Karri Kuivanen

Serverless-arkkitehtuuri ja järjestelmäkustannukset

Tietotekniikan kandidaatintutkielma

20. toukokuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Karri Kuivanen

Yhteystiedot: karri.m.h.kuivanen@student.jyu.fi

Työn nimi: Serverless-arkkitehtuuri ja järjestelmäkustannukset

Title in English: The Effects of Serverless on System Costs

Työ: Kandidaatintutkielma

Sivumäärä: 23+0

Tiivistelmä: Tämän tutkielman tavoitteena on selvittää, millä tavoin viime vuosina suurta suosiota saavuttaneen serverless-arkkitehtuurin hyödyntäminen vaikuttaa järjestelmän käyttökustannuksiin. Lisäksi tutkielma esittelee serverless-arkkitehtuurin (erityisesti FaaS) keskeisimmät periaatteet. Tutkielma on toteutettu kirjallisuuskatsauksen keinoin. Tarkasteltu kirjallisuus vaikuttaisi puoltavan usein esitettyä väitettä serverless-mallin tuomista kustannuseduista, mutta tämän lisäksi esille nousee kokoelma mallin nuoreen ikään liittyviä haasteita ja huoli riippuvuudesta palveluntarjoajaan (*vendor lock-in*).

Avainsanat: serverless, function as a service, FaaS, mikropalvelut

Abstract: The purpose of this thesis is to find out what effects the adoption of a serverless architecture has on the operational costs of a software system. In addition, the thesis aims to introduce the core concepts of serverless architecture (especially FaaS). The method used to produce the thesis is literature review. The examined literature suggests that the often repeated promise of serverless reducing costs might in fact be true, but does also bring up challenges related to the relatively young age of the model as well as concerns over vendor lock-in.

Keywords: serverless, function as a service, FaaS, microservices

Kuviot

Sisältö

1 Johdanto

Serverless on järjestelmäarkkitehtuurimalli, jonka keskeisenä tavoitteena on siirtää palvelimien hallinnan, niiden provisioinnin sekä skaalautumisen suunnittelun aiheuttama operatiivinen kuorma ohjelmiston tuottajalta (pilvipalvelun käyttäjältä) palveluntarjoajalle. Järjestelmä ei näyttäydy serverless-mallissa kehittäjälle enää kokoelmana palvelimia ja näillä suoritettavia monoliittisia[1] ohjelmistoja. Sen sijaan puhutaan tapahtumista (*event*) ja näitä käsittelevistä tilattomista funktioista (*FaaS, Function as a Service tai fPaaS, Function Platform as a Service*), joita suoritetaan tarvittaessa, tapahtumiin reagoiden. Kyseessä on siis abstraktiotason nosto tilanteeseen, jossa järjestelmä rakentuu koodista palvelimien sijaan. Tästä seuraa, että kehittäjät voivat siirtää resurssejaan infrastruktuurin hallinnasta ohjelmiston liiketoimintalogiikan kehittämiseen (van Eyk ym. 2018).

Lähestymistapa tuo mukanaan kehittäjäkokemuksen suoraviivaistamisen lisäksi muitakin etuja. Jatkuvasti suorituksessa olevat palvelinsovellukset on mallissa korvattu lyhytkestoisilla funktiosuorituksilla, ja käyttäjää laskutetaan vain tämän funktiokutsun vaatiman prosessointiajan mukaan. Tyhjäkäynnistä ei siis aiheudu kustannuksia, sillä sitä ei mallin määritelmän mukaan synny. Tämän lisäksi perinteisiin palvelinratkaisuihin verrattuna serverless-alustojen hinnoittelu on myös merkittävästi hienojakoisempaa. Siinä missä konesalista vuokrattu fyysinen tai virtuaalinen palvelin on tyypillisesti hinnoiteltu tunti- tai kuukausitasolla (kts. esim. Vultr 2019), on Amazonin AWS Lambda -palvelun pienin laskutettava yksikkö 100 millisekuntia (Amazon 2019).

Serverless-malli sai alkunsa Amazonin julkaistua AWS Lambda-palvelunsa vuonna 2014, ja kilpailevat yritykset seurasivat perässä hetkeä myöhemmin (McGrath ja Brenner 2017). Tutkimusyritys Gartnerin (2018) julkaisemassa katsauksessa serverlessin todetaan olevan yhtiön kehittämän *Hype Cycle*-mallin (kuvio 1) huippukohdan – *Peak of Inflated Expectations*, “paisuneiden odotusten huippu” – kynnyksellä, ja täydellisen ymmärryksen “tasangon” olevan vielä useamman vuoden päässä. Mallia ovat kuitenkin onnistuneesti hyödyntäneet jo myös

