Funktio palveluna -pilvipalvelumalli: skaalautuvuus, suorituskyky ja haasteet

Turun yliopisto
Tietotekniikan laitos
LuK-tutkielma
Tietojenkäsittelytieteet
Helmikuu 2026
Olli Tanhuanpää

TURUN YLIOPISTO
Tietotekniikan laitos

Olli Tanhuanpää: Funktio palveluna -pilvipalvelumalli: skaalautuvuus, suorituskyky ja haasteet

LuK-tutkielma, 20 s.
Tietojenkäsittelytieteet
Helmikuu 2026

Funktio palveluna (Function as a Service, FaaS) on serverless-arkkitehtuuriin perustuva pilvipalvelumalli, jossa sovelluslogiikka toteutetaan tapahtumavetoisina funktioina ilman omaa infrastruktuuria. Malli on herättänyt kasvavaa kiinnostusta sen lupaaman automaattisen skaalautuvuuden, kustannustehokkuuden ja kehitysprosessin yksinkertaistumisen vuoksi. FaaS:n vaikutukset suorituskykyyn ja käytännön rajoitteet eivät kuitenkaan ole ongelmattomia, mikä tekee sen arvioinnista ajankohtaista erityisesti pilvipalveluiden ja modernien sovellusarkkitehtuurien näkökulmasta.
Tutkielma on kirjallisuuskatsaus, jossa tarkastellaan FaaS-mallia osana pilvipalveluekosysteemiä. Aineistona käytetään erityisesti vertaisarvioituja katsausartikkeleita ja empiirisiä suorituskykymittauksia. Katsauksessa tutkitaan, miten FaaS vaikuttaa sovellusten skaalautuvuuteen ja suorituskykyyn, millaisia haasteita siihen liittyy sekä mitkä ovat sen keskeiset kehityssuunnat ja tulevaisuuden mahdollisuudet.
Tulosten perusteella FaaS skaalautuu hyvin tapahtumavetoisiin, lyhytkestoisiin ja epäsäännöllisesti kuormittuviin tehtäviin, mutta kylmäkäynnistys, tilattomuus, havaittavuusongelmat ja palveluntarjoajariippuvuus rajoittavat sen käyttöä erityisesti pitkäkestoisissa ja raskaissa työkuormissa. Kehityssuunnat, kuten kevyemmät suoritusympäristöt, ennustava skaalaus, paremmat orkestrointi- ja tilaratkaisut sekä reunaympäristöjen tuki, laajentavat FaaS:n soveltuvuutta entistä vaativampiin käyttökohteisiin.

Asiasanat: Function as a Service, serverless, pilvipalvelut, skaalautuvuus, suorituskyky, kylmäkäynnistys

Sisällys

1 Johdanto

Funktio palveluna (Function as a Service, FaaS) on pilvipalvelumalli, joka on herättänyt merkittävää kiinnostusta ohjelmistokehityksen alalla viime vuosina, mikä heijastaa sen kasvavaa merkitystä pilvipalveluiden ekosysteemissä. Sen tarjoama automaattinen skaalautuvuus, kustannustehokkuus ja kyky nopeuttaa kehitysprosessia tekevät siitä houkuttelevan vaihtoehdon monenlaisille sovelluksille ja palveluille. FaaS mahdollistaa kehittäjille keskittymisen sovelluslogiikkaan ilman tarvetta huolehtia alustan ylläpidosta tai resurssien hallinnasta, tukien nykyaikaisen ohjelmistokehityksen paradigmaa, jossa sovellusten nopea toimittaminen markkinoille on tärkeää. [1]

FaaS-teknologian, osana laajempaa serverless-arkkitehtuuria, etuna on, että se tarjoaa dynaamisen ympäristön, jossa sovellukset suoritetaan tilapäisissä konteissa vastauksena tapahtumiin tai pyyntöihin [1]. Tämä mahdollistaa resurssien käytön optimoinnin ja kustannusten minimoimisen. Tutkimuksen tavoitteena on tarjota ymmärrys FaaS:n potentiaalista, teknisistä haasteista ja sen roolista pilvipalveluiden ekosysteemissä. Tutkimuskysymykset ovat:

TK1: Miten FaaS vaikuttaa sovellusten skaalautuvuuteen ja suorituskykyyn?

TK2: Mitä haasteita FaaS-teknologiaan liittyy?

TK3: Mitkä ovat FaaS-teknologian kehityksen tärkeimmät suuntaukset ja tulevaisuuden mahdollisuudet?

Tutkielman tietoperusta on koottu kirjallisuuskatsauksen avulla. Kirjallisuutta haettiin Google Scholar -palvelusta rajaamalla lähteet serverless- ja FaaS-arkkitehtuureihin, suorituskyvyn mittaukseen, haasteisiin ja käyttötapauksiin keskittyviin artikkeleihin käyttämällä seuraavia hakulauseita: *"Function-as-a-Service scalability performance"*, *"Function-as-a-Service challenges"* ja *"Challenges in serverless computing"*. Hakujen perusteella valittiin erityisesti vertaisarvioituja artikkeleita, katsausartikkeleita ja konferenssijulkaisuja, joita täydennettiin pilvipalveluntarjoajien teknisellä dokumentaatiolla.

Tutkielma on jaettu useisiin alalukuihin, jotka keskittyvät FaaS-palveluiden perusteisiin, skaalautuvuuden ja suorituskyvyn analyysiin, sovelluksiin ja käyttötapauksiin, sekä tulevaisuuden näkymiin. Aluksi, luvussa 2 määritellään FaaS:n perusteet ja tekninen arkkitehtuuri, jonka jälkeen vertaillaan FaaS:n eroja perinteisiin pilvipalvelumalleihin. Seuraavaksi luvussa 3 tarkastellaan kirjallisuuskatsauksen kautta FaaS:n skaalautuvuutta, latenssia ja kylmäkäynnistystä, muita keskeisiä haasteita sekä tyypillisiä käyttötapauksia. Lopuksi luvussa 4 esitetään yhteenveto, jossa vastataan tutkimuskysymyksiin kirjallisuuskatsauksen perusteella ja tarkastellaan FaaS-teknologian tulevaisuuden mahdollisuuksia.

