Markku Lehtonen

SAAS-ohjelmiston laajentaminen serverless-funktioilla rakennetulla komponentilla

Tietotekniikan pro gradu -tutkielma

20. lokakuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Markku Lehtonen

Yhteystiedot: markkuolavi82@gmail.com

Ohjaajat: Ari Viinikainen

Työn nimi: SAAS-ohjelmiston laajentaminen serverless-funktioilla rakennetulla komponentilla

Title in English: Integrating serverless component into SAAS application

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 56+10

Tiivistelmä: Tutkielmassa ohjelmistoalan yrityksen käytännön ongelma ratkaistiin serverless-paradigman avulla. Tutkimusmenetelmänä käytettiin konstruktiivista tutkimusotetta. Tutkimustulokseksi saatiin muun muassa, että serverlessin ja FAASin avulla voidaan kustannustehokkaasti rakentaa isompiakin itsenäisiä komponentteja, jotka ovat hyvin ylläpidettäviä. Päästäkseen optimaaliseen tulokseen kehittäjän tulee olla selvillä serverlessin ja funktioiden kustannusrakenteesta, rajoitteista ja mahdollisuuksista, joiden nykytilaan tämä tutkimus myös pureutuu. Tutkimuksessa havaittiin myös, että serverless ja DevOps jakavat samoja päämääriä ja niiden yhteiskäytössä on valtavasti potentiaalia. Lisäksi tutkimuksen perusteella voidaan todeta konstruktiivisen tutkimuksen sopivan erinomaisesti modernien ohjelmistoalan teknologioiden käytännön sovellusten innovointiin ja sitä kautta teorian muovaamiseen ja uusien mielenkiintoisten tutkimuskysymysten syntyyn.

Avainsanat: Serverless computing, FAAS, pilvipalvelut, ohjelmistokehitysprosessi, integraatio, .Net, olio-ohjelmointi, DevOps**,** ohjelmistoarkkitehtuurit, ylläpito, konstruktiivinen tutkimus

Abstract: In this Master´s thesis a practical problem being raised by software industry company was solved with serverless paradigm. The study was made by using constructive research as research method. The research revealed inter alia that serverless and cloud

functions are excellent cost-effective building blocks for larger independent components as well without compromising maintainability. In order to gain optimal results, one should be aware of the current state of the cost model, possibilities and limitations of functions. This study dug into those issues and more. In this study I discovered that serverless and DevOps share same goals and using them parallel has enormous potential. Besides this study points out that constructive research method fits well for innovating modern software industry applications and thus shaping exciting theory and creating new fascinating research questions.

Keywords: Serverless, Function-As-A-Service, Cloud computing, Integration, .Net, object-oriented programming, DevOps, software architectures, software maintenance, constructive research

Termiluettelo

AI	Tekoäly (engl. Artificial Intelligence)
API	Ohjelmointirajapinta (engl. A Programming Interface)
AZURE	Microsoftin pilvipalvelualusta
CD	Jatkuva toimitus (engl. Continuous Delivery)
CDE	Jatkuva tuotantovalmius (engl. Continuous Deployment)
CGI	Standardi, jota käytetään, kun palvelimet generoivat dynaamisesti web-sivustoja (engl. Common Gateway Interface)
CI	Jatkuva integraatio (engl. Continuous Integration)
CLI	Komentorivirajapinta (engl. Command Line Interface)
CM	Jatkuva ylläpito (engl. Continuous Maintenance)
DEVOPS	Kehitys ja operaatiot (engl. DEVelopment + OPerationS)
DNS	Nimipalvelujärjestelmä (engl. Domain Name System)
IAAS	Infrastruktuuri palveluna (engl. Infrasrtucture-As-A-Service)
IAC	Infrastruktuuri koodina (engl. Infrastructure-As-Code)
IDE	Kehitysympäristö (engl. Integrated Development Enviroment)
IOT	Esineiden internet (engl. Internet Of Things)
JIRA	Tehtävienhallintaohjelmisto
KANBAN	Lean hallintametodi
FAAS	Funktiot palveluina (engl. Function-As-A-Service)
NUNIT	.NETin viitekehys yksikkötestaukseen
ON-PREMISES	Ohjelmistojen installointi ja operointi asiakkaan palvelimella ja infrastruktuurissa
REDIS	Tietokantapalvelu välimuistille

RAM	Keskusmuisti (engl. Random Access Memory)
REST	Suunnittelutapa hypermediaverkoille (engl. Representational State Transfer)
SAAS	Ohjelmistot palveluina (engl. Software-As-A-Service)
SLA	Palvelutasosopimus (engl. Service-Level Agreement)
SOA	Palvelukeskeinen arkkitehtuurityyli (engl. Service Oriented Architecture)
SCRUM	"Viitekehys monimutkaisten tuotteiden kehittämiseen, julkaisuun ja ylläpitoon." (Scrumguides.org 2017)
TDD	Testilähtöinen ohjelmistokehitys (engl. Test Driven Development)
YAML	Kieli, jolla voidaan tallentaa esimerkiksi konfigurointeja (engl. YAML Ain´t Markup Language)

Kuviot

Kuvio 1. Konstruktiivisen tutkimuksen mekanismit Oyegoken (2011) mukaan. 4
Kuvio 2. Konstruktiivisen tutkimusprosessin vaiheet 6
Kuvio 3. Kohdealueen arkkitehtuuri 10
Kuvio 4. Kohdeyrityksen CI/CDE-putki 11
Kuvio 5. Serverless-alustan yleinen arkkitehtuuri (mukaillen Sewak & Singh 2018; Baldini et al. 2017). 15
Kuvio 6. Ääretön DevOps-silmukka kuvaa ohjelmistokehityksen ja IT-palvelutoimintojen integroinnin (AppDynamics Inc. 2014) 20
Kuvio 7. CI-, CD- ja CDE-tehtävät (Shahin et al. 2017). 23
Kuvio 8. Toteutusratkaisun arkkitehtuuri osana kokonaisarkkitehtuuria 32
Kuvio 9. Komponentin ohjelmallinen rakenne 34
Kuvio 10. Julkaistut funktiot resursseina Azuren Kudussa 39

Taulukot

Taulukko 1. FAAS-palveluntarjoajien vertailua 2019 (https://aws.amazon.com/lambda/features/, https://azure.microsoft.com/en-us/services/functions/, https://cloud.google.com/functions/ ja https://www.ibm.com/cloud/functions. Haettu 28.5.2019) 18

Sisältö

1 JOHDANTO 1

2 TUTKIMUS 3
 2.1 Tutkimusmenetelmä 3
 2.2 Aiempi tutkimus ja tutkimuspotentiaali 5
 2.3 Tutkimuksen rakenne 6
 2.4 Tutkimusongelma 7
 2.5 Tutkimusyhteistyö kohdeorganisaation kanssa 8

3 KOHDEALUEEN YMMÄRRYS 10
 3.1 Kohdealueen arkkitehtuuri 10
 3.2 Ohjelmistokehitys 11
 3.3 Rakennettavan komponentin vaatimukset 12

4 SERVERLESS-PARADIGMA JA FUNKTIOT PALVELUINA 13
 4.1 Serverlessin määrittelyä ja taustaa 13
 4.2 Funktiot palveluina 14
 4.3 Yleistetty serverless-arkkitehtuuri 15
 4.4 Serverlessin ja FAASin mahdollisuuksista ja rajoitteista 15
 4.5 Palveluntarjoajien vertailua 17

5 DEVOPS 20
 5.1 DevOpsin määrittelyä ja taustaa 20
 5.2 Miksi DevOps? 22
 5.3 Continuous-käytänteet 23
 5.4 DevOps-työkalut ja niiden käyttö kohdeorganisaatiossa 24
  5.4.1 DevOpsissa käytetyt työkalut 24
  5.4.2 DevOps-työkalujen käyttö kohdeorganisaatiossa 25

6 OHJELMISTON YLLÄPITO 27

7 TOTEUTUKSESSA HUOMIOITAVAA 28
 7.1 Kohdealueen analysoinnin pohjalta 28
 7.2 Ylläpidettävyyden huomiointi 28
 7.3 Vaatimuksista 29
 7.4 DevOpsin huomiointi 29
 7.5 Kustannuksista 30

8 INNOVOINTI JA KONSTRUKTION TOTEUTUS 31
 8.1 Proof-Of-Concept 31
 8.2 Toteutuksen arkkitehtuuri osana kokonaisarkkitehtuuria 32
  8.2.1 Tapahtumankulku 32
  8.2.2 Arkkitehtuurin perustelua 33

8.3 Toteutus osana DevOps-putkea ... 36
8.3.1 Infrastruktuurin luonti Azureen ... 36
8.3.2 CI- ja CDE -putket ... 38
8.4 Kustannusten muodostuminen ... 39
8.4.1 Osio 1: SMS- ja e-mail-palveluntarjoajat ... 40
8.4.2 Osio 2: Jonojen kustannukset ... 40
8.4.3 Osio 3: Funktiot ... 41

9 KONSTRUKTION TOIMIVUUS JA SOVELLUSALA ... 42
9.1 Konstruktion hyödyllisyys ja markkinatestit ... 42
9.2 Soveltamisala ... 43

10 TEOREETTINEN KONTRIBUUTIO JA POHDINTA ... 46
10.1 Serverlessin ja FAASin mahdollisuuksien ja rajoitteiden peilaus tutkimustuloksiin ... 46
10.2 Tutkimusprosessin pohdintaa ... 51
10.3 Rajoitteet ja jatkotutkimus ... 52
10.4 Lopuksi ... 52

LÄHTEET ... 54

LIITTEET ... 57
Liite 1 Varausvahvistuksen data esitettynä JSON-schema-muodossa ... 57
Liite 2 Kululaskelmia ... 60
Liite 3 Readme (GitHubin yksityisestä Git-hakemistosta) ... 63

1 Johdanto

Nykyisin monet yritykset ovat siirtyneet kustannustehokkaaseen Software-As-A-Service (SAAS) toimitusmalliin, jossa ohjelmistoja tarjotaan palveluina pilvessä. Mallin tarjoamia etuja palveluntarjoajalle ovat muun muassa muiden pilvipalveluiden hyvät integrointimahdollisuudet ja se, että kerran kehitetyt resurssit voidaan jakaa useiden tenanttien kesken. Tilaajalle malli tarjoaa muun muassa joustavuutta palveluntarjoajan valinnassa, säästöjä infrastruktuurissa ja hyvää palvelun saatavuutta. Ohjelmistokehitysprosessin nopeutuminen ja tarvittavan alkupääoman (kuten infrastruktuurihankinnat) määrän väheneminen ovat johtaneet markkinoille pääsyn helpottumiseen ja kilpailun kasvamiseen. Myös tilaajilla on nykyään vaivattomampaa vaihtaa palveluntarjoajaa, ja siksi palveluntarjoajan onkin jatkuvasti etsittävä uusia potentiaalisia käyttäjiä ja pidettävä palvelut kiinnostavina. (Aleem et al. 2019).

Vuonna 2017 serverless-arkkitehtuurimarkkinan arvo oli 3,2 miljardia ja DevOps markkinan 2,9 miljardia dollaria. Lisäksi molempien ennustetaan kasvavan yli 20 prosentin vuosivauhtia (MarketsandMarkets 2018). Osaltaan suurten lukujen taustaa voisi selittää IBM:n PhD. Diazin (2016) näkemys:

> *”An important trend in* cloud computing *is the rapidly diminishing size and complexity of the infrastructure needed to create innovation.”*

Serverless (arkkitehtuuri) ja DevOps (prosessi) ovat moderneja keinoja keskittyä oleelliseen eli innovoimaan uusia asiakasta hyödyttäviä palveluja, joiden avulla pärjätään kovassa kilpailussa. Ne ovat tuoreita ja mielenkiintoisia aihealueita, joilla on paljon potentiaalia niin kaupallisesti kuin tieteellisestikin. Tutkimuksen aiheen rajauksen suuret linjat löytyivät, kun keskustelin edellä mainituista kiinnostuksen kohteistani keskisuomalaisen ohjelmistoalan yrityksen kanssa ja heiltä löytyi sopivan kokoinen käytännön ongelma tutkimuskohteeksi. Ongelma liittyi SAAS-mallilla tarjottavaan ohjelmistoon, jota pitää laajentaa uudella palvelulla. Tässä tutkimuksessa tärkein aihealue on serverless ja etenkin sen alle kuuluva Function-As-A-Service (FAAS), joilla käytännön ongelmaa lähdettiin ratkaisemaan. DevOpsilla on myös merkittävä rooli, sillä se määrittää puitteet ohjelmistokehitysprosessille.

Koska ongelma liittyy liiketoimintaan, yritystä kiinnostavat luonnollisesti rakennettavan sovelluksen kulut ja ylläpidettävyys. Luvussa 2 tarkennetaan tutkimuksen näkökulmia ja sitä, mitä vastauksia käytännön ongelman ratkaiseminen antaa tieteen saralle.

Tässä vaiheessa mainittakoon, että tutkimuksessa on käytetty muutamia englanninkielisiä termejä, sillä kaikille teknisille termeille ei löydy vakiintunutta suomenkielistä vastinetta.

2 Tutkimus

Ensin luvussa esitellään tutkimuksessa käytetty tutkimusmenetelmä ja tarkastellaan hieman tutkimuksen tutkimuspotentiaalia peilaten aiempaan serverlessiin liittyvään tutkimukseen. Sitä seuraa tutkimuksen rakenteen esittely, josta selviää myös, miten valittu menetelmä tarjoaa loogisen rakenteen tutkimukselle. Lopuksi pureudutaan tutkimusongelmaan, tutkimuksen näkökulmiin, arvioidaan tutkimusmenetelmän sopivuutta sekä selvitetään, miten tutkimusyhteistyö järjestetään kohdeorganisaation kanssa.

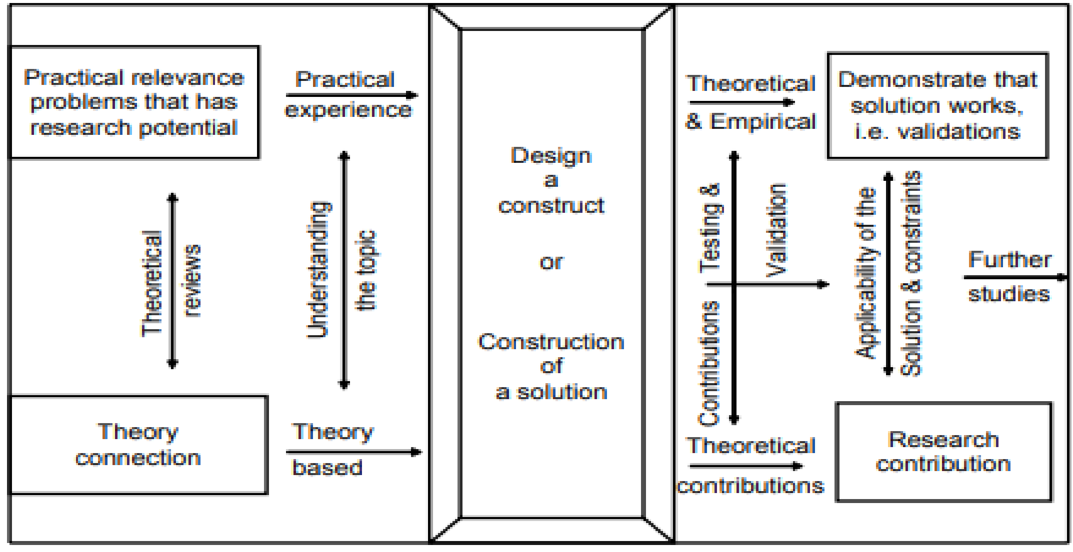
2.1 Tutkimusmenetelmä

Tutkimusmenetelmäksi valittiin konstruktiivinen tutkimusote. Menetelmä pohjautuu Eero Kasasen väitöskirjaan vuodelta 1986, ja sitä on käytetty paljon etenkin pohjoismaisessa liiketalouden, lääketieteen ja teknisten alojen tutkimuksessa (Kasanen et al. 1993). Seuraava Lukan (2001) määritelmä tiivistää hyvin menetelmän ytimen:

” Konstruktiivinen tutkimusote on innovatiivisia konstruktioita tuottava metodologia, jolla pyritään ratkaisemaan reaalimaailman ongelmia ja tällä tavoin tuottamaan kontribuutioita sille tieteenalalle, jossa sitä sovelletaan.”

Kun tarkastellaan konstruktiivisen tutkimusotteen sijoittumista tiedekenttään, toteaa Kasanen et al. (1993) konstruktiivisen tutkimuksen kuuluvan soveltavan tutkimuksen piiriin ja olevan luonteeltaan sekä empiiristä että normatiivista. Oyegoken (2011) mukaan sen voi myös linkittää pragmatismiin. Pragmatismi näkyy konstruktiivisessa tutkimuksessa siinä, että ongelmaa lähestytään käytännön sovelluksen kautta ja tulosten validointi perustuu konstruktion hyödyllisyyteen.

Konstruktiivisen tutkimuksen elementeistä löytyy useita havainnollistuksia (esim. Lukka 2001; Kasanen et al. 1993). Oyegoke (2011) onnistuu erityisen hyvin kiteyttämään, miten konstruktiivinen tutkimus syntyy ja mitkä seikat tulee huomioida (kuvio 1). Kuviosta on tärkeä havaita empiirisen tutkimuksen suhde teoriaan. Empiirinen osuus (konstruktion innovointi ja testaus) toteutetaan olemassa olevan teorian ymmärtämisen pohjalta. Lisäksi tutkimuksen lopuksi konstruktion tuomaa uutta tietoa peilataan takaisin kyseiseen teoriaan.



Kuvio 1. Konstruktiivisen tutkimuksen mekanismit Oyegoken (2011) mukaan.

Konstruktiivisen tutkimusmenetelmän valintaa tukee näkemys, jonka mukaan tietojärjestelmien tutkimuksessa (ISR) tulisi teorian lisäksi myös tuottaa tuloksia, jotka ovat sovellettavissa liike-elämän ongelmien ratkaisemiseksi (Piirainen & Gonzales 2014). Lukka (2001) määrittelee viisi konstruktiivisen tutkimuksen ydinpiirrettä, joista kaksi ensimmäistä sopivat hyvin tämän tutkimuksen tutkimusmenetelmän valinnan perusteeksi. Ensinnäkin on oltava tosielämän ongelma, joka tulee käytännössä ratkaista. Toiseksi tutkimuksen tulee tuottaa konstruktio, joka ratkaisee ongelman ja jonka käytäntöön soveltuvuutta toteutus testaa.

Työn tilaajaorganisaatiolla on kaksi selkeää tavoitetta. Suppeampi tavoite on tarjota ratkaisu asiakkaan käytännön ongelmaan. Laajempi tavoite liittyy siihen, että ongelman ratkaisutapa on vapaasti valittavissa ja olemassa oleva arkkitehtuuri sekä käytetty kehitysprosessi (DevOps) tarjoavat hedelmälliset lähtökohdat tutkia ratkaisun toteutusta serverless-paradigmalla. Toteutettava ominaisuus on tapahtumapohjainen itsenäinen komponentti, ja siksi serverless on hyvä vaihtoehto ratkaisun arkkitehtuuriksi. Tutkimuksella saataisiin tärkeää tietoa tulevaisuuden projekteihin, joissa sovelluksia tai niiden osia tehdään tai siirretään pilviympäristöön modernilla tavalla

2.2 Aiempi tutkimus ja tutkimuspotentiaali

Kuvion 1 teoriayhteys ja tutkimuspotentiaali tässä tutkimuksessa liittyvät serverless-aiheeseen. Serverlessistä ja sen alakäsitteestä FAASista on kirjoitettu useita artikkeleita eri näkökulmista. Teknologia on vielä nuori ja kehittymässä ja siihen liittyy paljon avoimia kysymyksiä. Tutkimusaiheen tuoreus näkyy muun muassa siinä, että vuosilta 2017 ja 2018 löytyy paljon artikkeleita, joissa vasta määritellään käsitteitä, visioidaan tai pohditaan etuja ja haittoja (esimerkiksi Varghese ja Buyya 2018; Savage 2018; Baldini et al. 2017). Jo artikkeleiden nimet kuten ”Going serverless” ja ”Demystifying serveless computing” näiltä vuosilta kertovat, että ilmiö on vielä hyvin nuori. Lisäksi aiheen tuoreus näkyy esimerkiksi julkaisussa, jossa Eyk et al. (2017) vastaavat kehittämällään mallilla serverlessin termistön ja yleisen vision puutteeseen. Tutkijat myös odottavat kasvavaa tiedeyhteisön kiinnostusta aiheeseen.

