Palvelimettoman tietojenkäsittelyn yleisimmät ongelmat ja ratkaisuja niihin

Antti Salonen

Haaga-Helia ammattikorkeakoulu
Tradenomi
Tutkimusraportti
2022

Tiivistelmä

Tekijä(t)
Antti Salonen

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Palvelimettoman tietojenkäsittelyn yleisimmät ongelmat ja ratkaisuja niihin

Sivu- ja liitesivumäärä
34 + 17

Tämä työ on opinnäytetyö, joka on tutkimus palvelimettomien tietojenkäsittelyn yleisimmistä ongelmista ja niihin tarjotuista ratkaisuista. Palvelimeton tietojenkäsittely on uusi teknologia, jossa kehittäjät voivat keskittyä vain itse sovelluksen kehittämiseen. Teknologiaa onkin kevyttä kehittää koska palveluntarjoaja hoitaa kaiken palvelimen puolella. Palvelimettoman tietojenkäsittelyn eduissa on kääntöpuolensakin, ja teknologia on vielä kypsymätöntä. Ongelmia ovat latenssi, joka voi välillä olla isoakin, bugien korjaamisen, testauksen ja monitoroinnin vaikeus ja puutteellisuus, sekä tietoturvaongelmat.

Tietoperustassa käsiteltiin neljää yleisintä ongelmaa: cold start, funktioiden tiladatattomuus, bugien korjaus, testaus ja monitorointi, sekä tietoturva, ja näihin tarjottuja ratkaisuja. Tietoperustan lähteenä oli käytetty tutkimuksia palvelimettomasta tietojenkäsittelystä.

Työ tehtiin tutkimustyyppisenä, ja ajankohtana oli syyslukukausi 2022. Työn tavoitteena oli tutkia mitkä ovat palvelimettoman tietojenkäsittelyn yleisimmät ongelmat, minkälaisia ongelmat ovat, ja mitä ratkaisuja niihin on tarjolla. Työn rajaus oli yleisimmissä ongelmissa ja niihin tarjotuissa ratkaisuissa. Tutkimusmenetelminä käytettiin kvantitatiivista -ja kvalitatiivista menetelmää. Kvantitatiivisella menetelmällä tutkittiin mitkä olivat yleisimmät ongelmat, ja kvalitatiivisella menetelmällä tutkittiin, minkälaisia nämä yleisimmät ongelmat olivat ja minkälaisia ratkaisuja niihin oli tarjolla.

Yleisimmät ongelmat ovat olleet cold start, funktioiden tiladatattomuus, bugien korjaus, testaus ja monitorointi, sekä tietoturva. Ongelmat ovat johtuneet mm. konttien kylmettymisestä, ulkoisten datavarastojen käytöstä, kehittäjiltä suljetusta palvelinpuolesta ja tietoturvan kehittämättömyydestä. Ratkaisemalla näitä yleisimpiä ongelmia, on voitu saada paljon aikaiseksi. Yhden ongelman tai puutteen korjaaminen on voinut myös ratkaista ongelmia enemmän kuin yhdessä osa-alueessa.

Asiasanat
Cold start, tiladata, funktio.

Sisällys

1 Johdanto

Tämä tutkimus on opinnäytetyö, ja sen aiheena on tutkia palvelimettoman tietojenkäsittelyn yleisimmät ongelmat ja niihin tarjottuja ratkaisuja. Tutkimus pyrkii tuomaan tiedon muodossa suoraa käytännöntason hyötyä palvelimettoman teknologian kehittäjille. Palvelimeton tietojenkäsittely on vielä uusi teknologia, joten kehittymisen varaa löytyy vielä paljon. Tämä teknologia tarjoaa paljon web-kehittämistä ja niiden ylläpitämistä helpottavia ja keventäviä etuja, mutta on myös paljon ongelmia, jotka vaativat ratkaisuja.

Tarvitsin opinnäytetyölleni jonkin aiheen, ja törmäsin palvelimettomaan tietojenkäsittelyyn, joka vaikutti kiinnostavalta. Tarkemmaksi aiheeksi valitsin ongelmat ja ratkaisut, koska halusin käytännöllisen näkökulman asiaan, josta voisi olla mahdollisimman paljon suoraa hyötyä kehittäjille.

Tutkimuskysymyksinä ovat: Mitkä ovat yleisimmät ongelmat palvelimettomassa tietojenkäsittelyssä? Minkälaisia nämä ongelmat ovat? Mitä ratkaisuja ongelmiin on esitetty? Tavoitteena on löytää vastaus näihin kysymyksiin, sekä tehdä johtopäätelmiä ja pohdintaa hankitusta tiedosta. Myös ongelmiin tarjottujen ratkaisujen tutkiminen on olennainen tavoite, jotta tutkimuksen suora käytännön hyöty toteutuisi. Työ on rajattu palvelimettoman tietojenkäsittelyn neljään yleisimpään ongelmaan ja niihin tarjottuihin ratkaisuihin, vaikkakin tietoperustassa tullaan myös käymään läpi pohjatiedoksi mitä palvelimeton tietojenkäsittely on.

Tutkimusmenetelmä on kvantitatiivinen ja kvalitatiivinen. Tutkimuksessa käydään läpi muita valmiita tutkimuksia palvelimettomasta tietojenkäsittelystä, ja katsotaan mistä ongelmista niistä kirjoitetaan. Määrällinen tutkimus toteutetaan laskemalla eri ongelmista kirjoitetut määrät, joiden perusteella voidaan arvioida mitkä ongelmat ovat yleisimpiä. Laadullinen tutkimus toteutetaan tutkimalla minkälaisia ongelmat ja niihin tarjotut ratkaisut ovat.

Palvelimeton tietojenkäsittely on nopeasti kasvava teknologia, jossa kehittäjät voivat keskittyä vain sovelluksen varsinaiseen bisneslogiikkaan. Teknologia on kevyttä kehittää, koska palveluntarjoaja hoitaa kaiken palvelinpuolella. Sovellukset muodostuvat palveluun ladattavasti yksittäisistä funktioista, joista muodostetaan workflow't. Nimi ”palvelimeton” ei viittaa siihen, etteikö palvelinta olisi, vaan tarkoittaa sitä, että palveluntarjoaja hoitaa palvelinpuolella kaiken. Sovelluksia onkin tästä syystä kevyttä kehittää ja ylläpitää tälle teknologialle. Palvelimettomat alustat ovat itsestään skaalautuvia, jolloin sovelluskehittäjien ei tarvitse huolehtia palvelimen resurssien jaosta. Hinnoittelu perustuu pelkästään funktioiden käyttöaikaan, eli silloin kun sovelluksen jotain toimintoa ei käytetä, ei siitä myös laskutetakaan.

Kehittämisen ja ylläpitämisen keveydellä on tosin hintansakin. Monia sovelluksia ei voi tällä teknologialla suoraan toteuttaa, jolloin ne täytyy tehdä kiertoteitse. Tämä tuottaa latenssia, ja voi tehdä sovelluksen rakenteesta turhan monimutkaisen. Palvelimettomassa tietojenkäsittelyssä voi myös olla suurtakin latenssia ”cold startin” muodossa, joka liittyy palveluntarjoajien tapaan laskuttaa käyttäjiä. Teknologian ollessa vielä uusi, ei ohjelmistokehityksen perinteisiä vaiheita, bugien korjausta, testausta ja monitorointia voi perinteisillä työkaluilla tehdä. Tietoturvaa ei myöskään ole tarpeeksi kehitetty palvelimettomalle teknologialle. Samat ongelmat, jotka vaivaavat kehittämistä, latenssia ja bugien korjaamista, ovat ongelmana myös tietoturvalle. Koska palvelimeton tietojenkäsittely on vielä niin uutta, on vielä paljon uusia ongelmia, eikä kaikkia hyväksi todettuja kehittämistapoja voi käyttää.