2 Funktio palveluna pilvipalvelumallina

Pilvipalveluita tarjotaan yleisesti useiden eri palvelumallien kautta. Se, kuinka suuri osa järjestelmän hallinnasta jää asiakkaalle ja mitä abstraktiotasoa hyödynnetään, riippuu valitusta mallista. Infrastruktuuri palveluna (Infrastructure as a Service, IaaS) tarjoaa käyttäjälle valmiin infrastruktuurin, jolloin alustojen ja sovellustason hallinnan vastuu jää käyttäjälle. Alusta palveluna (Platform as a Service, PaaS) puolestaan ei ainoastaan tarjoa infrastruktuuria, vaan sisältää myös kehitysalustan, mikä vapauttaa käyttäjän pohjana olevan alustan hallinnan taakasta. Nämä perinteiset pilvipalvelumallit eroavat selvästi ominaisuuksiltaan ja käyttötapauksiltaan FaaS-mallista, joka mahdollistaa yksittäisten toimintojen suorittamisen ilman infrastruktuurin tai pohjana olevien alustojen hallintaa. Siirtyminen IaaS-mallista FaaS-malliin tyypillisesti vähentää käyttäjän vastuuta infrastruktuurin hallinnasta, mutta puolestaan vähentää myös mahdollisuuksia alustan räätälöityvyyteen.

Huomion arvoista on, että vaikka usein FaaS- ja serverless-termejä termejä käytetään synonyymeinä, ne eivät kuitenkaan tarkoita täysin samaa asiaa. Serverless sisältää laajemman joukon palveluita ja ominaisuuksia, kuten tietokantoja, viestijonoja ja autentikaatiopalveluita, jotka kaikki toimivat ilman palvelimien hallinnan tarvetta. FaaS taas keskittyy nimenomaan koodin suorittamiseen, mikä on vain yksi Serverless-arkkitehtuurin monista komponenteista.

Tunnetuimmat kaupalliset FaaS-palvelut ovat AWS Lambda, Google Cloud Functions ja Microsoft Azure Functions. Näiden lisäksi on tarjolla myös avoimen lähdekoodin palveluja, kuten OpenFaaS, OpenWhisk, Kubeless ja Fission, jotka pohjautuvat Kubernetes-orkestrointiin ja joiden avulla on mahdollista käyttää FaaS-palveluita omalla infrastruktuurilla.

2.1 Palvelumallit

IaaS tarjoaa käyttäjälle virtuaalisia laskentaresursseja, kuten virtuaalikoneita, tallennustilaa ja verkkokapasiteettia. Palveluntarjoaja vastaa infrastruktuurista, mutta käyttäjä on vastuussa käyttöjärjestelmän, sovellusten sekä kaiken sovellusympäristön hallinnasta ja ylläpidosta. Vaikka tämä malli tarjoaa joustavan infrastruktuurin hallinnan, siihen liittyy myös merkittävä hallinnollinen vastuu, kuten resurssien skaalauksen ja ylläpidon tehtävät [2].

IaaS sopii erityisesti tilanteisiin, joissa organisaatiolla on tarve kontrolloida infrastruktuuria, mutta ei halua investoida fyysisiin palvelimiin. Esimerkiksi yritys, joka kehittää suurta datakeskusta vaativaa sovellusta, kuten omaa ERP-järjestelmää (Enterprise Resource Planning), voisi käyttää jotakin IaaS-palvelua, esimerkiksi Amazon Web Servicesin (AWS) EC2-instansseja. Käytännössä yritys tällöin vuokraisi virtuaalikoneita, joihin se asentaisi käyttäjärjestelmän (esim. Linux tai Windows Server), tietokannat ja muut sovellukset itse sekä vastaisi myös näiden ylläpidosta. Tämä antaa mahdollisuuden räätälöidä infrastruktuuri tarkasti tarpeiden mukaan, esimerkiksi valitsemalla tietyn määrän prosessoriytimiä tai muistia. Yritys joutuisi tällöin myös vastaamaan itse päivityksistä, tietoturvapäivityksistä ja resurssien skaalauksesta.

PaaS tarjoaa kehittäjille valmiita kehitys- ja suoritusympäristöjä. Tässä mallissa palveluntarjoaja hallitsee infrastruktuurin lisäksi käyttöjärjestelmiä, sovellusajoympäristöjä sekä tietokantaympäristöjä, jolloin kehittäjä voi keskittyä sovelluskehi-

tykseen eikä perusinfrastruktuurin ylläpitoon. Kuitenkin PaaS-mallissa kehittäjän tulee määrittää ja ylläpitää sovellusten rakennetta, konfiguraatioita sekä huolehtia sovellusten skaalauksen asetuksista ja säädöistä alustalla [3].

PaaS taas sopii hyvin kehittäjille, jotka haluavat keskittyä sovellusten kehittämiseen ja tarjoamiseen ilman infrastruktuurin hallintaa. Esimerkkitapauksena tällaisesta voidaan mainita startup-yritys, joka kehittää mobiilisovellusta reaaliaikaiseen data-analytiikkaan asiakkailleen. He käyttävät Google App Engineä, joka on PaaS-palvelu, tarjoamaan valmiin ympäristön sovelluksen suorittamiseen. Kehittäjät lataavat sovelluskoodinsa ja määrittelevät tarvittavat konfiguraatiot, kuten tietokantaintegraatiot (esim. Google Cloud SQL). Google App Engine huolehtii automaattisesti palvelimien hallinnasta, käyttöjärjestelmän päivityksistä ja skaalauksesta, esimerkiksi lisäämällä resursseja, kun sovelluksen käyttö kasvaa hetkellisesti vaikkapa verkkokaupan alennuskampanjan aikana. Kehittäjät kuitenkin vastaavat sovelluksen koodin optimoinnista ja sen arkkitehtuurin suunnittelusta.

FaaS tarjoaa vielä korkeamman abstraktiotason pilvipalvelumallin. Siinä käyttäjä voi suorittaa yksittäisiä ohjelmakoodin osia eli funktioita, tapahtumavetoisesti ilman oman infrastruktuurin hallintaa tai ylläpitoa. FaaS abstrahoi sovelluksen taustalla toimivat palvelimet, minkä ansiosta kehittäjän ei tarvitse huolehtia resursoinnista, skaalauksesta eikä palvelinten käytöstä. Nämä kaikki ovat palveluntarjoajan vastuulla. Lisäksi FaaS on kulutusperusteinen palvelumalli, jossa käyttäjä maksaa ainoastaan funktion suorittamiseen käytetyistä resursseista ja ajasta, eikä esimerkiksi toimintavalmiuden ylläpitämisestä tyhjäkäynnillä olevien palvelinten osalta. Mallin tarkoitus on siis vähentää käyttäjän hallinnollista vastuuta verrattuna IaaS- ja PaaS-malleihin, tarjoten joustavamman ja kustannustehokkaamman vaihtoehdon erityisesti sovelluksille, joissa on epäsäännöllinen käyttökuorma. FaaS:lla on kuitenkin tiettyjä rajoituksia ja haasteita esimerkiksi pitkäkestoisten suoritusten osalta [4]. Taulukko 2.1 kuvaa kolmen palvelumallin eroa vastuun jakautumisen näkökulmasta.