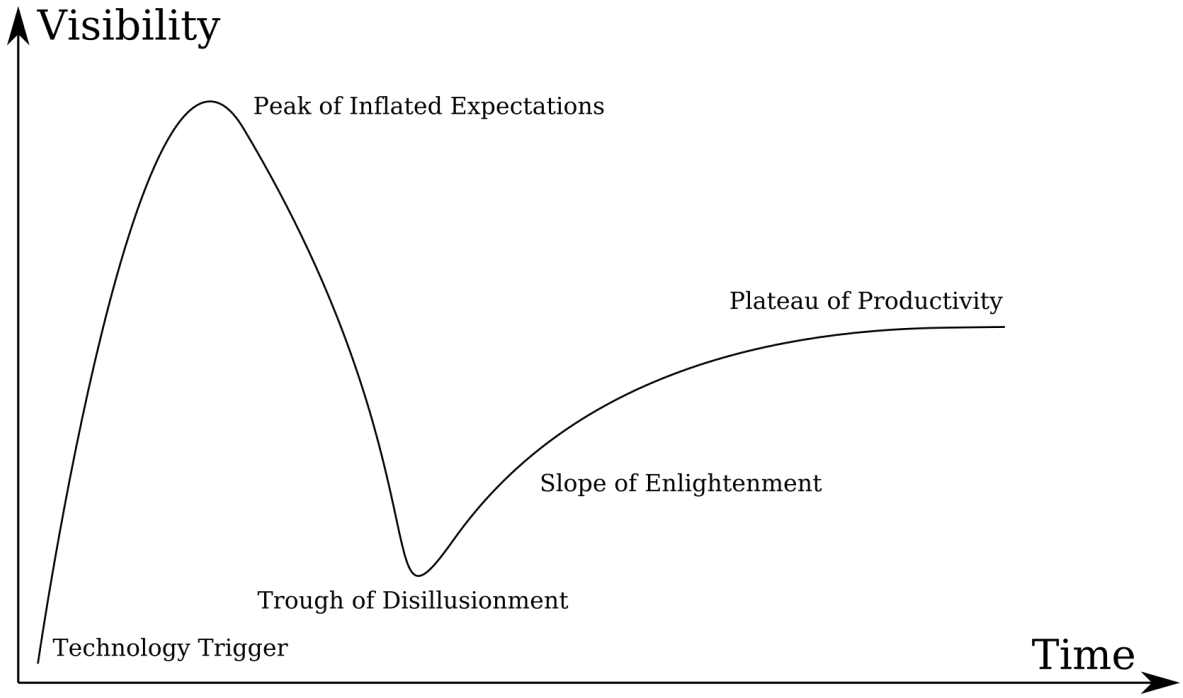
1. Arkkitehtuurimalli, jossa kaikki toiminnallisuus on sisällytetty yhteen ohjelmistoon (koodikantaan). Esimerkiksi ohjelmisto, jossa vaikkapa käyttäjähallinta, käyttäjien välinen viestintä sekä käyttöliittymä on toteutettu osana samaa ohjelmistoa.

monet alan suurimmista toimijoista, kuten esimerkiksi Netflix (2014).

Malliin liittyy paljon lupauksia, mutta parhaat käytännöt hakevat vielä muotoaan ja täysi ymmärrys mallin lainalaisuuksista puuttuu. Näin ollen ei ole myöskään lainkaan itsestäänselvää, millä tavoin serverless-arkkitehtuurin hyödyntäminen palvelun toteuttamisessa vaikuttaa kyseisen palvelun ylläpitokustannuksiin. Tämän tutkielman tavoitteena onkin vastata juuri tähän kysymykseen. Asian selvittämiseksi on kuitenkin tärkeää luoda katsaus myös siihen, mihin tämä lupaus perustuu. Siksi tutkielmassa tarkastellaan lisäksi sitä, miten serverless-malliin ollaan päädytty, ja sitä, mitä ovat ne mallin ominaisuudet jotka vaikuttavat järjestelmän kustannuksiin.

Tutkielma on toteutettu kirjallisuuskatsauksen keinoin ja lähteenä on käytetty lähinnä tieteellisissä julkaisuissa julkaistuja artikkeleita. Tämä toi tutkielman laatimiseen oman haasteensa, sillä aihe on etenkin tieteellisessä keskustelussa vielä hyvin tuore ja näin ollen saatavilla oleva tietomäärä rajallinen. Valtaosa lähdemateriaalista on julkaistu viimeisen kolmen vuoden aikana.

Tutkielman toisessa luvussa esittelen kuinka serverless jatkaa mikropalveluiden viitoittamalla tiellä ja monin tavoin mahdollistaa kyseisen mallin entistä paremman hyödyntämisen. Kolmannessa luvussa selvennän tarkemmin sitä, millä tavoin serverless-arkkitehtuurin noudattaminen vaikuttaa ohjelmiston ylläpitokustannuksiin, tarkastellen aihetta kahden tutkimuksen kautta (Adzic ja Chatley 2017; Villamizar ym. 2017). Neljännessä luvussa tutkin sitä, millä tavoin serverless voi tehostaa kehittäjän työskentelyä, sekä sitä, millaisia ongelmia serverless-yhteisön tulee ratkaista ennen laajempaa käyttöönottoa.



Kuvio 1. Gartnerin *Hype Cycle*. Piirretty mukaillen Kempin (2019) kaaviota

2 Mikropalvelut ja serverless

Baldinin ym. (2017) mukaan serverlessin suosion katalyyttinä on toiminut suurissa määrin ohjelmistoyhteisön viime vuosien aikana tapahtunut siirtymä kohti mikropalveluarkkitehtuuria (*microservices architecture*). Siinä missä mikropalvelut jatkokehittivät SOA-mallin (*Service-Oriented Architecture*) alulle panemaa itsenäisten, tilattomien ja loogisiin paloihin jaettujen palveluiden tuottamiseen keskittyvää ajattelumallia, voidaan serverlessin katsoa olevan edelleen jatkoa samalle trendille (van Eyk ym. 2018).

