jamk

Serverless-sovelluksen siirrettävyys ja mukautuva arkkitehtuuri

Marika Näivö

Opinnäytetyö, AMK
Joulukuu 2022
Tietojenkäsittely ja tietoliikenne

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Näivö, Marika

Serverless-sovelluksen siirrettävyys ja mukautuva arkkitehtuuri

Jyväskylä: Jyväskylän ammattikorkeakoulu. Joulukuu 2022, 106 sivua.

Tietojenkäsittely ja tietoliikenne. Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: Kyllä

Tiivistelmä

Serverless-sovellukset ovat voimakkaasti sidoksissa pilviympäristöönsä. Pilviympäristöjen erot, alan jatkuva kehittyminen ja samalla standardien puute tekevät kuitenkin sovelluksen siirtämisestä toiseen pilviympäristöön haasteellista. Sovellusten lukittuminen pilviympäristöönsä on kuitenkin herättänyt myös kysymyksiä ja pohdintoja koskien tilannetta, jossa haluttaisiin vaihtaa pilviympäristöä.

Tutkimuksen tavoitteena oli selvittää, miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys ovat merkityksellisiä asioita sekä tutkia miten eri pilviympäristöt vaikuttavat sovelluksen suunnitteluun ja toteuttamiseen, kun sovelluksen tulee olla helposti siirrettävissä pilviympäristöstä toiseen. Lisäksi tavoitteena oli selvittää, millaisia yleisiä haasteita edelliseen tavoitteeseen liittyy. Ongelman ratkaisemiseksi tavoitteeksi asetettiin löytää soveltuva arkkitehtuurimalli, jonka avulla voidaan toteuttaa siirrettävä serverless-sovellus.

Tutkimus tehtiin kvalitatiivisena tutkimuksena, koska tavoitteena oli myös saavuttaa riittävä ymmärrys serverless-teknologian ja pilviympäristön muodostamasta kokonaisuudesta. Siirrettävyyden ongelman ratkaisemisessa tarvitsee myös ymmärtää organisaatioiden tarpeet ja vaatimukset. Näiden selvittämistä varten luotiin menetelmä, jonka kautta saadut tiedot ohjasivat siirrettävyyteen liittyvien toteutusvaihtoehtojen selvittämisvaihetta. Toteutusvaihtoehtojen selvittäminen aloitettiin syvemmällä perehtymisellä serverless-teknologiaan ja valittujen pilvipalveluntarjoajien ympäristöihin.

Tuloksena saatiin hyvä yleiskuva kahden erilaisen pilviympäristön vaikutuksesta siirrettävän serverless-sovelluksen arkkitehtuurin suunnitteluun ja sovelluksen toteuttamiseen. Tutkimuksessa löydettiin arkkitehtuurimalli, joka mahdollisti sovelluksen mahdollisimman vaivattoman siirrettävyyden. Lisäksi laadittiin esimerkkisovellus tutkimustulosten ja johtopäätöksien perusteella, jota voidaan käyttää työvälineenä, kun toteutetaan siirrettäviä serverless-teknologiaa käytettäviä sovelluksia. Havaittiin, että haasteista ja rajoituksista huolimatta lukittumisen ongelma on ratkaistavissa. Saavutettu hyvä siirrettävyys nopeutti kehitystyötä, nosti sovelluksen laatua, teki siitä laadukkaammin testattavan ja vaikuttaa myös siihen, miten voidaan hyödyntää laajemmin ja paremmin monipilven mahdollisuuksia sekä käyttää parhaiksi koettuja eri pilvipalveluntarjoajien palveluita tai kun halutaan käyttää samanaikaisesti useita eri pilviympäristöjä.

Avainsanat (asiasanat)

Serverless, Monipilvi, Lukittuminen, Pilvipalvelu, FaaS, BaaS, Ohjelmistoarkkitehtuuri

Muut tiedot (salassa pidettävät liitteet)

Ei salassa pidettäviä liitteitä.

jamk

Description

Näivö, Marika

Serverless application portability and adaptative architecture

Jyväskylä: JAMK University of Applied Sciences, December 2022, 106 pages.

Information and Communications. Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

Serverless applications are strongly tied to their cloud environment. The differences in cloud environments, the continuous development of the industry and the lack of standards make moving the application to another cloud environment challenging. Locking the applications with their cloud environment has also raised questions and considerations regarding a situation where one would like to change the cloud environment.

The aim of the research was to find out why the adaptability of the serverless application architecture and the portability of the application are relevant issues and to investigate how different cloud environments affect the design and implementation of the application, when the application must be easily portable from one cloud environment to another. In addition, the goal was to find out what kind of general challenges are related to the previous goal. To solve this problem the goal was to find a suitable architecture model that can be used to implement a portable serverless application.

The study was conducted as a qualitative study, because the goal was also to achieve a sufficient understanding of serverless technology and the cloud environment as a whole. To solve the problem of portability, one also needs to understand the needs and requirements of organizations. To solve these issues, a method was created, through which the information obtained guided the investigation phase of implementation alternatives related to portability. Finding out the implementation options started with a deeper understanding of serverless technology and the environments of selected cloud service providers.

The result was a good overview of the impact of two different cloud environments on the architectural design of a portable serverless application and to implement the application. In the research, an architectural model was found that enabled the application to be portable as easily as possible. In addition, an example application was made based on the research results and conclusions, which can be used as a tool when implementing transferable applications using serverless technology. It was found that despite the challenges and limitations, the problem of being locked in can be solved. The good portability achieved accelerated the development work, increased the quality of the application, made it more testable and also affects how it is possible to make better use of the possibilities of multi-cloud and to use the best services from different cloud service providers or when you want to use several different cloud environments at the same time.

Keywords/tags (subjects)

Serverless, Multi-cloud, Vendor lock-in, Cloud service, FaaS, BaaS, Software architecture

Miscellaneous (Confidential information)

-

Sisältö

Sanasto **7**
1 Serverless: Pilvisovellusten kehitystyö muuttui **8**
1.1 Johdanto 8
1.2 Tavoite ja tutkimusasetelma 9
1.3 Aiemmat tutkimukset aiheesta 11
2 Serverless-arkkitehtuuri **13**
2.1 Serverless-arkkitehtuurin peruspiirteet 13
2.2 Haasteena kaksi eri tapaa: FaaS ja BaaS 14
3 Serverless-teknologia ja monipilviympäristö **18**
3.1 Lukittumisen pelosta parhaan toimittajan valintaan 18
3.2 Teknologiavalinta ja lukittumisriskiin liittyvien kysymysten huomiointi 19
3.3 Monipilviympäristö ja serverless-sovelluksen siirrettävyys 21
4 Tarpeet, vaatimukset ja tavoitteet ohjaavat pilviympäristövalintaa **24**
4.1 Ratkaisuna monipilviympäristö: Tarpeiden sekä vaatimusten tunnistaminen 24
4.2 Tavoitteet muodostuvat erilaisista tarpeista ja lähtökohdista 28
4.3 Valittujen tavoitteiden merkitys toteutusvaihtoehtojen kartoittamisessa 31
5 Erilaisten pilviympäristöjen vaikutus serverless-sovellukseen **33**
5.1 Serverless ja erilaiset pilviympäristöt 33
5.2 Eri pilvipalveluja hyödyntävän sovelluksen suunnittelu ja toteuttaminen 35
5.3 Serverless on tapahtumapohjainen teknologia 37
5.4 AWS Lambda ja tapahtumien käsittely 38
5.5 Azure Functions ja tapahtumien käsittely 41
6 Siirrettävän sovelluksen toteuttaminen: Tarjolla eri vaihtoehtoja **42**
6.1 Ohjelmistokehys apuna mukautumiskykyisen sovelluksen kehitystyössä 43
6.2 Vaihtoehtoisen ratkaisun etsiminen pilvipalveluntarjoajan palvelun tilalle 44
6.3 Serverless-sovelluksen arkkitehtuurin suunnittelu 45
6.4 Muut keinot, jotka helpottavat siirrettävän sovelluksen toteutusvaihetta 48
7 Tutkimustulokset ja johtopäätökset **50**
7.1 Arkkitehtuurin mukautumiskyvyn ja sovelluksen siirrettävyyden merkitys 50
7.2 Siirrettävän sovelluksen suunnitteluun ja toteuttamiseen liittyvät haasteet 52
7.3 Valittujen tavoitteiden täyttyminen tutkimustuloksissa 54
7.4 Millainen siirrettävä serverless-sovellus ja sen arkkitehtuuri voi olla 56
7.5 Toteutusvaihtoehtojen vaikutus valintojen tekemiseen 57

8 Esimerkkisovelluksen suunnittelu ja toteutus ... 61
8.1 Sovelluksen pilviagnostinen arkkitehtuuri ... 61
8.2 Esimerkkisovelluksen tarkoituksen kuvaus ja toiminnot ... 62
8.3 Hexagonal-arkkitehtuurimallin soveltaminen esimerkkisovelluksessa ... 63
8.4 Sovelluksen toiminnot hexagonal-arkkitehtuurissa ... 64
8.5 Tarkempaan tarkasteluun valitut sovelluksen toiminnot ... 68
8.6 Tiedosto- ja käännöspalvelua sekä tietokantaa käyttävä sovelluksen toiminto ... 70
8.7 Eri palveluja käyttävä ja toista serverless-funktiota kutsuva toiminto ... 81
9 Pohdinta ... 85
9.1 Tutkimuksen lähtökohdat ja tutkimustulokset ... 86
9.2 Tulosten merkitys ja soveltamismahdollisuudet ... 96
9.3 Aiemmat tutkimukset ja tutkimuksen tulosten suhde niihin ... 97
9.4 Tulosten luotettavuus, kehittämisehdotukset ja jatkotutkimusmahdollisuudet ... 99
Lähteet ... 102

Kuviot

Kuvio 1. Esimerkki serverless-sovelluksen funktioiden toiminnasta ... 14
Kuvio 2. Serverless-arkkitehtuurin vaihtoehdot ja niihin liittyviä palveluita ... 15
Kuvio 3. Serverless-sovellus verrattuna perinteiseen monoliittiseen sovellukseen ... 16
Kuvio 4. Tapahtumapohjaisuus AWS-ympäristössä ... 36
Kuvio 5. Esimerkki JavaScriptillä toteutetusta funktiosta AWS Lambda -alustalla ... 40
Kuvio 6. Sovellus kutsuu AWS Lambda -alustalla olevaa funktiota ... 40
Kuvio 7. Esimerkki funktiosta Azure Functions -alustalla, joka käyttää HTTP triggeriä ... 42
Kuvio 8. Hexagonal-arkkitehtuuria käyttävä sovellus AWS Lambda -alustalla ... 47
Kuvio 9. Esimerkki CloudEvents-standardia käyttävästä tapahtuman kuvaamisesta ... 50
Kuvio 10. Pilviagnostinen lähestymistapa, jossa sovellus voidaan julkaista eri ympäristöissä ... 62
Kuvio 11. Esimerkkisovelluksen toiminnot ja käytössä olevat pilviympäristöjen palvelut ... 63
Kuvio 12. Sovelluksen toiminnot: Tiedoston luonti, tekstisisällön käännös ja tiedostohaku ... 65
Kuvio 13. Sovelluksen toiminnot: Tietojen käsittelyt, viestien muodostus ja lähettäminen ... 65
Kuvio 14. Serverless-funktio, joka lukee tiedoston sisällön ja kääntää sen halutulle kielelle ... 70
Kuvio 15. Trigger-tehtävän hoitava AWS API Gateway Trigger adapteri ... 71
Kuvio 16. Trigger-tehtävän hoitava Azure Http Trigger adapteri ... 72
Kuvio 17. Funkion suorituksen käynnistymiseen liittyvän adapterin HTTP Trigger Port ... 73
Kuvio 18. Tiedoston tekstisisällön käännöstoimintoon liittyvä ydinkomponentti ... 75

Kuvio 19. Käännöstoiminnon ydinkomponentista kutsutaan porttia 76
Kuvio 20. Esimerkkisovelluksen Translator port -komponentti 77
Kuvio 21. Amazon Translate -palveluun liittyvä adapteri 78
Kuvio 22. Azure Translator -palveluun liittyvä adapteri 79
Kuvio 23. Serverless-funktio, joka kokoaa tiedoista viestin ja kutsuu toista funktiota 81
Kuvio 24. Portti välittää adapterille tiedon minkä nimistä serverless-funktiota kutsutaan 83
Kuvio 25. AWS Function Caller -adapteri 84
Kuvio 26. Azure Function Caller -adapteri 85

Taulukot

Taulukko 1. Monipilviympäristön hyödyntämiseen liittyvien tavoitteiden kartoitus 29

Sanasto

Amazon AWS	Pilvipalveluntarjoaja.
AWS Lambda	Omien FaaS-tavalla toteutettujen sovelluksen funktioiden ajoympäristö pilvipalveluntarjoajan palvelussa.
Backend as a Service, BaaS	Serverless-sovellus, joka on vahvasti riippuvainen sovelluksen kannalta ulkopuolisen toimijan tarjoamista ratkaisuista, käyttää BaaS-palveluja.
Event-driven architecture	Serverless-teknologia perustuu tapahtumapohjaisuuteen. Tapahtumapohjainen arkkitehtuurimalli painottaa johonkin tapahtumaan liittyvää vastausta tai toimintaa perustuen tapahtumiin.
Function as a Service, FaaS	Serverless-arkkitehtuurin toinen mahdollistama tapa toteuttaa funktio. Serverless-sovelluksen funktio, jonka toiminnot eli koodi on toteutettu itse. Otetaan käyttöön tarpeen mukaan lyhytaikaisesti pilvessä.
Microsoft Azure	Pilvipalveluntarjoaja.
Microsoft Azure Functions	Omien FaaS-tavalla toteutettujen sovelluksen funktioiden ajoympäristö pilvipalveluntarjoajan palvelussa.
Multi-cloud, multicloud	Monipilvi. Monipilviympäristö tarkoittaa vaihtoehtoja, jotka muodostuvat eri pilvipalveluntarjoajista ja itse pilvistä, joissa sovelluksia ajetaan.
Serverless architecture, Serverless	Serverless-arkkitehtuuri, ”palvelimeton” arkkitehtuuri.
Vendor lock-in	Sovelluksen lukittuminen tiettyyn pilvipalveluntarjoajan palveluun tai sovelluksesta saatavan toiminnallisuuden lukittuminen tietyn palveluntarjoajan palveluihin.

1 Serverless: Pilvisovellusten kehitystyö muuttui

Serverless-teknologia on monia innostava mahdollisuus, joka on tullut pilvipalvelujen suosion ja kehityksen myötä suosituksi. Moni harkitseekin siirtymistä tähän kiinnostavaan arkkitehtuuriin, ellei ole jo siirtynyt. Pilvipalveluntarjoajilla on valmis ja helposti käyttöönotettavissa oleva serverless-alusta, jonka päällä sovelluksia voidaan ajaa.

1.1 Johdanto

Pilvipalvelujen yleistymisen kautta sovellusten kehittäminen on muuttunut ja tuonut mukanaan uusia mahdollisuuksia. Uudet aiempaa kovemmat palveluihin, järjestelmiin ja yksittäisiin sovelluksiin liittyvät odotukset ja vaatimukset ovat muuttaneet monilla tavoin sovellusten kehitystyötä koko ajan. Kokonaisia uusia arkkitehtuureja syntyy myös, kuten pilvipalvelujen myötä serverless-arkkitehtuuri (serverless architecture) (Vega 2019). Nykyisin on jo myös tavallista, että sovellukset joko jollain tapaa jo toimivat tai voisivat toimia monipilviympäristössä (multi-cloud). Myös organisaatiot voivat saavuttaa selkeitä hyötyjä, kun tehdään perusteltu ja huolella suunniteltu valinta käyttää monipilviympäristöä serverless-arkkitehtuuria hyödyntävissä ratkaisuissa (McLellan 2019).

Serverless-sovellusten arkkitehtuurin ja toteutuksen mukautumiskyky eri pilvipalvelualustoille on herättänyt kysymyksiä, mutta myös pelkoja rajoittavasta sitoutumisesta tiettyyn ympäristöön, josta eroon pääseminen voi olla joko kallista ja hyvin vaikeaa (Tanasa 2019). Näihin kysymyksiin on haluttu luonnollisesti löytää vastauksia sekä on haluttu löytää tapoja ratkaista sovellusten mukautumiseen liittyvä ongelma tai vähintään ymmärtää, mitä vaikutusta tehdyillä valinnoilla on sovelluksen tai järjestelmän kannalta. Niin kehitystyön aikana kuin määrittely- ja suunnitteluvaiheissa tehdään valintoja, joiden seuraukset seuraavat mukana tyypillisesti pitkään. Keinot, joiden ansiosta ei lukittauduta yhteen pilvipalveluntarjoajaan, ovat herättäneet ymmärrettävää mielenkiintoa erityisesti kustannusvaikutusten ja liiketoimintahyötyjen näkökulmista. On selkeä etu, jos sovelluksen arkkitehtuuri kykenisi mukautumaan tarvittaessa moneen eri pilviympäristöön eli sovellus voitaisiin suunnitella ja toteuttaa niin, että se voidaan ottaa käyttöön eri pilvipalveluntarjoajien ympäristöissä mahdollisimman pienin ponnisteluin. Pilvipalveluntarjoajien serverless-alustoilla ja muilla palveluilla ei ole yhteistä standardia tapaa toimia. Palveluja käyttävät järjestelmät muuttuvat helposti epäyhteensopiviksi toisiin palveluihin nähden, kun tarkastellaan tilannetta, jossa haluttaisiinkin vaihtaa pilvialustaa (What is serverless? 2022).

1.2 Tavoite ja tutkimusasetelma

Tavoitteena oli selvittää, miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys toiseen pilviympäristöön ovat merkityksellisiä asioita sekä tutkia, miten vahvasti pilvipalveluntarjoajan ympäristöön sitoutuneen serverless-sovelluksen arkkitehtuurin suunnittelu voidaan toteuttaa niin, että sen avulla kehitetty sovellus voidaan siirtää helposti pilviympäristöstä toiseen. Tavoitteena oli myös selvittää millaisia haasteita ja vaihtoehtoja edellisen tavoitteen saavuttamiseen liittyy. Monipilviympäristö tarkoittaa ympäristöä, jossa pilvialustat ovat erilaisia ja monipilvikysymystä voi myös lähestyä eri näkökulmista kuin mikä tutkimuksessa valittiin lähtökohdaksi. Esimerkkinä toiseen pilveen voidaan myös haluta siirtää vain jokin osa sovelluksesta, jolloin hyödynnetään samanaikaisesti useaa eri pilviympäristöä. Opinnäytetyön tavoitteiden saavuttamiseksi määriteltiin seuraavat tutkimuskysymykset:

- Miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys toiseen pilviympäristöön on merkittävä asia?
- Mitä yleisiä haasteita liittyy toiseen pilviympäristöön mahdollisimman vaivattomasti siirrettävissä olevan serverless-sovelluksen suunnitteluun ja toteuttamiseen?
- Miten serverless-sovelluksen arkkitehtuuri voidaan suunnitella ja sovellus toteuttaa, kun halutaan lähestyä monipilviympäristökysymystä sovelluksen siirrettävyyden näkökulmasta?

Tutkimusmenetelmänä käytettiin kvalitatiivista eli laadullista tutkimusta. Tämän kaltainen tutkimus tuottaa tietoa aiheesta kuvaten sen kattavasti ja tuottaa tarvittaessa ratkaisun ongelmaan. Laadullinen tutkimus soveltuu parhaiten, kun ilmiötä ei tunneta ja kattavan kuvauksen lisäksi halutaan löytää syvällinen näkemys tutkimuskohteesta. Tutkittavan ilmiön syvällinen ymmärtämisen on siis mahdollista laadullisen tutkimuksen avulla. (Kananen 2015, 70-71.)

Laadulliseen tutkimukseen kuuluu luoda tutkittavaan ongelmaan liittyvä ratkaisu tai saavuttaa ongelmaa koskeva ymmärrys, varsinaista käytännön työtä ongelman varsinaiseksi poistamiseksi ei kuitenkaan toteuteta (Kananen 2015, 29). Kvalitatiivinen eli laadullinen tutkimus soveltuu tämän opinnäytetyön aiheeseen ja tavoitteeseen, koska tavoitteena oli perehtyä valitun teknologian taustalla olevaan keskeiseen tietoon, eri pilvipalveluihin sekä valitun teknologian ja erilaisten pilviympäristöjen nykypäivän mahdollisuuksiin. Työn tavoitteena oli tuottaa tietoa, jota on mahdollista soveltaa käytännön sovelluskehitystyössä sekä myös isomman kuvan kannalta osana

organisaatioiden muuta toimintaa. Kuten laadulliseen tutkimukseen kuuluu, työssä pyritään kuvaamaan ensin aiheeseen liittyvä tietoperusta kattavasti, jotta tutkimusongelman ymmärtäminen on mahdollista ja tutkimuksen aiheen kannalta keskeiset käsitteet tulevat tutuiksi. Tietoperusta tuo riittävästi pohjatietoa aiheesta sekä antaa näiden jälkeen perustelut sille, miksi tutkimusongelma on olemassa ja mikä on tutkimusongelman laajempi merkitys.

Tutkimuksen aineiston kerääminen

Laadullisen tutkimuksen tutkimusasetelmassa tulee esimerkiksi kertoa, miten aineiston kerääminen hoidetaan ja perustella valitut keruumenetelmät sekä myös dokumentaatiotyön hoitamisesta tulee kertoa ja esitellä analysointimenetelmät perustelujen kera (Kananen 2015, 379). Aineiston keräämisessä laadullisessa tutkimuksessa voi käyttää välineinä havaintojen tekemistä, dokumenttien laatimista, haastattelujen tekoa tietyistä teemoista ja kyselyitä (Kananen 2015, 20). Aineiston analysointimenetelmiä taas ovat esimerkiksi sisältöanalyysi ja mallintaminen (Kananen 2015, 65).

Laadullinen tutkimuksen luotettavuuden arviointi on erilainen kuin kvantitatiivisessa tutkimuksessa, koska käsitteitä reliabiliteetti ja validiteetti ei voi soveltaa suoraan. Laadullisessa tutkimuksessa arvioidaan tulosten luotettavuutta/totuudellisuutta, ilmiön ymmärtämistä (siirrettävyyttä/ sovellettavuutta), pätevyyttä ja toistettavuutta, vahvistettavuutta ja saturaatiota (jossa ei löydy enää uutta tietoa, vaan aineisto alkaa toistaa itseään, kun tiedon lähteitä on olemassa useita). (Kananen 2015, 352-355.)

Lähteet on valittu sen perusteella, että tutkimuksessa käytettävät tiedot ovat mahdollisimman ajankohtaisia, tekijää voidaan pitää asiantuntijana ja siten, että niiden kautta voidaan saavuttaa riittävä ymmärrys aiheesta ja selkeän kokonaiskuvan muodostaminen mahdollistuu. Tiedonhaussa on käytetty apuna aiheeseen liittyviä keskeisiä käsitteitä. Aineiston laatuun kiinnitetään myös huomiota sillä, että löydetty tieto pyritään vahvistamaan etsimällä muita lähteitä, joissa on sama viesti. Kuitenkin yksittäinenkin esitetty tieto tai näkemys voi olla arvokas osa tutkimuksen tulosta, jolloin se otetaan mukaan. Tutkimus tehtiin tutustumalla eri lähteisiin, pyrkimällä saamaan lopulta aikaan tilanne, jolloin ei enää löydy uutta keskeistä tietoa vastauksena tutkimuskysymyksiin.

Aineisto koostuu painetusta kirjallisuudesta sekä erilaisia sähköisessä muodossa olevista tiedon lähteistä. Näitä lähteitä ovat aiheeseen liittyvät aiemmat tutkimukset, asiantuntijoiden laatimat

artikkelit ja muut sähköisessä muodossa olevat julkaisut tai verkkosivustot sekä tutkittavan aiheen kannalta keskeiset tekniset dokumentaatiot. Tutkimuksessa käytetyt lähteet ovat vuosilta 2015-2022. Niitä tutkimuksen kannalta merkittäviä lähteitä (kuten pilvipalveluntarjoajien tuotteita koskevat), joista ei tyypillisesti ole tiedossa tarkkaa tekoajankohtaa voidaan kuitenkin pitää ajantasaisena tietona niiden tekijää edustavan tahon ja lähteiden sisältämien tietojen käyttötarkoituksen vuoksi.

Lähteiden valinta on ajankohtaisuuden lisäksi tehty sen perusteella, miten hyvin niiden kautta saatavat tiedot selventävät työssä esiintyviä käsitteitä sekä miten näistä lähteistä saatavien taustatietojen kautta lähteet antavat vastaukset valittuihin tutkimuskysymyksiin. Käsitteiden ymmärtämistä ja taustatietoja tarvitaan myös, kun pyritään löytämään tutkimuskysymyksiin liittyviä erilaisia tietoja, näkökulmia, ratkaisuvaihtoehtoja ja arvioidaan näiden merkitystä. Laadulliseen tutkimukseen kuuluva osuus, jossa luodaan ongelmaan liittyvä ratkaisu tai ymmärrys ongelmasta, toteutuu soveltuvien ratkaisuvaihtoehtojen perustana olevia lähtökohtia ja ratkaisuvaihtoehtoja esittelevissä luvuissa 4-8.

1.3 Aiemmat tutkimukset aiheesta

Kriteerivaliditeetti on laadulliseenkin tutkimukseen liittyvä keino saada vahvistusta oman tutkimuksen tuloksille. Alan teoria ja muiden tutkimukset antavat keinoja luotettavuuden arviointiin. Aiempia tutkimuksia tulisi olla olemassa ja laadullisen tutkimuksen osalta näin ei välttämättä aina ole. (Kananen 2015, 355.)

Pilvipalveluihin, pilvisovelluksiin ja pilvisovelluksiin liittyviin arkkitehtuureihin, kuten serverless-arkkitehtuuriin, liittyviä peruskysymyksiä on käsitelty useissa lähteissä ja eri näkökulmistakin, kuten esimerkiksi kirjoittajat Vega (2019), Dao (2018) ja Stigler (2018) ovat tehneet. Nopeasti muuttuvalla alalla ajankohtaisiin sisältöihin (kuten serverless-teknologiaan) on syventävämmin perehdytty lähinnä asiantuntijoiden laatimissa artikkeleissa kokonaisten painettujen julkaisujen sijaan. Näissä artikkeleissa on käsitelty serverless-sovellusten mukautumista eri pilviympäristöihin ja tähän näkökulmaan liittyviä ratkaisuvaihtoehtoja, kuten kolmannen osapuolen kehittämiä vaihtoehtoja (Kritikos & Skrzypek 2018, Wilde 2019) ja edelleen jatkaen aiheen käsittelyn syventämistä liittyen monipilviympäristöihin, organisaatioiden käyttämiin strategioihin ja lukittumiseen (vendor lock-in) liittyviin pelkoihin. Kolmen edellisen aiheen muodostamaan kokonaisuuteen eri tavoin

liittyvää tietoa ovat tuottaneet esimerkiksi Goasguen (2019), Vijayan (2018), Tanasa (2019), Kritikos & Skrzypek (2018) sekä McLellan (2019).

Pilviympäristöihin liittyvistä mikropalvelu- ja serverless-arkkitehtuureista on olemassa näitä molempia arkkitehtuureja sovellusten kehitystyössä soveltavia tai aiheita muutoin käsitteleviä opinnäytetöitä useita samoin kuin edellisiin arkkitehtuureihin sekä yleisesti pilvipalveluihin liittyvää painettua kirjallisuuttakin. Serverless-teknologian hyödyntämisen yleisiä vaikutuksia kustannuksiin on myös käsitelty eri tutkimuksissa, kuten Karri Kuivasen tekemässä kandidaatintutkielmassa vuodelta 2019 nimeltä ”Serverless-arkkitehtuuri ja järjestelmäkustannukset” (Jyväskylän yliopisto).

Edelleen yleisemmällä tasolla pysytellen on käsitelty sovelluksen muuttamista serverless-sovellukseksi esimerkiksi niin ikään vuonna 2019 tehdyssä Aleksi Pekkalan maisterivaiheen opinnäytetyössä nimeltä ”Migrating a web application to serverless architecture” (Jyväskylän yliopisto). Esimerkiksi tässä työssä viitataan lyhyesti myös lukittumisongelmaan sekä käsitellään pääasiassa erilaisia suunnittelumalleja ja itse sovelluksen muunnosprosessia serverless-teknologiaa käyttäväksi sovellukseksi. Työssä ei kuitenkaan keskitytä ratkaisemaan lukittumisongelmaa.

Vuonna 2022 valmistuneessa pro gradu -työssään ”Implementation of serverless computing with DSL framework in SWIFT” Lappeenrannan-Lahden Teknillinen Yliopisto LUT:in opiskelija Aleksei Sapitskii tutki Domain Specific Language (DSL) -työkaluja ja ehdottaa uutta DSL-työkalua, joka pyrkii ratkaisemaan pilvipalveluntarjoajaan liittyvän lukittumisongelman sekä lisäksi tarjoamaan luotettavamman tavan luoda ja hallita pilven infrastruktuurin konfiguraatioita.

Serverless-sovelluksia ja monipilviympäristöä on tutkittu 2019 valmistuneessa Stuttgartin yliopiston opiskelija Daniel Haggin pro gradu -tutkielmassa ”Serverless Applikationen in Multi-Cloud-Umgebungen: Architektur und Design Eventbasierter Kommunikation” ja serverless-sovelluksen siirrettävyyttä Business Process Modeling Notation (BPMN) -standardiin perustuvan mallintamisen avulla lukittumisongelman välttämiseksi on Tolunay Yüksel tutkinut vuonna 2022 valmistuneessa pro gradu -tutkiel-massaan ”Standards-based Modeling and Generation of Platform-specific Function-as-a-Service Deployment Packages” (Stuttgartin yliopisto).

2 Serverless-arkkitehtuuri

Sovelluskehittäjän käytössä ovat kaikki pilvipalvelujen mahdollisuudet ilman, että on pakko huolehtia monimutkaisesta taustalla olevasta infrastruktuurista. Kehittäjä laittaa sovelluksen koodin pilviympäristöön, eikä ensin tarvitse edes asentaa mitään tai säätää asetuksia vastaamaan julkaistavaa tuotetta. Sovelluksen julkaisu on sujuvaa ja nopeaa ja sen käyttö voi alkaa heti. Serverless-teknologian mahdollisuudet ovat edellä kuvatun kaltaisessa prosessissa juuri otettu käyttöön. (Stigler 2018, 1; Vega 2019.)

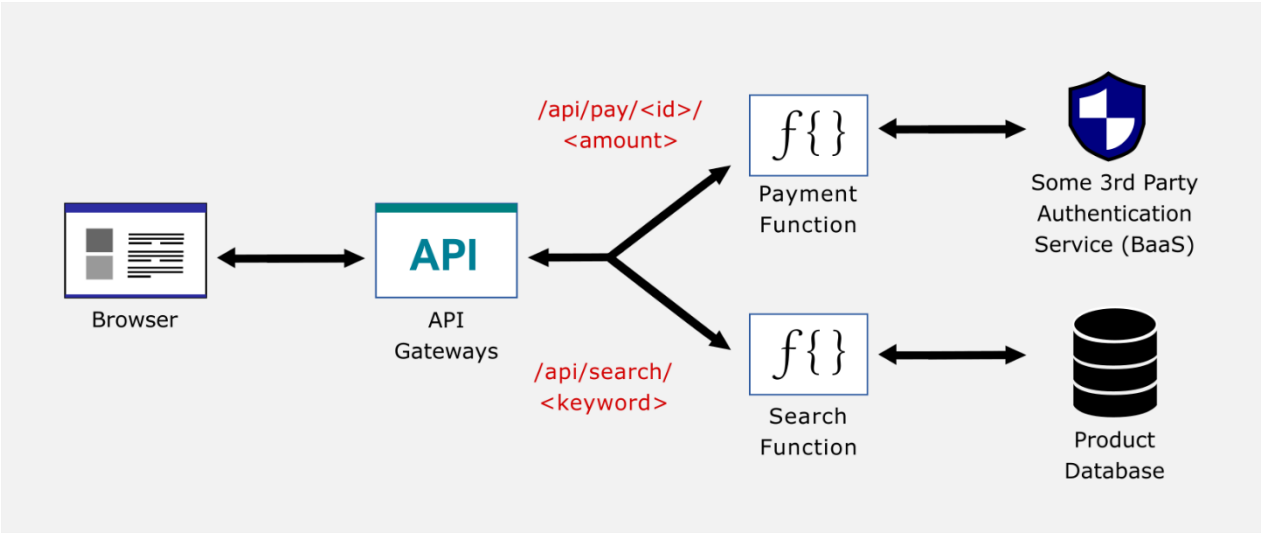
2.1 Serverless-arkkitehtuurin peruspiirteet

Pilvipalveluihin liittyy kaksi arkkitehtuuria, joilla on keskenään samankaltaisia piirteitä: Mikropalvelu- ja serverless-arkkitehtuurit. Tutkimuksessa keskityttiin serverless-arkkitehtuuriin. Käsitteenä serverless computing on pilvilaskentamalli, jossa sovelluksen koodia ajetaan palveluna pilvessä (Stigler 2018, 2). Arkkitehtuurina serverless jakaa sovelluksen pieniin funktioihin, joilla on jokin oma yksittäinen pieni tehtävänsä (Vega 2019). Serverless-sovelluksen funktioiden eli sovelluksen backend-puolen koodin voi ajatella olevan hajautettuna palvelimella eri puolille pienissä osissa ja tämä muuttaa olennaisesti kehitystyötä (De Sogos 2017). Sovellus muodostuu useista erillisistä serverless-funktioista, jotka erillisinä sovelluksen osina pystyvät toimimaan itsenäisesti ja muodostavat tällä tavoin yhdessä laajemman sovelluksen (Vega 2019). Perinteiseen ohjelmistojen arkkitehtuuriin verrattuna, edellä kuvatun kaltainen erilainen arkkitehtuuri vaikuttaa sekä kehittäjien työmäärään että kehitystyöstä syntyviin kustannuksiin aivan järjestelmien käyttöönottoon ja ylläpitoon asti (De Sogos 2017).

Serverless tuo arkkitehtuurina mukanaan erityistä ketteryyttä ja skaalautuvuutta sovelluksiin (Vega 2019; What is serverless? 2022). Serverless-funktiot sijaitsevat pilviympäristössä, jonka ominaisuudet ovat teknologiaa käyttävien sovellusten käytettävissä (Vega 2019). Serverless-sovellusten taustalla toimii pilvipalveluntarjoajan luoma ja ylläpitämä infrastruktuuri, joka kantaa vastuun kaikista niistä toiminnoista, joita tarvitaan sovelluksen suorittamisen aikana (Vega 2019). Nimityksestään serverless huolimatta, ei siis suinkaan tarkoiteta, että taustalla ei olisi lainkaan palvelinta vaan kyse on taustalla olevaan palvelimeen liittyvän infrastruktuurin roolista (Vega 2019; What is serverless? 2022). Kun käytetään tätä teknologiaa, sovelluskehittäjien ei tarvitse huolehtia enää taustalla olevaan infrastruktuuriin liittyvästä hallinnasta (Vega 2019; What is serverless? 2022).

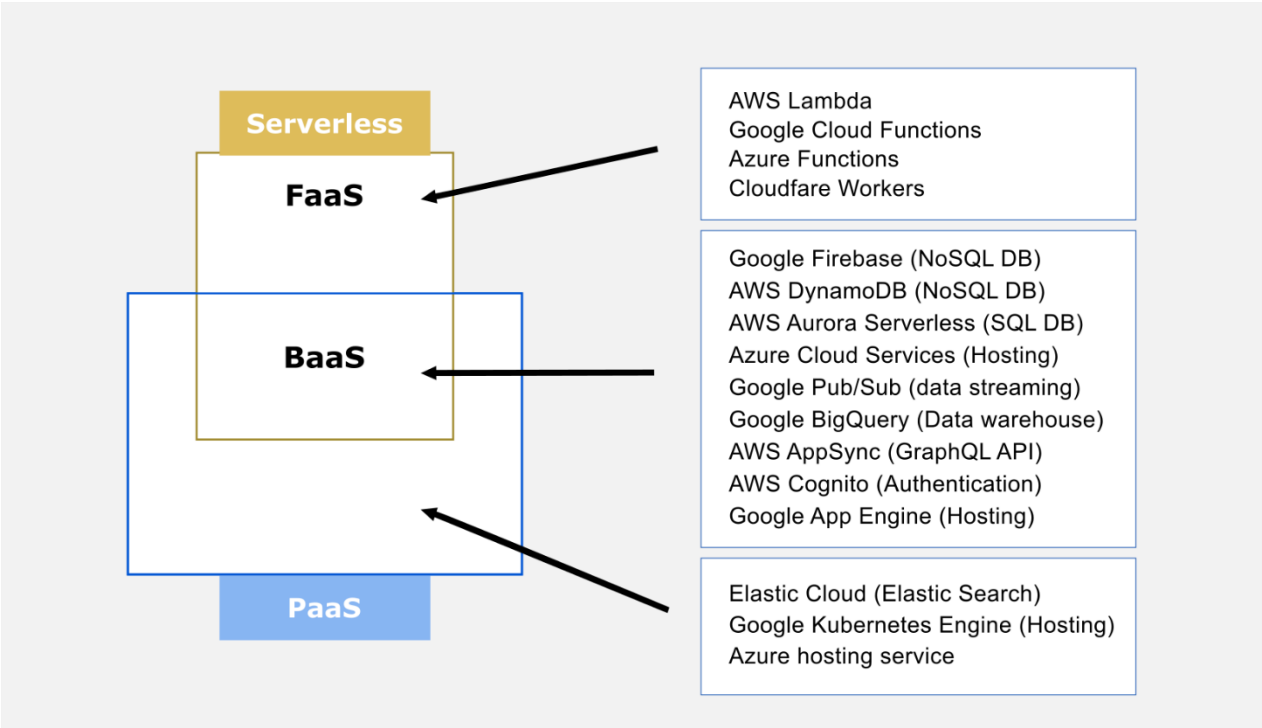
2.2 Haasteena kaksi eri tapaa: FaaS ja BaaS

Serverless-sovelluksen funktioita (ks. kuvio 1) voi toteuttaa pohjautuen kahteen erilaiseen tapaan, joilla molemmilla on omat selkeät merkityksensä ja siten erilaiset vaikutuksensa myös sovelluksen kehitysprosessiin. Nämä tavat, joilla serverless-arkkitehtuuri mahdollistaa sovelluksen toteuttamisen ovat Function as a Service (FaaS) tai Backend as a Service (BaaS). BaaS tarkoittaa sovelluksen näkökulmasta tarkasteltuna erilaisten pilvipalveluntarjoajan tai kolmannen osapuolen toteuttamien valmiiden palvelujen hyödyntämistä kiinteänä osana oman sovelluksen toimintaa (ks. kuvio 2). (De Sogos 2017; Tanasa 2019.)



Kuvio 1. Esimerkki serverless-sovelluksen funktioiden toiminnasta (Wong 2018)

Serverless-sovelluksen toimintaideaan (ks. kuvio 1) kuuluu, että sovellus sisältää konfiguraatioita, joiden pohjalta URL-osoitetta kutsuttaessa HTTP-pyynnöt ohjataan oikealle resurssille (serverless-funktiolle), joka vastaa tähän tiettyyn reittiin liittyvien tapahtumien käsittelystä. Pilvipalveluntarjoajan ympäristössä on API Gateway, jonka avulla hallitaan API-kutsuja, kun kutsutaan Serverless-sovelluksen back end -puolta. API Gateway tehtävä on toimiva serverless-funktion suoritukseen liittyen client-roolissa toimivan selaimen ja pilviympäristön palvelujen välissä. API Gateway sisältää HTTP-palvelimen, joka sisältää tiedot konfiguraatioista. (Wong 2018.)



