursseiksi (kuva 2). (Hiltunen 2010, 3.)

Virtualisointia on monenlaista, eikä termillä aina tarkoiteta palvelimen virtualisointia, vaan kyseessä voi olla esimerkiksi paikallinen käyttöjärjestelmän virtualisointi jonkin ominaisuuden käyttöönottamiseksi (Oikarinen 2021, 10). Palvelinvirtualisoinnin lisäksi muita keskeisiä virtualisointiteknologian osa-alueita ovat tallennus-, verkko- ja sovellusvirtualisointi (Hiltunen 2010, 3). Palvelinvirtualisointi on kuitenkin näistä yleisin ja se on myös usein käytetty IT-infrastruktuuria rakentaessa.

Virtualisointi on tuonut mukanaan useita hyötyjä. Virtualisoinnilla laitteistosta on mahdollista saada enemmän irti, sillä sen avulla voidaan nostaa laitteiston suorituskapasiteettia 10–15 prosentista jopa 70–80 prosenttiin. Virtuaalipalvelimet myös helpottavat skaalaamista, sillä niiden loogisia resursseja on mahdollista muuttaa nopeasti ja uusia virtuaali-instansseja voi pystyttää nopeasti kopioimalla jo olemassa olevan instanssin. (Hiltunen 2010, 4–5.)



Kuva 2. Fyysisen palvelimen käyttö verrattuna virtuaalipalvelimeen

3.1 Virtuaalikone

Virtuaalikone (VM, virtual machine), tai virtuaalipalvelin, tarkoittaa yhtä virtuaali-instanssia, joka on sijoitettu fyysiseen koneeseen tai palvelimeen (Hiltunen 2010, 10). Ulkoisesti virtuaalikone näyttää normaalilta fyysiseltä koneelta, mutta sen ominaisuudet ovat virtuaalisesti muutettavia, eivätkä

välttämättä vaadi koneen uudelleenkäynnistämistä. Virtuaalikoneet ovat olleet saatavilla jo pitkään, ja niitä pidetäänkin pilvipohjaisen tietojenkäsittelyn ensimmäisenä versiona (IBM 2021).

Jotta virtuaalikoneita voidaan käyttää, niin tarvitaan väliin hypervisor, eli ohjelmisto, jolla voidaan luoda, poistaa, hallinnoida ja ajaa virtuaalikoneita. Hypervisor käsittelee fyysisiä resursseja, kuten CPU:ta ja muistia varastoina, joista on helppo jaella resursseja virtuaalikoneille. Hypervisor pitää huolen siitä, missä järjestyksessä ja miten virtuaalikoneet voivat resursseja käyttää. (Red Hat 2020a.)

3.2 Kontti

Kontti (container) on samankaltainen virtualisoinnin ratkaisu, kuin virtuaalikone, mutta se ei tarvitse toimiakseen hypervisoria. Ulkopuolisesti kontti saattaa vaikuttaa samanlaiselta, kuin virtuaalikone, mutta kontti ei välttämättä sisällä käyttöjärjestelmää samalla lailla, kuin virtuaalikone. Kontti sisältää tarvittavan koodin käyttöjärjestelmän osalta, mutta ei välttämättä kokonaista käyttöjärjestelmää. Kontin käyttäminen perustuukin siihen, ettei konttia kohdella kuin virtuaalikonetta, vaan sillä on tarkkaan määritelty tarkoitus, esimerkiksi yksittäisen ohjelmiston tai palvelun ajaminen. (IBM 2021.)

Koska kontti on huomattavasti kevyempi kokonaisuus kuin virtuaalikone, niin on se myös nopeampi ja kevyempi hallinnoida. Kontit toimivat erinomaisesti automatisoiduissa prosesseissa, sillä ne eivät kuormita paljoa isäntäinstanssia, ne ovat nopeita luoda ja tuhota. Myös kaiken konttiin liittyvän voi helposti määritellä etukäteen, jolloin kontti on luodessa samanlainen joka kerta. Tätä varten kontteja usein käytetäänkin hyödyksi esimerkiksi ohjelmistojen kehitysvaiheissa ja testaamisessa. (Red Hat 2020b.)

4 Amazon Web Services

Amazon Web Services (AWS) on perustettu vuonna 2002 osana Amazon.com:ia (Amazon 2002). AWS:n perustamiseen johtivat Amazon.com:in omien kehittäjien toiveet nopeamman infrastruktuurin luomisesta. Ennen AWS:n perustamista, uusien applikaatioiden kehittäjillä saattoi kulua vähintäänkin yhtä paljon aikaa oikeanlaisen infrastruktuurin pystyttämisessä kuin mitä applikaation rakentamisessa menisi (Long 2021).

Alkuvaiheessa AWS tarjosi kehittäjille mahdollisuuden luoda ja kehittää omia sovelluksiaan ja neljä vuotta AWS:n synnystä alettiin myös myydä palveluita B2B-myynnin kautta verkkopalveluina (Web Services). Verkkopalvelulla tarkoitetaan palvelua, jota on mahdollista kontrolloida verkkokäyttöliittymän kautta (Wittig & Wittig 2016, 3).

4.1 Hinnoittelu

Amazon Web Services toimii Pay-Per-Use periaatteella, joka tarkoittaa sitä, että kustannukset syntyvät käytön mukaan. Tämä on esimerkiksi verrattavissa sähkölaskuun, jossa maksat vain sen mitä käytät (Wittig & Wittig 2016, 13). Voit käynnistää palvelimen milloin tahansa tarvitset ja samalla lailla myös sulkea sen, kun tarve on ohitse. Hinnoittelussa on myös useampi vaihtoehto samoin kuin sähkössäkin. Valittavina on kiinteä hinnoittelu tai vaihtoehtoisesti sen hetkisen kysynnän mukainen hinnoittelu.

Tämänlaisen käytönmukaisen hintamallin vuoksi palveluiden ja infrastruktuurin optimointi onkin kannattavaa, sillä jo pelkällä skaalaamisella voi saavuttaa suuria säästöjä. Myös AWS:n tarjoamat palvelut tukevat hintapolitiikkansa puolesta optimointia. Esimerkiksi ominaisuuksiltaan kaksi kertaa parempi palvelin maksaa kaksi kertaa enemmän, kuin pienempi palvelin. Näin ollen voit valita haluamasi palvelimen pienimmän tarpeesi mukaan ja skaalauksen avulla lisätä niitä tarpeen kasvaessa. Käytettyjen palvelinten määrää ei myöskään rajoiteta, joten skaalaamisen pitäisi periaatteessa selvitä mistä tahansa kuormasta. (Wittig & Wittig 2016, 11).

4.2 Turvallisuus

4.2.1 Identity and Access Management

Identity and Access Management (IAM) on palvelu, jonka kautta hallinnoidaan kaikkia AWS:n sisäisten palveluiden roolituksia ja käyttöoikeuksia. Sen avulla voidaan esimerkiksi antaa käyttäjälle tai ohjelmalle pääsyoikeudet joihinkin tiettyihin resursseihin, sekä määritellä pääsyoikeuden laajuudet. (Amazon s.a.a)

IAM toimintaperiaate perustuu siihen, että ensin luodaan pääkäyttäjä (root user), jolla on pääsy kaikkiin tarjolla oleviin resursseihin. Sen jälkeen voidaan luoda käyttäjiä ja rooleja eri tarkoituksiin niin, että niille määritellään niiden tarvitsemia oikeuksia, eli käytäntöjä (policy) (Wittig & Wittig 2016, 159–160). Näin tilin resurssit ovat huomattavasti paremmin suojattuja, sillä kaikilla ei ole pääsyä kaikkiin resursseihin.

IAM-rooleja voidaan käyttää tarpeen mukaan myös asioiden automatisoinnissa. IAM:ää käyttämällä voidaan esimerkiksi myöntää rooli EC2-palvelimelle, joka puolestaan voi roolin oikeuksien puitteissa tämän jälkeen sammuttaa tai tuhota itsensä. Näin voidaan automatisoida resurssien poistamista tarpeen tullen rooleja hyödyntämällä. (Wittig & Wittig 2016, 163.)

4.2.2 Secrets Manager

Secrets Manager on AWS:n versio salaisuuksienhallintajärjestelmästä, jonka voi integroida käytettäväksi useimmissa muissa palveluntarjoajan järjestelmissä. Secrets Manager:in käyttö perustuu siihen, että sensitiivistä dataa voi tallentaa esimerkiksi avain-arvo-pareina suoraan AWS:ään ja niitä pystyy käyttämään eri palveluissa viitteitä hyödyntämällä. Secrets Manager:illa voi myös automatisoida salaisuuksien hallintaa joidenkin palveluiden suhteen, joka tarkoittaa sitä, että Secrets Manager:ista voi hakea esimerkiksi tietokannan käyttäjätunnuksen ja salasanan ilman, että sitä tarvitsee lisätä ja päivittää palvelussa. (Amazon s.a.b.)

4.2.3 Web Application Firewall

Web Application Firewall, eli WAF, toimii verkkosovelluksen palomuurina, jolla voi suojata käyttäjän julkaisemia sovelluksia ja API-palveluita. WAF:iin voi itse luoda kustomoituja sääntöjä, joilla suojaus tapahtuu, tai vaihtoehtoisesti WAF:issa pystyy myös ottamaan nopeasti käyttöön valmiita suojaussääntöjä, joilla voidaan estää esimerkiksi bottien pääsy ohjelmistoon. Sääntöjä pystyy lisäämään joustavasti myös käytön aikana, ja niiden käyttöönottamisessa ei kulu juurikaan aikaa. (Amazon s.a.c.)

Sen lisäksi, että WAF suojaa käyttäjän palveluita, tuo se myös mukanaan mahdollisuuden seurata sovelluksen saapuvaa liikennettä syvemmällä tasolla. Käyttäjä pystyy myös itse valitsemaan, millaista dataa halutaan kerätä ja mitä kyseinen data sisältää. WAF kerää dataa pyynnöistä aina reaaliajassa, joten sen avulla pystyy myös nopeasti löytämään mahdollisia uhkia jo heti käyttöönoton yhteydessä.

4.3 Tietokoneresurssit

4.3.1 Elastic Compute Cloud

Amazon Web Services julkaisi vuonna 2006 ensimmäisen IaaS-palvelunsa, Elastic Compute Cloud:in (EC2), joka mahdollisti tietokoneresurssien (compute resources) vuokraamisen ostamisen sijasta (Amazon 2006). Palvelu mahdollisti fyysisten palvelinten sijasta virtuaalisten palvelinten vuokraamisen, joita oli mahdollista skaalata tarpeen mukaan (Wittig & Wittig 2016, 35, 53).

EC2:n myötä IaaS-palvelut alkoivat yleistymään, sillä sen tuomat edut olivat selkeästi erotettavissa. Aiemmin käyttäjien tuli omistaa palvelimet, joilla ajaa verkkoon näkyviä applikaatioita. Tämä vaatisi rahallisten resurssien lisäksi myös enemmän henkilöstöresursseja, sillä käyttäjäyrityksellä tulisi olla joku, joka hallinnoi ja valvoo fyysisiä palvelimia esimerkiksi datakeskuksessa.

Kustannuksia saattoi myös syntyä palvelinten hajoamisesta, sillä niiden hallinnointi oli kokonaan yrityksen vastuulla. EC2 ensimmäisenä palveluna mahdollisti myös uuden palvelimen saamisen hajonneen tilalle minuuteissa, jolloin ongelman aiheuttama häiriöaika (downtime) ei välttämättä olisi merkittävä (Wittig & Wittig 2016, 11).