Mikropalveluajattelun perusperiaate on monoliittisten palvelinsovellusten korvaaminen kokoelmalla pieniä, omavaraisia "mikropalveluita". Nämä palvelut kommunikoivat keskenään kevein metodein, kuten esimerkiksi paljastamalla REST[1]-rajapinnan (Balalaie, Heydarnoori ja Jamshidi 2016). Balalaie ym. (2016) katsovat mallin olevan oivallinen tapa välttää monoliittisiin sovelluksiin usein liitettäviä ongelmia. Ohjelmakoodin tulkinta helpottuu, kun järjestelmä rakentuu yksinkertaisista, itsenäisistä moduuleista. Tämän lisäksi modulaarisuus mahdollistaa myös ketteryyden teknologiavalinnoissa: koska palvelut toimivat itsenäisesti ja niiden välinen keskustelu tapahtuu yleisiä käytäntöjä noudattaen, on järjestelmään vaivatonta lisätä moduuleja, jotka on toteutettu olemassaolevista moduuleista poikkeavilla teknologioilla.

Mikropalvelumallin hyödyt kasvavat entisestään, kun järjestelmä toteutetaan serverless-alustalla. Ennen FaaS:n mahdollistamaa käytön mukaan tapahtuvaa laskutusta oli palvelun suorittamista varten resursseja varattava käyttöasteesta riippumatta kiinteä määrä. Taloudellisista syistä olikin usein perusteltua niputtaa useita mikropalveluita yhden järjestelmäkomponentin (palvelininstanssin) alle. Tämä johtaa helposti siihen, että yhdessä palvelussa tapahtuva virhe vaikuttaa negatiivisesti myös muihin samalla instanssilla suorituksessa oleviin palveluihin (Adzic ja Chatley 2017).

Singleton (2016) tuo artikkelissaan esille mikropalvelumallin aiheuttaman tarpeen palveluiden kommunikoinnin mahdollistavalle lisäinfrastruktuurille ja -koodille. Liiketoimintalogiikan lisäksi vaaditaan merkittävä määrä työtä esimerkiksi viestijonojen toteuttamiseen. Sa-

1. Representational State Transfer

man ovat huomanneet tapaustutkimuksessaan myös Adzic ja Chatley (2017). Adzic ja Chatley kertovat, kuinka serverless-alustaan siirtyminen vähensi tapauksen järjestelmässä merkittävästi juuri Singletonin kuvaaman infrastruktuurikoodin määrää.

Tämän tekee mahdolliseksi se, että suurimmat serverless-alustat sisältävät myös lukuisia funktioiden toimintaa tukevia palveluita. Tähän joukkoon lukeutuu muun muassa Amazonin *Simple Notification Service*, jonka yhdeksi käyttökohteeksi palvelun dokumentaatio mainitsee juuri järjestelmien välisen kommunikaation (Amazon 2019). AWS Lambda – Amazonin FaaS-palvelu – on integroitavissa SNS:ään. Käyttäjän luoma funktio voidaan kytkeä vastaanottamaan viestejä määritellyistä aiheista (*topic*), ja funktion koodissa määritellään, millä tavalla funktio reagoi vastaanottamaansa viestiin. AWS:n JavaScript-SDK:ta käyttämällä saadaan Lambda-funktio vastaanottamisen lisäksi luonnollisesti myös lähettämään viestejä. Listaus 2.1 sisältää toimivan esimerkin kyseisen SDK:n käytöstä. Listauksen funktio lähettää suoritettaessa viestin listauksen rivillä viisi määriteltyyn aiheeseen.

Listaus 2.1. Esimerkki SNS-notifikaatiopalvelun käytöstä AWS Lambda-ympäristössä. Esimerkki on laadittu tutkielmaa varten mukaillen Amazonin julkaisemaa tutoriaalia (Amazon 2019).

```
1  // Ladataan AWS SDK
2  const AWS = require('aws-sdk')
3
4  // SNS-aihetunniste
5  const TOPIC_ARN = 'arn:aws:sns:eu-west-1:123:esimerkkitopic'
6
7  // Määritellään funktio, jota kutsutaan suoritettaessa
8  module.exports.handler = async () => {
9    // Alustetaan SNS-asiakas
10   const sns = new AWS.SNS()
11   try {
12     // Lähetetään viesti
13     const result = await sns.publish({
```

```
14          Message: 'Viestin␣sisältö',
15          TopicArn: TOPIC_ARN
16        }).promise()
17        console.log(result)
18      } catch (err) {
19        // Virhetilanteessa kirjoitetaan virhe lokiin
20        console.error(err)
21      }
22  }
```

3 Serverless-järjestelmän käyttökustannukset

Kuten johdannossa mainittiin, on serverless-resurssien käytön hinnoittelu puhtaasti toteutuneeseen käyttöön perustuvaa ja hyvin hienojakoista. Palvelinpohjaisen ratkaisun käyttökustannukset muodostuvat puolestaan varatun prosessointikapasiteetin mukaan, riippumatta lainkaan siitä hyödynnetäänkö kapasiteettia vai ei.