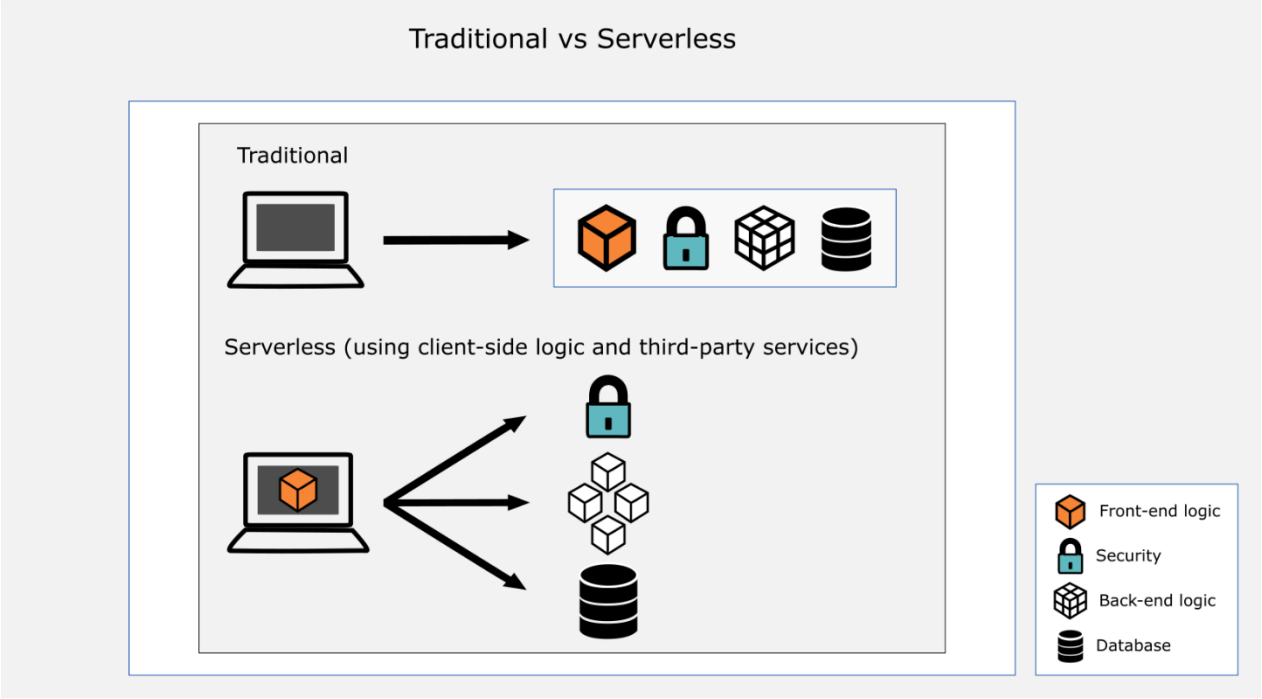
Kuvio 2. Serverless-arkkitehtuurin vaihtoehdot ja niihin liittyviä palveluita (Dao 2018, muokattu)

Backend as a Service (BaaS)

BaaS on pilvipalvelumalli, jossa kaikki web- tai mobiilisovelluksen taustalla olevat osat on ulkoistettu. BaaS-palveluita voi siis käyttää muidenkin kuin mobiilisovelluksen rakentamiseen. BaaS-tapaa käyttävästä sovelluksesta voi siis muodostua sellainen, että kehittäjien tarvitsee huolehtia vain sovelluksen front end -puolen eli käyttöliittymäpuolen toteuttamisesta ja ylläpidosta. BaaS-palveluntarjoajat tarjoavat valmiita sovelluksia erilaisiin toimintoihin, kuten käyttäjien todentamiseen (ks. kuvio 3), tietokannan hallintaan, etäpäivityksiin tai mobiilisovellusten push notifications -ilmoituksiin. BaaS-palvelujen toimittaja tarjoaa tarvittavat API-rajapinnat ja SDK-paketin kirjastojen kautta voidaan integroida kaikki back end -puolen toiminnot ilman, että kehittäjän tarvitsee rakentaa back end -puolta kun käyttää BaaS-palveluita. Kehittäjän tekemä sovellus voi hyödyntää näitä valmiita palveluita, joten kehitystyö nopeutuu. (What is BaaS? | Backend-as-a-Service vs. serverless n.d.)

Sovellus voi olla myös yhdistelmä FaaS- ja BaaS -tapoja, jolloin osa back end -puolesta on toteutettu itse ja osa sovelluksen toiminnoista käyttää valmiita pilvipalveluntarjoajan ympäristöstä löytyviä palveluita. BaaS-palvelujen kautta kehittäjä saa pääsyn pilvipalveluntarjoajan valikoimissa oleviin tiettyihin palveluihin (De Sogos 2017). Kuvio 3 selventää yleisesti serverless-teknologiaa, että BaaS-tapaa käyttävien sovellusten eroa perinteiseen monoliittiseen sovellukseen verrattuna:

BaaS-palveluja käytettäessä kolmannen osapuolen vastuulla voivat olla esimerkiksi tietokannan hallinta ja tiedon tallennukseen liittyvät palvelut sekä turvallisuuteen, kuten käyttäjien autentikointiin liittyvät palvelut (De Sogos 2017; Weston 2017). Kehittäjän tekemä sovellus voi hyödyntää näitä valmiita palveluita (De Sogos 2017). Voimakkaasti pilvinatiivi Serverless-sovellus käyttää pilvipalveluntarjoajan BaaS-palveluja (Tanasa 2019). Näihin kolmannen osapuolen tekemiin palveluihin, kuten vaikkapa pilvipalvelussa olevaan tietokantaan, päästään tavallisesti käsiksi kutsumalla niitä pilvipalveluntarjoajan laatiman palveluun liittyvän erityisen rajapinnan kautta (What is serverless? 2022).



Kuvio 3. Serverless-sovellus verrattuna perinteiseen monoliittiseen sovellukseen (Weston 2017)

Serverless-teknologiaa käyttävässä sovelluksessa kehittäjä rakentaa oman sovelluksen ohjelmakoodin haluamallaan tavalla, kun BaaS-palveluja käyttäen valmiin sovelluksen toimintoihin ei voi vaikuttaa. Serverless-teknologiasta poiketen, kun koko sovelluksen back end -puoli on toteutettu BaaS-lähestymistavalla, näin rakennetut sovellukset eivät normaalisti ole myöskään tapahtumapohjaisia. Serverless-teknologian etuna on myös skaalautuvuus, koska back end -puolen ulkoistaneet BaaS-sovellukset eivät automaattisesti skaalaudu samaan tapaan kuin serverless-sovellukset käytön kasvaessa, jolloin lyhytaikaisesti käytössä olevat esiintymät (instanssit) käynnistetään tarpeen mukaan. (What is BaaS? | Backend-as-a-Service vs. serverless n.d.)

Function as Service (FaaS)

BaaS ei ole kuitenkaan vastaus kaikkiin tilanteisiin. Räätälöityihin omiin ja vaihteleviin tarpeisiin liittyvät erityiset sovellusta koskevat vaatimukset tulee toteuttaa muilla tavoin. Tällöin laaditaan tarpeita ja vaatimuksia vastaava sovelluksen taustalogiikka, joka liitetään valittuun pilvipalveluun. Esimerkiksi pilvipalveluntarjoaja Amazon AWS -pilviympäristön Lambda-alustan avulla serverless-funktioilla voidaan toteuttaa edellä kuvattuja omia erityisiä tarpeita vastaavia monimutkaistakin taustalogiikkaa sisältäviä ja aikaa vieviä toimintoja. Tämän kaltaisia toimintoja ovat esimerkiksi runsaasti laskentatehoa vaativat analysoinnit.

Edellä kuvailtuja itse laadittuja funktioita kutsutaan nimellä Function as a Service ja lyhenteellä FaaS. Näitä funktioita (kuten vaikkapa edellä mainittua AWS Lambda -funktiota) käytettäessä kehittäjän tarvitsee huolehtia vain halutun toiminnon vaatimasta ohjelmakoodista. Amazon AWS -ympäristön Lambda oli juuri se alkuperäinen palvelu, joka toi käyttöön varsinaisesti Serverless-alustat suurelle yleisölle sovellusten kehittäjiä. FaaS-sovelluksia on kuitenkin mahdollista kehittää eri palveluntarjoajien alustoilla, kuten Microsoft Azure Functions ja Google Cloud Functions. (De Sogos 2017.)

FaaS-tapaa käytettäessä itse luodut serverless-funktiot ajetaan konteissa, joiden hallinta on kuitenkin pilvipalveluntarjoajan vastuulla. Funktion käynnistyminen liittyy aina johonkin tapahtumaan ja tapahtumiin reagoidaan automaattisesti sillä hetkellä, kun jotain yksittäistä funktiota tarvitaan suorittamaan joku tehtävä. Kontit ovat tilattomia eli käynnissä vain tarpeellisen ajan, joka on funktion suorittamiseen tarvittava aika. Koko sovellus on mahdollista suunnitella ja luoda käyttämällä näitä FaaS-tavalla toteutettuja funktioita. Sovellus voi myös olla esimerkiksi yhdistelmä Serverless-arkkitehtuuria ja olla muilta osiltaan perinteisemmän mikropalveluarkkitehtuurin komponenteista koostuva. Koska myös FaaS-sovellukset ajetaan konteissa, saadaan kontin kautta funktioille merkittävänä ominaisuutena kyky reagoida tapahtumiin eli käynnistyä automaattisesti, kun sovelluksen funktioita tarvitaan. Kontit myös ovat käynnissä vain sen aikaa kuin tarvitaan, myös aivan lyhytaikaisesti. Konttien kautta syntyvä tilattomuus eli se, että sovellus on käynnissä vain suorittamisen ajan, tekee datan integroimisesta helpompaa. (What is serverless? 2022.)

3 Serverless-teknologia ja monipilviympäristö

Moneen eri pilviympäristöön mukautumiskykyinen sovelluksen arkkitehtuuri ja siirrettävissä oleva sovellus on eri asia kuin monipilviympäristön hyödyntäminen osana kokonaisuutta, mikä merkitsisi samanaikaisesti käytössä olevia erilaisia pilviympäristöjä. Ensimmäisessä tapauksessa viitataan sellaiseen serverless-sovellukseen ja sen arkkitehtuuriin, jolla on tarvittaessa kyky mukautua toiseen, erilaiseen pilviympäristöön.

3.1 Lukittumisen pelosta parhaan toimittajan valintaan

Serverless-teknologian hyödyntämisen historiaan mahtuu monia vaiheita lukittumisen pelosta edeten toiveisiin parhaan pilvipalveluntarjoajan valinnasta: Ilmassa on ollut voimakasta kiinnostusta, epäilyksiä ja nähty samaan aikaan kiinnostavia mahdollisuuksia. Esimerkiksi pilvipalveluntarjoajien käyttövalmiisiin BaaS-palveluihin liittyy kuitenkin selkeiden hyvien puolien lisäksi myös riskejä, jotka liittyvät jopa suureksikin ongelmaksi koettuun sitoutumiseen näitä BaaS-palveluja tarjoavaan pilvipalveluun. Valitsi loppujen lopuksi sitten minkä tahansa teknologian ja palveluntarjoajan, tehty valinta on aina jonkinlainen potentiaalinen toteutumaton riski ja palvelun toimittajan valinta toinen (Vijayan 2018).

Serverless-arkkitehtuuri kiinnostaa yrityksiä, mutta lukittuminen tiettyyn palveluntarjoajaan epäilyttää. Syynä ovat ennen kaikkea kustannukset, joita lukittumisesta voi aiheutua ja palveluntarjoajan vaihtaminen on kallista, jos omat sovellukset ovat kiinteästi kiinni yhdessä palveluntarjoajassa. Ongelma koskee sekä BaaS- että FaaS-ratkaisuja. Lupaus saada sovellusten kehittäminen huomattavasti nopeammin tuottavaksi on saanut aikaan suurta innostusta serverless-arkkitehtuuria kohtaan. Samalla pelko lukittumisesta jarruttaa taustalla ja innostus voi johtaa varovaisuuteen, eikä uuden mahdollisuuden käyttöönottoon ryhdytäkään yhtä innokkaasti. On siis tarpeen ymmärtää, millaisia vaikutuksia lukittumisella on serverless-teknologiaan liittyen ja laatia strategioita, jotta lukittumiseen liittyvä riski olisi hyväksyttävällä tasolla. Teknologiaan liittyvien reaaliteettien selvittämisen avulla, strategialla on tarkoitus saavuttaa tilanne, jossa syntyneen ymmärryksen kautta on mahdollista minimoida lukittumiseen liittyviä riskejä. (Tanasa 2019.)

3.2 Teknologiavalinta ja lukittumisriskiin liittyvien kysymysten huomiointi

Lukittumisriskeihin liittyy perustarve ymmärtää mistä lukittumisriskit sitten käytännössä ja ylipäätänsä oikein muodostuvat. Sekä myös toinen tärkeä lähtökohta tietää, miten lukittumisriskiin on tarpeellista kiinnittää huomiota ennen kuin valitsee serverless-teknologian. Lukittumisen pelko on ymmärrettävää, koska on loogista odottaa, että pilvipalveluntarjoajan vaihtamisesta tulee lisäkustannuksia ja palveluntarjoajan vaihtamiseen tarvittava prosessi voi olla myös kallista (Tanasa 2019). Lukittumisen riskejä pidetään myös syynä, että serverless-teknologiaa ei välttämättä siirrytäkään käyttämään (Edwards 2020).

Mistä lukituksen kustannukset sitten syntyvät? Tanasan (2019) mukaan lukittumisen kustannukset eivät ole puhtaita kustannuksia, vaan myös edut tulisi huomioida. Jos tuote saadaan markkinoille nopeammin, tämä on selkeä etu. Serverless-ratkaisuun liittyvän lukittumisen kustannuksia voidaan myös tarkastella liiketoiminnan näkökulmasta niinkin, että ajatellaankin toisenlaisella tavalla, jolloin myös kustannusten kannalta lopputuloksena voikin olla myös nollakustannus tai jopa voittoakin. Tarvitaan siis ymmärrystä mitä juuri tällä hetkellä mahdollinen muutos voi merkitä liiketoiminnalle. (Tanasa 2019.)

Myös serverless-sovellusten kehityksessä voidaan käyttää apuna ohjelmistokehyksiä. Lukuisista eduista huolimatta ohjelmistokehyksiin liittyy kuitenkin huomioitava riskikysymys, johon tavalla tai toisella todennäköisesti törmätään jossain vaiheessa: Kun johonkin pilvipalveluntarjoajan ympäristöön ilmestyy uusi ominaisuus, milloin valittu ohjelmistokehys tukee sitä? Mitä kaikkia vaikutuksia ohjelmistokehyksen muutoksella tai muutosten viivästymisellä (jos uusi ominaisuus haluttaisiin ottaa käyttöön) voi olla oman sovelluksen kannalta? Vijayanin (2018) siteeraama henkilö arvioi, että avoimen lähdekoodin vaihtoehdot todennäköisesti vaikuttavat pilvialustojen tarjoajiin niin, että niiden on löyhennettävä otettaan. (Vijayan 2018.)

Riskejä aiheuttaviksi kohdiksi nousevat Vijayan (2018) mukaan API-rajapintojen tasolla tapahtuva lukittuminen (jossa API on rajapinta sovelluksen koodin ja palveluntarjoajan infrastruktuurin välillä), pilvipalveluntarjoajien palveluihin lukittuminen sekä käytännössä ennemmin tai myöhemmin ilmenevät käytössä olevien ohjelmistokehysten tuessa ilmenevät puutteet ja näistä aiheutuvat mahdolliset seuraukset koskien pilvipalveluntarjoajien jatkuvasti muuttuvia palveluita. Vijayan

(2018) huomauttaa, että lukittuminen on myös laajempi tilanne, eikä liity vain serverless-teknologiaan tai edes pilvipalveluihin. (Vijayan 2018.)

API-rajapintoja suuremmaksi riskiksi kuitenkin on nostettavissa lukittuminen pilvipalveluntarjoajien palveluihin. Riski on selkeästi havaittavissa, koska sidos on väistämättä tiukka, kun puhutaan pilvipalveluntarjoajien eri palveluista ja serverless-teknologiasta. Pilvipalveluntarjoajilla on lukemattomia palveluja, joiden kanssa sovelluksen koodi on vuorovaikutuksessa tai nämä palvelut voivat vaikuttaa koodin suorittamiseen (käynnistämällä asiakkaan omia koodeja palvelujen kautta). Riskissä on kyse nimenomaan tavassa, jolla sovelluksen koodi kommunikoi ja reagoi kaikkiin palveluihin, joita pilvipalveluntarjoaja tarjoaa. (Vijayan 2018.)

Pilvipalveluntarjoajan tarjoama API on rajapinta pilvipalveluntarjoajan asiakkaan koodin ja toimittajan serverless-alustan välillä. Riski syntyy siitä, että eri serverless-alustat voivat kutsua serverless-funktioita eri tavoilla. Jos sovelluksen koodi on vahvasti sidoksista toimittajan rajapintaan voi sovelluksen siirto toiseen alustaan vaikeutua. Ratkaisuksi yksinkertaisesti kehotetaan ottamaan etäisyyttä rajapintaan. Esimerkiksi AWS Lambda -alustan kanssa voi käyttää sellaisia palveluntarjoajan palvelusta löytyviä ohjelmia, joiden korvaaminen ei välttämättä onnistu muiden osapuolten sovelluksia käyttämällä. Olennaiseksi huomioitavaksi nousevat siis myös serverless-alustan ja sille tehtyjen sovellusten ympärillä olevat muut palvelut, jotka voivat vaikuttaa lukittautumiseen. Lukittuminen voi syntyä myös väistämättä, jos käytetään omaa serverless-sovellusta kerroksena saman pilvipalvelun kahden eri tuotteen välillä. (Vijayan 2018.)

Esimerkiksi BaaS-palvelujen kautta saatavat edut voivat tuoda mukanaan myös varjopuolia, kun on kyse serverless-teknologian käyttämisestä. BaaS-tapaa käyttävät sovellukset tekevät itsestään väistämättä voimakkaasti riippuvaisen palveluntarjoajasta (Tanasa 2019). BaaS-palveluissa suurimmaksi ongelmaksi on koettu lukittuminen tietyn palveluntarjoajan palveluun, koska yhteistä standardia tapaa toimia ei ole olemassa eri pilvipalveluntarjoajien välillä. BaaS-palvelu voi määrittää voimakkaastikin millainen oma BaaS-palvelua käyttävä sovellus voi olla: Tälläkin hetkellä jokainen pilvipalveluntarjoaja on tarkkaan määrittänyt ne keinot, mitä käyttäen on mahdollista olla omasta ohjelmakoodista vuorovaikutuksessa palveluntarjoajien valmiisiin sovelluksiin, eli on toimittava tiukkojen rajoitusten mukaan, jos halutaan käyttää valmiita kolmannen osapuolen komponentteja eli tapaa BaaS (What is serverless? 2022).

Mitä edellä kuvatusta tilanteesta sitten käytännössä voi seurata? Palveluntarjoajien ympäristöistä johtuvat rajoitukset vaikuttavat väistämättä jollain tavoin myös siihen, miten paljon omaa sovellusta voidaan muokata ja kuinka joustavaksi se on mahdollista rakentaa. Kun ajatellaan hetkeä, jolloin haluttaisiinkin vaihtaa pilvipalveluntarjoajaa, ei voida tarkalleen tietää mitä silloin tapahtuu ja miten ympäristön rajoitusten mukaan toteutettu sovellus vaikuttaa tähän prosessiin. Tästä seuraa mitä todennäköisimmin vain kustannuksia, jos on pakko muokata olemassa olevaa järjestelmää uuden palveluntarjoajan ympäristön vaatimusten mukaisesti. (What is serverless? 2022.)

Vijayanin (2018) mukaan olennaisinta on kuitenkin arvioida, kuinka paljon lukittuminen todella merkitsee. Jonkinasteista lukittumista tapahtuu väistämättä, valittiin mitä teknologia hyvänsä, serverless tai joku muu. Projektille valittu ohjelmointikielikin tai tietokanta sitoo johonkin. Jonkinlaisia valintoja kun on loppujen lopuksi pakko tehdä. Riskien toteutumisten todennäköisyyksien ja seurauksien arviointi on pelkoa kannattavampaa. Millä todennäköisyydellä alustaa tulee tarpeen vaihtaa? Millaisilla tilanteissa tämä voisi tulla eteen ja millä todennäköisyydellä näin voisi tapahtua? Jos näin tapahtuisi mitä edes maksaisi siirtyä toisen palveluntarjoajan asiakkaaksi? Mitä vaikutuksia muutoksella olisi liiketoimintaan? (Vijayan 2018.)

3.3 Monipilviympäristö ja serverless-sovelluksen siirrettävyys

Yleiskäsitteenä monipilviympäristö tarkoittaa, että on olemassa erilaisia pilviympäristövaihtoehtoja, joissa sovelluksia voidaan ajaa. Esimerkiksi kun jokin osa järjestelmän muodostamasta kokonaisuudesta on jossain toisessa pilviympäristössä, sovelluksen toteuttamisessa käytettävissä olevat mahdollisuudetkin ovat tämän seurauksena erilaisia. Erilaiset tarjolla olevat mahdollisuudet ovatkin yksi perustelu monipilviympäristöön siirtymiselle tai aiheesta kiinnostumiselle.

Sovelluksen arkkitehtuuriin liittyvän mukautumiskyvyn merkityksen idean voi lainata liiketoiminnan puolelta tulevasta ajattelusta: Kyseessä on valittuun strategiaan liittyvistä tavoitteista. Kaikenlaiset muutostarpeet tulevaisuudessa ovat haaste ja jos arkkitehtuuri kykenee joustamaan eli mukautumaan tarvittaessa saavutetaan selkeä etu. Jos markkinatilanne muuttuu, strategia muuttuu tai jokin muu asia sovellukseen liittyvässä ympäristössä muuttuu, on tarpeisiin perustuvalla mukautumiskyvyllä väliä. (Kokonaisarkkitehtuuri n.d.)

Huomiota kannattaa myös kiinnittää samankaltaisilta kuulostaviin käsitteisiin: Pilviympäristöön liittyvät käsitteet monipilvi ja hybridipilvi eivät kuitenkaan liity samaan asiaan. Hybridipilvellä tarkoitetaan perinteisesti sellaista käytössä olevaa pilveä, jossa on sekä yksityisen että julkisen pilven infrastruktuurit. Sovelluksen käyttöön liittyvät hallintatehtävät ja työkuormien jakaminen näiden pilvien välillä hoidetaan orkestrointityökaluilla. (McLellan 2019.)

Monipilviympäristöstä apua lukittumiseen

Monipilven valitsemiseen liittyvistä syistä yksi suurimmista on halu välttää lukittumista tietyn pilvipalveluntarjoajan palveluun. Lukittumisen seurauksena oma sovellus olisi kiinni tietyn palveluntarjoajan infrastruktuurissa, sen tarjoamissa lisäpalveluissa ja tietysti myös hinnoittelussa. Jos halutaan päästä eroon lukittumisesta ja käytetään voimakkaita toimenpiteitä lukittumisen ehkäisemiseksi, hyvällä tarkoituksella voi olla myös aivan päinvastaisia seurauksia: Voi käydä niin, ettei enää olekaan mahdollista hyödyntää täysipainoisesti valitun pilvipalveluntarjoajan alustan potentiaalia. (McLellan 2019.)

Monipilviympäristöön liittyvää mukautumiskykyä ja sovelluksen siirrettävyyttä voi lähestyä kahdesta näkökulmasta: Luodaan yksittäinen sovellus, joka ei ole riippuvainen mistään tietystä palveluntarjoajasta tai tarkastella pilviympäristöjä siitä näkökulmasta, mikä pilvi on paras vaihtoehto millekin projektille ja asiakkaalle tapauskohtaisesti. Asiakkaalla voi olla kuitenkin projekti, johon halutaan paras mahdollinen markkinoilta löytyvä ratkaisu, joka löytyy joltain toiselta pilvialustalta. Tässä vaiheessa on valittava, halutaanko siirtää koko sovellus toiseen pilviympäristöön vai integroida ainoastaan jokin yksittäinen sovelluksen osa (palvelu) muuhun sovellukseen ns. kolmannen osapuolen palveluna. Palveluna, joka sijaitsee toisessa pilviympäristössä. (Swail 2020.)

Monipilviympäristön hyödyntäminen on ajankohtainen asia ja vaatii strategian kehittämisen

Pilvipalvelujen hyödyntäminen on yleistä. Yhä useampi organisaatio käyttää samaan aikaan sekä useita julkisia että yksityisiä pilviä pilvipalvelujen resursseja hyödyntävien sovellusten käyttöönotossa. Tällä tavoitellaan kahta asiaa: Pyritään välttämään lukittautumista tiettyyn palveluntarjoajaan ja esimerkiksi pyritään hyödyntämään parhaita mahdollisia tarjolla olevia pilvipalveluihin liittyviä ratkaisuja omissa sovelluksissa. Monipilven käyttöön liittyy siis vahvasti tarve strategian olemassaololle: On tehtävä valintoja, kuinka halutaan käyttää eri pilvipalveluntarjoajia. Valinnat

perustuvat vaatimuksiin, joita tulee sekä sovelluksiin liittyvästä teknisestä puolesta että liiketoiminnasta. (McLellan 2019.)

Käyttökelpoisia strategioita päästä eroon lukittumisesta tai ainakin vähentää lukittumisriskiä on kuitenkin monenlaisia. Yleisiä monipilviympäristöön liittyviä selviytymisstrategioita ovat: Siirrettävyyteen panostaminen ja kompromissien teko (McLellan 2019), vastaavan vaihtoehtoisen ratkaisun etsiminen (Tanasa 2019; Yadav 2019), nopeamman kehitystyön kautta käyttöönottoajankohdan aikaistumisena saatava hyödyn maksimointi ja siirtokustannuksiin liittyvä strategia (Tanasa 2019), eri palveluntarjoajien alustojen välisen ohjelmointikielen valinta (Tanasa 2019), siirtokustannuksiin vaikuttaminen erottamalla sovellusalue alustasta suunnittelemalla sovellukselle sopiva arkkitehtuuri (Tanasa 2019) sekä standardisoidun teknologian käyttäminen (Tanasa 2019).

Yksinkertaisimmillaan monipilven voi ajatella tarkoittavan pilvinatiiveja sovelluksia, jotka ajetaan konteissa ja jotka koostuvat mikropalveluista, joissa komponentit sijaitsevat eri pilvipalveluntarjoajien alustoilla. Kontteihin ja mikropalveluihin perustuvat pilvinatiivit sovellukset on mahdollista suunnitella siirrettäviksi eri pilvipalveluntarjoajien välillä. Palveluntarjoajat kuitenkin pyrkivät tekemään alustoistaan sellaisia, että niiden sisältämien palvelujen käyttö johtaa tilanteeseen, jossa vaihtaminen ei ole helppoa eikä välttämättä houkuttelevaakaan, koska eri palveluntarjoajien alustojen sisältämä tarjonta voi ratkaisevasti erota kilpailijoista. (McLellan 2019.)

Pilvipalveluntarjoajien alustojen jatkuva kehitys tuo lisää haasteita myös monipilviympäristöihin liittyen. Monipilviympäristöjen hyödyntämiseen ja valintojen tekemiseen liittyy kuitenkin yksi erittäin suuri haaste: Koska pilvialustat kehittyvät niin nopeasti, nykyiset arviot ja tehdyt valinnat voivat nopeasti osoittautua myös virheellisiksi. Tilanne voisi olla vaikkapa sellainen, että jossain pilviympäristössä ei ole tietynkaltaista haluttua valmista palvelua ja kun mahdollinen muu ratkaisu ollaan ottamassa käyttöön tällä alustalla, onkin sitten jo vastaava tarjolla vaikka vielä jokin aika sitten sitä ei ollut. (Tanasa 2019.)

Monipilviympäristö ei kuitenkaan ole kaikkiin tilanteisiin ja kaikille paras ratkaisu tai mitenkään välttämätön valinta juuri tällä hetkellä. Monipilviympäristön voi pitää myös tulevaisuuden hyödyntämismahdollisuutena, vaikka tällä hetkellä ei nähtäisikään mitään pakottavaa tarvetta ottaa organisaatiossa huomioon. Monipilviympäristöä painottavan strategian sijaan yksi mahdollisuus onkin

keskittyä siihen pilvipalveluntarjoajan alustaan, josta koetaan saatavan eniten hyötyä ja integroida oma sovellus valitun pilvipalveluntarjoajan valikoimista niihin palveluihin, jotka on koettu hyviksi (Vijayan 2018). Hyvien valintojen tekemiseksi tarvitaan tietoa pilviympäristöistä, käytettävistä teknolioista sekä kaikista olemassa olevista tarpeista ja vaatimuksista, joiden kautta valintojen tekeminen helpottuu.

4 Tarpeet, vaatimukset ja tavoitteet ohjaavat pilviympäristövalintaa

Serverless-teknologialle ominaisiin piirteisiin sekä monipilviympäristöön liittyy monia kysymyksiä sekä haasteita, joiden taustalla vaikuttavat erilaiset asiat. Käsitteitä monipilviympäristön hyödyntäminen ja itse monipilviympäristöäkin voidaan konkreettisella toteutustasollakin lähestyä eri näkökulmista. Seuraavaksi tutkimuksessa tarkasteltiin ensin tarpeita ja vaatimuksia, jotka ovat perusteena tavoitteille liittyen monipilviympäristöön mahdollisuutena ja serverless-sovelluksiin.

4.1 Ratkaisuna monipilviympäristö: Tarpeiden sekä vaatimusten tunnistaminen

Tarpeet ja vaatimukset ovat sovelluksen kehitystyöhön ja organisaatioon liittyviä. Millaisiin kysymyksiin monipilviympäristö voisi sitten antaa vastauksia, on kysymys, johon vastausten löytämiseen täytyy nähdä organisaatiossa vaivaa. Aihetta monipilviympäristö voi lähestyä erilaisten tavoitteiden kannalta, joiden perustana on olemassa oleva konkreettinen tarve tai vaatimus. Esiin nousevat tarpeet ja vaatimukset sekä niiden kautta lopulta tarkentuvat tavoitteet voivat syntyä organisaatioissa, kun halutaan löytää ratkaisu havaittuun tilanteeseen sekä samalla haluun saada parhaimmalla mahdollisella tavalla pilviympäristöjen ja teknologioiden erilaiset mahdollisuudet palvelemaan omaa toimintaa.

Tavoitteeksi tutkimuksessa asetettiin tarkastella ensin organisaatioille tyypillisiä tarpeita ja vaatimuksia, joita eri kirjoittajat, kuten Swail (2020) tai McLellan (2019) olivat tuoneet esiin. Sitten havaittiin, että tyypillisiksi luokiteltavissa olevat tarpeet ja vaatimukset käsittelivät eri näkökulmia lähestyä aihetta monipilviympäristö. Eri näkökulmien lisäksi havaittiin, että on olemassa erilaisia tapoja käsitellä monipilviympäristön hyödyntämiseen, monipilviympäristöön siirtymiseen ja siellä toimimiseen liittyviä valmiuksia, haasteita sekä mahdollisuuksia. Havainnoista tunnistettiin kaksi eri lähestymistapaa: Halutaan panostaa sovellukseen siirrettävyyteen tai halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan.

Seuraavat tarpeet ja vaatimukset on jaoteltu kolmeen tarve- ja vaatimusryhmään, joista jokainen keskittyy tiettyyn tarkempaan alueeseen, joiden tarkoitus on auttaa syventämään käsitystä millaisiin kokonaisuuksiin voi ja kannattaa kiinnittää huomiota, kun kartoitustyötä tehdään omassa organisaatiossa. Seuraavat ryhmät muodostettiin tutkimalla ja vertailemalla esiin nousseita havaintoja ja etsimällä niistä yhteisiä piirteitä.

Tarve- ja vaatimusryhmät

Tarpeet ja vaatimukset jaettiin kolmeen eri ryhmään sen perusteella mikä näkökulma sisältyi tunnistettuun tarpeeseen tai vaatimukseen. Ensimmäinen tarve- ja vaatimusryhmä muodostui näkökulmasta, jossa halutaan hallita lukittumiseen liittyvät kysymykset. Näillä lukittumiseen liittyvillä kysymyksillä on suora vaikutus serverless-teknologiaa käyttävän sovelluksen arkkitehtuurin suunnitteluun ja sovelluksen toteuttamiseen:

- Löytää arkkitehtuurin suunnitteluun liittyvä ratkaisu tai ratkaisuja, joilla voidaan ainakin vähentää tai poistaa serverless-sovellusten lukittumiseen liittyvä seuraukset. Arkkitehtuurin suunnittelussa tulee kiinnittää huomiota serverless-sovelluksen siirrettävyyteen. Sovellus voi olla yhdistelmä FaaS- ja BaaS-palveluita ja lukittumisesta aiheutuvat siirtokustannukset vaikuttavat molempiin ratkaisuihin (Tanasa 2019).
- Tiedostetaan, että todennäköisesti luodaan omia FaaS-tapaa käyttäviä sovelluksia ja halutaan selvittää, onko itse serverless-alustan ympärillä sellaisia pilvipalveluntarjoajan palveluita ja niiden rajapintoja, jotka voivat aiheuttaa lukittumisen? (Vijayan 2018.)
- Ymmärtää, että BaaS-tapaa käyttämällä tullaan väistämättä sitoutuneeksi pilvipalveluntarjoajan palveluun (Tanasa 2019).
- Tulee varoa muodostamasta tilannetta, jossa käytettäisiin serverless-sovellusta kerroksena pilvipalvelun kahden eri tuotteen välillä, koska vahva lukittuminen voi syntyä tällä tavoin. (Vijayan 2018.)
- Selvittää tarvittaessa, voidaanko korvata muiden osapuolten sovelluksia käyttämällä sellaisia palveluntarjoajan sovelluksia, jotka aiheuttavat väistämättä lukittumisen tai voimakkaan lukittumisriskin kasvun. (Vijayan 2018.)
- Arkkitehtuurin kautta tulee olla mahdollista käyttää tarvittaessa jotain toista pilviympäristöä, jos niin halutaan esimerkiksi liiketoiminallisista syistä. Toiseen pilviympäristöön voidaan haluta siirtää koko sovellus tai vain jokin osa siitä. Edellä kuvattu tarve voi syntyä esimerkiksi asiakkaan vaatimuksesta käyttää tiettyä pilvipalveluntarjoajaa (Swail 2020).

Seuraava tarve- ja vaatimusryhmä numero kaksi muodostui puolestaan sen näkökulman ympärille, jossa halutaan tietää erityisesti, miten voitaisiin toteuttaa omassa organisaatiossa vallitsevan

lähtökohdan kannalta hyväksi luokiteltavissa oleva serverless-teknologiaa käyttävä kokonaisratkaisu. Tästä ryhmästä tunnistettiin vielä tarkemmin kolme alaryhmää, joista ensimmäinen määriteltiin ”yleisten tarpeiden ja vaatimusten” alaryhmäksi ja kahden muun alaryhmän tarpeet sekä vaatimukset tarkastelevat monipilviympäristökysymyksistä eri lähtökohdista:

- Yleiset tarpeet ja vaatimukset:
    - Etsitään parhaita serverless-sovelluksia tekevän sovelluskehittäjän työtä nopeuttavia ratkaisuja. Serverless-arkkitehtuurin avulla voidaan vähentää järjestelmien monimutkaisuutta ja nopeuttaa suunnittelutyön tekemistä (Tanasa 2019).
    - Etsitään koko prosessia helpottavia ratkaisuja: Sovelluksen kehitystyö, julkaisu, käyttöönotto ja ylläpito osana monipilvistrategiaa. (McLellan 2019.)
- Lähestymistapa 1: Kun halutaan panostaa sovellukseen siirrettävyyteen
    - Halutaan voida luoda sellainen serverless-teknologiaa hyödyntävä ratkaisu, että toteutettu sovellus on tarvittaessa siirrettävissä toiseen pilviympäristöön mahdollisimman vaivattomasti.
    - Halutaan välttää mahdolliset korkeat siirtokustannukset (Tanasa 2019).
- Lähestymistapa 2: Kun halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan
    - Halutaan ottaa käyttöön monipilviympäristö lukittautumisen välttämiseksi (McLellan 2019) ja on päätetty, että halutaan ratkaista asia tällä tavoin.
    - Halutaan käyttää parhaita palveluita eri pilvipalveluntarjoajilta samaan aikaan. Tästä seuraa tarve voida hallinta tapahtumia, jotka syntyvät monipilviympäristössä. Täytyy löytää keino, miten voidaan hallita serverless-sovelluksen tapahtumia monipilviympäristössä, jos tämä lähestymistapa on valittu. Tapahtumien hallintatarve luo tarpeen yhteiselle alustalle (substraattialusta), koska tarvitaan alusta, joka yhdistää sovelluksen komponentit yhteen, kun ne sijaitsevat eri ympäristöissä. (Goasguen 2019.)
    - Halutaan luoda järjestelmä, joka on sekä joustavampi ja turvallisempi että sietokykyisempi. Halutaan selvittää miten monipilviympäristöön liittyvien mahdollisuuksien ymmärtäminen ja huomiointi voi olla tässä avuksi. (McLellan 2019.)
    - Halutaan varmistaa paras mahdollinen suorituskyky käyttämällä useamman pilvipalveluntarjoajan ympäristöjä. Käyttämällä useamman pilvipalveluntarjoajan ympäristöjä voidaan saavuttaa optimaalinen tilanne. Pilvipalveluntarjoajien eri palveluiden aluekohtaisen suorituskyky voi vaihdella. Järjestelmä voi olla toiminnoltaan kriittinen, jolloin palvelun tulee olla saatavissa mahdollisimman luotettavasti. (McLellan 2019.)

Kolmas ja viimeinen tarve- ja vaatimusryhmä korostaa sekä halua tehdä että tarvetta löytää hyviä organisaation tai sen asiakkaan odotuksiin perustuvia ratkaisuja tarkastellen nimenomaan pilvipalveluntarjoajia. Eli kolmannessa ryhmässä on siirrytty arkkitehtuurin tai serverless-teknologian painottamisen sijaan etsimään ratkaisuja, jotka liittyvät eri pilvipalveluntarjoajien ympäristöihin sekä niissä olevien resurssien tarjontaan. Tässäkin ryhmässä halutaan löytää ratkaisuvaihtoehtoja, jotta voidaan tehdä hyviksi luokiteltavissa olevia valintoja. Myös tässä ryhmässä päädyttiin käyttämään eri lähestymistapoihin jakautuneita alaryhmiä, jotka lähestyivät monipilviympäristökysymystä eri lähtökohdista:

- Lähestymistapa 1: Kun halutaan panostaa sovellukseen siirrettävyyteen
    - Kun organisaatio itse, nykyinen asiakas tai mahdollinen uusi asiakas vaatii tai vastustaa jotain tiettyä pilvipalveluntarjoajaa (Swail 2020). Siirrettävyys voi tuoda tähän ratkaisun.
- Lähestymistapa 2: Kun halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan
    - Organisaatio itse, nykyinen asiakas tai mahdollinen uusi asiakas vaatii tai vastustaa jotain tiettyä pilvipalveluntarjoajaa (Swail 2020). Ratkaisuna toisen pilviympäristön käyttö.
    - Tietosuojaa koskevat vaatimukset, voivat edellyttää, että esimerkiksi asiakkaiden tietoja on säilytettävä tietyissä sijainneissa. Ainoa vaihtoehto voi olla tällöin monipilviratkaisu. (McLellan 2019.)
    - Pyritään hyödyntämään parhaita mahdollisia tarjolla olevia pilvipalveluihin liittyviä ratkaisuja omissa serverless-sovelluksissa (McLellan 2019; Swail 2020).
    - Käytetty ympäristö voi koostua useista sekä julkisista että yksityisistä pilvistä. Lukittumisen välttämiseksi sovellus on julkaisu niin, että se käyttää yhdistelmää alan parhaita ratkaisuja. (McLellan 2019.)
    - Halutaan valita parhaat palvelut omaan käyttöön, jos kilpailevalla pilvipalveluntarjoajalla on tarjolla parempi palvelu. (Goasguen 2019.)
    - Seurauksena osa serverless-sovelluksesta voidaan haluta pitää yhdessä pilviympäristössä ja osa jossain toisessa ympäristössä. Esimerkiksi pääosa sovelluksesta on yhden palveluntarjoajan ympäristössä ja toisessa ympäristössä oleva ”best-of-breed” -ratkaisu liitetään integroinnin kautta kolmannen osapuolen palveluna tähän pääasiassa käytettyyn pilviympäristöön. (Swail 2020.)
    - Esimerkiksi järjestelmien suorituskykyä voivat haitata monet pilvipalveluntarjoajista johtuvat seikat, joiden vaikutus halutaan luonnollisesti minimoida. (McLellan 2019.)

Tarve- ja vaatimusryhmien muodostumisen kautta havaittiin, miten ryhmissä mainittujen tarpeiden ja vaatimusten kautta on mahdollista saada määriteltyä perustelut niille valinnoille, joita tehdään, kun määritetään omaan sovellukseen tai järjestelmään liittyvät tavoitteet ja valitaan niiden saavuttamisessa käytettävät toteutustavat. Löydettyjen tarpeiden ja vaatimusten avulla voidaan muodostaa tavoitteet, joita voidaan käyttää työkaluna, kun organisaatioissa valitaan omia toteutettavia tavoitteita, jotka liittyvät monipilviympäristöön ja serverless-teknologian hyödyntämiseen. Tarpeita ja vaatimuksia nousee varmasti edelleen lisää, jos tarkasteluun mukaan tulee lisäksi oman organisaation, oman sovellustuotteen ja/tai asiakkaan tarkempi näkökulma.

Seuraavaksi määritettävät tavoitteet päätettiin muodostaa edellisten tarve- ja vaatimusryhmien sisältöjen pohjalta sekä valittuihin tutkimuskysymyksiin liittyen. Tavoitteiden muodostamisessa päätettiin keskittyä pääasiassa niihin tarpeisiin ja tavoitteisiin sekä kysymyksiin, jotka liittyvät serverless-sovelluksen arkkitehtuurin mukautumiskykyyn ja sovelluksen toteuttamiseen. Sovelluksen suunnittelua ja toteuttamista tarkastellaan siitä tavoitteesta, että sovellus olisi mahdollisimman vaivattomasti ja ongelmitta siirrettävissä johonkin toiseen pilviympäristöön.

4.2 Tavoitteet muodostuvat erilaisista tarpeista ja lähtökohdista

Organisaation ja toteutettavan sovelluksen tarpeista lähtevien eri tavoitteiden tunnistaminen ja niiden valinta on keskeinen kysymys. Merkittävä huomioitava asia omien tavoitteiden määrittelyssä on arvioida, miten voidaan ja miten kannattaa hyödyntää sovelluksen siirrettävyyteen ja/tai moneen eri pilviympäristöön liittyviä mahdollisuuksia. Arvioissa kannattaa ottaa huomioon myös ylipäätänsä pilviympäristössä toimimiseen liittyvät omat valmiudet.

Seuraavat tavoiteryhmien sisältämät tavoitteet (ks. taulukko 1) pohjautuvat määriteltyihin tutkimuskysymyksiin ja tutkimuksessa löydettyihin tarpeisiin ja vaatimuksiin. Näiden pohjalta määritettiin seuraavaksi tavoitteita, joissa korostuvat erilaiset serverless-teknologiaan, siirrettävyyteen ja aiheeseen monipilviympäristö liittyvät näkökulmat. Tavoitteet jaettiin kolmeen eri tavoiteryhmään vastaavien aihepiirien alle kuin mitä tarpeiden ja vaatimusten kartoittamisvaiheessakin tehtiin. Lisäksi tavoitteet jaettiin tarpeiden ja vaatimusten kartoituksen tapaan kahteen tunnistettuun eri päätavoitteeseen: Halutaan panostaa sovellukseen siirrettävyyteen tai halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan.