4.3.2 Elastic Beanstalk

Elastic Beanstalk (EB) on AWS:n vuonna 2011 perustama palvelu, jota voi käyttää verkkosovellusten julkaisemiseen ja skaalaukseen. Se on toiminnaltaan PaaS-palvelu, joka konkretisoituu monien valmiiksi määriteltyjen palveluiden muodossa. Tällainen on esimerkiksi ennalta määritelty EC2-palvelin. Vaikka useimmat Elastic Beanstalk:in tarjoamat palvelut ovat valmiiksi määriteltyjä, niin niiden määrityksiä voi muokata vapaasti. (GeeksforGeeks 2021.)

Elastic Beanstalk tarjoaa käyttäjälle ympäristöjen hyödyntämisen AWS palveluiden muodossa. Sen avulla voi luoda applikaatioita, joilla on useita eri ympäristöjä esimerkiksi kehitys- ja tuontantotarkoituksiin. Ympäristöjen tilaa voi seurata kokonaisuuksina, jolloin käyttäjille näkyvät virheet tulevat helposti esille. Esimerkiksi mikäli käyttäjä ei saa yhteyttä kyseiseen palveluun, voi se näkyä nousevana virheellisten HTTP-statuskoodien muodossa.

Suurimpiin Elastic Beanstalk:in tarjoamiin hyötyihin ja etuihin lukeutuu sen skaalautuvuus. Elastic Beanstalk:in voi asettaa automaattisesti skaalautuvaksi, jolloin se esimerkiksi lisää ja vähentää tarpeen mukaan EC2-palvelimia. Skaalaamisen hoitaa Elastic Beanstalk:iin sisältyvä Elastic Load Balancing -palvelu, joka jakaa saapuvan liikenteen tasaisesti palvelinten välille. Amazon Web Services:in omien sanojen mukaan skaalattavuudella ei ole myöskään maksimaalista rajaa, joten

autoskaalauksen pitäisi pystyä pitämään sovellus ylhäällä liikenteen määrästä riippumatta. (Wittig & Wittig 2016, 11.)

4.4 Kontit

4.4.1 Elastic Container Registry

Elastic Container Registry (ECR) on nimensä mukaisesti AWS:n tarjoama rekisteri kuville, joita voidaan käyttää konteissa. Säilömisen yhteydessä ECR mahdollistaa kuvien versionhallinnan, sekä eliniän hallinnoinnin. ECR:n kautta pystyy myös jakamaan kuviaan HTTPS-yhteyden kautta, mutta sen pääasiallinen hyöty tulee esiin, kun sitä käytetään muiden AWS:n kontteja käyttävien palveluiden kanssa. (Amazon s.a.d.)

Palvelu on myös yhteensopiva useimpien yleisten kontinhallintavälineiden kanssa, joka helpottaa sen hyödyntämistä eri prosesseissa, kuten ohjelmistojen julkaisemisessa ja testaamisessa. ECR salaa (encrypt) kaikki kuvat niiden tallettamisen yhteydessä, jonka seurauksena ne saavat yhden suojakerroksen lisää säilömisen yhteydessä. Suojaamista voi myös edistää entisestään käyttämällä IAM-rooleja niiden jakelussa. (Carty 2021.)

4.4.2 Elastic Container Service

Elastic Container Service (ECS) on AWS:n tarjoama konttien hallintapalvelu, eli CaaS-palvelu (Containers as a Service), joka on laajalti skaalattava ja nopea. ECS tarjoaa konttien käyttämiseen kahta eri ratkaisua, serverless-teknologian mahdollistavaa Fargate:a ja virtuaalikoneita käyttävää EC2:ta. Molemmissa tapauksissa käytetään kontteja, jotka määritellään ECS:ssä tehtävinä (task), joita voidaan hallita ryhmissä (cluster). Kontit puolestaan määritellään Docker:ia käyttämällä. (Amazon s.a.e.)

ECS kehitettiin alun perin EC2-instansseja varten silloin, kun konttien suosio alkoi nousemaan. Palvelu mahdollisti joustavamman lähestymistavan ohjelmistojen ajamisessa, sillä se oli paremmin skaalattavissa kuin esimerkiksi Elastic Beanstalk ja se mahdollisti uusien kehitysmallien käyttämisen ohjelmistojen luomisessa (Carty 2017). Palvelu myös vähensi huomattavasti ylläpidon tarvetta automatisoimalla eri prosesseja esimerkiksi instanssien luomiseen ja tuhoamiseen liittyen.

4.4.3 Fargate

Fargate on toinen ECS:n tarjoamista ratkaisuista, joka mahdollistaa konttien käyttämisen serverless-teknologiaa hyödyntämällä. Toisin kuin EC2:ssa, Fargate ei vaadi virtuaalikoneiden hallintaa, eikä ryhmien skaalaamista tai määrittelyä. Tällöin sinun ei periaatteessa tarvitse määritellä mitään muuta, kuin haluamasi käyttöjärjestelmän, vCPU:n ja muistin määrät ja verkon, jossa palvelu toimii. Näiden

lisäksi voi myös asettaa esimerkiksi autoskaalaukseen liittyviä asioita, mutta ne eivät ole välttämättömiä kontin ja sovelluksen käynnistämisessä. (Amazon s.a.f.)

Fargate:ssa voit määritellä sovelluksesi ajettavaksi yhdessä tehtävässä, tai vaihtoehtoisesti voit jakaa sovelluksen useisiin osiin mikropalvelun tavoin ja ajaa sitä useissa eri tehtävissä. Fargate:ssa voit myös määritellä tehtäville omat terveystarkastuksensa, joilla voi automatisoida palvelun ylläpitoa, sekä parantaa sen reagointikykyä ongelmatilanteissa. Fargate:n tarjoamat konttiratkaisut ovat ECS:n mukaisesti nopeita luomaan tarvittaessa uusia kontteja ja tuhoamaan vanhoja. Palvelussa on mahdollista käynnistää jopa tuhansia kontteja samanaikaisesti (Culverhouse 2018), mikä mahdollistaa skaalaamisen myös suuremassa mittakaavassa.

4.5 Työkalut

4.5.1 Command Line Interface

AWS Command Line Interface (CLI), eli komentoliittymä, on työväline, jolla voi hallinnoida AWS:n palveluita ja resursseja komentorivin kautta. Komentoliittymä on hyvä vaihtoehto erityisesti silloin, kun pitää luoda automatisoituja prosesseja, joissa suoritetaan useita eri infrastruktuuriin liittyviä tehtäviä. Prosessien automatisointi sujuu helposti esimerkiksi kirjoittamalla skriptejä, joissa on useita eri komentoja peräkkäin. (Amazon s.a.g.)

4.5.2 CloudFormation

CloudFormation (CF) on työkalu, jolla voidaan hallinnoida palveluita ja resursseja IaC-periaatteella (Amazon s.a.h). CF mahdollistaa infrastruktuurin hallinnan täyden automatisoinnin, sillä kaikki AWS:n resurssit on mahdollista määritellä sitä kautta. CloudFormation:in avulla infrastruktuureja on myös helppo kopioida, sillä samaa pohjaa voi käyttää uudelleen rajattomasti. Isona etuna on myös se, että kopioiminen on mahdollista eri alueiden välillä, eli ympäristön asetukset voi esimerkiksi kopioida Irlannista Yhdysvaltoihin. (Amazon s.a.i.)

CloudFormation:in käyttö perustuu pinoihin (stack), joihin määritellään kaikki tarvittavat palvelut ja resurssit, sekä niiden asetukset. Palveluiden ei kuitenkaan tarvitse olla yhdessä käytettyjä, vaan ne voi määritellä täysin vapaasti kaikista AWS:sä olevista palveluista. Pinojen käyttö mahdollistaa kokonaisten resurssiryhmien hallinnan samanaikaisesti, eli niitä voi luoda, muokata ja poistaa kokonaisuuksina (Amazon s.a.i). Tämä helpottaa esimerkiksi kehitysympäristön ja tuotantoympäristön eriyttämistä toisistaan.

CF toimii tekstitiedostoilla, malleilla (template), jotka määritellään joko JSON tai YAML formaattia käyttämällä. Formaatista riippuen myös syntaksi on hieman eroavaa näiden välillä. Formaatti ei

kuitenkaan vaikuta siihen, mitä kaikkea on mahdollista määritellä CF malleihin. Malleja voi luoda ja muokata lähes millä tahansa tekstieditorilla, tai niitä voi myös kirjoittaa käyttämällä Amazonin koodipohjaista kehitystyökalua, Cloud Development Kit:iä.

4.5.3 Cloud Development Kit

AWS Cloud Development Kit (CDK) on IaC periaatteella toimiva kehitystyökalu, jonka avulla käyttäjä voi määritellä AWS:ään liittyviä palveluita ohjelmoimalla. CDK on kehitetty käytettäväksi useimmille moderneille ja yleisille ohjelmointikielille, ja sen tarkoituksena on mahdollistaa ympäristön kehittäminen samalla kielellä, kuin itse ohjelmisto tai palvelu on kehitetty. CDK toimii käännöstyökaluna, joka luo CloudFormation malleja kirjoitetun koodin pohjalta. (Amazon s.a.j.)

CDK:ta käyttämällä voi säästää huomattavasti aikaa ja vaivaa verrattuna pelkän CF:n käyttämiseen. Esimerkiksi kuvan 3 mukaisesti luotu pino kääntyy yli 500 rivin YAML-malliksi huomattavasti pienemmällä koodimäärällä (Amazon s.a.j). CDK koodin kirjoittaminen totutulla IDE:llä on myös huomattavasti käyttäjäystävällisempää, kuin tekstieditorin käyttäminen YAML-koodin kirjoittamiseksi. CDK:n käyttämistä varten tarvitaan eri palveluiden vaatimia kirjastoja, jotka tukevat koodin kirjoittamista ja näin vähentävät myös mahdollisten virheiden määrää.

```
public class MyEcsConstructStack extends Stack {

    public MyEcsConstructStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyEcsConstructStack(final Construct scope, final String id,
            StackProps props) {
        super(scope, id, props);

        Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                .vpc(vpc).build();

        ApplicationLoadBalancedFargateService.Builder.create(this, "MyFargateService")
                .cluster(cluster)
                .cpu(512)
                .desiredCount(6)
                .taskImageOptions(
                       ApplicationLoadBalancedTaskImageOptions.builder()
                               .image(ContainerImage
                                       .fromRegistry("amazon/amazon-ecs-sample"))
                               .build()).memoryLimitMiB(2048)
                .publicLoadBalancer(true).build();
    }
}
```

Kuva 3. Pinon luominen CDK:ta käyttämällä (Amazon s.a.j)

5 Kehityssuunnitelma

Toimeksiannon mukaisesti suunnitelmana on luoda 10Duke:n ensimmäinen prototyyppi sovelluksen käyttöönotosta, jossa käytetään serverless-teknologiaa, ja sen jälkeen selvittää sen mahdollisuutta käyttöönottamisen suhteen yrityksessä. Prototyypin kehittäminen pohjautuu yrityksessä jo oleviin menetelmiin, tekniikoihin ja teknologioihin, jotta se olisi mahdollisimman helposti käyttöönotettava myös muissa yrityksen projekteissa. Yrityksen intressit näkyvät myös tutkimusmenetelmissä, joissa keskitytään yrityksen tarpeiden mukaisesti vertailuun uuden ja vanhan teknologian välillä.