Serverless-tutkimuksessa käsitellään myös jonkin verran kontteja (Pérez et al. 2018), edge computingia ja IOTia (Nastic et al. 2017). Tieteelliseltä kannalta erityisen mielenkiintoinen tutkimus on Malawskin (2016) tapaustutkimus, jossa tutkittiin FAASin soveltuvuutta tieteellisten työnkulkujen (engl. scientific workflows) kanssa. Diaz (2019) luettelee syitä, joiden vuoksi serverless tulee olemaan iso tekijä tulevaisuuden ohjelmistokehityksessä. Näitä ovat kustannusten säästö, yrityskasvu skaalautuvuuden kautta sekä IOT- ja mobiililaitteiden määrän kasvu sekä kognitiivisten palveluiden tarjoaminen serverlessiä hyödyntäen. Edellä mainituista tutkimuksista voi päätellä serverlessin monikäyttöisyyttä tulevaisuudessa ja että paradigman hallitsemisesta tullee tärkeää IT-alan yrityksille. Serverlessin monet mahdollisuudet uusien teknologioiden kanssa tarjonnevat myös paljon mielenkiintoisia aiheita tulevaisuuden tieteelliselle tutkimukselle

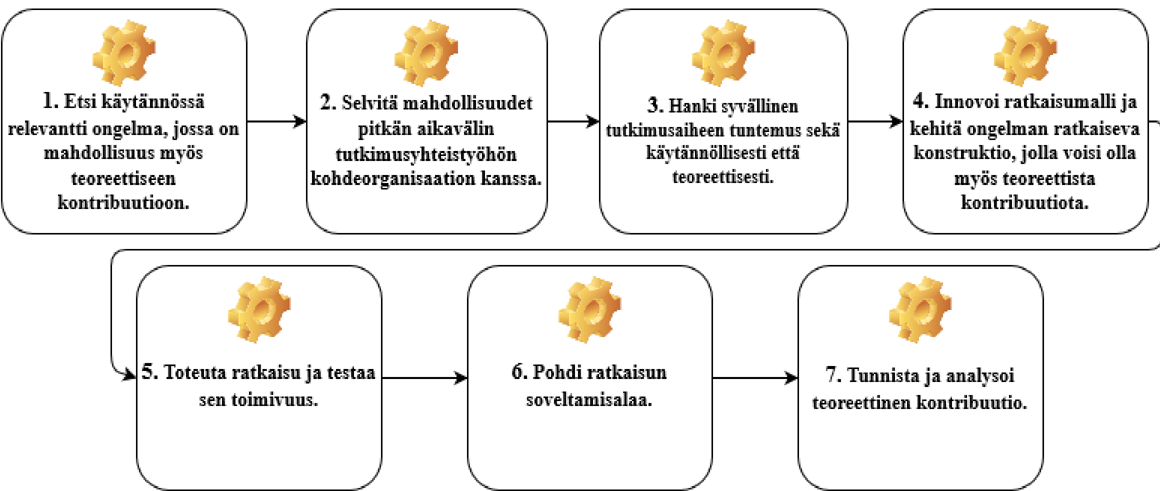
Hyvät teoreettiset lähtökohdat tälle tutkimukselle antaa pohdinta serverlesiin liittyvistä haasteista ja avoimista tutkimuskysymyksistä (Balidini et al. 2017) sekä haasteet, mahdollisuudet ja visiot, jotka Eyk et al. (2017) tunnistivat. Edellä mainituista tähän tutkimukseen relevantteja poimintoja kohdeyrityksen kannalta ovat muun muassa kustannukset, DevOps- ja kehityskäytänteet, kehittäjän näkökulma, työkalut sekä ylläpito. Lähellä tämän tutkimuksen aihealueita on myös monimenetelmällinen tutkimus, jossa Leitner et al. (2018) tutkivat FAAS-

sovellusten käyttökohteita, käytäntöjä sekä niiden etuja ja haasteita. Käytännönläheiselle tutkimukselle on tarvetta, sillä Leitner et al. (2018) mainitsevat avoimeksi tutkimusongelmaksi rakentaa funktioista kokonainen sovellus tai purkaa monoliittinen sovellus mikropalveluiksi (engl. microservice) FAASia hyödyntäen.

Artikkelia, jossa funktioilla laajennetaan olemassa olevaa sovellusta tai rakennetaan itsenäinen komponentti DevOps-putken (engl. pipeline) kanssa, ei löytynyt. Aihealueelta ei löytynyt myöskään aiempaa konstruktiivista tai sitä muistuttavaa Design-tutkimusta. Yleisesti ottaen aiempi tutkimus on pitkälti vertailevaa, kyselyhin perustuvaa tai niissä tehdään niin sanottu POC-toteutus jonkin laatuatribuutin (esimerkiksi suorituskyky) tutkimiseksi.

2.3 Tutkimuksen rakenne

Tutkimus ja sen rakenne noudattavat alla esiteltyä konstruktiivisen tutkimusotteen prosessia. Kuvan tekstit on lainattu Lukan (2003) artikkelista. Prosessin vaiheet ovat löydettävissä myös kuviosta 1.



Kuvio 2. Konstruktiivisen tutkimusprosessin vaiheet

Prosessin vaiheet antavat tutkimukselle rakenteen seuraavasti:

- Luvussa 2.4 esitellään tutkimusongelma (prosessin vaihe 1).
- Luku 2.5 avaa tutkijan roolia ja yhteistyötä kohdeorganisaation kanssa (prosessin vaihe 2)

- Teorialuvut 3-6 käsittelevät kohdealuetta, aihealueelle relevanttia teoriaa sekä ongelmaa (prosessin vaihe 3).
- Luvuissa 7 ja 8 innovoidaan ratkaisu edellä mainittujen lukujen tietämyksen pohjalta (prosessin vaiheet 4 ja 5) sekä kuvataan sen toteutus.
- Luvussa 9 pohditaan konstruktion validiutta ja toimivuutta (vaiheet 5-6).
- Luvussa 10 analysoidaan konstruktion teoreettinen kontribuutio (vaihe 7).

2.4 Tutkimusongelma

Tutkimuksen lähtökohtana on serverless-paradigman hyödyntäminen asetelmassa, jossa olemassa olevaa sovellusta laajennetaan integroimalla siihen serverless-arkkitehtuurilla toteutettu komponentti. Tutkimusongelmana on tarkastella toteutuksen vaikutuksia etenkin DevOpsin, kustannusten ja ylläpidettävyyden kannalta sekä pohtia tulosten hyödynnettävyyttä muihin käyttötapauksiin. Tutkimusongelmaa lähdetään ratkaisemaan konstruktiivisen tutkimuksen prosessin avulla (kuvio 2).

Tutkimusongelman ratkaisu pohjautuu ensinnäkin olemassa olevan sovelluksen arkkitehtuurin ja kehitysprosessin analysointiin. Analyysi sisältää myös tärkeimpien teknologioiden ja prosessien katsauksen tieteellisestä näkökulmasta. Sen jälkeen perehdytään siihen, millä keinoin käytännön ongelman voisi ratkaista serverless-paradigman avulla sekä mitä etuja ja haittoja siitä mahdollisesti olisi.

Tutkimuksessa serverless määritellään ja katsotaan, miltä sen mahdollisuudet ja rajoitteet näyttävät tämänhetkisen tutkimuksen valossa. Teknologian nopea kehitys ja se, että serverlessiä tarjotaan palveluna, johtavat myös tiedon nopeaan vanhenemiseen, sillä uusia innovaatioita syntyy jatkuvasti kilpailluilla markkinoilla. Tutkimus tarjoaa mahdollisuuden peilata, miten etenkin konstruktiota varten valitun palveluntarjoajan palvelut vastaavat tieteellisissä artikkeleissa ilmeneviin ongelmiin. Toisaalta myös huomioidaan, mitkä näistä ongelmista ovat relevantteja konstruktion kannalta ja millä valinnoilla juuri kyseiset ongelmat ratkaistaan.

Tutkimuksen toteutusvaiheessa otetaan myös kantaa vaihtoehtoisiin tapoihin tuottaa ratkaisun eri osia. Tutkimuksen tuloksena käytännön ratkaisun ohella on uusi tieto, jota servelessistä ja funktioista saadaan käytännön havaintojen kautta suhteutettuna nykytietoon ja kohdealueeseen. Kustannusten osalta on tarkoitus luoda suuntaa antava malli kustannusten arvioimiseksi vastaavissa toimeksiannoissa. Lopuksi pohditaan vielä, miten syntynyttä uutta tietoa voisi yleistää ja tarkastellaan tutkimuksen aikana esiin tulleita jatkotutkimus kysymyksiä ja ideoita.

2.5 Tutkimusyhteistyö kohdeorganisaation kanssa

Tutkimuksen kohdeorganisaatio on ohjelmistoalan yritys Keski-Suomesta. Tutkimuksen kohde liittyy suurempaan kokonaisuuteen, joka on yrityksen kehittämä toiminnanohjausjärjestelmä. Järjestelmää käyttävät asiakkaat tarvitsivat järjestelmään uuden toiminnallisuuden. Toiminnallisuuden suunnittelu, toteutus ja testaus ovat tämän tutkimuksen käytännön ongelma. Toiminnallisuuden vaatimukset esitetään luvussa 3.3.

Tutkijan tulisi tulla ongelman ratkaisuksi muodostetun työryhmän jäseneksi ja molempien osapuolten (tutkija ja kohdeorganisaatio) tulisi olla sitoutuneita tutkimukseen. Olisi myös hyvä sopia heti aluksi etenkin tulosten julkaisemiseen liittyvät ehdot, jotta tutkimuksesta saataisiin tieteellistä kontribuutiota. (Lukka 2001).

Tutkimuksessa tämä toteutui seuraavasti:

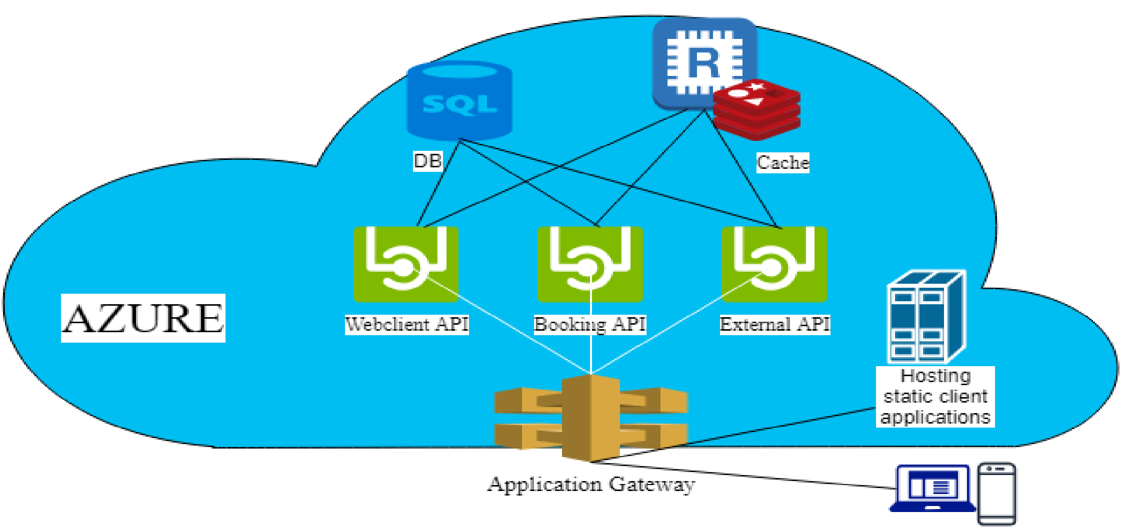
- Tutkija otettiin projektiryhmän jäseneksi.
- Tutkimuksen käynnistyksessä pidettiin palaveri, jossa projektin arkkitehti ja projektipäällikkö lupasivat sitoutua auttamaan tutkijaa hänen työssään, perehdyttämään hänet projektiin sekä auttamaan teknisissä ja käytännön asioissa.
- Tutkija sitoutui saamaan käytännön ongelman ratkaistua määräaikaan mennessä sekä sitoutua projektin toimintatapoihin.
- Tulosten julkisuudesta sovittiin, että tulokset voivat olla käytännön sovelluksen lähdekoodia lukuun ottamatta julkisia. Myös projekti ja sen asiakkaat pidetään anonyymeina.

3 Kohdealueen ymmärrys

Luvun tarkoitus on saada ymmärrystä konstruktiivisen tutkimuksen seuraavan vaiheen, ratkaisun innovoinnin, tueksi. Ymmärrys kohdealueesta hankitaan kahdella tavalla. Ensinnäkin kohdealueen arkkitehtuuri ja ohjelmistokehitysprosessi käydään läpi kohdeorganisaation ohjelmistoarkkitehdin kanssa sekä lähdekoodiin, Readme-tiedostoihin ja Azure-portaalin resursseihin tutustumalla. (Kohdeorganisaation sisäisiä dokumentteja). Näin saadaan pohjatiedot siitä, millaiseen ympäristöön konstruktio rakennetaan. Toiseksi ymmärrystä syvennetään tutustumalla avainaiheisiin tieteellisestä näkökulmasta. Tärkeimmät käsiteltävät avainaiheet ovat serverless sekä DevOps (luvut 4 ja 5).

3.1 Kohdealueen arkkitehtuuri

Kohdealueen arkkitehtuuri on arkkitehtuurityyliltään lähinnä Service Oriented Architecturea (SOA). Alla esitetyssä kuvassa (kuvio 3) määritellään kohdealueen tärkeimmät osat:



Kuvio 3. Kohdealueen arkkitehtuuri

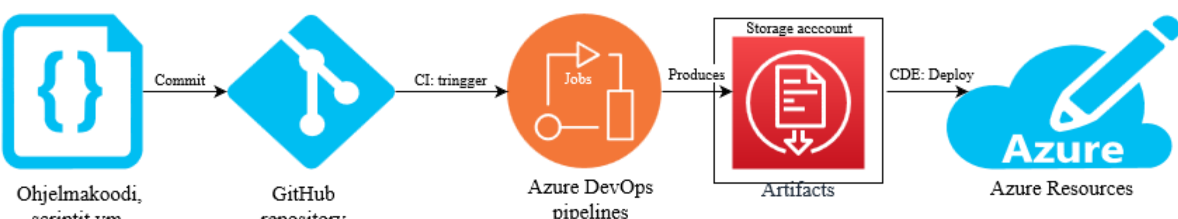
Toiminnanohjausjärjestelmä on Microsoft Azuren pilvipalvelualustaa käyttävä SAAS-sovellus. Järjestelmään kuuluu myös ajanvarauspalvelu, johon tämän tutkimuksen käytännön ongelma liittyy. Ajanvarauspalvelu voidaan toteuttaa upotettuna tai erillisenä selainohjelmana. Molemmissa tapauksissa asiakasohjelma keskustelee Booking APIn (kuvio 3) kanssa.

Sovelluksen asiakasohjelmia kehitetään TypeScript-ohjelmointikielellä Angular-viitekehyksessä (kirjoitushetkellä versio 7). Backend-osuus koostuu MS Sql Server -relaatiotietokannasta, muistinvaraisesta välimuistitietokannasta Rediksestä sekä backend-logiikasta, jonka palveluita tarjotaan kuvan kolmen rajapinnan kautta (APIt). Backend-puoli on ohjelmoitu C#-ohjelmointikielellä ja .NET-viitekehyksellä.

3.2 Ohjelmistokehitys

Kohdeyritys noudattaa Scrum-viitekehystä ja Kanban-metodia projektin ja kehitystyön organisoinnissa. Ohjelmistoa kehitetään ketterästi ja modernisti DevOps-käytänteiden mukaisesti. DevOps on tämän tutkimuksen osalta merkittävässä roolissa. DevOps määritellään ja siihen perehdytään tarkemmin luvussa 4. DevOps on kehitysprosessin perusta ja se on implementoitu Azure DevOps Services -tuotteen avulla. Ohjelmiston kehitystyökalut ovat muutenkin hyvin pitkälti Microsoftin ekosysteemistä. Työkaluihin kuuluvat muun muassa Visual Studio, Azure sekä PowerShell. Dokumentointi on GitHubissa Readme-tiedostoissa ja lähdekoodissa sekä rajapintojen kuvausten osalta Swaggerissa.

Prosessi uuden ominaisuuden koodista tuotteen osaksi kulkee pääpiirteittäin seuraavan kuvion mukaisesti:



Kuvio 4. Kohdeyrityksen CI/CDE-putki

Ensimmäisessä vaiheessa kehittäjä tekee versiohallintana käytettävän Gitin haaraan uuden ominaisuuden ja liittää sen GitHub-palvelun hakemistoon (engl. repository). Sen jälkeen hän ilmoittaa muille kehitystiimin jäsenille muutoksista (engl. pull request), jotta he katselmoisivat muutokset. Katselmoitu koodi liitetään master-haaraan ja samalla Azure DevOps Pipelinessa Continuous Integration (Ks. luku 6.3) -triggeri aktivoituu. Triggeri käynnistää Build-putken, joka koostuu erilaisista ennalta määritetyistä automatisoiduista töistä (engl. jobs). Työt koostuvat muun muassa liitännäisten latauksista, GitHubin työskentelytilan

päivityksestä (engl. checkout), ohjelman kääntämisestä, testeistä ja erinäisten lokien kirjaamisista. Putki tuottaa artefakteja, jotka toimivat lähteenä julkaisuputkelle, joka julkistaa uudet versiot Azuren pilviympäristöön resursseiksi. Julkaisuputken rooli DevOpsin kannalta on Continuous Deployment (CDE), joka esitellään niin ikään luvussa 6.3. Ohjelmistolla ei kirjoitushetkellä ole vielä tuotantoa, joten release-putki puuttuu. Infrastruktuuri ja Azure DevOps Services tukevat teknisesti sen rakentamista CDE-mallin (luku 5.3) mukaan, kun asia tulee ajankohtaiseksi.

3.3 Rakennettavan komponentin vaatimukset

Tilaaja on Ruotsissa toimiva yritys, joka käyttää kohdeorganisaation sovellusta 90 toimipisteessä. Sovelluksen logiikkaa käyttävä WEB-sivusto on kolmannen osapuolen toteuttama.

Rakennettavan komponentin vaatimukset ovat seuraavat:

1. Komponentin tulee lähettää asiakkaalle tekstiviesti- ja sähköpostivahvistus varauksesta.
   1.1. Komponentti saa varauksen tiedot, kuten aika ja vaaditut yhteystiedot, JSON-muodossa. Parametrit ja niiden tyypit on esitetty tarkemmin JSON-Schemana, joka on generoitu esimerkkidatasta osoitteessa https://www.liquid-technologies.com/online-json-to-schema-converter (liite 1).
   1.2. Ajan tulee olla ISO8601-muodossa ja Ruotsin aikaero tulee huomioida toteutuksessa.
2. Komponentin tulee lähettää asiakkaalle muistutusviesti varauksesta. (Ominaisuus on parametrisoitu siten, että sen voi kytkeä pois ja määritellä muistutuksen lähetysajankohdan).
   2.1. Ennen muistutusviestin lähetystä komponentin tulee tarkistaa varauksen tila.
3. Komponentti pitää olla testattu, dokumentoitu ja sen pitää olla osana DevOps-putkea.
4. Viestien kielet määräytyvät parametrina tulevan kielivalinnan mukaan.
5. Viestien lähetyksestä pitää tallentua tieto.
6. Toteutuksen tulee olla valmis elokuun 2019 loppuun mennessä.

4 Serverless-paradigma ja Funktiot palveluina

Luvussa määritellään serverless ja siihen liittyvät olennaiset käsitteet. Ensin perehdytään hieman serverlessin historiaan ja taustalla vaikuttaviin teknologioihin. Sen jälkeen määritellään avainkäsitteet *serverless* ja *Function-As-A-Service* (FAAS). Luvussa myös vertaillaan hieman FAAS-palvelujen tarjoajia sekä pohditaan, miksi serverless on hyvä tapa toteuttaa palveluita ja komponentteja ja toisaalta, mitä ongelmia voi esiintyä. Luvun teoria muodostaa osaltaan tietopohjan sekä mahdollisuuksia ja rajoja implementoinnille.

4.1 Serverlessin määrittelyä ja taustaa

Serverless computingin historia ulottuu aina 1960-luvulle saakka, ja sen syntyyn on vaikuttanut useita paradigmoja, standardeja, protokollia sekä arkkitehtuurityylejä. Näitä ovat muun muassa virtualisointi- ja konttiteknologiat sekä As-A-Service-tyyppiset pilvipalvelut kuten PAAS ja IAAS. Serverlessin kehityksen kannalta on ollut myös olennaista mikroservicejen synty merkittävien internetteknologioiden kuten DNS, SOA, REST ja CGI pohjalta. Lisäksi tapahtumapohjainen arkkitehtuuri, samanaikaisuus (engl. concurrency model) ja ohjelmoinnin työnkulkujen orkestroinnin kehittyminen yhdistyivät serverless-arkkitehtuurille tyypilliseksi tapahtumalähtöiseksi työnkuluksi. (Eyk et al. 2018).