Monipilviympäristöön usean samanaikaisen pilviympäristön käyttämiseen liittyvä päätavoite määriteltiin tässä tutkimuksessa niin, ettei se liity sovelluksen siirrettävyyteen tai arkkitehtuurin mukautumiskykyyn. Kaikki tutkimuksen aikana esiin nousseet tavoiteryhmään liittyvät ja aihetta monipilviympäristö käsittelevät tavoitteet otettiin tässä vaiheessa mukaan, koska tavoitteiksi haluttiin myös löytää yleisesti tunnistettuja monipilviympäristöjen hyödyntämisen aihepiiriin liittyviä haasteita ja tarpeita. Näin toimittiin, koska havaittiin, kuten tarpeita ja vaatimuksiakin kartoitettaessa, että nyt löydetyille tavoitteilla havaittiin olevan yhteinen piirre: Tavoitteille havaittiin olevan tyypillistä se, että tavoitteet nousevat eri lähtökohdista ja sisältävät oman näkökulmansa aiheeseen.

Tavoitteisiin sisältyvien erilaisten lähestymistapojen havaittiin tuovan samalla esiin mukautumiskykyisen arkkitehtuurin suunnitteluun ja toteuttamiseen liittyvään siirrettävyyteen liittyviä kysymyksiä ja näkökulmia esiin myös epäsuoralla tavalla: Perustellen lisää ja tarkemmin myös sitä kysymystä miksi siirrettävyyteen panostamisella on merkitystä. Vaikka päätavoite ei ollut sama, huomattiin, että molempien näkökulmien sisältämät tavoitteet voivat liittyä myös sekä arkkitehtuurin mukautumiskykyyn että toiseen pilviympäristöön siirrettävissä olevaan serverless-sovellukseen. Molempien näkökulmien tavoitteet tarkastelivat erilaisesta päätavoitteesta huolimatta myös serverless-sovelluksen siirrettävyyteen, suunnitteluun ja toteuttamiseen liittyviä haasteita sekä toteutusvaihtoehtoihin liittyviä kysymyksiä.

Taulukko 1. Monipilviympäristön hyödyntämiseen liittyvien tavoitteiden kartoitus

Tavoiteryhmä 1. Hallita lukittumiseen liittyvät kysymykset, jotka vaikuttavat sovelluksen arkkitehtuurin suunnitteluun	**Tavoiteryhmä 2.** Toteuttaa omien tarpeiden kannalta hyvä serverless-teknologiaa käyttävä kokonaisratkaisu	**Tavoiteryhmä 3.** Tehdä omiin tarpeisiin perustuvia pilviympäristöihin ja niiden palvelutarjontaan liittyviä hyviä valintoja
Yleiset tavoitteet Ymmärretään millainen teknologia serverless on, ja miten sen käyttäminen eri tavoin vaikuttaa lukittumiseen ja lukittumisriskiin (Tanasa 2019; Vijayan 2018). Ymmärretään yhteisen standardin puutteen vaikutus siihen, miten sovelluksen funktiot voivat olla vuorovaikutuksessa kunkin pilvipalveluntarjoajan palveluihin (What is serverless? 2022).	**Yleiset tavoitteet** Löydetään keinoja, jotka helpottavat koko prosessia alkaen sovelluksen kehitystyöstä ja jotka vaikuttavat myös julkaisuun, käyttöönottoon ja ylläpitoon. Ymmärretään serverless-teknologian valinnan vaikutukset: Pienissä osissa olevan sovelluksen komponenttien välillä tapahtuva kommunikointi vaikeutuu, kun monimutkaisuus kasvaa (Vega 2019).	**Päätavoite: Kun halutaan panostaa sovelluksen siirrettävyyteen** Pystytään ottamaan huomioon oman organisaation tai asiakkaan/ asiakkaiden vaatimukset ja vastaamaan niihin löytämällä ratkaisun serverless-sovelluksen siirrettävyydestä. Vaatimuksiin vastataan käyttämällä tarvittaessa tiettyä pilvipalveluntarjoajaa (Swail 2020).

Ymmärretään yhteisen standardin puutteen vaikutus siihen, miten sovelluksen serverless-funktiot voivat olla vuorovaikutuksessa keskenään. **Päätavoite: Kun halutaan panostaa sovelluksen siirrettävyyteen** Löydetään keino tai keinoja, joilla voidaan ainakin vähentää tai poistaa serverless-sovellukseen liittyvä lukittuminen tai lukittumisriski. Löydetään keino tai keinoja, joilla serverless-sovelluksen arkkitehtuurista voidaan suunnitella sellainen, että sovellus on tarvittaessa siirrettävissä mahdollisimman vaivattomasti yhden palveluntarjoajan pilviympäristöstä toiseen, erilaiseen pilviympäristöön. Ymmärretään, että riippuu sovelluksesta ja tehdyistä valinnoista, millainen arkkitehtuuri voidaan suunnitella sekä kuinka vaivattomasti siirtämi-nen voi todellisuudessa tapahtua. Halutaan varmistua, että koko serverless-sovellus on siirrettävissä toiseen ympäristöön ja toimii myös tässä toisessa pilviympäristössä. **Päätavoite: Kun halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan** Jos lähestymistavaksi valitaan monipilviympäristö, jossa sovelluksen eri osat ovat jakautuneet useampaan eri ympäristöön, tulee laatia tähän tarkoitukseen soveltuva arkkitehtuuri. Arkkitehtuurin suunnittelussa tulee huomioida sovelluksen komponenttien yhteen toimivuus monipilviympäristössä.	Ymmärretään, että serverless-sovelluksen monimutkaisuus vaikuttaa myös sovelluksen elinkaaren vaiheisiin. Ymmärretään, että kehitys- ja tuotantomalli DevOpsin mukainen automatisointi on myös monimutkaista. (Vega 2019.) **Päätavoite: Kun halutaan panostaa sovelluksen siirrettävyyteen** Löydetään keino tai keinoja, joilla serverless-sovelluksesta voidaan toteuttaa sellainen, että se on tarvittaessa mahdollisimman vaivattomasti ja edullisesti siirrettävissä oleva yhden palveluntarjoajan pilviympäristöstä toiseen, erilaiseen pilviympäristöön. Käytetään tarvittaessa muiden osapuolten tarjoamia ratkaisuja (sovelluksia) korvaaman palveluntarjoajan lukittumisen aiheuttavan valinnan tai sellaisen valinnan, joka kasvattaa huomattavalla tavalla lukittumisriskiä (Vijayan 2018). Löydetään keinoja, joilla sovelluskehittäjän työ nopeutuu, kun suunnitellaan siirrettävissä olevaa sovellusta ja sen toteuttamista. **Päätavoite: Kun halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan** Löydetään keinoja, joilla sovelluskehittäjän työ nopeutuu, kun suunnitellaan monipilviympäristöön olevaa sovellusta ja sen toteuttamista. Löydetään keino, miten voidaan hallita tapahtumapohjaisen serverless-sovelluksen tapahtumia myös monipilviympäristössä (eri pilvien välillä) (Goasguen 2019). Monipilviympäristössä on voitava hallita serverless-sovelluksia sisältävää järjestelmää ja ottaa se sujuvasti käyttöön, vaikka järjestelmä on hajautettu eri ympäristöihin (McLellan 2019).	**Päätavoite: Kun halutaan käyttää monipilviympäristöä eli hyödyntää useaa eri pilviympäristöä samaan aikaan** Voidaan tarvittaessa valita parhaat resurssit ja palvelut eri pilviympäristöistä. Löydetään hyviä keinoja, joilla järjestelmästä on mahdollista saada suorituskykyisempi, joustavampi, turvallisempi sekä sietokykyisempi monipilven avulla (McLellan 2019): • Halutaan esimerkiksi varmistaa paras mahdollinen suorituskyky käyttämällä useamman pilvipalveluntarjoajan ympäristöjä (McLellan 2019). • Sovellus voidaan esimerkiksi käyttöönottaa vähintään kahden eri pilviympäristön mahdollisuuksia hyödyntäen (McLellan 2019; Swail 2020). • Käytetään tarvittaessa sekä yhtä tai useampaa julkisista pilveä, tai rinnalla yksityisistä pilveä. Valinnat tehdään ajatuksella hyödyntää pilvipalvelujen parhaita resursseja. (McLellan 2019.) Pystytään vastaamaan oman organisaation tai asiakkaan/asiakkaiden vaatimuksiin ja ottamaan ne huomioon. Kuten käyttämällä tarvittaessa tiettyä pilvipalveluntarjoajaa (Swail 2020). Tunnetaan tietosuojaa koskevat vaatimukset, myös siltä osin kuin ne liittyvät monipilviympäristöön (McLellan 2019).

4.3 Valittujen tavoitteiden merkitys toteutusvaihtoehtojen kartoittamisessa

Tavoiteryhmien ja niiden sisältämien tavoitteiden kartoituksessa löydettiin useita erilaisia huomioitavia tavoitteita, jotka liittyvät valittuun monipilviympäristöön liittyvään lähestymistapaan eli siirrettävän serverless-sovelluksen suunnitteluun ja toteuttamiseen. Nämä tavoitteet määriteltiin, jotta ne voivat auttaa löytämään vastauksia tutkimuskysymyksiin. Näitä tutkimuksen kannalta merkitykselliset tavoitteet jaettiin tavoitetyhmiin sekä niiden sisältämiin alaryhmiin: Yleiset tavoitteet ja sovelluksen siirrettävyyteen keskittyvä alaryhmä.

Tavoitteet ja toteutusvaihtoehtojen tarkasteluun liittyvät rajaukset

Tutkimuskysymysten kautta tavoitteena oli etsiä vastauksia keskeisiin serverless-sovelluksen siirrettävyyteen ja mukautumiskykyiseen sovelluksen arkkitehtuuriin liittyviin kysymyksiin. Tutkittaessa löydettiin organisaatioille tyypillisiä edelliseen lähestymistapaan liittyviä tavoitteita. Näitä tavoitteita tutkimuksessa käsiteltiin tarkemmin ainoastaan tavoiteryhmien 1 ja 2 osalta (ks. taulukko 1). Tavoiteryhmissä olevista alaryhmistä valittiin toteuttamisvaihtojen kartoitukseen mukaan ne tunnistetut ryhmät, joita voitiin pitää serverless-sovelluksen siirrettävyyteen keskittyvän sovelluksen arkkitehtuurin suunnittelun tai siirrettävyyttä painottavan sovelluksen toteuttamiseen kannalta merkityksellisinä tavoitteina. Monipilviympäristön käyttämiseen liittyvän alaryhmien tavoitteita ei käsitelty tarkemmin tutkimuksessa.

Tavoiteryhmien muodostamisvaiheessa havaittiin, kuinka tavoitteet, tarpeet ja vaatimukset liittyen aiheeseen monipilviympäristö voivat nousta erilaisista lähtökohdista. Tavoitteet, tarpeet ja vaatimukset voivat olla huomattavasti laajemmastakin tai erilaisesta näkökulmasta muodostuvia, kuten erilaisiin pilviympäristöjen palveluihin vaihtoehtoja etsivästä näkökulmasta syntyviä. Vaihtoeh-toja etsittäessä voitaisiin vaikkapa valita tutkittavaksi erityyppisiä pilviympäristöihin liittyviä kiinnostavia palveluita ja selvittää niihin liittyviä vaihtoehtoja niin muista pilviympäristöistä kuin käytettävissä olevista kolmannen osapuolten sovelluksista (jonka tarjoaa joku muu kuin pilvipalveluntarjoaja). Näitä kolmannen osapuolen sovelluksia voisi olla mahdollista käyttää korvaavana ratkai-suna (Vijayan 2018).

Tavoiteryhmän 3 kysymykset ovat tärkeitä huomioitavia silloin, kun harkitaan teknologiaan siirtymistä tai suunnitellaan uutta sovellusta ja nykyisten ja/tai tulevaisuuden tarpeiden pohdinta ja niiden huomioonottaminen sovelluksen toteuttamisessa ovat jo suunnitteluvaiheessa mukana.

Tarpeet voivat olla myös organisaation ja asiakkaiden tarpeita, tai molempien tarpeet yhdistettynä. Tavoiteryhmässä 3 on mainittu erilaisia mahdollisuuksia, jotka liittyvät monipilvistrategiaan ja monipilviympäristöihin, ja myös ne perustelevat edelleen lisää miksi serverless-sovelluksen kyvyllä olla siirrettävä on merkitystä. Eri tavoiteryhmissä mainittuja liiketoimintaan, tietoturvaan ja sovelluksen käyttöönottoprosessiin, monipilviympäristön hallintaan, DevOpsiin ja ylläpitotyöhön liittyviä kysymyksiä ei lyhyitä mainintoja lukuun ottamatta käsitellä.

Valitut tavoitteet ja niiden konkreettinen hyödyntäminen osana tutkimusta

Seuraavaksi tutkimuksessa kartoitettiin toteutusvaihtoehtoja valittujen alaryhmien tavoitteiden antamien lähestymistapojen tukemana. Löydettyjä ja valittuja tavoitteita vastaavien toteutusvaihtoehtojen löytämiseksi ja arvioimiseksi päätettiin tarkastella ensin tarkemmin mistä serverless-teknologiassa on tarkemmin kyse. Tämän jälkeen tutkittiin, miten pilvipalveluntarjoajien erilaiset ympäristöt oikein vaikuttavat serverless-sovelluksien arkkitehtuurin suunnitteluun ja sovellusten toteuttamiseen tutkimalla millaisia toteutusvaihtoehtoja on olemassa eri pilvipalveluntarjoajien ympäristöissä. Tämän jälkeen pyrittiin löytämään ratkaisuja valittuihin tavoitteisiin löydettyjen tietojen pohjalta.

Toteutusvaihtoehtoja pyrittiin löytämään sekä sovelluksen mukautumiskykyisen arkkitehtuurin suunnittelun että siirrettävyyden mahdollistamiseen serverless-teknologiaan liittyvien yksityiskohtien kautta saatavia näkökulmia apuna käyttäen. Vaihtoehtoja sovelluksen suunnitteluun ja toteuttamiseen etsittiin ja tarkasteltiin valittuihin ryhmiin kuuluvien yksittäisten tavoitteiden kautta. Vastauksia haettiin myös kysymyksiin: Millainen sitten on ja voi olla eri pilviympäristöihin tarvittaessa mukautumiskykyinen ja siirrettävissä oleva serverless-teknologiaa käyttävä sovellus sekä tutkia miten tällainen sovellus on mahdollista saada aikaan. Löydettyjen toteutusvaihtoehtojen testaamiseksi tutkimustulosten analysoinnin jälkeen tavoitteena oli suunnitella serverless-teknologiaa käyttävä esimerkkisovellus.

Sovelluksen siirrettävyyttä painottavan teknisen näkökulman lisäksi organisaatioiden on kannattavaa tarkastella aiheen kannalta keskeisiä kysymyksiä muistakin näkökulmista. Kuten liiketoiminnan kannalta hyvien ja käyttökelpoisten valintojen sekä ratkaisujen synnyttämiseksi, että tulevaisuuden huomioon ottamiseksi. Esimerkiksi monipilvistrategiaksi on voitu valita useamman pilvipalveluntarjoajan ympäristön hyödyntäminen, jolloin toteutetun järjestelmän hallinta ja käyttöönotto

voidaan ottaa haltuun turvallisemmin, vikasietoisemmin ja pystyä katastrofienkin sattuessa palautumaan ongelmatilanteista paremmin (McLellan 2019).

5 Erilaisten pilviympäristöjen vaikutus serverless-sovellukseen

Erilaiset pilviympäristöt vaikuttavat siihen millaisia ovat ne yleisimmät mahdollisuudet vähentää tai välttää lukittumisesta aiheutuvia riskejä, jotka liittyvät myös serverless-sovelluksen arkkitehtuurin suunnitteluun. Koska serverless on suosittuna teknologiana kiinteä osa pilvipalveluntarjoajien ympäristöjä ja teknologian käyttämien erityisten serverless-alustojen olemassaolon kautta keskeinen siirrettävyyteen liittyvä kysymys on myös se, millä tavoin siirrettävyyteen vaikuttavat alustan ympärillä olevat erilaiset pilvipalveluntarjoajien muut ratkaisut.

5.1 Serverless ja erilaiset pilviympäristöt

Osa serverless-teknologian ymmärtämistä on tutustua eri pilvipalveluntarjoajien serverless-alustojen toimintaan, eroihin, niiden tarjoamiin mahdollisuuksiin ja mahdollisiin rajoituksiin. Serverless-teknologia on tapahtumapohjainen teknologia (What is serverless? 2022). Tämä tuo mukanaan alustojen toimintaan piirteitä, jotka määrittävät miten serverless-sovelluksen funktioita toteutetaan ja käsitellään eri pilvipalveluntarjoajien ympäristöissä, kun tapahtuu jokin tapahtuma, johon serverless-funktio reagoi.

Valitun tutkimuskysymyksen ohjaamana tavoitteena oli tarkastella serverless-sovelluksen siirrettävyyttä sovelluksen arkkitehtuurin kautta. Koska pilvipalveluntarjoajilla on lukemattomia eri palveluja ja nämä palvelut ovat myös keskenään erilaisia niistä jokaisella voi olla huomattavakin vaikutuksensa arkkitehtuurin suunnitteluun ja mahdollisimman hyvään siirrettävyyteen tähtäävän sovelluksen toteuttamiseen. Tutkittaessa keskityttiin ensin serverless-alustoihin ja serverless-funktioiden toimintaan sekä yleisimpiin palveluihin, joita sovellusten toteuttamisessa käytetään valittujen pilvipalveluntarjoajien ympäristöissä. Tavoitteena oli saada tietoa, miten siirrettävä sovellus voidaan suunnitella ja toteuttaa kun tunnetaan ensin ympäristö, jossa toimitaan.

Kun pilviympäristössä on tarjolla käyttöönotettavaksi standardisoitua teknologiaa kannattaa sellaista käyttää. Esimerkiksi HTTP-protokollalle löytynee tuki kaikilta pilvipalveluntarjoajilta, joten sovelluksen käyttöliittymän osuuden puolesta SPA-sivujen siirto toiseen ympäristöön ei aiheuta

juurikaan kustannuksia, koska muutoksia ei tarvitse tehdä. Käyttöliittymäpuolen koodi on taas jo abstraktoitu HTTP:n toimesta, kun käyttöliittymä kommunikoi backend-puolen kanssa. Jos HTTP:n käyttö ei jostain syystä olisi mahdollista, voidaan ottaa käyttöön oma vaatimuksia vastaava palvelin. Mitään huomattavia kustannuksia ei pitäisi Tanasan (2019) mukaan tulla myöskään API-yhdyskäytävän kannalta sovelluksen siirrossa koskien sekä frontend- että backend-puolia. Esimerkiksi AWS API Gateway -palvelun käyttäminen HTTP-tapahtumien muuntamiseen Lambda-funktioiksi on yleinen tapa toimia. (Tanasa 2019.)

Eri palveluntarjoajien alustojen välisen ohjelmointikielen valinnalla on niin ikään merkitystä sillä myös ohjelmointikielen valinta auttaa estämään lukittumista tiettyyn pilvipalveluntarjoajaan (Tanasa 2019). Huomiota kannattaa kiinnittää serverless-alustojen tukemien eri ohjelmointikielien lisäksi niiden erilaisten versioiden tukeen, koska nämä vaihtelevat eri palveluntarjoajilla. Aivan ensimmäiseksi kannattaa selvittää mitä ohjelmointikieliä harkinnassa olevat pilvipalveluntarjoajat tukevat, sillä jos toisella toimijalla on sama ohjelmointikielen (ja sen version) tuki, siirtokustannukset ovat luonnollisesti pienemmät (Tanasa 2019). Esimerkiksi Amazon AWS tukee nativisti seuraavia ohjelmointikieliä: Java, Go, PowerShell, Node.js, C#, Python ja Ruby (AWS Lambda FAQs n.d.) ja Microsoftin Azure Functions tukee puolestaan: C#, JavaScript, F#, Java, Python, PowerShell ja TypeScript (Microsoft Azure 2021).

Tanasa (2019) korostaa, että lukittumisesta ja korkeista siirtokustannuksista voi päästä eroon, kun ensin erittelee konkreettisesti millaisista pienemmistä osista nämä siirtotyöstä tulevat kokonaiskustannukset voivat koostua, jotta selviää mihin asioihin tulee kiinnittää huomiota. Kun siirtoon liittyvät tarpeet ovat tiedossa, on mahdollista, että voidaan tämän jälkeen standardeja teknologioita käyttämällä saada siirtokustannukset alhaisiksi. Tanasan (2019) mainitsemat standardin mukaiset keinot, kun eivät ole mitään uusia standardeja ja lisäksi ne soveltuvat myös muihin kuin serverless-sovellusten siirtoon toiseen ympäristöön. (Tanasa 2019.)

Pilvipalveluiden palvelujen tarkasteluun liittyvät rajaukset

Koska pilvipalveluiden tarjoajilla on tarjolla lukemattomia erilaisia palveluja, joita voi käyttää myös serverless-sovellusten kanssa, suurin osa niistä jää väistämättä käsittelemättä tutkittaessa. Vaihtoehtojen tarkastelussa päätettiin olla syventymättä esimerkiksi selkeänä erityiskysymyksenäkin nähtävissä oleviin tietokantoihin liittyvään suunnitteluun tai tietokantojen siirrettävyyteen eri

pilvipalveluntarjoajien välillä. Pilviympäristöjen tietokannat voivat olla erityyppisiä tietokantoja. Tietokantoihin liittyen Tanasa (2019) mainitsee, että jos relaatiotietokannat kuuluvat omaan sovellukseen, on SQL-standardi yksi vaihtoehto. Esimerkiksi AWS sisältää relaatiotietokannan nimeltä Amazon Aurora Serverless, jonka käyttämä standardoitu SQL tekee yhdistämisen toisen palveluntarjoajan vastaavaan tietokantaan helpoksi (Tanasa 2019). NoSQL-tietokantojen piirre on se, ettei niillä ole olemassa yhteistä standardia, joten esimerkiksi tietokanta DynamoDB on Amazonin pilvipalvelussa tarjolla oleva NoSQL-tietokanta ja on sellainen kuin sen tehnyt AWS on halunnut siitä tehdä (Al-Tukhi 2022). Esimerkiksi tietokannat DynamoDB ja Azuren Cosmos DB ovat molemmat NoSQL-tietokantoja, mutta keskenään erilaisia: Azuren Cosmos DB tukee SQL-kyselyitä tyypillisen relaatiotietokannan tapaan, mutta DynamoDB ei tue tätä tapaa (Samaranayake 2022).

5.2 Eri pilvipalveluja hyödyntävän sovelluksen suunnittelu ja toteuttaminen

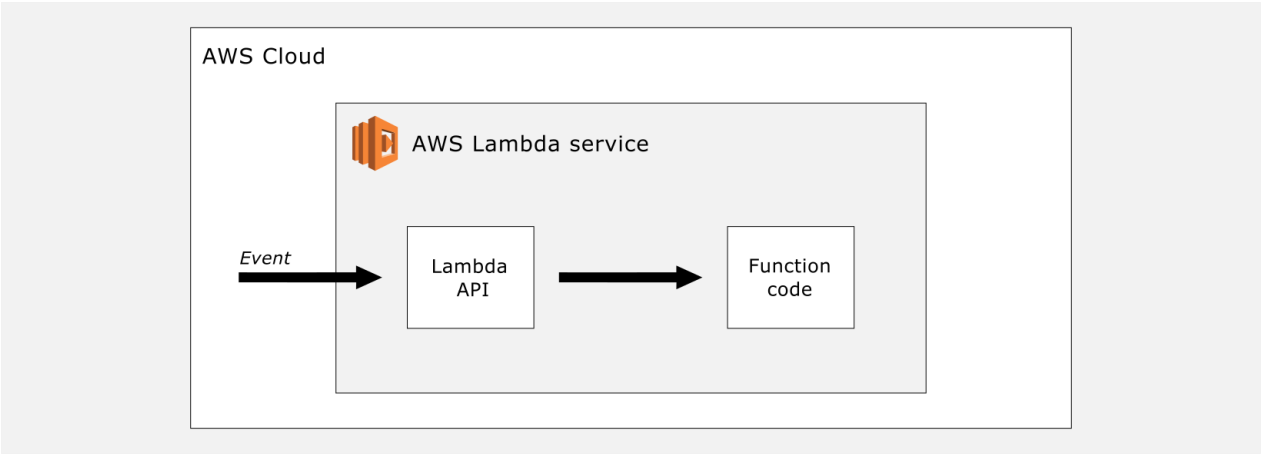
Erilaisia pilvipalvelun tarjoajia on paljon ja niillä on tyypillisesti serverless-funktioiden toteuttamiseen ja ajamiseen liittyvä palvelu. Tutkimukseen valittiin tarkemmin tarkasteltavaksi pilvipalveluntarjoajien Amazon AWS- ja Azure -pilviympäristöt. Yhteistä näille molemmille keskenään erilaisille pilviympäristöille on tarjota alusta serverless-sovelluksen funktioiden ajamiseen ja lisäksi tarvittavat palvelut, kuten palvelut sekä työkalut sovelluksen toteuttamiseen, yhdistämiseen toisiin palveluihin ja julkaisemiseen.

AWS Lambda

Serverless-sovelluksen funktiot ovat AWS-pilviympäristössä Lambda-funktioita. AWS Lambda on alusta, jonka päällä sovelluksen funktiot suoritetaan. Alusta suorittaa funktiota vain tarvittaessa ja ympäristö skaalautuu automaattisesti kysynnän mukaan. AWS Lambda-funktion (serverless-funktion) suorituksen käynnistämiseen on monta tapaa. Funktioita voi kutsua käyttämällä Lambda API -rajapintaa tai Lambda-alusta voi suorittaa funktioita vastauksena erilaisiin tapahtumiin. Funktion suorittamisen voi käynnistää melkein mikä tahansa tapahtuma: Amazon API Gateway -palvelun kautta tulevasta HTTP-request-pyynnöstä ajastettuun toimintoon tai vaikkapa tietokannassa tapahtuvaan muutokseen. (What is AWS Lambda? n.d.)

Lambda-alusta ajaa serverless-funktioita esiintyminä käsitelläkseen tapahtumia (Lambda concepts n.d.). Funktiota voidaan kutsua suoraan Lambda API -rajapinnan (ks. kuvio 4) avulla tai voidaan määrittää AWS-pilviympäristön palvelu tai tietty resurssi kutsumaan funktiota (Lambda concepts

n.d.). Esimerkiksi Amazon API Gateway -palvelun avulla serverless-funktiota voi kutsua HTTPS:n kautta määrittämällä halutun kaltaisen REST API -rajapinnan, jolloin funktio voi vastata REST-kutsuihin (AWS Lambda FAQs n.d.; Poccia 2017, luku 7.1.). Funktio on myös yksi resurssi AWS-ympäristössä, jota voidaan kutsua. Funktion koodi käsittelee tapahtuman, joka on sille toimitettu (Lambda concepts n.d.).



Kuvio 4. Tapahtumapohjaisuus AWS-ympäristössä (Beswick 2021)

Azure Functions

Aivan samoin kuin Lambda-alustallakin, Azuren alustalla serverless-funktio on ajossa vain, kun sitä tarvitaan ja myös ympäristön suorituskyky skaalautuu kysynnän mukaan. Azuren ympäristössä serverless-funktiot sijaitsevat Function App -resurssissa: Function App -resurssi on tavallaan kuin kontti, joka isännöi yhden tai useamman funktion suoritusta (Kurniawan & Lau 2019, 10). Kokoamalla funktiot samaan ryhmään niiden hallinta ja käyttöönottovaihe ovat helpompia samoin kuin funktioiden käytössä olevien resurssien jakaminenkin (Create Function App n.d.). Kaikki serverless-funktiot, jotka luodaan samaan Function App -resurssiin perivät samat konfiguraatiot (Kurniawan & Lau 2019, 10).

Azure Functions -alustalla funktion suorittamisen käynnistäviä triggereitä voi olla vain yksi. Funktion kutsumisen lisäksi tietyt Azuren pilviympäristön triggerit toimivat myös sidoksina (bindings). Yleisiä triggereitä ovat esimerkiksi HTTP-triggeri, joka kutsuu funktiota REST APIn tapaan tai Blob storage, joka käynnistää funktion suorittamisen, kun tiedosto tai kansio luodaan tai sitä muutetaan Blob storagen tallennustilassa. Sidontoja voidaan määrittää useita yhteen triggeriin. Triggeri on vastuussa funktion kutsusta. Sidos (binding) vastaa vaaditun datan toimittamisesta funktiolle.

Sidokset ovat integraatiolla muodostettu yhteys datan ja Azure Functions -alustan funktion välillä. Sidontoja on kahta tyyppiä. Input binding on se data, jonka funktio saa ja output binding on data, jonka funktio lähettää. Sidonta voisi olla esimerkiksi yhteys Cosmos DB -tietokantaan tai tiedostopalveluun. (Likness 2022, 29.)

5.3 Serverless on tapahtumapohjainen teknologia

Yhteisen standardin puute vaikuttaa siihen, miten sovelluksen funktiot voivat olla vuorovaikutuksessa kunkin pilvipalveluntarjoajan palveluihin ja toisiin funktioihin. Serverless-funktiot voivat vastaanottaa dataa ja lähettää dataa. Jokainen pilvipalveluntarjoajan palvelu voi tuottaa erilaista dataa ja voi odottaa, että sille toimitettava data sisältää tietyt asiat sekä on halutussa muodossa. Pilvipalveluntarjoajan API-rajapinnat toimivat yhdistäjänä palveluntarjoajan eri palvelujen, kuten serverless-alustan ja kehittäjän tekemän koodin välillä ja eri palveluntarjoajien serverless-alustat kuitenkin voivat kutsua serverless-funktioita eri tavoin (Vijayan 2018).

Serverless-teknologia perustuu tapahtumapohjaisuuteen eli sovellukset reagoivat niiden ympäristössä ilmeneviin tapahtumiin. Tapahtumapohjaisuus (event-driven architecture), on sovellusarkkitehtuurin malli, jossa pääpaino on reagoinnissa johonkin tapahtumaan tai tapahtumaan liittyvään sovelluksen suorittamaan toimintaan. Tämän kaltainen arkkitehtuuri auttaa löyhästi toisiinsa sidoksissa olevia sovelluksia toimimaan oikealla hetkellä, kun niiden funktioita otetaan käyttöön. (Stigler 2018, 1.)

Serverless-funktioiden suoritus käynnistetään liittyen aina johonkin tapahtumaan. Triggerit ovat käytännössä tapahtumia. Triggerit ovat pilviympäristön palveluja ja HTTP-pyyntöjä, jotka luovat tapahtumia herättääkseen serverless-sovelluksen funktioita ja saadakseen aikaan vastauksen. Triggerit on tyypillisesti määritetty pilvipalveluntarjoajan selainpohjaisessa konsolissa tai CLI-työ-kalulla. Funktioihin liittyvät triggerit ovat tyypillisesti luotu samassa pilvipalveluntarjoajan ympäristössä. Esimerkiksi AWS-pilviympäristössä triggeri voi olla HTTP-pyyntö tai toisen AWS-pilviympäristön palvelun kutsuminen. (Stigler 2018, 20.)

Esimerkkinä AWS Lambda ja funktion käynnistäminen

Lambda-alustalla funktiolla voi olla useita triggereitä ja jokainen näistä triggereistä käyttäytyy client-osapuolen tapaan ja kutsuu itsenäisesti toimivana triggerinä funktiota (Invoking Lambda

functions n.d.). Lambda-alusta välittää tapahtuman (event) funktiolle, ja yksi tapahtuma sisältää tietoja ainoastaan yhdestä client-osapuolesta tai triggeristä (Invoking Lambda functions n.d.). Client on pilviympäristön ulkopuolinen funktion suorittamiseen vaikuttava osapuoli ja client-roolissa toimiva sovellus kutsuu serverless-funktiota eli Lambda-funktiot sijaitsevat tällöin kutsuvan sovelluksen back end -puolella ja niitä voidaan kutsua selaimessa JavaScript-koodin kautta (Poccia 2017, luku 7.1.).

Yleensä funktiot käynnistetään jonkin palvelun kautta, mutta funktioita voi kutsua myös suoraan (Abideen 2022). AWS Lambda-funktioita voi kutsua suoraan käyttämällä Lambda-konsolia, function URL HTTP(S) endpoint -osoitetta, Lambda API -rajapintaa sekä AWS SDK -pakettia, AWS CLI ja AWS toolkits -työkaluja (Invoking Lambda functions n.d.). Kehittäjä voi testata Lambda-funktioita kutsumalla niitä AWS CLI-työkalua tai AWS SDK -pakettia käyttäen (Poccia 2017, luku 7.1.). Function URL on käytännössä HTTP(S) endpoint -osoite, joka voidaan määrittää Lambda-funktiolle (Abideen 2022). Function URL -osoite on yksilöllinen, julkinen sekä muuttumaton ja noudattaa seuraavaa muotoa: https://<url-id>.lambda-url.<alue>.on.aws (Lambda function URLs n.d.). Function URL-osoitetta voi kutsua verkkoselaimen, curl-sovelluksen, Postmanin tai minkä tahansa http-client-roolissa toimivan osapuolen kautta (Lambda function URLs n.d.).

Myös AWS-pilviympäristön muita palveluita voi määrittää kutsumaan funktiota tai voidaan määrittää, että Lambda lukee stream-virrasta tai jonosta ja kutsuu funktiota. Funktiota kutsuttaessa tehdään valinta kutsuntatavasta eli kutsutaanko sitä synkronisesti vai asynkronisesti. Synkronisessa kutsussa odotetaan, että funktio käsittelee tapahtuman ja palauttaa vastauksen. Asynkronisessa kutsussa Lambda asettaa tapahtuman jonoon käsittelyä varten ja palauttaa välittömästi vastauksen kutsujalle. (Invoking Lambda functions n.d.)

5.4 AWS Lambda ja tapahtumien käsittely

Serverless-sovelluksen arkkitehtuuri suunnitellaan tapahtumapohjaiseksi arkkitehtuuriksi, joten serverless-funktioiden tulee pystyä käsittelemään erilaisia tapahtumia sillä alustalla missä niitä suoritetaan. Jotkin AWS-pilviympäristön palvelut voivat luoda tapahtumia, jotka edelleen voivat kutsua serverless-funktiota. Tapahtumapohjaisessa arkkitehtuurissa tapahtuman luovalle palvelulle annetaan lupa kutsua funktiota. Jokin palvelu siis luo tapahtuman, joka edelleen kutsuu serverless-funktiota. (Using AWS Lambda with other services n.d.)

Ohjelmointimallit eroavat AWS Lambda ja Azure Functions -alustoilla. AWS Lambda toimii yksinkertaisia JSON ja YAML -muotoisia määrittelyjä käyttäen funktioiden määrityksessä (Shah 2022). AWS Lambdalla on verrattain suoraviivainen ohjelmointimalli, vaikka yksityiskohdat riippuvatkin valitusta ajoympäristöstä (Shilkov 2019). Tapahtumien välittäminen serverless-sovelluksen funktioille tapahtuu event input -parametreina (Shilkov 2019). AWS-pilviympäristössä tapahtumat ovat JSON-muodossa kuvattuja tapahtumia (Using AWS Lambda with other services n.d.). Lambda-funktio voi vastaanottaa JSON-objektin syötteenä (input) ja voi palauttaa toisen JSON-objektin tulosteena (output) (Shilkov 2019).

Tapahtuman luonut palvelu luo JSON-muodossa olevan kuvauksen ja sen sisällön: Tapahtuman tyyppi määrittää JSON-objektien rakenteen (skeeman) eli tapahtuman JSON-kuvauksen rakenne vaihtelee riippuen siitä, mikä palvelu on luonut tapahtuman ja millaisesta tapahtuman tyypistä on kyse (Shilkov 2019; Using AWS Lambda with other services n.d.). Skeema on määritelty ja dokumentoitu kunkin ohjelmointikielen SDK-pakkauksessa (Shilkov 2019). Kaikissa erilaisten tapahtumien kuvauksissa on siten aina kuitenkin mukana kaikki ne tiedot, joita serverless-funktio tarvitsee käsitelläkseen tapahtuman. Lambda-alusta muuttaa tapahtuman kuvausdokumentin objektimuotoon ja välittää sen serverless-funktion käsittelystä vastaavalle handler-käsittelijäfunktiolle (Using AWS Lambda with other services n.d.).

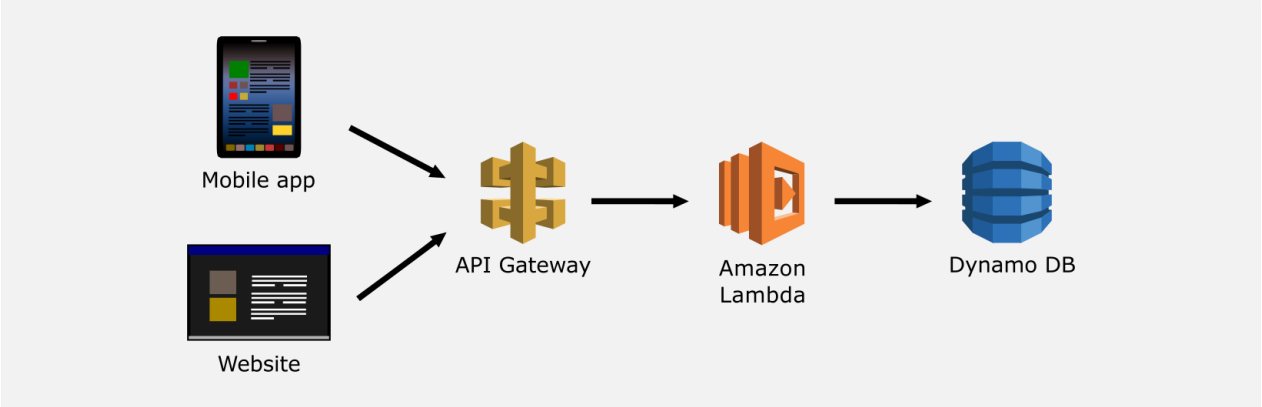
Tapahtuman lähde on jokin AWS-ympäristön palvelu tai kehittäjän luoma oma sovellus, joka synnyttää tapahtuman (AWS Lambda FAQs n.d.). AWS Lambda -alustalla funktiolle välitetään parametreina aina ensin event-tapahtuma ja sitten context (ks. kuvio 5) (AWS Lambda context object in Node.js n.d.). Tapahtuma käynnistää AWS Lambda-alustalla sijaitsevan serverless-sovelluksen funktion suorittaminen (AWS Lambda FAQs n.d.). Funktio saa parametrit suorituksen aikana Lambda-alustalta ja kun Lambda-alusta suorittaa funktiota, context-objekti tarjoaa ominaisuuksia ja toimintoja, jotka antavat tietoa funktion kutsusta, itse funktiosta ja suoritusympäristöstä (AWS Lambda context object in Node.js n.d.). Jotkut AWS-ympäristön palvelut julkaisevat tapahtumat niin että ne kutsuvat suoraan funktiota (kuten Amazon S3 -palvelu) (AWS Lambda FAQs n.d.).

```
module.exports.handler = async function (event, context) {

 console.log('Event: ', event);
 console.log('Context: ', context);

 const response = {
  statusCode: 200,
  body: 'Function executed successfully! Your event: ' + JSON.stringify(event)
 };

 return response;
};
```

Kuvio 5. Esimerkki JavaScriptillä toteutetusta funktiosta AWS Lambda -alustalla

Esimerkki tapahtuman välittämisestä serverless-funktiolle ja funktion vastaus: AWS API Gateway ja Lambda proxy -integraatio

API Gateway on palvelu ja yksi tapa, jota käyttämällä voidaan käynnistää serverless-funktion suoritus ja API Gatewayta käytetään REST, HTTP- ja WebSocket-API-rajapinnoissa (What is Amazon API Gateway? n.d.). API Gateway luo RESTful API- palveluita kun käytetään REST API ja HTTP API -vaihtoehtoja: REST API sisältää laajemman valikoiman eri ominaisuuksia kuin HTTP API, joten valinta tapahtuu tarvittavien ominaisuuksien kautta (Choosing between REST APIs and HTTP APIs n.d.). API Gateway on reitti, jolla pääsee käsiksi esimerkiksi dataan, sovelluksen back end -puoleen ja erilaisiin pilvipalveluntarjoajan palveluihin ja voi hallita niitä (What is Amazon API Gateway? n.d.). Kun Amazon AWS -ympäristön API Gateway käsittelee REST-rajapinnoista tulevia HTTP-pyyntöjä, serverless-alustalla oleva sovelluksen funktio hoitaa tapahtuman käsittelyn halutulla tavalla (Stigler 2018, 2). Kuviossa 6 tapahtuman käsittelyssä on mukana myös pilviympäristössä oleva tietokanta, kun funktiota kutsuu joko mobiilisovellus tai verkkosivuun liittyvä sovellus, ja prosessin aikana tarvitsee käsitellä tietokannassa olevia tietoja.