Taulukko 2.1: IaaS-, PaaS- ja FaaS-palvelumallien vastuunjako

Vastuun osa-alue	IaaS	PaaS	FaaS
Verkko	Palveluntarjoaja	Palveluntarjoaja	Palveluntarjoaja
Tallennustila	Palveluntarjoaja	Palveluntarjoaja	Palveluntarjoaja
Palvelimet	Palveluntarjoaja	Palveluntarjoaja	Palveluntarjoaja
Virtualisointi	Palveluntarjoaja	Palveluntarjoaja	Palveluntarjoaja
Käyttöjärjestelmä	Käyttäjä	Palveluntarjoaja	Palveluntarjoaja
Ajoalusta	Käyttäjä	Palveluntarjoaja	Palveluntarjoaja
Skaalaus	Käyttäjä	Osittain	Palveluntarjoaja
Sovellukset ja data	Käyttäjä	Käyttäjä	Käyttäjä (funktiot)
Liiketoimintalogiikka	Käyttäjä	Käyttäjä	Käyttäjä

FaaS soveltuu parhaiten tapahtumavetoisiin ja lyhytkestoisiin laskentatehtäviin, joissa resurssien käyttö on epäsäännöllistä. Esimerkiksi verkkokauppa-alusta voisi käyttää AWS Lambda -FaaS-palvelua kuvien automaattiseen käsittelyyn. Kun asiakas lataa tuotteen kuvan alustalle, se laukaisee funktion, joka muuttaa kuvan kokoa, optimoi sen ja tallentaa sen Amazon S3 -tallennustilapalveluun. Kehittäjä kirjoittaa vain kuvankäsittelylogiikan sisältävän funktion, eikä hänen tarvitse huolehtia palvelimien ylläpidosta, skaalauksesta tai käyttöjärjestelmästä. Jos kuvia ladataan vain ajoittain, palvelu skaalautuu automaattisesti nollaan, kun kuvankäsittelylle ei ole tarvetta, eikä kustannuksia synny tyhjäkäynnistä. Tämä vähentää kustannuksia verrattuna perinteiseen palvelinmalliin ja sopii erinomaisesti epäsäännöllisiin kuormiin.

2.2 FaaS:n toimintamalli ja arkkitehtuurin rakenne

FaaS-palvelussa sovelluksen toiminta perustuu tyypillisesti seuraaviin peruspalikoihin ja toimintaperiaatteisiin:

- **Tapahtumat ja tapahtumankäsittelijät**: Funktio käynnistyy jonkin ulkoisen tapahtuman perusteella, kuten HTTP-pyynnöllä, tiedostojen lisäyksellä tallennustilaan tai ajastetulla aikarajalla.

- **Orkestraattori ja resurssinhallinta**: Alusta huolehtii sovelluksen funktioiden suorituksesta, käynnistämisestä, lopettamisesta sekä resurssien varaamisesta tekniikoilla kuten kontit tai virtualisointi. Kontit ovat kevyitä, eristettyjä ympäristöjä, jotka sisältävät kaiken tarvittavan koodin suorittamiseen, mukaan lukien kirjastot ja riippuvuudet. Tämä mahdollistaa nopean skaalauksen ja resurssien tehokkaan hyödyntämisen, kun useita konteissa ajettavia toimintoja voidaan hallita rinnakkain ilman, että fyysisiä palvelimia tarvitsee mukauttaa erikseen.

- **Eristetyt suoritusalustat**: Jokainen funktio suoritetaan omassa eristetyssä ympäristössään, tavallisesti konttiteknologian tai kevyiden virtuaalikoneiden avulla. Funktioilla ei pääsääntöisesti ole pysyvää tilaa, vaan ne ovat tilattomia (engl. stateless).

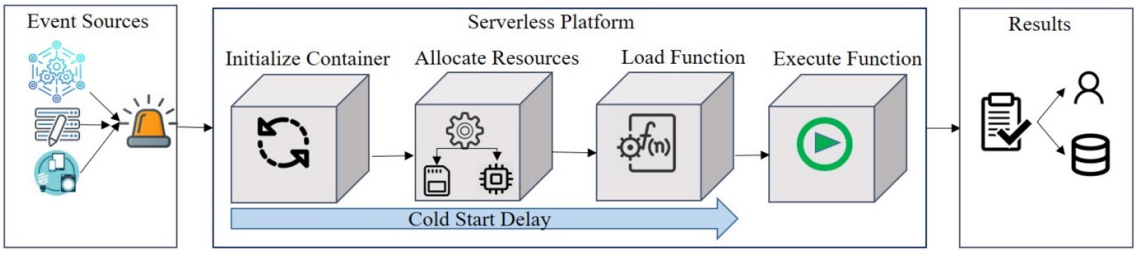
- **Funktiorekisteri ja hallinta**: Alusta ylläpitää listaa julkaistuista funktioista, joiden suoritukset voidaan laukaista tapahtumilla.

- **Lokien keräys ja monitorointi**: Suorituksista kerätään lokitietoa ja mahdollisia virheitä jatkokäsittelyä varten.

Ohjelmointikielten osalta FaaS-palvelut mahdollistavat koodin toteutuksen usealla eri kielellä riippuen siitä, mitä eri palveluntarjoajat tukevat. Käyttäjän vastuulla on vain liiketoimintalogiikan (funktion) ohjelmointi, jonka jälkeen se ladataan alustalle ja määritellään millä tapahtumalla funktio käynnistyy.

2.3 Toimintaprosessi

FaaS-alustan suorituksen prosessi rakentuu tyypillisesti kolmen päävaiheen ympärille: **alustus**, **suoritus** ja **sammutus**. Nämä vaiheet määrittävät, miten ja missä järjestyksessä palvelu käynnistää funktion, suorittaa sen ja lopulta vapauttaa resurssit



Kuva 2.1: FaaS-suoritusprosessi [6]

käytön jälkeen [5]. Suorituksen peruslogiikka pysyy samana kaikilla palveluntarjoajilla mutta yksityiskohdat voivat vaihdella.

1. **Ulkoinen tapahtuma laukaisee funktion suorituksen.**

   Funktio aktivoituu esimerkiksi silloin, kun uusi tiedosto lisätään pilvitallennustilaan tai HTTP-pyyntö saapuu. Tapahtuma laukaisee koko suoritusprosessin.