Serverless-laskenta on pilvilaskennan ilmentymä, joka antaa käyttäjälle mahdollisuuden suorittaa tapahtumapohjaisia sovelluksia ilman, että hänen tarvitsisi huolehtia operationaalisesta logiikasta. (Eyk et al. 2018). Serverlessille tyypillisiä ominaisuuksia ovat automaattinen skaalautuvuus sekä pilvipalveluiden tarjoajien ekosysteemien ja konttiteknologioiden suomat edut. (Johnson 2017). Siinä missä PAAS ja IAAS vähensivät tarvetta ostaa ja ylläpitää omia palvelimia, vie serverless abstraktion vielä astetta pidemmälle. Kehittäjän ei tarvitse miettiä infrastruktuuria tai käyttää aikaansa virtuaalikoneen konfiguroimiseen tai siihen, paljonko muistia tai prosessointitehoa hänen pitäisi hankkia. (Savage 2018).

Termi “serverless” (ilman palvelinta) on sinänsä harhaanjohtava, että todellisuudessa teknisesti aina jossain on palvelin. Nykyisin palvelin on monesti virtuaalinen ja pilvipalveluna tarjottu. Pilvipalvelun tarjoajat huolehtivat palvelinten hallinnoinnista. “Serverless”-termin

voikin hyvin määritellä sen käyttäjäroolin kautta. Kehittäjälle eli ohjelmiston rakentajalle kehitystyö on ”serverless”, koska hänen ei tarvitse välittää sovelluksen tai koodiblokin ajamiseen käytetystä laitteesta eikä sen vaatimista resursseista kuten RAM:sta ja CPU:sta. Palveluntarjoajan vastuulle jää se, miten palvelin abstrahoituu käyttäjälle ”serverlessiksi”. (Knorr 2016; Savage 2018; Sewak & Singh 2018).

4.2 Funktiot palveluina

Serverlessin tärkein alakäsite on Function-As-A-Service (FAAS). Funktiot ovat erilaisten asiakkaiden kutsujen prosessointiin tarkoitettuja tapahtumankäsittelijöitä. (Adzic & Chatley 2017). Kehittäjät tekevät itsenäisiä tilattomia funktioita, joiden elinkaari on palveluntarjoajan vastuulla (Eyk et. al. 2018). Johnson (2017) kiteyttää hyvin Serverlesin ja FAASin suhteen: ”Serverless viittaa arkkitehtuuriin, kun taas Function-As-A-Service mekanismiin, jolla kehittäjä implementoi bisneslogiikkaa kyseiseen arkkitehtuuriin”.

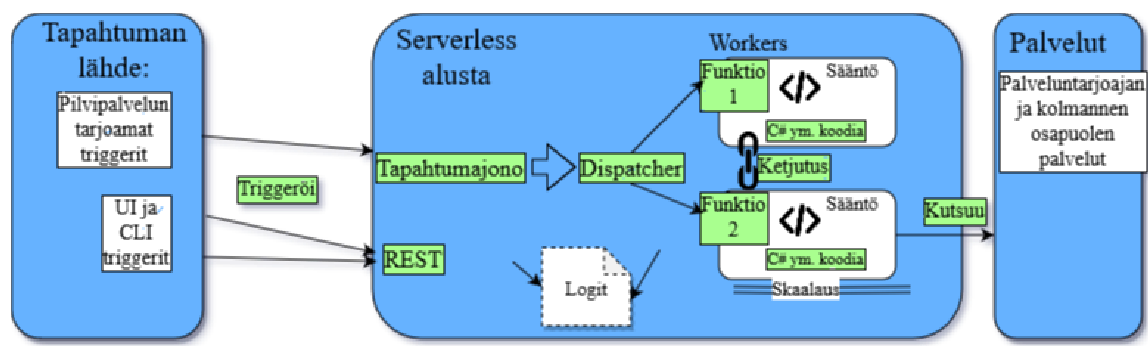
Funktioille tyypillistä on se, että niiden käytöstä maksetaan vain suoritukseen käytetyn CPU-ajan mukaan. Tämä on hyvin erilainen lähestymistapa kuin historiassa aiemmin käytetyt tavat. Perinteisissä malleissa palvelin on (ainakin lähes) koko ajan käytössä ja kulut muodostuvat käyttömäärästä riippumatta. Näin on esimerkiksi ollut vuokratuilla palvelimilla tai pilvipalveluissa, joissa maksu voi olla esimerkiksi tuntiperusteista. (Adzic & Chatley 2017).

Kuten aiemmin mainittiin, mikropalveluarkkitehtuuri on vaikuttanut myös serverlessin syntyyn. Se näkyy hyvin etenkin funktioissa. mikropalvelutekniikka antaa serverlessille konseptin yhdentarkoituksen palveluista, joita voidaan käyttää rajapinnan kautta. Funktioiden voidaan ajatella olevan tämänkaltaisten palveluiden rakennuspalikoita. (Knorr 2016). Yhteys voidaan nähdä myös toisin päin: Serverlessiä voitaisiin ajatella mikropalvelujen_mahdollistajana. Serverlessin itsenäisesti julkistettavat komponentit ja arkkitehtuuri, jossa kehittäjän ei tarvitse huolehtia infrastruktuurista, on hyvä lähtökohta rakentaa kokoelma palveluita bisneksen osa-alueisiin perustuen (mikropalvelut) (Sewak & Singh 2018). Monista itsenäisistä palveluista voi siis rakentaa kokonaisen sovelluksen. Ohjelman kulun ja rakenteen kannalta on merkittävää, että funktioita voi myös ketjuttaa. Esimerkiksi kun jokin tapahtuma laukaisee funktion suorituksen kontissa, voi kyseinen funktio käynnistää toisen funktion

suorituksen (Johnson 2017). Edellä mainituista syistä johtunee myös nimi funktiot palveluina.

4.3 Yleistetty serverless-arkkitehtuuri

Tapahtuman kulku (kuvio 5) alkaa syötteestä (käyttäjän tai palvelun). Syöte toimii triggerinä, joka herättää sääntöperusteiset kehittäjän kirjoittamat funktiot. Tapahtuma menee aina ensin jonoon, josta ne ohjataan sieltä oikealle funktiolle, jolle hoidetaan resurssien allokointi sekä orkestrointi. Funktiot voivat kutsua muita funktioita tai palveluita ja alusta huolehtii myös niiden monitoroinnista ja automaattisesta skaalautuvuudesta. (Sewak & Singh 2018; Baldini et al. 2017).



Kuvio 5. Serverless-alustan yleinen arkkitehtuuri (mukaillen Sewak & Singh 2018; Baldini et al. 2017).

4.4 Serverlessin ja FAASin mahdollisuuksista ja rajoitteista

Yksi serverlessin tavoitteista on se, että kehittäjät voivat käyttää aikansa paremmin kehittämällä bisneslogiikkaa ja edistämällä kilpailukykyä tuomalla nopeammin uusia ominaisuuksia markkinoille. Teknisestä näkökulmasta serverless tarjoaa skaalautuvuutta ja palvelinten provisioinnin automaattisesti. Niin sanottu Pay-As-You-Go-laskutusmalli vähentää kuluja ja lisää tehokkuutta. FAASin eduksi voidaan lukea se, että se mahdollistaa skaalautuvat funktiot, jotka käsittelevät eristettyinä suuria kutsumääriä. Serverlesistä ja funktioista houkuttelevia tekee se, että niiden kautta sovellusta voi helposti laajentaa integroimalla

monipuolisia palveluja. Kattavat integrointimahdollisuudet voivat olla esimerkiksi middlewarea kuten lokien kirjaus, hälytykset ja autentikointi tai vaikka jonkin triggerin laukaisema kutsu tekoälypalveluun. Palveluntarjoajat käyttävät funktioita myös osana muita palvelujaan. (Esimerkkinä mainittakoon Azuren Data Lake Analytics). (Sewak & Singh 2018; Fox et al. 2017).

Sewak ja Singh näkevät rajoitteina muun muassa funktioiden tilattomuuden, kontrollin puutteen, rakeisuuden sekä työkalujen kehittymättömyyden. Funktion lyhyen elinkaaren takia on tilan oltava muualla tallessa. Kontrollin puutteen mainitsee Sewakin & Singhin (2018) ohella myös Back (2018) ja kertoo syyksi funktioiden suorituksessa käytetyn monikerroksisen virtualisoinnin. Rakeisuuden ongelma sen sijaan voi johtaa kulujen kumuloitumiseen ja hankaloittaa integraatiotestausta, jos funktioita on paljon. Työkalujen osalta kaivataan parempia työkaluja muun muassa monitorointiin, debuggaukseen, julkaisuun ja testaukseen. Back mainitsee myös ongelmalliseksi funktioiden laskutusmallin kompleksisuuden ja mahdolliset yllättävät kulut. Mielenkiintoista on myös tutkimustulos (Leitner et al. 2018), josta selvisi, että serverless- ja FAAS-komponenttien rakentaminen vaatii kehittäjältä uudenlaista mentaalimallia, jossa FAAS toimii ikään kuin liimana ja funktionaalinen ohjelmointi sekä mikropalvelut korostuvat. (Sewak & Singh 2018; Back 2018, Leitner at al. 2018). Rajoitteista Fox et al. (2017) mainitsevat muun muassa SLA:n ja standardien puuttumisen. Jossain tapauksissa ongelmalliseksi voi muodostua myös niin sanottu kylmäkäynnistys (engl. cold start). Kylmäkäynnistys tarkoittaa, että funktiot sisältävä kontti ei ole enää kutsun saapuessa käynnissä ja sen seurauksena uudelleenkäynnistämisestä ja funktioiden alustuksesta aiheutuu viivettä (Manner et al. 2018). Lisäksi serverlessin ja funktioiden katsotaan sopivan huonosti pitkään elävien tehtävien suorittamiseen, tietokantoihin ja syväoppimiseen. (Fox et.al 2017).

Rajoitteista luettaessa nousee monesti esille niin kutsuttu vendor-lock, joka tarkoittaa, että ollaan lukittuna yhteen palveluntarjoajaan. Mielestäni se on kuitenkin hyvä esimerkki siitä, mikä toiselle on mahdollisuus ja toiselle rajoite. Palveluntarjoajalle se on hyvä mahdollisuus muiden palveluiden kauppaamiseen, monitorointiin ja tulevaisuuden suunnan näyttämiseen. He päättävät esimerkiksi, mitä valmiita triggereitä tarjotaan ja mitä serverless ja FAAS-palvelut maksavat (Fox et al. 2017).

4.5 Palveluntarjoajien vertailua

Taulukossa 1 on vertailtu suurimpien palveluntarjoajien FAAS-alustojen tarjontaa kirjoitushetkellä. Sewak & Singh (2018) luettelevat funktioiden lisäksi myös muita serverless-määritelmään sopivia palveluita kuten Amazon Kinesis ja Azure Cosmos DB. Tämän tutkimuksen kannalta oleellisessa roolissa ovat funktiot ja niillä rakennettava toteutus. Suurimmiksi palveluntarjoajiksi mainitaan (Sewak & Singh 2018; Baldini et al. 2017; Yegulalp 2017) Microsoft, IBM, Google ja Amazon. Näistä vanhin on Amazonin AWS Lambda vuodelta 2014 (Back 2018). Vaikka edellä mainitut palveluntarjoajat ovat kaikki kaupallisia, löytyy toki myös ilmaisia avoimen lähdekoodin palveluita. Niistä tärkeimmiksi mainitaan Apache OpenWhisk, Fission, OpenLambda, Gestalt ja IronFunctions. (Yegulalp 2017). Näistä etenkin Kubernetisissa ajettava Fission vaikutti mielenkiintoiselta hyvän tarjontansa vuoksi. Open source -vaihtoehdot rajataan tässä kuitenkin pois, sillä konstruktio luodaan osaksi kaupallista ohjelmistoa. Syy rajaukselle on se, että suuret kaupalliset toimijat tarjoavat loppuasiakkaalle tärkeitä asioita kuten tietoturvaa ja SLA:ta. Lisäksi palveluita kehitetään aktiivisemmin ja kaupalliset toimijat todennäköisemmin huolehtivat asiakkailleen korvaavan palvelun, jos tuotteen kehitys loppuu.

	Amazon AWS Lambda	**Microsoft Azure Functions**	**Google Cloud Functions**	**IBM Cloud Functions**
Tuetut ohjelmointikielet	Java, JavaScript, Python, C#	C#, Java, JavaScript, Python, F#	JavaScript, Python ja Go	Suoraan JavaScript, Python, and Swift 4, (Dockerin kautta muitakin)
Hinta	$0.0000166667 $ / GB-sekuntia	0.000016 $ / GB-sekuntia. 0.20 $ miljoonaa suoritusta kohti.	0.4 $ miljoonaa kutsua kohden + 0,00000025 GB-sekuntia kohti + 0,12 $ (networking).	0.000017 $ / GB-sekuntia.

Ilmaista	1 miljoonaan kutsuun asti tai 400,000 GB-Sekuntia CPU-aikaa kuukaudessa.	1 miljoonaan kutsuun asti tai 400,000 GB-Sekuntia CPU-aikaa kuukaudessa.	2 miljoonaa kutsua tai 400,000 GB-sekuntia kuukaudessa.	5 000 000 kutsua (5 sekuntia, 128MB muistilla.
Työnkulku ja ketjutus	Aws step functions	Azure Logic Apps, Durable functions	-	IBM Watson® APIt.
Tukipalveluita	Lokit, monitorointi, tietoturva, AWS palvelut integrointi.	Monitorointi, DevOps pipelinet, CLI, tuki eri IDE:ille, tietoturva, Integrointi Azuren palveluihi, IOT-kehitys ja Kubernetes-tuki	Firebase, GCP, Google Assistant	Kognitiivinen analyysi, Watson, Cloudant (DB), message hub, CLI, monitorointi
Palveluntarjoajan käyttöehdotuksia	Kustomoitu Backend-As-A-Service, stream- ja tiedosto prosessointi.	APIt, web-sovellukset AI: n kanssa, mikropalvelut, koneoppimisen ja datan prosessoinnin työnkulut.	IOT- ja moblile-backendit, reaaliaikainen datan prosessointi. tekoäly.	Serverless- ja mobile backendit, IOT, chatbotit, cognitiivinen datan prosessointi.
Merkittäviä palvelun käyttäjiä	Coca Cola, Thompson Reuters, iRobot	Fujifilm, direct.one, Quest	Smart parking, semios, meetup	GreenQ, Articoolo, SiteSpirit

Taulukko 1. FAAS-palveluntarjoajien vertailua 2019 (https://aws.amazon.com/lambda/features/, https://azure.microsoft.com/en-us/services/functions/, https://cloud.google.com/functions/ ja https://www.ibm.com/cloud/functions. Haettu 28.5.2019)

Perusteet taulukon 1 vertailtaville attribuuteille ovat seuraavat:

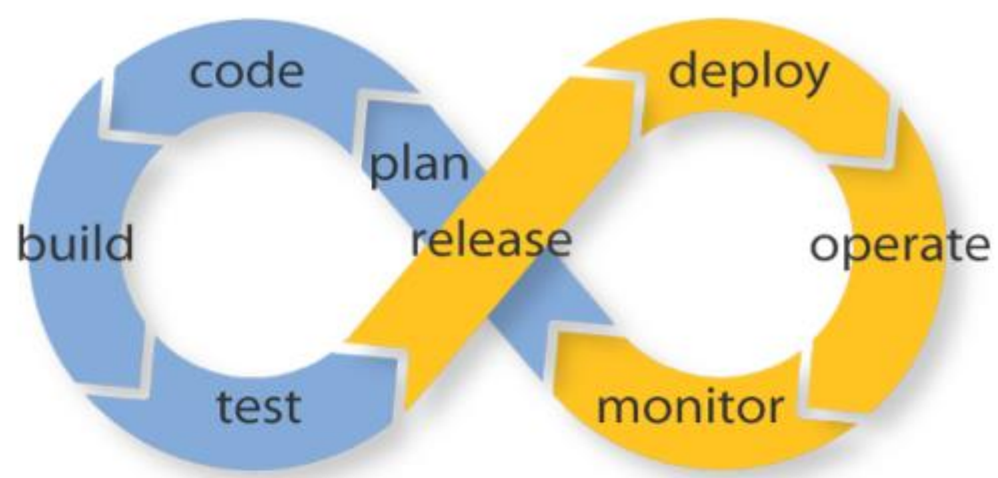
1. Hinta on olennainen attribuutti, sillä tutkimuksessa on tarkoitus tarkastella komponentin vaikutuksia kustannuksiin.
2. Ilmaisten kutsujen tarjonta vaikuttaa hinnan kokonaisuuden muodostumiseen.
3. Tuetut ohjelmointikielet vaikuttavat ylläpitoon ja ratkaisun toteutukseen.
4. Tukipalvelut voivat vaikuttaa ratkaisun toteutukseen ja jatkokehitysmahdollisuuksiin.
5. Käyttöehdotuksista käy ilmi palveluntarjoajan innovatiivisuus, ja niistä voi saada mielenkiintoisia jatkokehitysideoita
6. Merkittäviä palvelun käyttäjiä -attribuutin tarkoitus on toimia referenssinä.

Alaluvun tarkoitus on saada yleiskäsitys palveluntarjoajien tarjonnasta ja huomioida se konstruktion suunnittelussa. Vertailua tehdessä huomasi, että FAAS-palveluihin on panostettu kaupallisesti kaikilla suurilla palveluntarjoajilla. Se näkyi muun muassa hyvänä dokumentaationa, mielenkiintoisina sovellusehdotuksina ja lähes samansuuruisina hintoina. Alan suuret tekijät myös kehuvat kilvan teknologialla saavutettavia etuja ja mahdollisuuksia sekä mainostavat tunnettujen yritysten funktioilla rakennettuja menestystarinoita. Tämän voi hyvin katsoa myös korostavan tutkimuksen ajankohtaisuutta ja tärkeyttä.

5 DevOps

Luvussa lukijalle annetaan käsitys siitä, mitä DevOpsilla tarkoitetaan. Kun termi ja tärkeät käsitteet on määritelty, tarkastellaan hieman, mikä on johtanut DevOpsin syntyyn, mitä ongelmia se ratkaisee ja mitä sillä voidaan saavuttaa. Lopuksi tutustutaan vielä hieman Azure DevOps Serviceen ja siihen, miltä osin sitä on hyödynnetty kohdeorganisaatiossa. Yhdessä edellisen serverlessiä käsittelevän luvun kanssa tämän luvun tehtävä on konstruktiivisen tutkimuksen näkökulmasta saada ymmärrystä aihealueesta teorian kautta.

5.1 DevOpsin määrittelyä ja taustaa



Kuvio 6. Ääretön DevOps-silmukka kuvaa ohjelmistokehityksen ja IT-palvelutoimintojen integroinnin (AppDynamics Inc. 2014)

DevOps on kuin pilvilaskenta (engl. cloud computing) jokin vuosi sitten. Termiä viljellään laajalti niin ohjelmistokehitysyhteisöissä kuin markkinoinnissa. Siksi termille löytyykin useita erilaisia määritelmiä. (Greene 2015). DevOpsissa kahden ohjelmistotuotannon keskeisen osaston toiminnot sulautuvat yhteen. Nämä osastot ovat ohjelmiston kehityksestä vastaava osasto (DEVelopment) ja operaatioista vastaava osasto (OPerationS). Dev-osaston tehtävät on kuvattu kuviossa 6 sinisellä, ja ne koostuvat tyypillisistä iteratiivisen

ohjelmistokehitysprosessin tehtävistä kuten ohjelmoinnista ja testauksesta. Ops-osaston tehtävät (keltaisella pohjalla) tähtäävät muun muassa ohjelmiston julkaisun, konfiguraation ja infrastruktuurin hallintaan sekä tuotteen monitorointiin. (AppDynamics Inc. 2014). Greenen (2015) määritelmän mukaan DevOpsilla tarkoitetaan käytänteitä, työkaluja ja linjauksia, jotka johtavat parempaan laatuun ja automatisoituun ohjelmiston toimitukseen (Automated Delivery). DevOpsille tyypillistä on myös, että asiakkaille toimitetaan laatua uusien ominaisuuksien muodossa lyhyillä sykleillä ja pieninä päivityksinä. DevOpsiin siirtyminen tarkoittaa kulttuurista ja organisatorista muutosta. Sen sijaan, että Dev- ja Ops-osastot toimisivat erillään toisistaan, DevOpsissa kehitys, laadunvarmistus ja operaatiot kulkevat käsi kädessä. (Ebert et al. 2016).