Kuvio 6. Sovellus kutsuu AWS Lambda -alustalla olevaa funktiota (Stigler 2018, 2)

API Gateway yhdistetään integraatioilla resursseihin, kuten AWS Lambda-alustan serverless-sovelluksen funktioihin (Choosing between REST APIs and HTTP APIs n.d.). Tapahtumaan liittyvän API-metodin integroiminen serverless-funktioon tapahtuu joko Lambda proxy -integraatiota tai mukautettua Lambda non-proxy -integraatiota (Set up Lambda integrations in API Gateway n.d.). Proxy-integraatio on yksinkertaisempi tapa, ja mukautetussa tavassa määritetään lisäksi, kuinka saapuva request-pyyntöjen data yhdistetään integrointipyyntöön sekä määritetään, kuinka integraa-tion tuloksena syntyvän vastauksen sisältämä data liitetään API-metodin vastaukseen (Set up Lambda integrations in API Gateway n.d.).

5.5 Azure Functions ja tapahtumien käsittely

Serverless-funktion toteuttaminen Azure Functions -alustalla eroaa Lambda-ympäristöstä. Funktiot käsittelevät erilaisia tapahtumia tälläkin alustalla ja niitä kutsutaan erilaisista palveluista tai sovelluksen muista komponenteista. Azuren pilviympäristössä on käytössä monimutkaisempi tapa toimia verrattuna AWS Lambdaan. Kuten AWS-pilviympäristössäkin mahdollisia tapahtuman synnyttäviä lähteitä on paljon. Azure Functions -alustalla on käytössä triggerien lisäksi sidonnat, ja nämä yhdessä vaikuttavat siihen mitä funktio saa syötteenä ja tuottaa tulosteena (Shah 2022).

Azure Functions -alustalla funktio sisältää kaksi osaa: Konfiguraatiotiedoston nimeltä function.json ja funktion koodin. Käytetystä ohjelmointikielestä riippuu, luodaanko konfiguraatiotiedosto automaattisesti vai tuleeko se luoda itse. Tiedosto function.json määrittää mikä on funktion triggeri, sidokset ja muut konfiguraatiot. Jokaisella funktiolla voi olla ainoastaan yksi triggeri. Ajoympäristö käyttää tätä konfiguraatiotiedostoa määrittääkseen tapahtumat, joita sen tulee valvoa ja siihen, että se tietää kuinka välittää data funktiolle ja kuinka palauttaa data funktion suorituksen jälkeen. Bindings-ominaisuus tarkoittaa konfiguraatiotiedostossa kohtaa, jossa määritetään sekä triggerit että sidokset. Erilaisilla sidoksilla on kaikilla muutama yhteinen asetus (type, direction ja name) sekä joitain sidontatyyppikohtaisia asetuksia. Asetus name on identifioiva nimi, jota käytetään funktioon sidotulle tiedostolle. JavaScript-kielessä nimi olisi avain/arvo -listan avain. Kun käytetään JavaScriptiä Azure Functions -alustan Node.js-ajoympäristö etsii index.js-tiedostoa, joka on samassa hakemistossa kuin tiedosto function.json. Funktio saa useita parametreja, joista ensimmäinen on aina context-objekti. (Azure Functions JavaScript developer guide 2022.)

AWS-pilviympäristön tapaan event- ja context -objektit ovat käytössä myös Azure Functions -alustalla. Ajoympäristö käyttää context-objektia välittämään tietoa funktioon ja funktiolta funktion suorituksen aikana ja context-objekti sisältää erilaisia tarpeellisia ominaisuuksia (Azure Functions JavaScript developer guide 2022). Azuren pilviympäristössä serverless-funktiolle välitetään parametrina ensin context ja sitten event (ks. kuvio 7). Event-parametrin nimi voi olla mitä tahansa kuten myTimer, req tai vaikkapa databaseEvent.

Eri pilvipalveluntarjoajilla on valikoimissaan usein samankaltaisia toimintoja mahdollistavia palveluja. Esimerkkinä saman toiminallisuuden mahdollistava triggeri ja AWS-pilviympäristön API Gateway triggerin vastine Azuren pilviympäristössä on triggeri nimeltä HTTP Trigger. HTTP ja webhook -triggerit ja HTTP output -sidonnat käyttävät request ja response -objekteja esittämään HTTP-viestintää (Azure Functions JavaScript developer guide 2022). Kun työskennellään HTTP triggereiden kanssa, HTTP request ja response -objekteihin on mahdollista päästä käsiksi usealla eri tavalla (Azure Functions JavaScript developer guide 2022).

```
module.exports.handler = async function (context, event) {

 context.log('Context: ', context);
 context.log('Event: ', event);

 context.res = {
  status: 200,
  body: 'Function executed successfully! Your event: ' + JSON.stringify(event)
 };

 context.done();
}
```

Kuvio 7. Esimerkki funktiosta Azure Functions -alustalla, joka käyttää HTTP triggeriä

6 Siirrettävän sovelluksen toteuttaminen: Tarjolla eri vaihtoehtoja

Seuraavaksi tutkimuksessa etsittiin vaihtoehtoja, miten serverless-sovelluksesta voidaan toteuttaa sellainen, että se on tarvittaessa mahdollisimman vaivattomasti ja edullisesti siirrettävissä oleva yhden palveluntarjoajan pilviympäristöstä toiseen, erilaiseen pilviympäristöön. Lopullisen vaihtoehdon valinnassa on ratkaisevaa itse sovellus ja olemassa oleva tarve, kaikki vaihtoehdot eivät sovellu kaikkiin tarpeisiin. Sovelluksen arkkitehtuurin suunnittelun ja toteuttamisvaihtoehtojen osalta kysymyksiä tarkasteltiin siirrettävyyttä painottavan sovelluksen näkökulmasta.

6.1 Ohjelmistokehys apuna mukautumiskykyisen sovelluksen kehitystyössä

Myös serverless-sovellusten toteuttamisessa voi käyttää eri tavoin apuna erilaisia työkaluja, kuten ohjelmistokehyksiä. Eri serverless-kehyksiin liittyy samaan aikaan monia riskejäkin tai haasteita ja niiden välillä tehtävä valinta riippuu kehysten erilaisista ominaisuuksista. Lisäksi serverless-kehityksessä voidaan käyttää apuna erilaisia ympäristöjä sekä voidaan käyttää sovelluksen suunnittelussa sellaista ohjelmistoarkkitehtuuria, jolla on erityinen rakenne, joka tukee sovelluksen mukautuvuutta monipilviympäristöjä ajatellen. Serverless Framework on esimerkki työkalusta, joka tukee monipilviympäristöjä. Lukittumiseen on ohjelmistokehysten muodossa ratkaisuja, kuten Serverless Framework: Ohjelmistokehys antaa keinot tehdä alustojen välille yhteensopivia sovelluksia (Swail 2020; Vijayan 2018). Cross-cloud-sovellukset voivat kuitenkin olla vuotavia, minkä vuoksi Swailin (2020) mukaan on useimmissa tapauksissa järkevämpää käyttää kunkin alustan tarjoamia natiiveja mahdollisuuksia kuten työkaluja ja rajapintoja (Swail 2020).

Vuotava abstraktio on sellainen abstraktio, joka vuotaa yksityiskohtia, joista tulisi olla päästy abstraktion avulla eroon. Vuotaminen on huono asia sen vuoksi, koska se voi aiheuttaa ohjelmiston toiminnassa virheitä. Mikäli kehittäjä luottaa abstraktioon, että se toimii aina virheettömästi, voi aiheutua yllättäviä ongelmia. Abstraktion taustalla vaikuttava monimutkaisuus voi aiheuttaa toisinaan ongelmia, kun tämä monimutkaisuus ”vuotaakin” yllättäen abstraktointia käyttävää välinettä käyttävään sovellukseen. Abstraktiot ovat tarpeellisia ja auttavat kehittäjää työssä, mutta silti olisi hyvä tuntea mitä niiden taustalla on. Mitä korkeammalla tasolla abstraktio on, sitä vaikeampi on selvittää mitä asioita abstraktio yrittää piilottaa taustalta. (Padmanabhan 2022.)

Ohjelmistokehysten avulla tuotetaan joko pienimuotoinen Serverless-alusta, joka toimii olemassa olevien pilvipalvelujen päällä tai yksinkertaistaa suurten Serverless-alustojen asettavia vaatimuksia (Kritikos & Skrzypek 2018, 161-162). Esimerkiksi AWS Lambda on laaja Serverless-alusta, jonka käyttäjät voivat käyttää serverless-ohjelmistokehyksiä (Kritikos & Skrzypek 2018, 161-162). Ohjelmistokehys Serverless Framework suunniteltiin työvälineeksi tuottamaan sovelluksesta julkaisuvaiheessa esimerkiksi Amazon AWS -pilvipalveluntarjoajan ympäristön käyttöön soveltuvia AWS Lambda -funktioita, tapahtumia ja infrastruktuurin resursseja turvallisella ja nopealla tavalla (AWS – Deploying n.d.). Serverless Framework -ohjelmistokehys tukee monia AWS Lambda ja Azure Functions -alustojen event-tapahtumia, jotka voivat käynnistää serverless-funktion suorittamisen

(AWS Lambda Events n.d.; Azure – Events n.d.). Käyttöönottovaiheeseen tällä ohjelmistokehyksellä on eri tapoja erilaisia käyttöönottotilanteita varten (AWS – Deploying n.d.).

6.2 Vaihtoehtoisen ratkaisun etsiminen pilvipalveluntarjoajan palvelun tilalle

Jos lukittumiseen halutaan etsiä vastausta vaihtoehtoisen ratkaisun kautta ja halutaan säilyttää mahdollisuus siirrettävyyteen, voidaan etsiä vastaavia ratkaisuja pilvipalveluntarjoajien palveluille. Esimerkkinä vaihtoehtona Amazon AWS -pilviympäristön BaaS-palvelulle nimeltä Kinesis voidaan käyttää Kafkaa: Kinesis ja Kafka ovat sovelluksina tarkoitettu reaaliaikaisten datavirtojen keräämisen ja analysointiin (Levy 2022, Yadav 2019). Kun vaihtoehtona on pilvialustan hyödyt antava Kinesis ja lukittumisen ”pelossa” valittu itse asennettava ja käyttöönotettava mahdollisuus, kuten Kafka, lukittumisen riski vähenee ja silti oman sovellustuotteen saa valmiiksi nopeammin valmista ratkaisua käyttämällä (Tanasa 2019). Vaikka vaihtoehtoinen ratkaisun käyttö, kuten edellä mainitun Kafkan, voi tuoda alhaisemman infrastruktuuriin liittyvät kustannukset, käyttöönotto vaatii enemmän suunniteltua, aikaa ja jatkuvaa ylläpitoa, koska palvelua ei hankita valmiina (Levy 2022). Eli ratkaisun seurauksena kuluu resursseja asentamiseen, itse sovellukseen perehtymiseen, ympäristön pystyttämiseen mahdollisimman tehokkaalla tavalla ja lopulta tämän käyttöönottoon (Tanasa 2019).

Mikäli vastaava ratkaisu on käytettävissä myös toisessa pilvipalvelussa, tämä voi olla yksi huomionarvoinen valintaperuste lisää, kun esimerkiksi ratkaisu on jo ennalta tuttu. Edellä mainittua Kafkaa on mahdollista myös käyttää Azuren ympäristössä (Choose a stream processing technology in Azure 2022). Esimerkiksi AWS-pilviympäristön Kinesiksen valitseminen ja sen runsas käyttö puolestaan tarkoittaisi samalla valitusta pilvipalvelusta saatavan hyödyn maksimointia, jolloin pienimmän yhteisen nimittäjän lähestymistavan vaikutustakaan ei synny (Tanasa 2019). Kafkan valitseminen taas merkitsisi vapautta, mutta vaatisi resursseja vievän asennusprosessin, aikaa itse Kafkaan perehtymiseen eli riittävää osaamista, jotta siitä saisi eniten irti ja aikaa ympäristön pystyttämiseen mahdollisimman tehokkaalla tavalla sekä sen työn, että varmistutaan ratkaisun skaalautumisen toimivuus tarpeen mukaan pilvessä (Tanasa 2019).

Pilvipalveluntarjoajat haluavat selvästi sitouttaa asiakkaitaan käyttämään heidän hallinnassaan olevia valmiita palveluja. Pilvipalveluntarjoajien palveluiden valikoimat muuttuvat kaiken aikaa ja esimerkkinä AWS tarjoaa nykyisin mahdollisuuden käyttää Kafkaa joko itse hallittuna Kafka

klusterina tai sen hallinnassa olevana palveluna nimeltä Amazon Managed Streaming for Apache Kafka (Amazon MSK) (Using Lambda with self-managed Apache Kafka n.d.). Koska vaihtoehtoiselle ratkaisulle Kafkalle on AWS-pilviympäristössä tarjolla myös valmis palvelu itse ylläpidetyn palvelun tilalle, palvelua tarvitsevan käyttäjän ei tarvitse enää käyttää resursseja palvelun käytöstä seuraaviin hallintatehtäviin, mikäli niin ei halua edelleen toimia (Jeyakumaran 2022; Using Lambda with self-managed Apache Kafka n.d.).

OpenFaas-ohjelmistokehyksen ja Kubernetes-ympäristön avulla eroon lukittumisesta

Pilvipalveluntarjoajien serverless-alustoillekin on olemassa open-source-vaihtoehtoja. Suosittu OpenFaaS on ohjelmistokehys, jota käyttäen voi luoda serverless-sovelluksia käyttämällä kontteja ja Kubernetes-ympäristöä. Houkuttelevaksi valinnaksi OpenFaas-ohjelmistokehyksen voi tehdä se, että sitä käytettäessä pilvipalveluntarjoajan erilaiset palvelut ovat edelleen käytettävissä. Eli aivan samalla tavoin kuten silloinkin jos käytettäisiin pilvipalveluntarjoajan omia serverless-alustoja. (Hein 2018; Using OpenFaaS on AKS 2022.)

Azuressa Azure Kubernetes Service (AKS) on palvelu, jonka kautta voi käyttää OpenFaaS-ohjelmistokehystä funktioiden luomiseen (Using OpenFaaS on AKS 2022) ja myös AWS-pilviympäristössä on mahdollista käyttää OpenFaaS-vaihtoehtoa käyttämällä Amazon Elastic Kubernetes Service (EKS) -ympäristöä (Hein 2018). Wilde (2019) korostaa, että OpenFaas onkin yksi keino päästä eroon lukittumisen huolista: Kubernetes-ympäristön avulla on OpenFaaS-ohjelmistokehystä mahdollista käyttää myös yhdessä Lambda-alustan kanssa, ja kutsua Lambda-alustaa käyttäviä funktioita, mikäli halutaan edelleen käyttää pilvipalveluntarjoajankin alustaa. Wilde (2019) mainitsee edelliseen toimintaan perustelluksi syyksi sen, että tällä tavoin on mahdollistaa siirtää olemassa olevat Lambda-alustan päällä toimivat funktiot käyttämään OpenFaaS-ohjelmistokehystä ja silti rinnalla edelleen on mahdollista käyttää AWS Lambda-alustaa, kunnes ollaan täysin valmiina siirtämään funktiot kokonaan Kubernetes-ympäristöön. Lisäksi AWS-ympäristön arkkitehtuurissa on joitain integraatioita, joita ei voi käyttää ilman Lambda-alustaa (Wilde 2019).

6.3 Serverless-sovelluksen arkkitehtuurin suunnittelu

Seuraavaksi tutkimuksessa etsittiin vastauksia siihen, miten mukautumiskykyisen arkkitehtuurin suunnittelu voidaan ratkaista. Tavoitteena oli löytää keino tai keinoja, joilla voidaan ainakin vähentää lukittumisriskiä tai jopa kokonaan poistaa serverless-sovellukseen liittyvä lukittuminen tai sen

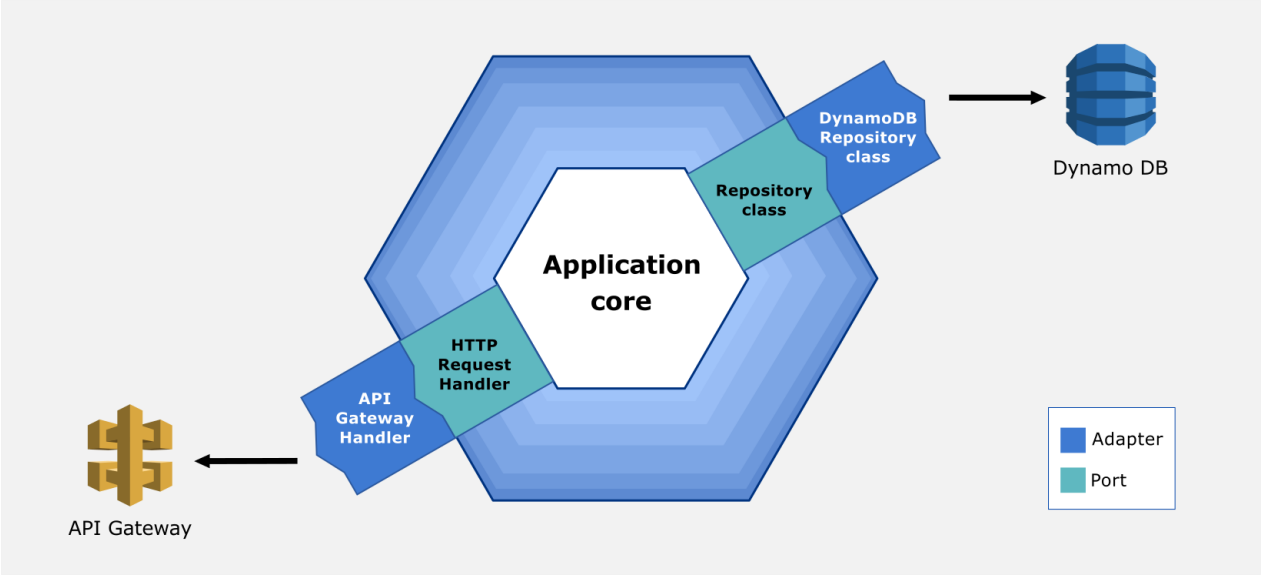
riski. Tavoitteena oli pystyä suunnittelemaan sellainen serverless-sovelluksen arkkitehtuuri, että sovellus on tarvittaessa siirrettävissä mahdollisimman vaivattomasti yhden palveluntarjoajan pilviympäristöstä toiseen, erilaiseen pilviympäristöön.

Siirrettävissä olevan sovelluksen arkkitehtuurin merkityksestä

Serverless-teknologiaa käyttävien sovellusten arkkitehtuurin suunnittelun keinoin voi vaikuttaa lukittumiseen, joka on voitu myös nähdä suurenakin ongelmana. Siirtymisen helpottamisessa tarkastellaan tavallisesti ensimmäisenä abstraktiota. Abstraktion voi ajatella olevan käyttöjärjestelmä, joka erottaa sovelluksen taustalla olevasta fyysisestä laitteistosta. Abstraktio, jota tarvitaan Serverless-sovelluksen siirtämisessä toiseen pilvipalveluun tarkoittaa, että käytössä on toimiva sovelluksen arkkitehtuurimalli. Tanasa (2019) painottaa, että serverless-sovelluksen arkkitehtuuri kannattaa suunnitella huolella, eikä sännätä suin päin toteutusvaiheeseen, jotta tulevaisuudessa tapahtuvat sovellusten siirrot toisiin ympäristöihin olisivat helpompia. Sovelluksen mukautuvuus ja siirrettävyys monipilviympäristöissä tarkoittaa, että sovelluksen tulee siis voida irrottautua pilvipalveluntarjoajan ympäristöstä. Abstraktio, joka mahdollistaa siirrettävyyden saadaan käyttämällä soveltuvaa arkkitehtuurimallia. Hyvin laadittu ja toimiva sovelluksen arkkitehtuurimalli kuitenkin luo sellaisen abstraktion Serverless-sovellukseen, että sen uudelleen käyttäminen on mahdollista toisessa pilvialustassa. (Tanasa 2019.)

Hexagonal-arkkitehtuuri

Sovellus koostuu tässä arkkitehtuurimallissa adapteri-, portti- ja ydinosio -roolien komponenteista (ks. kuvio 8). Hexagonal-arkkitehtuurissa on merkittävää se, että sovelluksen ydin ei riipu pilvipalvelun tarjoajan ympäristön ekosysteemistä (Tanasa 2019). Sovellusalue eristää ajossa olevat sovellukset toisistaan, jotta ne eivät vaikuta toisiinsa (What is Application Domain 2020). Tanasa (2019) mainitseekin, että sovellusalue eristyy sovelluksen ulkopuolisista ympäristöistä hyvin käyttämällä arkkitehtuurimallina Hexagonal-arkkitehtuuria. Kuviossa 8 esitetään arkkitehtuurimallin toimintaidea AWS Lambda -funktioita käyttävässä sovelluksessa. Siirtokustannukset pienenevät myös silloin, kun sovelluksen ydinosion koodia ei tarvitse muuttaa, vaikka sovellus siirretään toiseen ympäristöön ja tässä arkkitehtuurimallissa tarvitsee vain luoda uudet sovittimet ja kytkeä ne sovellukseen (Tanasa 2019).



Kuvio 8. Hexagonal-arkkitehtuuria käyttävä sovellus AWS Lambda -alustalla (Tanasa 2019)

Hexagonal-arkkitehtuurimallia käyttävän sovelluksen testaaminen ja siirtokustannukset

Yksikkötestit, jotka ovat kytkettyjä tiettyyn pilvipalveluntarjoajaan ovat kuitenkin haastavia yhdistää toisiin ympäristöihin, ja tämän vuoksi siirron toteuttaminen on kallista. Kuitenkin hyvää arkkitehtuurimallia käyttämällä tästä riskipaikasta voi kuitenkin päästä eroon. Testien tulisi kommunikoida täysin erillään sovelluksen taustalogiikasta, kun ne kommunikoivat esimerkiksi Amazon AWS -pilviympäristön omien palveluiden kanssa. Mitä enemmän sovelluksen komponentit ovat kiinni pilvipalveluntarjoajan ympäristöön sitä tärkeämmän roolin saavat integraatiotestit. Integraatiotestien koodin on kommunikoitava pilvipalvelun valmiiden palvelujen kanssa. (Tanasa 2019.)

Hexagonal-arkkitehtuurissa on selkeä piirre, joka helpottaa yksikkötestausta, jolloin ei ole tarvetta simuloida AWS SDK-kutsujakaan mocking-menetelmällä (Linden 2020). Koska Hexagonal-arkkitehtuuria noudattavan sovellus ei ole riippuvainen ulkoisista tekijöistä, myös sovelluksen riippuvuuksien mocking-tavalla tehty testaaminen on helppoa (Martinez 2021). Mock-testauksessa luodaan sisäisestä tai ulkoisesta sovelluksen käyttämästä palvelusta sitä simuloiva versio, joka käyttäytyy kuin alkuperäinen (Linden 2020). Tämä tapahtuu simuloimalla sovelluksen adapterien toimintaa. Linden (2020) näkee tässä arkkitehtuurimallissa paljon etuja, se tekee ohjelmakoodin helpommaksi tuottaa ja lukea sekä sillä on monia etuja, kuten ylläpidettävyys ja jatkokehittämismahdollisuudet, mutta alussa tarvitaan luonnollisesti panostuksia.

6.4 Muut keinot, jotka helpottavat siirrettävän sovelluksen toteutusvaihetta

Erilaisia hyödyllisiä työkaluja voi käyttää myös rajatummassa käyttötarkoituksessa, jolloin ei välttämättä synny mitenkään merkittävää riippuvuutta mihinkään tiettyyn työkaluun. Serverless-sovellusten käyttöönottoon on kehitetty useita erilaisia työkaluja. Serverless-arkkitehtuuria käyttävän sovelluksen käyttöönottovaiheeseen on myös olemassa erilaisia työkaluja, joilla kehitystyö nopeutuu kuten Serverless Framework, Apex ja Claudia.js (Tanasa 2019). Näissä työkaluissa, joissa kuvataan sovellusta ja pilviympäristön infrastruktuuria on hyvä tiedostaa, että ne eivät ole automaattisesti samalla tasolla ohjelmakoodin kanssa, joten se mitä sovelluskoodissa lukee ei päivity automaattisesti työkaluihin.

Suosittuja sovellusten käyttöönottovaiheeseen liittyviä työkaluja ovat esimerkiksi Terraform, jo aiemmin käsitelty Serverless Framework sekä AWS SAM. Käsitteellä Infrastructure As Code (IaC) viitataan työkaluihin, joilla voi koodin kautta hallita pilvipalvelun tarjoajan infrastruktuuria. IAC-työkaluilla infrastruktuuri kuvataan tiedostossa tietyllä syntaksilla, ja ideana on, että työkalun avulla on mahdollista luoda, päivittä ja hallita pilven infrastruktuuria julkaisuvaiheessa. Teigerin (2022) mukaan Terraform on tähän tehtävään hyvä työkalu, mutta varjopuolena serverless-teknologian kaikki hyvät puolet eivät kuitenkaan enää ole samoilla tavoin käytössä kuin jos jättäisi infrastruktuuriin liittyvät tehtävät pilvipalveluntarjoajan kannettavaksi. Terraformia käytettäessä täytyy ottaa omalle hallintavastuulleen kaikki resurssit itse, eikä pilvipalveluntarjoaja enää tee tätä ohjelmistokehittäjän puolesta automaattisesti. Teiger (2022) mainitsee, että Terraform on suunniteltu perinteisempiin infrastruktuureihin kuin serverless, joissa on ollut vähemmän erilaisia komponentteja ja vuorovaikutusta niiden välillä. Joten suuren konfiguraatioiden määrän hallinta on haastavaa ja altistaa virheille. (Teiger 2022.)

Serverless Framework -ohjelmistokehystä voi käyttää myös käyttöönottotyökaluna. YAML-syntaksia käyttävässä tiedostossa kuvataan pilviympäristön infrastruktuuri ja työkalu mahdollistaa julkaisuun usean eri pilvipalveluntarjoajan ympäristöön. Serverless Framework -työkaluun on olemassa yli 1000 lisäosaa, mutta Teiger (2022) huomauttaa että niiden suhteen kannattaa olla varma niihin liittyvistä mahdollisista tietoturvaongelmista, mikäli käyttää lisäosia. Työkalun heikkoutena on myös usein epäselvä dokumentaatio. (Teiger 2022.)

AWS Serverless Application Model (SAM) on avoimen lähdekoodin ohjelmistokehys serverless-sovellusten luomiseen AWS-pilviympäristöön. YAML-syntaksia käyttäen kuvataan funktiot, API-rajapinnat, tietokannat ja tapahtuman lähteet. SAM CLI sisältää Lambda-alustan kaltaisen ympäristön, jossa voi lokaalisti rakentaa ja testata sovelluksia. SAM CLI -työkalun avulla voi myös julkaista (deploy) sovelluksen AWS-pilviympäristöön ja luoda CI/CD-putkia. (Teiger 2022.)

Myös serverless-ympäristöihin liittyvää standardisointia on haluttu kehittää ja tämä standardisointiin liittyvä työ jatkaa kaiken aikaa kehittymistään. Niin pilviympäristöissäkin kuin serverless-teknologian kanssa voidaan käyttää jo aiemmin mainittua tietokantoihin liittyvän perinteistä SQL-standardia. Nimenomaan pilvikehitykseen ja tapahtumapohjaiseen serverless-teknologiaan liittyvää datan siirtoa koskevan standardin suunnitelmallinen kehitystyö alkoi vaikuttaa kehittämisen arvoiselta. Tapahtumien käsittelyyn liittyvät valinnat ovat olennainen osa serverless-sovellusten suunnittelua (Stigler 2018, 1) ja toteuttamista. Datan siirtoon toiseen ympäristöön vaikuttavaan standardisointityöhön liittyy edelleen käynnissä oleva määritelmän kehitystyö nimeltään CloudEvents-standardi. Tämän standardin kehittämistyön käynnisti CNCF Serverless Working Group (CloudEvents. A specification for describing event data in a common way 2022; Tanasa 2019).

CloudEvents on määritelmä, joka pyrkii yksinkertaistamaan sovellusten käytön aikana syntyvien tapahtumien yhteydessä liikkuvan datan (event data) määritystä ja toimittamista esimerkiksi eri palvelujen ja alustojen välillä. Vaikka edellä kuvattu määrityksiin liittyvä työ on vasta uutta ja määritystyö kehittyy kaiken aikaa, sitä kohtaan on osoitettu selkeää kiinnostusta suurista pilvipalveluntarjoajista alkaen. Nykyisin CloudEvents-määritysten kehitystyötä jatkaa Cloud Native Computing Foundation. (CloudEvents. A specification for describing event data in a common way. 2022.)

CloudEvents-standardia voi soveltaa esimerkiksi seuraavalla tavalla: Kehittäjä voi hyödyntää CloudEvents-standardia ohjelmakoodissa siinä vaiheessa, kun oman sovelluksen back end -puolen koodissa valmistaudutaan lähettämään itse luodulle serverless-funktiolle event-tapahtumaan liittyviä tietoja (ks. kuvio 9). Koodissa on tällöin funktio, jolle riittää, että se ottaa vastaan kaksi parametria, jotka kertovat mikä on tapahtuman sisältö ja tapahtuman nimi (Alvarez 2022.)

```
export class ScheduledEvent {

 private constructor(private readonly _cloudEvent: CloudEvent<unknown>) {}

 public static create(message: unknown, eventName: string): ScheduledEvent {
   // Add your Validations in here

   return new ScheduledEvent(
     new CloudEvent({
       specversion: "1.0.1",
       id: `uuid`,
       type: eventName,
       source: "my-system",
       time: new Date().toISOString(),
       datacontenttype: "application/json",
       data: message,
     })
   );
 }

 // getters
}
```

Kuvio 9. Esimerkki CloudEvents-standardia käyttävästä tapahtuman kuvaamisesta (Alvarez 2022)

CloudEvents-määrityksessä data sisältää hyötykuorman. Kuvion 8 ohjelmakoodin kohdassa datacontenttype kuvataan sen datan tyyppi, joka välitetään. Tieto id on tapahtuman uniikki tunniste (joka kuviossa 8 generoidaan uuid-sovelluksella). Source kuvaa tapahtuman lähteen. Specversion-kentässä kerrotaan mitä CloudEvents-määrityksen versiota käytetään. Type kertoo event-datan tyypin (mikä tapahtuma). Time kertoo tapahtuman luontihetken. (CloudEvents - JSON event format 2022.)

7 Tutkimustulokset ja johtopäätökset

Vaikka pilvipalveluntarjoajat pyrkivätkin selvästi houkuttelemaan asiakkaita sitoutumaan heidän ympäristöönsä, rinnalta on löydettävissä suorastaan vastavoima, joka on pyrkinyt aktiivisesti kehittämään ja on kehittänyt erilaisia vaihtoehtoja, joiden vaikutuksesta sitoutuminen eli lukittuminen jo vähenee tai mahdollisesti jopa poistuu. Palveluntarjoajien välinen kilpailutilanne on myös saanut aikaan, että suosittujen pilviympäristöjen, kuten esimerkiksi AWS ja Azure tarjoavat samankaltaisia palveluja.

7.1 Arkkitehtuurin mukautumiskyvyn ja sovelluksen siirrettävyyden merkitys

Ensimmäisessä tutkimuskysymyksessä kysyttiin miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys toiseen pilviympäristöön on merkittävä asia. Serverless-

funktiot ajetaan pilvipalveluntarjoajien omissa serverless-alustoissa, joiden hallinta ja kehittäminen tapahtuu käytännössä loppujen lopuksi täysin palveluntarjoajan lähtökohdista. Vielä tänäkin päivänä lukittuminen tiettyyn palveluntarjoajaan nousee esiin riskinä, kun kiinnostutaan serverless-teknologiasta. Tämä loogista, koska serverless on jo teknologiana voimakkaasti ja kiinteästi sidottu pilvipalveluntarjoajan infrastruktuuriin ja pilvipalveluntarjoajan serverless-alustan lisäksi pilviympäristön palveluihin. Monia palveluita on käytännössä myös pakko käyttää, jotta serverless-sovellus kykenisi suorittamaan mitään sellaisia toimintoja, että sovellus kannattaisi tällä ylipäätänsä teknologialla toteuttaa. Lukittumisriskien vähentämiseen tai poistamiseen on eri keinoja, joista arkkitehtuurin mukautumiskykyyn ja siirrettävyyden panostaminen on yksi vaihtoehto.

Miksi sitten siirrettävyys on merkittävä asia? Sovellusten kehittäminen on hidasta ja maksaa. Se voi olla kallista ja moniin eri tavoin vaikeaa. Ei todennäköisesti voida täysin varmasti myöskään tietää haluttaisiinko jossain vaiheessa organisaation omana päätöksenä esimerkiksi vaihtaa pilvipalveluntarjoajaa tai voisiko käydä niin, että jokin uusi asiakas edellyttäisi, että käytössä onkin jokin toinen pilvipalveluntarjoaja. Tarve muutokseen voi olla myös pienempi ja kohdistua johonkin yksittäiseen toisen pilviympäristön tarjoamaan vastaavaan kiinnostavaan palveluun. Jonkinasteinen kompromissin teko sitoutumisen ja vapauden liikkua esimerkiksi ympäristöstä toiseen on väistämätöntä, ja muutoksiin on silti sopeuduttava koska teknisissä kysymyksissä kehitys on jatkuvaa ja myös vaikkapa asiakkaiden omaan liiketoimintaansa liittyvät odotukset ja vaatimukset muuttuvat.

Yhteisten standardien puute näkyy siinä, että erilaisia toteutusvaihtoehtoja ja ympäristöissä tarjolla olevia palveluita on paljon, mutta niistä jokainen toimii omalla tavallaan. Pilvipalveluntarjoajien tavoitteena on tavallaan enemmänkin sitoa asiakas omaan ympäristöönsä, kuin rohkaista toimimaan toisin. Tästä toimii esimerkkinä vaihtoehtojen kartoituksen yhteydessä esitelty ja käyttäjien keskuudessa suosiota saavuttanut Kafka, jota AWS tarjoaa nykyisin myös itse hallinnoimanaan valmiina palveluna (Jeyakumaran 2022; Using Lambda with self-managed Apache Kafka n.d.). Kaikki tämä vaikuttaa huomattavasti serverless-sovellusten suunnitteluun ja toteuttamiseen. Jokainen erilainen työkalu, palvelu tai ympäristö tulee myös varta vasten opetella. Mitä syvemmin jokainen voimakkaasti erilainen ratkaisu on kiinteä osa sovellusta, voidaan helposti päätellä, että tästä seuraa suuriakin haasteita siinä vaiheessa, kun sovellusta on pakko muuttaa tai laajentaa. Sovellus ei yksinkertaisesti ole helposti muokattavissa käytettäväksi erilaisen valinnan rinnalla. Pilvinatiivista sovelluksesta on kuitenkin mahdollista tehdä siirrettävä.

Mikäli serverless-sovelluksen arkkitehtuuri on suunniteltu ja toteutettu siirrettävyyttä painottaen muutoksiin sopeutuminen käy helpommin, nopeammin ja vaatii vähemmän resursseja. Sovelluksen kyvyn kyetä muuntautumaan tarvittaessa voi hyvin nähdä olevan selkeä kilpailuetukin, koska mahdollisuudet reagoida esimerkiksi nopeammin eri tilanteissa kasvavat ja muutostyö voi nopeuden lisäksi olla sen kautta johtaa laadukkaampaan lopputulokseen, jos muutosvaihe vie vähemmän resursseja ja vähäisempien resurssien tarve voi tuoda mukanaan pienemmän kustannukset.

Koska moni asia vaikuttaa lukittumiseen, voi sovelluksen arkkitehtuurilla ja siirrettävyyteen panostamisella olla iso rooli vähentää lukittumiseen liittyvän riskin pienentämisessä ja jopa poistamisessa kokonaan. Jokaisen sovelluksen kohdalla on kuitenkin hyvä harkita huolella missä määrin ja miten lukittuminen on juuri tässä tapauksessa merkittävä kysymys ja halutanko panostaa siirrettävyyteen ja millaiset resurssit tähän tehtävään on käytettävissä? Käytännössä kompromisseja joutuu jollain tasolla aina tekemään. Missä kohdassa hyväksytään sitoutuminen ja missä ei on keskeinen kysymys. Näitä kysymyksiä on kannattavinta tarkastella osana organisaation laajempaa strategiaa, eikä vain monipilvistrategian tai teknisten kysymysten kautta. Valintoja on joka tapauksessa tehtävä ja jokainen valinta on, kuten Vijayan (2018) totesi, aina myös jonkinlainen potentiaalinen toteutumaton riski.

Serverless-teknologian tapahtumapohjaisuus, serverless-alustat ja ympäristöt, joissa sovelluksia ajetaan sekä vaihtoehdot FaaS- ja BaaS tuovat omat vaikutuksensa arkkitehtuurin mukautumiskyvyn ja sovelluksen siirrettävyyden merkityksiin. Tapahtumat liittyvät käytännössä johonkin toisen tai kolmannen osapuolen ratkaisuun tai toiseen sovelluksen serverless-funktioon. Tapahtumiin liittyy tietoa, joka tulee voida välittää kaikkialle missä sitä tarvitaan. Arkkitehtuuri tulee siten suunnitella niin, että se huomioi tapahtumapohjaisen sovelluksen toimintatavat ja on samalla sellainen, että sen avulla toteutettu sovellus voi mukautua erilaisiin ympäristöihin ja ratkaisuihin, joita siihen liitetään.

7.2 Siirrettävän sovelluksen suunnitteluun ja toteuttamiseen liittyvät haasteet

Seuraavassa tutkimuskysymyksessä kysyttiin mitä yleisiä haasteita liittyy toiseen pilviympäristöön mahdollisimman vaivattomasti siirrettävissä olevan serverless-sovelluksen suunnitteluun ja toteuttamiseen. Koska serverless-sovellukset ovat tapahtumapohjaisia, keskeistä on tapahtumien hallintaan ja käsittelyyn liittyvät keinot, kun sovelluksesta tehdään monipilviympäristöön tarvittaessa

mukautuva ja siirrettävä. Pilvipalveluntarjoajien omien serverless-alustojen erilaisuus ja se, miten ne käsittelevät ja muodostavat tapahtumia erilaisten pilviympäristön palvelujen API-rajapinnoissa kuitenkin vaihtelee. Serverless-funktioiden tulee silti kyetä vastaanottamaan tarvittavat tiedot ja myös toimittamaan tietoa eteenpäin. Tapahtumalla on väistämättä tietty rakenne ja sisältö, joka halutaan välittää eteenpäin funktiolle, ja nämä tiedot ovat elintärkeitä serverless-funktiolle, jotta se kykenee suorittamaan tehtävänsä. Yhteisen standardin puute vaikuttaa väistämättä siihen, miten sovelluksen funktiot voivat olla vuorovaikutuksessa kunkin pilvipalveluntarjoajan palveluihin ja toisiin funktioihin. Tapahtumat on pystyttävä myös välittämään sovelluksen komponentilta toiselle komponentille ja komponenttien on pystyttävä kommunikoimaan keskenään.

Sellaista standardia ei ole olemassa, joka kirjaimellisesti pakottaisi pilvipalveluntarjoajat toimimaan yhteisesti sovitulla tavalla. Tämän vuoksi kehittäjät joutuvat panostamaan erilaisten palvelujen ja pilvipalveluntarjoajien ympäristöjen opetteluun. Pilvipalveluntarjoajat lisäävät palvelujensa joukkoon jatkuvasti uusia palveluita ja myös kehittävät edelleen sekä muuttavat olemassa oleviakin palveluja. Serverless-sovellusten ajoympäristöt ovat myös ohjelmointikielikohtaisia ja ohjelmointikieliinkin liittyy eri versioita. Jatkuva ja nopea muutos koskee aivan samalla tavoin pilviympäristöjäkin kuin alaa muutenkin.

Koska pilvipalveluntarjoajien laajojen ja monimutkaisten ympäristöjen kehittäminen nykyiselle tasolle on ollut vuosikausien työ, ei varmaankaan ole kovin odotettavaa, että nämä massiiviset ympäristöt olisivat koskaan jonkin saman standardin tai standardien mukaisella tavalla ainakaan laajassa mittakaavassa eli suuremmissa määrin ohjelmistokehittäjän näkökulmasta samalla tavoin toimivia. Realistista onkin odottaa, ja toivoa, kuten esimerkiksi CloudEvents-standardin kehittämiseen liittyvissä ponnisteluissa on ollut tavoitteena, että standardin kautta syntyisi yhteinen tapa toimia liittyen johonkin pieneen, mutta huolella valittuun ja merkitykselliseen yksityiskohtaan. Yksityiskohtaan, joka olisi serverless-teknologiaa käyttävien sovellusten ja ohjelmistokehittäjienkin kannalta kuitenkin sellainen asia, että pieneltäkin vaikuttavalla tavoitteella voitaisiin saavuttaa selkeä hyöty.

Osana haasteita ovat myös organisaatioissa olevat tarpeet ja vaatimukset. Tarpeisiin kuuluu esimerkiksi serverless-teknologiaan ja pilvipalveluntarjoajien ympäristöihin sekä muihin valintoihin liittyvä riittävä ymmärrys ja osaaminen, organisaation valmiudet sekä myös lähestymistapa, jonka

kautta halutaan tarkastella monipilviympäristön hyödyntämistä. Lähestymistavat erosivat toisistaan siinä, että toisessa haluttiin panostaa siirrettävyyteen ja toisessa hyödyntää samaan aikaan useaa eri pilviympäristöä. Valittu lähestymistapa vaikuttaa suunnittelun ja toteutuksen haasteisiin, mutta molemmissa on myös yhteisiä piirteitä. Omien tarpeiden ja vaatimusten tunnistaminen sekä valintojen tekeminen ylipäätänsä voi olla haasteellista. Valinnat vaikuttavat myös koko ohjelmistokehitysprosessiin.