2. **Alustusvaihe: uuden suoritusympäristön valmistelu.**

   Tässä vaiheessa alustetaan tarvittavat laajennukset, käynnistetään ohjelmointikielen ajonaikainen ympäristö ja suoritetaan funktion ulkopuolinen staattinen koodi. Esimerkiksi riippuvuuksien lataus ja yhteyksien alustaminen kuten tietokantaan yhdistäminen.

   Alustus voidaan jakaa tarkemmin kolmeen alavaiheeseen: *kontin alustaminen*, *resurssien allokointi* ja *funktion lataus*. Näitä vaiheita kuvataan kuvassa 2.1. Tämä prosessi aiheuttaa niin sanotun kylmäkäynnistysviiveen (engl. cold start delay), joka voidaan välttää lämpimässä käynnistyksessä (engl. warm start), mikäli aiempi suoritusympäristö on edelleen aktiivinen. Monet palveluntarjoajat mahdollistavat myös funktioiden pitämisen lämpimänä jatkuvasti lisäkustannuksia vastaan.

3. **Suoritusvaihe: ohjelmakoodin suoritus.**

Varsinainen funktion logiikka suoritetaan, kun ympäristö on valmis. Alusta kutsuu määriteltyä käsittelijäfunktiota, joka ottaa vastaan tapahtuman ja kontekstin. Esimerkiksi kuvankäsittelyfunktio voi muokata ladattua kuvaa ja tallentaa sen uudelleen.

4. **Sammutusvaihe: resurssien vapautus.**

   Suorituksen päätyttyä ja kun funktiota ei enää tarvita, alusta voi sulkea ympäristön. Tämä vaihe mahdollistaa resurssien vapauttamisen sekä mahdollisten jälkitoimintojen suorittamisen. Resurssien automaattinen hallinta auttaa pitämään kustannukset alhaisina erityisesti epäsäännöllisessä kuormituksessa.

3 Kirjallisuuskatsaus

Tässä kirjallisuuskatsauksessa käydään ensin läpi FaaS:n skaalautuvuutta, jonka jälkeen tarkastellaan latenssia ja erityisesti kylmäkäynnistyksiä. Tämän jälkeen käsitellään muita keskeisiä haasteita ja lopuksi tyypillisiä käyttötapauksia. Katsaus nojaa sekä yleisesityksiin että FaaS:n suorituskykyä mittaaviin tutkimuksiin. Näiden perusteella pyritään muodostamaan mahdollisimman selkeä kokonaiskuva siitä, mitä asioita FaaS:sta on käytännössä mitattu, millaisilla koeasetelmilla, ja millaisiin tuloksiin eri tutkimuksissa on päädytty.

Kirjallisuuden mukaan FaaS skaalautuu kuormapiikkeihin hyvin, koska alusta lisää rinnakkaisia suorituksia automaattisesti. Samalla kuitenkin kylmäkäynnistys ja alustusvaihe tuovat lisäviivettä, joka näkyy erityisesti harvakäyttöisissä funktioissa ja ketjutetuissa työnkuluissa. Tämä on toistuva havainto sekä laajoissa katsauksissa että mittaustutkimuksissa [7], [8]. Kun mitataan kokonaisia työnkulkuja, latenssi kertyy helposti useista peräkkäisistä kutsuista ja ulkoisista palveluista, jolloin pelkkä yksittäisen funktion nopeus ei riitä selittämään käyttäjän kokemaa suorituskykyä [8].

3.1 Skaalautuvuus

Skaalautuvuudella viittaan kykyyn palvella kasvavaa samanaikaista kuormaa lisäämällä suoritusresursseja. FaaS-kontekstissa tämä näkyy siinä, että alusta kasvattaa rinnakkaisten instanssien määrää kuorman noustessa. Siinä kohtaa, jossa saavutetaan yläraja, alkaa kuristus. Tämä voidaan toteuttaa esimerkiksi palauttamalla

HTTP 429 -virhe, joka osoittaa, että pyyntöjen määrä on ylittänyt sallitun rajan tietyssä ajassa. Tarjoajakohtaiset rajat määrittävät käytännön katon: esimerkiksi AWS Lambda asettaa oletuksena samanaikaisuuden ylärajaksi noin 1000 rinnakkaista suoritusta, Azure peruskulutusmallissa skaalaa noin 200 instanssiin (instanssi voi palvella useampaa pyyntöä), ja IBM:llä on vastaavan suuruusluokan rajoitteita; rajan ylittyessä pyyntöjä hylätään tai kuristetaan [9].

Kirjallisuudessa skaalautuvuutta on mitattu kuormitustesteillä, joissa samanaikaisuutta ja pyyntömäärää varioidaan systemaattisesti. Testeissä on tarkasteltu erityisesti instanssien kertymistä, ylösajokestoa, läpimenoa (engl. throughput) ja vasteaikaa (engl. response time). Esimerkiksi eräässä tutkimuksessa, jossa kuormaa generoitiin sadalla rinnakkaisella asiakkaalla, ylösajoaika oli 1–10 sekuntia ja kokonaispyyntömäärät 1000–7000. Mittareina käytettiin instanssien lukumäärää, alasajoaikaa, läpimenoa (req/min) ja mediaanivastetta. Näin pystyttiin tutkimaan miten kapasiteetti kasvaa samanaikaisuuden mukana ja missä pisteessä läpimeno lakkaa kasvamasta eli saavutetaan saturaatio. [9]

Tulosten tarkastelu osoittaa kaksi toistuvaa havaintoa. Ensinnäkin, automaattinen skaalaus toimii kaikilla tarkastelluilla alustoilla ja lisää kapasiteettia kuorman kasvaessa, mutta toteutustapa ja reagointinopeus vaihtelevat tarjoajittain. Esimerkiksi AWS ja IBM skaalaavat instanssien määrää lähes samassa suhteessa kuin rinnakkaisten pyyntöjen määrä, kun taas Azure luottaa siihen, että yksi instanssi pystyy käsittelemään useita pyyntöjä yhtä aikaa [9]. Toiseksi, ylärajat määrittävät skaalautuvuuden katon: kun saturaatio saavutetaan, läpimeno ei enää kasva ja ylimääräiset pyynnöt alkavat epäonnistua tai viivästyä [9].