Olennaisia tutkimuksen käsitteitä:

- **Cold start**. Funktion käynnistämiseen liittyvä latenssi, kun funktio käynnistetään sen ollessa jonkin aikaa käyttämättä.
- **Warm start**. Jos funktio on jo ollut vähän aikaa sitten käytössä, ei siihen tule käynnistyslatenssia.
- **Tiladata**. Välillistä dataa, jota käytetään määrittämään sovelluksen tila.
- **Tiladataton funktio**. Funktio, jossa ei ole tiladataa. Funktiolla on vain yksi tila, ja se toimii aina samanlailla. Funktiosta palautettu data riippuu siitä mitä funktiolle syötetään sisään.
- **Tiladatallinen funktio**. Tiladatallinen funktio, jolla voi olla eri tiloja. Funktion toiminta riippuu siitä missä tilassa se on.
- **Ulkoinen datavarasto**. Tietokanta, joka on erillinen pilvipalvelu käytetystä palvelimettomasta alustasta. Ulkoisia datavarastoja käytetään tiladatan tallentamiseen.
- **Workflow**. Funktioista koostuva ohjelma, jolla suoritetaan monimutkaisempia tehtäviä.

2 Tietoperusta

2.1 Mitä palvelimeton tietojenkäsittely on?

Palvelimettomassa tietojenkäsittelyssä kehittäjien täytyy toteuttaa vain itse sovellus. Se toteutetaan funktioina, jotka ovat erillisiä, tapahtumakäyttöisiä, tiladatattomia yksiköitä ja jotka hoitavat tietyn tehtävän. Palvelimeton alusta hoitaa resurssien hallinnoinnin, jolloin kehittäjät voivat keskittyä vain sovelluksen bisneslogiikkaan, jolloin palvelinta tai virtuaalikoneita ei tarvitse hoitaa. Sovelluksen käyttämät funktiot ja niiden käyttämät kirjastot ladataan palvelimettomalle alustalle (Wen ym. 2022). Kehittäjien ei tarvitse huolehtia suoritusinfrastruktuurin tai alustan skaalaamisesta, koska palvelimeton tietojenkäsittely on automaattisesti skaalautuvaa (Shafiei, Khonsari & Mousavi 18.2.2022). Palvelimetonta tietojenkäsittelyä kutsutaan myös termillä FaaS (Function-as-a-Service), koska siinä ei vuokrata infrastruktuuria kuten perinteisissä pilvipalveluissa, vaan kehittäjät vain lähettävät funktiokoodin pilvipalveluntarjoajalle (Kuhlenkamp, Werner, Tai 2020).

Kun funktio laukaistaan käyttöön, suoritetaan sen toiminnallisuus funktioinstanssilla, jonka palvelimeton alusta käynnistää. Funktioinstanssina toimii kontti tai jokin muu hiekkalaatikko. Jos funktiota ei käytetä jonkin aikaa, vapauttaa palvelimeton alusta resurssit pois käytöstä. Näin resurssienhallinta on kevyttä ja tehokasta (Wen ym. 2022). Palvelimettomassa tietojenkäsittelyssä funktion käytöstä maksetaan vain silloin kun sitä suoritetaan. Ajasta, jolloin funktiota ei suoriteta, ei myös maksetakaan (Vahidinia, Farahani & Shams Aliee 2020).

2.2 Cold start-ongelma

2.2.1 Mitä cold start on?

Palvelimeton tietojenkäsittely perustuu siihen, ettei resursseja pidetä koko ajan valmiina pyörimässä, jolloin käyttäjät pääsevät maksamaan vain käytetyistä resursseista. Kääntöpuolena tästä syntyy ”cold start”, jossa syntyy huomattavaa latenssia funktion käynnistyessä ensimmäisen kerran ”kylmiltään”. Shahradin, Balkindin & Wentzlaffin tutkimuksessa kuvataan cold starttia seuraavanlaisesti: ennen funktion käynnistystä täytyy suorittaa alustaminen. Ensin valmistellaan hiekkalaatikko, joka on kevyt ja eristetty ympäristö funktiota varten. Jos hiekkalaatikko täytyy käynnistää ennen alustusta, on kyseessä cold start, ja jo pyörivän hiekkalaatikon alustamisessa on kyse warm startista. Funktion suorittamiseen tarvitaan ladata sen koodi (mm. käyttäjän koodi ja runtimen kirjastot) muistiin. Warm startissa koodi on jo ladattuna muistiin, jolloin funktio voidaan käynnistää nopeasti, eikä sitä tarvitse tuoda datavarastosta. Cold startissa instanssien täytyy aloittaa aivan alusta, johon kuuluu ohjelmistoympäristön alustaminen, ja sovelluskohtaisen koodin lataaminen. Jotta palvelimettoman tietojenkäsittelyn idea resurssien käyttämisestä vain tarvittaessa toteutuisi, poistetaan

funktioiden kontit, jos funktiota ei ole kutsuttu jonkin aikaa. Cold start tapahtuukin, kun funktio kutsutaan uudestaan kontin poistamisen jälkeen. (Li, Lin, Wang, Ye & Xu 2022, 5; Li, Guo, Cheng, Chen, He & Guo 2022, 2; Xu, Zhang, Geng, Wu & Ma 2019, 9; Shahrad ym.2020.)

Palvelimettoman tietojenkäsittelyn ideana on tehdä palvelun kehittämisestä ja ylläpitämisestä kevyttä, antamalla kehittäjien keskittyä vain bisneslogiikkaan. Tämän teknologian keveyden kannalta cold startin luoma käynnistyslatenssi on merkittävä ongelma, koska funktioiden suoritusajat ovat usein lyhyitä, ja pitkä käynnistyslatenssi voi johtaa suhteellisen korkeisiin kustannuksiin. Etenkin jos funktiota kutsutaan epäsäännöllisesti, on cold start paha ongelma, koska silloin sen luoma latenssi täytyy käydä joka kutsukerta läpi. (Li, Lin, Wang, Ye & Xu 2022, 5; Xu, Zhang, Geng, Wu & Ma 2019, 9.)

2.2.2 Yleisiä asioita

Cold start on suurin latenssin aiheuttaja palvelimettomassa tietojenkäsittelyssä, ja sen pituus on moninkertainen verrattuna varsinaisen funktion suorituksen pituuteen. Cold startilla on kaksi vaihetta: latausvaihe ja valmisteluvaihe, jotka jakautuvat tasaisesti. Keskimäärin latausvaihe on 46,24 % -, ja valmisteluvaihe 42,46 % kokonaisvastauslatenssista. AWS Lambda on Amazonin palvelimeton alusta. Sillä toteutetun tutkimuksen mukaan cold start käsittää 88,7 % kokonaisvastauslatenssista, kun taas funktioiden latenssi on keskimäärin vain 7,5 %. Asioita, jotka vaikuttavat cold startin pituuteen, ovat ympäristön varaus, käynnistysprosessi, ohjelmointikieli, varattu muistin koko ja koodin koko. (Wen ym. 2022; Lee, Yoon, Yeo & Oh 2021, 2-3. Amazon)

Eri palveluntarjoajilla funktioiden kontit ”kylmettyvät” eri aikojen päästä. AWS Lambdassa cold start tapahtuu 5-7 minuutin päästä edellisestä funktiokutsusta, kun taas Azure Functionissa se tapahtuu pääasiallisesti 20-30 minuutin päästä. Azuressa cold start voi myös joskus tapahtua ennen 10:tä minuuttia, tai 50:nen minuutin jälkeen funktion kutsusta. (Mikhail Shilkov 2021; Mikhael Shilkov 2021)

Hitautta funktioiden suorittamiseen tuo myös datan hakeminen tietokannasta (lisää luvussa ”stateless”). Ustigovin, Amariucain & Grotin (2021, 51-60) tutkimuksessa selvisi, että datan hakeminen tietokannasta ja funktioiden kutsumispurkaukset ovat suurimpia tekijöitä latenssin syntymisessä. Tutkimus toteutettiin tiimin kehittämällä vapaan lähdekoodin ohjelmalla STeLLAR, jolla analysoidaan palvelimettomien järjestelmien suorituskykyä. Datan hakemiseen tietokannasta kuuluu funktioiden image:ien hakeminen funktioiden käynnistyksessä, sekä funktioiden välinen kommunikointi, koska sekin tapahtuu tietokannan välityksellä.