Edellisessä luvussa käytiin läpi, kuinka Serverless-paradigma on ottanut vaikutteita laajalla skaalalla läpi tietotekniikan historian. Samaa voidaan sanoa DevOpsista, jonka taustalla vaikuttavat tietotekniikan tutkimusten lisäksi myös hyväksi havaitut käytänteet teollisuudesta ja johtamisesta. Merkittävät vaikutteensa DevOps on saanut Toyotan tuotannon filosofiaan perustuvasta Lean-ajattelusta (1990) sekä Agile Manifestin (2001) periaatteista. (Kim, Humble, Debois, Willis 2016, 3-10).

Olennaisimmat poiminnat Lean-ajattelusta DevOpsiin ovat arvovirta (engl. Value Stream), Kanban board, laadun varmistus (QA) sekä ajatus pienissä erissä toimittamisesta (small batch sizes). Näistä merkittävin on ehkä arvovirta, joka muovautui DevOpsissa teknologia-arvovirraksi. Sen ydinajatus on, että asiakas saa arvoa vasta tuotannossa olevasta palvelusta, jonka teknologia on mahdollistanut. (Kim et al. 2016, 3-10).

Agilen vaikutus DevOpsiin sen sijaan näkyy pienten tiimikokojen käyttämisessä sekä nopeissa sykleissä uusien, toimivien sovellusversioiden toimittamisessa. DevOpsin kehityksen kannalta huomattavia olivat myös kaksi Agile-konferenssia, joissa syntyivät jatkuvan integraation prosessi (Continuous Integration) sekä jatkuva toimitus konsepti (Continuous Delivery). Muista vaikuttimista mainittakoon vielä operaatioiden automatisointiin pyrkivä Infrastructure-As-Code (IAC) ja Toyotan Improvement Kata, joka keskittyy toiminnan päivittäiseen kehittämiseen ja oppimiseen. (Kim et al. 2016, 3-10). Dörnenburg (2018) lähestyy DevOpsia teknologian roolin kautta. Teknologian rooli on vuosien saatossa siirtynyt tuki-,

ja mahdollistavasta roolista liiketoiminnan osaksi. Hän pitääkin oleellisena, että organisaatioiden tulisi saada sykli uusista bisnesideoista tuotannossa oleviksi tuotteiksi mahdollisimman nopeasti.

5.2 Miksi DevOps?

Nelson-Smith (2017) tunnistaa muun muassa seuraavan ongelman, jota DevOpsilla pyritään ratkaisemaan: Koska ohjelmistojen julkaisusykli on perinteisesti ollut harva, julkaisut aiheuttavat epäluottamusta ja pelkoa. Uusien ominaisuuksien julkaisu ja ongelmien ratkaisu kestävät kauan eikä olla ollenkaan varmoja siitä, miten ohjelma toimii tuotantoympäristössä. Edellisiin kappaleihin peilaten ongelma on suuri, sillä asiakashan saa arvoa vasta tuotannossa olevasta ohjelmasta.

Iteratiivisen ohjelmistokehityksen tavoitteena on ollut, että iteraation päätteeksi ohjelmistosta on uusi tuotantovalmis versio. Se ei ole kuitenkaan aina onnistunut kehitystiimin ja operaatioista vastaavien tiimien toiminnan välisen kuilun vuoksi. (Greene 2015). Kim et al. 2016, 25) toteavat osapuolten toiminnasta löytyvän myös ristiriidan. Kehitystiimi pyrkii toimittamaan nopeasti ominaisuuksia muuttuviin kilpailullisiin markkinoihin, kun Ops-puolen huolenaiheena on tarjota turvallinen, vakaa ja luotettava palvelu. Nelson-Smith (2017) näkee siiloutumisen johtavan kommunikaation ja kokonaiskuvan puutteeseen sekä ongelmien turhaan siirtelyyn. Lisäksi siiloutuminen johtaa liian manuaalisen työn kautta hidastamaan uuden version kulkua asiakkaalle. Myöskin tietoturva, testaus ja ongelmien korjaaminen ovat perinteisesti olleet prosessissa liian myöhään. (Kim et. al. 2016, 22).

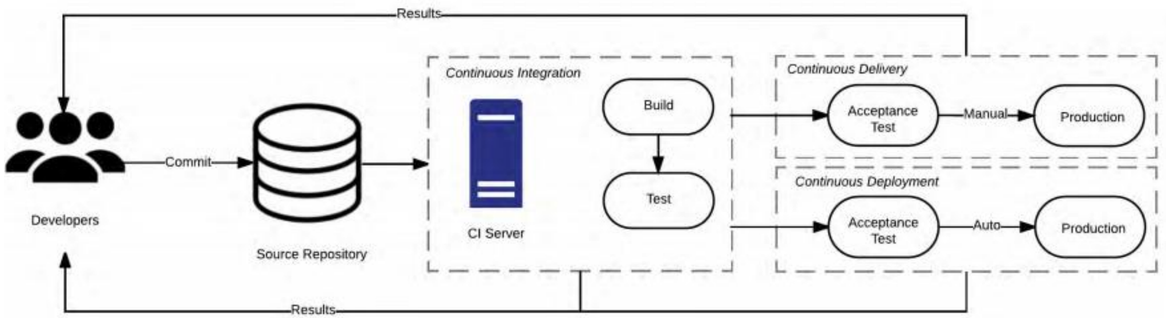
DevOpsin periaatteet, joilla muun muassa edellä kuvattuja ongelmia pyritään ratkaisemaan, on tiivistetty niin sanottuun kolmeen tiehen. Kolmen tien periaatteita noudattamalla saadaan hyötyä asiakkaalle ja kehitysorganisaatiolle. Periaatteet ovat virta (engl. flow), palaute sekä jatkuva oppiminen ja kokeilu. Flow-periaatteessa virta kulkee kehittäjiltä operaatiolle ja samalla arvo yritykseltä asiakkaille. Olennaista on keskeneräisten töiden [WIP], odottelun ja toimitusviiveiden minimointi sekä se, että tiimeillä on mahdollisuus kehittää, testata ja julkaista ketterästi ja laadukkaasti muutoksia tuotantoon. Palaute-periaatteen voi katsoa myös olevan eräänlainen virta. Siinä pienet toimituserät ja automaatio mahdollistavat nopean

palautteen ja informaation saannin tuotannosta ja kokonaisuudesta. Näin virheet pysyvät myös pieninä ja helpommin havaittavina, korjaustyötä on vähemmän ja virheistä opitaan jatkuvasti suhteellisen halvalla. Kolmas periaate tähtää siihen, että organisaatiossa olisi jatkuvan oppimisen ja kokeilemisen kulttuuri. Koska isoissa monimutkaisissa järjestelmissä väistämättä tapahtuu virheitä, virheet tulisi hyväksyä, sillä ne tarjoavat erinomaisia tilaisuuksia, kokeilla, oppia ja parantaa päivittäistä työtä. (Kim et al. 2016, 15-46).

Asiakkaalle hyödyt näkyvät muun muassa laatuattribuuttien kuten ohjelmiston turvallisuuden, laadun ja luotettavuuden kasvuna. Kehitysorganisaatiolle DevOps-periaatteiden noudattaminen johtaa markkina-aseman vahvistumiseen, tuottavuuden kasvuun sekä organisaatiossa tapahtuvaan oppimiseen. Markkina-aseman vahvistumista edesauttaa etenkin se, että automatisoinnilla, sopivan väljällä arkkitehtuurilla ja nopeilla esteistä vapailla kehityssykleillä pyritään kehittäjien tuottavuuden lisäämiseen. Kehittäjät voivat siten keskittyä uusien arvoa tuovien bisnesideoiden innovointiin ja testaamiseen asiakkailla. (Kim et. al. 2016. 23, 348).

5.3 Continuous-käytänteet

Lukiessa Agilesta ja DevOpsista ei voi olla törmäämättä erilaisiin Continuous-käytänteisiin, kuten Continuous Integrationiin, Deliveryyn ja Deploymentiin, joiden avulla pyritään kiihdyttämään ohjelmiston kehitys- ja julkaisuprosessia unohtamatta laatunäkökulmaa. Alla olevasta kuvasta (kuvio 7) ilmenee hyvin tyypillinen tapa toteuttaa Continuous Software Engineeringiä, johon käytänteet lukeutuvat:



Kuvio 7. CI-, CD- ja CDE-tehtävät (Shahin et al. 2017).

Continuous integration luo perustan Continuous Deliverylle (CD) ja Continuous Deploymentille (CDE), jotka taas ovat vaihtoehtoisia tapoja viedä muutoksia tuotantoympäristöön. CI:ssä kehittäjien muutoksia integroidaan jatkuvasti versionhallinnasta ja muutokset käännetään ja testataan automaattisesti. CDE:n ja CD:n ero on siinä, että CDE:ssä pyritään kokoaikaiseen julkaisuvalmiuteen, mutta bisnes määrittää manuaalisesti, mitä ja milloin julkaistaan. CD:ssä sen sijaan tuotantoon vienti laukeaa (engl. triggers) muutosten varastoinnista Git-hakemistoon (engl. commit), ja se on täysin automatisoitua. Continuous-käytänteitä käytetään muun muassa build- ja testaustulosten nopeuttamiseen ja näkyvyyden lisäämiseen, automatisoinnin tukemiseen ja vikojen ja konfliktien löytämiseen ajoissa. (Shahin et al. 2017). Kuvio 7:ää on hyvä verrata kohdeorganisaation prosessiin (kuvio 4), jossa näkyy esimerkki, miten continuous-käytänteitä voidaan implementoida. Kuviossa 7 olevat tulokset (results) voidaan ajatella olevan DevOpsin toinen periaate, palautevirta (ks. edellinen alaluku) käytännössä.

Kuriositeettina mainittakoon, että continuous-käytänteitä on toki muitakin, kuten esimerkiksi Continuous Testing, Inspecting ja Security. Yksi mielenkiintoisimmista ja vain vähän huomiota saaneista käytänteistä on Continuous Maintenance (CM), joka tavallaan laajentaa CI:tä ja CD/CDE:tä. Pang ja Hindle (2016) määrittelevät CM:n olevan ”Continuous-prosesseja, jotka huolehtivat kehitys- ja operaatioartefaktien ylläpidosta”. Käytännettä toteutetaan esimerkiksi automaation, yhteenvetojen ja arkistoinnin avulla. (Pang & Hindle 2016).

5.4 DevOps-työkalut ja niiden käyttö kohdeorganisaatiossa

Seuraavissa alaluvuissa tutustutaan hieman yleisempiin DevOps-työkaluihin ja siihen, miten niitä on hyödynnetty kohdeorganisaatiossa.

5.4.1 DevOpsissa käytetyt työkalut

Ebert et al. (2016) korostavat, että DevOps-työkalut tulee aina räätälöidä kohdeorganisaation tarpeisiin nähden. Työkaluja voidaan luokitella DevOpsin eri vaiheiden mukaan. Ebert et al. (2016) jakavat työkalut Build-työkaluihin, CI-työkaluihin ja operaatioiden työkaluihin. Työkalujen tarkoitus on automatisoida manuaalisia tehtäviä ja mahdollistaa siten nopea

työnkulku ja julkaisusykli. Build-työkaluilla suoritetaan muun muassa kääntämiseen, testaukseen, riippuvuuksien hallintaan ja dokumentaation luontiin liittyviä tehtäviä. CI-työkaluilla testataan integroitavaa osaa yhdessä kokonaisuutena. Tunnetuin työkalu tähän tarkoitukseen on Jenkins automation server. Operaatioihin liittyvät työkalut sisältävät muun muassa lokien kirjaus-, monitorointi- sekä julkaisutyökaluja. (Ebert et al. 2016). DevOpsissa infrastruktuuria kohdellaan datana, toisin sanoen koodina tiedostoissa. Konfigurointiin voidaan soveltaa siten ohjelmistokehityksen parhaita käytänteitä kuten TDD:ia. Lisäksi kaikki installointi ja konfigurointi hoidetaan käyttäen samaa versionhallintaa kuin ohjelmiston lähdekoodissa. (Johann 2017).

Information week (2017) listaa kymmenen kategoriaa työkaluille ja toteaa, että DevOpsin laajuuden takia yksi työkalu ei yleensä pysty vastaamaan kaikkiin tarpeisiin. Ebertin ja hänen tutkimusryhmänsä mainitsemien työkalujen lisäksi Information weekin listauksessa on mukana myös yhteistyö- ja konttityökalut. Tutkimuksen kannalta mielenkiintoista on, että myös Serverless-työkalut mainitaan. Devops-työkalujen palveluntarjoajien työkalut perustuvat useimmiten avoimeen lähdekoodiin ja ne on tarjottu pilvipalveluina. Kohdeorganisaation käyttämän Microsoft Azure DevOps Servicen kilpailijoita lähes vastaavalla tarjonnalla ovat muun muassa AWS, Puppet, IBM, Oracle ja XebiaLabs. Yksittäisten tehtävien tarjonnasta mainitaan esimerkiksi Atlassin (yhteistyö), Logz.io (lokit) sekä Nagios (monitorointi). (Information week 2017).

5.4.2 DevOps-työkalujen käyttö kohdeorganisaatiossa

Kohdeorganisaatiossa DevOps-käytänteiden työkaluina käytetään pääosin Microsoftin Azure DevOps Servicen palveluja. Azure DevOps Services on alusta, joka tarjoaa DevOps-työkaluja palveluina pilvessä tai on-premisena. Siinä on myös kattavat integrointimahdollisuudet (esimerkiksi Azuren palvelujen ja konttiteknologioiden kanssa) sekä visualisointiin tarkoitettu kustomoitava dashboard (Microsoft 2019). Kohdeorganisaatiossa käytetään DevOps servicen tarjoamista palveluista Azure Pipelinesia CI- ja CDE-putkien rakentamiseen sekä Azure Artifactsia pakettien hallintaan. Lisäksi kohdeorganisaatio on integroinut DevOps Servicen GitHubin kanssa. DevOps Servicen Boards-, Repos- ja Testplan-palveluita

ei ole ainakaan toistaiseksi hyödynnetty. Projektinhallinta on hoidettu Jiran kautta ja testit ajetaan automaattisesti komentorivikäskynä osana CI-putkea.

6 Ohjelmiston ylläpito

Tutkimuksen yhtenä tavoitteena oli arvioida serverless-paradigmalla toteutetun komponentin vaikutuksia ylläpitoon. Jotta vaikutuksia voitaisiin arvioida, on hyvä ensin määritellä, mitä ylläpito on.

IEEE:n standardi 610.12.1991 määrittelee ohjelmiston ylläpidon seuraavasti: ”Ylläpito on tuotantoon viennin jälkeistä ohjelmistojärjestelmän tai -komponentin muokkaamista, jonka tarkoitus on korjata vikoja, parantaa suorituskykyä, adaptoida uusia ominaisuuksia tai mahdollistaa uudessa ympäristössä toimiminen.” (IEEE 2005).

Ohjelmiston ylläpito voidaan jakaa perinteiseen korjaavaan ylläpitoon sekä ohjelmiston evoluutioon liittyvään ylläpitoon. Korjaava ylläpito on monesti käyttäjien bugiraporttien pohjalta tulevaa ohjelmiston muuttamista vikojen (kuten suunnittelun, logiikan ja koodin) korjaamiseksi. Evoluutioon liittyvä ylläpito voidaan jakaa ehkäisevään, adaptoivaan ja perfektiiviseen ylläpitoon. Ehkäisevän ylläpidon tarkoitus on tehdä ohjelmistosta jo mahdollisimman aikaisessa vaiheessa ylläpidettävä esimerkiksi dokumentoinnin, koodikommenttien ja huolellisen suunnittelun avulla (kuten strukturoinnin ja modulaarisuuden). Adaptoiva ylläpito sen sijaan koskee ympäristön muutosta, ja sitä voidaan auttaa monitoroinnin avulla. (Huom. Ympäristön muutos voi olla myös bisnessääntö- ja policy-muutoksia.). Ohjelmiston toimituksen jälkeen tulee myös usein uusia vaatimuksia ja käyttötapauksia, jotka vaativat ohjelmiston muuttamista. Tästä käytetään nimitystä perfektiivinen ylläpito. (Erdil et al. 2003).

Ylläpidolla yleisellä tasolla pyritään siihen, että ohjelmistoa on helppo muuttaa ja ymmärtää. Huonosti ylläpidettävä ohjelmisto kuluttaa resursseja ja onkin tutkittu, että noin 50 % ohjelmistokehityksen kustannuksista tulisi juuri ylläpidosta. Ylläpitoon vaikuttavia seikkoja ovat muun muassa ohjelmiston kompleksisuus ja koodin määrä, ohjelmointikieli sekä henkilöstön tietämys ja vaihtuvuus. Ketterien kehitysmenetelmien käyttö on luontaisesti hyvä keino lisätä ylläpidettävyyttä niiden iteratiivisen ja inkrementaalisen luonteen vuoksi, sillä kuten edellisessä luvussa todettiin, tulevat muutokset pieninä testattuina paloina toimivaan ohjelmistoon. Automaatio ja järjestelmän osien ymmärrys suhteessa ylläpidettävyyteen ovat oleellista kokonaisylläpidettävyyden kannalta (Erdil et al. 2003)

7 Toteutuksessa huomioitavaa

Luvussa luodaan joidenkin edellisissä luvuissa nousseiden seikkojen perusteella raamit konstruktion toteutukselle sekä tuodaan esiin kysymyksiä, joihin implementoinnissa pitää vastata. Tarkoitus on luoda suuret linjat ratkaisun tueksi, kun taas toteutusluvussa 8 perustellaan valinnat teknisemmin ja yksityiskohtaisemmin.

7.1 Kohdealueen analysoinnin pohjalta

Kohdealueen ymmärryksen pohjalta on luontevaa lähteä rakentamaan komponenttia Azure funktioilla ja C#-ohjelmointikielellä. Kielivalintaa perustelee se, että ohjelmiston backend-osuus ja API:t ovat jo rakennettu kyseisellä kielellä. Ylläpidon kannalta on hyvä asia, ettei ohjelmointikielten määrä projektissa lisäänny. Ohjelmiston muiden kehittäjien on siten helpompi ylläpitää komponenttia jatkossa.

Palveluntarjoajien vertailusta selvisi, että hinnoissa ei juuri ole eroa, kun taas tuetuissa ohjelmointikielissä ja tukipalveluissa oli. Palveluista Azure ja AWS tukivat suoraan C#-kieltä. Azure Funktioiden valintaa perustelee Azuren muiden palveluiden hyödynnettävyys, monipuoliset integraatiomahdollisuudet ja se, että sovellus on valmiiksi Azuren pilviympäristössä. Azure on palveluntarjoajana hyvä valinta, sillä kohdeorganisaatio on Microsoftin kumppani, Azure on käytössä myös muissa projekteissa ja organisaatiosta löytyy Azure-osaamista ennestään. Kohdeorganisaatio saa siten tutkimuksesta lisätietoa Azuresta Serverlessin osalta. Asiaa voisi katsoa myös siltä kannalta, että ei löytynyt myöskään hyvää syytä olla valitsematta Azurea.

7.2 Ylläpidettävyyden huomiointi

Luvussa 4 ylläpito jaettiin korjaavaan ja evoluution ylläpitoon. Evoluution kannalta ylläpidettävyyteen voidaan toteutuksessa vaikuttaa preventiivisen ylläpidon kautta kiinnittämällä huomiota dokumentointiin ja kommentointiin. Lisäksi perfektiivisen ylläpidon voi huomioida tekemällä komponentista helposti muutettavan, laajennettavan sekä yleiskäyttöisen. Uudelleenkäytettävyys on tärkeää, jotta komponentti ei jäisi yhdentarkoituksen

toteutukseksi. Uudelleenkäytettävyys lisää myös konstruktion hyödyllisyyttä ja kuten luvussa 2 todettiin, hyödyllisyys on yksi konstruktiivisen tutkimuksen validiuteen vaikuttava tekijä. Adaptiivista ylläpitoa ei ole mahdollista huomioida tutkimuksessa, koska ympäristö ei ole vaihtumassa. Simuloiminen rajataan ajan ja työn laajuuden vuoksi pois, mutta jatkotutkimusmahdollisuuden se toki tarjoaa.