Skaalautuvuutta on tarkasteltu myös sovellustason asetelmissa. Yksittäisen funktion suorituskykyä mittaavat kokeet raportoivat AWS:n ja GCP:n skaalautuvan lähes lineaarisesti samanaikaisuuden kasvaessa, kun taas Azure ei kaikkien testien mukaan yllä samaan lineaarisuuteen [8]. Sovellustasolla (HTTP-rajapinnat, ulkoiset tie-

tovarastot) on havaittu, että FaaS-pohjainen toteutus suoriutuu korkeilla kuormilla hyvin juuri skaalautuvuuden ansiosta, ja että AWS:n toteutus saavutti keskimäärin pienemmät vasteajat ja korkeammat samanaikaisuusrajat kuin Azure vastaavassa työnkulussa [8].

Scheunerin ja Leitnerin katsauksessa selviää, että samanaikaisuuskäyttäytyminen on olennainen osa FaaS-suorituskyvyn arviointia ja että sitä tulisi sisällyttää järjestelmällisesti mittauksiin. Samalla havaitaan puutteita muun muassa tapahtumalähteiden vertailussa ja kokeiden toistettavuudessa, mikä vaikeuttaa tulosten yleistämistä yli alustojen [7]. Yhteenvetona FaaS skaalautuu hyvin samanaikaisiin kuormapiikkeihin ja korkeisiin rinnakkaisuuslukemiin, mutta käytännön skaalautuvuus määräytyy ylärajojen, kuormaprofiilin, tapahtumalähde- ja API-polkujen ja tarjoajakohtaisten toteutusten ehdoilla.

3.2 Latenssi ja kylmäkäynnistys

Latenssilla viittaan tässä päästä päähän -viiveeseen yksittäisen funktion tai työnkulun kutsusta vastauksen saamiseen. FaaS-ympäristöissä keskeinen latenssia selittävä tekijä on kylmäkäynnistys, joka syntyy, kun alusta joutuu alustamaan suoritusympäristön ennen funktion ajoa. Lämmin käynnistys taas ohittaa tämän vaiheen. Kylmäkäynnistystä pidetään kirjallisuudessa ehkä tutkituimpana serverless-ilmiönä, ja sen vaikutus korostuu harvakäyttöisissä tai purskeisissa kuormissa [10]. Lisäksi on raportoitu, että vaikka hiekkalaatikon valmistelu virtualisointikerroksessa voi kestää alle sekunnin, sovellusympäristön ja kirjastojen alustaminen voi käytännössä vaikuttaa enemmän kokonaisviiveeseen [11].

Kylmäkäynnistysviiveeseen vaikuttavat useat tekijät. Ensinnäkin ohjelmointikieli ja suoritusympäristö: tulkattujen kielten (esim. Python, JavaScript) kylmäkäynnistys on toistuvasti mitattu lyhyemmäksi kuin käännettyjen ajoalustojen (kuten Java/JVM, .NET), joissa raskas suoritusympäristön alustaminen kasvattaa viivettä.

Erot voivat olla huomattavia [10], mikä korostaa ohjelmointikielen ja suoritusympäristön valinnan merkitystä erityisesti kylmäkäynnistysherkissä työkuormissa. Toiseksi riippuvuuksien ja pakettien koko sekä alustuksessa suoritettava kirjastolataus pidentävät alustusvaihetta ja näkyvät suoraan ensimmäisen pyynnön vasteessa. Tätä on kuvattu sekä katsauksissa [7], [10] että mittauksissa. Esimerkiksi Puripunpinyo ja Samadzadeh [12] mittasivat Java-ajoympäristön julkaisuartefaktien optimoinnin vaikutuksia AWS Lambdassa [12]. Kolmanneksi tarjoajakohtaiset toteutukset ja instanssien kierrätyskäytännöt vaikuttavat siihen, kuinka usein kylmäkäynnistyksiä ilmenee: perääntymismittauksissa (engl. backoff), joissa samaa funktiota kutsutaan yhä pitenevin odotusvälein, kunnes lämmin instanssi on kierrätetty ja seuraava kutsu osuu kylmäkäynnistykseen, Azure Functions altistui kylmäkäynnistykselle jo noin 5 minuutin toimettomuuden jälkeen, kun taas AWS Lambda ja Google Cloud Functions eivät osoittaneet kylmäkäynnistyksen merkkejä vielä 30 minuutin tauonkaan jälkeen [8]. Eroja on raportoitu myös alustan resurssipolitiikasta ja instanssien sijoittelusta johtuen [10].

Latenssia tarkastellaan kirjallisuudessa sekä yksittäisen funktion että työnkulun tai sovellustasolla. Yksittäisen funktion kohdalla keskeisiä mittareita ovat p50 / p95 / p99 -viiveet, joissa kylmäkäynnistykset näkyvät erityisesti häntäviiveissä (engl. tail latency): p50 (mediaani) kuvaa tyypillistä vasteaikaa, kun taas p95 ja p99 ovat yläprosenttipisteitä (95. ja 99. persentiili), jotka kertovat häntäviiveistä: 95 % tai 99 % pyynnöistä on tätä nopeampia, joten ne paljastavat harvinaisemmat, mutta käyttäjäkokemuksen kannalta merkittävät hidastumiset. Sovellus- ja työnkulkutasoilla kokonaislatenssi kertyy useista peräkkäisistä kutsuista, tapahtumalähteistä ja ulkoisista palveluista (esim. tietovarastot, API-yhdyskäytävät), jolloin yksittäisen funktion vakiotilan nopeus ei yksin selitä käyttäjän kokemaa suorituskykyä. Tätä on mitattu yhdistämällä HTTP-pyynnön ja funktion keston tarkastelua sekä analysoimalla kestojakaumia [7], [8]. Samalla on tuotu esiin mittausasetelmien vertail-

tavuuteen liittyviä puutteita (esimerkiksi tapahtumalähteiden erot, toistettavuus), mikä vaikuttaa alustojen välisten tulosten keskenäiseen vertailukelpoisuuteen [7], [10].

Cold start -vaikutusten lieventämiseksi kirjallisuudessa esitetään sekä alustalähtöisiä että sovellus- ja operaatiolähtöisiä keinoja. Alustalähtöisissä lähestymistavoissa hyödynnetään muun muassa esilämmitysmenetelmiä, joissa instansseja pidetään valmiina etukäteen, sekä ennakkorakennustekniikoita, joissa instanssien alustuksen raskaat vaiheet suoritetaan jo ennen varsinaista tarvetta. Näiden lisäksi väliinstanssipoolit sekä ennustavat tai heuristiset resurssinvaraukset pienentävät kylmäkäynnistysten todennäköisyyttä tai kestoa. Katsauksissa kuvataan yksityiskohtaisesti muun muassa esilämmitys-arkkitehtuurit ja kevyet hiekkalaatikko-ratkaisut [1], [11]. Sovellustasolla tunnettuja käytäntöjä ovat esimerkiksi yhteyden ylläpitokuorma, eli niin sanottu keep-alive-kuorma, joilla instanssit pidetään lämpiminä, sekä pakettien pienentäminen ja riippuvuuksien minimointi, jotka lyhentävät alustusvaihetta [7]. Käytännön kompromissina esilämmitys- ja vastaavat kapasiteetin etukäteisvalmistelut parantavat latenssia, mutta lisäävät resurssien käyttöä ja kustannuksia. Siksi valinnat tulisi sovittaa kuormaprofiilin ja kustannus–suorituskyky -tavoitteiden mukaan [10] [11].