Kaiken kaikkiaan cold start aiheuttaa huomattavia ongelmia, eikä palvelimeton tietojenkäsittely kaikille sovelluksille edes sovi. Palvelimeton tietojenkäsittely ei sovi sovelluksille, joilla on tiukat latenssivaatimukset. Rahan tuoton puolesta kylmät kontit ovat ongelma palveluntarjoajille, koska niiden käytöstä ei laskuteta käynnistyksen aikana (Marin, Perino & Di Pietro 2022, 11).

2.2.3 Cold startin vaikutus funktioden suoritukseen

Palvelimettomassa tietojenkäsittelyssä funktiot ovat lyhyitä suoritusajaltaan, jonka takia pitkä cold start-latenssi on erityisen haitallinen. Dun ym. ja Shahradin ym. tutkimuksissa todetaan kaiken ylimääräisen rasitteen aiheuttavan suorituksille suuremman lisärasitteen, koska palvelimettomien funktioiden suoritusaika on niin lyhyt. IoT:lle cold start on erityisen haitallinen. Shahradin ym. samassa tutkimuksessa todetaan, että Microsoftin Azuressa enemmän kuin puolet IoT-palveluiden funktioista suoriutuu keskimäärin alle sekunnissa, joten cold startin latenssi on IoT:lle haitallinen. (Pan, Wang, Chen & Liu 2022; Shin, Kim, & Min 2022, 664.) Palvelimettoman tietojenkäsittelyn suoritusnopeutta voidaan myös mitata funktioiden suoritus- ja kokonaislatenssin välisellä suhteella. Dun, Yun, Xian, Zangin, Yanin, Qinin, Wun ja Chenin (2020, 468) tutkimuksessa testattiin 14 palvelimetonta funktiota Google gVisorille, joille laskettiin suoritus- ja kokonaislatenssin välinen suhde. Latenssi jaettiin kahteen osaan, suoritusosaan ja käynnistysosaan. Funktioista 12 funktiota, eli valtaosa, ei yltänyt edes 30 %:iin, ja loput ei yltänyt edes 65,54 %:iin. Tämä kertoo siitä, että käynnistys kattaa suurimman osan latenssista.

2.2.4 Cold start ja workflow

Leen, Yoonin, Yeon & Oh:n (2021, 2-3) tutkimuksen mukaan palvelimettomassa workflow’ssa cold start-lantenssi kasaantuu. Tämä johtuu siitä, että workflow’ssa funktiot suoritetaan peräkkäin sarjana, jolloin cold start tapahtuu kaikille funktioille vuoron perään. Cold start-latenssin kasaantumista voi helpottaa fuusioimalla workflow’n funktiot yhdeksi funktioksi. Se täytyy tosin toteuttaa huolellisesti, jotta latenssista ei tule vieläkään isompaa. Tutkimuksessa esitettiin esimerkki, jossa fuusioinnissa latenssi piteni eikä lyhentynyt. Esimerkissä fuusioitiin workflow, jossa suoritettiin keskellä funktioiden sarjaa yksi rinnakkaissuoritus. Fuusiossa rinnakkain suoritettavat funktiot pistettiin peräkkäin sarjaan, jolloin ne ei päässyt suorittamaan cold starttia samanaikaisesti. Tästä seurasi, että jälkimmäisen funktion cold start pystyttiin suorittamaan vasta, kun edellisen funktion cold start saatiin suoritettua, jolloin fuusioidun funktion kokonaislatenssi oli korkeampi kuin alkuperäisellä workflow’lla.

2.2.5 Ohjelmointikielien vaikutus cold starttiin

Ohjelmointikielen valinnalla on merkitystä käynnistysajan latenssiin. Suurin osa palvelimettomista funktioista on kirjoitettu tulkkauskielillä (kuten Node.js ja Python), jotka vaikuttavat käynnistysaikaan. Vaikutus tulee kielien runtimen ja itse sovelluksen latausajasta. Tutkimuksessa (Carreira, Kohli, Bruno & Fonseca 2021, 58–64) todetaan myös suoritusajan sakon pidentävän käynnistysaikoja, varsinkin tulkkauskielissä (Shin, Kim, & Min 2022, 664). Toinen ohjelmointikieliin liittyvä ongelma on JIT-kääntö. Modernit ohjelmointikieli-runtimet, kuten Node.js, Python ja Java, hidastaa funktioiden lyhyttä suoritusaikaa, koska runtimen koodin profiili on päättämässä JIT-käännöstä. JIT-käännös on myös ongelmallinen palvelimettomien hiekkalaatikoiden prosessoreiden suhteen, koska prosessoreita on hiekkalaatikoissa vain yksi. Ongelmallisuus johtuu siitä, että JIT-käännön täytyy kilpailla sovelluksen varsinaisen suorituksen kanssa prosessorin käytöstä. (Shin, Kim, & Min 2022, 664.)

2.2.6 Cold startin korjaamiseen liittyvät haasteet

Kaupallisten palvelimettomien alustojen konfiguraatioon ei käyttäjät voi vaikuttaa, jolloin koodin muuttaminen on tehokkain tapa vaikuttaa cold starttiin. Valmisteluvaiheenkaan latenssiin ei sovelluskehittäjillä oli juuri vaikutusmahdollisuutta, koska palvelimettomat alustat ovat suljettuja. Ohjelmointikielillä on eroa cold startin pituuteen, mutta jälkeenpäin kielen muuttaminen on vaikeaa. Koodin koko vaikuttaa cold startin pituuteen, ja yllättäen isompi koko vaikuttaa lyhentävästi, vaikkakin koon vaikutus on tosin merkityksettömän pieni. Myös muistin määrän lisäämisellä voi lyhentää cold starttia, mutta muisti tosin maksaa. (Lee, Yoon, Yeo & Oh 2021, 2-3; Wen ym. 2022.) Cold startin korjaaminen voi myös johtaa tarpeettomaan monimutkaisuuteen ja lisäresurssien käyttöön. Mannerin, Endreß:n, Heckelin & Wirtzin, Oakes ym. ja Puripunpinyon & Samadzadehin tutkimuksissa on yritetty helpottaa cold starttia. Suurin osa näistä lähestymistavoista vaatii kuitenkin lisäohjelmistoja ja -arkkitehtuuria, joka puolestaan vaatii lisää laskennallisia tehoja ja ylläpitoa (Dantas, Khazaei & Litoiu 2022). Tämä on vastoin palvelimettoman teknologian ideaa yksinkertaistaa verkkopalveluiden kehittämistä ja ylläpitoa.

Koodin suoritusta voidaan parantaa optimoimalla sitä. Carrieran, Kohlin, Brunon & Fonsecan (2021, 59) mukaan koodin optimointi on kuitenkin ongelma palvelimettomissa alustoissa. Modernit ohjelmointikielet keräävät statistiikkaa saman koodin suorittamisesta, jolloin ajan mittaan statistiikalla voidaan generoida optimisoitua koodia. Palvelimettomissa alustoissa ei taas koodista voi tehokkaasti kerätä statistiikkaa, koska ne on räätälöity suorittamaan funktioita satunnaisesti ja pisimmillään muutaman sekunnin mittaisina. Suurin osa palvelimettomien funktioiden suoritusajoista on myös niin lyhyitä, että niiden koodia on mahdotonta optimoida ja profiloida. Ja vaikka funktion

suoritus aika olisikin tarpeeksi pitkä, ei suorituksen jälkeen profilointi- ja käännösinformaatiota yleensä säästetä, jolloin sitä ei voi käyttää saman funktion uudelleen suorittamiseen.