7.3 Vaatimuksista

Vaatimuksien perusteella huomioitava seikka on muun muassa se, miten lähtökohtaisesti tilaton funktio käsittelee muistutusviestin lähettämisen. Toinen käytännön ongelma on löytää sopivat E-mail- ja SMS-palveluntarjoajat viestien lähettämistä varten, kun huomioidaan kustannukset ja ylläpidettävyys. Funktioiden testaus on tunnistettu ongelma myös kirjallisuudessa ja siihen koitetaan löytää mahdollisimman hyvä ratkaisu. Dokumentointi tehdään versiohallintaan omana Readme-tiedostona. Myös komponentin kutsurajapinta dokumentoidaan, jotta komponentin käyttäjät voivat helposti käyttää komponenttia oikealla kutsulla.

7.4 DevOpsin huomiointi

DevOpsin kannalta konstruktiossa tärkeintä on huomioida toimivien kokonaisuuksien palastelu ja prosessin käyttöönotto heti prototyypin jälkeen. Näin saavutetaan DevOpsin etuja kuten oppiminen ja se, että toiminnallisuutta päästään nopeasti testaamaan tuotantoympäristöä vastaavassa ympäristössä Azuressa. Lisäksi näin voidaan myös testata inkrementaalisesti korjaavaa ja adoptoivaa ylläpitoa. DevOpsin inkrementaalisesta ja iteratiivisesta luonteesta on myös se hyöty, että konstruktiota luodessa voidaan kokeilla eri toteutusratkaisuja ja saada palaute nopeasti ja hylätä toimimattomat toteutusratkaisut.

Käytännössä DevOpsin huomiointi tarkoittaa muun muassa työn jakamista Jiran boardille tehtäviksi sekä CI-putken ja infrastruktuurin automaation (IAC) luomista heti alkuvaiheessa. Toteutuksesta tehdään oma Git-hakemisto (engl. repository), jotta komponentti olisi mahdollisimman helppo ottaa käyttöön myös muissa projekteissa. Konstruktion osana syntyvä toteutusratkaisu implementoidaan ensisijaisesti kohdeorganisaation käyttämään Azure DevOps -palveluun. Koska Azure DevOpsissa on hyvät integrointimahdollisuudet ja

kohdeorganisaatiossa ei ole käytössä läheskään kaikki DevOps-työkalut, voidaan kuitenkin tarpeen vaatiessa ja perustellusti ottaa tulevaisuudessa mukaan myös muiden toimijoiden yksittäisiä palveluja täydentämään tarjontaa. Näitä voisivat olla muun muassa analysointi, lokien kirjaus ja monitorointipalvelut. Monitorointi Devopsin kohdalla tarkoittaa putken, ei sovelluksen monitorointia. Nämä kaksi tulee ymmärtää omina kokonaisuuksinaan.

7.5 Kustannuksista

Kustannusarvio (pohjautuen liitteeseen 2) muodostui kolmessa osassa. Ensimmäisen osion tarkoitus oli selvittää sopiva palveluntarjoaja sähköposti- ja tekstiviestivahvistusten lähetykseen. Palvelun tarjoajaksi valikoitui palvelu nimeltä Mailjet. Kustannuslaskelmaa laatiessa huomattiin myös tekstiviestien suuri kuukausikustannus arvioidulla käyttömäärällä ja päätettiin optimoida kustannuksia tekemällä tekstiviestin lähetyksestä valinnainen parametri. Kululaskelma lähetettiin tämän vaiheen osalta asiakkaalle hyväksyttäväksi ja ehdotettu palveluntarjoaja hyväksyttiin.

Toinen osio syntyi, kun konstruktiota kehittäessä huomattiin, että Service Bus -tuote sopisi Storage Queueta paremmin muistutusviestien lähetyksen arkkitehtuuriin. Kustannukset arvioiduilla käyttömäärillä jäivät hyvin alhaisiksi molemmilla. Service Bus päätettiin ottaa mukaan tuomaan etua ylläpidettävyyteen ja vähentämään jonoihin liittyvien häiriöiden mahdollisuuksia vähentämällä operaatioita.

Kustannuslaskelman viimeinen osio, jossa arvioidaan funktioiden käytön kustannuksia, voitiin tehdä vasta komponentin valmistuttua. Kun valmista komponenttia ajetaan Azuresta, saadaan dataa funktioiden suoritusajoista ja muistinkäytöstä laskutoimitusta varten. Siinä missä kaksi ensimmäistä osiota vaikuttivat luotavaan konstruktion toteutukseen, on kolmannen osion tarkoitus ennemminkin antaa sidosryhmille arvio funktioiden kuluista ja avata laskutusmalli jatkohyödyntämistä varten.

8 Innovointi ja konstruktion toteutus

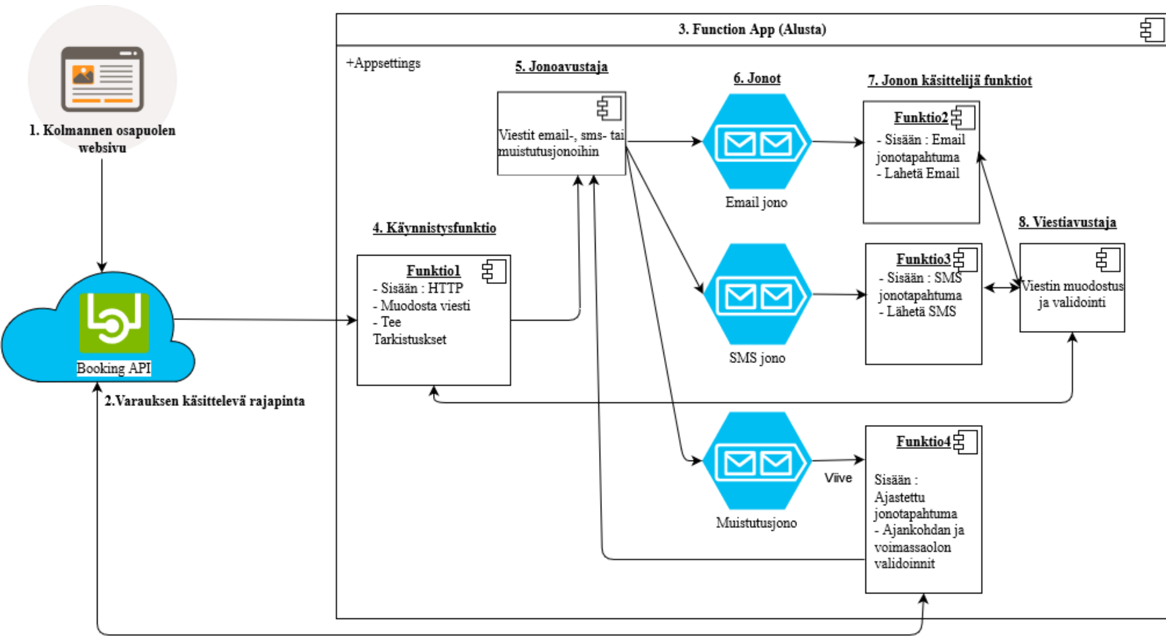
Luvussa esitellään ensin ennen varsinaista toteutusta tehty Proof-Of-Concept (POC) ja sen vaikutukset konstruktioon. Sen jälkeen kuvataan innovoitu toteutuksen arkkitehtuuri, jonka ratkaisut käydään perustellen läpi. Konstruktio koostuu kolmesta osasta: 1) asiakkaan ongelman ratkaiseva serverless-komponentti, 2) Komponentin integrointi DevOps-putkeen, 3) Kustannuslaskelma. Lähdekoodi ei ole julkista kohdeyrityksen toiveesta, mutta osassa kohdissa on tarjottu havainnollistavia yleisiä esimerkkejä.

8.1 Proof-Of-Concept

Azure funktioiden (versio 2) C#-toteutukset käyttävät .NET Core -viitekehystä. Siitä seuraa, että kehitystyössä voi hyödyntää helposti IDE:ien .Net-työkaluja ja luoda komponentin rakenteen kuin tekisi perinteisempää sovellusta. Lisäksi Microsoft tarjoaa funktioille oman komentorivityökalun (Azure functions core tools) sekä emulaattorin, joiden avulla funktioita voidaan ajaa ja debugata paikallisesti. Ennen varsinaista konstruktiota tehtiin POC ikään kuin keskustelun tueksi. POCissa selvisi hyvin esimerkiksi puutteita kutsun tarvitsemissa parametreissa sekä se, minkälaista validointia tarvitaan. POCissa luotu toteutus oli yksinkertainen sähköpostin lähettävä funktio, jossa kehittäjälle tuli tutuksi funktioiden toimintalogiikka ja syntaksi. POC loi myös pohjan konstruktion rakentamiselle ja onnistuessaan lisäsi kohdeyrityksen luottoa lopullisen toteutuksen onnistumiseen.

8.2 Toteutuksen arkkitehtuuri osana kokonaisarkkitehtuuria

Alla olevassa kuviossa (kuvio 8) näkyy suunnitelma arkkitehtuurista, jonka pohjalta konstruktiota lähdettiin rakentamaan. Seuraavissa alaluvuissa käydään perustellen läpi sen tapahtumankulku.



Kuvio 8. Toteutusratkaisun arkkitehtuuri osana kokonaisarkkitehtuuria

8.2.1 Tapahtumankulku

Sovelluksen tapahtumankulun käynnistää uusi varaus varauspalvelussa (kuvio 8, kohta 1), josta lähtee viesti Booking APIlle (kohta 2). Rajapinnan HTTP-asiakas muodostaa tapahtumasta JSON-scheman mukaisen Post-viestin (liite 1). Viesti käynnistää Function appissa (kohta 3) käynnistysfunktion (kohta 4). Funktio validoi syötteet käyttämällä avustajaluokkaa (kohta 8) sekä tarkastaa, onko tekstiviestin lähetys käytössä ja muistutukset aktivoituina. Jos kaikki on kunnossa Käynnistysfunktio palauttaa käyttäjälle 200 OK -viestin, muutoin funktio palauttaa 400-viestin, jonka sisältönä on validoinnin virheet. Käynnistysfunktio pyytää jonoavustajaa asettamaan vahvistusviestit jonoihin (kohta 6). Tekstiviesti asetetaan jonoon

vain, jos kutsuja on aktivoinut toiminnallisuuden. Jos taas muistutukset on aktivoitu, asetetaan tapahtuma myös muistutusjonoon, jonka viesti on viivästetty parametrissä annettuun ajankohtaan.

E-mail- ja SMS-jononkäsittelijäfunktiot (kohta 7) laukeavat uuden viestin saapuessa jonoon. Funktiot muodostavat asiakkaalle lähetettävän viestin viestiavustajaluokan (kohta 8) avulla ja käyttävät Mailjet-palvelun rajapintoja tekstiviestin ja sähköpostiviestien lähetykseen. Muistutusfunktio laukeaa ajastettuna ja kutsuu Booking APIa RestSharp-kirjaston asiakkaan avulla tarkastaakseen, onko varaus vielä voimassa. (Huom. kehitysvaiheessa käytettiin Logic Appsilla tehtyä endpointia validointiin). Jos varaus on voimassa, funktio kutsuu jonoavustajaa, joka asettaa viestit e-mail- ja SMS-jonoihin ja tapahtumankulku jatkuu kuin vahvistusviestin kohdalla. Lopullinen viestin muoto määräytyy viestissä määritetyn kielen ja IMessageHelper-rajapinnan toteuttavan FormMessage()-metodin avulla.

8.2.2 Arkkitehtuurin perustelua

Käynnistysfunktio käynnistää tapahtumaketjun. Funktio on toteutettu HTTP-triggerillä, jolloin sitä voidaan helposti kutsua tuotannossa Booking APIlta sekä kehitysvaiheessa esimerkiksi Postman-sovellusta käyttäen. Alustana käytetty Function app luodaan storage accountin kanssa, joka tarjoaa sisäänrakennetun jonomekanismin. Azuren tarjontaan kuuluu myös Durable Functions, joka on palvelu työnkulkujen orkestrointiin ja pitkäkestoisten tehtävien ajamiseen. Toteutus haluttiin rakentaa puhtaasti perinteisiä funktioita hyödyntäen, jotta serverless-arkkitehtuurin mekanismit tulisivat ymmärrettyä paremmin. Microsoft Azuren dokumentaatiosta myös selviää, että pinnan alla palvelu käyttää yhtäällä Storage queueita.

Ratkaisun kannalta olennainen osa on viestinvälitysarkkitehtuurityylin hyödyntäminen. Arkkitehtuurityyli valittiin, sillä jonot tarjoavat mahdollisuuden viivytettyyn viestin välitykseen. Toinen syy valintaan on se, että niiden avulla arkkitehtuurista saatiin push- eikä pull-tyylinen. Vaihtoehtona olisi ollut esimerkiksi varastoida varaukset ja tietyin aikavälein kysyä muistutettavia varauksia. Edellä mainittu tapa olisi tuonut lisäkustannuksia varastoinnin ja lisääntyneiden funktiokutsujen myötä. Viestinvälitykseen käytettiin varausvahvistusten osalta Storage queueita ja muistutusten osalta Service Bus-jonomekanismia.

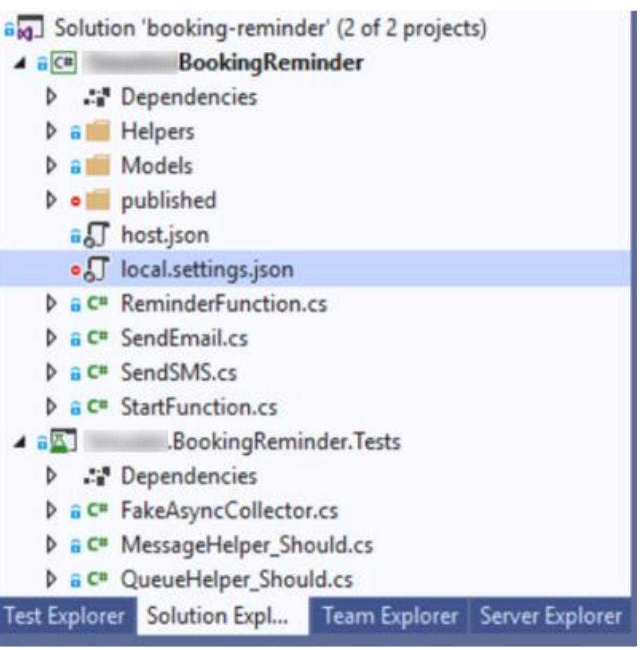
Azure tarjoaa syötteen sitomismekanismin injektoimalla funktion käynnistävään Task run -metodiin jonototeutuksen ja sitä käyttävän viestin kerääjän:

```
[Queue("sendemail-queue")] IAsyncCollector<QueueItem> emailQueue
```

Käyttämällä IAsyncCollector-rajapintaa voidaan viestit kerätä ja toimitus jonoon hoitaa asynkronisesti vastaanottajasuorituksen jatkuessa.

Storage queueita olisi voinut käyttää myös muistutuksien rakentamiseen, mutta niissä on tällä hetkellä seitsemän päivän maksimi viestin elinajassa. Service Bussissa rajoitusta ei ollut, ja vaikka se onkin hieman kalliimpi (ks. kustannuslaskelma, liite 2, osio 2), vähentää se tarvittavan koodin määrää tuoden parempaa ylläpidettävyyttä ja toimintavarmuutta. Storage Queuet olisivat tarvinneet logiikan, jossa ne laitetaan aina tarpeen vaatiessa uudelleen jonoon. Lisäksi kutsujen määrä olisi kasvanut.

Arkkitehtuurin yhtenä kantavana ajatuksena on vastuiden jako funktioiden ja apuluokkien välillä.



Kuvio 9. Komponentin ohjelmallinen rakenne

Kuten aiemmin mainittiin, on serverlessin ajatus tarjota yhden tarkoituksen palveluja. Lisäksi hyvään olio-ohjelmointiin kuuluu SOLID-periaatteiden (Martin, 2002) mukaisesti

yhden vastuun periaate (engl. single responsibility). Vastuunjaosta ja rajapinnoista saadaan useita hyötyjä. Ensinnäkin koodista tulee helppolukuisempaa ja siten ylläpidettävämpää. Toiseksi funktioita laskutetaan kutsumäärien sekä ajoon käytetyn ajan perusteella. Jokaisella funktiolla on myös tietty määrä ilmaista suoritusaikaa. Käyttämällä apuluokkia ja jonoja saadaan funktioille selkeät vastuut ja yhden funktion suoritusaikaa pienennettyä. Lisäksi konstruktion aikana tuli ilmi, että apuluokkien avulla saatiin vähennettyä merkittävästi toistoa funktioiden kesken. Kuviosta 9 näkyy vastuunjaon lisäksi myös Azure funktioilla (v2) rakennetun sovelluksen rakenteen samankaltaisuus perinteisen .Net Core -sovelluksen kanssa.

Toinen merkittävä hyöty apuluokkien käytöstä saadaan sovelluksen testauksessa. Microsoftin dokumentaatio suosittelee, että serverless-funktioiden bisneslogiikka eristetään ja testit kirjoitetaan eristetyille luokille, joita funktiot käyttävät (kuviossa 9 Helpers-kansio ja Booking.Reminder.Tests-projekti). Jos logiikka olisi funktioissa, olisi testien kirjoittaminen huomattavasti haastavampaa, sillä funktiot ovat staattisia ja niistä ei voi siten luoda uutta instanssia. Apuluokat mahdollistavat funktion rajapintaparametrien kuten IAsyncCollector ja ILoggerin korvaamisen mock-toteutuksilla. Funktion tärkeä logiikka tulee testattua eikä samalla tarvitse tehdä testejä varten maksullisia kutsuja, jotka kuluttavat oikeita resursseja Azuressa. Testauksessa käytettiin MockQ-kirjastoa sekä Nunit-viitekehystä.

Konstruktion varmatoimisuuden takeena testien ohella ovat monitorointi, validointi ja virheenkäsittely olennaisilta osin. Monitorointiin käytetään Microsoftin ILogger-laajennosta.

```
log.LogInformation("New booking received");
```

Yllä olevan esimerkin mukaisesti lokeihin kirjatut viestit näkyvät paikallisesti konsolissa ja funktioittain Azuressa. Koska sovelluksessa. johon komponentti integroidaan, on käytössä Azuren monitorointipalvelu Application Insigths, oli luontevaa hyödyntää sitä myös Function appissa. Tämä liitos tehtiin function appin luovassa ARM-konfiguraatiossa (ks. luku 8.3). Application Insights osaa valmiiksi tunnistaa ILoggeriin kirjatut merkinnät (LogInformation ja LogError). Käynnistysfunktion syötteen validoiva metodi tehtiin apuluokkaan. Jos validointi ei mene läpi, loppuu käynnistysfunktion suoritus, eikä se enää kutsu muita funktioita. Validoinnissa on huomioitu parametrit, joiden virheelliset arvot estävät muiden

funktioiden oikeanlaisen toiminnan. Validoinnin virheet kerätään listaan alla olevaan tapaan ja palautetaan post-kutsun lähettäjälle.

```
if (string.IsNullOrWhiteSpace(validoitavaParanmetri)) errors.Add("virheviesti");
```

Muistutusfunktiossa oleva tarkistus BookingAPIlta sen sijaan estää muistutusviestin lähetyksen henkilöille, joiden varaukset eivät enää ole voimassa. Validointi säästää rahaa vähentämällä turhia funktiokutsuja.

Virheenkäsittely huomioitiin eritoten puutteellisten konfiguraatioarvojen osalta. Jos esimerkiksi API-avain puuttuu, heittää Funktio4 poikkeuksen eikä edes yritä jatkaa suoritusta. Storage queueista on myös hyödynnetty ominaisuutta, jossa poikkeus viestin vastaanottajalla johtaa viestin asettamiseen uudelleen jonoon. Ominaisuudesta on konfiguroitavissa toistojen määrä sekä toistovälin pituus. Ominaisuus on erinomainen, sillä jos esimerkiksi poikkeuksista tehdään jatkossa hälytys, voidaan vika selvittää ja korjata, ja viestit vain hieman viivästyvät katoamisen sijaan

Preventiivisestä ylläpidosta huolehdittiin lähdekoodin puolella metodien kommentoinnilla ja yleisellä tasolla dokumentoimalla function appin käyttö GitHubin Readme-tiedostoon. Readme-tiedostossa (liite 3) kerrotaan muun muassa, miten function appia kehitetään, ajetaan, konfiguroidaan sekä monitoroidaan. Koska function appilla on vain yksi Post-kutsuja vastaan ottava metodi, ei erillistä API-dokumentointia tarvittu, vaan se otettiin osaksi Readmetä. Lisäksi Azure Funktiot eivät tällä hetkellä tue kohdearkkitehtuurissa käytettyä Swagger-sovellusta.

8.3 Toteutus osana DevOps-putkea

Komponentin vaatimus oli, että se pitää olla osana kohdesovelluksen DevOps-putkea. Seuraavista alaluvuista selviää, miten vaatimus toteutettiin.