Yhteenvetona kylmäkäynnistys on FaaS-latenssin keskeinen selittäjä etenkin harvakäyttöisissä funktioissa ja ketjutetuissa työnkuluissa. Vaikutus riippuu kielen ja pakettien ominaisuuksista, alustakohtaisista käytännöistä sekä työnkulun arkkitehtuurista. Mittaustulokset tukevat johtopäätöstä, että sekä alustalliset esilämmityskeinot että sovellustason optimoinnit vähentävät merkittävästi häntäviiveitä, mutta optimaalinen ratkaisu on kuormasta sekä kustannus- ja suorituskykyvaatimuksista riippuvainen [8], [10], [11].

3.3 Muut haasteet

Tässä alaluvussa tarkastellaan lyhyesti muita FaaS-malliin liittyviä haasteita, joita ei ole käsitelty aiemmin kirjallisuuskatsauksessa. Yksi keskeisistä ongelmista on testattavuus ja havaittavuus. Koska serverless-funktiot ovat tyypillisesti tilattomia ja hajautettuja, niiden kokonaisjäljitettävyys sekä toistettavien mittausten suorittaminen ovat vaikeita. Kirjallisuudessa korostetaan päästä päähän-mittausten merkitystä ja suorituskyvyn arvioimista p95- ja p99-latenssien avulla, mutta toistettavuudessa ja standardoiduissa käytännöissä on edelleen merkittäviä puutteita [7], [10], [13].

Toinen keskeinen haaste liittyy palveluntarjoajariippuvuuteen eli niin sanottuun vendor lock-in -ilmiöön. Eri alustojen kieli- ja ajoympäristötuet, hallitut integraatiopalvelut sekä erikoislaitteiston, kuten GPU- tai FPGA-resurssien, rajallinen saatavuus vaikeuttavat sovellusten siirrettävyyttä ja optimointia. Nämä erot voivat sitoa kehittäjät vahvasti tietyn ekosysteemin ratkaisuihin ja rajoittaa monipilviarkkitehtuurien käyttöä [11], [14], [15].

Kustannusten hallinta muodostaa oman ongelmansa. FaaS hinnoittelu perustuu yleensä gigatavusekuntikohtaiseen laskentamalliin, jossa sekä muistikonfiguraatio että suoritusaika vaikuttavat kokonaiskustannuksiin. Tämän seurauksena optimointiprosessi on epälineaarinen ja kustannusten ennustaminen hankalaa. Myös laskutusgranulariteetti, kuten minimiveloituksen pituus, voi vääristää resurssien käytön arviointia [11], [16].

Tilattomuus puolestaan siirtää vastuun tilanhallinnasta ulkoisiin palveluihin, kuten tietovarastoihin tai välimuisteihin, mikä kasvattaa kokonaisviivettä ja lisää riippuvuutta muiden palvelujen vasteajoista. Funktioketjutus eli useiden funktioiden peräkkäinen kutsuminen lisää edelleen päästä päähän -latenssia ja tekee virheenkäsittelystä monimutkaisempaa [11], [13]. Lisäksi monilla alustoilla käytössä oleva *vähintään kerran* -toimitusmalli voi johtaa siihen, että sama funktio suoritetaan useammin kuin kerran. Tämä edellyttää idempotenttien käsittelijöiden suunnitte-

lua sekä huolellista virheenkäsittelyä järjestelmän konsistenssin säilyttämiseksi [13], [17].

Suorituskyvyn näkökulmasta merkittävä haaste ovat niin sanotut häntäviiveet. Vaikka keskimääräinen vasteaika pysyisi hyväksyttävänä, pitkät p95- ja p99-latenssit voivat heikentää käyttäjäkokemusta ja vaikeuttaa palvelutasosopimusten (engl. service level agreement, SLA) täyttämistä. Siksi käytännön ohjeistuksissa korostetaan erityisesti häntäviiveiden seurantaa ja optimointia [7], [17].

Suoritus- ja kokorajoitteet vaikuttavat myös merkittävästi arkkitehtuurivalintoihin. Funktioiden maksimiajoaika, asennuspakettien enimmäiskoko ja integraatiopalveluiden käyttörajat voivat estää raskaiden kirjastojen tai koneoppimismallien hyödyntämisen ilman erikoisratkaisuja [5], [15]. Automaattinen skaalaus tuo omat ongelmansa: kuormituspiikit voivat aiheuttaa alavirran pullonkauloja, kuten tietokantayhteyksien ylikuormittumista. Tämän ehkäisemiseksi suositellaan yhteyspoolauksen, välityspalvelinten ja kuristusmekanismien käyttöä [17]. Samanaikaisuuden katot ja kutsujen ylärajat on lisäksi otettava huomioon jo palvelutasotavoitteiden (engl. service level objective, SLO) määrittelyssä, sillä ne vaikuttavat suoraan järjestelmän päästä päähän -suorituskykyyn [9].

3.4 Käyttötapaukset

Kirjallisuuden mukaan FaaS:n eri käyttötapaukset paljastavat, miten hyvin se skaalautuu, kuinka suuri latenssi on, mitä se maksaa ja millaisia kompromisseja sen käyttöön liittyy käytännön tasolla. Tyypillistä on tapahtumavetoisuus, lyhytkestoiset suoritukset ja kuormien epäsäännöllisyys. Vastaavasti jatkuva, pitkäkestoinen tai erikoislaitteiston varassa oleva työ on usein heikosti soveltuva FaaS-malliin [14]. Seuraavaksi kootaan toistuvia käyttötapauksia ja niihin liitettyjä havaintoja.

API-pohjaiset backendit ja kevyt mikropalveluarkkitehtuuri ovat luontevia FaaS-sovelluksia. FaaS-funktio altistetaan päätepisteenä API-yhdyskäytävän kautta. Alus-

toissa on tuki synkronisille ja asynkronisille kutsuille sekä HTTP ja gRPC-protokollille, mukautetuille domaineille ja TLS:lle. Nämä ominaisuudet tekevät mallista hyvän valinnan vaihtelevalle liikenteelle ja hienojakoiseen pilkkomiseen [15].