7.3 Valittujen tavoitteiden täyttyminen tutkimustuloksissa

Monipilviympäristön hyödyntämiseen liittyviä organisaatioilla olevia tavoitteita havaittiin olevan olemassa useita erilaisia. Tutkimuskysymyksiä käytettiin ohjaamaan näiden tavoitteiden tunnistamisesta tutkimuksen aikana sekä myös tarkasteluvaiheessa auttavana lähtökohtana silloin, kun kartoitettiin ensin tarpeita ja vaatimuksia sekä silloin, kun näistä muodostettiin tavoitteita. Nämä jaettiin vielä tavoiteryhmiin. Tavoiteryhmistä valittiin tutkimuskysymyksiä vastaavat tavoiteryhmät ja tavoiteryhmien sisältämien kahteen alaryhmään jaetut tavoitteet ohjaamaan toteutusvaihtoehtojen kartoittamista. Näitä tavoitteita käytettiin tukemaan tutkimuksen sitä osuutta, jossa tarkasteltiin ensin erilaisten pilviympäristöjen vaikutusta serverless-sovellukseen ja tämän jälkeen edettiin toteutusvaihtoehtojen kartoitusvaiheeseen. Tarkemman käsittelyn ulkopuolelle rajattiin ne tavoiteryhmät ja niiden sisältämät tavoitteet, jotka liittyvät samaan aikaan tapahtuvan eri pilviympäristöjen hyödyntämiseen ja edellyttävät että suunnitteluun vaikuttaa jokin organisaatio ja sen asiakas/asiakkaat.

Tavoiteryhmässä 1 haluttiin löytää vastauksia siihen, miten hallita lukittumiseen liittyvät kysymykset, jotka vaikuttavat sovelluksen arkkitehtuurin suunnitteluun. Tavoiteryhmän 1 yleisiin tavoitteisiin annettiin vastaus, kun kuvattiin tapahtumapohjaisen sovelluksen idea sekä kerrottiin, miten tapahtumapohjainen serverless-sovellus toimii valittujen pilvipalveluntarjoajien (AWS ja Azure) ympäristöissä ja serverless-alustoilla. Saavutetun ymmärryksen kautta oli mahdollista muodostaa käsitys, miten teknologiavalinta käytännössä vaikuttaa lukittumiseen ja lukittumisriskeihin. Samalla selkeni, miten standardin puute näkyy tilanteissa, joissa sovelluksen funktiot ovat vuorovaikutuksessa palveluihin ja keskenään.

Toteutusvaihtoehtojen kartoitusvaiheessa oli mahdollista muodostaa vaatimusten avulla ensimmäiset päätökset millaisia asioita tutkimukseen kuuluvassa esimerkkisovelluksessa olisi

huomioitava sekä löydettiin arkkitehtuurimalli, joka mahdollistaa serverless-sovelluksen arkkitehtuurin suunnittelemisen niin että sovelluksesta voi toteuttaa pilviagnostisen sovelluksen ja alustavat esimerkkisovellukseen liittyvät valinnatkin olivat mukana arkkitehtuurimallin valinnassa. Ensimmäinen päätökset olivat, että esimerkkisovelluksessa tultaisiin käyttämään sekä FaaS- että BaaS-tapoja serverless-funktioissa. Ympäristöihin tutustuttaessa opittiin, että pilvipalveluntarjoajien palveluissa olisi väistämättä useita palveluita, jotka vaikuttavat lukittumiseen, koska serverless-teknologiaa käyttävän sovelluksen toteuttaminen niin ettei se käyttäisi juuri mitään pilviympäristön palveluja ja samalla kuitenkin sisältäisi hyödyllisiä toimintoja ei käytännössä ole mahdollista.

Tavoiteryhmässä 2 haluttiin löytää keinoja siihen, miten voidaan toteuttaa omien tarpeiden kannalta hyvä serverless-teknologiaa käyttävä kokonaisratkaisu ja saavuttaa laajempaa ymmärrystä teknologian valinnan vaikutuksista sekä itse teknologiasta myös sovelluksen elinkaaren eri vaiheissa. Tavoiteryhmän 2 yleisiin tavoitteisiin annettiin vastauksia ratkaisuvaihtoehtojen esittelyn yhteydessä sekä edellä toteutusvaihtoehtojen analysoinnin yhteydessä.

Tavoiteryhmien 1 ja 2 päätavoitteeseen vastattiin esittämällä useita erilaisia ratkaisuvaihtoehtoja, joiden lopulliseen valintaan vaikuttaa millaista sovellusta ollaan toteuttamassa ja erilaisista valinnoista, mihin sitten päädytäänkään. Ei ole olemassa ratkaisua, jota voisi automaattisesti suositella. Lopulliset valinnat ovat tilanne- ja sovelluskohtaisia. Eri vaihtoehtoihin perehtyminen kannattaa kuitenkin tehdä, tutkia ja kokeilla niitä, sekä laatia vaikkapa demo ja testata siirrettävyyttä valittua arkkitehtuurimallia käyttäen. Osa löydetyistä ratkaisuvaihtoehdoista liittyvät ohjelmiston elinkaaren eri vaiheisiin, kuten käyttöönottovaiheessa hyödylliset IaC-työkalut ja niitä apuna käyttäen sovelluksen kehitystyökin voi helpottua. Omien tavoitteiden määrittelyssä ja niiden soveltuvuuden arvioinnissa on kannattavaa ja tarpeellista ottaa myös oman organisaation pilviympäristössä toimimiseen liittyvät valmiudet huomioon. Lisäksi arvioinnissa on tarpeellista pohtia, miten tapauskohtaisesti voidaan ja kannattaa hyödyntää sovelluksen siirrettävyyden ja/tai moneen eri pilviympäristön samanaikaiseen hyödyntämiseen liittyviä mahdollisuuksia.

Käytetty tavoiteryhmäjaottelu, niiden sisältämät tavoitteet ja tehdyt rajaukset osoittautuivat toimiviksi ja vastasivat tutkimuksen tavoitteita. Tutkimuksen yhtenä tavoitteena oli löytää vastaukset tutkimuskysymyksiin niin, että tutkimuksen tuloksia on mahdollista myös soveltaa erilaisiin serverless-teknologiaa käyttäviin ohjelmistoprojekteihin ja laatia niiden avulla esimerkkiratkaisu, joka

antaa konkreettisen vastauksen sekä ratkaisuehdotuksen osana tutkimusta. Tavoitteiden kautta toteutusvaihtoehtojen kartoitusvaihe helpottui ja tavoitteiden määrittäminen auttoi varmistamaan, että keskitytään tutkimusongelman kannalta keskeisiin toteutusvaihtoehtoihin, joilla on myös kyky antaa tarvittavat vastaukset.

7.4 Millainen siirrettävä serverless-sovellus ja sen arkkitehtuuri voi olla

Viimeisessä tutkimuskysymyksessä haluttiin löytää siirrettävissä olevan sovelluksen arkkitehtuurin suunnitteluun liittyvä vastaus. Siirrettävyyttä painottavaksi ja tutkimuskysymyksen ratkaisevaksi arkkitehtuurimalliksi löydettiin Hexagonal-arkkitehtuuri, joka soveltuu hyvin tapahtumapohjaiseen sovellukseen eikä ole sidottu mihinkään tiettyyn ohjelmointikieleen ja soveltuu moniin erilaisiin tarkoituksiin, ja myös serverless-teknologialla toteutettujen sovellusten arkkitehtuurien suunnitteluun. Sovellukselle laaditaan muuttumaton ydin ja sen ympärille tarvittaessa mukautumiskykyiset sovelluksen komponentit, jotka mahdollistavat siirrettävyyden. Myös hexagonal-arkkitehtuurin peruspiirre, jossa sovelluksen osat ovat eristettyinä toisistaan liittyy siirrettävyyteen eli vähentää lukittumisen riskiä. Sovelluksen suunnittelu arkkitehtuurimallia käyttäen ohjaa toteuttamaan siirrettävän sovelluksen mutta vaatii myös huolellisen suunnittelu- ja toteutusprosessin ja riittävää osaamista, että sovelluksen siirrettävyys säilyy toteutusvaiheessakin. Hyvääkään arkkitehtuurimallia käyttämällä ei voi ratkaista kaikkia lukittumisriskejä, koska lukittumisriskeihin vaikuttavat monet asiat.

Arkkitehtuurissa ja sovelluksen toteuttamisessa ei ole vielä riittävä saavutus se, että serverless-funktioita voidaan ajaa kunkin pilvipalveluntarjoajan serverless-alustalla ja kutsua näistä funktioista toisia itse tehtyjä funktioita. Edelläkin kuvatussa tapauksessa käytetään jo pilviympäristön infrastruktuuria ja sen palveluja, joista serverless-alusta on yksi. Toinen kriittinen osa tutkimuskysymyksen ratkaisevassa arkkitehtuurissa on se, miten erilaisten pilviympäristöissä olevien palvelujen käyttäminen onnistuu niin, että siirrettävyys on mahdollista. Serverless-teknologiaa käytettäessä halutaan tyypillisesti hyödyntää hyviä palveluita, joita pilvipalveluntarjoajilla on tarjolla, eikä esimerkiksi itse suunnitella ja toteuttaa vastaavia toiminnallisuuksia alusta alkaen.

Sovelluksen arkkitehtuurin suunnittelun tulee lähteä siten serverless-sovelluksen tapahtumapohjaisesta tavasta toimia. Arkkitehtuuria käyttämällä voidaan muodostaa sovellukselle pilviagnostinen kerros, jonka tehtävä on kyetä käsittelemään kaikista valituista pilviympäristöistä tulevat

serverless-funktioiden kutsut ja laatia niitä kutsuville funktioille tai palveluille sellaiset vastaukset, joita kohdeympäristö ymmärtää. Eri palvelujen erilaisia API-rajapintoja varten laaditaan yleiskäyttöisiä funktioita. Hexagonal-arkkitehtuurin porttien ja adapterien avulla edellä kuvatut tarpeet on mahdollista toteuttaa. Tuki uusien palveluiden lisäämiseksi osaksi sovellusta tai muutosten tekeminen olemassa oleviin palveluihin liittyviin ohjelmakoodeihin ei ole ongelma, koska sovelluksen komponentit ovat itsenäisiä ja sovelluksen ydin on eristetty sovelluksen niistä osista, jotka mahdollistavat pilviagnostisuuden.

Esitelty Hexagonal-arkkitehtuuri ei sido sovelluksen arkkitehtuurin suunnittelua mihinkään tiettyyn pilviympäristöön tai työkaluun. Mikäli arkkitehtuuri suunnitellaan huolella ja sovellus toteutetaan hyvin, sovellus tulee sisältämään sellaisia yleiskäyttöisiä sovelluskomponentteja, jotka mahdollistavat sovelluksessa niiden käyttämisen useiden muiden eri komponenttien toimesta. Samojen palveluiden API-rajapintoihin liittyvissä komponenteissa olevia toimintoja ei tällöin tarvitse toteuttaa yhä uudelleen, kun sovellukseen tulee uusia ominaisuuksia.

7.5 Toteutusvaihtoehtojen vaikutus valintojen tekemiseen

Kolmannessa tutkimuskysymyksessä asetettiin tavoitteeksi painottaa siirrettävyyttä toiseen pilviympäristöön ja tarkastella tätä tavoitetta sovelluksen arkkitehtuurin suunnittelun näkökulmasta. Valitun ratkaisun ja siirrettävyyden kannalta valittiin vielä edellistä tavoitetta tarkentava lähestymistapa, kun mahdollisimman vaivattomasti tapahtuvan siirrettävyyden tavoitetta tarkennettiin. Tässä tarkennetussa näkökulmassa sellainen vaihtoehto olisi paras, joka tuo mukanaan mahdollisimman vähän rajoituksia sovelluksen suunnitteluun ja toteuttamiseen. Eli paras valinta olisi sen kaltainen vaihtoehto, joka ei ainakaan kovin helposti synnyttäisi riippuvuutta tiettyyn palveluun tai tuotteeseen ja niitä tarjoavaan osapuoleen.

Tutkimuksen kattavuudesta ja luotettavuudesta

Koska pilvipalveluiden tarjoajilla on valikoimissaan lukemattomia erilaisia palveluja, tutkimuksen aikana nousi ensin esiin kysymys tehdyn kartoituksen kattavuudesta ja luotettavuudesta, koska useimmat näistä palveluista jäivät täysin käsittelyn ulkopuolelle. Kun aiheeseen liittyvää tutkimusta oli tehty enemmän ja edetty tutkimustulosten analysointivaiheeseen, havaittiin ettei tutkimuskysymyksiin liittyvien vastausten löytämiseksi näin olisikaan tarpeen tehdä: Pohjimmiltaan

kyse on perusongelman ymmärtämisestä ja perusongelman ratkaisemiseksi soveltuvien toteutusvaihtoehtojen löytämisestä.

Ratkaisuvaihtoehtojen tilannekohtainen arviointi ja soveltaminen tulee todellisen sovelluksen tai järjestelmän suunnittelussa ja toteuttamisessa väistämättä eteen. Luodaanhan jokainen niistä johonkin tiettyyn tarpeeseen ja vaihtoehtoja selvästi riittää. Valmistunut sovellus tai järjestelmä on siten omanlaisensa eri lähtökohdista tehtyjen valintojen kautta muodostunut kokonaisuus, joka koostuu pilvipalveluntarjoajien palveluista, jotka ovat kiinteä osa sen toimintaa. Joten voitiin päätyä lopputulokseen, että perusongelman tunnistaminen, ymmärtäminen ja soveltuvien vaihtoehtojen löytäminen on riittävä vastaus tutkimusongelmaan.

Tutkimusongelman kannalta ratkaisevan tarkastelukohdan tunnistaminen

Pilviympäristöistä valittiin tarkasteluun kaksi keskenään erilaista pilvipalveluntarjoajan ympäristöä. Nämä tarkasteltavat ympäristöt, niiden palvelut sekä serverless-alustat ja tavat käsitellä tapahtumapohjaisen sovelluksen kannalta tapahtumien käsittelyä erosivat selkeästi toisistaan. Kun tilanne oli edellä kuvattu, voitiin muodostaa päätelmä, että nämä kaksi valittua pilvipalveluntarjoajaa sekä niiden serverless-alustat antoivat sen kaikkein ratkaisevimman tarkastelukohdat tutkimusongelmaan liittyvän ratkaisun löytämiselle. Lukittumiseen ja siirrettävyyteen liittyvä keskeinen ongelmakohta oli tunnistettu. Samalla voitiin päätellä, että edellä mainitut lähtökohdat huomioimalla voidaan riittävän luotettavasti tutkia, onko ylipäätänsä mahdollista luoda siirrettävissä oleva sovellus ja lopulta esittää ratkaisuksi ongelmaan arkkitehtuuria, jonka tarkoituksena on ratkaista siirrettävyysongelma. Erilaiset serverless-alustat sellaisenaan ovat jo selkeä lukittumisen aiheuttava ominaisuus ja lisäksi tapahtumapohjaisessa sovelluksessa tapahtumien välittäminen ja niiden käsittelymahdollisuudet ohjelmakoodissa ovat aivan keskeisessä roolissa. Joten tarvittiin keino tai keinoja, joka poistaisi tämän ongelman. Ratkaisuksi ongelmaan löydettiin hexagonal-arkkitehtuuri.

Toteutusvaihtoehdot ja valittu arkkitehtuurimalli

Mikäli valitaan avuksi jokin kolmannen osapuolen työkalu, kuten ohjelmistokehys ja halutaan välttää lukittuminen, ollaan käytännössä tekemässä samalla valintaa, jonka seurauksena lukittaudutaan käyttämään tätä palvelua. Kun käsiteltiin lukittumiseen liittyvä riskikysymyksiä, Vijayan (2018) muistutti siitä, että valitsi sitten minkä tahansa teknologian, tehty valinta on väistämättä samalla

myös mahdollinen riski: Riski, joka ei vain ole vielä toteutunut ja riskikysymyksen suhteen tilanne on sama, kun tehdään valinta palvelun toimittajasta.

Tutkimuskysymyksissä määritettiin, että keskeiset kysymykset ovat serverless-sovelluksen siirrettävyys ja se, miten sovelluksen arkkitehtuurin suunnittelun keinoin voidaan tähän kysymykseen vaikuttaa niin, että sovellus olisi mahdollisimman vaivattomasti siirrettävissä toiseen pilviympäristöön. Tutkittaessa havaittiin, että siirrettävyyteen voi vaikuttaa monellakin eri tavalla. Koska tavoitteeksi asetettiin nimenomaan mahdollisimman vaivattomasti tapahtuva siirrettävyys sovellusarkkitehtuurin kautta, ne vaihtoehdot, jotka eivät liity sovelluksen arkkitehtuuriin tai jollain tapaa toisivat lisää lukittumisen riskejä, päätettiin jättää esimerkkisovellukseen liittyvästä suunnittelussa ja toteutusvaiheessa käyttämättä. Myös ne vaihtoehdot, jotka olisivat voineet olla tavoitteessa avuksi, mutta joiden käyttämättä jättäminen ei myöskään estänyt suunnittelemasta ja toteuttamasta tutkimuksen tavoitteen mukaista esimerkkiratkaisua, päätettiin jättää käyttämättä.

Mahdollisimman vaivattomasti tapahtuva siirrettävyys tarkentui siis tutkimuksen edetessä tavoitteeksi, jossa pyritään minimoimaan sovelluksen lukittuminen laajemminkin, kuin vain pilvipalveluntarjoajan ympäristöön. Jos osaksi arkkitehtuuria ja sovelluksen toimintaa otettaisiin mukaan vaikkapa OpenFaas ja Kubernetes-ympäristö tulisi eteen uusia haasteita ja huomioitavia asioita, joita ei ole välttämätöntä kohdata koska sovelluksen arkkitehtuurista on mahdollista suunnitella ja toteuttaa hexagonal-arkkitehtuuria käyttäen pilviagnostinen sovellus. Mikäli arkkitehtuurin suunnittelu tehdään hyvin ja siihen panostetaan sekä pyritään toteuttamaan yleiskäyttöisiä komponentteja, voidaan saavuttaa yleisesti erilaiseen pilviympäristöön mukautumiskykyisenä ja siirrettävänä pidettävä arkkitehtuuri ja sovellus. Näin toimittaessa ei mieleen pitäisi tulla toteutetun työn merkitystä kyseenalaistava mielikuva. Eli sellaista mielikuvaa, jossa tehty ratkaisu synnyttäisi ajatuksen siitä, että nyt on lähdetty keksimään turhaan ratkaisuja kysymyksiin, jotka joku muu on jo valmiiksi aiemmin ratkaissut (ja kenties aikaan saatua toteutusta paremminkin).

Ei ole mitään takeita, että kehitystyö on riittävän aktiivista jonkin kolmannen osapuolen työkalun, kuten esimerkiksi Serverless Framework -ohjelmistokehyksen tekijöillä. Kun työkalun tukemat useiden pilvipalveluntarjoajien palvelut muuttuvat, ylläpidetäänkö myös Serverless Framework -ohjelmistokehystä ajan tasalla samaan tahtiin? Ei voida olla myöskään varmoja, katoaako vaikkapa jokin ohjelmistokehys kokonaan markkinoilta jossain vaiheessa. Ohjelmistokehysten käyttämiseen

liittyy etua tuovien mahdollisuuksien rinnalla se riski, että ohjelmistokehyksiin kuuluu lisäksi aina mahdollisuus toimimattomuudenkin riskiin, koska taustalla olevat ympäristöt ovat monimutkaisia ja voivat aiheuttaa toisinaan ongelmia, joihin kolmannen osapuolen työkalua käyttämällä ei voida itse mitenkään vaikuttaa (Padmanabhan 2022). Abstraktiokerroksen (jonka ohjelmistokehykset tuottavat) toteuttaminen ei ole myöskään yksinkertaista ja kun jokin palvelu pyrkii yksinkertaistamaan voimakkaasti alla piilevää laajaa ja monimutkaista pilviympäristöä, millaisia tänä päivänä pilvipalveluntarjoajien ympäristöt selvästi käsitettävissä ovat satoine eri mahdollisuuksineen, riskien toteutumisen mahdollisuuden voi selvästi nähdä olevan olemassa. Työkalujen avulla voidaan kyllä saavuttaa nopeutta kehitystyöhön ja säästää kustannuksissa, joten kaikkiin tilanteisiin sopivaa ”oikeaa” tai ”parasta” vastausta millaisia valintoja kannattaa tehdä ei valmiiksi vastaukseksi kysymykseen ole mahdollista määrittää.

Valitun työkalun käyttö tulee myös opetella, ja silti kehitystyössä vaaditaan valitun pilviympäristön tai pilviympäristöjen tuntemusta. Hyödyllistä on myös, jos ohjelmointikieli, jolla työkalu (mikäli sen lähdekoodi on avoimesti tarkasteltavissa) on tehty olisi tuttu kehittäjälle, jolloin lähdekoodin avoimuus ja osaaminen mahdollistaa työkalun toimintaan tutustumisen. Kehitystyöhön ja tehtyihin valintoihin vaikuttaa myös millainen sovellus on kyseessä ja kuinka paljon halutaan ja voidaan loppujen lopuksi panostaa siirrettävyyteen. On mahdollista, että halutaankin käyttää esimerkiksi jotain tiettyä pilviympäristön palvelua, jota ei ole muilla pilvipalveluntarjoajilla tai vastaavaan valintaan on jokin muu painava peruste, vaikka valinta silloin vaikeuttaisi sovelluksen siirrettävyyttä.

Mikäli halutaan varmistaa, että koko serverless-teknologiaa käyttävä sovellus on siirrettävissä toiseen pilviympäristöön, tulee tutkia omien toiveiden ja tarpeiden kautta riittävän kattavalla tasolla ja riittävän yksityiskohtaisesti useita pilviympäristöjä sekä niiden palveluita ja erilaisia toteutusvaihtoehtoja. Tämä tehtävä vaatii myös riittävän hyvää tarkastelun kohteeksi valittujen pilvitarjoajien ympäristöjen tuntemusta, jotta luotettavana pidettävissä olevaa arviointia voi ylipäätänsä tehdä. Tutkimusvaihe antaa myös samalla mahdollisuuden kasvattaa omaa tai organisaation osaamista, laatia demosovelluksia ja prosessin aikana opitut asiat auttavat lopullisten päätöksien teossa, sovelluksen arkkitehtuurin suunnittelussa sekä sovelluksen toteutusvaiheessa.

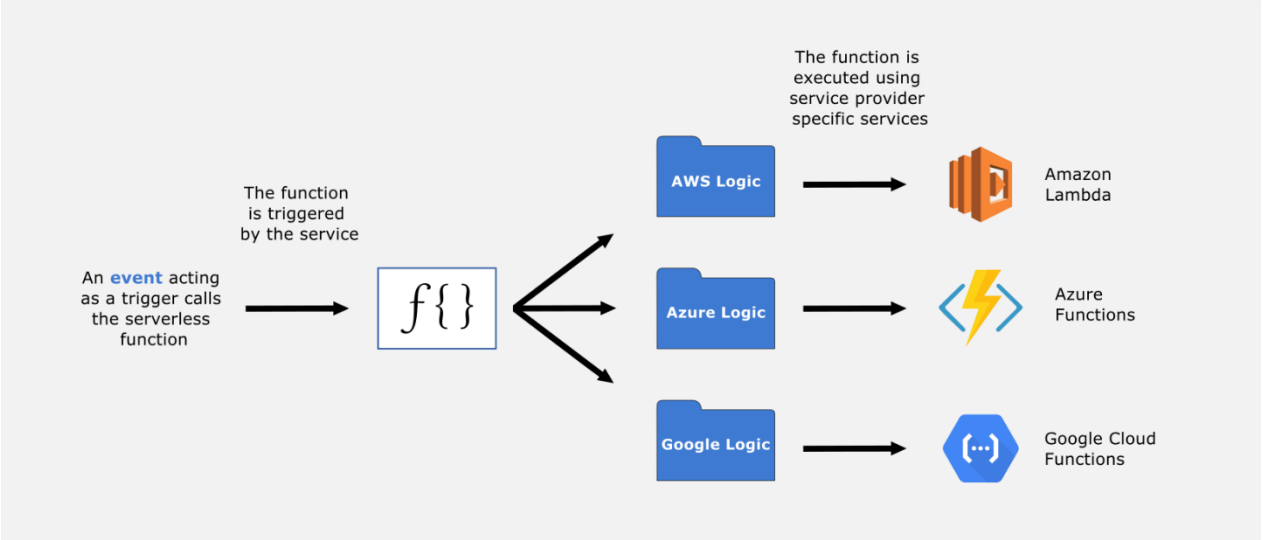
8 Esimerkkisovelluksen suunnittelu ja toteutus

Tutkimustulosten analysoinnin ja tehtyjen johtopäätöksien perusteella esimerkkisovelluksen arkkitehtuurimalliksi valittiin hexagonal-arkkitehtuuri. Sovelluksen tavoitteeksi asetettiin pyrkiä toteuttamaan arkkitehtuurin suunnittelu niin, että vaikka käytetään useamman eri pilvipalveluntarjoajan ympäristön erityyppisiä palveluita, sovellus on edelleen siirrettävissä toiseen pilviympäristöön. Valitun arkkitehtuurimallin soveltuvuutta ratkaisuvaihtoehtona sekä esimerkkisovellukselle suunnitellun arkkitehtuurin toimivuutta tutkittiin käytännössä toteuttamalla suunniteltu sovellus.

8.1 Sovelluksen pilviagnostinen arkkitehtuuri

Pilviagnostisessa lähestymistavassa serverless-teknologiaa käyttävässä sovelluksessa on sellaisia ohjelmakoodissa olevia komponentteja, että käytössä olevan halutun pilvipalveluntarjoajan ympäristön mukaisen serverless-funktion lähettämien tietojen vastaanotto, välittäminen eteenpäin, niiden tulkinta ja tietojen käyttäminen onnistuu sovelluksen sisältämissä eri komponenteissa itse sovelluksen tarkoituksen kannalta tarkoituksenmukaisella tavalla. Sovelluksen tulee myös kyetä muodostamaan tähän samaan pilvipalveluntarjoajan ympäristön serverless-funktioiden ymmärtämiä viestejä, kun se ottaa yhteyttä pilviympäristössä oleviin palveluihin. Yksi keskeinen palvelu on luonnollisesti itse serverless-alusta.

Pilviagnostisen sovelluksen tulee kyetä serverless-funktion lähettämän datan vastaanottajana tunnistamaan mistä ympäristöstä serverless-funktiolta tulevat tiedot ovat peräisin sekä tietojen lähettäjänä olemaan tietoinen mihin ympäristöön tietoja ollaan lähettämässä. Kun sovellus ottaa yhteyttä pilvipalveluntarjoajan ympäristön serverless-funktioon, tulee toimia tavalla, jota pilvipalveluntarjoajan ympäristö ymmärtää (ks. kuvio 10). Esimerkkisovelluksen arkkitehtuuriksi valittiin hexagonal-arkkitehtuuri, koska sen avulla voidaan suunnitella ja toteuttaa edellisen kaltaiset tarpeet täyttävä arkkitehtuuri. Pilviympäristöiksi valittiin Amazon AWS ja Azure, joiden serverless-alustojen ja palveluiden erojen vaikutukset siirrettävän sovelluksen suunnitteluun sekä toteuttamiseen ovat ratkaistavissa tällä arkkitehtuurilla.



Kuvio 10. Pilviagnostinen lähestymistapa, jossa sovellus voidaan julkaista eri ympäristöissä

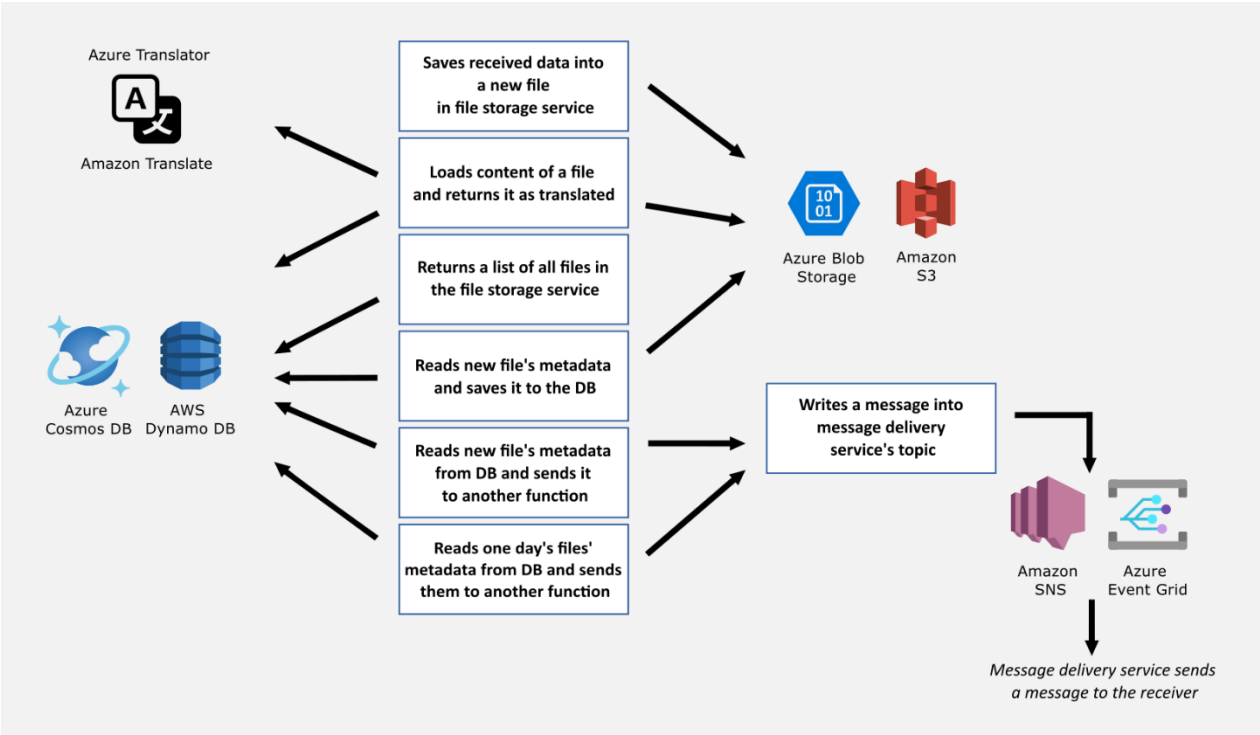
8.2 Esimerkkisovelluksen tarkoituksen kuvaus ja toiminnot

Pilviympäristöiksi valittiin kaksi ympäristöä, Amazon AWS ja Microsoft Azure. Ajoympäristöksi valittiin ohjelmointikieli Node.js. Esimerkkisovellukseen on valittu joukko erilaisia pilvipalveluntarjoajien palveluita. Palvelut valittiin niin että ne ovat usein käytössä erilaisissa serverless-teknologiaa käyttävissä sovelluksissa ja että vastaavan toiminnon mahdollistama palvelu oli saatavissa molemmissa pilviympäristöissä. Serverless-funktion näkökulmasta valitut samankin pilvipalveluntarjoajan ympäristössä olevat palvelut ovat keskenään erilaisia. Palvelujen tarkoitukseen ja toimintaan liittyvä erilaisuus saa aikaan sen, että myös serverless-funktioille välitettävät tiedot vaihtele-vat. Valitsemalla useita erilaisia palveluita ja asettamalla tavoitteeksi suunnitella hexagonal-arkki-tehtuurin mukainen arkkitehtuuri serverless-teknologiaa käyttävälle sovellukselle, jossa nämä palvelut ovat mukana, on mahdollista saavuttaa tilanne, joka vastaa tutkimukselle asetettuihin tavoitteisiin. Toteuttamalla valitun arkkitehtuurimallin mukainen sovellus, syntyvä lopputulos antaa mahdollisuuden todeta, onnistuttiinko toteuttamaan sellainen arkkitehtuuri ja sovellus, joka on siirrettävissä toiseen ja erilaiseen pilviympäristöön.

”Kielikone”-sovellus mahdollistaa tekstien tallentamisen pilviympäristöön ja niiden kääntämisen toiselle halutulle kielelle ja käännöksen tuloksen palauttamisen tarkasteltavaksi. Tekstit ovat tiedostoissa, jotka tallennetaan pilviympäristön tiedostopalveluun. Esimerkkisovellus sisältää 8 erilaista toimintoa. Suunnitellun esimerkkisovelluksen kaikki toiminnallisuudet käyttävät osana

toimintaansa pilvipalveluntarjoajien Amazon AWS ja Azuren erilaisia palveluita (ks. kuvio 11). Sovellukselle on valittu seuraavat toiminnot, joita sen avulla voidaan tehdä:

- Tekstiä sisältävän tiedoston tallentaminen pilviympäristön tiedostopalveluun
- Valitun tiedoston sisältämän tekstin palauttaminen käännettynä valitulle kielelle
- Palauttaa listan kaikista tiedostoista, jotka on tallennettu sovellukseen
- Tiedostopalveluun lisätyn tiedoston kuvaustietojen kirjoittaminen tietokantaan
- Hakee uuden tiedostopalveluun lisätyn tiedoston metatiedot tietokannasta ja kutsuu toista funktiota, jolle välittää nämä tiedot.
- Kerää tiedostopalveluun lisättyjen uusien tiedostojen metatiedot yhteen viestiin, joka välitetään toiselle funktiolle
- Lähettää toiselta funktiolta saamansa viestin viestinvälityspalvelulle
- Pilviympäristön viestinvälityspalvelu lähettää viestin vastaanottajan sähköpostiin



Kuvio 11. Esimerkkisovelluksen toiminnot ja käytössä olevat pilviympäristöjen palvelut

8.3 Hexagonal-arkkitehtuurimallin soveltaminen esimerkkisovelluksessa

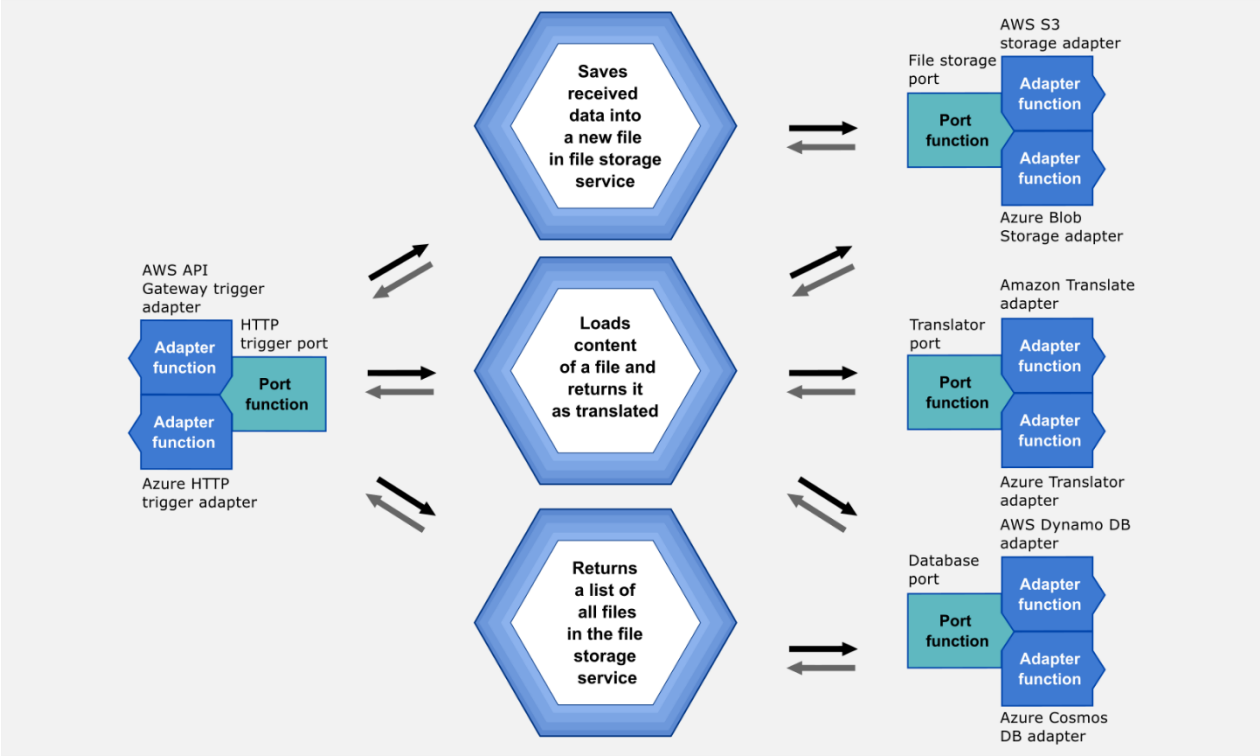
Seuraavaksi tutkittiin, miten hexagonal-arkkitehtuuria voidaan soveltaa esimerkkisovelluksen arkkitehtuurin suunnittelussa. Serverless-teknologiaa käyttävässä sovelluksessa on

pilvipalveluntarjoajan ympäristöön luotuja yksilöllisesti nimettyjä serverless-funktioita, joilla on erityinen käsittelijäfunktio, jolle välitetään event- ja context -objektien tiedot. Hexagonal-arkkitehtuurissa sovelluk-sen komponenteilla, joiden rooli on olla adapterina (adapter) tai porttina (port) on ratkaiseva vaikutus siirrettävyyteen. Serverless-funktion suoritus käynnistyy esimerkkisovelluksessa triggeristä tai AWS-pilviympäristön tapauksessa käynnistyminen voi tapahtua myös toisen funktion suorasta kutsusta. Azuren ympäristössä toista funktiota kutsutaan käyttäen triggeriä samaan tapaan kuin jos kutsuja olisi ollut jokin pilviympäristön palvelu.

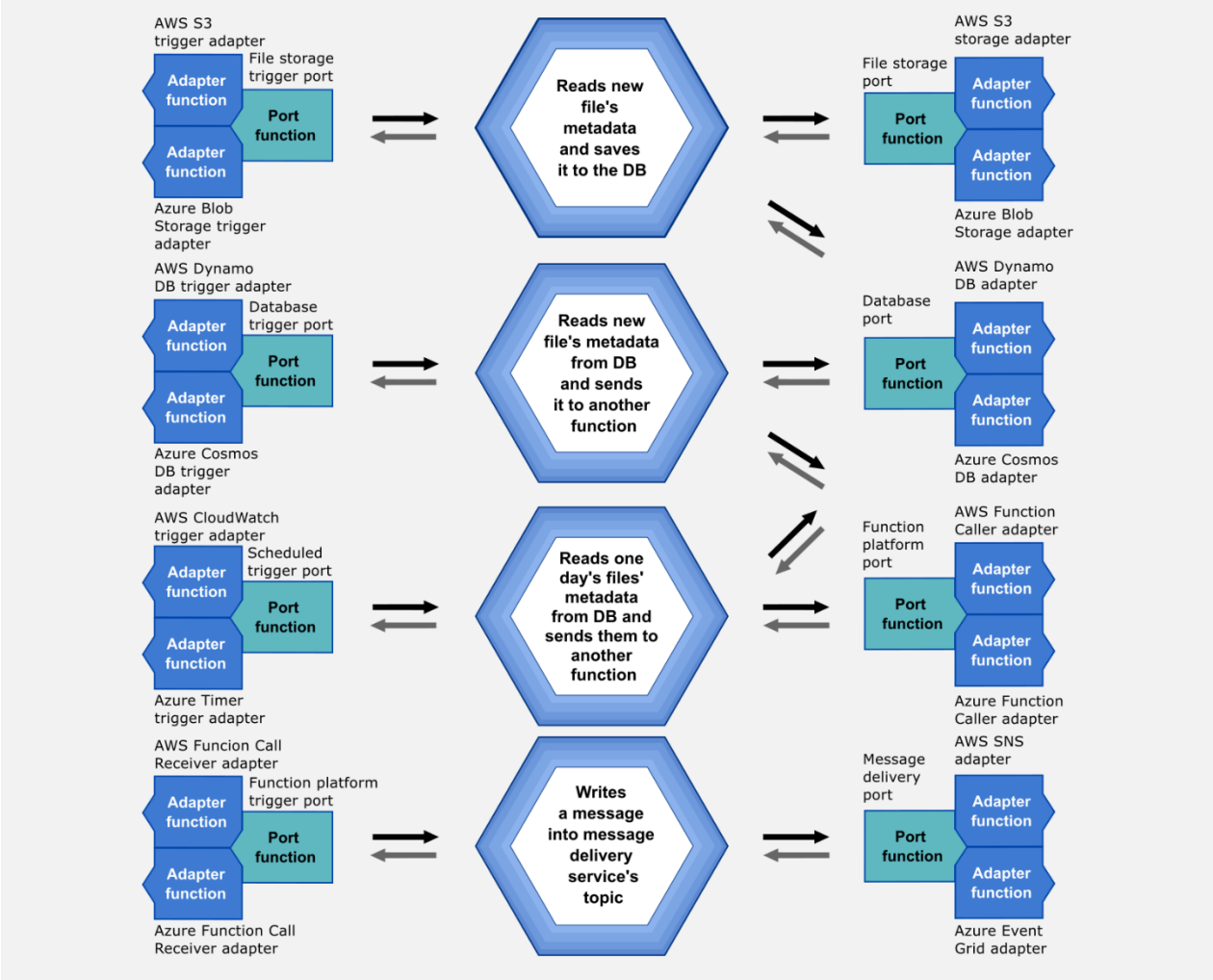
Esimerkkisovelluksen arkkitehtuurin suunnittelun tavoitteena oli osoittaa valitun arkkitehtuurimallin kyky ratkaista siirrettävyyden ongelma. Ratkaisevinta esimerkkitoteutuksessa oli se, että siinä käytetty tapa tekee molemmille alustoille yhteisistä adapter- ja port -roolien komponenteista huolellisesti suunniteltuina ja toteutettuina yleiskäyttöisiä eli uudelleen käytettäviä. Hexagonal-arkkitehtuurin ydinroolin osiossa olevien funktioiden määrällä hyvin laaditussa siirrettävyyttä varten suunnitellussa arkkitehtuurissa ei ole merkitystä, koska ydin osuuden toiminnot ovat itsenäisiä ja käyttävät hyväksi samoja port- ja adapter -roolien komponentteja. Kun adapterit ja adaptereille yhteinen port-roolin komponentti on rakennettu oikein, uuden ydinosion toiminnallisuuden lisäämisen yhteydessä ei tarvitse tehdä mitään muutoksia adapter- tai port -komponentteihin, koska ne voidaan laatia niin, että ne osaavat kutsua oikein jokaista uuden toiminnallisuuden sisältävää ydinosiossa olevaa koodia.