8.3.1 Infrastruktuurin luonti Azureen

Infrastruktuuri luodaan DevOps-periaatteiden mukaisesti koodin avulla (IAC). Sovelluksessa, johon konstruktio implementoidaan, on olemassa jaetut resurssit, joita kaikki tulevat

instanssit käyttävät. Azuressa on käytössä resurssiryhmät, joihin resursseja luodaan. Alun perin oli tarkoitus laittaa function app jaettujen resurssien kanssa samaan resurssiryhmään, mutta selvisi että tällä hetkellä Azuressa ei ole mahdollista samassa resurssiryhmässä olla kahta erityyppistä Service Plania (laskutusmallia). Tästä syystä function appille päätettiin luoda oma. Toisaalta ratkaisu on selkeämpi, sillä funktion appin resurssit ovat nyt omana kokonaisuutenaan.

Function appin resurssit luodaan kuitenkin samaan aikaan jaettujen resurssien kanssa PowersShell-skriptissä, kutsumalla sen sisällä skriptiä DeployFunctionAppInfra.ps1 alla olevan komennon avulla. Skripti voidaan myös ajaa itsenäisesti.

```
$functionAppNameObject = & "$PSScriptRoot\scripts\DeployFuncti-
onAppInfra.ps1" -ReminderResourceGroup $reminderResourceGroup -AiInstance
$aiKeyOutput
```

Skripti ottaa parametriksi monitorointiin käytettävän Application Insightsin sekä resurssiryhmän, johon se luodaan. Skriptin olennaiset osat ovat Service Busin ja muistutusjonon luonti sekä function appin luonti edellä mainittuja parametrejä hyödyntäen.

```
$functionApp = RunArm -file $parent\arm\reminder-app.json -parameters @{
        aiKey = @{ value = $AiInstance }
        sbConnection = @{ value = $connectionString }
    }
```

Yllä esitetyssä function appin luovassa komennossa on käytetty ARM-templaatteja. Komennossa parametrina oleva JSON-tiedosto sisältää muun muassa konfiguraatiot, kuten tarvittavat API-avaimet ja Zipdeployn enabloinnin. RunArm-metodissa on käytetty Azure Resource Manager (AzRM) Cmletiä resurssien julkaisuun. Infrastruktuurin ja resurssien julkaisemiseksi Azureen on useita tapoja kuten Azuren Az Cli -komentorivityökalu, Azure function core toolsin työkalut tai päivitys REST-APIn avulla. AzRm ja templaatit valittiin ylläpidettävyyden vuoksi, sillä niitä on käytetty kohdesovelluksen muissa osissa. Kehittäjät eivät siten suotta joudu opettelemaan montaa tapaa ja asentamaan useita työkaluja kehityskoneilleen.

8.3.2 CI- ja CDE -putket

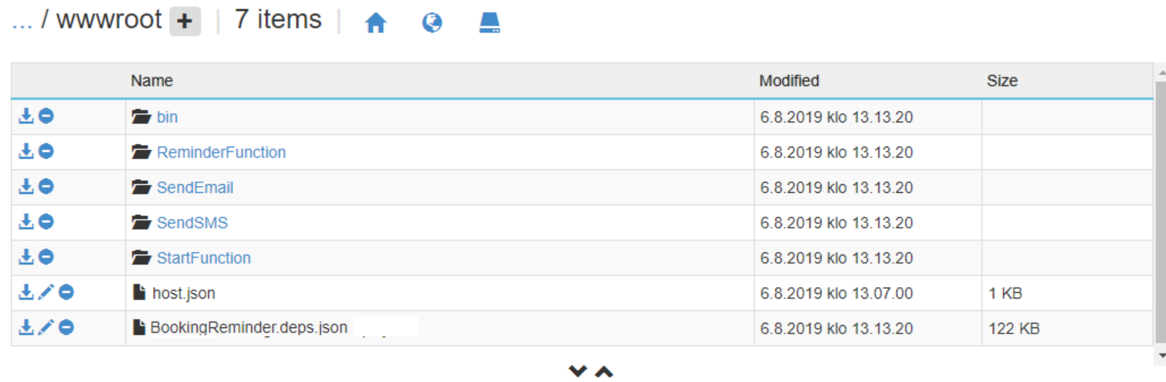
Function appille luotiin oma CI-putki, jonka tarkoitus on mahdollistaa iteratiivinen ja inkrementaalinen sisällön lisääminen function appiin (kuten funktiot ja niiden käyttämät apuluokat, korjaukset ja muutokset). CI-putki on Azure Devopsissa, ja se käyttää versiohallinnassa olevaa YAML-tiedostoa build-tehtävien suoritukseen.

YAML-tiedoston ajo käynnistyy aina, kun Azure DevOps havaitsee uuden pull requestin tai kun Master-haaraan liitetään uusia muutoksia versiohallinnassa (merge). YAML-tiedossa on määritelty tehtävät: käännä, julkaise, testaa, pakkaa ja vie artefaktiksi Azuren Blob storageen. Build- ja test-tehtävät ajetaan YAMLista käyttäen .Net CLIn dotnet -komentoja (build, test, publish) ja niiden onnistuminen on edellytys myöhemmille tehtäville. Paketoinnista tehdään mergen yhteydessä myös ”viimeisin”-tagilla merkitty paketti, jota hyödynnetään seuraavassa vaiheessa. CI-putken tuloksia voidaan hyödyntää julkaisuskriptissä, joka ajetaan automatisoidusti tai komentoriviltä manuaalisesti. Tässä vaiheessa siirrytään continuous deploymentiin, jolla tarkoitettiin jatkuvaa julkaisuvalmiutta.

Julkaisuskriptissä AzRm:ää hyödynnetään hakemalla Azuren blob storagesta edellisen CI-putken tuottama ZIP-paketti käyttämällä komentoa Get-AzStorageBlobContent. Sen jälkeen paketin sisältö julkaistaan Rest-kutsutapaa käyttäen alla olevan koodiesimerkin mukaisesti. Valinta johtuu siitä, että AzRm Cmletistä ei vielä kirjoitushetkellä löydy toteutusta funktioiden julkaisulle ja muut edellisessä alaluvussa esitellyt tavat ovat poissuljettuja samoista kyseisessä luvussa mainituista syistä.

```
Invoke-RestMethod -Uri $apiUrl -Headers @{Authorization=("Basic {0}" -f
$base64AuthInfo)} -UserAgent $userAgent -Method POST -InFile $filePath -Con-
tentType "multipart/form-data"
```

Function appiin on konfiguroitu ”run-from-package”-moodi, joka tarkoittaa, että funktioille on portaalissa vain lukuoperaatiot sallittuja ja ne ajetaan suoraan Kudu-enginen paketista niiden saadessa herätteen. Function appin alustapalvelu Kudusta voidaan myös tarkastella julkaistuja resursseja (Kuvio 10 seuraavalla sivulla.)



Kuvio 10. Julkaistut funktiot resursseina Azuren Kudussa

Lopuksi mainittakoon, että sekä infrastruktuurin luonnissa että julkaisuputkessa on pyritty yhtenäiseen tyyliin kohdesovelluksen kanssa. Lisäksi skriptien toteutukset on implementoitu käyttäen Microsoftin dokumentaatiota apuna (docs.microsoft.com).

8.4 Kustannusten muodostuminen

Luvussa 7.5 kustannuslaskelmien (liite 2) osiot käytiin läpi siitä näkökulmasta, miten ne vaikuttavat konstruktion syntyyn. Tässä luvussa laskelman sisältö avataan tarkemmin sekä esitetään siitä saadut tulokset.

Kulut voidaan jakaa Azuren ulkopuolisiin kuluihin sekä serverless-toteutukseen suoraan liittyviin kuluihin. Toteutuksesta riippumattomia kuluja ovat e-mail- ja SMS-palveluntarjoajien viestin lähetyksestä aiheutuvat kulut. Työntekijä ja muut ohjelmistokehitysprojektin normaalit kulut rajataan tässä tutkimuksessa pois. Serverless-toteutukseen suoraan liittyvät kulut muodostuvat funktioiden kutsuista sekä oheispalvelujen käytöstä (tässä tapauksessa Service Bus ja Storage queue). Epäsuorasti kuluihin liittyy myös monitorointi Application Insightsilla, joka ei näissä arvioissa myöskään ole mukana. Palvelu on muutenkin käytössä kohdesovelluksen jaetuissa resursseissa, ja sen kulut muodostuvat käytön mukaan. Lopulliset todelliset kulut selviävät vasta, kun toteumaa tarkastellaan esimerkiksi vuoden päästä käyttöönotosta. Silloin nähdään muun muassa, kuinka paljon keskiarvoisesti kutsuja on tullut ja miten esimerkiksi monitorointipalvelua on hyödynnetty.

Loppuasiakkaalta saadun arvion pohjalta kuukausittainen varauksien määrä on yleensä 60 000 – 80 000 varausta. Laskettaessa pieni turvamarginaali (10 000 varausta) saadaan viestien maksimimääräksi 180 000 viestiä kuussa (sis. 90 000 varausvahvistusta ja 90 000 muistutusta). Näitä lukuja on käytetty kaikkien osioiden laskelmissa.

8.4.1 Osio 1: SMS- ja e-mail-palveluntarjoajat

Ensimmäisen vaiheen vertailussa selvisi, että Azuren valmiit integraatiopalvelut olivat kalleimmat sekä SMS- että e-mail-palveluissa. SMS-palveluista halvin oli Mailjet ja e-mail-palveluista SendInBlue. E-mail-palvelujentarjoajista Mailjet oli kuitenkin paremmin integroitavissa Nuget-pakettiasennuksen ja .Net Core -tuen vuoksi. Mailjet on siten ylläpidollisesti parempi valinta, vaikka onkin hieman kalliimpi. Lisäksi ratkaisulla sekä sähköposti- että tekstiviestipalvelu tulisivat samalta tarjoajalta, ja näin saavutettaisiin lisää sujuvuutta ylläpitoon ja laskutukseen.

Halvimmilla valinnoilla kulut ovat seuraavat:

SMS (Mailjet): 180 000 * 0,380 SEK = 68 400 SEK = (6405,66 €) kuukaudessa.

E-mail (SendinBlue): 1097,66 SEK (103,20 €) kuukaudessa.

Jos otetaan E-mail tarjoajaksi Mailjetm saadaan hinnaksi: 1200,63 SEK (112,46 €) kuukaudessa.

Yhteensä SMS ja e-mail -palvelut otettuna Mailjetilta maksavat 68400 SEK + 1200,63 SEK = 69 600,63 SEK (6518,099 €) kuukaudessa. (Yläkanttiin arvioiduilla käyttäjämäärillä sekä skenaariolla, jossa kaikki haluaisivat muistutukset ja SMS-viestit.)

8.4.2 Osio 2: Jonojen kustannukset

Jonojen (Service Bus ja Storage queue) laskutus perustuu perusoperaatioihin, joista konstruktiossa relevantteja ovat jonoon asetus ja jonosta lukeminen. Jonojen kustannuslaskelmista opittiin, että jonopalveluiden käyttö on lähtökohtaisesti hyvin pieni kustannus riippumatta palvelusta (alle 10 senttiä kuukaudessa tarkastelluilla kutsumäärillä). Siksi

kannattaakin valita palvelu, jolla logiikka saadaan yksinkertaisemmaksi ja mahdollisesti karsittua funktiokutsujen määrää ja suoritusaikaa pohtimatta hintaa sen enempää.

8.4.3 Osio 3: Funktiot

Funktioiden kustannukset muodostuvat funktioiden suoritukseen allokoidusta muistista, suorittamiseen kuluneesta ajasta sekä funktiokutsujen määrästä. Kun kerrotaan suoritukseen käytetty muisti ja aika, saadaan mittayksikkö GB-sekunti. (ks. taulukko 1). (Bach 2018).

Kululaskelmia (liite 2, osio 3) varten valmista komponenttia kutsuttiin Postman-sovelluksella siten, että tekstiviestit ja muistutusviestit olivat mukana. Azuren portaalin monitorointityökaluilla saatiin suoritusten käyttöaikoja funktioittain sekä function appin keskimääräinen muistinkulutus. Muistinkulutus pyöristettiin lähimpään 128 megatavuun, jolloin arvoksi saatiin 256 megatavua (Azuren laskutustapa). Funktioiden suoritusajoista laskettiin keskiarvot. Laskelmista paljastui, että niin suoritusyksiköt (78 930 GB-s) kuin kutsumäärätkin (540 000) jäivät ilmaisten rajojen sisään. (400 000 GB-s ja miljoona kutsua). Azuren dokumentaatiosta huomiona vielä todettakoon, että storage accountista ja verkon käytöstä saattaa tulla hieman lisäkuluja. Yhteenvetona kustannuksista todettakoon, että suurimmat kulut tulevat SMS- ja e-mail-palveluista, kun taas Azuren palvelut ovat näillä käyttömäärällä vain minimaalinen kustannus.

9 Konstruktion toimivuus ja sovellusala

Konstruktiivisen tutkimuksen validiteetti perustuu konstruktion hyödyllisyyteen kohdeorganisaatiossa ja sen ulkopuolella. Validiteetti voidaan jakaa sisäiseen ja ulkoiseen validiteettiin. Sisäinen validiteetti näkyy konstruktion toimivuudessa, joka on saatu aikaan empiirisesti teoreettisen ymmärryksen pohjalta. Ulkoinen validiteetti sen sijaan tulee esille konstruktion sovellettavuuden kautta. (Rautiainen et al. 2017)

9.1 Konstruktion hyödyllisyys ja markkinatestit

Rakennettua konstruktiota voidaan pitää onnistuneena, sillä se täytti sille asetetut vaatimukset (ks. luku 3.3). Konstruktion avulla ratkaistiin käytännön ongelma, joka konstruktion oli tarkoituskin ratkaista. Tilaajan edustajat (projektipäällikkö ja arkkitehti) olivat tyytyväisiä ratkaisuun ja hyväksyivät sen. Kasanen et al. (1993) toteavat, että organisaation hyväksyttyä konstruktio, voidaan sillä katsoa olevan käytännöllistä arvoa. Tällöin myös konstruktio on läpäissyt niin kutsutun ensimmäisen markkinatestin. Konstruktiiviseen tutkimukseen kuuluvien toisen ja kolmannen markkinatestin läpäisy ei tämän tutkimuksen aikarajoissa ole mahdollista, sillä ne vaatisivat taloudellisen hyödyn tutkimista ympäristössä, johon se on luotu, ja myöhemmin myös organisaation laajuisesti.

Konstruktion hyödyllisyyttä voidaan tarkastella moneltakin kannalta. Ilmeisin niistä on organisaation saama hyöty käytännön toteutuksesta. Hyöty ilmenee muun muassa rahallisena arvona, jonka organisaatio saa esimerkiksi palvelun kehityksestä ja ylläpidosta. Samalla myös kohdesovelluksen asema vahvistuu kaivatun lisäpalvelun kautta. Konstruktion hyödyllisyys voi myös kasvaa jatkossa.

Epäsuorempi hyöty organisaatiolle on uusi tieto serverlessistä. Tutkimuksen jälkeen organisaatiolla on serverlessillä ja funktioilla rakennettu komponentti, josta voidaan ottaa oppia, kun seuraavan kerran kohdataan ongelma, johon kyseinen paradigma ja mekanismi voisivat tarjota ratkaisuja. Tutkimus toi organisaatioon lisätietoa muun muassa monista huomioitavista seikoista liittyen ohjelmointikäytänteisiin, prosessiin ja arkkitehtuuriin käytettäessä serverlessiä ja funktioita yleisellä tasolla sekä etenkin Microsoftin työkaluilla Azuren

ympäristössä. Konstruktiota voidaan tulevaisuudessa hyödyntää suunnittelun referenssinä, sillä siitä saa suuntaa antavan käsityksen huomioitavista kuluista sekä toimivista ratkaisuista. Toisaalta jos suunnittelussa analysoi konstruktion, voidaan välttyä samoilta ongelmakohdilta, joita tutkija kohtasi. Näitä olivat muun muassa ongelmat resurssiryhmien kanssa sekä ensiksi valitun viestinvälitysmekanismin rajallisuus.

Konstruktion hyödyllisyyttä voi tarkastella myös tieteellisestä näkökulmasta. Konstruktio oli hyödyllinen, koska siitä syntyi tieteellistä kontribuutiota teorian ymmärryksen ja käytännön toteutuksen aikana. Jos tutkimus olisi osoittanut serverless-paradigman huonoksi vaihtoehdoksi ratkaista ongelma tai jos ratkaisu ei olisi onnistunut, olisi konstruktiivinen tutkimus silti ollut hyödyllinen. Se olisi tuonut arvokasta tietoa organisaatiolle ja tiedekenttään siitä, mikä ei toimi.

9.2 Soveltamisala

Konstruktion yleistettävyyttä ja sovellusalan laajentamista on huomioitu jo toteutusta suunniteltaessa (muun muassa luku 7.2). Rajapinnoitetut apuluokat bisneslogiikalle, omat CI- ja CDE-putket ja ympäristöluontiskriptit, dokumentointi ja selkeät vastuut luokille tekevät konstruktiosta helposti muunnettavan ja lisäävät yleiskäyttöisyyden mahdollisuutta. Lisäksi tärkeimmät asiakaskohtaiset konfiguraatiot (kuten palveluiden rajapinta-avaimet) ovat yhdessä templaatissa, joka ajetaan ympäristön luonnin yhteydessä. Jatkokehitys ja ylläpito on tulevaisuudessa helppoa. Kehittäjän täytyy vain ajaa parametrisoidut infrastruktuurin luontiskriptit, ottaa klooni projektista koneelle ja tehdä haluamansa muutokset. Sen jälkeen automaatio validoi muutoksen ja vie sen tuotantoon.

Konstruktion käytännön sovelluksen hyödyllisyys voi kasvaa tulevaisuudessa esimerkiksi komponenttia jatkokehittämällä tai uusien asiakkuuksien ottaessa sen käyttöön. Itsenäinen komponentti on myös helposti integroitavissa muille liiketoiminta-alueille, joissa vastaavanlaista muistutuspalvelua tarvitaan. Komponenttia voi käyttää mistä tahansa, mistä voi lähettää HTTP-post-kutsun, ja sen viestien sisältöä ja toimintalogiikkaa voidaan muovata toteuttamalla IMessageHelper-rajapinta uudelleen. Käytännön toteutuksen ohella sovellusalan voisi katsoa koskevan myös konstruktiivisessa tutkimuksessa syntynyttä tietoa.

Kustannuslaskelmat ja prosessista opittu tietous ovat tämän tutkimuksen konstruktion osa, jota voidaan hyödyntää yleistettynä tulevaisuudessa arvioitaessa serverless-toteutuksien kustannuksia, ylläpidettävyyttä, mahdollisuuksia ja rajoitteita.

Oyegoken (2011) mukaan konstruktiiviseen tutkimukseen kuuluu myös oleellisesti se, että konstruktion adoptoinnin mahdollisia rajoitteita pohditaan. Ensimmäiseksi mieleen tulee muutosvastarinta. Toisaalta kohdeorganisaatiossa käytetään jo DevOpsia ja kiinnostus serverlessiinkin on nousussa. Lähtökohdat ovat siten hyvät. Kasanen et al. (1993) korostavatkin, että konstruktion tulosten on oltava merkityksellisiä, yksinkertaisia ja helppokäyttöisiä. Edellä mainittujen määreiden täyttyminen voisi hyvinkin vähentää muutosvastarintaa. Merkityksellisyys konstruktiossa tulee esille ainakin siinä, että se todistaa serverless-paradigman sopivan hyvin tämän tutkimuksen ongelman kaltaisten ongelmien ratkaisemiseen. Teorialuvuista voi huomata, että serverlessillä ja DevOpsilla on osittain samat päämäärät. Kehittäjät pääsevät mahdollisimman paljon keskittymään uusien bisnesideoiden kehittämiseen ja luomaan arvoa asiakkaalle. Tämän yhtäläisyyden luulisi kiinnostavan myös liiketoiminnasta vastaavia henkilöitä.

Toinen konstruktion adoptioon vaikuttava seikka on tietouden leviäminen organisaatiossa. Jotta tutkimuksesta opittu tietotaito leviäisi läpi organisaation ja konstruktiota hyödynnettäisiin laajemmin muissakin projekteissa, tulee tietoutta levittää. Tietoutta voisi levittää esimerkiksi pitämällä aiheesta demotilaisuus organisaatiolle ja tiedottamalla, jos toteutuksesta saadaan hyvää palautetta asiakkailta. Yksinkertaisuuden ja helppokäyttöisyyden huomiointi on oleellista jatkokehittäjien ja ylläpitäjien kannalta. Niiden huomioimisen konstruktiossa todistaa hyvä dokumentointi sekä viimeisen luvun esimerkki korjaavan ylläpidon tehtävästä, joka tutkimuksessa tuli esille.