Ajastetut taustatehtävät ovat toinen tyypillinen käyttökohde. Cron-tyyppiset ajastimet ovat tuettuja tapahtumalähteitä, joilla käynnistetään säännöllisiä siivous-, raportointi- tai eräprosesseja. Kyky skaalautua nollaan, eli siirtyä tilaan jossa yhtään instanssia ei pidetä käynnissä kuorman puuttuessa, pitää kustannukset alhaisina hiljaisina aikoina, mutta kapasiteetti on käytettävissä purskeisiin [15].

Viestijonot ja tietovirta-alustat integroituna funktioihin muodostavat keskeisen integraatiomekanismin. Viestijonot ja hajautetut tietovirta-alustat (esimerkiksi SQS ja Kafka) tukevat löyhää kytkentää ja kuormapiikkien tasaamista. Tapahtumavälitteinen yhdistäminen on toistuva kaava serverless-komponenttien integroinnissa [15].

Data- ja ETL-putket hyödyntävät FaaS-mallia, kun objektivarastojen ja tietokantojen tapahtumat käynnistävät muunnoksia, validointeja ja rikastuksia. On esitetty nimenomaisia serverless-dataputkiarkkitehtuureja, joissa funktiot muodostavat ketjutettuja käsittelyvaiheita [13]. Käytännön valinnassa tapahtumalähteiden ja tallennuspalvelujen tuki FaaS-alustassa on keskeinen [15].

Tieteelliset työnkulut soveltuvat FaaS:lle silloin, kun vaiheet ovat lyhytkestoisia ja rinnakkaistettavissa, esimerkiksi parametripyyhkäisyissä ja simulaatioiden osatehtävissä. Esimerkiksi HyperFlow AWS- ja GCP-ympäristöissä hyödyntää elastista skaalautuvuutta ilman klusterin manuaalista ylläpitoa [13].

IoT ja reunalaskentaskenaariot hyötyvät FaaS:sta usealla tavalla. reunalaskennalla on kolme toistuvaa hyötyä: päästä päähän -viiveen pieneneminen, runkoverkon kuormituksen keventäminen ja yksityisyyden parempi hallinta, kun raakadataa ei siirretä keskuspilveen. Skenaarioissa on tyypillisesti harvakäyttöisiä (engl. low-frequency) tapahtumalähteitä ja liikenteen purskeita, jolloin automaattinen skaalaus ja käytönmukainen laskutus ovat eduksi. Reunaympäristöissä samalla korostuvat re-

surssirajoitteet ja tilan ulkoistamisen haasteet, mikä ohjaa kevyisiin funktioihin ja erilliseen tilanhallintaan [18].

Käytännön valintaa tukevat havainnot kytkeytyvät edellä käsiteltyihin suorituskykyteemoihin. API- ja ajastinkäyttötapauksissa kylmäkäynnistyksen vaikutus näkyy erityisesti häntäviiveissä ja harvakäyttöisissä funktioissa, jolloin esimerkiksi esilämmitys-strategioiden tai pakettikoon optimoinnin tarve nousee esiin. Viestijono- ja stream-pohjaisissa integraatioissa skaalautuvuus ja takaisinkuittausmekanismit auttavat tasaamaan purskeita, mutta päästä päähän -viive määräytyy ketjun heikoimman lenkin ja ulkoisten palvelujen mukaan. Reunaympäristöissä latenssihyödyt voivat olla merkittäviä, mutta rajalliset resurssit ja tilan ulkoistus voivat kumuloida viiveitä ilman erillisiä ratkaisuja [18].

Yhteenvetona FaaS on luontevimmillaan tapahtumavetoisissa, lyhytkestoisissa ja purskeisissa tehtävissä, joissa “scale-to-zero” ja automaattinen skaalaus realisoivat kustannus–suorituskyky -hyödyt. Sen sijaan pitkäkestoiset, IO-sitkeät tai erikoislaitteiston (esim. GPU/FPGA) edellyttämät työt eivät tyypillisesti ole kustannustehokkaita tai teknisesti hyvin tuettuja FaaS-mallissa [14]. Toteutusvalintoja ohjaavat viime kädessä alustakohtainen tapahtumalähde- ja orkestrointituki sekä reunaympäristön erityispiirteet, kuten resurssirajoitteet, tila ja verkko [15], [18].

4 Yhteenveto

Tässä kandidaatintutkielmassa on tarkasteltu FaaS -pilvipalvelumallia osana serverless-arkkitehtuuria kirjallisuuskatsauksen menetelmin. Tavoitteena on ollut muodostaa kokonaiskuva siitä, miten FaaS vaikuttaa sovellusten skaalautuvuuteen ja suorituskykyyn, millaisia haasteita malliin liittyy sekä mitkä ovat tärkeimmät kehityssuunnat ja tulevaisuuden mahdollisuudet.

Tutkimuskysymyksessä TK1 kysyttiin ”Miten FaaS vaikuttaa sovellusten skaalautuvuuteen ja suorituskykyyn?”. Katsauksessa havaittiin, että FaaS-alustat skaalautuvat hyvin samanaikaisiin kuormapiikkeihin lisäämällä rinnakkaisia funktioinstansseja kuorman kasvaessa. Läpimeno kasvaa tietyssä rajassa lähes lineaarisesti samanaikaisuuden mukana, kunnes alustakohtaiset samanaikaisuus- ja resurssirajat saavutetaan. Suorituskyvyn kannalta todettiin, että FaaS soveltuu erityisesti tapahtumavetoisiin, lyhytkestoisiin ja epäsäännöllisiin työkuormiin, joissa automaattinen skaalaus ja ”scale-to-zero” -ominaisuus mahdollistavat korkean läpimenon ja kustannustehokkuuden. Kokonaisvasteaikaan vaikuttavat kuitenkin myös työnkulkuun kuuluvat ulkoiset palvelut, kuten tietovarastot ja API-yhdyskäytävät.

Tutkimuskysymyksessä TK2 kysyttiin ”Mitä haasteita FaaS-teknologiaan liittyy?”. Katsauksessa ilmeni, että keskeisimpiä haasteita ovat kylmäkäynnistyksen aiheuttama lisäviive, hajautettujen ja tilattomien työnkulkujen havaittavuus ja testattavuus, palveluntarjoajariippuvuus, kustannusten ennustettavuus sekä tilanhallinnan ja funktioketjutuksen monimutkaistuminen. Kylmäkäynnistyksen todettiin pi-

dentävän erityisesti häntäviiveitä harvakäyttöisissä funktioissa ja ketjutetuissa työnkuluissa, kun taas tilattomuus siirtää tilan hallinnan ulkoisiin palveluihin ja lisää näiden vaikutusta kokonaislatenssiin. Lisäksi havaittiin, että suoritus- ja kokorajoitteet sekä erikoislaitteiston rajallinen tuki rajoittavat FaaS:n käyttöä pitkäkestoisissa, IO-sidonnaisissa ja raskasta laskentaa vaativissa tehtävissä.