5.1 Prototyyppi

Prototyypin tavoitteena on olla 10Duke:n ensimmäinen kokeilu serverless-teknologian hyödyntämisestä yrityksen perustoiminnoissa. Mikäli kokeilu on onnistunut ja teknologia osoittautuu kannattavaksi, niin voidaan yrityksen muitakin palveluita tarpeiden mukaisesti siirtää ylläpidettäväksi serverless-ympäristöihin. Nämä tavoitteet toimivat osittain pohjana myös toimeksiannossa annetuille vaatimuksille.

Toimeksiannon mukaisesti prototyypin vaatimukset olivat seuraavanlaiset:

- Prototyyppi pohjautuu AWS:n Fargate-palveluun
- Prototyyppi rakennetaan AWS CDK:ta käyttämällä
- Prototyyppi kirjoitetaan Java-ohjelmointikielellä
- Prototyypin asetukset voi määritellä Typesafe-kirjastoa käyttämällä

Prototyypin kehittämisessä käytetyt teknologiat olivat siis pitkälti ennalta määriteltyjä. Jotta Fargate-palvelua voidaan tukea, niin pitää prototyyppiin rakentaa tuki myös sen vaatimia AWS-palveluita varten. Koska tavoitteena oli kuitenkin keskittyä Fargate-puoleen, sekä sen vertailemiseen olemassa olevan vaihtoehdon kanssa, niin päätettiin ettei jaettavissa olevien resurssien luomista erikseen lisätä prototyyppiin ensimmäisessä kehitysvaiheessa. Sen sijaan päädyttiin ratkaisuun, jossa luodaan testiympäristö, jossa jo olemassa oleva ratkaisu jakaa osan resursseistaan Fargate:n käyttöön. Näin molemmissa tavoissa voitiin olla varmoja siitä, että palveluiden käytössä olevat resurssit ovat mahdollisimman identtisiä keskenään.

5.2 Prototyypin käytettävyyden validointi

Osana toimeksiantoa yrityksen jo ennalta olemassa olevaa ohjelmistojen julkaisu- ja käyttöönottoprosessia vertaillaan serverless-teknologiaan pohjautuvaan julkaisu- ja käyttöönottoprosessiin. Näin pyritään validoimaan serverless-teknologian soveltuvuus yrityksen palveluihin. Vertaileminen keskittyy neljään pääkategoriaan ja sen tarkoituksena on validoida serverless-teknologian käyttämisen mahdollisuus. Kaikissa neljässä pääkategoriassa on omat

alakategoriansa, joiden mukaisesti tehdään arvio teknologian käyttämisen mahdollisuudesta. Ympäristöjen vertaileminen eri kategorioissa mahdollistaa mahdollisimmin laajan ja samanaikaisesti koko julkaisu- ja käyttöönottoprosessin kattavan analysoinnin.

Opinnäytetyön vertailu keskittyy yrityksen intressien mukaisesti neljään pääkategoriaan:

- Työn määrä
- Julkaiseminen ja ylläpito
- Suorituskyky
- Kustannukset

Työn määrän vertailemisessa käydään läpi vaadittavaa koodimäärää, sekä sen kompleksisuutta. Vertailtavina kohteina toimivat jo olemassa oleva koodi ja opinnäytetyössä luotava prototyyppi, jotka molemmat käyttävät AWS:n CDK-kirjastoja ympäristöjen määrittelemisessä ja Shell-skriptausta prosessien helpottamisessa ja osittaisessa automatisoinnissa.

Julkaiseminen ja ylläpito pitää sisällään prosessit, jotka suoritetaan ohjelmistoa tai ympäristöä käyttöönottaessa. Julkaisemisessa käsitellään siis tarvittavia toimenpiteitä, joilla voidaan luoda tai päivittää ympäristöä, sekä sen sisältämää ohjelmistoa. Ylläpito puolestaan sivuaa myös myöhemmin käsiteltävää skaalautuvuutta ja sen lisäksi myös selvittää yleisimpiä toimenpiteitä, joita ympäristöt saattavat vaatia ylläpitäjältään.

Suorituskyvyn vertailemisessa keskitytään erityisesti siihen, kuinka ympäristöt kykenevät suoriutumaan erinäisissä simuloiduissa käyttötilanteissa. Simuloiduilla tilanteilla pyritään selvittämään ympäristöjen tarjoamia suoriutumisnopeuksia, sekä kokeilemaan ympäristöjen rasituksensietokykyä ja kovasta rasituksesta selviytymistä esimerkiksi automaattisella skaalautumisella.

Viimeisimpänä pääkategoriana on kustannusten vertaileminen ja arviointi pidemmällä käytöllä. Kustannuksia pyritään mittaamaan lyhyellä aikavälillä erinäisissä käyttötilanteissa ja mittausten pohjalta tullaan arvioimaan myös mahdollisia kulueroja pidempiaikaisessa käytössä.

6 Prototyypin kehittäminen

6.1 Valmistelut

Prototyypin kehittämiseen käytettäväksi työkaluksi valikoitui Apache:n kehittämä NetBeans, sillä se on yrityksessä käytetyin ohjelmointiympäristö (integrated development environment, IDE). Käyttömäärän mukaan valikoitui myös ohjelmointikieleksi Java, jolla myös esimerkiksi toimeksiannossa annettu ohjelmisto on kirjoitettu.

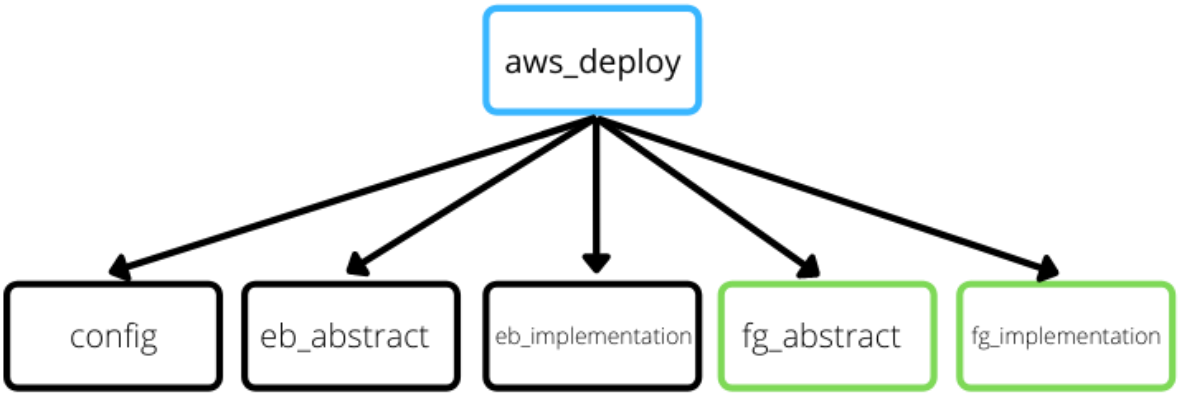
Toimeksiannon mukaisesti käytettävässä ohjelmistossa on jo entuudestaan moduuli, jossa tapahtuu Amazon Web Services:in Cloud Development Kit:llä automatisoitu julkaisu ja käyttöönotto Elastic Beanstalk:ia käyttämällä. Tässä moduulissa on alamoduuleina luotu omat tasonsa erinäisiin tarpeisiin, esimerkiksi määritysten asettamiseen, ympäristön pystyttämiseen ja ohjelmiston julkaisemiseen. Jotta ohjelmiston rakennetta ei tarvitse muuttaa, niin loin kaksi uutta alamoduulia nykyisten rinnalle moduuliin, jossa CDK julkaisu Elastic Beanstalk:iin on tehty.

Näissä uusissa moduuleissa tapahtuu varsinainen prototyypin kehittäminen ohjelmoinnin ja asetusten määrittelyn muodossa. Uusien moduulien luominen rinnalle mahdollisti myös nykyisen rakenteen mukaisten riippuvuuksien hyödyntämisen ohjelmoidessa, joten rakenteesta voidaan tehdä samankaltainen ja jo olemassa olevaa koodia voidaan hyödyntää helposti ja ketterästi.

6.2 Rakenne

Prototyypin rakenteen suunnittelu alkoi Fargate-palvelun vaadittujen riippuvuuksien selvittämisellä, jotta voitiin määritellä tarvittavat komponentit, sekä niihin liittyvät määriteltävät arvot. Tämän lisäksi piti selvittää nykyisen asetusten määrittelyn toimintaperiaatetta, jotta prototyyppi olisi rakenteeltaan mahdollisimman samankaltainen ja näin ollen myös helposti ymmärrettävä ja käytettävä.

Moduulin aws_deploy rakenne muotoutui kuvan 4 mukaiseksi, kun lisättiin uudet vihreällä merkityt alamoduulit. Kuvassa lyhenteellä ”eb” viitataan jo olemassa oleviin Elastic Beanstalk:ia käyttäviin alamoduuleihin ja lyhenteellä ”fg” viitataan uusiin, Fargate:a käyttäviin alamoduuleihin. Näiden ohella oli yhteinen pohjatason asetusten määritelmä, joka sijaitsi alamoduulissa config ja josta muut moduulit perivät niille tarpeelliset luokat.



Kuva 4. Moduulirakenne

Luotujen alamoduulien rakenne määräytyi yhtä lailla jo olemassa olevan koodin mukaisiksi. Sekä abstraktiotason (fg_abstract) että implementaatiotason (fg_implementation) moduulit sisälsivät samankaltaisen rakenteen, jossa on luokat asetusten määrittelemiselle (config) ja pinolle (stack). Config-luokka toimii implementaationa Typesafe-määrityksiä lukevasta alaluokasta, joka peritään config-moduulista. Pinoluokka on puolestaan implementaatio AWS CDK:n tarjoamasta luokasta Stack.java, joka toimii pohjana resurssikokonaisuudelle, joka voidaan myöhemmin julkaista AWS:ään.

6.3 Toteutus

6.3.1 Palvelun määrittely

Prototyypin toteuttaminen alkoi tarvittavien resurssien lisäämisellä, sekä niiden ottamien parametrien läpikäymisellä. Halusin prototyypissä ottaa mahdollisimman paljon hyötyä irti Amazon Web Services:in tarjoamasta Cloud Development Kit-kirjastosta, jotta prototyyppiä olisi mielekkäämpi työstää. Mielekkyyden lisäksi prototyyppi ei toimisi tällöin vain tekstin kääntäjänä työvälineiden välissä, vaan myös sen automatisoituja prosesseja olisi mahdollista hyödyntää.