2.2.7 Kontin lämpimänä pitämiseen liittyvät haasteet

Konttia voidaan myös pitää lämpimänä, mutta siihen liittyy ongelmia. Wangin, Lin, Zhangin, Ristenpartin & Swiftin tutkimuksessa todetaan, että cold start-latenssia voi tehokkaasti helpottaa kontin välimuistilla, joka perustuu kontin pitämiseen muistissa päällä funktion suorittamisen jälkeen. Jonaksen ym. tutkimuksessa todetaan kontin lämpimänä pitämisen olevan vastoin palvelimettoman tietojenkäsittelyn ideaa, koska silloin resursseja pidetään jatkuvasti käytössä, sen sijaan että niitä käytettäisiin vain tarpeen mukaan. (Pan, Wang, Chen & Liu 2022.)

Googlen oppaassa ja Wangin, Lin, Zhangin, Ristenpartin & Swiftin tutkimuksessa esitetään tapa helpottaa cold startteja käyttämällä lämpimiä altaita, jotka esikäynnistävät hiekkalaatikon ennen kuin funktio suoritetaan. Näin käynnistysaika piilotetaan, kuitenkaan tinkimättä eristyksen tasosta. Tämä tapa kuitenkin vie laitteistolta resursseja ja hidastaa palvelimen korkeaa yhtenäisyyttä. Lämpimät altaat ovat myös erityisen tehottomia ei-suosituille funktiolle. Shahradin ym. tutkimus näyttää, että vain 18,6 % funktiosta kutsutaan enemmän kuin kerran minuutissa. Enemmistölle funktioista lämpimän altaan käyttö ei ole tehokasta. (Shin, Kim, & Min 2022, 664.) Tavallaan konttien lämmittämisellä palvelimeton tietojenkäsittely saattaa päätyä kohti perinteistä pilviteknologiaa, jossa resurssit ovat jatkuvasti käyttövalmiina pyörimässä.

Joskus lämpimät kontit eivät edes ole suojassa cold startilta. Jos funktiota kutsutaan tarpeeksi usein, voi Leen, Yoonin, Yeon & Ohin (2021, 2-3) mukaan cold start voi tulla ensimmäisenkin funktiokutsun jälkeen. Tällöin palvelimettomassa alustassa otetaan käyttöön automaattinen skaalaus. Siinä kutsutusta funktiosta otetaan käyttöön uusi instanssi, jolloin cold start joudutaan käymään uudestaan läpi, koska kopioidun funktion näkökulmasta tämä on ensimmäinen kutsumiskerta.

2.3 Funktioiden tiladatattomuus ongelmana

2.3.1 Mitä funktioiden tiladatattomuus on?

Palvelimettomassa tietojenkäsittelyssä funktiot ja tiladata on erotettu toisistaan. Jindal, Gerndt, Chadha, Podolskiy & Chen (2021, 2-3) kertoo tutkimuksessa palvelimettomien funktioden olevan ”stateless”, eli tilattomia, joka tarkoittaa, ettei niiden tilaa pidetä yllä. Tiladata varastoidaankin tietokantoihin tai viestipalveluihin. Palvelimettomassa tietojenkäsittelyssä tiladatattomat funktiot hoitavat vain sovelluksen logiikan, ja kaikki tarvittava tiladata tuodaan ulkoisista datavarastoista. Tiladataton malli tekee palvelimettomasta tietojenkäsittelystä kestävämmän kaatumisille, koska menetettyä tiladataa ei tarvitse toivuttaa. (Chard & Foster 2021, 76; Wu, Mi & Xia 1.9.2020.)

Monesti sovellukset kuitenkin vaativat funktioiden olevan ”stateful”, eli tiladatallisia. Chardin & Fostering (2021, 46-76) mukaan tällä hetkellä puhtaasti palvelimettomista funktioista ei voi rakentaa sovelluksia, jotka tarvitsevat tiladataa. Monet sovellukset vaativatkin, että dataa voidaan viedä ja tuoda funktiolle, sekä kuljettaa funktioiden välillä. Tällä hetkellä olemassa olevat datavarastopalvelut kuitenkin rajoittavat tiladatallisia sovelluksia. Rajoituksia ovat esim. rajoitettu läpisyöttö ja latenssit.

2.3.2 Tiladata

Palvelimettomassa teknologiassa funktioiden tiladatan säilyminen ei ole taattua, joten kaikki tiladata ja välillinen data on tallennettava ulkoiseen varastoon. Jopa silloinkin, kun tiladata tallennetaan workerin paikalliselle tallennustilalle, ei ole varmaa, että tiladata säilyy funktioiden kutsujen välillä. Workerilla tarkoitetaan konttia tai virtuaalikonetta. Funktioiden konteksti voidaan päättää osana dynaamista resurssienhallintaa, tai kuormituksen tasaaminen siirtää jatkokutsut toiselle tai uudelle funktioinstanssille. Tiladatattomien funktioiden ulostuloihin ei voi päästä suoraan käsiksi sen jälkeen, kun ne on suoritettu, joten ne täytyy tallentaa ulkoiseen varastoon. (Klimovic ym. 2018, 427-428; Zhang, H., Cardoza, R., Baile Chen, P., Angel, S. & Liu, V. 2020, 1188; Basu Roy, Patel & Tiwari 2021, 75.) Tiladatalliset funktiot täytyykin palvelimettomassa tietojenkäsittelyssä toteuttaa kiertoteitse.

2.3.3 Ulkoinen datavarasto

Tiladatallisten funktioiden luomisessa ulkoisten varastopalveluiden käyttö osoittautuu ongelmaksi. Sieltä datan hakeminen on hidasta, sekä tiladatan ja välillisen datan hallinnointi voi olla myös hankalampaa. Olemassa olevat varastopalvelut eivät ole hyviä jakamaan lyhytikäistä dataa palvelimettomissa alustoissa. Pun, Venkataramanin & Stoican (2019, 193) tutkimuksessa otetaan esimerkiksi CloudSort-suorituskykymittaus. Jos sillä ajetaan 100TB:iä dataa AWS Lambdassa, voi se olla korkeimmillaan 500 kertaa hitaampi kuin jos data olisi ajettu virtuaalikoneilla. AWS Lambdad voivat käyttää kahdenlaisia ulkoisia varastoja: Amazon Simple Storage Servece (S3) ja Elastic File System (EFS). S3 on virtuaalinen avainarvo oliovarasto. Kun data tallennetaan sinne, annetaan sille avainarvo, jota käytetään datan hakemiseen varastosta. Uusi olio luodaan jokaiselle tallennukselle ja uudelleen tallennukselle. Vaikka tämä tapa toimii hyvin joillekin sovelluksille, siltä puuttuu kansiorakenne, kustomointimahdollisuus eikä tietojärjestelmälle ole ominaisuutta monikäyttäjätietoturvallisuudelle. (Klimovic ym. Klimovic ym. 2018, 427-428; Pu, Venkataraman & Stoica 2019, 193; Basu Roy, Patel & Tiwari 2021, 75.) Ulkoisiin varastoihin tallennettava tiladata voi myös olla arkaluontoista. Lin ym. (2022) mukaan arkaluonteisen datan varastoinnissa serverin ulkopuolelle on huomattavat tietoturvaseuraukset.

2.3.4 Muita tiladatan ongelmia

Joissakin tapauksissa tiladata on välttämätöntä, jolloin palvelimettoman tietojenkäsittelyn funktiot on taivutettava tiladatallisiksi. Muutetut funktiot eivät kuitenkaan toimi ongelmitta. Nykyinen FaaS-malli estää tiladatallisten funktioiden tehokasta toteuttamista palvelimettomassa tietojenkäsittelyssä. Jotkut sovellukset tarvitsevat tiladatan, jotta niitä pystytään ylläpitämään. Jos näitä sovelluksia uudelleen kirjoitettaisiin versioiksi ilman tiladataa, olisi se kohtuuttoman tehotonta. Esimerkiksi big data-analytiikan työtaakalla ei ole varaa ladata kokonaista data-aineistoa jokaiselle funktiokutsulle. Jos palvelimetonta alustaa käyttää suoraan data-analytiikan työtaakkoihin, voi se johtaa äärimmäisen tehottomiin suorituksiin. (Copik, Calotoiu, Bruno, Böhringer & Hoefler 2022, 3; Denninnart & Amini Salehi 2021, 3-4; Pu, Venkataraman & Stoica 2019, 193.)