Adzic ja Chatley (2017) vertailivat kuvitteellisen järjestelmän kustannuksia AWS Lambdan ja kolmen virtuaalipalvelinpohjaisen alustan välillä. Virtuaalipalvelinpohjaisella alustalla tarkoitetaan tässä sellaista palvelua, joka tarjoaa käyttäjälle mahdollisuuden ottaa käyttöön valitsemallaan käyttöjärjestelmällä varustettuja virtuaalisia palvelimia, joille käyttäjä voi asentaa haluamansa ohjelmistopinon. Käyttäjä maksaa palvelimesta tyypillisesti tunti- tai kuukausihinnan. Artikkelin esimerkin järjestelmän käyttö koostui viiden minuutin välein suoritettavasta 200 millisekunnin kestoisesta laskentasuorituksesta. Esimerkki on äärimmäinen, mutta koska järjestelmän käyttö oli näin vähäistä ja merkittävä osa näin ollen tyhjäkäyntiä, oli kustannusero hätkähdyttävä: ero AWS Lambdan ja edullisimman vertaillun virtuaalipalvelinpohjaisen ratkaisun (Heroku Hobby 512 MB) välillä oli lähes 50000%.

Tähän epäkohtaan Adzic ja Chatley mainitsevat mahdolliseksi ratkaisuksi konttiteknologioiden (*containerization*, esim. Docker) käytön. Konttiteknologia tarkoittaa mallia, jossa yhdelle palvelimelle voidaan asentaa toisistaan tehokkaasti eristettyjä palvelinsovelluksia eli “kontteja”. Konteilla saavutetaan virtuaalipalvelimien tuoma isolaatio ilman virtualisointikerroksen aiheuttamaa suorituskykysakkoa. Sen sijaan, että kullekin ohjelmistolle vuokrattaisiin oma palvelin – mikä aiheuttaisi mahdollisesti tyhjäkäyntiä – voidaan kustakin ohjelmistosta luoda kontti, joka asennetaan konttipalvelimelle. Koska näin voidaan yhdellä palvelimella suorittaa turvallisesti useita sovelluksia, voidaan vuokrattua laskenta-aikaa hyödyntää tehokkaammin. Sekä Google että Amazon tarjoavat hallitun tavan ajaa Docker-kontteja (Google Kubernetes Engine, Amazon Elastic Container Service).

Virtuaalipalvelin- tai konttipohjaista ratkaisua käytettäessä tulee kuitenkin ottaa huomioon varajärjestelmän ja kuormituksentasaimen (*failover, load balancer*) tarve. Mikäli toinen järjestelmistä – tai useamman kuin yhden varajärjestelmän tapauksessa yksi järjestelmistä –

lakkaa vastaamasta esimerkiksi liian suuren kuorman tai ohjelmistovirheen ansiosta, ohjataan kutsut varajärjestelmälle palvelun tästä kärsimättä. Mikäli jokainen varajärjestelmä on erikseen provisioitu (virtuaalinen) palvelin, kasvattaa kuormituksentasauksen toteuttaminen luonnollisesti kustannuksia sen mukaan kuinka monta varajärjestelmää tarvitaan.

Konttiratkaisuja hyödynnettäessä varajärjestelmät voidaan toteuttaa luomalla "solmuja" (*nodes*), eli erillisiä palvelimia, jotka suorittavat samoja kontteja (kts. esim. *Docker Swarm, Kubernetes*). Serverless-arkkitehtuuria hyödyntämällä huoli varajärjestelmien ja kuormituksentasaimien provisioinnista ja ylläpidosta siirtyy kuitenkin käyttäjältä piiloon palveluntarjoajalle, ilman ylimääräisiä kustannuksia.

Kappaleen alussa mainitun teoreettisen esimerkin lisäksi Adzic ja Chatley (2017) suorittivat kaksi tapaustutkimusta. Molemmat tarkastelluista yrityksistä olivat saavuttaneet serverless-arkkitehtuuriin siirtymällä huomattavia säästöjä palvelinkustannuksissa. Ensimmäinen yrityksistä pääsi noin 66% säästöihin, kun taas toinen jopa noin 95% säästöihin. Erityisesti jälkimmäisen tapauksessa säästöt johtuivat suurilta osin tyhjäkäynnin eliminoimisesta. Tapauksen järjestelmä oli yllättäviin korkean kuorman piikkeihin taipuvainen sosiaalisen median palvelu, joten järjestelmän oli oltava valmis reagoimaan tähän skaalautumalla automaattisesti. Saatavilla olleet *auto scaling*[1]-ratkaisut eivät kuitenkaan kyenneet skaalaamaan järjestelmää riittävän nopeasti, joten kynnys skaalaamiselle jouduttiin asettamaan tavanomaista matalammaksi. Tämä johti siihen, että laskenta-aikaa varattiin monesti turhaan, johtaen turhiin kustannuksiin.

Villamizar ym. (2017) suorittivat niin ikään tapaustutkimuksen vertaillakseen serverless-arkkitehtuurin kustannuksia vaihtoehtoisiin järjestelmäarkkitehtuureihin. Tarkastelun kohteena oli erään ohjelmistoyrityksen kehitteillä oleva lainojen takaisinmaksusuunnitelmien muodostamiseen suunniteltu järjestelmä. Järjestelmästä valittiin koetta varten kaksi alijärjestelmää ja molemmista toteutettiin neljä implementaatiota: kaksi monoliittista implementaatiota (kahdella eri Java-viitekehyksellä), yksi serverless-implementaatio sekä yksi implementaatio mikropalvelu-arkkitehtuuria noudattamalla. Kahdesta järjestelmästä ensimmäinen suoritti varsinaisen maksusuunnitelman muodostamisen, kun taas toisen tehtävänä oli olemassaolevan maksusuunnitelman noutaminen ja palauttaminen asiakasohjelmalle. Ensim-

1. alusta provisioi resursseja (esim. virtuaalikoneita) automaattisesti riippuen kuormasta

mäisen järjestelmän suorittama operaatio oli laskennallisesti toista raskaampi, ja tyypillinen pyynnön käsittelyaika pidempi (järjestelmä$_1$: 3000 ms, järjestelmä$_2$: 300 ms).