8.4 Sovelluksen toiminnot hexagonal-arkkitehtuurissa

Koko esimerkkisovelluksen arkkitehtuuri suunniteltiin käyttäen hexagonal-arkkitehtuuria. Sovellus muodostettiin toimintojen ympärille ja jokaiseen toimintoon liittyy oma pilvipalveluntarjoajan ympäristössä oleva serverless-funktio. Kaikkien esimerkkisovelluksen toimintoihin liittyvien komponenttien koodeissa esiintyvät funktiot ovat synkronisia, joten ne odottavat vastausta kutsumaltaan toiselta sovelluksen komponentilta (ks. kuvio 12). Sovelluksen toimintoa ”pilviympäristön viestinvälityspalvelu lähettää viestin vastaanottajan sähköpostiin” ei näy arkkitehtuurikuvassa koska toiminnon suorittaminen tapahtuu käyttäen pilviympäristön tarjoamaa ja hallinnoimaa palvelua. Viestin lähettämisessä käytetään Amazon Simple Notification Service (SNS)- ja Azure Event Grid -palveluita (ks. kuvio 13).



Kuvio 12. Sovelluksen toiminnot: Tiedoston luonti, tekstisisällön käännös ja tiedostohaku



Kuvio 13. Sovelluksen toiminnot: Tietojen käsittelyt, viestien muodostus ja lähettäminen

Arkkitehtuurin adapter- ja port -roolien komponentit jaettiin kahteen ryhmään: Funktion käynnistämiseen liittyviin komponentteihin ja pilviympäristöjen palvelukohtaisen toimintoon liittyviin komponentteihin. Esimerkkisovelluksen toimintoihin liittyy monia pilviympäristökohtaisia palveluita, joilla on keskeinen rooli sovelluksen toiminnassa. Adapter- ja port -roolien komponenttien kautta muodostetaan yhteys pilviympäristön palveluihin.

Funktion käynnistämiseen liittyvät adapter- ja port -roolien komponentit

Serverless-funktion käynnistämiseen liittyviä komponentteja päätettiin kutsua tässä sovelluksessa ”trigger-tehtävään” liittyviksi komponenteiksi. Trigger-tehtävässä toimiva adapter-komponentti laaditaan sekä tietyn pilviympäristön että tämän tietyn pilviympäristöstä valitun palvelun triggerikohtaiseksi adapteriksi. AWS-pilviympäristön tapauksessa adapteri voi toimia myös oman funktion suoran kutsun yhteydessä funktion kutsuun vastaavana adapterina. Käytännössä tämä tarkoittaa, että adapter-komponentille muodostuu merkittäviä tehtäviä, koska se joutuu toimimaan ns. tulkkina pilviympäristön palvelusta (ja myös toiselta funktiolta AWS-pilviympäristön tapauksessa) tulevien tietojen käsittelijänä. Lisäksi adapterin tulee kyetä välittämään portille sille pilviympäristöstä tulleet tiedot yhtenäisessä formaatissa. Yhtenäinen formaatti tarkoittaa sitä, että kun pilviympäristöstä tulee erilaista dataa (event ja context) omassa formaatissaan parametreina adapter-roolin funktiolle, adapterifunktioiden tehtävä on muuntaa saadut tiedot port-funktion koodin odottamaan yhtenäiseen formaattiin.

Adapter-komponentin tulee kutsua porttia ja välittää sille yhtenäiseen muotoon muutetut sovelluksen toiminnan kannalta tärkeimmät tiedot, jotka se on kerännyt event- sekä context-objektien sisällöistä. Node.js-ajoympäristön tapauksessa myös callback-funktio toimitetaan koodina parametrin sisällä. Esimerkkisovelluksen adapter-funktiossa on määritettynä minkä pilviympäristön adapteri se on ja tämä tieto välitetään eteenpäin, jotta muut komponentit voivat tarvittaessa käyttää tätä tietoa hyväksi suorittaessaan tehtäväänsä. Kun pilviympäristön palvelu (kuten AWS API Gateway) joka on käynnistänyt funktion suorittamisen odottaa saavansa jonkin vastauksen, on adapterin tehtävänä muodostaa vastaus tekemänsä toiminnon lopuksi ja palauttaa se tälle palvelulle.

Trigger-tehtävässä toimivan adapter-roolin funktioon liittyvän portin tulee vastaanottaa adapterilta tarvitsemassaan formaatissa kaikki tarvitsemansa tiedot sekä ne tiedot, joita core-roolissa

toimiva osa sovelluksesta tarvitsee oman tehtävänsä suorittamiseen. Portin odottama yhteinen formaatti tulee laatia hyvin ja määritellä portin sisältö, jotta portista tulee uudelleen käytettävä. Esimerkkisovelluksessa port-roolin komponentin päätehtävänä on saamiensa tietojen avulla valita se sovelluksen ydinosion (core) sisältämä kooditiedosto, jota se kutsuu seuraavaksi, ja jolle se välittää adapterilta saamansa tiedot. Port-roolin komponentit on laadittu serverless-funktion suorituksen käynnistävien triggerien tyyppikohtaisesti. Porttina toimiva komponentti ei tiedä mitään siitä, millaisia tietoja pilviympäristöstä sille tiedot toimittaneelle adapterille on tullut. Portin ei myöskään tarvitse edes tietää sitä millainen pilviympäristö tai millaisia pilviympäristöjä on adapterin taustalla mahdollisesti käytössä (vaikka nämä tiedot ovatkin saatavilla adapterin toimittamassa JSON-objektissa).

Core-komponentin tehtävä

Core-roolin tehtävän hoitavaa komponenttia kutsutaan port-komponentista, ja se vastaanottaa portin välittämät tiedot. Näitä tietoja ydinosion komponentti käyttää oman tehtävänsä suorittamisessa. Core-komponentin tehtävänä on vastata esimerkkisovelluksen toiminnon ydinosion sovelluslogiikasta. Voidakseen hyödyntää pilviympäristön palveluita tehtävänsä suorittamiseksi, se käyttää hyväkseen pilviympäristön palveluihin liittyviä port- ja adapter -roolien komponentteja. Esimerkiksi jos ydinosio haluaa käyttää pilviympäristön tiedostopalvelua osana suorittamaansa tehtävää, niin se käyttää tähän toiminnallisuuteen tarkoitettua porttia ja valittu portti osaa puolestaan käyttää oikeaa pilviympäristökohtaista adapteria. Koska esimerkkisovelluksen funktiot ovat synkronisia, ydinosiota kutsunut portti odottaa vastausta kutsumaltaan komponentilta. Kun toiminnon core-roolin hoitava osuus on saanut suoritettua omat tehtävänsä, se palauttaa prosessin tuloksena tiedon onnistuneesta tai epäonnistuneesta suorituksesta sitä kutsuneelle trigger-tehtävän portille.

Pilviympäristöjen palvelukohtaisen toiminnon adapter- ja port-roolin komponentit

Pilviympäristön palveluihin liittyviä palvelukohtaisen toiminnon komponentteja päätettiin kutsua tässä sovelluksessa ”service-tehtävään” liittyviksi komponenteiksi. Service-tehtävän portit ja adapterit tarjoavat ydinosuuden koodille mahdollisuuden hyödyntää pilvipalveluntarjoajien ympäristöjen eri palveluja. Service-tehtävän portti ottaa vastaan ydinosuuden ohjelmakoodilta parametrina ne tiedot mitä se tarvitsee suorittaakseen pilviympäristön palveluun liittyvän palvelukohtaisen toiminnon.

Port-komponentin päätehtävä on päätellä missä pilviympäristössä sovellusta ajetaan ja käyttää tässä tehtävässä apuna sitä tietoa, mikä on saatu tietoon ihan ensimmäisessä vaiheessa esimerkkisovelluksen jokaisessa toiminnossa eli trigger-tehtävässä toimivassa adapterissa. Tieto pilviympäristöstä on toimitettu jokaisen komponentin toimesta portille asti. Pilviympäristön kertovaa tietoa käytetään, jotta voidaan valita oikea adapteri suorittamaan sovelluksen toiminnan kannalta tiettyyn pilviympäristön palveluun liittyvä ja haluttu toiminto. Samoin toimitaan, kun palvelun sijaan kutsutaan toista serverless-funktiota (tämä tilanne koskee vain AWS-pilviympäristöä, jossa funktiota voidaan kutsua suoraan).

Port-komponentti käyttää valittua adapteria suorittaakseen halutun toiminnon ja se myös odottaa adapterilta tähän toimintoon liittyvää vastausta. Kun portti saa adapterilta vastauksen, se välittää vastauksen sellaisenaan takaisin ydinosion komponentille. Service-tehtävän adapteri saa portilta parametreina kaikki tiettyyn pilviympäristöön ja palvelukohtaiseen toimintoon liittyvät tiedot. Adapteri käyttää näitä tietoja suorittaessaan palvelukohtaisen tehtävänsä. AWS-pilviympäristön tapauksessa adapteri käyttää apuna tarvittavat toiminnot sisältävää SDK-kirjastoa. Azuren pilviympäristön tapauksessa kaikki esimerkkisovelluksen toimintojen service-tehtävän adapterit käyttävät Blob Storage-, Cosmos DB- tai Event Grid -palveluihin liittyviä Azuren Github-repositoriosta saatuja kirjastoja. Kirjastot asennettiin osaksi projektia npm-työkalun avulla. Suoritettuaan pilviympäristön palveluun liittyvän palvelukohtaisen toiminnon, adapteri palauttaa toimintoon liittyvän tuloksen portille. Toiminnon suorituksen tulos sisältää tiedon siitä onko toiminnon suoritus onnistunut tai epäonnistunut sekä toiminnosta riippuen myös pyydettyä dataa.

8.5 Tarkempaan tarkasteluun valitut sovelluksen toiminnot

Tarkempaan tarkasteluun valittiin kaksi sovelluksen toimintoa, joita esittelemällä voitiin kuvata sekä arkkitehtuurimallin soveltuvuus ratkaisuvaihtoehtona että sovelluksen toteuttamista tavalla, joka samalla kertoo millä tavoin koko sovellus on mahdollista suunnitella sekä toteuttaa. Valitut toiminnot käyttävät useita erilaisia pilviympäristöjen palveluita. Ensimmäisessä toiminnossa luetaan tiedoston sisältö tiedostopalvelusta ja suoritetaan tiedoston tekstisisällön kääntäminen lähdekielestä halutulle kohdekielelle (ks. kuvio 14). Toisessa toiminnossa kerätään tietokannasta halutun päivämäärän aikana tiedostopalveluun luotujen tiedostojen kuvaustiedot (metatiedot) ja koostetaan nämä tiedot yhteen viestiin. Tämän jälkeen kutsutaan toista serverless-funktiota, jolle välitetään viestin sisältö (ks. kuvio 23).

Jokaisen esimerkkisovelluksen toiminnon ohjelmakoodi on jaoteltu hexagonal-arkkitehtuurimallin mukaisesti adapter-, port- sekä core-roolien komponentteihin ja jokainen yksittäinen komponentti on oma tiedostonsa. Toiminnon toteutus koostuu useista tiedostoista ja yhden tiedoston sisällä voi olla yksi tai useampi funktio. Eri komponenttiroolien tiedostot on jaoteltu kansiorakenteessa rooliaan vastaaviin kansioihin, eli adapter-rooliin liittyvät tiedostot ovat adapter-kansiossa, port-rooliin liittyvät tiedostot port-kansiossa ja core-roolin tiedostot core-kansiossa. Lisäksi sovelluksessa on konfiguraatiotiedostoja ja apufunktioita, jotka on tarkoitettu kaikkien muiden eri komponenttien käyttöön. Adapteri käyttää näitä tietoja apunaan suorittaessaan palvelukohtaisen tehtävänsä.

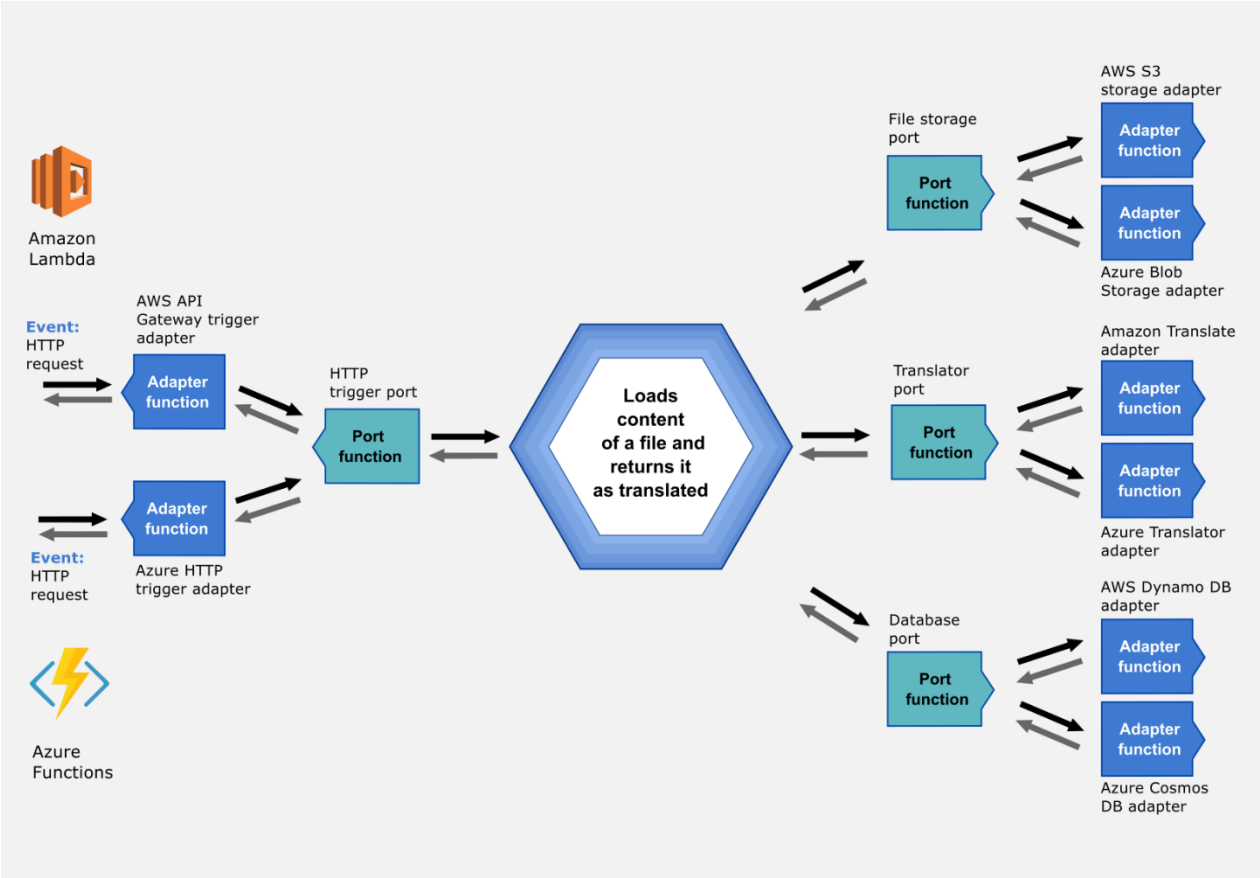
Hexagonal-arkkitehtuuria käyttävän sovelluksen toteutus tapahtuu käyttäen samankaltaisia ratkaisuja toistaen eri sovelluksen komponenteissa. AWS- ja Azure -pilviympäristöissä ilmeni kuitenkin esimerkkisovelluksen toteutukseen liittyen joitain eroja, jotka oli myös huomioitava osana suunnittelua ja toteuttamisvaihetta. Sovelluksessa on toiminto, jonka päätteeksi käytetään pilviympäristön viestinvälityspalvelua. Viestinvälityspalvelun rooli sovelluksessa on hoitaa viestien lähetys halutulle vastaanottajalle. Viestien lähettämiseen liittyy molemmissa pilviympäristöissä aihepiiri (topic), johon sovelluksen toiminto ”Writes a message into message delivery service’s topic (ks. kuvio 11) lisää viestin kun se halutaan välittää eteenpäin.

Aihepiiri on määritelty molemmissa pilviympäristöissä omaksi nimetyksi resurssikseen viestinvälityspalveluun, ja siihen voidaan liittää eteenpäin välitettävää sisältöä. Resurssin tehtävänä on olla osa prosessia, jossa viestintävälityspalvelu lähettää aihepiiriin tulleet uudet sisällöt esimerkiksi sitä seuraaville toisille palveluille tai henkilöille, jotka ovat rekisteröityneet eli lisätty aihepiirin tilaajiksi. Tilaaminen tarkoittaa käytännössä esimerkkisovelluksessa sitä, että aihepiiri tietää mihin sähköpostiosoitteisiin sen tulee välittää saamansa viestit. AWS-pilviympäristössä viestinvälityspalvelussa oli mahdollista konfiguroida sähköpostiosoite halutun aihepiirin tilaajaksi. Azuren ympäristössä ei ollut mahdollista käyttää viestinvälityspalvelua sähköpostiviestin lähettämiseen. Azuressa tähän tehtävään voi kuitenkin käyttää kolmannen osapuolen sovellusta, kuten SendGrid-sähköpostialustaa, joka tarjoaa tarvittavan sähköpostipalvelimen. SendGrid-alustan käyttöön ottamiseksi kannattaa luoda oma serverless-funktio, joka saa lähetettävän viestin tiedot ensin Azuren Event Grid -palvelulta ja lähettää ne edelleen SendGrid-sähköpostipalvelulle.

8.6 Tiedosto- ja käännöspalvelua sekä tietokantaa käyttävä sovelluksen toiminto

Tässä esimerkkisovelluksen toiminnossa keskeinen tehtävä on saada valitun tiedoston sisältämä teksti käännettyä halutulle kielelle käyttäen pilvipalveluntarjoajan käännöspalvelua. Toiminnon serverless-funktion suoritus käynnistyy HTTP-pyynnön muodostaman tapahtuman kautta, eli funktion suorituksen käynnistävä event-tapahtuma on HTTP-pyyntö. AWS-ympäristössä HTTP-pyynnön vastaanottaa API Gateway-palvelu, jonka hallinnoima URL-osoite on liitetty triggerinä toiminnon serverless-funktioon ja jonka kautta funktion suoritus aloitetaan (ks. kuvio 14).

Azuren pilviympäristössä HTTP-pyynnön vastaanottaa funktioon liittyvä URL-osoite, joka luodaan automaattisesti serverless-funktiolle, kun se luodaan. Funktioon liitetty HTTP trigger vastaa funktion suorittamisen käynnistämisestä kyseistä URL-osoitetta kutsuttaessa. Molemmissa pilviympäristöissä serverless-funktion suoritus alkaa trigger-tehtävässä toimivasta adapterista, joka on käsittelijäfunktio (handler). Se saa parametreina sekä suorituksen käynnistävään HTTP-pyyntöön liittyvän event-objektin sekä suoritusympäristöön liittyvän context-objektin.



Kuvio 14. Serverless-funktio, joka lukee tiedoston sisällön ja kääntää sen halutulle kielelle

Käännöstoiminnon trigger-tehtävän adapterit

Tähän sovelluksen toimintoon liittyy kaksi erilaista trigger-tehtävän adapter-roolin sovelluskomponenttia: Yksi molemmille valituista pilviympäristöistä. Nämä komponentit reagoivat sovelluksen käynnistävään tapahtumaan ja ovat serverless-funktion käsittelijäfunktioita. AWS API Gateway Trigger -adapteri (ks. kuvio 15) sisältää funktion, joka ottaa vastaan parametrina event- ja context-objektit. Azure Http Trigger -adapteri (ks. kuvio 16) sisältää myös funktion, joka vastaanottaa event- ja context-objektit. Event-objektit sisältävät HTTP-pyyntöön liittyviä arvoja, kuten esimerkiksi Method, Path ja Query Parameters. Adapterin ohjelmakoodissa poimitaan tärkeimmät arvot sekä event- ja context-objekteista AWS- tai Azure -pilviympäristön formaatin mukaisena.

```
const httpTriggerPort = require('../ports/http_trigger_port');
const helperLib = require('../lib/helper_lib');

module.exports.handler = function(event, context, callback) {

 var logger = {}; // AWS logging fix
 logger.log = console.log;
 const cloudEnv = 'AWS'; // set cloudEnv-value

 // Parse values from the AWS Http Trigger event
 var eventValues = {
  'method': event.httpMethod,
  'path': event.path,
  'queryParameters': event.queryStringParameters,
  'pathParameters': event.pathParameters,
  'body': event.body
 };

 // ContextValues-object is created in helperLib.getContext function
 var contextValues = helperLib.getContext(event, context, cloudEnv, logger);

 // Wrap eventValues and contextValues-objects into one parameterValues-JSON object
 var parameterValues = {
  'eventValues': eventValues,
  'contextValues': contextValues
 };

 // Call port with parameterValues (contains event and context), logger-object for logging and callback-function
 httpTriggerPort.httpTrigger(parameterValues, logger, function(error, results) {
  callbackFunction(error, results, callback); // Function to handle AWS API Gateway callback and HTTP-responses
 });

};
```

Kuvio 15. Trigger-tehtävän hoitava AWS API Gateway Trigger adapteri

Adapteri muodostaa niistä uuden objektin (parameterValues), jonka se välittää seuraavalle komponentille parametrina. Tähän parametriin tallennetaan myös tieto siitä, että käytössä oleva pilviympäristö on AWS tai Azure. Koska havaittiin, että lokien kirjoittaminen toimii eri tavoin eri pilviympäristöissä, trigger-tehtävän adaptereissa luodaan vielä lokitietoja varten oma objekti (logger), joka myös välitetään seuraavalle komponentille. Näiden kahden pilviympäristön adapterfunktiot eroavat toisistaan vain siinä, miten ne vastaanottavat event-objektissa tulevat HTTP-

pyynnön arvot, lokien kirjoittamiseen luotu muuttuja luodaan ja miten paluuviesti muodostetaan. Toiminnon päätteeksi adapter-funktio saa paluuarvona onnistuiko vai epäonnistuiko koko toiminnon suoritus ja tekee siitä HTTP-response-objektin palautettavaksi. HTTP-response-objekti palautetaan adapter-funktion kautta pilviympäristöön sille, joka lähetti alkuperäisen HTTP-requestin event-tapahtuman yhteydessä

```
const httpTriggerPort = require('../ports/http_trigger_port');
const helperLib = require('../lib/helper_lib');

module.exports.handler = function(context, event) {

 var logger = {}; // Azure logging fix
 logger.log = context.log;
 const cloudEnv = 'Azure';  // set cloudEnv-value

 // Parse values from the Azure Http Trigger event
 var eventValues = {
  'method': event.method,
  'path': event.url,
  'queryParameters': event.query,
  'pathParameters': event.params,
  'body': JSON.stringify(event.body)
 };

 // ContextValues-object is created in helperLib.getContext function
 var contextValues = helperLib.getContext(event, context, cloudEnv, logger);

 // Wrap eventValues and contextValues-objects into one parameterValues-JSON object
 var parameterValues = {
  'eventValues': eventValues,
  'contextValues': contextValues
 };

 // Call port with parameterValues (contains event and context), logger-object for logging and callback-function
 httpTriggerPort.httpTrigger(parameterValues, logger, function(error, results) {
  callbackFunction(error, results, context); // Function to handle Azure Http Trigger HTTP-responses
 });
};
```

Kuvio 16. Trigger-tehtävän hoitava Azure Http Trigger adapteri

Sama toimintatapa pätee kaikkiin muihinkin trigger-tehtävän adaptereihin

Kaikki triggerit AWS-pilviympäristössä käynnistävät serverless-funktion samalla tavalla riippumatta mikä palvelu toimii triggerinä ja funktio saa aina saman nimiset parametrit. Azuren ympäristössä serverless-funktiot toimivat samalla tavoin, erona välitettävät parametrit, jotka ovat päinvastaisessa järjestyksessä. Pilviympäristöstä ja käytössä olevasta palvelusta riippuen parametreina vastaanotetut event- ja context -objektit eroavat sisällöltään. Tämän vuoksi palvelukohtaisen adapterin tärkein tehtävä onkin valita näistä kahdesta parametrista tärkeimmät arvot, koostaa ne yhtenäisen formaatin mukaiseen muotoon uudeksi JSON-objektiksi (parameterValues) sovellusta varten ja välittää tiedot eteenpäin seuraavalle sovelluksen komponentille. Edellinen toimintatapa mahdollistaa sen, että jokaista käytössä olevaa pilviympäristön palvelua ja pilvessä olevaa triggeriä

varten on mahdollista luoda triggeriin liittyvä adapteri, joka osaa tehdä edellä kuvatun muunnoksen yhteiseen muotoon sovelluksen toimintaa varten. Kun adapteri kutsuu seuraavaksi trigger-tehtävään liittyvää porttia se saa tiedot tarvitsemassaan formaatissa. Tämän toimintatavan kautta minkä tahansa pilvipalveluntarjoajan ympäristössä olevan palveluun liittyvä trigger on mahdollista liittää serverless-funktioon, jonka sisältämä ohjelmakoodi käyttää hexagonal-arkkitehtuuria.

Käännöstoiminnon trigger-tehtävän portti: HTTP Trigger Port

Pilviympäristökohtaiset HTTP-triggereiden tapahtumia käsittelevät molemmat adapterit kutsuvat seuraavaksi samaa komponenttia eli molemmille yhteistä HTTP Trigger Port -komponenttia (ks. kuvio 17). Se ottaa vastaan adaptereilta parametreina yhtenäiseen muotoon muunnetut tiedot JSON-objektissa (parameterValues), lokitietojen objektin (logger) ja callback-funktion. Callback-funktiota käytetään myöhemmin, kun toiminnon lopputulosta palautetaan takaisin trigger-tehtävän adapterille. Portti osaa valita sen ydinosion komponentin, jota se kutsuu seuraavaksi ja välittää sille eteenpäin kaikki saamaansa parametrit parameterValues ja logger, sekä oman callback-funktionsa. Oikean ydinosion komponentin valinta tapahtuu parameterValues.contextValues. functionName-muuttujan avulla.

```
const helperLib = require('../lib/helper_lib');

module.exports.httpTrigger = function(parameterValues, logger, callback) {

 // Parse function name from contextValues-object
 const functionName = parameterValues.contextValues.functionName;

 if (functionName) {
  const filePath = helperLib.getCoreFunctionName(functionName, logger);
  const coreFunction = require(filePath);

  if (coreFunction) {

   coreFunction.coreFunction(parameterValues, logger, function(error, results) {

    if (error) {
     logger.log('Error in Http Trigger port callback: ', error);
     callback(error, null);
    }
    else {
     logger.log('Successful callback in Http Trigger port: ', results);
     callback(null, results);
    }
   });
  }
  else {
   logger.log('Error in http_trigger_port, cant find Core-file: ', filePath);
   callback({ 'error': 'Undefined error in function' }, null);
  }
 }
};
```

Kuvio 17. Funkion suorituksen käynnistymiseen liittyvän adapterin HTTP Trigger Port

Edellinen toimintatapa pätee kaikkiin muihinkin trigger-tehtävän portteihin

Port-komponentit on esimerkkisovelluksessa rakennettu niin, että sillä tiedolla missä pilviympäristössä serverless-funktiota suoritetaan tai kumman adapter-roolin komponentin kautta sitä on kutsuttu, ei ole vaikutusta portin toimintaan. Tämä on mahdollista, koska portti saa parametrinsa aina saman formaatin mukaisena ja pystyy tämän ansiosta välittämään ne samassa muodossa eteenpäin toiminnon ydinosion sovelluslogiikasta vastaavalle core-komponentille. Trigger-tehtävään liittyvät portit eivät siis ole pilviympäristökohtaiset, mutta esimerkkisovelluksessa ne ovat trigger-tehtävän tyypin mukaisesti luotuja. Jokaiselle trigger-tehtävän tyypille esimerkiksi HTTP ja Storage on hexagonal-arkkitehtuurissa tehtävä omat port-komponentit. Esimerkkisovelluksessa kaikki sen portit on toteutettu noudattaen samaa periaatetta. Jokaiselle trigger-tehtävän tyypille on luotu oma port-komponenttinsa, jotta näissä porteissa voitaisiin tulevaisuudessa tarvittaessa myös tehdä triggerin tehtävän tyypin mukaista datan käsittelyä sekä sovelluslogiikkaa.

Koska portit voidaan laatia niin, että sille välitettävät parametrit ovat aina samassa formaatissa ja ne myös välittävät sellaisenaan saamansa tiedot eteenpäin ydinosion komponentille, portti ei ole riippuvainen mistään tietystä adapterista tai ydinosion komponentista. Portti on täysin riippumaton sen suhteen millaiset trigger-tehtävän adapterit ovat siihen yhteydessä tai millaiseen ydinosion komponenttiin se itse on yhteydessä. Trigger-tehtävän portti on siten täysin ympäristöstä riippumaton ja uudelleenkäytettävä. Tämä esimerkkisovelluksen toteutustapa noudattaa hexagonal-arkkitehtuurimallin periaatteita eli täyttää arkkitehtuurimallin vaatimukset siitä, että sovelluksen kunkin toiminnon ydinosion sovelluslogiikka core-komponentissa tulee olla myös riippumaton ympäristöstä. Toteutustapa täyttää myös sen vaatimuksen, että ydinosion tulee olla myös uudelleenkäytettävä eli sitä on mahdollista käyttää minkä tahansa portti- ja adapteri -komponentin yhdistelmän kanssa.

Käännöstoiminnon core-komponentti

Ydinosion komponentissa tapahtuu jokaisen esimerkkisovelluksen toiminnon varsinaisen ydintehtävän suorittaminen. Komponentti ottaa vastaan parametreina portilta objektit parameterValues ja logger sekä callback-funktion. Komponentti poimii tarvitsemansa tiedot parameterValues.eventValues -objektista (ks. kuvio 18). Tässä toiminnossa core-komponentti käyttää näitä tietoja hakiessaan pilviympäristön tiedostopalvelusta sinne tallennetun valitun tiedoston sisällön ja silloin kun se lähettää tiedoston sisällön pilviympäristön käännöspalvelulle käännettäväksi toiselle kielelle.

Nämä edellä kuvatut toiminnot ydinosio tekee yksi vaihe kerrallaan, käyttäen ensin Database port ja adapter -komponentteja, sitten File storage port ja adapter -komponentteja ja lopuksi Translator port ja adapter -komponentteja (ks. kuvio 19). Näitä eri vaiheisiin liittyvien tehtävien portteja ja adaptereita käytetään pilviympäristön palvelujen kutsumiseen ja ydinosio odottaa vastausta jokaisen vaiheen kohdalla ennen kuin se voi siirtyä seuraavaan vaiheeseen.

```
const fileStorage = require('../ports/filestorage_port');
const database = require('../ports/database_port');
const translator = require('../ports/translator_port');
const helperLib = require('../lib/helper_lib');

module.exports.coreFunction = function(parameterValues, logger, callback) {

 const queryParameters = parameterValues.eventValues.queryParameters;
 const tableId = 1;
 const bucketId = 1;

 if (queryParameters) {

  const fileId = queryParameters.fileid;
  const fromLang = queryParameters.fromlang;
  const toLang = queryParameters.tolang;

  if (fileId && fromLang && toLang) { // Step 1: Get one record with Id from the Database

   database.getRecord(tableId, fileId, parameterValues.contextValues, logger, function(error, results) {

    if (error) {
     returnError('getAllFiles_core - Error in database.getRecord-callback: ', error, callback, logger);
    }
    else {

     if (results && results.item) { // Steps 2 and 3 in one function

      const loadFileName = results.item.fullfilename;

      loadFileAndTranslate(loadFileName, bucketId, fromLang, toLang, parameterValues.contextValues, logger, callback);
     }
     else {
      returnError('translateFile_core - Results dont have item: ', results, callback, logger);
     }
    }
   });
  }
  else {
   returnError('translateFile_core - fileId, fromLang or toLang not set: ', queryParameters, callback, logger);
  }
 }
};
```

Kuvio 18. Tiedoston tekstisisällön käännöstoimintoon liittyvä ydinkomponentti

```
function loadFileAndTranslate(loadFileName, bucketId, fromLang, toLang, contextValues, logger, callback) {

 // Step 2: Load the content of the file with the data got from the Database
 fileStorage.loadFile(loadFileName, bucketId, contextValues, logger, function(error, results) {

  if (error) {
   returnError('translateFile_core - Error in fileStorage.loadFile callback: ', error, callback, logger);
  }
  else {

   if (results && results.content) { // Step 3: Send the content of the file to the translator service to be translated

    const originalText = results.content;

    translator.translate(fromLang, toLang, originalText, contextValues, logger, function (error, results) {

     if (error) {
      returnError('translateFile_core - Error in translator.translate callback: ', error, callback, logger);
     }
     else {
      callback(null, results);
     }
    });
   }
   else {
    returnError('translateFile_core - Results dont have context: ', results, callback, logger);
   }
  }
 });
}


function returnError(logMessage, returnError, callback, logger) {

 logger.log(logMessage, returnError);

 callback({ 'error': logMessage + JSON.stringify(returnError) }, null);
}
```

Kuvio 19. Käännöstoiminnon ydinkomponentista kutsutaan porttia

Database-, File storage- ja Translator port sekä palvelukohtaiset adapter -komponentit

Esimerkkisovelluksen tapauksessa tietokannasta haetaan tietoja käyttäen apuna pilviympäristön palvelukohtaisia adaptereita sekä Database port -komponenttia ja fileid-tunnistetiedon avulla on mahdollista hakea tietokannasta halutun tiedoston kuvaustiedot (ks. kuvio 18). Tiedostopalveluun liittyvää adapteria käytetään noutamaan valitun tiedoston sisältämä teksti ja adapteri käyttää tiedoston kuvaustietoja tässä tehtävässään apuna. Ohjelmakoodissa adapterit hoitavat varsinaisen pilviympäristökohtaiseen osuuteen liittyvän vaiheet. Tiedostosta saatu tekstisisältö toimitetaan pilviympäristön käännöspalvelun kutsumista varten luodulle adapterille, jolle Translator port -komponentti ensin toimittaa tiedot mistä kielestä ja mille kielelle käännös tulee tehdä.

Käännöstoiminnon service-tehtävän portti (ks. kuvio 20) poimii myös saamastaan parametrista sille tärkeitä suoritusympäristöön liittyviä tietoja (parametri contextValues), joita se käyttää hyväkseen hakiessaan adapterien tarvitsemia pilviresursseihin liittyviä tietoja. Myös logger ja callback-funktio on välitetty parametreina edelleen kaikille service-tehtävän porteille. Tämä esimerkkisovelluksen sisältämä toiminto käyttää kolmea erilaista molempien pilvipalveluntarjoajien

ympäristön palvelua service-tehtävän adapter-komponenttiensa kautta. Koska nämä adapterit sekä portit ovat toteuttavissa lähes samankaltaisina, tarkempaan tarkasteluun valittiin tekstikäännökseen liittyvä Translator port -komponentti ja seuraavat adapterit: Amazon Translate ja Azure Translator.

```
const helperLib = require('../lib/helper_lib');

module.exports.translate = function(sourceLanguage, targetLanguage, textToTranslate, contextValues, logger, callback) {

 if (contextValues.cloud == 'AWS') {
  const amazonTranslateAdapter = require('../adapters/amazon_translate_adapter');

  amazonTranslateAdapter.translate(sourceLanguage, targetLanguage, textToTranslate, contextValues,
                                   logger, function(error, results) {
   if (error) {
    logger.log('translator_port, error in amazonTranslateAdapter.translate: "', error);
    callback(error, null);
   }
   else {
    callback(null, results);
   }
  });
 }
 else if (contextValues.cloud == 'Azure') {
  const azureTranslatorAdapter = require('../adapters/azure_translator_adapter');
  const translatorKey = helperLib.getEnvValue("TRANSLATOR_KEY", contextValues);
  const translatorEndpoint = helperLib.getEnvValue("TRANSLATOR_ENDPOINT", contextValues);
  const region = helperLib.getEnvValue("REGION", contextValues);

  azureTranslatorAdapter.translate(sourceLanguage, targetLanguage, textToTranslate, translatorKey,
                                   translatorEndpoint, region, contextValues, logger, function(error, results) {
  if (error) {
   logger.log('translator_port, error in azureTranslatorAdapter.translate: "', error);
   callback(error, null);
  }
  else {
   callback(null, results);
   }
  });
 }
};
```

Kuvio 20. Esimerkkisovelluksen Translator port -komponentti

Pilviympäristökohtaiset kääntämiseen liittyvät adapterit toimivat hieman eri tavoin, koska niiden kautta käytettävät pilviympäristöt ja -palvelut ovat erilaisia. AWS-pilviympäristössä käännöspalvelun kutsumisessa käytettiin SDK-kirjastoa, joka on käytettävissä AWS Lambda -ympäristön kautta ilman että sitä tarvitsee liittää mukaan projektiin esimerkiksi npm-työkalulla. Amazon Translaten funktio translateText suorittaa tekstin kääntämisen toiselle kielelle (ks. kuvio 21). Kun Translate-palvelua kutsunut adapteri saa paluuarvona käännetyn tekstin, se muodostaa siirtää sovelluksen kannalta yhteistä formaattia noudattavan JSON-objektin ja palauttaa sen callback-funktion kautta portille. Kun ydinosio on saanut tekstin käännettynä port-komponentilta, se palauttaa käännöksen callback-funktion avulla HTTP trigger port -komponentille ja edelleen trigger-tehtävän

pilviympäristökohtaiselle adapterille. Adapteri vastaa tulosten palauttamisesta HTTP response -objektin avulla koko toiminnon suorittamisen päätteeksi.

```
const AWS = require('aws-sdk');
const translate = new AWS.Translate({apiVersion: '2017-07-01'});
const helperLib = require('../lib/helper_lib');

module.exports.translate = function(sourceLanguage, targetLanguage, textToTranslate, contextValues, logger, callback) {

 var params = {
  SourceLanguageCode: sourceLanguage, /* required */
  TargetLanguageCode: targetLanguage, /* required */
  Text: textToTranslate, /* required */
 };

 translate.translateText(params, function(error, results) {

  if (error) {
   logger.log('amazon_translate_adapter.translate, error in translate.translateText: ', error);
   callback(error, null);
  }
  else {

   const returnValue = {
    "originaltext": textToTranslate,
    "translatedtext": results.TranslatedText,
    "sourcelanguage": sourceLanguage,
    "targetlanguage": targetLanguage,
    "translationservice": "Amazon Translate"
   };

   callback(null, returnValue);
  }
 });
};
```

Kuvio 21. Amazon Translate -palveluun liittyvä adapteri

Käännöstehtävään liittyvän Azuren ympäristön adapterille tulee toimittaa pilviympäristöjen palvelujen erilaisuuden vuoksi lisäksi tietoja serverless-funktion ympäristömuuttujista (ks. kuvio 22). Nämä tiedot ovat Translator Key-, Translator Endpoint- ja Region -tiedot. Lisäksi erona AWS-pilviympäristöön, Azuren Translator -palvelun käyttämiseksi ei ollut tarjolla vastaavaa SDK-kirjastoa ja ohjelmakoodin oli tehtävä HTTP-pyyntö. Azure Translator adapterin tarvitsee lähettää HTTP-pyynnön mukana Azuren Translator -palvelulle edellä mainitut tiedot. Nämä tiedot on tallennettu Azuren pilviympäristön Function App -resurssin yhteyteen ympäristömuuttujina, ja Translator port -komponentti osaa lukea nämä tiedot sille toimitetuista parametreista. Function App sisältää toiminnon serverless-funktion.