Tiladatalliseksi funktioksi muuttamisesta syntyvä latenssi on erillistä cold start-latenssista. Tilan luominen tiladatattomalle funktiolle voi myös tuottaa latenssia, vaikka funktion hiekkalaatikko olisikin pidetty lämpimänä. Tätä tapahtuu silloin, jos funktioiden tila täytyy hakea ulkoisesta varastosta, koska jokaisen kutsun täytyy käydä varaston rasite läpi (Copik, Calotoiu, Bruno, Böhringer & Hoefler 2022, 3).

Funktioiden ja ulkoisten varastojen välinen yhteistyö on haavoittuvaista virhetilanteille. Palvelimettoman tietojenkäsittelyn tapa pitää funktioiden tilaa yllä, toimii huonosti virheidenkäsittelyn kanssa. Workflow’n kaatuessa tai workerin jumittuessa, palveluntarjoaja joko ei tee mitään, jättää workflow’n kesken, tai käynnistää funktion toiselle workerille, mahdollisesti kasvattaa laskuria yhdellä, tai korruptoi tietokannan tilaa. Googlen dokumentoinnissa suositellaan tällä hetkellä kehittäjiä tekemään funktioita, joilla on turvallista tehdä uudelleensuoritus. Tämä tosin siirtää vastuun kokonaan kehittäjille (Zhang, H., Cardoza, R., Baile Chen, P., Angel, S. & Liu, V. 2020, 1188-1189).

2.4 Bugien korjaus, testaus ja monitorointi ongelmana

2.4.1 Bugien korjaus, testaus ja monitorointi palvelimettomassa tietojenkäsittelyssä

Bugien korjaus, testaus ja monitorointi ovat osa-alueita, joissa palvelimettoman tietojenkäsittelyn kypsymättömyys näkyy. Nämä osa-alueet ovat tehty perinteisille sovelluksille ja alustapalveluille, eikä niitä voi täysin soveltaa palvelimettomiin sovelluksiin. Bugien korjauksissa kehittäjien täytyy käyttää kiertoteitä, testausta ei voi suoraan toteuttaa paikallisessa ympäristössä ja nykyiset monitorointimahdollisuudet ovat rajallisia.

2.4.2 Bugien korjaus

Palvelimettomien sovellusten bugien korjaus eroaa perinteisten sovellusten bugien korjaamisesta. Yksi suuri ongelma on, ettei kehittäjillä ole pääsyä palvelinpuolelle. Hassan, Barakat & Sarhan (12.7.2021) toteaa tutkimuksessaan työkaluista bugien korjaukseen olevan palvelimettomille sovelluksille vajetta. Perinteiset työkalut bugien korjaamiseen olettavat pääsyn palvelimelle, esim. mahdollisuutena root priviledgeen, jotta ne pystyisivät monitoroimaan ja korjaamaan sovellusten bugeja. Nämä työkalut eivät kuitenkaan sovellu palvelimettomiin sovelluksiin, joten uusia tapoja tarvitaan (Castro, Ishakian, Mutusamy & Slominski 7.6.2019).

Bugien korjaamisessa kehittäjien täytyy tehdä päätelmiä ongelmien synnyistä epäsuorin keinoin. Baldinin ym. (10.6.2017) mukaan servereihin ei ole suoraa pääsyä katsomaan, mikä on mennyt väärin, joka tekee palvelimettomien sovellusten bugien korjaamisesta paljon vaikeampaa. Sen sijaan palvelimettomien alustojen täytyy kerätä kaikki data silloin kun koodia suoritetaan, ja tehdä se saatavaksi myöhemmin. Bugien korjaus on myös erilaista, jos sen sijaan että oltaisiin tekemisissä yhden kokonaisuuden kanssa, ovat kehittäjät tekemisissä suuren määrän pienien koodipätkien kanssa. Palvelimettomien sovellusten bugien korjaamisessa logit funktioiden kutsuista täytyy lähettää kehittäjille, ja tarjota tarkat stack tracet. Kun virhetilanne tapahtuu, on hyvä tapa vaatia raporttitiedot tapahtuneesta kehittäjälle. Koska kehittäjillä ei ole pääsyä servereihin, täytyy palvelimettomien palveluiden ja työkalujen keskittyä kehittäjien tuottavuuteen. Ongelmien ja pullonkaulojen paikantaminen on vaikeaa, koska lyhyitä funktiota on iso määrä pyörimässä. Ainoa jälki funktion suoriutuessa loppuun, on se mitä palvelimettomien alustojen monitorointi on tallentanut. (Hassan, Barakat & Sarhan 12.7.2021; Baldini ym. 10.6.2017.) Palvelimettomien sovellusten bugien korjaamisessa kehittäjien täytyy siis tehdä päätelmiä tapahtumien tallennetusta datasta ja logeista.

Amazon on kehittänyt AWS X-Ray:n palvelimettomien sovellusten bugien korjaamista varten. X-Ray liittää funktioaktiviteetit toisiinsa, käyttämällä uniikkeja tunnisteita jokaiselle funktiokutsulle, esittämällä suoritus- ja riippuvuusdatan kehittäjille logeina, sekä yhteenvedon palvelugraafista jokaiselle sovellukselle. X-Ray on kuitenkin rajoittunut mm. seuraavissa asioissa: se ei tarjoa tapahtumien syy-seuraussuhteita, ei tee jäljitystä mm. funktioiden välisistä riippuvuuksista, ja sen historia rajoittuu 24:ään tuntiin (Lin, Krintz, Wolski & Zhang 17.5.2018).

2.4.3 Testaus

Palvelimettomien sovellusten testaaminen paikallisesti on vaikeata tai joskus jopa mahdotontakin. Haasteellisin asia palvelimettomassa testauksessa on mallintaa aito pilviympäristö paikallisiin koneisiin. Pilvipalveluissa tämä on vaikeaa, koska etukäteen ei voi tietää mitä konttia tullaan käyttämään käyttöönotossa. Paikalliseen ympäristöön jäljittäminen on myös sen takia vaikeaa, koska

suurin osa ympäristöriippuvuuksista on saatavilla vain ajon aikana. Paikallinen testaus paikallisissa koneissa eroaa myös testauksesta pilviympäristössä, koska pilvipalveluntarjoajat hallitsevat resursseja. Näin koodin testaaminen realistisessa ympäristössä tulee paljon vaikeammaksi. (Wen, Chen & Liu 27.7.2022; Lenarduzzi & Panichella 13.10.2020.)

Muita vaikeita aiheita testauksessa on testata kaikki mahdolliset tilanteet mitä lambdafunktio voi saada toisilta funktioilta, sekä asynkronisien tapahtumien testaus. Palvelimettomat systeemit ovat myös löyhästi toisiinsa kytkettyjä komponentteja, jonka hyvänä puolena on skaalautuvuus ja helppo rakentaminen, mutta huonona puolena niiden vaikea testaus (Lenarduzzi & Panichella 13.10.2020).