Suorituskykyvertailua varten kehitettiin kolme skenaariota järjestelmille lähetettävien kutsujen suhteen mukaan: *20/80*, *50/50*, *80/20* – eli 20% kutsuista ensimmäiseen järjestelmään ja 80% toiseen järjestelmään (*20/80*), tasan jakautunut kuorma (*50/50*), sekä 80% kutsuista ensimmäiseen järjestelmään ja 20% toiseen järjestelmään (*80/20*).

Koska monoliittisen ja mikropalvelutoteutuksen kustannuksiin ei vaikuta se, missä suhteessa järjestelmiä kutsutaan, oli näille malleille yhden kiinteän kuukausikustannuksen määrittely verrattain suoraviivaista. Serverless-järjestelmän kustannukset määrittyvät sen sijaan toteutuneen käytön mukaan, joten kuukausihinta vaihtelee sen mukaan, kumpi järjestelmä vastaanottaa enemmän kutsuja. Vertailua helpottaakseen Villamizar ym. laskivat kullekin implementaatiolle *hinta per miljoona kutsua* -arvon.

Villamizarin ym. tulokset ovat samankaltaisia kuin Adzicin ja Chatleyn tutkimuksessaan havaitsemat. Toteuttamalla järjestelmä monoliittisen tai mikropalvelumallin sijaan serverless-arkkitehtuurilla, päädyttiin vertailukohdasta ja skenaariosta[2] riippuen noin 50% – 77% säästöihin palvelinkustannuksissa. Merkittävää on, että vaikka sekä mikropalvelu- että serverless-toteutukset olivat molemmat edullisempia palvelinkustannuksiltaan kuin monoliittinen toteutus, toi mikropalveluratkaisu monoliittiseen ratkaisuun verrattuna mukanaan vain noin 9% – 13% säästöt, kun taas serverless-ratkaisulla säästettiin monoliittisen ratkaisun kustannuksiin verrattuna jopa noin 62% – 77%.

Erityisen tärkeää on huomata, ettei tutkimuksessa havaittua kustannusten laskua voida selittää tyhjäkäynnin eliminoinnilla. Implementaatiot suunniteltiin siten, että ne olivat suorituskyvyltään vastaavia, eli kykenivät ylläpitämään samaa määrää liikennettä (kutsuja sekunnissa). Järjestelmiä ajettiin arvioinnin aikana maksimikapasiteetilla, joten tyhjäkäyntiä ei syntynyt. Kuten edellä mainittiin, saatiin tämän ansiosta kullekin implementaatiolle määriteltyä laskennan todellista hintaa kuvaava “hinta per miljoona kutsua”-arvo. Koska tällä tavoin saatiin vertailtavat teknologiat asetettua reilusti vastakkain, voidaan päätyä johtopäätökseen: sekunti laskentaa on lähtökohtaisesti edullisempaa suorittaa serverless-ympäristössä, kuin

2. Yllä esitellyt *20/80*, *50/50* ja *80/20*.

suorituskyvyltään vastaavassa ei-serverless-ympäristössä. Tämän voisi tulkita johtuvan siitä, että serverless-malli tekee palveluntarjoajalle mahdolliseksi hyödyntää paremmin mittakaavaetuja (*economies of scale*) tehokkaamman resurssien optimoinnin kautta. Tyhjäkäynnin eliminointi on siis palveluntarjoajan harteilla ja säästöt heijastuvat myös asiakkaalle.

4 Kehittäjän tuottavuus ja haasteet

Ylläpito- ja infrastruktuurikustannusten vähentäminen ei ole ainoa positiivinen väite joka serverlessiin usein liitetään. Toinen keskeinen periaate on lupaus siitä, että ohjelmistokehittäjän työ helpottuu. Google kuvaili serverlessiä vuonna 2017 julkaistussa artikkelissa seuraavalla tavalla:

> *Steve Jobs promised a new age of programmer productivity. With serverless, it's here.*
>
> – Google LLC (2017)

Adzic ja Chatley (2017) havaitsivat tapaustutkimuksissaan kohonneen tuottavuuden toteutuneen. Kahdesta tapauksesta molemmissa kehittäjät raportoivat uusien ominaisuuksien kehittämisestä tulleen nopeampaa ja helpompaa, ja toinen tutkituista yrityksistä peräti kymmenkertaisti kuukauden aikana saavutettujen tuotantojulkaisujen määrän. Syyksi mainittiin muun muassa siirtyminen monoliittisesta arkkitehtuurista palveluorientoituneeseen arkkitehtuuriin (*Service-Oriented Architecture, SOA*). Monoliittisen sovelluksen pilkkominen pienempiin paloihin mahdollisti tehokkaamman kehitystyön, sillä työtä kyettiin rinnakkaistamaan tehokkaammin, kun esimerkiksi tuotantoonvienti ei vaatinut koordinointia toisen kehitystiimin kanssa. Palveluorientoituneen arkkitehtuurin hyödyntäminen ei toki vaadi serverlessin käyttöä, mutta vahvaa synergiaa mallien väliltä löytyy: Uuden serverless-funktion luominen on yksinkertainen toimenpide, joten ohjelmiston pilkkominen pieniin loogisiin kokonaisuuksiin (palveluihin) on vaivatonta.