Request-kirjaston avulla Adapterin suorittama HTTP-pyyntö palauttaa Azure Translator adapter -komponentille vastauksena arvon, joka on joko onnistuneen käännöksen tilanteessa käännetty teksti tai jos kääntämisessä tapahtui jokin virhetilanne, palautetaan virheviesti. Saatuaan vastauksen käännöstehtävän adapteri luo uuden JSON-objektin, joka noudattaa samaa formaattia kuin

AWS-pilviympäristönkin tapauksessa. Sitten adapteri palauttaa sen takaisin Translator port -komponentille, komponentin callback-funktion kautta. Service-tehtävän adapterien vastuu on myös rajoitettu pelkän pilviympäristön palveluun liittyvän palvelukohtaisen toiminnon suorittamiseen eikä adapterien tehtäviin kuulu käsitellä ympäristömuuttujia.

```
const request = require('request');
const { v4: uuidv4 } = require('uuid');
const helperLib = require('../lib/helper_lib');

module.exports.translate = function(sourceLanguage, targetLanguage, textToTranslate,
                                    translatorKey, translatorEndpoint, region, contextValues, logger, callback) {
 const options = {
  method: 'POST',
  baseUrl: translatorEndpoint,
  url: '/translate',
  qs: {
   'api-version': '3.0',
   'from': sourceLanguage,
   'to': [targetLanguage]
  },
  headers: {
   'Ocp-Apim-Subscription-Key': translatorKey,
   'Ocp-Apim-Subscription-Region': region,
   'Content-type': 'application/json',
   'X-ClientTraceId': uuidv4().toString()
  },
  body: [{
   'text': textToTranslate
  }],
  json: true,
 };

 request(options, function(error, response, body){

  if (error) {
   logger.log('azure_translator_adapter.translate, error in calling translator URL-endpoint with request: ', error);
   callback({ 'error': 'Error in Azure translator Http call' }, null);
  }
  else {

   if (body && body.length > 0 && body[0].translations && body[0].translations.length > 0) {
    const resultTranslation = body[0]["translations"][0];
    const returnValue = {
     "originaltext": textToTranslate,
     "translatedtext": resultTranslation.text,
     "sourcelanguage": sourceLanguage,
     "targetlanguage": targetLanguage,
     "translationservice": "Azure Translator"
    };

    logger.log('azure_translator_adapter.translate, Successful Http-response from translator URL-endpoint: ', response);
    callback(null, returnValue);
   }
   else {
    logger.log('azure_translator_adapter.translate, response body missing translation: ', body);
    callback({ 'error': 'Error in Azure translator Http call' }, null);
   }
  }
 });
};
```

Kuvio 22. Azure Translator -palveluun liittyvä adapteri

Sovelluksen kaikkien toimintojen core-osuudet ja ydinosion hyväkseen käyttämät service-tehtävän portit ja adapterit toteutetaan samalla periaatteella

Toiminnon ydintehtävästä vastaava ydinosio saa tiedot aina portilta tietyssä formaatissa, ja kuten porttikomponentinkin tapauksessa, hexagonal-arkkitehtuurimallissa käytössä oleva pilviympäristö ei vaikuta ydinosion toiminallisuuteen. Ydinosion komponenttien ei tarvitse tehdä mitään pilvipalveluihin- tai ympäristöihin liittyviä päätöksiä tai hakea niihin liittyviä tietoja. Näin ydinosion sovelluslogiikka ei ole lainkaan riippuvainen eikä edes tietoinen siitä ympäristöstä, jossa sovellusta suoritetaan tai mitä pilviympäristöjen palveluita sen ympärillä on käytössä, koska ydinosio on ulkoistanut tehtävän hyväksi käyttämilleen porteille ja adaptereille. Koska esimerkkisovelluksen ydinosion komponentit ovat löyhästi sidoksissa muihin sovelluksen osiin, sen kanssa on mahdollista käyttää mitä tahansa trigger-tehtävän adapteria ja porttia. Sama periaate jatkuu, kun ydinosio käyttää toiminnon suorituksen edetessä hyväkseen service-tehtävän portteja eikä ole itse kytköksissä yhteenkään pilviympäristön palveluun.

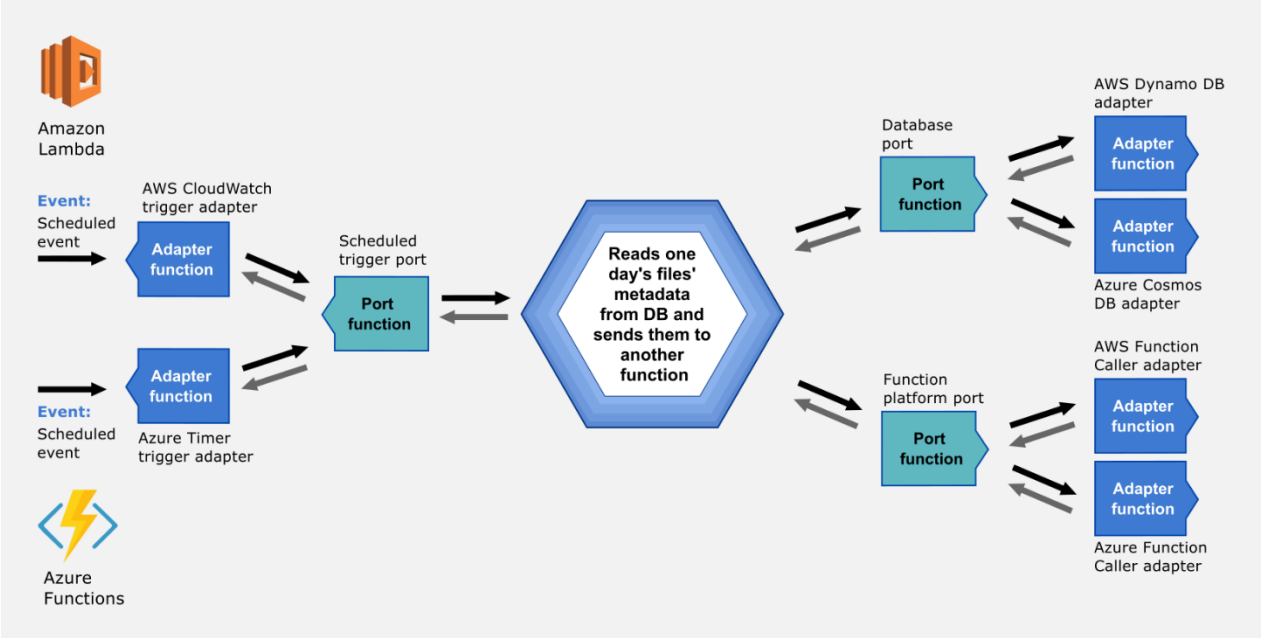
Service-tehtävää hoitava portti vaatii, että ydinosio toimittaa sille kaikki tarvittavat parametrit, jotka liittyvät portin tarjoamiin eri pilvialustoihin liittyviin toimintoihin. Service-tehtävän portit välittävät pilviympäristökohtaisille adaptereille kaikki saamansa parametrit sellaisenaan. Esimerkiksi service-tyyppistä tehtäväänsä hoitava File Storage Port osaa valita oikean pilviympäristön palvelun ja ympäristössä tarvittavan oikean adapterin sen mukaan missä pilviympäristössä sovellusta ajetaan. Muuttujaan contextValues.cloud on tallennettuna tieto siitä, missä pilviympäristössä toiminnon serverless-funktiota suoritetaan.

Hexagonal-arkkitehtuurimallia noudattavat pilviympäristökohtaiset service-tehtävän adapter-komponentit eroavat toisistaan kaikkein eniten, koska ne on laadittu tiettyä pilviympäristön palvelua varten. Näiden adapterien tehtävä on varmistaa, ettei port- tai core-roolien komponenttien tarvitse sisältää pilviympäristö- tai palvelukohtaista toiminallisuutta. Palveluita käytetään niiden omien API-rajapintojen kautta, jotka on kehitetty kyseisen pilvipalveluntarjoajan ympäristössä olevan palvelun tarpeisiin. Esimerkkisovelluksen service-tehtävän adaptereista suurin osa käyttää sellaista palvelua, joita muut sovelluksen komponentit eivät käytä. Monet Azuren pilviympäristön palveluista ovat käytettävissä vain kutsumalla niitä URL-osoitteen kautta, ja tähän tehtävään käytetään request-kirjastoa useassa service-tehtävän adapterissa. Jos osaksi sovellusta halutaan liittää jokin uusi pilviympäristön palvelu, jota ei vielä käytetä, ja luodaan service-tehtävää palvelevat

sopivat ja valitun palvelun tarpeiden kannalta oikein toimivat portit ja adapterit, voidaan hexagonal-arkkitehtuurimallin puolesta periaatteessa käyttää mitä tahansa pilviympäristöstä löytyvää palvelua.

8.7 Eri palveluja käyttävä ja toista serverless-funktiota kutsuva toiminto

Tässä esimerkkisovelluksen toiminnossa kerätään tiedostopalveluun lisättyjen uusien tiedostojen metatiedot yhteen viestiin, jotka välitetään toiselle serverless-funktiolle. Toiminnon serverless-funktion suoritus käynnistyy siihen liitetyn ajastimen kautta, eli funktion suorituksen käynnistävä event on ajastettu tapahtuma (ks. kuvio 23). AWS-pilviympäristössä suoritettava serverless-funktio on liitetty triggerillä EventBridge-palveluun (AWS CloudWatch) lisättyyn ajastettuun tapahtumaan, joka on määritetty suoritettavan kerran vuorokaudessa aina samaan kellonaikaan. Azuren pilviympäristössä serverless-funktion käynnistämisen hoitaa siihen liitetty Timer trigger, jonka määrityksissä serverless-funktio on asetettu myös käynnistymään kerran vuorokaudessa tiettyyn kellonaikaan. Suoritus alkaa myös tässä toiminnossa trigger-tehtävässä toimivasta adapter-roolin komponentista ja komponentti saa parametreina suorituksen käynnistävään ajastettuun tapahtumaan liittyvän event-objektin, että suoritusympäristöön liittyvän context-objektin.



Kuvio 23. Serverless-funktio, joka kokoaa tiedoista viestin ja kutsuu toista funktiota

Metatietojen välitys -toiminnon trigger-tehtävän adapterit ja portit sekä core

Tähän sovelluksen toimintoon liittyy kaksi erilaista trigger-tehtävän adapter-roolin komponenttia, yksi AWS-pilviympäristöä varten ja yksi Azuren pilviympäristöä varten. Toiminnon suoritus aloitetaan aina trigger-tehtävän adapter-komponenteista, jotka ovat samalla serverless-funktion käsittelijäfunktioita. Toiminnon trigger-tehtävän adapterit ja portit on toteutettu samalla periaatteella, kuin käännöstoiminnon vastaavat komponentit. Myös ydinosion komponentti toimii samoin periaattein tässäkin toiminnossa, eli se käyttää hyväkseen pilviympäristön palveluja service-tehtävän port- ja adapter -rooleissa toimivien komponenttien kautta. Myös tämä ydinosion komponentti saa portilta parametreina objektit parameterValues, logger ja callback-funktion. Ydinosion komponentin tehtävänä on poimia tarvitsemansa arvot ParameterValues.eventValues -objektista ja käyttää niitä hakiessaan tietokannasta kaikki yhden päivän aikana tiedostopalveluun lisättyjen tiedostojen metatiedot. Seuraavaksi komponentti muodostaa niistä JSON-muotoisen event-objektin ja Function Caller adapter kutsuu lopuksi toista serverless-funktiota välittäen tämän objektin muuttumattomana sille niin, että toinen serverless-funktio vastaanottaa sen omana event-parametrinaan.

Metatietojen välitys -toiminnon core- ja service-tehtävän port -komponentit

Ydinosion roolissa toimiva sovelluksen komponentti käyttää oman tehtävänsä suorituksessa kahta service-tehtävän porttia: Tietokantahaun suorittamiseen Database port- ja toisen funktion kutsumiseen Function platform port -komponenttia. Database port on rakennettu samoja periaatteita käyttäen kuin käännöstehtävän suorittavassa toiminnossa esitelty Translator port. Function platform port on komponentti, jonka kautta ydinosion roolissa toimivat komponentit voivat kutsumalla ja porttia hyväksi käyttäen lopulta käynnistää toisen pilviympäristön serverless-funktion ja välittää sille tarvittavan event-parametrin (ks. kuvio 24). Function platform port -tehtävässä toimivalle portille toimitetaan useita parametreja.

Parametri functionId määrittää mistä ympäristömuuttujasta suoritettavan funktion nimi ja tiedot haetaan. Parametri eventObject on JSON-muotoinen objekti ja aikanaan se lähetetään kutsuttavalle funktiolle event-objektina. Parametri contextValues on objekti, joka sisältää tärkeitä suoritusympäristöön liittyviä tietoja, kuten ympäristömuuttujia, joita service-tehtävän portti käyttää hyväkseen hakiessaan tietoja toiminnalle tarpeellisista pilviresursseista. Logger on objekti, jonka kautta voidaan sovelluksen koodissa kirjoittaa lokia suorituksen aikana. Callback-parametri on

funktiokoodi, jota ydinosio haluaa kutsuttavan sitten kun service-tyypin tehtävää suorittava portti on saanut valmiiksi oman osuutensa. Samoin periaattein kuin miten toimittiin käännöstehtävänkin tapauksessa, muuttujaan contextValues.cloud on tallennettu tieto siitä, missä pilviympäristössä serverless-funkiota suoritetaan. Service-tehtävän portti valitsee sen avulla sen service-tehtävän adapterin, jota se käyttää pilviympäristökohtaisen toiminnon suorittamiseen. Portti hakee suoritettavan serverless-funktion nimen ympäristömuuttujista käyttämällä tässä functionId-arvoa, joka toimitettiin sille parametrina. Service-tehtävän portti kutsuu saman tehtävän adapteria ja välittää sille kaikki saamansa parametrit sekä hakemansa tiedon suoritettavan serverless-funktion nimestä.

```
const helperLib = require('../lib/helper_lib');

module.exports.startFunction = function(functionId, eventObject, contextValues, logger, callback) {

 if (contextValues.cloud == 'AWS') {
  const awsLambdaFunctionAdapter = require('../adapters/aws_function_caller_adapter');
  const functionName = helperLib.getEnvValue("LAMBDA_FUNCTION_NAME_" + functionId, contextValues);
  const eventObjectText = JSON.stringify(eventObject);

  awsLambdaFunctionAdapter.startFunction(functionName, eventObjectText, contextValues, logger, function(error, results) {

   if (error) {
    logger.log('functionplatform_port, error in awsLambdaFunctionAdapter.startFunction: "', error);
    callback(error, null);
   }
   else {
    callback(null, results);
   }
  });
 }
 else if (contextValues.cloud == 'Azure') {
  const azureFunctionsFunctionAdapter = require('../adapters/azure_function_caller_adapter');
  const functionAppURL = helperLib.getEnvValue("FUNCTION_APP_URL_" + functionId, contextValues);
  const functionName = helperLib.getEnvValue("AZURE_FUNCTION_NAME_" + functionId, contextValues);
  const functionUrl = helperLib.getEnvValue("AZURE_FUNCTION_ENDPOINT_PATH_" + functionId, contextValues);
  const eventObjectText = JSON.stringify(eventObject);

  azureFunctionsFunctionAdapter.startFunction(functionName, functionAppURL, functionUrl, eventObjectText,
                                              contextValues, logger, function(error, results) {

   if (error) {
    logger.log('functionplatform_port, error in azureFunctionsFunctionAdapter.startFunction: "', error);
    callback(error, null);
   }
   else {
    callback(null, results);
   }
  });
 }
};
```

Kuvio 24. Portti välittää adapterille tiedon minkä nimistä serverless-funktiota kutsutaan

Metatietojen välitys -toiminnon palvelukohtaiset adapter -komponentit

Myös tämän toiminnon erilaiset pilviympäristöjen palveluihin liittyvien tehtävien suorittamiseen tarkoitetut adapter-roolin komponentit ovat toimintaidealtaan ja toteutukseltaan lähes samanlaisia keskenään. Niistä valittiin tarkemmin esiteltäväksi yksi adapteri, joka kutsuu toista serverless-funktiota sen sijaan, että kutsuisi jotain valmista palvelua. Sekä AWS-pilviympäristön että Azuren

Function Caller adapterit ottavat vastaan contextValues-, logger- ja callback parametrit. Parametrien kautta adapteri saa tiedot siitä minkä nimistä serverless-funktiota sen tulee kutsua ja tiedot millainen event-objekti sille tulee välittää. AWS-pilviympäristön SDK-kirjastosta käytetään funktiota nimeltä invokeAsync, jonka avulla adapteri kutsuu ja käynnistää toisen saman pilviympäristön serverless-funktion suorituksen sekä välittää sille event-objektin (ks. kuvio 25).

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda({ region: process.env.AWS_REGION });
const helperLib = require('../lib/helper_lib');

module.exports.startFunction = function(functionName, eventObjectText, contextValues, logger, callback) {

 var params = {
  FunctionName: functionName,
  InvokeArgs: eventObjectText
 };

 lambda.invokeAsync(params, function(error, results) {

  if (error) {
   logger.log('aws_function_caller_adapter.startFunction, error in lambda.invokeAsync: ', error);
   callback(error, null);
  }
  else {
   logger.log('aws_function_caller_adapter.startFunction, Successful lambda.invokeAsync: ', results);
   callback(null, results);
  }
 });
};
```

Kuvio 25. AWS Function Caller -adapteri

Pilvipalveluntarjoajan ympäristössä sijaitsevan serverless-funktion kutsumisessa adapteri käyttää apuna joko AWS Lambda-alustan kautta saatavaa SDK-kirjastoa tai Azuren Function App -resurssiin liitettyä Base URL -osoitetta. Azuren ympäristössä adapterin tarvitsee lisäksi tietää kutsuttavan serverless-funktioon viittaava osuus URL-osoitteesta, jonka avulla funktiota kutsutaan HTTP-pyynnön kautta. Nämä edellä mainitut tiedot ovat tallennettuna Function App -resurssin yhteyteen ympäristömuuttujina ja portti on rakennettu niin, että se osaa lukea nämä tiedot ympäristömuuttujista ja toimittaa ne Azuren ympäristön adapterille. Kuten käännöstehtävänkin toiminnossa tässä vaiheessa käytettiin apuna request-kirjastoa, jonka avulla suoritettu HTTP-pyyntö palauttaa adapterille response-vastauksen (ks. kuvio 26). Request-kirjaston funktion palauttama arvo on käytännössä serverless-funktion palauttama arvo.

Metatietojen välitys -toiminto ja adapteri joutuvat odottamaan sen aikaa, kunnes kutsutun serverless-funktion suoritus on päättynyt, jotta ne saavat HTTP-response-vastauksessa tämän toisen serverless-funktion suorittaman tehtävän lopputuloksen. Vasta tämän jälkeen adapteri voi saamansa

paluuviestin avulla muodostaa uuden JSON-objektin, jonka on oltava samassa formaatissa molempien pilviympäristöjen tapauksessa koska se toimitetaan service-tehtävään liittyvälle yhteiskäytössä olevalle port-komponentille. Kutsutun serverless-funktion käynnistyessä sen handler-käsittelijä-funktio saa adapterin toimittaman event-objektin oman event-parametrinsa kautta. Kun serverless-funktion kutsuminen on ohi, service-tehtävän adapteri kutsuu sitä kutsuneen portin callback-funktiota ja palauttaa tiedon siitä, onnistuiko toisen funktion käynnistäminen vai ei. Callback-funktiota kutsutaan, kun toisen serverless-funktion kutsuminen on tehty.

```
const request = require('request');
const helperLib = require('../lib/helper_lib');

module.exports.startFunction = function(functionName, functionHost, functionPath,
                                        eventObjectText, contextValues, logger, callback) {

 const functionUrl = "https://" + functionHost;

 const options = {
  method: 'POST',
  baseUrl: functionUrl,
  url: functionPath,
  qs: {},
  headers: {
   'Content-Type': 'application/json'
  },
  body: {
   'event': JSON.parse(eventObjectText)
  },
  json: true,
 };

 request(options, function(error, response, body){

  if (error) {
   logger.log('azure_function_caller_adapter.startFunction, error in calling other function: ', error);
   callback({ 'error': 'Undefined error in Azure function invoke Http call' }, null);
  }
  else {
   const resultObject = {
    'results': response
   };