Valinta osoittautui myös nopeasti hyväksi ja hyödylliseksi, sillä heti ensimmäisissä tarvittavissa luokissa ilmeni selkeitä hyötyjä. CDK:n tarjoama ECS-kirjasto sisälsi luokan ApplicationLoadBalancedFargateService.class, jonka parametrina sisään ottama luokka ApplicationLoadBalancedFargateServiceProps.class oli monipuolisesti määriteltävä. Kyseinen luokka tarjosi useita eri metodeja (kuva 5), joilla oli mahdollista esimerkiksi luoda Fargate-palvelu, jolla on oma automaattisesti asetettu kuorman tasaaja (load balancer) ja ryhmä (cluster), johon se kuuluu. Näiden lisäksi oli myös mahdollista määritellä aliverkot (subnet) ja turvaryhmät (security

group) joissa se toimii, sekä määritellä palvelun terveystarkastuksiin liittyvät arvot ja asettaa sen DNS-asetukset kohdilleen.

```
ApplicationLoadBalancedFargateServiceProps props = ApplicationLoadBalancedFargateServiceProps
        .builder()
        .serviceName(serviceName)
        .cluster(cluster)
        .cpu(getAppStackConfig().fargateCpu())
        .memoryLimitMiB(getAppStackConfig().fargateMemory())
        .desiredCount(getAppStackConfig().fargateCount())
        .taskImageOptions(imageOptions)
        .publicLoadBalancer(true)
        .assignPublicIp(false)
        .securityGroups(Arrays.asList(
                SecurityGroup.fromSecurityGroupId(this, "mercury-sg", getAppStackConfig().awsVpcWebSecurityGroupId())
        ))
        .taskSubnets(subnetSelection)
        .redirectHttp(true)
        .certificate(certificate)
        .domainName(getAppStackConfig().domainName())
        .domainZone(hostedZone)
        .healthCheckGracePeriod(Duration.minutes(5))
        .build();
```

Kuva 5. ApplicationLoadBalancedFargateServiceProps.java

Vaikka valikoima on kattava ja sisältää kaiken aiemmin mainitun lisäksi myös kohdat tarpeellisille määriteltäville arvoille, niin samalla se rajoittaa itseään. Esimerkiksi DNS-asetusten asettaminen on suuri etu verrattuna siihen, että käyttäjän pitää ladata Route 53:n sisältämän kirjaston moduuliin ja asettaa sitä kautta tarvittavat arvot. Samalla ApplicationLoadBalancedFargateServiceProps-luokka kuitenkin rajoittaa verkkotunnusten määrää, joihin palvelu voidaan liittää. Tämän opinnäytetyön kannalta kyseinen asia ei ole haitallinen, mutta tulevaisuudessa se voi vaatia lisäkehitystä kyseisen kohdan osalta.

Koska toimeksiannossa haluttiin keskittyä prototyypin ja olemassa olevan ratkaisun vertailemiseen ja prototyypin käytettävyyden validointiin, niin tehtiin samalla päätös, että osa resursseista jaettaisiin näiden kahden version välillä. Jaettavia resursseja olivat esimerkiksi virtuaalinen yksityinen pilvi (VPC, virtual private cloud), siihen liittyvät resurssit, kuten aliverkot ja turvaryhmät, sekä tietokanta. Näiden jakaminen ympäristöjen välillä ei aiheuta haittaa vertailemisen näkökulmasta myöhemmin, mutta tällä tavalla voitiin minimoida kustannuksia ja varmistaa yhteneväiset ympäristön resurssit. Kuvasta 5 on mahdollista nähdä, kuinka esimerkiksi olemassa oleva turvaryhmä on haettu sen tunnisteen (id) perusteella.

6.3.2 Ympäristön määrittely

Fargate-palvelun luomisen ja määrittelyn lisäksi yhtenä tärkeänä kohtana on myös ympäristön määrittely ohjelmistoa varten. Koska Fargate pohjautuu konttien ja Docker-kuvien käyttämiseen, on ympäristöä varten luotava Elastic Container Registry, johon voidaan päivittää oikea kuva

viimeisimmästä ohjelmistoversiosta. ECR tarjoaa näin paikan kuvien säilyttämiselle ja versiohallinnalle, josta Fargate voi hakea tarvitsemansa kuvan ympäristöjen päivittämisen yhteydessä. Koska Docker-kuvan päivittäminen on erillinen osa, eikä liity ympäristön päivittämiseen, niin kuvan lataaminen ECR:ään jätettiin CDK-koodista pois. Kuvan lataaminen ECR:stä on kuitenkin oleellinen osa ympäristön päivittämistä, joten se otetaan mukaan pinoon.

Toisin kuin Elastic Beanstalk:issa, Fargate:en on luotu tuki AWS:n Secrets Manager:ia varten. Tämä helpottaa huomattavasti ympäristön määrittelyä ohjelmistoa varten. Käytännössä asetusten määrittely siis vaatisi CDK-tasolla yhteyden luomista Secrets Manager:iin, josta voidaan hakea tarvitut etukäteen asetetut arvot. Tämän lisäksi pitää hakea haluttu Docker-kuva ECR:ästä ja määritellä sille halutut salaisuudet Secrets Manager -viittauksilla (kuva 6).

```
protected ApplicationLoadBalancedTaskImageOptions configureImageOptions() {
    //
    IRepository imageRepository = Repository.fromRepositoryArn(
            this,
            getAppStackConfig().imageRepositoryId(),
            getAppStackConfig().imageRepositoryArn()
    );
    ISecret secret = software.amazon.awscdk.services.secretsmanager.Secret
            .fromSecretNameV2(this, "mercury-secret-manager", getAppStackConfig().secretsManagerName());
    ApplicationLoadBalancedTaskImageOptions imageOptions = ApplicationLoadBalancedTaskImageOptions
            .builder()
            .image(ContainerImage.fromEcrRepository(imageRepository))
            .containerPort(getAppStackConfig().imageContainerPort())
            .secrets(new HashMap<String, Secret>() {
                {
                    put("DB_USER", Secret.fromSecretsManager(secret, "username"));
                    put("DB_PASSWD", Secret.fromSecretsManager(secret, "password"));
                    put("DB_URL", Secret.fromSecretsManager(secret, "connectionUrl"));
                    put("DB_NAME", Secret.fromSecretsManager(secret, "dbname"));
                }
            })
            .build();
    return imageOptions;
}
```

Kuva 6. Yhteys ECR:ään ja Secrets Manager:iin

6.3.3 Terveystarkastusten määrittely

Jotta Fargate olisi yhteensopiva julkaistavan ohjelmiston kanssa, niin on myös muokattava automatisoituja terveystarkastuksia. Alun perin määrittelin ensimmäisen terveystarkastuksen tapahtuvan 5 minuuttia ohjelmiston käynnistymisen jälkeen (kuva 5), jotta voidaan varmistaa, että ohjelmisto on ehtinyt käynnistyä kokonaan. Tämän jälkeen määrittelin myös sopivamman polun ja hyväksytyn HTTP-statuskoodin (kuva 7), joita käyttämällä voidaan todentaa, että ohjelmisto on toiminnassa ja toimii oikein.

```
service.getTargetGroup().configureHealthCheck(HealthCheck
        .builder()
        .path(getAppStackConfig().healthCheckPath())
        .healthyHttpCodes("200")
        .port(String.valueOf(getAppStackConfig().imageContainerPort()))
        .build()
);
```

Kuva 7. Terveystarkastusten määrittely

6.3.4 Suojauksen määrittely

Seuraavana vaiheena oli lisätä suojaus, jolla voidaan hallita sisään tulevaa liikennettä, sekä määritellä linkit, jotka ovat ohjelmistossa julkisesti avoinna. Toimeksiantaja toivoi myös suojausta esimerkiksi anonyymejä IP-osoitteita vastaan, jotta voidaan minimoida ylimääräinen liikenne, joka voisi rasittaa ohjelmistoa. Suojauksesta on myös hyötyä myöhemmässä vertailemisvaiheessa, sillä ulkopuolelta mahdollisesti tulevia pyyntöjä saadaan karsittua, eivätkä ne silloin voi rasittaa palvelua.

Suojaus pohjautuu AWS WAF:iin, eli Amazon Web Services:in tarjoamaan verkkosovelluspalomuuriin (web application firewall). Palomuuria varten tuli lisätä web pääsynhallintalista (ACL, access control list), johon määriteltiin erinäisiä sääntöjä pääsyn rajoittamiseksi (kuva 8) ja lokien keräämiseksi (kuva 9), jotta mahdolliset uhkat on myöhemmin helpompi tunnistaa. Kuvasta 8 on myös mahdollista nähdä miten AWS:n ylläpitämiä sääntöjä voidaan helposti hyödyntää ohjelmiston suojauksessa.

```
CfnWebACL.DefaultActionProperty defaultAction = CfnWebACL.DefaultActionProperty
        .builder()
        .block(CfnWebACL.BlockActionProperty.builder().build())
        .build();

List<CfnWebACL.RuleProperty> rules = new ArrayList<>();
rules.add(CfnWebACL.RuleProperty
        .builder()
        .name(applicationName + "-BlockAnonIPsRule")
        .priority(1)
        .overrideAction(CfnWebACL.OverrideActionProperty
                .builder()
                .none(new HashMap<String, Object>())
                .build()
        )
        .visibilityConfig(CfnWebACL.VisibilityConfigProperty
                .builder()
                .sampledRequestsEnabled(true)
                .cloudWatchMetricsEnabled(true)
                .metricName(applicationName + "-BlockAnonIPsRule")
                .build()
        )
        .statement(CfnWebACL.StatementProperty
                .builder()
                .managedRuleGroupStatement(CfnWebACL.ManagedRuleGroupStatementProperty
                        .builder()
                        .vendorName("AWS")
                        .name("AWSManagedRulesAnonymousIpList")
                        .excludedRules(Collections.singletonList(CfnWebACL.ExcludedRuleProperty
                                .builder()
                                .name("HostingProviderIPList")
                                .build()
                        ))
                        .build()
                )
                .build()
        )
        .build()
);
```

Kuva 8. Anonyymien IP-osoitteiden eston määrittely web pääsynhallintalistalle

```
CfnLoggingConfiguration.Builder
        .create(this, "mercury-logging-config")
        .resourceArn(webAcl.getAttrArn())
        .logDestinationConfigs(Arrays.asList(stream.getAttrArn()))
        .loggingFilter(new HashMap<String, Object>() {
            {
                put("DefaultBehavior", "DROP");
                put("Filters", Arrays.asList(new HashMap<String, Object>() {{
                    put("Behavior", "KEEP");
                    put("Conditions", Arrays.asList(new HashMap<String, Object>() {
                        {
                            put("ActionCondition", new HashMap<String, Object>() {
                                {
                                    put("Action", "BLOCK");
                                }
                            });
                        }
                    }));
                    put("Requirement", "MEETS_ALL");
                }}));
            }
        })
        .redactedFields(Arrays.asList(
                CfnLoggingConfiguration.FieldToMatchProperty
                        .builder()
                        .singleHeader(new HashMap<String, Object>() {{
                                put("Name", "cookie");
                        }})
                        .build()
                ))
        .build();
```