Valittu palveluntarjoaja sekä teknologiavalinnat voivat myös vaikuttaa laajempaan adaptointiin jatkossa, jos komponentti halutaan ottaa kehitykseen toiseen projektiin, jossa on käytössä eri valinnat. Muutokset ovat toteutettavissa, mutta niiden huomiointi ja tutkiminen rajataan tämän tutkimuksen ulkopuolelle. Rautiainen et al. (2017) toteavat konstruktiivisen tutkimuksen relevanssin ja validiuden lopullisen tilan selviävän vasta ajan kuluessa. Sama pätee varmasti soveltamisalaan. Näkisin, että jos konstruktion adoptiolta poistetaan esteitä, se

edesauttaisi konstruktion hyödyllisyyden lisääntymistä sekä toisen ja kolmannen vaiheen markkinatestien läpäisemisen mahdollisuutta.

10 Teoreettinen kontribuutio ja pohdinta

Lukan (2001) mukaan konstruktiivisessa tutkimuksessa teoreettista kontribuutiota voidaan saavuttaa kahdella tavalla. Ensinnäkin kontribuutiota voidaan saada nykytietoon sillä lisäyksellä, jonka konstruktio toimiessaan itsessään tuottaa. Tässä korostuu tutkimuksen empiirinen luonne ja keino - lopputulossuhteet. Toisena tapana kontribuution saavuttamiseksi Lukka (2001) mainitsee tiedon, jota saadaan konstruktiivisesta tutkimusprosessista, kun tarkastellaan rakenteita ja riippuvuussuhteita. Tutkimus voi myös synnyttää uusia potentiaalisia mielenkiintoisia riippuvuussuhteita havaittujen lisäksi.

Tutkimuksen tavoitteena oli tutkia serverless-paradigmalla ja funktioilla rakennetun komponentin soveltuvuutta SAAS-sovelluksen laajentamiseen. Näkökulmina olivat kehitysprosessi (DevOps), ylläpito ja kustannukset. Toinen tutkimuksen tavoite oli havainnoida teorian ja konstruktion kautta serverlessin mahdollisuuksien ja rajoitteiden nykytilaa. Tutkimuksessa tietämys lisääntyi läpi tutkimusprosessin ja havaintoja syntyi sekä teorian ymmärryksestä ja empiriasta. Konstruktio peilattuna teoriaan vahvisti osaa tunnistetuista serverlessin ongelmista ja rajoitteista. Tutkimus osoitti osan aiemmista kirjallisuuden tiedoista myös vanhentuneeksi. Lisäksi löytyi myös uusia rajoitteita ja potentiaalisia riippuvuussuhteita sekä mahdollisuuksia.

10.1 Serverlessin ja FAASin mahdollisuuksien ja rajoitteiden peilaus tutkimustuloksiin

Ensimmäinen havainto tuli ilmi palveluntarjoajien vertailussa (luku 4.5). Taulukosta huomaa, että osaan aiemmin mainituista serverlessin ongelmista on hyvin vastattu palveluntarjoajien alustoilla ja tukipalveluilla. Nämä käsitykset vahvistuivat käytännöntoteutusta rakentaessa. Palveluntarjoajaksi valittiin Azure. Palveluista käytettiin function appia, v2-funktioita sekä lisäpalveluina Service Busia, Application Insightsia ja Storage Queueta. Näillä työkaluilla yhdessä Visual Studion kanssa poistuivat muun muassa debuggaukseen, monitorointiin ja tilattomuuteen liittyvät rajoitteet. Jonot ratkaisivat ainakin osittain tilattomuuden ja pitkäkestoisten tehtävien rajoitteen, kun taas storage-emulaattori debuggauksen

ongelmallisuuden. Lisäksi Application Insights yhdessä function appin kanssa tarjosivat hyvät työkalut monitorointiin ja ylläpitoon.

Vendorlock-rajoitetta on mielestäni jopa tietoisesti vahvistettu. Palveluntarjoaja tarjoaa suuren joukon lisäpalveluita, integrointimahdollisuuksia ja ylläpidettävyyttä tarjonnalla, joka kattaa lähes koko ohjelmistokehitysekosysteemin (työkalut ja prosessit kehittämiseen, testaamiseen, julkaisuun ja ylläpitoon). Näin palveluntarjoajat tarjoavat käyttömukavuutta ja ylläpidettävyyttä ja sitouttavat käyttämään heidän tuotteitaan. Kaupallisissa ratkaisuissa SLA:kaan ei enää ole ongelma, sillä ainakin Azure lupaa kirjoitushetkellä 99.95%:n palvelutason. (Microsoft, 2018). Vendor-lockissa ongelmallista on palveluntarjoajan kontrolli hinnoista ja tarjottavista palveluista. Palveluntarjoaja voi myös lopettaa palveluitansa tai mennä konkurssiin. Tähän riskiin konstruktiossa vastattiin asettamalla bisneslogiikka erillisiin luokkiin, jolloin siirrettävyys toiselle palveluntarjoajalle helpottuisi funktioiden logiikan pysyessä yksinkertaisena. Jatkossa jos siirrettävyyttä haluttaisiin kehittää, voitaisiin kokeilla funktioiden julkaisua esimerkiksi Dockerin kontteihin, jolloin niitä voisi ajaa useammasta ympäristöstä. Vendor-lockin hyväksymällä saa paljon etuja, kunhan tiedostaa riskit.

Luvussa 4 laskutusmallin todettiin olevan kompleksinen ja serverlessin kulujen laskeminen haasteellista. Funktioiden kustannusten muodostuminen GB-sekuntien ja kutsumäärien summana vaati jonkin aikaa perehtymistä. Etukäteisarviointi onkin haastavaa varsinkin, jos tulevia kutsumääriä sekä funktioiden suoritusaikoja ei voida arvioida. Kustannusten auki laskeminen on kuitenkin mielestäni erittäin tärkeää ja opettavaista, sillä siinä tulee samalla löydettyä funktioiden pullonkaulat muistin käytön ja suoritusajan osalta. Näin tarjoutuu mahdollisuus optimointiin. Suoritusaikoja tutkimalla näki myös selkeän eron käynnistys- ja muistutusfunktioiden sekä viestinvälitysfunktioiden välillä. Jos aikaa oli kulunut komponentin kutsujen välillä, huomasi ensin mainituissa moninkertaisen suoritusajan. Syynä tähän lienee juuri luvussa 4 esitelty kylmäkäynnistys. Kylmäkäynnistyksen välttämiseksi on jonkin verran tutkimusta (esim. Lin & Glikson 2019). Tämän tutkimuksen kannalta se ei kuitenkaan ole suuri ongelma, sillä sekuntitason viivästys vahvistusviestin saapumisessa käyttäjille ei haittaa.

Tutkimuksen aikana ilmeni myös, että serverless-paradigmalla ja pilviympäristössä ohjelmistoja kehittäessä perinteiset oliosuuntautuneen ohjelmistokehityksen hyvät käytänteet pätevät ja parantavat ylläpidettävyyttä. Rakentamalla järkevä arkkitehtuuri ja tuntemalla käyttäjän tarpeet voidaan säästää turhissa funktiokutsuissa. Konstruktiossa kutsujen määrää pienennettiin muun muassa parametrien validoinnin ja SMS-lähettämisen valinnaisuuden avulla. Toinen tärkeä havainto on mentaalimalli, jonka Leitner et al. (2018) mainitsevat sekä uudenlaiset taidot, joita kehittäjä tarvitsee. Täydentäisin heidän näkemystään edelleen seuraavalla huomiolla: Kehittäjän pitää tuntea pilvialustojen lainalaisuuksia, mutta toisaalta serverlessin ansiosta hänen ei tarvitse välttämättä tuntea niin hyvin ohjelmistoa, johon komponenttia integroidaan. (Esimerkiksi ohjelmiston sisäiset riippuvaisuudet (engl. dependencies), se miten kohdeohjelmistoa ajetaan, tai mihin kohtaan uusi komponentti tulisi sijoittaa.) Tutkimuksessa kävi ilmi, että itsenäistä komponenttia rakennettaessa tärkeintä oli tietää rajapinnat, kohdealueen yleinen arkkitehtuuri sekä käytetyt teknologiat ja DevOps-prosessi. Monessa kohtaa tuli myös tilanne, jossa tutkijan tuli punnita kustannuksia suhteessa ylläpidettävyyteen. Se on varmasti asia, joka kehittäjällä PAAS- ja FAAS -maailmassa tulee useammin esiin kuin perinteisimmissä kehitysympäristössä johtuen muun muassa laajoista integraatio- ja lisäpalvelumahdollisuuksista.

Kulujen muodostumiseen on hyvä perehtyä huolella, vaikka kulut näennäisesti näyttäisivätkin halvoilta. Kun kulujen muodostuminen on suurin piirtein selvillä, voi rakennettavasta sovelluksesta optimoida hyvinkin edullisen käyttää serverlessin kulutukseen perustuvalla hinnoittelumallilla. Kuten luvussa 4.1 huomattiin, serverlessiä käsittelevässä kirjallisuudessa mainitaan, että kehittäjän ei tarvitse huolehtia CPU:sta ja RAMista. Tutkimus osoitti, että tätä kannattaisi tarkentaa seuraavalla tavalla kirjallisuudessa: vaikka kehittäjän ei aluksi tarvitse välittää edellä mainituista määreistä, myöhemmin niiden ymmärtäminen ja mittaaminen on hyvinkin oleellista.

Hintalaskelma paljasti, että komponentin käyttö Azuren palveluiden osalta on lähes ilmaista annetuilla kutsumäärillä. Liiketoimintojen on kuitenkin tarkoitus kasvaa ja silloin skaalautuvat samalla myös kulut kutsumäärien kasvaessa. Laskutusmalli sitoo serverlessin ja DevOpsin yhteen seuraavalla tavalla: Serverless ja DevOps tavoittelevat innovointia, oppimista ja sitä, että uusia ominaisuuksia saadaan nopeasti markkinoille. Kun kulut ovat alussa pienet,

on kynnys yrittää ja erehtyä myöskin pieni. Lisäksi tuottoa saadaan alusta asti ilman suurempia investointeja tai kuukausimaksuja. Palveluntarjoajan ja organisaation tuotteen voi siis katsoa kasvavan yhdessä.

Osa aiemmassa tutkimuksissa mainituista haasteista ja rajoitteista jäivät tämän tutkimuksen jälkeenkin avoimiksi. Hieman hankalan laskutusmallin lisäksi suurimmat haasteet liittyivät työkaluihin ja testaukseen. Laskutuksen osalta olisi hyvä tarjota palveluna havainnollistuksia siitä, paljonko funktiot yksittäin ja kokonaisuutena maksavat. Julkaisutyökalujen osalta (Az Cli, AzPowerShell ym.) ilmeni, että vaikka mahdollisuuksia on paljon, on yhtenäisen linjan löytäminen välillä haastavaa, sillä samat toiminnallisuudet ilmestyvät eri ajankohtana (tai ei ollenkaan) eri työkaluihin.

Testauksesta haastavaa tekee funktioiden staattisuus ja kunnollisen testauskonseptin puuttuminen. Yleisenä ohjeena on eriyttää bisneslogiikka ja testata se erillään. Näin myös konstruktiossa toimittiin. Entä miten tehdään kattavat integraatiotestit funktioiden yhteistoiminnalle? Ongelmaksi muodostuu triggereiden (esimerkiksi Service Busin) maksullisuus tai toisaalta niiden ja monien asynkronisten kutsujen korvaaminen mock-toteutuksilla. Voisi olla hyvä ajatus kehittää storage-emulaattorin ympärille testaustoiminnallisuutta.

Monitoroinnin osalta oli paljon tietoa saatavilla, ja erityisesti omien lokimerkintöjen tutkiminen jälkikäteen oli luontevaa. Ongelma monitoroinnin suhteen oli tietojen hajanaisuudessa. Osa tiedoista oli saatavissa function appin ja osa Application Insightsin kautta. Lisäksi resurssien kulutuksen resurssiryhmäkohtaisesti esittävä Cost Analysis -työkalu ei ollut organisaatiolle saatavilla. Application Insights on toki kattava työkalu, josta saisi kootusti enemmänkin tietoa, mutta sen tehokäyttö vaatisi sen oman hakukielen taitamisen, jota ei tämän tutkimuksen rajoissa ehditty opettelemaan.

Viimeisenä haasteista mainittakoon vielä rakeisuuden ongelma. Konstruktiossa ongelmaa poistettiin arkkitehtuurisuunnittelulla sekä ylläpidettävyyden ja dokumentaation laadulla. Rakeisuuden ongelma voisi olla merkittävämpi haaste, jos funktioita olisi enemmän tai sovelluksessa olisi useampi funktioilla rakennettu komponentti. Silloin tulisi myös kulujen kanssa olla tarkkana. Edellä mainituilla keinoilla sekä kulurakenteen ymmärryksellä voi kuitenkin pienentää ongelman vaikutuksia merkittävästi.

Tutkimus osoitti, että serverless funktioilla toteutettu komponentti on hyvä ratkaisu laajentamaan olemassa olevaa SAAS-sovellusta. Tutkimuksessa kehitetty komponentti myös osoittautui kustannustehokkaaksi ja ylläpidettäväksi. Tutkimuksen aikana paljastui myös DevOpsin, serverlessin ja ylläpidettävyyden positiivinen yhteys. Seuraava käytännön esimerkki havainnollistaa sitä hyvin. Kun valmista komponenttia testattiin ajaa lokaalin kehitysvaiheen jälkeen Azuresta, havaittiin että komponentti ei toiminut oikein. Vian selvittämiseksi avattiin Azure Portaalista funktioiden monitorointi. Monitorointi kertoi virheestä:

"Put token failed. status-code: 404, status-description: The messaging entity 'sb://reminder.servicebus.windows.net/test-reminder' could not be found."

Virheilmoituksen syyksi pääteltiin, että function appiin oli koodissa määritelty edelleen lokaalissa testauksessa käytetty test-reminder Service Bus -jono, vaikka Azureen julkaistussa versiossa oli konfiguroituna infrastruktuuriskriptin Azuren master-versiota varten luoma jono.

Tehtiin seuraava korjaus koodiin, jotta triggerinä käytettäisiin konfiguraatiossa määriteltyä.

```
-        public static async Task RunAsync([ServiceBusTrigger("test-reminder", Connection = "AzureWebJobsServiceBus")]Message reminderMsg,
+        public static async Task RunAsync([ServiceBusTrigger("%ServiceBusQueue%", Connection = "AzureWebJobsServiceBus")]Message reminderMsg,
```

Muutos laitettiin versionhallintaan, se katselmoitiin ja liitettiin master-haaraan, josta se päivittyi automaation avulla Azureen. Aikaa virheen havaitsemista siihen, että korjaus oli tuotantoympäristöä vastaavassa ympäristössä Azuressa, kului alle tunti ja siitäkin osa ajasta kului katselmointivastausten odotteluun. Itsenäinen serverless-komponentti ilman riippuvuuksia liitettävään kohdesovellukseen ja DevOps mahdollistivat, että virheistä opittiin nopeasti ja halvalla. DevOpsin vuoksi säästettiin kuluissa, koska yksi henkilö teki kaikki Dev- ja Ops-puolen tehtävät, jotka vaadittiin, jotta sovellus toimisi tuotannossa jälleen niin kuin piti. Kun konstruktion suunnittelussa oli huomioitu luvun 6 pyrkimys saavuttaa ylläpidettävyyttä tekemällä komponentista helpon muuttaa ja ymmärtää, löytyy virhekohta koodista myös nopeasti. Serverless ja DevOps ovatkin yhdessä todella hyvä kombinaatio, sillä ne mahdollistavat kokeilevan ja oppivan kehitystavan, jossa molempien edut tulevat esille. Lisäksi niiden

avulla päästään nopeasti ja halvalla tilanteeseen, jossa tuote on jo tuotantoympäristössä tuottamassa asiakkaalle arvoa ja kehittäjälle palautetta.

10.2 Tutkimusprosessin pohdintaa

Konstruktiivinen tutkimus sopi erityisen hyvin tutkimusmenetelmäksi, sillä se ohjasi niin kirjoitusprosessia kuin konstruktion kehitystäkin. Konstruktiivisen tutkimuksen vaiheet toimivat kuin arkkitehtuurikuvana tutkijalle, sillä ne antoivat raamit ja synnyttivät vision tutkijalle siitä, miten prosessin vaiheiden noudattaminen johtaisi tiedon kasvuun ja lopulta valmiiseen konstruktioon ja tutkimukseen. Menetelmä ohjaa tutkijaa luomaan ja innovoimaan konstruktion, jossa valinnat ovat perusteltuja, ongelmaa pohditaan eri näkökulmista ja aiheesta on vankka tietämys. Menetelmä sopiikin siten hyvin sovellettavaksi ICT-alan yrityselämän tarpeisiin, etenkin jos ollaan tutkimassa uuden teknologian mahdollisuuksia. Toki resurssien rajallisuus on otettava huomioon, sillä perusteellisuus vie kuitenkin aikaa ja rahaa. Menetelmällä ja DevOpsilla on myös yhteisiä tekijöitä: jatkuva oppiminen ja kokeilu. Sekä tutkimusprosessissa että DevOpsissa pienet palat synnyttävät kokonaisuuden ja uusi tieto johtaa prosessin ja tuotteen tai konstruktion kehittymiseen.

Lukka (2001) mainitsee kaksi konstruktiivisen tutkimuksen haastetta. Ne ovat ongelmat yhteistyössä kohdeorganisaation kanssa sekä se, että tutkimustuloksista osa voi jäädä pois liikesalaisuuksien tai arkaluontoisen tiedon vuoksi. Tässä tutkimuksessa yhteistyö kohdeorganisaation kanssa sujui hyvin, sillä organisaation edustajat olivat tavoitettavissa, kun apua tarvittiin ja he antoivat myös palautetta läpi prosessin. Tutkijalla oli myös selkeä aikataulullinen tavoite, joka tuli loppuasiakkaalta. Konstruktio tulikin valmiiksi aikataulun rajoissa. Konstruktiivisessa tutkimuksessa yhteistyön onnistumisen voi katsoa olevan oleellista myös siitä syystä, että uskoisin muutosvastarinnan olevan pienempää, kun organisaation jäsenet ovat olleet tutkimuksessa mukana antamassa palautetta esimerkiksi koodikatselmuksissa. Julkaisusäännöistä sovittiin etukäteen, joten pelisäännöt olivat selvät. Esimerkiksi lähdekoodin puuttuminen voi hankaloittaa jonkin verran mahdollista jatkotutkimusta, ainakin organisaation ulkopuoliselta taholta. Lukan mainitsemien haasteiden lisäksi huomasin kolmannenkin huomioitavan haasteen. Liike-elämän mukana olo ohjaa jonkin verran konstruktion

valintoja. Näin kävi esimerkiksi palveluntarjoajan kanssa, koska oli järkevää valita kaupallinen, organisaatiossa jo käytössä oleva palveluntarjoaja.

10.3 Rajoitteet ja jatkotutkimus

Tutkimuksen heikkoutena voidaan pitää sitä, että se perustuu yhden palveluntarjoajan tuotteilla tehtyihin palveluihin. Se tarjoaa kuitenkin hyvän mahdollisuuden jatkossa tutkia serverless-paradigmalla toteutetun komponentin siirtämistä toiselle palveluntarjoajalle ja edelleen analysoida, miten se näkyisi esimerkiksi kustannuksissa, ylläpidossa ja DevOpsissa. Toinen tutkimuksen rajallisuuteen vaikuttava huomio on, että kaikkiin serverlessiin ja funktioihin liittyviin rajoitteisiin ja haasteisiin ei löytynyt vastausta tai niitä ei ajan puutteen vuoksi ehditty käsittelemään. Kuten huomattiin, on kehitys tällä saralla nopeaa ja onkin varmasti mielenkiintoista tutkia esimerkiksi kahden vuoden päästä, mikä silloin on tilanne kirjallisuuden ja käytännön osalta. Kolmas seikka, joka vaikutti tutkimustuloksiin, oli se, että tuotteella ei vielä ollut tuotantoa. Tästä syystä ei päästy havainnoimaan DevOpsin palautevirtaetua. Lisäksi luotua konstruktiota voisi varmasti vielä jatkokehittää monella tavalla. Mahdollisuuksia voisi olla esimerkiksi suoritusaikojen ja muistinkäytön optimointi, hälytykset virheistä, datan kerääminen esimerkiksi varausten peruutuksista ja continuous maintenancen ottaminen mukaan. Tässä tutkimuksessa oli skenaario, jossa serverlessiä käytetään laajentamaan olemassa olevaa toteutusta. Serverless taipuu varmasti myös moneen muuhun mielenkiintoiseen skenaarioon, esimerkiksi isomman kokonaisuuden rakentamiseen funktioilla rakennetuilla komponenteilla. Entä jos laajentavia komponentteja on useita? Miten käy funktioiden määrän kasvaessa?