Leitner ym. (2019) selvittivät monimenetelmätutkimuksessaan[1] millä tavoin serverlessiä tällä hetkellä ohjelmistokehittäjien toimesta hyödynnetään, ja mitä etuja ja haittoja mallilla koetaan olevan. Tutkimuksen perusteella kehittäjät todella kokevat tuottavuutensa kasvaneen, suurilta osin infrastruktuurin hallintatyön vähenemisen ansiosta. Kyselyssä havaittiin kuitenkin myös, että serverless-arkkitehtuurin hyödyntäminen vaatii uudenlaisen mentaalisen mallin omaksumisen. Mielenkiintoisena yksityiskohtana mainittakoon, että 51% vastanneis-

1. kirjallisuuskatsaus "harmaaseen" (Garousi, Felderer ja Mäntylä 2016) kirjallisuuteen (blogit ym.), haastatteluja, kysely

ta arvioi mallin omaksumisen olevan helpompaa vähemmän kokeneille kehittäjille – heillä kun ei ole yhtä voimakkaita ennakkokäsityksiä ohjelmistokehityksestä ja -arkkitehtuureista.

Epäkohtiakin Leitner ym. löysivät, ja artikkelissa mainitaan monilta osin samoja asioita kuin van Eykin ym. (2017) kaksi vuotta aikaisemmin julkaisemassa artikkelissa. Molemmat artikkelit mainitsevat kehittäjän näkökulmasta haasteeksi serverless-ympäristöön kehitettyjen funktioiden testaamisen. Erityiseksi haasteeksi muodostuu järjestelmän toisintaminen paikallisessa ajoympäristössä eli kehittäjän työasemalla. Tämän takia järjestelmän osien yhteistoiminnan testausta eli integraatiotestausta on käytännössä lähes mahdotonta suorittaa paikallisesti. Leitnerin ym. (2019) tutkimuksessa selvisikin, että paikallisesti suoritetaan lähinnä yksikkötestejä (*unit tests*), jotka testaavat funktioiden liiketoimintalogiikkaa. Funktioiden yhteistoiminnan testausta varten käytännöksi vaikuttaa muodostuneen erillisen kehitysympäristön perustaminen. Palveluun siis perustetaan yksi tai useampia erillisiä käyttäjätilejä, joille järjestelmä toisinnetaan. Näissä ympäristöissä järjestelmää voidaan kehittää ja testata ilman huolta siitä, että tuotantojärjestelmän toiminta häiriintyy.

Suljettu ajoympäristö aiheuttaa testausongelmien lisäksi myös uhan *vendor lock-in*-ilmiöstä. Sekä van Eyk ym. (2017) että Leitnerin ym. (2019) haastattelemat ohjelmistokehittäjät toivat esille tämän vaaralliseksi kokemansa ilmiön. Ilmiö johtuu siitä, että Amazonin, Googlen ja Microsoftin FaaS-ympäristöt ja -rajapinnat eivät lähtökohtaisesti ole yhteensopivia keskenään. Mikäli käyttäjä päättää vaihtaa serverless-toimittajaa, on sovelluspinon siirtäminen toimittajalta toiselle todennäköisesti vaivalloista ja kallista. Lisähaasteen tuovat alustan FaaS-palvelua tukevat palvelut (esim. tietokanta- ja viestijonopalvelut), jotka eivät niin ikään ole rajapinnoiltaan yhteensopivia.

Ilmiön välttämiseksi van Eyk ym. (2017) esittävät alustariippumattoman FaaS-määritelmän kehittämistä. Tällainen määritelmä mahdollistaisi sen, että serverless-funktio voitaisiin toteuttamisen jälkeen asentaa minkä tahansa palveluntarjoajan FaaS-ympäristöön. Lähelle tätä kuvausta pääsee Serverless Framework-viitekehys, joka lupaa mahdollistavansa ylläkuvatun kaltaisen kehityskokemuksen.

> *All your cloud services are now compatible with one another: share cross-cloud functions and events with AWS Lambda, Microsoft Azure, IBM OpenWhisk and Google Cloud Platform.*
>
> – Serverless Inc. (2019)

Viimeisenä haasteena mainittakoon serverless-funktioiden sopimattomuus pitkäkestoisen laskennan toteuttamiseen. AWS Lambda -alustalla pisin tapahtuman käsittelyyn sallittu aika on 15 minuuttia (Amazon 2019) kun taas Google Cloud Functions -alustalla vastaava rajoite on yhdeksän minuuttia (Google 2019). Mikäli tavoitteena on esimerkiksi pakata pitkiä videotiedostoja vähemmän tilaa vievään formaattiin, ei pelkkä FaaS ole tämän rajoitteen ansiosta järkevin työkalu tavoitteen saavuttamiseen. Tämän ongelman voi kiertää hyödyntämällä esimerkiksi Amazonin tarjoamaa *Elastic Transcoder*-videoenkoodauspalvelua (Amazon 2019) yhdessä AWS Lambdan kanssa, pysyen näin edelleen serverless-mallin sisällä.