   logger.log('Successful Http-response from function "' + functionName + '" URL-endpoint: ', response);
   callback(null, resultObject);
  }
 });
};
```

Kuvio 26. Azure Function Caller -adapteri

9 Pohdinta

Tutkimusprosessi osoittautui mielenkiintoiseksi ja haasteelliseksikin, eikä vähiten siksi että tutkimusongelma koskettaa laajasti koko organisaatiota, eikä vain sovellusten toteuttamisessa mukana olevia henkilöitä, kuten ohjelmistokehittäjiä ja niiden suunnittelijoita. Tutkimuksen aikana selkeni ymmärrys siitä, miten siirrettävyyttä painottavan serverless-sovellusten suunnitteluun ja toteuttamiseen todellakin voivat vaikuttaa hyvinkin erilaiset ja erilaisista lähtökohdista nousevat asiat.

Mahdollisten ratkaisuvaihtojen sekä tutkimusongelmaan vaikuttavien asioiden erilaisuus ja määrä oli lopulta yllättäväkin ja vaikutti siten, että valintoja sekä rajauksia jouduttiin tekemään tutkimuksen aikana sille määritettyjen tavoitteiden saavuttamiseksi.

9.1 Tutkimuksen lähtökohdat ja tutkimustulokset

Tutkimuksen lähtökohta muodostui kysymyksestä millä tavoin voidaan ratkaista lukittautumisongelma pilviympäristöissä toimivissa serverless-sovelluksissa. Ongelman ratkaisemiseksi etsittiin soveltuvaa arkkitehtuurimallia, jonka avulla voidaan toteuttaa siirrettävä serverless-sovellus. Serverless-sovellukset ajetaan jonkin pilvipalveluntarjoajan ympäristössä. Sovelluksen siirrettävyys tarkoittaa sitä, että sovellus voidaan siirtää yhdestä pilviympäristöstä johonkin toiseen pilviympäristöön. Nämä ympäristöt ovat kuitenkin erilaisia mikä tekee siirrettävyydestä ongelman, kun kyse on ympäristöön voimakkaasti sidoksissa olevasta serverless-sovelluksesta. Tutkimukselle määritettiin kolme tutkimuskysymystä, joiden avulla pyrittiin saavuttamaan sille asetetut tavoitteet:

- Miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys toiseen pilviympäristöön on merkittävä asia?
- Mitä yleisiä haasteita liittyy toiseen pilviympäristöön mahdollisimman vaivattomasti siirrettävissä olevan serverless-sovelluksen suunnitteluun ja toteuttamiseen?
- Miten serverless-sovelluksen arkkitehtuuri voidaan suunnitella ja sovellus toteuttaa, kun halutaan lähestyä monipilviympäristökysymystä sovelluksen siirrettävyyden näkökulmasta?

Saavutetut tutkimustulokset tiivistettynä

Serverless-funktiot ovat lähtökohtaisesti vahvasti sidoksissa pilvipalveluntarjoajan ympäristöön. Standardien puute teki jo eri serverless-alustojen tavoista käsitellä tapahtumia ja suorittaa ohjelmakoodia toisistaan selkeästi erilaisilla keinoilla aiheutti sen, että lukittumisen vaikutukset alkoivat jo niistä alustoista, joilla serverless-funktioita ajetaan. Tutkimuksen kautta löydetty hexagonal-arkkitehtuuri mahdollisti sillä toteutettavan sovelluksen mahdollisimman vaivattoman siirrettävyyden. Arkkitehtuurimallin tapa jakaa sovelluksen toiminnot erilaisiin komponentteihin, joilla oli omat roolinsa, teki sovelluksen komponenteista sellaisia, että niiden siirrettävyys parani. Vain pieni osa komponenteista oli millään tavoin yhteydessä pilviympäristöön ja nekin olivat uudelleenkäytettäviä.

Hyvä siirrettävyys mahdollisti edelleen nopeamman kehitystyön, paremman sovelluksen laadun sekä laadukkaammin toteutettavissa olevan yksikkötestaamisen. Hyvä sovelluksen siirrettävyys tuo mukanaan myös sen, että organisaatio pystyy hyödyntämään laajemmin ja paremmin monipilven mahdollisuuksia. Lisäksi se antaa mahdollisuuden käyttää parhaita eri pilvipalveluntarjoajien palveluita ja vaikuttaa myös monipilven hyödyntämiseen silloin kun halutaan käyttää samanaikaisesti useita eri pilviympäristöjä. McLellanin (2019) mainitseman siirrettävyyteen panostamisen ja kompromissien teon strategian lisäksi Hexagonal-arkkitehtuurimallin ansiosta käytetään myös Tanasan (2019) mainitsemaa strategiaa, jossa siirtokustannuksiin vaikutetaan erottaen sovellusalue ympäristöstä arkkitehtuurin suunnittelulla. Tutkimuksen tulosten merkittävänä puolena on se, että ne tarkastelevat asioita sekä liiketoiminnan että teknisen näkökulman kautta. Esimerkkisovelluksen tavoitteena oli antaa sovelluskehittäjille hyödyllinen työväline, jonka avulla he voivat toteuttaa siirrettäviä serverless-teknologiaa käyttäviä sovelluksia. Tutkimus sisältää paljon esimerkkejä soveltamisen tueksi.

Tutkimuksessa kehitetty menetelmä auttoi toteutusvaihtoehtojen kartoituksessa

Tutkimuksessa tarkasteltiin jopa ristiriitaiselta vaikuttavaa tavoitetta, jossa halutaan käyttää teknologiaa, joka on voimakkaasti sidoksissa palveluntarjoajaan ja lisäksi halutaan käyttää palveluntarjoajan ympäristön mahdollisuuksia mutta pyrkiä silti samaan aikaan suhtautumaan agnostisella tavalla palveluntarjoajaan. Agnostisuuden tavoittelua pidetään myös tarpeettomana ja todetaan että monet rakentavat ja käyttävät tyytyväisinä voimakkaasti pilvinatiiveja sovelluksia. Esimerkiksi Koljonen (2020) tuo esiin näkemyksen, että ainoastaan lainsäädäntö olisi ainoa hyvä peruste toteuttaa pilviagnostinen sovellus ja silloinkin vain, jos viranomaisten määräykset ehdottomasti tätä edellyttävät. Tutkimuksen aikana löydettiin erilaisia näkemyksiä ja lähtökohtia, joista kaikki eivät ole niin jyrkkiä ja ehdottomia kuin edellä mainittu Koljonen (2020) esittää. Tutkimuksen aikana selvisi, että myös käsitettä monipilviympäristö oli mahdollista tarkastella eri näkökulmista. Lisäksi todettiin, että on olemassa eri tapoja käsitellä monipilviympäristön hyödyntämiseen, monipilviympäristöön siirtymiseen ja ympäristössä toimimiseen liittyviä valmiuksia, haasteita sekä mahdollisuuksia.

Niin ensimmäiseen tutkimuskysymykseen ”Miksi serverless-sovelluksen arkkitehtuurin mukautumiskyky ja sovelluksen siirrettävyys toiseen pilviympäristöön on merkittävä asia?” kuin tutkimuskysymykseen numero kaksi ”Mitä yleisiä haasteita liittyy toiseen pilviympäristöön mahdollisimman vaivattomasti siirrettävissä olevan serverless-sovelluksen suunnitteluun ja toteuttamiseen?”

havaittiin tutkimuksen aikana löytyvän vastaukseksi erilaisia näkökulmia ja lähtökohtia. Tutkimuskysymyksiin ja tutkimuksen tavoitteeseen vastaaminen edellytti, että nämä havaitut erilaiset näkökulmat, lähtökohdat sekä niihin liittyvät käsittelytavat ja tarkemmat yksityiskohdat kyettiin tunnistamaan ja luokittelemaan tutkimuksen tavoitteen kannalta soveltuvalla tavalla. Havaintojen luokittelua tarvittiin ohjaamaan tutkimusta eteenpäin, jotta löydettäisiin vastaukset tutkimuskysymyksiin. Tätä varten oli keksittävä menetelmä, jonka avulla pystyttiin jakamaan löydetyt havainnot ensin tarpeisiin ja vaatimuksiin. Menetelmän avulla oli tässäkin vaiheessa mahdollista todeta, miten tarpeet ja vaatimukset voidaan edelleen jakaa kahteen eri lähestymistapaan. Toinen lähestymistavoista oli se, joka vastasi tämän tutkimuksen tavoitteisiin. Nämä kaksi eri lähestymistapaa monipilviympäristön hyödyntämiseen olivat sovelluksen siirrettävyyteen panostaminen ja halu hyödyntää useaa eri pilviympäristöä samaan aikaan. Lähestymistapojen tunnistaminen oli keskeistä, koska näiden lähestymistapojen tarpeet ja vaatimukset erosivat toisistaan.

Lisäksi voitiin todeta, että oli myös olemassa tarpeita ja vaatimuksia, jotka ovat yhteisiä molemmilla lähestymistavoille. Tarpeet ja vaatimukset jaettiin vielä kolmeen eri ryhmään koska havaittiin, että niihin liittyvän kolmen erilaisen näkökulman huomiointi ja tunnistaminen oli keskeistä tutkimuksen tavoitteiden kannalta. Kehitetty menetelmä osoittautui toimivaksi. Kun kolme eri tarve- ja vaatimusryhmää oli muodostettu, huomattiin, että on löydetty keino määrittää perustelut sellaisten valintojen tekemiselle, joiden kautta on mahdollista muodostaa konkreettiset oman sovelluksen tai järjestelmän toteuttamiseen liittyvät tavoitteet. Selkeästi määritetty tavoite, mikä se sitten onkaan, mahdollistaa edelleen ratkaisuvaihtoehtojen etsiminen valintojen kautta.

Tarpeiden ja vaatimusten jaottelussa käytetty jako kolmeen ryhmään toimi yhdistävänä tekijänä näiden kahden eri tutkimuksen vaiheen välillä ja osoittautui olevan edelleen toimiva ratkaisu tutkimuksen toteuttamisessa. Tutkimuksen etenemisen kannalta näiden tavoitteiden määrittämisen tärkeä merkitys oli siinä, että niiden kautta saatiin tutkimuksen seuraavaa vaihetta varten tietoa, jota voitiin käyttää ohjaamaan toteutusvaihtoehtojen kartoittamista. Tarpeisiin, vaatimuksiin ja tavoitteisiin liittyneen kartoituksen tuloksia hyväksi käyttäen oli myös mahdollista tehdä perusteltuja ja tarpeellisia rajauksia, jotta tutkimuksessa pystyttiin keskittymään sille valittuihin tavoitteisiin.

Arkkitehtuurin mukautumiskyvyn ja sovelluksen siirrettävyyden merkitys

Serverless on teknologia, joka on lähtökohtaisesti hyvin kiinteästi sidoksissa pilvipalveluntarjoajan ympäristöön. Serverless-sovellusten suoritus tapahtuu erityisillä serverless-alustoilla, ja ne ovat erilaisia eri pilviympäristöissä. Lisähaasteena on sovellusten itsenäisyys ja tapahtumapohjaisuus, jolloin sovelluksen tulee pystyä kommunikoimaan pilviympäristössä olevien palvelujen kanssa. Serverless-funktioiden tulee myös pystyä kommunikoimaan toisten serverless-funktioiden kanssa. Kommunikoinnin tulisi siirrettävissä olevassa sovelluksessa tapahtua niin, että kaikki osapuolet pystyvät ymmärtämään toisiaan, vaikka ympäristöt, niiden palvelut ja serverless-funktiot sekä näiden välinen kommunikaatio ei ole standardoitua eikä tapahdu millään yhtenäisellä tavalla.

Edelliseen kuvaukseen peilaten on jo nähtävissä, miksi arkkitehtuurin mukautumiskyky ja siirrettävyys on merkittävää. Kun sovellus on arkkitehtuurinsa ansiosta mahdollista siirtää joko sellaisenaan tai pienin muutoksin toiseen erilaiseen pilviympäristöön mukana seuraa monia uusia mahdollisuuksia ja vapauksia. Siirrettävyys sovelluksen tai järjestelmän keskeisenä arkkitehtuurin kautta syntyneenä ominaisuutena vaikuttaa ohjelmistojen kehittämiseen ja ylläpitoon mutta voi myös avata organisaatiotasollakin ja liiketoiminnan näkökulmasta erilaisia mahdollisuuksia.

Ohjelmistojen kehitystyön kannalta siirrettävyys mahdollistaa vaikkapa uuden kiinnostavan toisen pilviympäristön palvelun testaamisen entistä nopeammin, antaa mahdollisuuksia hyödyntää monipilveä eri tavoin tulevaisuudessa vaivattomalla tavalla, pienentää ohjelmakoodin toimimattomuuden riskejä, kun ei ole tarvetta käyttää yhtä laajaa ”palettia” erilaisia kirjastoja tai palveluja yksittäisiin sovelluksen suorittamiin tehtäviin ympäristön vaihtuessa, tekee ohjelmakoodin muuttamisesta nopeampaa ja ennen kaikkea, mahdollistaa pilviympäristöjen parhaiden puolien hyödyntämisen osana serverless-teknologiaa käyttävää sovellusta. Jos siirrettävyys vähentää henkilöresurssien tarvetta, voi myös jäädä aikaa luoda laadukkaampi lopputulos.

Organisaatioiden ja liiketoiminnan näkökulmasta serverless-sovelluksen siirrettävyyden ominaisuuden mukanaan tuomat edut voivat näkyä kustannussäästöinä, silloin kun kehitystyö nopeutuu ja sovelluksen ylläpidettävyys on hyvä sekä sovellus on toteutettu laadukkaasti ja ylläpidettävyys huomioiden. Edut voivat näkyä myös niin, että ne voivat tuoda uusia asiakkaita, lisätä nykyisten asiakkaiden tyytyväisyyttä, voivat antaa erilaisia ennakoimattomiakin mahdollisuuksia hyödyntää monipilveä eri tavoin tulevaisuudessa vaivattomalla tavalla, kehittää uusia tuotteita ja voivat

kasvattaa liiketoimintaa. Sovellusten kehitystyö on tyypillisesti hidasta, kallista ja vaativaa. Sellaisen sovelluksen tai järjestelmän laajamittaisesta muuttamisesta ja siirtämisestä puhumattakaan, joka pitäisi muokata tukemaan jotain erilaista ympäristöä.

Sovelluksen siirrettävyys ja sen käyttämän arkkitehtuurimallin mukautumiskyky voi tuoda mukanaan ainutlaatuista joustavuutta, kun jokin toteuttamisen arvoinen idea tai tarve syntyy myöhemmin, joka ei ole ollut osana organisaation aiempia suunnitelmia silloin kun siirrettävyys merkitsee, että sovelluksessa on uudelleen käytettäviä ja hyvin suunniteltuja komponentteja. Pilvipalveluntarjoajat ja muut tahot kehittävät uusia palveluita ja tuotteita kaiken aikaa, joten ehkä vaikkapa vuoden päästä tulee tarjolle jokin palvelu, joka aiheuttaa voimakasta kiinnostusta.

Serverless-teknologian luonne

Tutkimuksen aikana tutustuttiin serverless-teknologiaan ja niihin näkemyksiin, joita liittyy lukittumiseen pilviympäristöissä ja niiden sisältämiin erilaisiin palveluihin. Serverless-teknologia synnyttää ensimmäisenä voimakkaan mielikuvan, että sillä tehdyt sovellukset ovat luonnollisesti hyvin sidottuja pilvipalveluntarjoajan ympäristöön. Onhan luotu alusta, jonka keskeinen tavoite on yksinkertaistaa taustalla vaikuttavan infrastruktuurin osuutta niin, ettei kehittäjien tarvitse huolehtia mitä palvelimella oikeastaan tapahtuu, kun sovellusta ajetaan alustalla. Taustalla olevan ympäristön hallinta, kehittäminen ja ylläpitovastuu tästä aiheutuvine ylläpitokustannuksineen ja vastuineen on pilvipalveluntarjoajan osuus.

Moni haluaa pyrkiä pilviagnostisuuteen, jolloin tilanne, jossa sovellus ei ole siirrettävissä on ongelma. Ongelmaan ei kuitenkaan voida antaa yhtä ainoaa jokaiseen tilanteeseen ja organisaatioon soveltuvaa ratkaisua. Myös tarve sovelluksen kyvylle mukautua ja olla siirrettävä vaihtelee. Osalle lukittuminen ei ole ongelma lainkaan ja osa kokee, että on kannattavaa voimakkaasti panostaa siirrettävyyteen. Erilaiset abstraktiokerroksiin liittyvät lukemattomat pyrkimykset ja niiden ansiosta aikaansaadut lopputulokset, jotka eivät kosketa vain serverless-teknologiaa, ovat kuitenkin alan todellisuutta, ja tämä näkyi tutkimuksen aikana eteen tulleiden vaihtoehtojen suurena kirjona. Artikkelissaan Hohpe (2022) mainitseekin, miten on käytetty suuriakin summia erilaisten abstraktiokerrosten luomiseen, joiden tarkoituksena on ollut vähentää lukittumista pilvialustoihin.

Siirrettävyys ei ole myöskään ainoa keino, jota voidaan käyttää pyrkimyksissä lukittumisriskin pienentämiseen tai sen poistamisessa. Se kannattaa myös tiedostaa, että lukittuminen on laajempi kysymys eikä liity vain serverless-arkkitehtuuriin ja ottaa huomioon, minkä verran siirrettävyyteen panostaminen veisi erilaisia resursseja. Siirrettävyys koskettaa koko organisaatiota, joten kysymyksiä kannattaa tarkastella osana organisaation yleistä strategiaa, monipilvistrategian ja teknisten kysymysten lisäksi. Tutkimuksen aikana mieleen heräsi myös ajatus, että siirrettävyyteen panostamisessa ensisijainen syy ei välttämättä tarvitsekaan olla lukittumiseen liittyvä pelko tai huoli: Entä jos lähtökohdaksi ja tavoitteeksi otetaan yksinkertaisesti erilaisten mahdollisuuksien toteuttamisessa auttavan keinon liittämistä osaksi sovellusta tulevaisuutta silmällä pitäen? Tämä on mielestäni aivan yhtä hyvä peruste panostaa sovelluksen siirrettävyyteen kuin muut aiemmin esitetyt perusteet sekä on perusteltu valinta liiketoiminnan näkökulmastakin.

Siirrettävän sovelluksen suunnitteluun ja toteuttamiseen liittyvät haasteet

Aiemmin pohdintaluvussa kuvattiin jo, miten serverless-sovellukset ovat kiinteä osa pilvipalveluntarjoajien ympäristöjä: Miten niiden suoritus tapahtuu tässä ympäristössä, miten erilaiset pilviympäristöt ja niiden serverless-alustat vaikuttavat siirrettävyyteen sekä miten serverless-sovellusten itsenäisyys ja tapahtumapohjaisuus liittyy siirrettävyyteen. Nämä asiat vaikuttavat suoraan ja voimakkaasti suunnitteluun ja toteuttamiseen liittyviin haasteisiin. Tarve oli löytää toimiva ratkaisu pilviympäristöjen ja niiden palveluiden erilaisuuden mukanaan tuomiin haasteisiin, jotka syntyvät serverless-alustojen erilaisiin tapoihin käsitellä ja muodostaa tapahtumia eri palvelujen API-rajapinnoissa. Koska yhteistä standardia ei ole olemassa, haasteena on toimiva kommunikointi serverless-funktioiden ja palvelujen välillä sekä funktioiden kesken silloin kun sovellus joutuu toiseen pilviympäristöön, joka ei ole samanlainen.

Serverless-sovelluksille on tyypillistä ja käytännössä väistämätöntäkin, että palveluntarjoajan tuotevalikoimasta käytetään monia erilaisia valmiita sovelluksia eli palveluita, joilla on jatkuvasti käytössä oleva ja myös käytännössä keskeinen osa omaa sovellusta niin kauan kuin niitä käytetään. Aivan samoin kuin pilven infrastruktuuri, nämä palvelut ovat toisen osapuolen tekemiä ja vastuulla olevia valmiita ratkaisuja, joten niiden käyttäjien vaikutusmahdollisuudet ovat tästä näkökulmasta olemattomat mitä tulee siihen millaisia palveluita ne ovat ja voiko niiden toimintaa jotenkin tarvittaessa muuttaa. Edut ovat toki selkeät samaan aikaan, nopeuttavathan valmiit ratkaisut

kehitystyötä eikä samanlaisia vaikutuksia synny, kuin jos samat toteutukset päätettäisiin tehdä itse ja vielä ylläpitovastuukin tulisi omille harteille.

Sovelluksen lukittuneeksi muuttuminen johonkin ei kuitenkaan liity vain pilvipalveluntarjoajaan ja pilviympäristössä olevien resursseihin. Sovelluksen käyttämä resurssi voi olla mikä tahansa verkon ylitse hyödynnettävä asia kuten tietokanta, tekoälysovellus, prosessointiteho tai tiedostoille varattu säilytystila. Jonkinasteiset lukittumiseen liittyvät vaikutukset ovat aina läsnä ja osa ICT-alan luonnetta, sovelluksen toteuttaminen ei onnistu ilman ohjelmointikieltä ja jokin ympäristö on valittava, jota käyttäen sovellus toteutetaan. Pilvipalveluntarjoajien tapauksessa vaikutukset ovat kuitenkin suuremmat, koska vaikutusmahdollisuudet ovat pienemmät helppouden ja nopeuden kustannuksella. Lukittumiseen liittyvät kysymykset eivät siten kosketa vain serverless-teknologiaa, koska pilviympäristöihin voi toteuttaa sovelluksia käyttäen eri teknologioita. Serverless-teknologiaa käyttävä sovellus on pilvinatiivi ratkaisu eli hyvin riippuvainen pilviympäristöstä koska se hyödyntää pilvialustan vahvuuksia. Siirrettävyyttä painotettaessa halutaan taas olla pilviagnostisia, ja olla sujuvasti siirrettävissä olevia alustasta toiseen. Mihin halutaan vetää raja pilviagnostisuudessa, on myös huomionarvoinen ja hyvä kysymys. Joku näkee pilviagnostisuuden tavoitteen hyvin jyrkkänä rajoitteena, ettei valmiita komponentteja käytettäisi lainkaan. Tällöin menetettäisiin pilvipalveluntarjoajien ympäristöjen monet edut.

Sovelluksen suunnittelun ja toteuttamisen haasteisiin vaikuttaa myös organisaatiossa olevat tarpeet ja vaatimukset, kuten kehittäjien riittävä osaaminen ja organisaation valmiudet sekä valittu lähestymistapa, jonka kautta monipilviympäristökysymystä tarkastellaan. Tarpeiden ja vaatimusten tunnistaminen sekä valintojen tekeminen voi olla myös haasteellista koska ne vaikuttavat koko ohjelmistokehitysprosessiin. Tämän organisaatiokohtaisen näkökulman tarkempi tarkastelu ei kuitenkaan kuulunut tähän tutkimukseen, jossa ei ole mukana tapaustutkimusta, jota käsiteltäisiin.

Monipilviympäristön hyödyntämiseen havaittiin liittyvän kaksi erilaista lähestymistapaa, joista toinen painottaa sovelluksen siirrettävyyttä ja toinen hyödyntää samanaikaisesti useaa eri pilviympäristöä eli sovellus tavalla tai toisella toimii usean pilvipalveluntarjoajan ympäristössä samaan aikaan. Koska lähestymistavat eroavat näin paljon toisistaan ne myös vaikuttavat eri tavoin mitä tulee suunnittelun ja toteutuksen haasteisiin. Riippuen sovelluksen arkkitehtuurista ja toteutuksesta, näiden lähestymistapojen yhteiset piirteet saattavat vaikuttaa niin, että niissä on

molempien lähestymistapojen hyödynnettävissä olevaa ohjelmakoodia. Samanaikaisesti tapahtuvaan usean eri pilviympäristön hyödyntämiseen on useita tapoja, alkaen sovelluksen kopion sijoittamisesta kokonaisuudessaan toiseen pilviympäristöön tai vain jonkin pienen toiminnon sijoittamisesta johonkin toiseen ympäristöön. Jos siirrettävyyttä painottanut sovellus haluttaisiin kokonaan siirtää toiseen pilviympäristöön ja edelleen käyttää samaa sovellusta molemmissa pilviympäristöissä, esimerkiksi eri asiakkaiden vuoksi, tämä operaatio ei välttämättä vaadi kovin suuria muutoksia ohjelmakoodiin.

Siirrettävän sovelluksen arkkitehtuurin suunnittelu

Toteutusvaihtoehtojen kartoittamisvaiheessa keskityttiin etsimään konkreettisia sovelluksen suunnitteluun ja toteuttamisvaiheeseen liittyviä vastauksia koskien kolmatta tutkimuskysymystä ”Miten serverless-sovelluksen arkkitehtuuri voidaan suunnitella ja sovellus toteuttaa, kun halutaan lähestyä monipilviympäristökysymystä sovelluksen siirrettävyyden näkökulmasta?”. Samalla myös ensimmäinen ja toinen tutkimuskysymys tarkentuivat koska niin yksityiskohtaiset teknologiaan liittyvät kysymykset kuin pilvialustatkin ovat osa mukautumiskyvyn ja siirrettävyyden merkittävyyden tarkastelua ja siirrettävyyteen liittyvät haasteet olivat aiemmin käsiteltyjen kysymysten lisäksi luonnollisesti myös teknisiä haasteita.

Siirrettävyyttä painottavaksi ja tutkimuskysymyksen ratkaisevaksi arkkitehtuurimalliksi valittiin Hexagonal-arkkitehtuuri, joka vaikutti soveltuvan hyvin tapahtumapohjaiseen sovellukseen ja ratkaisevan siirrettävyyden ongelman. Arkkitehtuurimalliin tutustuminen vahvisti, että sen tapa jakaa sovellus portteihin, adaptereihin sekä ydinosioon toimisi ratkaisuna serverless-funktioiden ja palveluiden väliseen kommunikaation sekä funktioiden keskenään käymään kommunikaatioon silloin kun tarvitaan kykyä mukautua erilaiseen ympäristöön.

Siirrettävän sovelluksen toteuttaminen

Jotta tutkimusongelman luonteesta saatiin selkeämpi kuva, toteutusvaihtoehtojen kartoitusvaihe alkoi yksityiskohtaisemmalla perehtymisellä serverless-teknologiaan ja valittujen pilvipalveluntarjoajien ympäristöihin. Näin oli mahdollista saavuttaa ymmärrys, miten serverless-teknologia itsessään vaikuttaa siirrettävyyteen sekä miten pilvipalveluntarjoajien erilaiset serverless-alustat oikein toimivat. Edellisten tietojen tukemana pystyttiin etsimään perusteltuja vastauksia, jotka ovat

sellaisina ratkaisuja, jotka huomioivat nämä valitun teknologiaan ja erilaisten ympäristöjen mukanaan tuomat haasteet ja pystyvät antamaan vastauksen tutkimusongelmaan.

Siirrettävissä olevan sovelluksen toteuttamisvaihtoehtojen kartoitusvaiheessa käytiin läpi erilaisia vaihtoehtoja, jotka liittyvät sovelluksen arkkitehtuurin suunnitteluun ja sovelluksen toteuttamiseen. Tässä vaiheessa huomattiin, että vaihtoehtoja voi toisaalta olla paljon, mutta ei ole itsestään selvää mikä on paras vaihtoehto. Vaihtoehdot sisältävät hyvien puolien lisäksi riskejä, haasteita ja vaativat lisäksi esimerkiksi riittävää osaamista niiden väliseen vertailuun ja sovelluksen toteuttamiseen, mikäli päädytään niiden käyttämiseen. Yksi hyvä vaihtoehto voi olla käyttää johonkin yksittäiseen tehtävään soveltuvaa kolmannen osapuolen ratkaisua, jos ja kun halutaan lisätä omia vaikutusmahdollisuuksia sekä vähentää sitoutumisesta aiheutuvia riskejä liittyen pilvipalveluntarjoajan ympäristöön ja sen palveluihin. Tässä tapauksessa otettaisiin kuitenkin vastuuta huomattavasti enemmän itse, alkaen tuotteen valinnasta, siihen liittyvästä osaamistarpeesta ja ylläpitovastuusta. Samalla menetettäisiin valmiiden palvelujen hyödyntämisestä saatavat edut ja resurssejakin kuluu eri tavoin. Kartoituksen tuloksena päädyttiin pitämään parhaana vaihtoehtona sitä, joka tuo mukanaan mahdollisimman vähän rajoituksia ja aiheuttaisi mahdollisimman pientä sovelluksen suunnittelu- ja toteutusvaiheissa syntyvää riippuvuutta (lukittumista) yhteenkään pilviympäristön palveluun, muuhun sovellustuotteeseen, johonkin ohjelmakirjastoon tai niitä tarjoavaan osapuoleen.

Siirrettävyyden ongelman ratkaiseva esimerkkisovellus

Hexagonal-arkkitehtuuri osoittautui tämän tutkimuksen tavoitteiden kannalta hyväksi valinnaksi, joka vastaa sekä sovelluksen arkkitehtuuriin liittyvään tavoitteeseen ja antoi esimerkkisovelluksen toteuttamiselle hyvät lähtökohdat. Esimerkkisovelluksen laajuudessa valittu arkkitehtuurimalli ei myöskään osoittautunut millään tavoin liian työlääksi suunnitella tai toteuttaa, koska sen avulla laadittiin yleiskäyttöisiä erillisiä komponentteja, jotka liittyivät kunkin pilvipalveluntarjoajan palveluun. Arkkitehtuurimallin ideana on se, että kerran laadittu komponentti on kaikkien niiden sovelluksen toimintojen käytettävissä, jotka haluavat käyttää samaa pilvipalvelussa olevaa palvelua. Tämä toimintatapa mahdollistaa samalla sovelluksen jatkokehittämisen ja ylläpidon helppouden.

Esimerkkisovelluksen suunnittelu sekä toteutus osoittautui toimivaksi ratkaisuksi siirrettävyyteen, koska sovelluksen jokaisella komponentilla oli oma määritelty rooli ja tehtävä, jonka se suorittaa. Sovelluksen komponentit myös kommunikoivat yhteisesti sovittujen rajapintojen kautta

keskenään sekä toistensa kautta. Näin sovelluksen komponenteista tuli samaan aikaan uudelleenkäytettäviä, siirrettäviä sekä myös helpommin yksikkötestattavia. Mikäli pilvipalveluntarjoajan ympäristössä tapahtuvien muutosten vuoksi olisi tarvetta muuttaa nykyistä toteutusta, jokaisen jollain tavoin muuttuneen pilviympäristön palvelun kohdalla tämä muutos tarvitsee tehdä vain yhteen itsenäiseen komponenttiin. Toteutusvaihtoehtojen tarkasteluvaiheessa todettiin, ettei jonkin kolmannen osapuolen työkalun käyttäminen ole välttämättä tarpeen, vaan tämän tutkimuksen tavoitteen mukaiseen lopputulokseen voidaan päästä yksinkertaisesti käyttämällä soveltuvaa arkkitehtuuria ilman, että on tarpeen käyttää esimerkiksi ohjelmistokehystä osana sovellusta.

Vaikka toteutetussa esimerkkisovelluksessa oli toki kyseessä pienimuotoinen ratkaisu, sen perusteella voidaan todeta, että serverless-sovellukselle mukautumiskykyinen ja siirrettävyyttä painottavan arkkitehtuurin suunnittelu on täysin mahdollista sekä toimivan sovelluksen toteuttaminen käyttäen suunniteltua arkkitehtuuria. Sovelluksen toteuttamisessa tarvitaan myös huolellisuutta, suunnitelmallisuutta ja hyviä ohjelmointikäytäntöjä. Näin on mahdollista saada aikaan ratkaisu, joka on ylläpidettävissä jatkossakin ilman liiallista työmäärää ja sovellus on toteutettu niin, että se on tarvittaessa siirrettävissä toiseen pilviympäristöön melko vaivattomasti eikä prosessista automaattisesti aiheudu suuria siirtokustannuksia, koska suurinta osaa sovelluksesta ei tarvitse muuttaa. Toteutetun esimerkkisovelluksen ydin on itsenäinen, eikä siihen vaikuta lainkaan millaiseen pilviympäristöön se mahdollisesti liitetään, koska ytimen komponenteille riittää, että ne saavat kaikki tarvitsemansa tiedot oman tehtävänsä suorittamiseksi. Kun sovelluksen ydin sisältää suurimman osan sovelluksen logiikasta ja ne komponentit, joita tarvitaan pilviagnostisen kerroksen toteuttamiseksi ovat erillisiä ja huolellisesti suunniteltuja sekä helposti ylläpidettäviä voidaan todeta että tutkimuksen tuloksessa saavutettiin onnistuneesti sille asetettu tavoite.

Pilviympäristöjen vahvuudet ja parhaat puolet ovat edelleen täysin serverless-sovelluksen käytettävissä eli sovellus pystyy käyttämään useita erityyppisiä kahden keskenään erilaisen pilviympäristön palveluita osana toimintaansa. Mikäli serverless-teknologiaa käyttävä sovellus haluttaisiin siirtää johonkin toiseen pilviympäristöön, jota koodissa ei vielä tueta, ohjelmakoodin osalta riittää, että muokataan adapter- ja port -tehtäviä hoitavia komponentteja ottamaan huomioon uuden ympäristön. Lisäksi tarvittaessa muokataan sovelluksessa olevia tiedostoja, jotka liittyvät pilviympäristöjen konfiguraatioihin ja kuten esimerkkisovelluksen tapauksessa, sovelluksen eri komponenttien tarvitsemiin apufunktioihin. Näillä apufunktioilla vältetään samojen ohjelmakoodien toistamista ja

varmistetaan omalta osaltaan samalla, että sovellus säilyy siirrettävänä. Mikäli halutaan aloittaa käyttämään jotain uutta käytössä olevan pilvipalveluntarjoajan palvelua, voi palvelua varten luoda tarvittavat yleiskäyttöiset adapter- ja port -komponentit ja muut tarvittavat ohjelmakoodit osaksi sovellusta.

9.2 Tulosten merkitys ja soveltamismahdollisuudet

Tutkimus ja sen lopputulos muodostavat yhdessä kokonaisuuden, joka pystyy antamaan monelle organisaatiolle sekä eri rooleissa toimiville henkilöille ohjelmistokehittäjien lisäksi vastauksia serverless-sovellusten lukittumiseen ja siirrettävyyteen liittyviin kysymyksiin. Moni on esittänyt näitä kysymyksiä löytämättä selkeää ratkaisua ongelmaan. Pilvipalveluntarjoajien palveluihin ja ympäristöihin perehtyminen vie aikaa ja vaatii myös osaamista. Yksinkertaisilta kuulostavat kysymykset mihin tulisi kiinnittää huomiota, eli kysymykset miksi ja miten eivät ole niin yksinkertaisia kuin miltä ne kuulostavat, kun siirrettävyyttä tutkii ja asettaa tavoitteeksi siirrettävän serverless-sovelluksen toteuttamisen.

Tutkimukseen valittua lähestymistapaa ei vaikuta olevan tutkittu missään opinnäytetyössä aiemmin samalla tavoin eikä samassa laajuudessa. Tutkimuksessa muodostettiin ratkaisuvaihtoehto siirrettävyysongelmaan, joka on sovellettavissa myös muihin pilvipalveluntarjoajien ympäristöihin, joissa on serverless-alusta. Tutkimuksen aikana kehitetyn menetelmän avulla tarpeiden, vaatimusten sekä tavoitteiden kartoittamiseen liittyvissä luvuissa koostettiin näiden aihepiirien alle erilaisia monipilviympäristön hyödyntämiseen ja serverless-teknologiaan liittyviä asioita. Edellä kuvatun kartoitusosuuden tuloksien arvioitiin voivan olla hyötyä muillekin, koska niissä kuvatut asiat ovat yleisesti organisaatioista ilmeneviä tarpeita, vaatimuksia sekä tavoitteita. Aihepiireissä mainittua asioita on mahdollista käyttää esimerkiksi silloin kun valitaan omia monipilviympäristön hyödyntämiseen liittyviä tavoitteita.

Hexagonal-arkkitehtuurimallia käyttävän siirrettävän serverless-sovelluksen arkkitehtuurin suunnittelun ja sovelluksen toteutuksen kuvaaminen on toteutettu niin, että ne yhdessä antavat työvälineen kaltaisen hyödynnettävissä olevan selkeän pohjan, jonka avulla voidaan toteuttaa siirrettäviä serverless-teknologiaa käyttäviä sovelluksia. Esitetyt ratkaisut ovat sovellettavissa silloin kun halutaan parantaa erilaisten serverless-sovellusten siirrettävyyttä arkkitehtuurin suunnittelun

keinoin. Lisäksi tutkimus esittelee muitakin mahdollisuuksia, jotka saattavat soveltua valintoina paremmin muunlaisiin sovelluksiin kuin tutkimuksen esimerkkisovellus.

Myös liiketoiminta ohjaa usein sovellusten kehitystyötä, joten ohjelmistojen kehittäjän näkökulman lisäksi pilviagnostisuuden tavoitteluun liittyviä erilaisia ja huomionarvoisia näkökulmia tulee organisaatioista, asiakkailta, toimialasta ja lainsäädännöstäkin. Maailma muuttuu ja tarpeet muuttuvat kaiken aikaa, ja vaikka moni onkin ollut ja on sitä mieltä, ettei pilviagnostisuus ole lainkaan tarpeellista ja lukittumiskysymys ei ole edes tarpeellinen huolenaihe, silti moni kokee edelleen asian myös merkitykselliseksi. Pilvinatiivit valmiit palvelut ovat yksinkertaisesti käteviä, eikä niitä kannata jättää hyödyntämättä. Tätä tarkoitusta vartenhan pilviympäristöt on nimenomaan kehitetty. Mahdollisuus toimia tarvittaessa pilviagnostisesti ei ainakaan sulje yllättäviäkin uusia vaihtoehtoja pois monipilven hyödyntämismahdollisuuksien joukosta tulevaisuudessa ja voi parhaimmillaan vaikkapa synnyttää uudenlaisia tuotteita.

Mikäli pilviagnostinen näkökulma tarkoittaa mahdollisuutta toteuttaa sovellus suhteellisen pienin lisäkustannuksin ja niin, ettei se aiheuta välttämättä mitään suuria lisäkustannuksia tai tuo mukanaan muita haasteita toteutusvaiheeseen tai vaikuta ei-toivotulla tavalla ylläpidettävyyteen, voi myös kysyä miksi jättää se mahdollisuus käyttämättä, että tarvittaessa voidaan siirtää koko sovellus suhteellisen helposti johonkin toiseen pilviympäristöön? Toki tämä edellyttää, että kaikki vastaavat palvelut löytyvät toisesta ympäristöstä, kuin ne mitä sovelluksen toiminnot silläkin hetkellä käyttävät. Kuten Swail (2020) mainitsi, vaikkapa asiakas voisi olla se taho, joka ilmoittaa, että tulee käyttää tiettyä palveluntarjoajaa. Entä jos yritys, joka on kehittänyt sovelluksen, esimerkki käyttäen AWS-pilvipalveluntarjoajan ympäristöä tulee osaksi toista yritystä, joka toivoisi, että käytössä onkin vaikkapa Azuren tai Googlen pilvi? Jatketaanko mahdollisesti toimintaa, niin että säilytetään kaikki ennallaan, mikä sitoo esimerkiksi resursseja eri tavoin, kuin jos sovellus olisi siirrettävissä suhteellisen vaivattomasti ja pienin kustannuksin.

9.3 Aiemmat tutkimukset ja tutkimuksen tulosten suhde niihin

Tutkimuksen tavoitteiden mukaista sellaista serverless-sovelluksen arkkitehtuurin suunnitteluun sekä sovelluksen toteuttamiseen liittyvää opinnäytetyötä, jossa tavoitteena olisi ollut selvittää miten voidaan luoda toiseen pilvipalveluntarjoajan ympäristöön siirrettävissä oleva sovellus, ei ilmeisesti ole tehty. Niitä opinnäytetöitä, joiden sisältö on liittynyt serverless-teknologiaan, ja joissa on

todettu lukittumisen vaikuttavan tutkimuksen aiheeseen, on useitakin, mutta niissä ei ole pyritty ratkaisemaan lukittumisen ongelmaa.

Pro gradu -työssään ”Implementation of serverless computing with DSL framework in SWIFT” Aleksei Sapitskii keskittyi ratkaisemaan lukittumisongelmaa käsittelemällä sovelluksen käyttöönottovaiheessa käytettäviin IAC-työkaluihin liittyviä eri pilvipalveluntarjoajien infrastruktuuria kuvaavaa koodia. Tavoitteena Sapitskiilla oli selvittää, miten voitaisiin luoda sellainen tapa hoitaa kuvaustyö tavalla, joka vähentäisi ohjelmakoodin ja IAC-työkalujen käyttöön liittyvien konfiguraatiokuvausten välisiä synkronointiongelmia. Lisäksi Sapitskii halusi löytää keinoja, joilla esimerkiksi parannettaisiin laajemminkin yhteensopivuutta sovelluksen ja konfiguraatioita sisältävän kuvaustiedoston välillä. Sapitskiin tekemä tutkimus käsitteli lukittumiskysymystä ja siirrettävyyttä sovelluksen käyttöönottovaiheessa IAC-työkalujen avulla. Vaikka molemmat tutkimukset pyrkivät ratkaisemaan lukittumisongelman, Sapitskiin tutkimus ja sen tulokset eroavat tämän tutkimuksen tavoitteista ja lopputuloksesta, jossa keskityttiin ratkaisemaan lukittumisongelma serverless-sovelluksen siirrettävyyden kautta.

Daniel Hagg tutki pro gradu -työssään ”Serverless Applikationen in Multi-Cloud-Umgebungen: Architektur und Design Eventbasierter Kommunikation” miten serverless-funktiot voivat kommunikoida keskenään, kun toinen funktio on eri pilvipalveluntarjoajan ympäristössä. Haggin tutkimuksessa lähestyttiin monipilviympäristöä lähtökohdasta, jossa käytössä oli useampia eri pilvipalveluntarjoajien ympäristöjä saman sovelluksen käytössä. Yhteistä tälle tutkimukselle ja Haggin tutkimukselle oli, miten niissä tutkittiin keinoja, miten monipilviympäristössä toimivat serverless-funk-tiot voivat kommunikoida keskenään. Hagg ei kuitenkaan tutkinut sovelluksen siirrettävyyteen vaikuttavia kysymyksiä laajemmin eikä siirrettävän sovelluksen arkkitehtuurin suunnittelua tai toteuttamista vaan keskittyi sovelluksiin, jotka hyödyntävät samanaikaisesti useita eri pilvipalveluntarjoajien ympäristöjä.

Tolunay Yükselin vuonna 2022 tekemä pro gradu -tutkielma "Standards-based Modeling and Generation of Platform-specific Function-as-a-Service Deployment Packages" liittyy serverless-funktioiden työnkulkuihin. Yükselin tutkimuksen tulosten tärkeimmät asiat olivat tarjota osittain palveluntarjoaja-agnostinen työnkulkumalli, joka auttaa erityisesti niitä kehittäjiä, joille serverless-teknologian hyödyntäminen eri pilviympäristöissä ei ole tuttua. Lisäksi mallista voitiin muuttaa

pilvipalveluntarjoajakohtainen FaaS-sovelluksen käyttöönottopakkaus. Tutkimuksen tulosten muodostamiseksi hän tutki mahdollisuuksia mallintaa eri pilviympäristöjen työnkulkuja Business Process Modeling Notation (BPMN) -standardin avulla niin, että BPMN-mallinnuksen pohjalta olisi mahdollista generoida FaaS-funktioita moneen eri pilviympäristöön. Tutkimuksessa keskityttiin generoimaan FaaS-funktioiden ohjelmakoodia template-pohjien perusteella. Yükselin tutkimuksen tavoite oli helpottaa sovelluskehittäjien työtä tarjoamalla yhtenäinen tapa ja malli työnkulkujen luomiseen eri pilviympäristöihin ja sekä generoida ohjelmakoodia. Yükselin työ siis ratkaisi tutkimukselle valituista lähtökohdista osittain lukittumiseen liittyvän ongelman ja liittyi vahvasti useiden serverless-funktioiden työnkulkuihin, joten tutkimus ei käsitellyt lukittumisongelmaa samalla tavoin kuin tämä tutkimus.

9.4 Tulosten luotettavuus, kehittämisehdotukset ja jatkotutkimusmahdollisuudet

Kuten laadulliseen tutkimukseen kuuluu (Kananen 2015, 29), tutkimuksessa luotiin tutkittavaan ongelmaan liittyvä ratkaisu sekä saavutettiin myös ymmärrys millaiset asiat vaikuttavat tutkimusongelmaan. Ymmärtämällä riittävän syvällisesti millainen teknologia serverless on, millaisia erilaiset pilviympäristöt sekä niiden palvelut ovat ja miten tapahtumapohjainen kommunikaatio niissä tapahtuu, mahdollisti, että löydettiin ratkaisu siirrettävissä olevan serverless-sovelluksen arkkitehtuurin suunnittelemiseksi ja sovelluksen toteuttamiseksi.

Tämän tutkimuksen tulokset keskittyivät serverless-teknologiaa käyttävän sovelluksen arkkitehtuurin suunnitteluun ja toteuttamiseen. Sovelluksen siirrettävyyteen voivat vaikuttaa myös kysymykset, joita tämä tutkimus ei ole käsitellyt. Siirrettävyyden ominaisuuden vaikutukset, jotka voivat olla positiivisia tai negatiivisia, eivät liity vain sovelluksen arkkitehtuuriin vaan vaikkapa datan siirrettävyyteen. Jos sovellus on julkaistu ja sitä on käytetty, syntyy myös yleensä jonnekin tallennettua dataa. Yksi suuri kysymys siirrettävyydessä onkin silloin data, jota on kerääntynyt johonkin pilviympäristön palveluun, kuten esimerkiksi tietokantaan. Sovellus voi myös käyttää toiminnassaan pilviympäristön ulkopuolisia osapuolia, joita käyttämällä syntyy kuitenkin dataa, jota säilytetään pilvipalveluntarjoajan ympäristössä. Data on todennäköisesti keskeinen osa esimerkiksi liiketoimintaa, ja jos siirrettävyys haluttaisiin laajentaa koskemaan myös datan siirrettävyyttä, ollaan jo aivan uuden kysymyksen äärellä, johon tämä tutkimus ei anna vastauksia.

Toteutusvaihtoehtojen kartoittamisessa ja esimerkkisovelluksessa käytettiin loppujen lopuksi vain pientä osaa erityyppisiä palveluita. On mahdollista, että eri pilvipalveluntarjoajien palveluissa on sellaisia palveluita, joiden välillä tapahtuva siirrettävyys on vaikeampi kysymys ja kenties perusteltavissa kannattaako joka tilanteessa pyrkiä toteuttamaan pilviagnostinen ratkaisu. Seuraava on esimerkki edellä kuvatun kaltaisesta vaikeammasta kysymyksestä, jota tutkimuksessa ei ole käsitelty: Lambda-alustalla serverless-funktiot voivat vastaanottaa samaan aikaan useita event-tapahtumia eli funktiolla voi olla useita triggereitä. Hohpe (2022) mainitsee, etteivät kaikki palvelut (kuten AWS-pilviympäristön Step Functions), osaa kuitenkaan käsitellä hyvin useita tapahtumia esimerkiksi silloin kun kahden tapahtuman välillä on riippuvuus toisistaan. Toisaalta Hohpe (2022) mainitsee, että edellä mainitun ongelman kiertämisessä voi kuitenkin käyttää muita saman pilviympäristön palveluja.

Tutkimuksen tekijälle opinnäytetyön aiheeksi valittu serverless-teknologia, monipilviympäristöjen maailma, tapahtumapohjaisen serverless-sovelluksen arkkitehtuurin suunnittelu ja toteuttaminen sekä valitut pilviympäristöt olivat myös ennen tutkimuksen tekoa vain jonkin verran ennalta tuttuja. On siten mahdollista, että tutkimuksen tekijän kokemuksen ja osaamisen taso on vaikuttanut tutkimuksen tuloksiin ja tehtyihin johtopäätöksiin siten, ettei kaikkia siirrettävyyteen liittyviä esittelyn ja huomioon ottamisen arvoisia yksityiskohtia ole välttämättä tutkimuksessa käsitelty.

Esimerkkitoteutuksessa ei käytetty ohjelmakoodissa tapahtuman kuvaamisessa CloudEvents-standardia. Sovellusta kannattaisikin siten kehittää edelleen huomioiden myös tämä standardi. Tutkimuksen tavoitteen kannalta ja esimerkkisovelluksen kyvyllä ratkaista siirrettävyyden haaste ei ollut merkitystä sillä, käytettiinkö CloudEvents -standardia. Tämä vahvisti tutkimuksen tekijälle standardin luonteen ja tarkoituksen, jossa sen on tarkoitus helpottaa sovellusten kehittäjien työtä ja vaikuttaa auttavana standardina datan välittämisessä eri osapuolten välillä, ei toimia pakottavana menetelmänä.

Jatkotutkimusmahdollisuuksista

Hyödyllinen sekä mielenkiintoinen jatkotutkimuksen aihe voisi olla siirrettävän hexagonal-arkkitehtuuria käyttävän serverless-sovelluksen testaamiseen liittyvät kysymykset. Tanasa (2019) kertoo, että yksikkötestit, jotka ovat kytkettyjä tiettyyn pilvipalveluntarjoajaan ovat kuitenkin haastavia yhdistää toisiin ympäristöihin ja jatkaa täydentäen, että kuitenkin hyvää arkkitehtuurimallia

käyttämällä testattavuuden ongelmasta voi päästä eroon. Edellinen kuvaus nosti esiin ajatuksen myös vertailevaan näkökulmaan keskittyvästä tutkimuksesta, jossa tutkittaisiin, miten serverless-sovelluksen testattavuus muuttuu, kun se onkin toteutettu käyttäen siirrettävyyden mahdollistavaa hexagonal-arkkitehtuuria ja selvittäisi millä tavoin muutos vaikuttaa siihen, miten yksikkötestien tulisi kommunikoida pilviympäristön palvelujen ja toisten serverless-funktioiden kanssa. Tutkimuksen arvoinen voisi olla myös se, että tutkitaan, miten integraatiotestaaminen voidaan toteuttaa, kun sovellus käyttää siirrettävyyttä painottavaa arkkitehtuuria ja on myös toteutettu siirrettävyyden näkökulmasta. Miten tällöin kommunikointi eri osapuolten välillä tapahtuu?

Myös vertaileva tapaustutkimus voisi olla yksi jatkotutkimuksen aihe, jossa tutkittaisiin, nopeutuuko sovelluksen kehitystyö vai mitä tapahtuu, kun ns. perinteisen serverless-sovelluksen sijaan toteutetaankin siirrettävän ominaisuuden sisältävä sovellus. Vertaileva tapaustutkimus voisi olla laadittu myös liiketoiminnan näkökulmaa painottaen ja siinä tutkittaisiin esimerkiksi, saavutettiinko kustannussäästöjä siirrettävyyden ansiosta, silloin kun siirrettävyyden ominaisuuden omaava serverless- sovellus siirretään toiseen pilviympäristöön. Kuten Tanasa (2019) painotti artikkelissaan, huolella harkituilla ja tietoon perustuvilla valinnoilla on mahdollista vaikuttaa laajemminkin sovellukseen kehittämistyöhön liittyvään prosessiin eli muihinkin sovellusten elinkaaren eri vaiheisiin kuten julkaisuun, käyttöönottoon ja ylläpitoon. Kolmas tapaustutkimuksena tehtävä hyödyllinen jatkotutkimusaihe voisi olla CloudEvents-standardin soveltaminen ja sen käytöstä syntyvien vaikutusten arviointi.

Lähteet

Abideen, Z. 2022. An Introduction to Available Triggers to Invoke a Lambda Function. Artikkeli Linux Hintin www-sivuilla. Viitattu 26.11.2022. https://linuxhint.com/available-triggers-invoke-lambda-function/

Al-Tukhi, Z. 2022. NoSQL Database. Artikkeli kirjoittajan www-sivuilla. Viitattu 10.11.2022. https://altukhizm.com/2022/06/08/nosql-database/

Alvarez, F. 2022. Don't Waste Your Time: Reacting to the Passage of Time Using Cloud Events and AWS Step Functions. Artikkeli Mediumin www-sivustolla. Viitattu 26.11.2022. https://medium.com/ssense-tech/dont-waste-your-time-reacting-to-the-passage-of-time-using-cloud-events-and-aws-step-functions-2a572cf82f4

AWS – Deploying. N.d. Serverless Framework -dokumentaation www-sivu. Viitattu 5.12.2022. https://www.serverless.com/framework/docs/providers/aws/guide/deploying/

AWS Lambda context object in Node.js. N.d. AWS Lambda Developer Guide. Amazon AWS -pilviympäristön www-sivut. Viitattu 7.11.2022. https://docs.aws.amazon.com/lambda/latest/dg/nodejs-context.html

AWS Lambda Events. N.d. Serverless Framework -ohjelmistokehyksen dokumentaation www-sivu. Viitattu 8.11.2022. https://www.serverless.com/framework/docs/providers/aws/guide/events

AWS Lambda FAQs. N.d. Amazon AWS -pilviympäristön www-sivut. Viitattu 24.9.2022. https://aws.amazon.com/lambda/faqs/

Azure – Events. N.d. Serverless Framework -ohjelmistokehyksen dokumentaation www-sivu. Viitattu 8.11.2022. https://www.serverless.com/framework/docs/providers/azure/guide/events

Azure Functions JavaScript developer guide. 2022. Azuren dokumentaation www-sivut. Viitattu 3.10.2022. https://learn.microsoft.com/en-us/azure/opbuildpdf/azure-functions/toc.pdf?branch=live

Beswick, J. 2021. Operating Lambda: Understanding event-driven architecture – Part 1. AWS Compute Blog -blogin kirjoitus. Amazon AWS -pilviympäristön www-sivut. Viitattu 24.9.2022. https://aws.amazon.com/blogs/compute/operating-lambda-understanding-event-driven-architecture-part-1/ serverless-sovellusten suunnittelua

Choose a stream processing technology in Azure. 2022. Artikkeli Microsoftin Azure-dokumentaation www-sivuilla. Viitattu 10.11.2022. https://learn.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/stream-processing

Choosing between REST APIs and HTTP APIs. N.d. Amazon AWS -pilviympäristön www-sivut. Viitattu 23.9.2022. https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html

CloudEvents. A specification for describing event data in a common way. 2022. Viitattu 24.9.2022. CloudEventsin www-sivut. https://cloudevents.io/

CloudEvents - JSON event format. 2022. Artikkeli Google Cloud -pilviympäristön www-sivuilla. Viitattu 26.11.2022. https://cloud.google.com/eventarc/docs/workflows/cloudevents

Create Function App. N.d. Azure portal. Microsoft Azuren www-sivut. Viitattu 6.12.2020. https://portal.azure.com/#create/Microsoft.FunctionApp

Dao, N. 2018. THE SERVERLESS SERIES – What Is Serverless? Artikkeli Neapin www-sivuilla. Viitattu 4.12.2022. https://blog.neap.co/the-serverless-series-what-is-serverless-d651fbacf3f4

De Sogos, D. 2017. Serverless Architecture: From buzzword to reality – Part I. Artikkeli Hexactan www-sivuilla. Viitattu 27.11.2022. https://www.hexacta.com/serverless-architecture-from-buzzword-to-reality-part-one/

Edwards, J. 2020. Serverless computing: Ready or not? Artikkeli Network Worldin www-sivuilla. Viitattu 9.6.2020. https://www.networkworld.com/article/3514188/serverless-computing-ready-or-not.html

Goasguen, S. 2019. Serverless and multi-cloud: Hype or reality? Artikkeli TechBeaconin www-sivuilla. Viitattu 13.7.2020. https://techbeacon.com/enterprise-it/serverless-multi-cloud-hype-or-reality

Hein, C. 2018. Deploy OpenFaaS on Amazon EKS. Amazon AWS -pilviympäristön www-sivut. Viitattu 13.11.2022. https://aws.amazon.com/blogs/opensource/deploy-openfaas-aws-eks/

Hohpe, G. 2022. Concerned about Serverless Lock-in? Consider Patterns! The Architect Elevatorin www-sivut. Viitattu 4.12.2022. https://architectelevator.com/cloud/serverless-design-patterns/

Invoking Lambda functions. N.d. AWS Lambda Developer Guide. Amazon AWS:n www-sivut. Viitattu 26.11.2022. https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html

Jeyakumaran, S. 2022. The Pro’s and Con’s of using AWS MSK Serverless. Artikkeli Digion www-sivuilla. Viitattu 10.11.2022. https://digio.com.au/learn/blog/the-pros-and-cons-of-using-aws-msk-serverless/

Kananen, J. 2015. Opinnäytetyön kirjoittajan opas. Näin kirjoitan opinnäytetyön tai pro gradun alusta loppuun. Jyväskylän ammattikorkeakoulun julkaisuja 202. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Kokonaisarkkitehtuuri. N.d. Bisnesteknologiamallin (BT-standardin), johtamisen viitekehyksen verkkoversion luku 2.2. Business Technology Forumin www-sivut. Viitattu 29.9.2022. https://btmalli.fi/book/demand/enterprise-architecture/

Koljonen, M. 2020. Pilvinatiivi vai pilviagnostinen? Artikkeli LinkedInin www-sivuilla. Viitattu 27.11.2022. https://www.linkedin.com/pulse/pilvinatiivi-vai-pilviagnostinen-matti-koljonen/

Kritikos, K. & Skrzypek, P. 2018. A Review of Serverless Frameworks. Artikkeli 17–20 December 2018 pidetystä konferenssista tehdystä julkaisusta 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, Institute of Electrical and Electronics Engineers, 161-168. Viitattu 12.5.2020. https://janet.finna.fi, IEEE Xplore Digital Library.

Kurniawan, A. & Lau, W. 2019. Practical Azure Functions: A Guide to Web, Mobile, and IoT Applications. New York: Apress.

Lambda function URLs. N.d. AWS Lambda Developer Guide. Amazon AWS:n www-sivut. Viitattu 26.11.2022. https://docs.aws.amazon.com/lambda/latest/dg/lambda-urls.html

Lambda concepts. N.d. AWS Lambda Developer Guide. Amazon AWS -pilviympäristön www-sivut. Viitattu 24.9.2022. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-concepts.html

Levy, E. 2022. Apache Kafka vs Amazon Kinesis – Comparing Setup, Performance, Security, and Price. Artikkeli Upsolverin www-sivuilla. Viitattu 10.11.2022. https://www.upsolver.com/blog/comparing-apache-kafka-amazon-kinesis

Likness, J. 2022. Serverless apps: Architecture, patterns, and Azure implementation. Microsoft Developer Division, .NET, and Visual Studio product teams. Microsoft Corporation. Viitattu 5.11.2022. https://raw.githubusercontent.com/dotnet-architecture/eBooks/main/current/serverless/Serverless-apps-Architecture-patterns-and-Azure-implementation.pdf

Linden, J. V. D. 2020. Serverless testing strategy. Artikkeli Mediumin www-sivuilla. Viitattu 4.12.2022. https://jeromevdl.medium.com/serverless-testing-strategy-b12ada2252f

Martinez, P. 2021. Hexagonal Architecture, there are always two sides to every story. Artikkeli Mediumin www-sivuilla. Viitattu 4.12.2022. https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c

McLellan, C. 2019. Multicloud: Everything you need to know about the biggest trend in cloud computing. Artikkeli ZDnetin sivustolla. Viitattu 26.11.2022. https://www.zdnet.com/article/multicloud-everything-you-need-to-know-about-the-biggest-trend-in-cloud-computing/

Microsoft Azure. 2021. Supported languages in Azure Functions. Artikkeli Microsoftin Azure -dokumentaation sivustolla. Viitattu 1.10.2022. https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages

Padmanabhan, A. 2022. Leaky Abstractions. Artikkeli Devopedian www-sivulla. Viitattu 2.12.2022. https://devopedia.org/leaky-abstractions

Poccia, D. 2017. AWS Lambda in Action: Event-Driven Serverless Applications. Manning Publications. Viitattu 26.11.2022. https://janet.finna.fi, Skillsoft Books ITPro.

Samaranayake, M. 2022. DynamoDB vs Azure Cosmos DB - The Ultimate Comparison. Artikkeli Dynobasen www-sivuilla. Viitattu 10.11.2022. https://dynobase.dev/dynamodb-vs-cosmos/

Set up Lambda integrations in API Gateway. N.d. Amazon API Gateway Developer Guide. Amazon AWS -pilviympäristön www-sivut. Viitattu 23.9.2022. https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-integrations.html

Shah, V. 2022. AWS Lambda vs Azure Functions: Serverless Computing. Artikkeli TatvaSoftin www-sivuilla. Viitattu 6.1.2022. https://www.tatvasoft.com/blog/aws-lambda-vs-azure-functions/

Shilkov, M. 2019. AWS Lambda vs. Azure Functions: 10 Major Differences. Artikkeli IOD Cloud Technologies Research Ltd:n www-sivuilla. Viitattu 3.10.2022. https://iamondemand.com/blog/aws-lambda-vs-azure-functions-ten-major-differences/

Stigler, M. 2018. Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud. Apress. Viitattu 21.5.2020. https://janet.finna.fi, Books24x7 ITPro.

Swail, P. 2020. Should development firms support multiple clouds for their serverless client projects? Artikkeli Serverless Firstin www-sivuilla. Viitattu 16.8.2020. https://serverlessfirst.com/dev-firm-multicloud-capability/az

Tanasa, W. 2019. Mitigating serverless lock-in fears. Artikkeli ThoughtWorksin www-sivuilla. Viitattu 26.5.2020. https://www.thoughtworks.com/insights/blog/mitigating-serverless-lock-fears

Teiger, M. 2022. Serverless infrastructure using different frameworks (part 2). Artikkeli Padokin www-sivustolla. Viitattu 4.12.2022. https://www.padok.fr/en/blog/severless-iac-tools

Using AWS Lambda with other services. N.d. AWS Lambda Developer Guide. Amazon AWS -pilviympäristön www-sivut. Viitattu 24.9.2022. https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html#intro-core-components-event-sources

Using Lambda with self-managed Apache Kafka. N.d. AWS Lambda Developer Guide. Amazon AWS:n www-sivut. Viitattu 5.12.2022. https://docs.aws.amazon.com/lambda/latest/dg/with-kafka.html

Using OpenFaaS on AKS. 2022. Artikkeli Microsoftin Azure-dokumentaation sivustolla. Viitattu 13.11.2022. https://learn.microsoft.com/en-us/azure/aks/openfaas

Vega, M. A. P. 2019. Quick guide: Monolith vs Microservices vs Serverless. Hexactan www-sivut. Viitattu 8.5.2020. https://www.hexacta.com/quick-guide-monolith-vs-microservices-vs-serverless/

Vijayan, J. 2018. Serverless l lock-in: Should you be worried? Artikkeli TechBeaconin sivustolla. Viitattu 5.12.2022. https://techbeacon.com/enterprise-it/serverless-vendor-lock-should-you-be-worried

Weston, R. 2017. Serverless Architectures and Continuous Delivery. Artikkeli GoCD:n www-sivustolla. Viitattu 24.5.2020. https://www.gocd.org/2017/06/26/serverless-architecture-continuous-delivery/

What is Amazon API Gateway? N.d. Amazon AWS -pilviympäristön www-sivut. Viitattu 23.9.2022. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

What is Application Domain. 2020. Artikkeli QA Platformsin www-sivustolla. Viitattu 27.5.2020. https://qa-platforms.com/what-is-application-domain/

What is AWS Lambda? N.d. AWS Lambda Developer Guide. Amazon AWS -pilviympäristön www-sivut. Viitattu 24.9.2022. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

What is BaaS? | Backend-as-a-Service vs. serverless. N.d. Artikkeli Cloudflaren www-sivuilla. Viitattu 5.11.2022. https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/

What is serverless? 2022. Artikkeli Red Hatin sivustolla. Viitattu 26.11.2022. https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless

Wilde, E. 2019. Introducing OpenFaaS for AWS Lambda (faas-lambda). Artikkeli OpenFaaS Ltd:n www-sivuilla. Viitattu 13.11.2022. https://www.openfaas.com/blog/introducing-openfaas-for-lambda/

Wong, Y. Y. 2018. Serverless/FaaS (eg AWS Lambda) vs PaaS (eg Azure Web Apps). Artikkeli Mediumin www-sivustolla. Viitattu 24.11.2022. https://medium.com/@houdinisparks/serverless-faas-eg-aws-lambda-vs-paas-eg-azure-web-apps-ca0e7146b1c4

Yadav, A. K. 2019. Apache Kafka VS AWS Kinesis. Artikkeli Mediumin www-sivustolla. Viitattu 17.8.2020. https://medium.com/faun/apache-kafka-vs-apache-kinesis-57a3d585ef78