2.4.4 Monitorointi

Perinteiset testaustyökalut eivät ole kelvollisia palvelimettomaan arkkitehtuuriin, koska niillä on oletuksena pääsy serveriin monitoroimaan. Monitorointi on yleisesti aika monimutkaista palvelimettomissa sovelluksissa. Kehittäjät tarvitsevat monitorointityökaluja, koska sovellusta täytyy monitoroida ja havainnoida miten funktiot toimivat. Palvelimettomasta tietojenkäsittelystä tosin puuttuu työkaluja, jotka auttavat hallitsemaan ja monitoroimaan palvelimettomia sovelluksia. Vaikkakin palvelimettomille alustoille löytyy monia kaupallisia monitorointiratkaisuja, kuten Epsagon11, Datafog12 ja Dynatrace 13, ovat nämä ratkaisut kuitenkin yleensä rajoitettuja perusmittareihin kuten CPU:n käyttöön. (Baldini ym. 10.6.2017; Khatri, Kumar Khatri & Mishra 15.9.2020; Hassan, Barakat & Sarhan 12.7.2021; Shafiei, Khonsari & Mousavi 18.2.2022.)

2.5 Tietoturvaongelmat

2.5.1 Syitä tietoturvahaavoittuvuuksille

Palvelimettoman tietojenkäsittelyn tietoturvaongelmat johtuvat mm. samoista asioista, joista hitaus ja bugien korjaamisen vaikeus johtuu. Yksi syy hitauteen johtuu funktioiden tiladatattomuudesta (”stateless”), joka myös tuottaa tietoturvaongelmia. Kun kehittäjillä ei ole pääsyä palvelimelle, vaikeutuu tietoturvan kehittäminen, samoin kuin bugien korjaaminen. Marinin, Perinon & Di Pietron mukaan palvelimettomassa tietojenkäsittelyssä pilvipalveluntarjoajat ovat vastuussa kaiken toiminnallisuuksien ja infrastruktuurin hoitamisesta, mukaan lukien infrastruktuurin ja hostattujen sovellusten suojelemiselta sisäisiltä ja ulkoisilta uhilta. Ongelma on siinä, että pilvipalveluntarjoajat tyypillisesti pitävät suurimman osan niiden infrastruktuurin informaatiosta salassa, jolloin tietoturva-asiantuntijoiden on vaikea tarkastella palvelun tietoturvaa ja yksityisyyssuojaa. Andersonin mukaan tietoturvapiireissä tätä kutsutaan nimityksellä ”security-through-obscurity”, ja se on yleisesti tunnettu vaarallisena tapana. Pilvipalveluntarjoajat myös usein uhraavat tietoturvaa, jotta ne voisivat saada mahtumaan enemmän käyttäjiä infrastruktuuriinsa, jolloin ne pystyvät käyttämään

resurssejaan tehokkaammin, tai tarjota parempaa suorituskykyä asiakkailleen (Marin, Perino & Di Pietro 8.8.2021).

Toinen ongelma on, että palvelimettomassa tietojenkäsittelyssä on laajempi hyökkäyspinta-ala. Tämä johtuu funktioiden tilattomuudesta. Funktioiden on jatkuvasti oltava vuorovaikutuksessa muiden funktioiden ja pilvipalveluiden kanssa, jotta ne pystyisivät toteuttamaan toimintansa. On haasteellista määritellä funktioille, mihin pilvipalveluihin ja toisiin funktioihin niillä on pääsy, koska palvelimeton ympäristö on niin dynaaminen ja monimutkainen. Funktiot voivat myös tulla laukaistuksi monista ulkoisista ja sisäisistä tapahtumalähteistä, joissa on monia formaatteja ja koodauksia (Marin, Perino & Di Pietro 8.8.2021).

2.5.2 Eri tietoturvahyökkäyksiä

Palvelimeton tietojenkäsittely on altis palvelunestohyökkäyksille kuten perinteinenkin teknologia. Sen ollessa niin uusi teknologia, ei sille vielä ole kehitetty kunnolla suojausvälineitä näille hyökkäyksille. Shafiein, Knonsarin & Mousavin mukaan palvelimettomat palvelut ovat herkkiä toistohyökkäyksille. Näissä hyökkäyksissä hyökkääjä sieppaa suojatun funktiosuorituspyynnön ja toistaa sen uudestaan sabotoidakseen systeemin normaalia toimintaa. Palvelimettomassa tietojenkäsittelyssä ei nykyisellään ole tapaa havaita tai estää toistohyökkäyksiä. Resurssien uuvuttamishyökkäyksissä taas kohteen resursseja yli-käytetään, jotta palvelun toiminta häiriintyisi, tai luodaan liiallisia finanssi- tai rahavirtakuormia. Vaikka palvelimettomaan tietojenkäsittelyyn kuuluu autoskaalautuvuus, voi kuorma ylittää SLA-vaatimukset. Tällöin myöhemmän pyynnöt estetään, tai ainakin sovelluksen omistajalle voi aiheutua raskas finanssikuorma. Palvelimettomassa tietojenkäsittelyssä ei ole monitorointitapoja resurssien uuvuttamishyökkäyksiä varten (Shafiei, Khonsari & Mousavi 18.2.2022).

Yksityisyydensuojahyökkäyksissä hyökkääjän tarkoitus ei ole muuttaa systeemin normaalia toimintaa, vaan päätellä tietoa käyttäjistä. Tämä tehdään käyttäen minimaalinen määrä saatua tietoa. On paljon kontekstuaalista dataa, jota hyökkääjä voi kerätä päätelläkseen tietoa käyttäjästä. Tämä data on erityisesti saatavilla, kun tietoverkko käyttää pelkästään sovelluskerroksen tietoturvaprotokollia (Shafiei, Khonsari & Mousavi 18.2.2022).

O’Meara & Lennon (31.8.2020) tutkimuksessa listataan palvelimettomalla testisovelluksella seuraavia tietoturvahaavoittuvuuksia: API-porttia käytetään validoimaan pyyntöjen autentikointi. Haavoittuvuudet autentikointiprotokollissa saattaa mahdollistaa front endin sivuuttamisen, kun API:a käytetään suoraan lähettämään dataa API-porttiin. Haavoittuvuuksia voi olla pyyntöjen todentavissa funktioissa, koska niille voidaan antaa pahantahtoista dataa odottamattomista lähteistä. Pyyntöjä käsittelevillä funktioilla voi olla tietoturvahaavoittuvuuksia, jos autentikointiprotokolla on

heikko, ja datan lähetykseen funktiolle käytetään suoraan API:a sivuuttaen kaikki todennusvaiheet. Tietokannan tietoturva-aukot saattavat aiheuttaa datamurtumia, koska luvattomat komennot saattavat paljastaa arkaluonteista dataa. Simple Email Systemiä käytetään tiedon jakeluun sähköpostin kautta. Jos se on huonosti suojattu, voi hyökkääjä lähettää arkaluontoista dataa sovelluksen arkkitehtuurin ulkopuolelle.

2.5.3 Auktorisointi ja autentikointi

Palveluttomille palveluille auktorisointi on tietoturvaongelma. Palvelimettomissa palveluissa funktio tai käyttäjä auktorisoidaan kutsumaan toinen funktio, ja kunnollinen auktorisoinnin puute voi olla vakava uhka sovelluksen tietoturvalle. Ilman auktorisointia funktioita voidaan käyttää ilman sovelluksen omistajan lupaa. Amazon Lambdassa käyttäjät sijoittavat staattisesti funktiot rooleihin, jotka liittyvät eri lupiin. Huonona puolena tämä on rajoitettu vain yksittäisiin funktioihin, eikä workflow-tason pääsyä pysty kontrolloimaan (Shafiei, Khonsari & Mousavi 18.2.2022).

Vankka autentikointimalli, jossa voi kontrolloida pääsyä ja tarjota suojaa oleellisille funktioille, tapahtumatyypeille ja laukaisijoille, voi olla monimutkainen hanke. Se myös tarvitsee jatkuvia uudelleen katsauksia. Esimerkkinä sovelluksella voi olla julkisia API:ja, jotka ovat tietosuojattuja kunnolliselta autentikoinnilta. Back endissä sovellus voi taas lukea dataa pilvipalveluvarastosta ilman kunnollista autentikointia, joka altistaa epäautentikoiden raon hakkereille (Patnayakuni & Patnayakuni 3.5.2019).