Jos taas kyseessä on erikoistuneempi käyttötapaus kuten tieteellinen laskenta, on perusteltua hyödyntää tapaukseen paremmin soveltuvaa ratkaisua kuten AWS Batch (Amazon 2019), joka mahdollistaa pitkäkestoisten töiden suorittamisen hallitussa ympäristössä. Täysin sopimaton malli ei FaaS tieteelliseenkään laskentaan kuitenkaan ole, etenkään jos ongelma on jaettavissa pienempiin osaongelmiin (Spillner, Mateos ja Monge 2018). Osaongelmia ratkaisevat funktiot voidaan saada toimimaan yhdessä joko yksinkertaisesti kutsumalla funktioita funktioiden sisältä kuten Spillnerin ym. artikkelissa, tai esimerkiksi muodostamalla funktioista tilakoneen käyttämällä AWS Step Functions -palvelua (Amazon 2019).

5 Yhteenveto

Tutkielmassa esiteltiin aluksi serverless-mallin keskeinen ajatus – abstraktiotason nosto ja hallintatyön siirtäminen palveluntarjoajalle – minkä jälkeen tarkasteltiin sitä, miten serverless rakentaa aiemman mallin (mikropalvelut) periaatteiden päälle. Tämän jälkeen käytiin läpi sitä, millä tavoin yksi mallin keskeisistä lupauksista (edullisemmat käyttökustannukset) toteutuu. Aihetta tarkasteltiin kahden tapaustutkimuksen kautta ja lopputulos oli lupauksen kannalta positiivinen: molemmat tarkastelluista tutkimuksista päätyivät puoltamaan väitettä. Erityisen merkittävä havainto oli se, että serverless-laskenta vaikuttaisi olevan verrokkeja lähtökohtaisesti edullisempaa, vaikka yhtälöstä poistettaisiin tyhjäkäynnin eliminoinnin tuoma etu.

Viimeisessä varsinaisessa kappaleessa tarkasteltiin sitä, miten serverless vaikuttaa kehittäjän tuottavuuteen ja sitä, millaisia haasteita malli pitää sisällään. Käytetty lähdemateriaali tuki väitettä kohoavasta tuottavuudesta, mutta toi esille myös epäkohtia. Keskeisiksi haasteiksi nousivat muun muassa integraatiotestauksen haastavuus sekä suljetun ympäristön haitat (kuten *vendor lock-in*).

Serverless on ilmiönä vielä tuore, ja siten tieteellisessä tutkimuksessa aliedustettu. Tärkeistä ja mielenkiintoisista tutkimusaiheista ei kuitenkaan olisi puutetta. Vaikka serverless-kustannuksista on jo nyt julkaistu joitakin tutkimuksia, olisi aihetta tärkeää tutkia enemmänkin, etenkin kustannussuunnittelun näkökulmasta. Tällä hetkellä suuri osa kyseistä aihetta käsittelevästä tutkimustyöstä lienee tieteellisen tutkimuksen sijaan teknologiayritysten toteuttamia sisäisiä selvityksiä.

Kustannussuunnittelun lisäksi mielenkiintoista olisi nähdä lisää tutkimusta siitä, miten serverless-mallin optimaalinen hyödyntäminen voi tehostaa sovelluskehittäjän työtä, vauhdittaa tuotteen saattamista markkinoille, ja tällä tavoin tuoda mallia soveltavalle ohjelmistotalolle kilpailuetua. Leitnerin tuore tutkimus sivusi aihetta ja on erinomainen alku, mutta aiheeseen olisi syytä palata muutaman vuoden kuluttua käytäntöjen ja työkalujen kehityttyä.

Serverless on oikein käytettynä erinomainen työkalu kustannustehokkaan ohjelmistokehitysprosessin toteuttamiseen. Kaikkiin käyttökohteisiin malli ei kuitenkaan oletettavasti sovellu, ja tärkeää olisikin selvittää, missä tapauksissa serverlessin käyttö on perusteltua sekä hyödyllistä ja missä taas ei. Tästä voitaisiin jatkaa selvittämällä niitä syitä, jotka tekevät serverlessistä sopimattoman tiettyihin tarkoituksiin, ja kenties pohtimalla sitä, miten näitä syitä voitaisiin poistaa.

Lähteet

Adzic, Gojko, ja Robert Chatley. 2017. “Serverless Computing: Economic and Architectural Impact”. Teoksessa *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* 884–889. ESEC/FSE 2017. ACM. ISBN: 978-1-4503-5105-8. doi:10.1145/3106237.3117767.

Netflix & AWS Lambda Case Study. 2014. Amazon Web Services, Inc. Viitattu 22. huhtikuuta 2019. https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/.

AWS | Amazon Elastic Transcoder - Media & Video Transcoding in the Cloud. 2019. Amazon Web Services, Inc. Viitattu 17. huhtikuuta 2019. https://aws.amazon.com/elastictranscoder/.

AWS Batch — Easy and Efficient Batch Computing Capabilities - AWS. 2019. Amazon Web Services, Inc. Viitattu 17. huhtikuuta 2019. https://aws.amazon.com/batch/.

AWS Lambda – Pricing. 2019. Amazon Web Services, Inc. Viitattu 14. helmikuuta 2019. https://aws.amazon.com/lambda/pricing/.