Tutkimuskysymyksessä TK3 kysyttiin ”Mitkä ovat FaaS-teknologian kehityksen tärkeimmät suuntaukset ja tulevaisuuden mahdollisuudet?”. Katsauksen perusteella todettiin, että merkittäviä kehityssuuntia ovat kylmäkäynnistyksen lieventäminen kevyempien suoritusympäristöjen, esilämmitys- ja ennustavan skaalaustekniikan avulla, havaittavuuden ja suorituskykymittauksen parantaminen sekä orkestrointi- ja tilaratkaisujen kehittäminen monimutkaisempia työnkulkuja varten. Lisäksi reuna- ja IoT-skenaarioiden yhteydessä tunnistettiin mahdollisuuksia latenssin pienentämiseen ja datansiirron vähentämiseen siirtämällä FaaS-suoritusta lähemmäs datan lähdettä. Monipilvi- ja palveluntarjoajariippuvuutta vähentävien lähestymistapojen sekä tilallisten funktioiden nähdään laajentavan FaaS-mallin soveltamista entistä vaativampiin käyttötapauksiin.

Yhteenvetona voidaan todeta, että FaaS tarjoaa selkeitä etuja erityisesti tapahtumavetoisissa, lyhytkestoisissa ja epäsäännöllisesti kuormittuvissa sovelluksissa, joissa automaattinen skaalaus ja kulutusperusteinen hinnoittelu tuottavat hyvän kustannus–suorituskyky-suhteen. Vastineeksi on hallittava kylmäkäynnistyksen, tilattomuuden, havaittavuuden ja alustariippuvuuden tuomat rajoitteet sekä sovitettava arkkitehtuuriset valinnat alustakohtaisiin rajoituksiin. Jatkokehityksen onnistuessa FaaS-mallin odotetaan laajenevan yhä useammille sovellusalueille, erityisesti reunaympäristöihin ja vaativampiin, orkestroinnin ja tilan hallintaa edellyttäviin työnkulkuihin.

Lähdeluettelo

[1] Y. Li, Y. Lin, Y. Wang, K. Ye ja C. Xu, ”Serverless Computing: State-of-the-Art, Challenges and Opportunities”, *IEEE Transactions on Services Computing*, vol. 16, nro 2, s. 1522–1539, 2023. DOI: 10.1109/TSC.2022.3166553.

[2] Red Hat. ”What is IaaS?”, viitattu 10. marraskuuta 2025. url: https://www.redhat.com/en/topics/cloud-computing/what-is-iaas.

[3] Red Hat. ”What is PaaS?”, viitattu 10. marraskuuta 2025. url: https://www.redhat.com/en/topics/cloud-computing/what-is-paas.

[4] Red Hat. ”What is FaaS?”, viitattu 10. marraskuuta 2025. url: https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas.

[5] Amazon Web Services. ”Lambda runtime environment”, viitattu 28. toukokuuta 2025. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html.

[6] P. Vahidinia, B. Farahani ja F. S. Aliee, ”Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach”, *IEEE Internet of Things Journal*, vol. 10, nro 5, s. 3917–3927, maaliskuu 2023.

[7] J. Scheuner ja P. Leitner, ”Function-as-a-service performance evaluation: A multivocal literature review”, *Journal of Systems and Software*, vol. 170, s. 110 708, 2020.

[8] B. Hamiti ja B. Selimi, "Function-as-a-Service Performance Evaluation with Application-Level Workflows", Master's thesis, South East European University, North Macedonia, 2023. url: https://repository.seeu.edu.mk/sites/thesis/ThesisSharedDocs/MA_129568.pdf.

[9] K. L. Ngo, J. Mukherjee, Z. M. Jiang ja M. Litoiu, "Evaluating the scalability and elasticity of function as a service platform", teoksessa *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, s. 117–124.

[10] A. Raza, I. Matta, N. Akhtar, V. Kalavri ja V. Isahagian, "Sok: Function-as-a-service: From an application developer's perspective", *Journal of Systems Research*, vol. 1, nro 1, 2021.

[11] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He ja M. Guo, "The Serverless Computing Survey: A Technical Primer for Design Architecture", *ACM Comput. Surv.*, vol. 54, nro 10s, syyskuu 2022, ISSN: 0360-0300. DOI: 10.1145/3508360.

[12] H. Puripunpinyo ja M. H. Samadzadeh, "Effect of Optimizing Java Deployment Artifacts on AWS Lambda", teoksessa *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2017, s. 438–443. DOI: 10.1109/INFCOMW.2017.8116416.

[13] H. B. Hassan, S. A. Barakat ja Q. I. Sarhan, "Survey on serverless computing", *Journal of Cloud Computing*, vol. 10, nro 39, 2021. DOI: 10.1186/s13677-021-00253-7.

[14] B. Pijnacker ja J. van der Zwaag, "Opportunities and Challenges in the Adoption of Function-as-a-Service Serverless Computing", *20th SC@ RUG 2022-2023*, s. 65,

[15] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi ja F. Leymann, "FaaSten your decisions: A classification framework and technology review of function-

as-a-Service platforms", *Journal of Systems and Software*, vol. 175, s. 110 906, 2021, ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.110906.

[16] A. Jindal, M. Chadha, S. Benedict ja M. Gerndt, "Estimating the capacities of function-as-a-service functions", teoksessa *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, sarja UCC '21, Leicester, United Kingdom: Association for Computing Machinery, 2022, ISBN: 9781450391634. DOI: 10.1145/3492323.3495628.

[17] Amazon Web Services. "AWS Well-Architected Framework: Serverless Computings Lens", viitattu 28. toukokuuta 2025. url: https://docs.aws.amazon.com/pdfs/wellarchitected/latest/serverless-applications-lens/wellarchitected-serverless-applications-lens.pdf.

[18] J. Ding, C. Jiang, J. Liu, S. Tang, Y. Chen ja Y. Chen, "Function as a Service for Edge Computing: Challenges and Solutions", teoksessa *2024 4th International Conference on Communication Technology and Information Technology (ICCTIT)*, 2024, s. 623–630. DOI: 10.1109/ICCTIT64404.2024.10928463.