Kuva 9. Pääsynhallintalistan lokien kerääminen

6.3.5 Lokien kerääminen ja lukeminen

Lokien kerääminen määriteltiin niin, että kerätään tietoa vain pyynnöistä, jotka on estetty. Näin voidaan vähentää kerätyn datan määrää ja helpottaa myös haittakohtien huomaamista, sillä lokitiedot eivät ole täynnä onnistuneita pyyntöjä. Lokien keräämistä varten syntyi myös tarve IAM-roolin luomiselle ja määrittelemiselle (kuva 10), jotta lokit voidaan siirtää talteen automaattisesti. Talteen siirtämistä varten tarvittiin myös S3-ämpäri, joka piti myös luoda ja määritellä, jotta lokit voitiin siirtää sinne turvallisesti.

```
return CfnRole.Builder
        .create(this, "mercury-logging-role")
        .assumeRolePolicyDocument(PolicyDocument.Builder
                .create()
                .statements(Arrays.asList(PolicyStatement.Builder
                        .create()
                        .effect(Effect.ALLOW)
                        .principals(Arrays.asList(ServicePrincipal.Builder.create("firehose.amazonaws.com").build()))
                        .actions(Arrays.asList("sts:AssumeRole"))
                        .build()
                ))
                .build()
        )
        .policies(Arrays.asList(
                CfnRole.PolicyProperty
                        .builder()
                        .policyName("AllowWriteLogsToS3")
                        .policyDocument(PolicyDocument.Builder
                                .create()
                                .statements(Arrays.asList(
                                        PolicyStatement.Builder
                                                .create()
                                                .effect(Effect.ALLOW)
                                                .actions(Arrays.asList(
                                                        "s3:AbortMultipartUpload",
                                                        "s3:GetBucketLocation",
                                                        "s3:GetObject",
                                                        "s3:ListBucket",
                                                        "s3:ListBucketMultipartUploads",
                                                        "s3:PutObject"
                                                ))
                                                .resources(Arrays.asList(
                                                        bucket.getBucketArn(),
                                                        bucket.getBucketArn() + "/*"
                                                ))
                                                .build(),
                                        PolicyStatement.Builder
                                                .create()
                                                .effect(Effect.ALLOW)
                                                .actions(Arrays.asList(
                                                        "kinesis:DescribeStream",
                                                        "kinesis:GetShardIterator",
                                                        "kinesis:GetRecords",
                                                        "kinesis:ListShards"
                                                ))
                                                .resources(Arrays.asList(
                                                        "arn:aws:kinesis:"
                                                        + getAppStackConfig().awsRegionCode()
                                                        + ":"
                                                        + getAppStackConfig().awsAccountId()
                                                        + ":stream/aws-waf-logs-"
                                                        + getAppStackConfig().activeEnvName()
                                                ))
                                                .build()
                                ))
                                .build()
                        )
                        .build()
        ))
        .build();
```

Kuva 10. IAM-roolin määrittely

Lokitietojen siirtämistä varten määriteltiin virtaus (stream), joka siirtää kerätyt lokit aikavälein kompressoidussa muodossa S3-ämpäriin (kuva 11). Näin lokitietojen säilyttäminen on

kustannusystävällisempää ja niiden käyttäminen on myös joustavampaa verrattuna oletuksena käytettävään CloudWatch:iin. Tällöin lokitietoja on myös mahdollista käyttää muissa palveluissa integroimalla ne käyttämään S3-ämpäriä datasijaintina.

```
return CfnDeliveryStream.Builder
        .create(this, "mercury-delivery-stream")
        .deliveryStreamName("aws-waf-logs-" + applicationName)
        .deliveryStreamType("DirectPut")
        .s3DestinationConfiguration(CfnDeliveryStream.S3DestinationConfigurationProperty.builder()
                .bucketArn(bucket.getBucketArn())
                .prefix("logs/")
                .bufferingHints(CfnDeliveryStream.BufferingHintsProperty.builder()
                        .intervalInSeconds(300)
                        .sizeInMBs(5)
                        .build()
                )
                .compressionFormat("GZIP")
                .roleArn(role.getAttrArn())
                .build()
        )
        .build();
```

Kuva 11. Virtaus lokien siirtämistä ja paketointia varten

Jotta lokitietoja voidaan käsitellä mahdollisimman helposti suoraan AWS:n kautta, tarvitaan jonkinlainen tietokantaratkaisu, jolla voidaan lukea S3:een tallennettua dataa ja jolla voidaan tehdä kyselyitä dataa vasten. Tähän valikoitui ratkaisuksi jo yrityksellä aiemmin käytössä ollut Glue-palvelu, jota voidaan käyttää yhdessä toisen palvelun Athena:n kanssa. Glue:lla voidaan kerätä S3-ämpäristä dataa ja kääntää se tietokantamallin mukaiseksi (kuva 12). Tämän jälkeen on mahdollista käyttää Athena:a, jolla voidaan suorittaa kyselyitä kyseistä dataa vasten (kuva 13).

```
.storageDescriptor(CfnTable.StorageDescriptorProperty
        .builder()
        .location("s3://" + bucket.getBucketName() + "/logs/")
        .inputFormat("org.apache.hadoop.mapred.TextInputFormat")
        .outputFormat("org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat")
        .serdeInfo(CfnTable.SerdeInfoProperty
                .builder()
                .parameters(new HashMap<String, Object>() {{
                    put("paths", "action,formatVersion,httpRequest,httpSourceId,httpSourceName,nonTerminatingMatchingRules,"
                            + "rateBasedRuleList,ruleGroupList,terminatingRuleId,terminatingRuleType,timestamp,webaclId");
                }})
                .serializationLibrary("org.openx.data.jsonserde.JsonSerDe")
                .build()
        )
        .compressed(Boolean.TRUE)
        .storedAsSubDirectories(Boolean.TRUE)
        .columns(Arrays.asList(
                CfnTable.ColumnProperty.builder().name("timestamp").type("bigint").build(),
                CfnTable.ColumnProperty.builder().name("formatversion").type("int").build(),
                CfnTable.ColumnProperty.builder().name("webaclid").type("int").build(),
                CfnTable.ColumnProperty.builder().name("terminatingruleid").type("string").build(),
                CfnTable.ColumnProperty.builder().name("terminatingruletype").type("string").build(),
                CfnTable.ColumnProperty.builder().name("action").type("string").build(),
                CfnTable.ColumnProperty.builder().name("httpsourcename").type("string").build(),
                CfnTable.ColumnProperty.builder().name("httpsourceid").type("string").build(),
                CfnTable.ColumnProperty.builder().name("rulegrouplist").type("array<string>").build(),
                CfnTable.ColumnProperty.builder().name("ratebasedrulelist").type("array<string>").build(),
                CfnTable.ColumnProperty.builder().name("nonterminatingmatchingrules").type("array<string>").build(),
                CfnTable.ColumnProperty.builder().name("httprequest").type("struct<clientip:string,country:string"
                + "headers:array<struct<name:string,value:string>>,uri:string,args:string,httpversion:string,"
                + "httpmethod:string,requestid:string>").build()
        ))
        .build()
)
.build()
```

Kuva 12. Ote tietokantakolumnien luomisesta AWS Glue:en

```
CfnNamedQuery.Builder
  .create(this, "mercury-athena-named-query-getblockedips")
  .database(database.getRef())
  .description("Gets the latest blocked IPs from WAF log. Update the time range.")
  .name("Get blocked IPs")
  .queryString(
          "select count(httpRequest.clientIp) as count, terminatingruleid, httpRequest.clientIp "
          + "from waf_logs "
          + "where action='BLOCK' "
          + "and datehour>='2021/08/16/00' and datehour<'2021/08/17/00' "
          + "group by terminatingruleid, httpRequest.clientIp "
          + "order by count desc "
          + "limit 100"
  )
  .workGroup(workgroup.getRef())
  .build();
```

Kuva 13. Esimerkki tietokantakysely Athena:a käyttämällä

6.4 Julkaisu- ja käyttöönottoprosessien automatisointi

Toimeksiannossa toivottiin myös julkaisu- ja käyttöönottoprosessien helpottamista automatisoimalla suoritettavia prosesseja mahdollisuuksien mukaan. Päädyin luomaan skriptit yleisimpiin

skenaarioihin, joita ympäristönhallinta edellyttää. Yleisimmiksi skenaarioiksi valikoituvat seuraavat tilanteet ja tehtävät:

- Ohjelmiston julkaiseminen ECR-palveluun
- Ympäristön luonti ja päivittäminen
- Viimeisimmän ohjelmistoversion julkaiseminen Fargate-palveluun

Kaikissa skripteissä on oletusarvot esimerkiksi käytettävälle käyttäjätunnukselle ja ympäristölle, mutta käyttäjälle tarjotaan mahdollisuus niiden asettamiselle suorittamisen alkuvaiheessa. Skriptit etenevät portaittain ja kysyvät käyttäjältä vahvistuksen seuraavan vaiheen suorittamiselle aina vaiheiden välissä. Vahvistuksia käyttämällä käyttäjä voi suorittaa skriptistä vain haluamansa asiat, eikä esimerkiksi koodin validointia tehdessä ole pakko ajaa myös julkaisuosuutta.

Jotta ohjelmiston voi julkaista ECR-palveluun, niin täytyy se saattaa julkaisukuntoon ensin. Prototyypissä käytetyn ohjelmiston osalta tämä tarkoittaa sovelluksen paketointia ja Docker-kuvan rakentamista. AWS:n ohjeiden mukaisesti luotu kuva pitää myös merkitä (tag) käytettävän rekisterin osoitteella ja versionumerolla ennen julkaisua. Ensimmäisessä skriptissä suoritetaan ensin yllä mainitut asiat ja lopuksi kirjaudutaan ECR-palveluun käyttämällä AWS:n CLI:tä, joka suorittaa kuvan julkaisemisen kirjautumisen jälkeen (kuva 14).

```
read -p "Upload to ECR (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    echo "----------------------------"
    echo "Uploading package to ECR: ${ECR_URL}"

    aws ecr get-login-password --profile ${awsProfile} --region ${awsRegion} | docker login --username AWS --password-stdin ${ECR_URL}
    docker push ${ECR_URL}

    echo "----------------------------"

    echo "----------------------------"
    echo "All complete"
    echo "Deployed to: ${ECR_URL}"
    echo "----------------------------"
else
    echo "Skipped upload, bye bye"
fi
```

Kuva 14. Kuvan julkaiseminen ECR-palveluun

Myös ympäristön luominen ja päivittäminen vaatii koodin paketoinnin ennen kuin sitä voi käyttää. Kuvan 15 mukaisesti toinen skripti suorittaa paketoinnin automaattisesti ajon alussa ja sen jälkeen validoi koodin syntaksin kääntämällä sen CLI:n avulla CloudFormation-malliksi. Tämän jälkeen käyttäjä voi hyväksyä julkaisuprosessin aloittamisen ajaessaan skriptiä, mikäli syntaksin validointi on onnistunut. Julkaisuprosessin jälkeen ympäristö on päivitetty, mikäli olemassa olevaan ympäristöön nähden on jonkinlaisia muutoksia.

```
mvn -q compile

read -p "Run CDK Synth (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    echo ""
    echo "----------------------------"
    echo "Running CDK Synth..."
    echo "----------------------------"
    cdk synth --app "mvn -e -q compile exec:java -Dexec.mainClass=tenduke.mercury.aws.deploy.licensed.AwsLicensedFgAppMain"
else
    echo "Skipped cdk synth"
fi

read -p "Run CDK Deploy (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    echo ""
    echo "----------------------------"
    echo "Running CDK Deploy..."
    echo "----------------------------"
    cdk deploy "$@" --app "mvn -e -q compile exec:java -Dexec.mainClass=tenduke.mercury.aws.deploy.licensed.AwsLicensedFgAppMain"
else
    echo "Skipped deploy, bye bye"
fi
```

Kuva 15. Ympäristön julkaisuskripti

Kolmannen skriptin tarkoituksena on päivittää ympäristössä ajettava ohjelmisto (kuva 16). Koska toinen skripti päivittää ympäristön vain silloin, kun ympäristön CDK-koodiin on tehty muutoksia, niin se ei toimi tilanteessa, jossa halutaan päivittää pelkästään ohjelmisto. Kolmas skripti hakee valitun AWS-ympäristön perusteella ECS-palvelussa olevat ryhmät, joista käyttäjä valitsee haluamansa. Tämän jälkeen skripti listaa ryhmän palvelut, joista käyttäjä valitsee sen, jota haluaa käyttää. Lopuksi skripti hakee käytettävän Docker-kuvan ECR-rekisteristä ja luo uudet korvaavat instanssit sen pohjalta.

```
read -p "List clusters?  (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    aws ecs list-clusters --profile ${awsProfile} --region ${awsRegion}
else
    echo "Skipped listing clusters"
fi

echo "Input cluster name:"
read clusterName
clusterName=${clusterName}
export clusterName=${clusterName}

read -p "List services inside cluster?  (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    aws ecs list-services --cluster ${clusterName} --profile ${awsProfile} --region ${awsRegion}
else
    echo "Skipped listing services"
fi

echo "Input service name:"
read serviceName
serviceName=${serviceName}
export serviceName=${serviceName}

read -p "Restart services and force new image?  (y/n)? " -n 1 -r
echo
if [[ $REPLY =~ ^[Yy]$ ]]
then
    aws ecs update-service --cluster ${clusterName} --service ${serviceName} --force-new-deployment
    echo "Restarting completed!"
else
    echo "Skipped restarting"
fi
```

Kuva 16. Skripti, jolla päivitetään ympäristön käyttämä kuva

7 Prototyypin validointi ympäristöjä vertailemalla

Toimeksiantoon kuului serverless-prototyypin kehittämisen lisäksi teknologian käytettävyyden validointi. Käytännössä tämä tarkoittaa sitä, että prototyypin ominaisuuksia vertaillaan olemassa olevaan ratkaisuun ja sen pohjalta analysoidaan teknologian käytettävyyttä. Käytettävyyden validoinnin ja analysoinnin tarkoituksena on luoda pohja mahdolliselle jatkokehitykselle, mikäli tulokset sitä puoltavat.