10.4 Lopuksi

Leitner et al. (2018) käyttävät serverlessistä termiä ”NoOps”, ja MarketsandMarketsin (2017) servelessiä käsittelevässä tutkimuksessa puhutaan siirtymästä DevOpsista serverless computingiin, jossa kehittäjien ja operatiivisen osastojen yhdistymisen sijaan onkin kyse operaatioiden kokonaan poisjäämisestä. Suunta näyttäisi olevan lisääntyvässä määrin kohti mikropalveluja, automaatiota ja uusien innovointien mahdollistamista poistamalla

kehittäjiltä esteitä. Tähän kohtaa sopii hyvin lainata Microsoftin MVP Mark Heathia: ”Kehittäjillä pitäisi olla parempaa tekemistä kuin kaitsea virtuaalikoneita.” (Kohdeorganisaation sisäiset koulutusmateriaalit)

Tämän tutkimuksen valossa näkisin, että DevOps ja Serverless ennemminkin täydentävät toisiaan, kuin että serverless olisi DevOpsin jatkumo. Johdannossa esitettyihin MarketsAndMarketsin tutkimustuloksiin viitaten onkin useamman miljardin dollarin kysymys, mitkä ovat niiden mahdollisuudet yhdessä yhdistettynä muiden tulevaisuuden kärkiteknologioiden kanssa. Serverlessin täyden potentiaalin saavuttaminen vaatii kuitenkin sen haasteiden selvittämistä, kokeilua, innovointia ja parhaiden käytänteiden luomista.

Vaikka tutkimus oli vain raapaisu mielenkiintoisesta, nopeasti kehittyvästä ja paljon potentiaalia omaavasta serverless-aihepiiristä, toivon sen tarjoavan mielenkiintoisia näkökulmia ja herättävän ajatuksia niin tiedeyhteisössä kuin ohjelmistokehittäjissäkin sekä organisaatiotasolla.

Kuriositeettina mainittakoon, että tutkimuksessa syntyi 673 riviä C#-koodia ja noin 135 riviä PowerShell-skriptiä. Lisäksi syntyi YAML- ja ARM-templaatteja, joiden osalta on hyvä mainita, että niitä osin generoitiin AzureDevopsilla ja osin hyödynnettiin kohdeorganisaatiossa käytössä olevia valmiita pohjia.

Lähteet

https://www.appdynamics.com/media/uploaded-files/White_Paper_-_An_Intro_to_DevOps.pdf. 2014. (Haettu 14.5.2019).

Anonyymi. 2005. Software Maintenance. *IEEE Software, 22*(4), p. 103.

Adzic, G ja Chatley, R. 2017. Serverless computing: economic and architectural impact. Proceeding ESEC/FSE 2017. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Pages 884-889.

Aleem, S., Ahmed F., Faheem, B., Batool, R., Rabia, K. ja A. Khattak, A. 2019. Empirical Investigation of Key Factors for SaaS Architecture Dimension. *IEEE Transactions On Cloud Computing* PP, no. 99.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski A. ja Sute, P. 2017. Serverless Computing: Current Trends and Open Problems. Procedia Computer Science 2017, Vol.108, pp.685-694.

Back T. 2018. Using a Microbenchmark to Compare Function as a Service Solutions. Service-Oriented and Cloud Computing, pp. 146-160.

Diaz, A. 2016. Three Ways That "Serverless" Computing Will Transform App Development In 2017. https://www.forbes.com/sites/ibm/2016/11/17/three-ways-that-serverless-computing-will-transform-app-development-in-2017/ (Haettu 19.8.2019).

Dörnenburg, E. 2018. The Path to Devops. IEEE Software September 2018, Vol.35(5), pp.71-75.

Erdil, K., Finn, E., Keating, K., Meattle, J., Park, S. ja Yoon, D. 2003. Software maintenance as part of the software life cycle. Comp180: Software Engineering Project, pp.1-49.

Ebert, C., Gallando, G., Hernantes, J. ja Serrano, N. 2016. DevOps. IEEE Software May 2016, Vol.33(3), pp.94-100.

Eyk, E., Iosup, A., Seif, S., ja Thömmes, M. 2017. The SPEC cloud group's research vision on FaaS and serverless architectures.

Eyk, E., Erwin, Toader, L., Talluri, S., Versluis, L. ja Iosup, A. 2018. Serverless is More: From PaaS to Present Cloud Computing. IEEE Internet Computing September 2018, Vol.22(5), pp.8-17.

Fox, G.C., Ishakian, V., Muthusamy, V. ja Slominski, A. 2017. Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. First International Workshop onServerless Computing (WoSC) 2017.

Gonzales, R. ja Piirainen, K. 2014. Constructive Synergy in Design Science Research: A Comparative Analysis of Design Science Research and the Constructive Research Approach.

Greene, D. 2015. What is devops? https://techcrunch.com/2015/05/15/what-is-devops/?guccounter=1. (Haettu 14.5.2019).

Johann, S. 2017. Kief Morris on Infrastructure as Code. IEEE Software January 2017, Vol.34(1), pp.117-120.

Johnson, P. 2017. How serverless changes application development. InfoWorld.com June 7, 2017.

Kasanen, E., Lukka, K. ja Siitonen, A. 1993. The Constructive Approach in Management Accounting Research.

Kim, G., Humble, J., Debois, P. ja Willis, J. 2016. DevOps handbook. IT Revolution Press LCC.

Knorr, E. 2016. What serverless computing really means. InfoWorld.com July 11, 2016.

Leitner, P., Wittern, E., Spillner, J. ja Hummer, W. 2018. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. PeerJ PrePrints.

Lin, P-M. ja Glikson, A. 2019. Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach. arXiv preprint arXiv:1903.12221.

Lukka, K. 2001. Konstruktiivinen tutkimusote. https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/ Menetelmäartikkelit (haettu 16.4.2019).

Malawski, M., Gajek, A., Zima, A., Balis, B. ja Figiela, K. 2017. Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions." Future Generation Computer Systems.

Manner, J., Endreß, M., Heckel, T. ja Wirtz, G. 2018. Cold Start Influencing Factors in Function as a Service. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).

MarketsandMarkets Research reports. 2018. https://www.marketsandmarkets.com/Market-Reports/devops-824.html (Haettu 21.8.2019).

MarketsandMarkets Research reports. 2018. https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html?gclid=EAIaIQobChMI9b2Ox-puT5AIVyqMYCh0Erw9sEAAYASAAEgIZ6fD_BwE (Haettu 21.8.2019).

Martin, R.C. 2002. Agile software development: principles, patterns, and practices. Prentice Hall.

Nastic, S. 2017. A Serverless Real-Time Data Analytics Platform for Edge Computing. IEEE Internet Computing, 21(4), pp. 64-71.

Nelson-Smith, S. 2012. Blogiteksti. http://www.jedi.be/blog/2010/02/12/what-is-this-devops-thing-anyway/ (Haettu 31.8.2019).

Oyegoke, A. 2011. The constructive research approach in project management research", International Journal of Managing Projects in Business, Vol. 4 Iss 4 pp. 573 – 595.

Pang, C. ja Hindle, A. 2016. Continuous Maintenance. 2016 IEEE International Conference on Software Maintenance and Evolution.

Pérez, A., Moltó, G., Caballer, M. ja Calatrava, A. 2018. Serverless computing for container-based architectures. Future Generation Computer Systems, 83, pp. 50-59.

Rautiainen, A., Sippola, K. ja Mättö T. 2017. Perspectives on Relevance: The Relevance Test in the Constructive Research Approach. Elsevier.

Savage, N. 2018. Going Serverless, Communications of the ACM 23 January 2018, Vol.61(2), pp.15-16.

Shahin, M., Ali Babar, M. ja Zhu, L. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access 2017, Vol.5, pp.3909-3943.

Sewak, M. ja Singh, S. 2018. Winning in the era of Serverless Computing and Function as a Service. 2018 3rd International Conference for Convergence in Technology (I2CT). IEEE.

Varghese, B. ja Buyya, R. 2018. Next generation cloud computing: New trends and research directions.Future. Generation Computer Systems, 79(P3), pp. 849-861.

Yegulalp, S. 2017. Get functional! 5 open source frameworks for serverless computing. InfoWorld.com.

https://docs.microsoft.com/en-us/azure/devops/user-guide/?view=azure-devops.(Haettu 22.5.2019).

https://www.informationweek.com/devops/25-devops-vendors-to-consider-in-2018/a/d-id/1330394. (Haettu 22.5.2019).

Konstruktion rakentamisessa on lisäksi hyödynnetty Microsoftin serverlessiä, Azurea, AzureDevOpsia, AzPowershelliä ja Powershellillä käsittelevien dokumentaatioiden (https://docs.microsoft.com/en-us/) sekä kohdeyrityksen sisäisten dokumenttien ja koulutusmateriaalien sisältöjä.

Liitteet

Liite 1 Varausvahvistuksen data esitettynä JSON-schema-muodossa

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "bookingNumber": {
      "type": "string"
    },
    "calendarEventId": {
      "type": "integer"
    },
    "time": {
      "type": "string"
    },
    "station": {
      "type": "string"
    },
    "address": {
      "type": "string"
    },
    "products": {
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
            "name": {
              "type": "string"
            },
            "price": {
              "type": "integer"
            }
          },
          "required": [
            "name",
            "price"
          ]
        }
      ]
    },
    "campaignCode": {
      "type": "string"
```

```
    },
    "emailSender": {
      "type": "string"
    },
    "emailReceiver": {
      "type": "string"
    },
    "senderName": {
      "type": "string"
    },
    "receiverPhoneNumber": {
      "type": "string"
    },
    "language": {
      "type": "string"
    },
    "notifications": {
      "type": "object",
      "properties": {
        "smsEnabled": {
          "type": "boolean"
        },
        "notificationTime": {
          "type": "string"
        }
      },
      "required": [
        "smsEnabled",
        "notificationTime"
      ]
    }
  },
  "required": [
    "bookingNumber",
    "calendarEventId",
    "time",
    "station",
    "address",
    "products",
    "campaignCode",
    "emailSender",
    "emailReceiver",
    "senderName",
    "receiverPhoneNumber",
    "language",
    "notifications"
  ]
}
    },
    "emailSender": {
```

```
      "type": "string"
    },
    "emailReceiver": {
      "type": "string"
    },
    "senderName": {
      "type": "string"
    },
    "receiverPhoneNumber": {
      "type": "string"
    },
    "language": {
      "type": "string",
      "enum": [
        "swedish",
        "english”,
        ”finnish”
      ]
   },
    "notifications": {
      "type": "object",
      "properties": {
        "smsEnabled": {
          "type": "boolean"
        },
        "notificationTime": {
          "type": "string"
        }
      },
      "required": [
        "smsEnabled",
        "notificationTime"
      ]
    }
  },
  "required": [
    "bookingNumber",
    "calendarEventId",
    "time",
    "station",
    "address",
    "products",
    "campaignCode",
    "emailSender",
    "emailReceiver",
    "senderName",
    "receiverPhoneNumber",
    "language",
    "notifications"
  ]
}
```

Liite 2 Kululaskelmia

Osio 1: Viestin lähetykseen käytettyjen palveluntarjoajien vertailua

Tekstiviestipalveluiden hintavertailun hinnat on annettu palveluissa dollarimääräisenä. Koska asiakas on Ruotsissa, on hinnat laskettu kruunuina 18.6.2019 kurssin mukaan. (1 dollari = 9,50708 kruunua, https://www.kauppalehti.fi/porssi/valuutat).

	Twilio	**Nexmo**	**D7Networks**	**ClickSend**	**Mailjet**
Hinta	0,0550$ (0,529 SEK) /viesti	0,0523$(0,497 SEK) /viesti	0,049$ (0,466 SEK) /viesti	0,0516$ (0,490 SEK) /viesti	0,04$ (0,380 SEK) / viesti

SMS-palveluntarjoajien hintavertailu. Tekstiviestien lähetys Ruotsissa tarjoajan APIn kautta

	SendGrid	**Sendinblue**	**Mailjet**	**MailGun**
Ilmainen taso	100 ilmaista viestiä päivässä	300 ilmaista viestiä päivässä	200 viestiä päivässä tai 6000 viestiä kuukaudessa.	10000 viestiä kuukaudessa.
Perus maksullinen taso	14,90 € per 40000 viestiä kuukaudessa	19 € per 40000 viestiä (15,20 $ jos maksaa vuosittain.)	7,16 € per 30000 viestiä kuukaudessa. (Laskutus vuosittain)	10 € per 30000 viestiä kuukaudessa.
Pro maksullinen taso	147,40 € per 200 000 viestiä dedikoiduilla IP.llä (Ei spam)	103,20€ per 350 000 viestiä tasolla. (sis. dedikoidut IP:t)	112,46 € per 450 000 viestiä kuukaudessa (sis. dedikoidut IP:t)	124,15€ per 200 000 viestiä (sis. dedikoidut IP:t)
Palvelun käyttäjiä	Spotify, Airbnb, Uber	BMW, Michelin, Spiegel	Microsoft, Johnson & Johnson	GitHub, Slack, reddit
Integraatio	Suora Azure integraatio, C#-kirjasto	REST/JSON	REST/JSON Suora Azure integraatio	.Net-integraatio

E-mail- ja SMS-palveluntarjoajien vertailuun valittiin palveluntarjoajia, jotka ovat tunnettujen yritysten käytössä, joiden laskutus on lähetettyihin viestien määrään perustuva ja joilla on RestAPI, C#-kirjasto tai Azure-integraatio. Lähteinä toimivat yritysten kotisivut ja yritykset ovat löytyneet Google-haulla.

Osio 2: Jonopalvelujen hinta-arviota (lähde Azuren dokumentaatio)

1. Hinnat:

   Service Bus -operaatioiden hinta: € 0.043 per miljoona operaatiota.

   Storage queue -operaatioiden hinta: €0.0004 per 10 000 operaatiota.

2. Vahvistusviestit:
   Storage queue:

   > Operaatiot: Jonoon asetus x 2 (SMS ja e-mail). Luku x 2 (SMS ja e-mail funktiot) = 4 => Tarvittavat operaatiot = 4 x 180 000 (tapahtumaa) = 720 000 operaatiota. Hinnaksi muodostuu siten
   >
   > 720 000 / 10 000 = 72
   >
   > 72 x 0.0004 = **0,0288 € / kk**

   Service Busilla sama maksaisi 720 000 operaatiota => **0,043 € / kk.**

3. Muistutusviestit:

   Service Bus:

   > Asetus muistutusjonoon + luku = 2 x 180 000 operaatiota + vahvistusviestinhinta 0,043 € kk + 0,0288 €/kk = **0,0718 € kk**
   >
   > Käytetty standard tier, jossa ei viestin maksimielinaikaa.

   Storage Queue: (7 päivän elinaika, pitää laittaa viesti takaisin jonoon seitsemän päivän välein).

Kuukaudessa mahdollista 30 / 7 => maksimissaan 5 (esimerkiksi 1,8,15,22,30 päivät).

Tarvitaan tällöin: asetusmuistutus jonoon x 4 +vahvistusviestinhinta. Operaatioiden määrä tuplaantuu.

hinta: 6 x 180 000 operaatiota = 1080000 operaatiota.

1080000 / 10 000 = 108

108 x 0,0004 = **0.0432 € / kk**

Osio 3: Funktiot

Alla olevassa taulukossa on esitetty function appin käyttöön liittyviä mittareita sekä laskettu kustannuksia niihin perustuen. Tarvittavat arvot selvisivät Azuren dokumentaatiosta (https://azure.microsoft.com/en-us/pricing/details/functions/) ja ne saatiin käyttämällä Azure-portaalin function appin monitorointityökaluja.

Suure/ Funktio	Start	Email	SMS	Reminder	Total
Funktioiden keskimääräinen muistinkäyttö ja pyöristys (MB).	193 ~ 256	193 ~ 256	193 ~ 256	193 ~ 256	
GB **256 MB / 1025 MB**	0,25 GB	0,25 GB	0,25 GB	0,25 GB	
Kutsujen määrä.	90 000	180 000 Muistutus ja varaus.	180 000 Muistutus ja varaus.	90 000	540 000
Keskimääräinen suoritusaika s	1,356	0,229	0,436	0,822	
Resurssien kulutus s	90 000 * 1,356 = 122 040	180 000 * 0,229 = 41 220	180 000 * 0,436 = 78480	90000 * 0,822 = 73 980	

Resurssien kulu-tus GB-s	122 040 * 0,25 = 30510	41220 * 0,25= 10 305	78480 * 0,25 = 19620	73 980 * 0,25 = 18 495	78 930 GB-s

Liite 3 Readme (GitHubin yksityisestä Git-hakemistosta)

Local development

Clone the repository. Azure functions can be run locally. It uses localhost and storage emulator.
For running function app locally you´ll need to prepare local configuration in local.settings.json:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet",
    "MockBookingApi": "Set mock booking api url here",
    "MailjetSMSApiKey": "Set SMS APIKey here",
    "MailjetSMSApiEndpoint": "https://api.mailjet.com/v4/sms-send",
```

```
        "AzureWebJobsServiceBus": "Set Service Bus endpoint here"
    }
}
```

Start solution and you will see endpoint to test function with Postman etc. See configuration chapter.

1. VS Code

pre:

- npm install -g azure-functions-core-tools
- install Azure function extension

new function: func new
start: func host start (shows url to call when running)

2.Visual Studio

pre:

- install Azure Development workload (modify)

Configuration

All the necessary configuration (apikeys etc) are set in function app ARM-template. For testing function app, you should register into Mailjet to get API keys for SMS and e-mail. (free tier: 10 emails/per hour). SMS is not free, but you can buy fe. 100 messages with Apr. 4 euros. You can also set smsEnabled to false in json call. Notice you´ll need real Service Bus instance.

DevOps & Deployment

New commit to booking-reminder will trigger own pipeline in Azure DevOps that builds, tests, zips and archives files to Blobstorage. Function App (platform) is created in deploy.ps1 scripts with Arm deployment. Zip package is fetched from Azure and functions in it are published to related function app with DeployFunctions.ps1.

Monitoring

Can be easily switch in reminderApp.json ARM template to use any AI by just changing aiKey parameter when deployed (link from Functionapp).
Costs: go to Azure functionapp platformfeatures > metrics or check subscriptions > subscription for consumption details. By function: In Azure you can monitor functions

separately for every run. (Functionapp > functions > function > monitor) shows you Log.Information-level logs from source code.

Related resources:

For deployment:

AZPowershell:
https://docs.microsoft.com/en-us/powershell/module/Az.Resources/?view=azps-2.4.0

Zip deployment:

https://docs.microsoft.com/en-us/azure/azure-functions/deployment-zip-push

Azure functions common:

https://docs.microsoft.com/en-us/azure/azure-functions/

Mailjet (service for sending e-mail & SMS messages):

https://app.mailjet.com/ (needs registration)

API documentation

Function app can be called with any Http Client with json format body. To run function app, one must call StartFunction that is triggered by Http post request.

Running locally needs URL provided by Azure or localhost. When running from Azure Functions security level is set to “Function” which means it needs to be called with header parameter "x-functions-key"="..." (in prod) or secret in query string. (URL is found in azure > functionapp > StartFunction Get function URL) In portal it is possible to generate a function key for each customer.

example json for request:

```
{
   "bookingNumber": "1234",
   "calendarEventId": 123456789,
   "time": "2019-07-11 22:12 PM",
   "station": "station",
   "address": "street",
   "products": [{ "name": "as", "price": 45
}],
   "campaignCode": "5953",
   "emailSender": "putvalidemailsender@somemail.com",
   "emailReceiver": "putvalidemailreceiver@somemail.com",
```

"senderName": "companyX",
"receiverPhoneNumber": "+358.......",
"language": "finnish",
"notifications": {
"smsEnabled": false,
"notificationTime": "2019-07-30T13:06:41Z"
}
}

Note:

- bookingNumber and calendarEventId must be exactly those if tested with fake API (Logic App) in terms to pass validation.
- NotificationTime is utc (in example it will be triggered 16:06 Finnish local time).
- SenderName max length is 11 char (= max phone number length).
- Language: supported values "swedish, english & finish"
- Time is shown in receivers time zone
- NotificationTime: null (= notifications disabled).