2.6 Cold startin ratkaisuja

2.6.1 Funktioiden fuusioiminen

Leen, Yoonin, Yeon & Ohin (2021) tutkimuksessa esitetään automatisoitu tapa helpottaa workflow’n cold start-latenssia fuusioimalla funktiot, ja ottamalla fuusiossa huomioon rinnakkaiset funktioajot. Tämä automaatioprosessi on toteutettu AWS Lambdalla, mutta sen voi mukauttaa helposti muille palvelimettomille alustoille. Tutkimuksissa ratkaisulla on onnistuttu saamaan viidelle workflow’lle vastausajaksi 28-86 %:ia alkuperäisestä vastausajasta.

Tutkimuksen ratkaisussa perättäisiä funktioita fuusioidaan vain, jos suoritusaika ei nouse verrattuna yksittäisten funktioiden suorittamiseen. Funktiot, joista haarautuu monta ulostuloa, fuusioidaan myös. Tässäkin arvioidaan fuusioimisen kannattavuus: jos latenssin väheneminen painaa enemmän kuin ali-workflow’n suoritusajan lisääntyminen, toteutetaan fuusioiminen. Funktioiden fuusioimisessa yhdistetään funktioita kaksi tai enemmän yhteen yhdeksi funktioiksi. Näin cold start tarvitsee käydä läpi vain yhden kerran. Jos funktion ulostulot haarautuvat rinnakkaisiksi funktioiksi, muutetaan nämä fuusiossa peräkkäisiksi funktioiksi. Rinnakkaiden funktioiden fuusioimisessa voi

tosin olla vaarana, että rinnakkaisen suorittamisen menettämisestä johtuva suoritusajan piteneminen on suurempi kuin cold startin lyheneminen. Tutkimuksen ehdottama ratkaisu ottaa selville optimaalisen fuusiointistrategian. Jotta ratkaisu voi ennakoida täyden workflow'n keston ja selvittää tuottaako fuusio workflow'n keston lyhenemistä, käy ratkaisu workflow'n läpi, ja fuusioi kaksi peräkkäistä workflow'n osiota sekä vertailee tuloksia (Lee, Yoon, Yeo & Oh 2021).

2.6.2 Konttien lämpimänä pitämisen optimointi

Vahidinian, Farahanin & Alieen (5.4.2022) tutkimuksessa esitetään kaksikerroksinen tapa vähentää cold starttia, ja tapaa testattiin palvelimettomalle OpenWhisk alustalla. Yleisin tapa vähentää cold starttia on pitää kontit lämpiminä tietyn kiinteän ajan verran funktion suorittamisen jälkeen. Tutkimuksessa esitetyn ratkaisun tapa eroaa tästä niin, että ratkaisun ensimmäinen kerros päättää parhaan ajan siitä, kuinka pitkään kontit pidetään lämpimänä vähentääkseen cold starttien ja resurssien käyttämisen määrää. Resurssien käyttäminen otetaan huomioon sen takia, koska mitä pidempään kontti on lämpimänä, sitä enemmän kulutetaan resursseja.

Ensimmäinen kerros oppii algoritmin avulla funktion kutsumiskuviot tarkkailemalla menneiden kutsujen aikavälejä, ja päättää siitä, kuinka pitkään kontti pidetään lämpimänä. Koska kontin lämpimänä pitäminen kuluttaa resursseja, kerroksen algoritmi tasapainottaa cold starttien määrän ja muistin, eli resurssien hukkaamisen. Toinen kerros vähentää cold startin pituutta. Koska jotkut funktiokutsut tapahtuvat pitkillä aikaväleillä, konttien lämpimänä pitäminen hukkaa muistia. Jotkut kutsut päätyvätkin kylmiin kontteihin. Kutsuja voi myös tulla samanaikaisesti, jolloin lämpimiä konttеja ei välttämättä riitä kaikille. Toinen kerros ennustaa maksimimäärän samanaikaisia kutsuja tietylle ajalle, ja päättää sen mukaan esilämmitettyjen konttien määrän (Vahidinia, Farahani & Aliee, 5.4.2022).

2.6.3 Koodin optimointi (LambdaLite)

LambdaLite (Wen ym. 17.7.2022) on sovellustason lähestymistapa, joka optimoi cold start-latenssia pienentämällä suoritettavan koodin määrää, esim. lataamalla vain vaaditun koodin. LambaLite tunnistaa ja erottelee koodit välttämättömiksi ja valinnaisiksi. Välttämätön koodi on yhteydessä sovelluksen toiminnallisuuksiin. Muu koodi, joka on valinnaista, erotellaan alkuperäisestä sovelluksesta analysoimalla sovelluksen koodin välillinen esitystapa. Valinnainen koodi pakataan kevyeksi tiedostoksi, ja se ladataan käyttöön vasta tarvittaessa. Näin koodin kokoa voi pienentää lataamisprosessissa varmistaen samalla palvelimettoman sovelluksen toimivuuden. LambdaLite toimii automaattisesti, eikä vaadi palvelimettomien alustojen mukauttamista. Kun kehittäjät lataavat sovelluksensa palvelimelle, optimoi LambdaLite koodin automaattisesti.

Wenin ym. (17.7.2022) Tutkimuksessa LambdaLitea kokeiltiin 15:sta oikean maailman sovelluksessa. Tuloksissa koodin latauslatenssia voitiin pienentää keskimäärin 28,78 %, ja parhaimmillaan 78,95 %, jolloin myös cold start-latenssi lyheni. Tuloksena kokonaisvastauslatenssia voidaan lyhentää keskimäärin 19,21 %, ja parhaimmillaan 42,05 %. Lisähyötynä LambdaLite voi vähentää ajonaikaista muistia keskimäärin 14,79 %, ja parhaimmillaan 58,82 %, joka johtuu ladatun koodin vähenemisestä. Verrattuna uusimpiin teknologioihin, LambdaLite saavuttaa 21,25 kertaisen parannuksen kokonaisvastauslatenssiin.

2.6.4 Funktiokoodin optimointi runtimessa (Ignite)

Ignite (Carreira, Kohli, Bruno & Fonseca 3.6.2021) on holistinen systeemi, jossa runtimet tekevät yhteistyötä generoidakseen optimoitua funktiokoodia, ja palvelimettomat vuorottajat voivat vuorottaa funktiot riippuen jokaisen runtimen optimointiputken tilasta. Ignite vähentää ylimääräistä profilointia ja käännöstä, jakamalla koodioptimointi-informaatiota runtimessa. Kun Ignite on generoinnut optimaallisen koodin tietylle funktiolle, ovat näiden funktioiden kutsut aina ”kuumia”, eikä ylimääräistä profilointia ja käännöstä tule.

Cold startissa funktioiden on ladattava kirjastonsa. Kun funktiot voidaan seuraavan kerran warm starttina, ei funktion suoritus ole vielä optimaalinen, koska funktion koodia ei ole vielä optimoitu. Esim. Java WM Hotspot vaatii oletuksena tuhat kutsua funktioille, kunnes funktio käännetään konekielelle. Lopulta runtime tunnistaa ja kääntää kuumia metodeja, jolloin jatkossa funktioiden kutsut suoritetaan optimoidulla koodille, joka on ”hot start”. Ignitessa runtime pystyvät palauttamaan käännösputken tilan funktion kutsun yhteydessä. Näin funktiot voidaan suorittaa täysin optimoidulla koodilla alusta lähtien, jolloin suurin osa cold -ja warm starteista voidaan vaihtaa hot starteiksi (Carreira, Kohli, Bruno & Fonseca 3.6.2021).