7.1 Valmistelut

Jotta ympäristöjä voidaan vertailla keskenään, tulee niiden olla mahdollisimman samankaltaiset. Prototyypin kehittämisvaiheessa oli tehty päätös, että prototyyppi käyttäisi mahdollisuuksien mukaan samoja resursseja, kuin mitä olemassa oleva ratkaisu käyttää. Ympäristöt jakavat siis keskenään esimerkiksi tietokannan ja verkon. Samankaltaisuutta varten myös ympäristöjen ominaisuudet tulee määritellä samankaltaisiksi.

Konfiguroitavien arvojen määrittelemistä varten pitää aluksi selvittää EC2-instanssin koko, joka voidaan määritellä ominaisuuksiltaan samankaltaiseksi Fargate:ssa. Määrittelyn olisi hyvä tapahtua mahdollisimman pienillä arvoilla, jotta voidaan kuormittaa palvelua riittävästi, sekä testata autoskaalausta ja ympäristöjen kestävyyttä samalla. Tärkeimmät määriteltävät arvot tässä kohtaa ovat vCPU ja muisti, sillä ne on mahdollista määritellä sekä EC2:ssa että Fargate:ssa. EC2:ssa päädyttiin käyttämään T3-luokan instanssia, sillä se tarjoaa jo entuudestaan käytetyssä AWS-alueessa pienimmät mahdolliset instanssit.

Seuraavaksi piti löytää ympäristöjen välillä yhteneväiset, mahdollisimman pienillä arvoilla määritellyt asetukset. T3-luokan instansseissa on poikkeuksetta tarjolla instansseja, joissa minimi vCPU määrä on 2 (kuva 17). Fargate:ssa puolestaan kahdelle vCPU:lle on tarjolla pienimmillään 4 GB:tä muistia (kuva 18). Näin ollen ensimmäinen mahdollinen instanssityyppi käytettäväksi vertailussa on t3.medium. Olemassa olevaan ympäristöön päivitettiin käytettäväksi kyseinen instanssi ja Fargate:en sitä vastaavat asetukset, eli 2 vCPU:ta ja 4 GB muistia.

Name	vCPUs	Memory (GiB)	Baseline Performance/vCPU	CPU Credits earned/hr	Network burst bandwidth (Gbps)	EBS burst bandwidth (Mbps)	On-Demand Price/hr*	1-yr Reserved Instance Effective Hourly*	3-yr Reserved Instance Effective Hourly*
t3.nano	2	0.5	5%	6	5	Up to 2,085	$0.0052	$0.003	$0.002
t3.micro	2	1.0	10%	12	5	Up to 2,085	$0.0104	$0.006	$0.005
t3.small	2	2.0	20%	24	5	Up to 2,085	$0.0209	$0.012	$0.008
t3.medium	2	4.0	20%	24	5	Up to 2,085	$0.0418	$0.025	$0.017
t3.large	2	8.0	30%	36	5	Up to 2,780	$0.0835	$0.05	$0.036
t3.xlarge	4	16.0	40%	96	5	Up to 2,780	$0.1670	$0.099	$0.067
t3.2xlarge	8	32.0	40%	192	5	Up to 2,780	$0.3341	$0.199	$0.133

Kuva 17. Amazon EC2 T3 instanssit (Amazon s.a.k)

CPU	Memory Values
0.25 vCPU	0.5 GB, 1 GB, and 2 GB
0.5 vCPU	Min. 1 GB and Max. 4 GB, in 1 GB increments
1 vCPU	Min. 2 GB and Max. 8 GB, in 1 GB increments
2 vCPU	Min. 4 GB and Max. 16 GB, in 1 GB increments
4 vCPU	Min. 8 GB and Max. 30 GB, in 1 GB increments

Kuva 18. Amazon Fargate:n tukemat asetukset (Amazon s.a.l)

7.2 Työn määrä

Työmäärän vertaileminen on vaikeaa eri ympäristöjen välillä, sillä eri ympäristöissä voidaan tarvita erilaisia asioita mahdollisimman samankaltaisen lopputuloksen aikaansaamiseksi. Työmäärää arvioidessa pyritään saamaan ymmärrys siitä, olisiko samankaltaisen ympäristön luominen mahdollisesti kannattavaa myös muita yrityksen palveluita ajatellen. Tähän vaikuttaa muun muassa valmiin tuotoksen vaatima työmäärä ja -aika. Sen vuoksi työmäärän arviointiperusteet ovat seuraavanlaiset:

- Kirjoitettavan koodin määrä
- Kirjoitettavan koodin kompleksisuus

Määrällisesti Fargate-prototyypin koodia ei tarvitse kirjoittaa yhtä paljoa, kuin jo olemassa olleessa Elastic Beanstalk:issa. Fargate:ssa yhdistellään useita asioita paremmin kuin Elastic Beanstalk:issa, esimerkiksi yhden luokan määrittelemisellä (kuva 5) voi asettaa useiden palveluiden arvot kohdilleen. Fargate:en on myös esimerkiksi luotu tuki Secrets Manager:in käyttämistä varten, mitä ei ole suoraan tarjolla Elastic Beanstalk:issa tällä hetkellä. Koodia Fargate:a käyttäessä tarvitsee luoda noin 30 prosenttiyksikköä vähemmän kuin Elastic Beanstalk:ia käyttäessä, mutta koska kumpikaan ratkaisu ei kuitenkaan vaadi suurta määrää koodia CDK-tasolla, niin eroa ei tarvitse pitää merkittävänä.

Määrittelemisen suhteen Fargate on joustavampi kuin Elastic Beanstalk. Elastic Beanstalk:ia käyttäessä pitää tietää mitä instanssia haluaa käyttää, jotta saa virtuaalipalvelimen, jossa on halutut ominaisuudet. Fargate:ssa instanssityyppiä ei tarvitse määritellä, vaan se määräytyy automaattisesti määriteltyjen arvojen mukaisesti. Fargate:ssa on kuitenkin myös jonkinlaisia rajoitteita määrityksiä tehdessä, joten myös se vaatii hieman taustatuntemusta.

7.3 Julkaisu- ja käyttöönotto

Julkaisu- ja käyttöönottoprosessien vertailulla pyritään saamaan tietoa siitä, onko ympäristöjen välillä merkittäviä hallintaeroja. Tämä kattaa siis kaikki yleisimmät prosessit, joilla ohjelmisto saatetaan kohdeyleisön käytettäväksi, joko ensimmäistä kertaa julkaistavana tai päivitettävänä versiona. Vertailuun kuuluu myös ylläpidollisten prosessien selvittäminen, sillä sekin on oleellinen osa työtehtäviä, jotka kuuluvat ohjelmiston laadunvalvontaan.

Olemassa olevassa ratkaisussa Elastic Beanstalk vaatii palvelinten uudelleenkäynnistämisen aina, kun julkaistua ohjelmistoa halutaan päivittää. Tilanne pysyy samanlaisena Fargate:n kohdalla, joten tässä ominaisuudessa ei suoranaista eroa ole. Fargate on kuitenkin huomattavasti nopeampi luomaan uusia kontteja, kuin mitä Elastic Beanstalk virtuaalipalvelimia. Tästä johtuen prototyypin päivittäminen on nopeampaa ja sama etu tulee esiin myös palvelua skaalatessa. Elastic Beanstalk:illa voi uuden palvelimen pystyttämisessä mennä muutama minuutti, kun taas Fargate pystyy luomaan ja käynnistämään kontin kymmenissä sekunneissa. Prototyyppiä testatessa sovellus oli käynnissä nopeimmillaan alle puolessa minuutissa, joka on reipas etu olemassa olevaan ratkaisuun verrattuna.

Skaalauksen hallinta vaikuttaa Elastic Beanstalk:issa laajemmalta, kuin mitä Fargate tarjoaa. Peruskäytössä se ei kuitenkaan aiheuta suurta eroa, sillä Fargate:n skaalausvaihtoehdot ovat yhtä lailla toimivia. Yhtenä isona erona skaalauksessa on kuitenkin se, että Fargate:n voi skaalata myös nollaan instanssiin asti. Tällä voi saada kustannuksellisia etuja sovelluksessa, joka voi olla pitkiä aikoja käyttämättömänä. Fargate tukee myös aikamääreellistä skaalaamista, eli palvelun voi

skaalata nollaan esimerkiksi viikonlopun ajaksi. Isoin skaalaukseen liittyvä ero on todennäköisesti kuitenkin siinä, että olemassa oleva ratkaisu käyttää virtuaalipalvelimia, kun taas Fargate käyttää kontteja. Tämän vuoksi Fargate on huomattavasti nopeampi skaalatessa sekä ylös että alaspäin.

Elastic Beanstalk vaatii käyttäjältä palvelun hallinnointia ja ylläpitämistä. Sen tarjoamilla alustavaihtoehdoilla on yleensä jonkinlainen elinikä, jonka jälkeen ne poistetaan käytöstä. Käytännössä tämä tarkoittaa sitä, että käyttäjän tulee seurata palvelua mahdollisen alustan vanhentumisen vuoksi ja korvata se uudemmalla tuetulla versiolla. Yksittäisessä ympäristössä tämä ei välttämättä vaadi merkittävää työmäärää, mutta mikäli käyttäjä hallinnoi esimerkiksi kymmeniä ympäristöjä, voi työmääräkin kasvaa suureksi. Fargate:ssa ei samanlaista ongelmaa tule vastaan, sillä sen on tarkoitus olla itsenäinen kokonaisuutensa, jossa käyttäjän ei tarvitse välittää ympäristön hallitsemisesta. Tämä on samalla suuri etu käyttäjän näkökulmasta, sillä se mahdollistaa resurssien käyttämisen toisaalla.