AWS Lambda Limits - AWS Lambda. 2019. Amazon Web Services, Inc. Viitattu 17. huhtikuuta 2019. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.

AWS Step Functions. 2019. Amazon Web Services, Inc. Viitattu 17. huhtikuuta 2019. https://aws.amazon.com/step-functions/.

Publishing Messages in Amazon SNS - AWS SDK for JavaScript. 2019. Amazon Web Services, Inc. Viitattu 13. huhtikuuta 2019. https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/sns-examples-publishing-messages.html.

Using Amazon SNS for System-to-System Messaging with an AWS Lambda Function as a Subscriber - Amazon Simple Notification Service. 2019. Amazon Web Services, Inc. Viitattu 6. maaliskuuta 2019. https://docs.aws.amazon.com/sns/latest/dg/sns-lambda-as-subscriber.html.

Balalaie, Armin, Abbas Heydarnoori ja Pooyan Jamshidi. 2016. "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report". Teoksessa *Advances in Service-Oriented and Cloud Computing,* toimittanut Antonio Celesti ja Philipp Leitner, 201–215. Communications in Computer and Information Science. Springer International Publishing. ISBN: 978-3-319-33313-7.

Balalaie, Heydarnoori ja Jamshidi. 2016. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". *IEEE Software* 33, numero 3 (toukokuu): 42–52. ISSN: 0740-7459. doi:10.1109/MS.2016.64.

Baldini, Ioana, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell ym. 2017. "Serverless Computing: Current Trends and Open Problems". *arXiv:1706.03178 [cs]* (kesäkuu). arXiv: 1706.03178 [cs].

Garousi, Vahid, Michael Felderer ja Mika V. Mäntylä. 2016. "The Need for Multivocal Literature Reviews in Software Engineering: Complementing Systematic Literature Reviews with Grey Literature". Teoksessa *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering,* 26:1–26:6. EASE '16. ACM. ISBN: 978-1-4503-3691-8. doi:10.1145/2915970.2916008.

Steve Jobs Promised a New Age of Programmer Productivity. With Serverless, It's Here. 2017. Google LLC. Viitattu 9. maaliskuuta 2019. https://cloud.withgoogle.com/build/productivity/serverless-computing-accelerates-programmer-productivity/.

Quotas | Cloud Functions Documentation | Google Cloud. 2019. Google LLC. Viitattu 17. huhtikuuta 2019. https://cloud.google.com/functions/quotas.

Kemp, Jeremy. 2019. *Gartner Hype Cycle.svg.* Viitattu 13. huhtikuuta 2019. https://commons.wikimedia.org/wiki/File:Gartner_Hype_Cycle.svg.

Leitner, Philipp, Erik Wittern, Josef Spillner ja Waldemar Hummer. 2019. “A Mixed-Method Empirical Study of Function-as-a-Service Software Development in Industrial Practice”. *Journal of Systems and Software* 149 (maaliskuu): 340–359. ISSN: 0164-1212. doi:10.1016/j.jss.2018.12.013.

McGrath, Garrett, ja Paul R. Brenner. 2017. “Serverless Computing: Design, Implementation, and Performance”. Teoksessa *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* 405–410. Kesäkuu. doi:10.1109/ICDCSW.2017.36.

Serverless Framework - Build applications on AWS Lambda, Google CloudFunctions, Azure Functions, AWS Flourish and more. 2019. Serverless, Inc. Viitattu 11. huhtikuuta 2019. https://serverless.com/framework/.

Singleton, Andy. 2016. “The Economics of Microservices”. *IEEE Cloud Computing* 3, numero 5 (syyskuu): 16–20. ISSN: 2325-6095. doi:10.1109/MCC.2016.109.

Smith, David, ja Ed Anderson. 2018. *Hype Cycle for Cloud Computing, 2018,* heinäkuu.

Spillner, Josef, Cristian Mateos ja David A. Monge. 2018. “FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC”. Teoksessa *High Performance Computing,* toimittanut Esteban Mocskos ja Sergio Nesmachnow, 154–168. Communications in Computer and Information Science. Springer International Publishing. ISBN: 978-3-319-73353-1. doi:https://doi.org/10.1007/978-3-319-73353-1_11.

van Eyk, Erwin, Alexandru Iosup, Simon Seif ja Markus Thömmes. 2017. “The SPEC Cloud Group’s Research Vision on FaaS and Serverless Architectures”. Teoksessa *Proceedings of the 2Nd International Workshop on Serverless Computing,* 1–4. WoSC ’17. ACM. ISBN: 978-1-4503-5434-9. doi:10.1145/3154847.3154848.

van Eyk, Erwin, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță ja Alexandru Iosup. 2018. “Serverless Is More: From PaaS to Present Cloud Computing”. *IEEE Internet Computing* 22, numero 5 (syyskuu): 8–17. ISSN: 1089-7801. doi:10.1109/MIC.2018.053681358.

Villamizar, Mario, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas ym. 2017. “Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures”. *Service Oriented Computing and Applications* 11, numero 2 (kesäkuu): 233–247. ISSN: 1863-2394. doi:10.1007/s11761-017-0208-y.

Vultr: High Performance SSD Cloud. 2019. Vultr Holdings Corporation. Viitattu 14. helmikuuta 2019. https://www.vultr.com/pricing/.