7.4 Suorituskyky

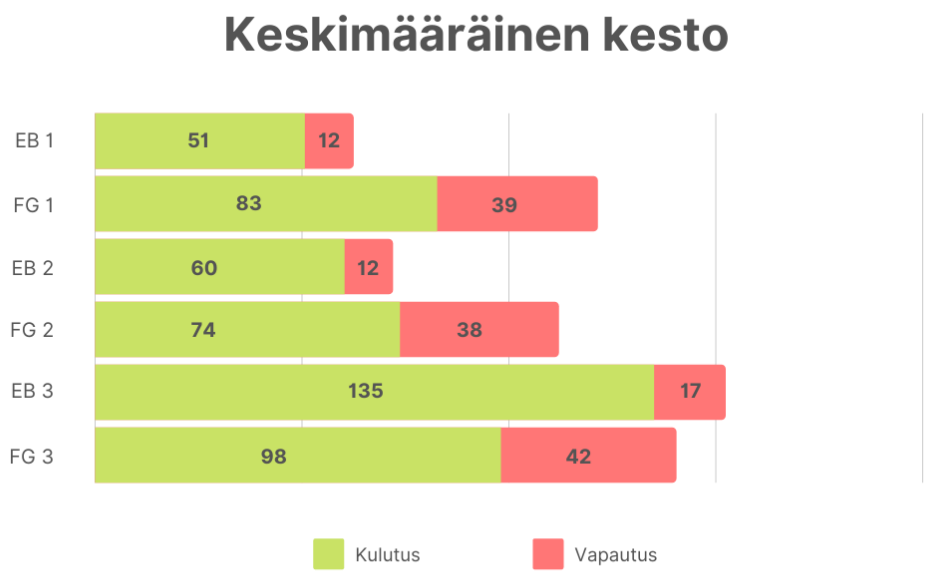
Suorituskykyä tarkasteltiin kolmessa eri osa-alueessa. Ensimmäisenä osa-alueena oli ympäristön selviytyminen suuressa rasituksessa ilman, että siitä aiheutuu käyttökatkoja. Seuraavana tarkasteltiin palvelun skaalautumiskykyä, eli sitä, kuinka nopeasti palvelu kykenee skaalaamaan sekä ylöspäin että alaspäin. Viimeisenä osa-alueena suorituskyvyn vertailussa oli suoriutumisnopeus, eli se, kuinka nopeasti palvelu kykenee palvelemaan käyttäjiä.

Suorituskyvyn testaamisessa käytettiin JMeter-ohjelmistoa, jolla luotiin keinotekoisesti tilanteet, joissa lisenssejä kulutetaan ja vapautetaan REST-rajapinnan kautta. Lisenssien kuluttaminen tarkoittaa tässä lisenssin käyttämistä, eli sitä että kyseisestä lisenssistä on varattu paikka käyttäjälle. Lisenssin vapauttaminen puolestaan tarkoittaa sitä, että käyttäjä vapauttaa aiemmin varatun paikan. Testaamisessa luotiin erilaisia skenaarioita, joissa pienimmillään kulutettiin ja vapautettiin 4 000 lisenssiä samanaikaisesti ja suurimmillaan kulutettiin ja vapautettiin 12 000 lisenssiä samanaikaisesti. Kyseisiä skenaarioita ajettiin 1–10 kertaa peräkkäin ympäristöä kohden testitapauksesta riippuen.

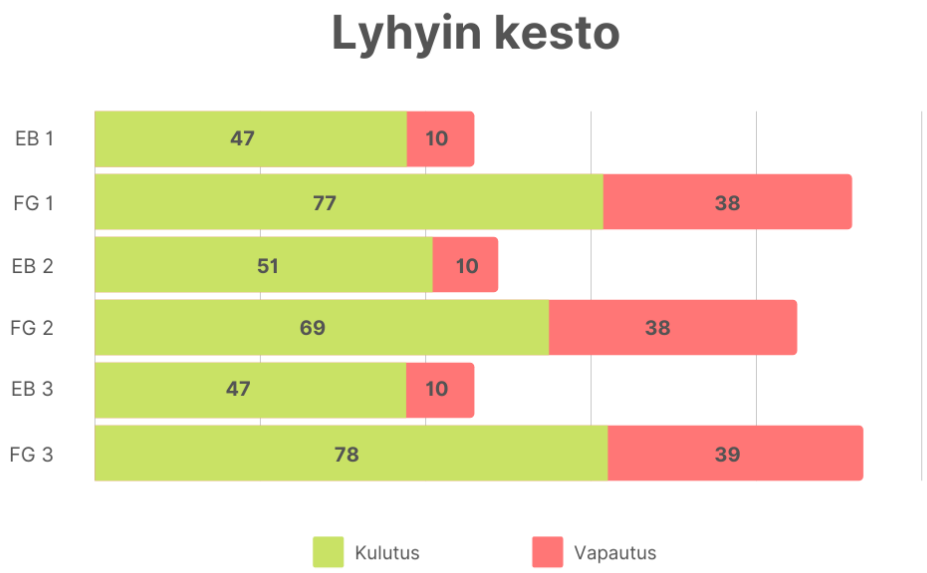
Testien aikana kumpikaan ympäristö ei kokenut käyttökatkoja laisinkaan. Elastic Beanstalk-ympäristössä yksittäisen virtuaalipalvelimen CPU-käyttö kävi kerran jopa 97 prosentissa, mutta muuten maksimi oli noin 76 % ja Fargate-ympäristössä päästiin korkeimmillaan 71 % yksittäisellä kontilla. CPU-käytön keskiarvo testien aikana oli EB:ssä noin 50 % ja Fargate:ssa noin 40 %. Fargate:ssa konttien kuormitus ei siis ollut yhtä kovaa kuin virtuaalipalvelimien Elastic Beanstalk:issa.

Molempien ympäristöjen skaalaaminen onnistui hyvin. Fargate oli kuitenkin parempi nopeutensa puolesta skaalatessa, sillä se reagoi 50 % rajan ylitykseen CPU:n käytössä nopeasti ja sai jo ensimmäisen testin aikana skaalattua toisen kontin ensimmäisen rinnalle. Elastic Beanstalk:in skaalaaminen alkoi myös nopeasti ensimmäisen testin aikana, mutta toisen instanssin hyöty näkyi vasta seuraavan testin aikana.

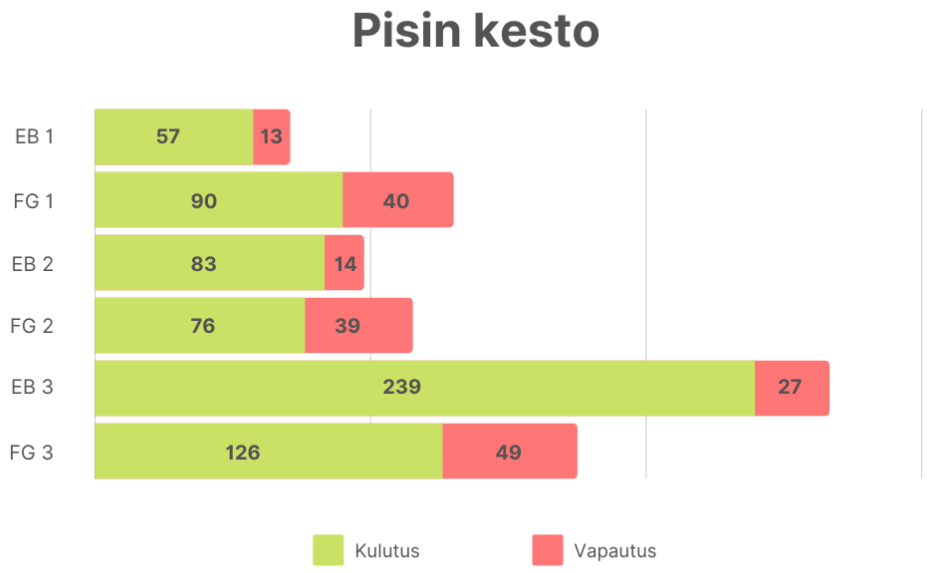
Ympäristöjen suoriutumisnopeudet vaihtelivat paljon kellonajan ja päivän suhteen. Joinakin testikertoina Elastic Beanstalk:issa oleva sovellus suoritti huomattavasti nopeammin pyynnöt kuin Fargate:ssa, mutta toisina kertoina tilanne saattoi olla täysin päinvastainen. Yleisellä tasolla Elastic Beanstalk pärjäsi kuitenkin Fargate:a paremmin pyyntöjen palvelemisessa (kuva 19), joista erityisesti lisenssin vapautuspyynnöt onnistuivat nopeasti. Elastic Beanstalk oli myös parempi pyyntöjen minimi- (kuva 20) ja maksimipituuksissa (kuva 21), joissa erityisesti lyhyimpien pyyntöjen kestot olivat lähes identtisiä pituudeltaan. Kolmannella testikierroksella EB-ympäristö hävisi pyyntöjen kokonaispituuksissa niukasti Fargate:lle, mutta erot eivät silloinkaan olleet suuria. Testien perusteella virtuaalipalvelimella Elastic Beanstalk:issa pyörivä ohjelmisto sai alustastaan siis enemmän irti, sillä vaikka sen CPU-käyttö oli korkeammalla, niin myös sen suoriutuminen oli parempaa testeissä.



Kuva 19. Lisenssin hallintapyynnön keskimäärinen kesto millisekunneissa



Kuva 20. Lisenssin hallintapyynnön lyhin kesto millisekunneissa



Kuva 21. Lisenssin hallintapyynnön pisin kesto millisekunneissa

7.5 Kustannukset

Kustannusten vertailemista varten piti selvittää AWS:n tarjoamat arviot molempien ympäristöjen kuluista, kerätä dataa toteutuneista kuluista testauksen aikana ja käydä läpi mahdollisia tapoja, joilla kuluja voi optimoida. Amazon tarjoaa kulujen arvioimista varten erilaisia kaavioita ja laskureita, joilla kuluerien suuruutta pystyy ennakoimaan mahdollisimman tarkasti. AWS kerää myös tarkkaa dataa siitä, minkä verran eri palveluiden resurssit ovat kustantaneet. Näiden avulla pitäisi olla mahdollista saada kohtuullisen tarkka kuva siitä, millaisia kustannuseroja ympäristöjen välillä on.

T3-instanssien kohdalla Amazon kertoo kustannuksiksi noin 0,0052–0,3341 dollaria tunnissa riippuen instanssien koosta. Fargate:n hinnoittelu määräytyy puolestaan käytettyjen vCPU:iden ja muistin mukaan niin, että yksi vCPU maksaa noin 0.0445 dollaria ja yksi gigatavu (GB) noin 0.0049 dollaria tunnissa. Näiden hintojen lisäksi molemmat palvelut tarjoavat myös hinnoittelumallin sen hetkisen kysynnän mukaisesti, eli mikäli kysyntä on matalaa niin myös hinnat ovat matalampia ja sama päinvastoin. Vertailussa käytettiin Elastic Beanstalk:issa t3.medium-instanssia, jonka tuntihinnaksi on merkitty 0.0418 dollaria ja se sisältää 2vCPU:ta ja 4 GB muistia. Listahinnoittelun mukaan Fargate olisi siis Elastic Beanstalk:ia kalliimpi vaihtoehto, jos rakennetaan ympäristö samoilla määrityksillä.

Toteutuneiden kustannusten vertailua varten kerättiin dataa ympäristöistä sekä silloin kun niitä käytetään, että silloin kun niitä ei käytetä. Vertailua varten kerättiin dataa kustannuksista viikon ajan vaihtelevalla käytöllä, jotta voidaan simuloida oikeaa käyttöä pienemmässä mittakaavassa. Molempien palvelujen kustannukset pysyivät lähes samoina käytöstä riippumatta. Kovemman käytön päivinä hinnat saattoivat nousta joitakin prosenttiyksikköjä korkeammalle, mutta ei kuitenkaan merkittävästi. Palvelut, joita molemmat ympäristöt käyttävät karsittiin pois vertailusta. Näin saatiin eroavien resurssien kustannuserot paremmin esille ja selkeämmin vertailtaviksi. Elastic Beanstalk:in käyttökustannukset olivat keskimäärin 1,29 dollaria päivässä ja Fargate:n 2,61 dollaria päivässä. Fargate on siis myös käytännössä kalliimpi, noin kaksi kertaa korkeammin kokonaiskustannuksin.

Vertailussa ei ole kuitenkaan otettu huomioon sitä, että Fargate:n voi optimoida palvelittomaan tilaan halutessa. Toisin sanoen Fargate:n kustannukset on mahdollista laskea nollaan esimerkiksi viikonlopun ajaksi, mikäli palvelua ei silloin käytetä. Elastic Beanstalk:issa ei ole samanlaista mahdollisuutta, joten minimikustannukset pysyvät yhden instanssin mukaisina jatkuvasti. Fargate:a ei todennäköisesti tarvitsisi myöskään määritellä samanlaisin arvoin kuin Elastic Beanstalk:ia, sillä sen kustannukset määräytyvät käytön mukaan, kun taas Elastic Beanstalk:in kustannukset määräytyvät instanssikoon mukaan. Teoriassa olisi siis mahdollista optimoida kuluja esimerkiksi

asettamalla Fargate:n oletusmääritykset noin neljännekseen EC2-instanssin määrityksistä ja antamalla sen skaalata ylöspäin tarvittaessa, mikäli sen kuorma kasvaa.

Tässä vertailussa käytetyssä ohjelmistossa skaalaaminen nollaan ei kuitenkaan ole mahdollista, sillä se on HA-tyyppinen ohjelmisto, eli sen tulee olla aina saatavilla. Pienempien resurssien määrittäminen tosin voisi olla mahdollista, mutta sitä ei tämän opinnäytetyön rajoitusten puitteissa lähdetty selvittämään. Näin ollen tässä vertailussa Fargate on kalliimpi vaihtoehto käyttää, mutta samalla se säästää ylläpidollisen työn määrässä.

8 Yhteenveto

Opinnäytetyön tavoitteena oli luoda ensimmäinen serverless-teknologiaa hyödyntävä prototyyppi yrityksen käyttöön ja validoida sen käytettävyys yrityksen palveluissa. Prototyypin luominen onnistui odotusten mukaisesti ja käytetyt teknologiat mahdollistivat ohjelmiston käyttöönoton ilman, että siihen tarvitsisi tehdä muutoksia. Kehitetty prototyyppi on malliltaan sellainen, että sen jatkokehittämien on mahdollista ja sitä voi halutessa hyödyntää myös muissa yrityksen palveluissa.

Prototyypin käytettävyys on myös validoitu ja serverless-teknologia voidaan todeta yrityksen palveluihin soveltuvaksi. Fargate:n käyttöönotto sujui yhtä helposti kuin aiemmilla teknologioilla toteutetut ympäristöt, eikä se vaadi yhtä paljoa ylläpidollista työtä kuin edeltäjänsä. Pakollisen ylläpidon vähentyminen näkyy kuitenkin myös palvelun hinnoittelussa, jossa Fargate on selkeästi edeltäjäänsä kalliimpi.

Suorituskyvyllisesti molemmat palvelut pärjäsivät hyvin, tosin mittaustuloksien vaihteluvälistä ei löytynyt selkeää säännöllisyyttä, vaan tulokset vaihtelivat paljon ajankohdan mukaan. Näin ollen tuloksia ei voida pitää täysin mittauskelpoisina, eikä niiden pohjalta voida tehdä johtopäätöksiä käytettävyyden suhteen. Suorituskyvyn laajempi mittaaminen ja eroavaisuuksien selvittäminen ympäristöjen välillä voisi olla hyvä jatkoaihe työlle.

Opinnäytetyö on kokonaisuutena ollut onnistunut, sillä sen avulla saatiin vastaukset alun perin asetettuihin kysymyksiin. Prototyypin kehittäminen ja teknologian mahdollisuuksien tutkiminen jatkuu yrityksessä vielä tulevaisuudessakin, ja seuraavaksi olisikin hyvä selvittää teknologian käyttämisen kannattavuutta. Tulevina tutkimuskohteina olisi myös selvittää syitä mahdollisille eroille suorituskyvyn suhteen ja kokeilla ja vertailla teknologian hyötyjä suuremmassa skaalassa.

Lähteet

Amazon. 2002. Amazon.com Launches Web Services; Developers Can Now Incorporate Amazon.com Content and Features into Their Own Web Sites; Extends "Welcome Mat" for Developers. Luettavissa: https://press.aboutamazon.com/news-releases/news-release-details/amazoncom-launches-web-services. Luettu 27.1.2022.

Amazon. 2006. Announcing Amazon Elastic Compute Cloud (Amazon EC2) – beta. Luettavissa: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/. Luettu 15.3.2022.

Amazon. s.a.a. What is IAM? User Guide. Luettavissa: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html. Luettu 14.3.2022.

Amazon. s.a.b. AWS Secrets Manager. Luettavissa: https://aws.amazon.com/secrets-manager/. Luettu 19.3.2022.

Amazon s.a.c. AWS WAF – Web Application Firewall. Luettavissa: https://aws.amazon.com/waf/. Luettu 19.3.2022.

Amazon. s.a.d. Amazon Elastic Container Registry (Amazon ECR). Luettavissa: https://aws.amazon.com/ecr/. Luettu 19.3.2022.

Amazon s.a.e. What is Amazon Elastic Container Service? Luettavissa: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html. Luettu 20.3.2022.

Amazon s.a.f. What is AWS Fargate? Luettavissa: https://docs.aws.amazon.com/AmazonECS/latest/userguide/what-is-fargate.html. Luettu 20.3.2022.

Amazon s.a.g. AWS Command Line Interface. Luettavissa https://aws.amazon.com/cli/. Luettu 20.3.2022.

Amazon s.a.h. AWS CloudFormation. Luettavissa: https://aws.amazon.com/cloudformation/. Luettu 20.3.2022.

Amazon s.a.i. What is AWS CloudFormation? Luettavissa: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html. Luettu 20.3.2022.

Amazon s.a.j. What is the AWS CDK? Luettavissa: https://docs.aws.amazon.com/cdk/v2/guide/home.html. Luettu 20.3.2022.

Amazon s.a.k. Amazon EC2 T3 Instances. Luettavissa: https://aws.amazon.com/ec2/instance-types/t3/. Luettu 22.3.2022.

Amazon s.a.l. AWS Fargate Pricing. Luettavissa: https://aws.amazon.com/fargate/pricing/. Luettu 22.3.2022.

Carty, D. 2021. Amazon Elastic Container Registry (Amazon ECR). Luettavissa: https://www.techtarget.com/searchitoperations/definition/Amazon-EC2-Container-Registry. Luettu 28.3.2022.

Carty, D. 2017. Amazon Elastic Container Service (Amazon ECS). Luettavissa: https://www.techtarget.com/searchaws/definition/Amazon-EC2-Container-Service. Luettu 28.3.2022.

Cloudflare. Why use serverless computing? | Pros and cons of serverless. Luettavissa: https://www.cloudflare.com/learning/serverless/why-use-serverless/. Luettu 13.2.2022.

Coupaye, T. & Estublier, J. 2000. "Foundations of enterprise software deployment," Proceedings of the Fourth European Conference on Software Maintenance and Reengineering. s. 65–73.

Culverhouse, T. 2018. AWS Fargate. Luettavissa: https://www.techtarget.com/searchaws/definition/AWS-Fargate. Luettu 28.3.2022.

Dearle, A. 2007. Software Deployment, Past, Present and Future. Future of Software Engineering, s. 269–284.

Ek, J. 2020. Ohjelmistoalan toimialaraportti 2020. Luettavissa: http://urn.fi/URN:ISBN:978-952-327-493-8. Luettu 27.1.2022.

GeeksforGeeks, 2019. Introduction to AWS Elastic Beanstalk. Luettavissa: https://www.geeksforgeeks.org/introduction-to-aws-elastic-beanstalk/. Luettu 28.1.2022.

Grey, T. 2019. 5 principles for cloud native architecture – what it is and how to master it. Luettavissa: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it. Luettu 19.3.2022.

Hayes, B. 2008. Cloud computing. Communications of the ACM. Luettavissa: http://doi.acm.org/10.1145/1364782.1364786. Luettu 14.3.2022.

Hiltunen, J. 2010. Palvelimen virtualisointi. Luettavissa: https://urn.fi/URN:NBN:fi:amk-2010121518366. Luettu 19.3.2022.

IBM. 2021. Containers vs. Virtual Machines (VMs): What's the Difference? Luettavissa: https://www.ibm.com/cloud/blog/containers-vs-vms. Luettu 19.3.2022.

Klein, E. 2019. A Guide to the World of Cloud-Native Applications. Luettavissa: https://logz.io/blog/cloud-native-applications/. Luettu 22.3.2022.

Lange, K. 2021. What's Serverless? Pros, Cons & How Serverless Computing Works. DevOps Blog. Luettavissa: https://www.bmc.com/blogs/serverless-computing/. Luettu 11.3.2022.

Long, K. 2021. In the 15 years since its launch, Amazon Web Services transformed how companies do business. The Seattle Times. Luettavissa: https://www.seattletimes.com/business/amazon/in-the-15-years-since-its-launch-amazon-web-services-has-transformed-how-companies-do-business/. Luettu 14.3.2022.

McGrath G. & Brenner P. R. 2017 Serverless Computing: Design, Implementation, and Performance.

Mikovic, I., Lolić, T., Stefanović, D. & Sladojevic, S. 2017. Utilizing web and cloud-based technologies to support corporate business operations. XIII International May conference on strategic management IMKSM 2017.

Oikarinen, J. 2021. Virtualisointi. Luettavissa https://urn.fi/URN:NBN:fi:amk-2021111820583. Luettu 19.3.2022.

Pramanik, D. 2021. Cloud Computing Vs Traditional IT Infrastructure Models – What is the Difference? Luettavissa: https://www.cloudcodes.com/blog/conventional-it-infrastructure.html. Luettu 19.3.2022.

Red Hat. 2019a. What is IT infrastructure? Luettavissa: https://www.redhat.com/en/topics/cloud-computing/what-is-it-infrastructure. Luettu 27.1.2022.

Red Hat. 2019b. What is cloud infrastructure? Luettavissa: https://www.redhat.com/en/topics/cloud-computing/what-is-cloud-infrastructure. Luettu 27.1.2022.

Red Hat. 2018. Understanding cloud-native applications. Luettavissa: https://www.redhat.com/en/topics/cloud-native-apps. Luettu 14.3.2022.

Red Hat. 2020a. What is a hypervisor? Luettavissa: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor. Luettu 19.3.2022.

Red Hat. 2020b. Containers vs VMs. Luettavissa: https://www.redhat.com/en/topics/containers/containers-vs-vms. Luettu 19.3.2022.

Roberts, M & Chapin J. 2017. What is Serverless? Symphonia.

Sumo Logic. s.a. What is Cloud Infrastructure? Luettavissa: https://www.sumologic.com/glossary/cloud-infrastructure/. Luettu 1.2.2022.

Tozzi, C. 2021. What Is Serverless Computing? Luettavissa: https://www.itprotoday.com/serverless-computing/what-serverless-computing. Luettu 28.3.2022.

Wittig A. & Wittig M. 2016. Amazon Web Services in Action. Manning. Shelter Island.