Carreira, Kohli, Bruno & Fonseca kehittivät kokeita varten simulaattorin, joka simuloi palvelimetonta alustaa. Siinä simuloitiin sarjaa kutsuja samalle funktiolle. Simuloinnille käytettiin laskennallisesti intensiivisiä sovelluksia: numerohashausta, HTML-renderöintiä ja sanojen laskemista tekstikokoelmassa. Simulaatiossa simuloitiin 50 itsenäistä konttia. Kokeessa parannuksia tuli skaalalta 1,26-kertaisesta (HTML-renderöinti) 5,5-kertaiseen (numerohashaus) suoritukseen. Parannukset tulivat kulujen ja latenssin vähenemisenä (Carreira, Kohli, Bruno & Fonseca 3.6.2021).

2.6.5 Runtimen uudelleen käyttö (HotC)

Suon, Sonin, Chengin, Chening & Baidyan (13.10.2021) tutkimuksessa esitetään HotC, joka on yksinkertainen ja kevyt ratkaisu, jossa voi uudelleen käyttää runtimeja sitä mukaan, kun asiakkailta tulee pyyntöjä. HotC on väliohjelmisto asiakkaiden ja back endin välillä. HotC pitää yllä

runtimeallasta, ja käyttää tehokkaasti uudelleen kontteja asiakkaiden pyyntöjen mukaan. Resurssien kontrollointiin ja jaksottaisten cold starttien helpottamiseen käytetään hyödyksi konttien runtimehistoriaa.

Kun uusi pyyntö saapuu, yrittää HotC aina suorittaa käyttäjän koodin olemassa olevassa ja vapaassa kontissa. Jos tällaista konttia ei löydy, käynnistetään uusi kontti. Kun kontti on tehnyt suorituksen loppuun, palauttaa se tulokset asiakaspuolelle, jonka jälkeen kontti tyhjennetään, ja valmistaudutaan seuraavalle pyynnölle. Jotta kontteja voitaisiin uudelleen käyttää tehokkaasti, on tärkeää pitää käytetyt kontit puhtaina. Kun kontitettujen sovellusten käyttö lopetetaan, ottaa käyttöjärjestelmä CPU- ja muistiresurssit takaisin itselleen. HotC pitää yllä runtimeallasta konteille, ja päivittää allasta mukautuvasti ajan kuluessa. Tässä tavassa on etunsa, koska se on yksinkertainen ja suoraviivainen, eikä sisällä häiritseviä muutoksia olemassa olevaan arkkitehtuuriin. Resurssien kulutus tulee pääasiassa sovelluksen suorittamisesta, eikä kontista itsestään. On kevyttä ylläpitää joukkoa päällä olevia kontteja ilman että tulisi ylimääräistä rasitetta. Saman kontin runtimen uudelleen käyttö voi myös tarjota kuuman välimuistin (Suo, Son, Cheng, Chen & Baidya 13.10.2021).

HotC:n ensimmäisenä askeleena on analysoida käyttäjän komento tai konfiguraatiotiedosto, ottaakseen selville konttiruntimen parametriasetus. Tässä parametrissa on kontin image, verkkokonfiguraatiot, UNIX Time Sharing-asetukset, Inter Processor Communication-asetukset, suoritusvaihtoehdot jne. Jotta olemassa olevasta kontista voidaan saada hyötyä, on ideaalia suorittaa sovellus samassa runtimeympäristössä. HotC selvittää käyttäjän syötön, ja etsii kontteja ehdokkaiksi uudelleen käytölle, joilla on sama runtime. Jos tällainen kontti löytyy ja on vapaana käyttöön, ladataan käyttäjän koodi tähän konttiin suoritettavaksi (Suo, Son, Cheng, Chen & Baidya 13.10.2021).

Tutkimuksessa tehtiin kokeita, jossa katsottiin käynnistysaikaa kahdelta kuvantunnistussovellukselta käyttäen HotC:tä. Tulokset olivat keskimäärin kymmeneltä suorituskerralta. Toiselta sovellukselta suoritusaika väheni 33,2 %:lla, ja toiselta 23,9 %:lla. Kokeessa selvitettiin myös HotC:n suorituskykyä pyyntöpurkauksissa. Kokeessa tehtiin pyyntöjä, joissa oli kahdeksan pyyntökertaa. Ajan mittaan pyyntöjen määrää kymmenenkertaistettiin useita kertoja. Ensimmäisessä pyyntöpurkauksessa HotC pystyi vähentämään latenssia noin 9 %. Myöhemmissä purkauksissa latenssia pystyttiin vähentämään parhaimmillaan 73 %. Latenssin vähenemisen parantuminen johtuu siitä, että saman tyyppisiä kontteja on enemmän saatavilla edellisten purkauksien jälkeen (Suo, Son, Cheng, Chen & Baidya 13.10.2021).

2.6.6 Geneerinen moottori (Nuka)

Nuka on geneerinen moottori, jolla on millisekuntien alustusaika palvelimettomalle tietojenkäsittelylle. Nuka vähentää aikakuluja cold startissa kolmella tekniikalla. Ensimmäisessä tekniikassa

eristysallas uudelleen käyttää namespaceä ja cgroupia kevyiden taukokonttien kautta altaassa. Se välttää huolellisesti Docker-kontin luomisesta ja poistamisesta tulevan pullonkaulan. Tätä toteutetaan korkealla samanaikaisuudella. Eristysaltaan olennaisin idea on mitätöidä sen hetkisen verkko-namespacen, cgroupsin ja mount namespacen luomisesta tulevat korkeat kustannukset (Qin ym. 12.11.2022).

Toisessa tekniikassa paikallinen pakkausvälimuisti välttää imagen vetämisen. Tämä tehdään dynaamisesti selvittämällä ja tuomalla vaaditut pakkaukset paikallisesta pakkausvälimuistista. Paikallisen pakkausvälimuistin päätarkoitus on poistaa imagen vetämisen ulkoisesta Docker rekisteristä, sekä vähentää ohjelmistopakkausten latausta ja asentamista. Olennaisia asioita, joita paikallinen pakkausvälimuisti sisältää, on tuki usealle ohjelmointikielelle sekä välimuististrategia suosituille ja epäsuosituille pakkauksille. Suositut pakkaukset asennetaan jokaiselle worker-koneelle pysyvästi. Muut pakkaukset laitetaan välimuistiin (Qin ym. 12.11.2022).

Kolmannessa tekniikassa käytetään itsemukautuvaa kontin uudelleen käyttöstrategiaa, jossa kontrolloidaan kontin taukoaikaa ja kopioiden määrää. Verrattuna Dockeriin, pystyy Nuka saamaan millisekuntien alustamisen korkealla samanaikaisuudella. Nuka myös vähentää cold startin keskimääräistä aikakustannusta kuusinkertaisesti verrattuna olemassa oleviin palvelimettomiin alustoihin. Nuka käyttää taukokonttia, jossa on vain kevyt daemon-tarkistaja, joka tarkastaa ja palauttaa konfiguraatiot ja tiedostot, kun uusi prosessiryhmä on lisätty taukokontin namespaceen. Tämä tehdään todella nopeasti, ja keskimääräinen aikakustannus on vain useita millisekunteja. Näin tarkistaja varmistaa puhtaan namespace-ympäristön, eikä prosessiryhmää häiritä. Suoritettavaan funktioon liittyvä tarkistaja ja prosessiryhmä aktivoidaan vapaaseen taukokonttiin, jolloin funktio voidaan suorittaa eristyksissä (Qin ym. 12.11.2022).

Nukassa on kahdenlaisia pyyntökuormia: vakaat kuormat ja flash crowdit. Vakaassa kuormassa laukaistua kontin taukoa pidetään päällä jonkin aikaa, ilman että sitä lopetetaan funktion suorittamisen jälkeen. Kun seuraava pyyntö on otettu vastaan, jatketaan taukokontin suorittamista käsittelemään pyyntö. Näin cold starttia ei tapahdu. Flash crowd on yhtäkkinen ja iso määrä liikennettä workloadissa. Nuka hoitaa flash crowdin ison kuorman konttien automaattisella skaalaamisella (Qin ym. 12.11.2022).

Tutkimuksen kokeessa testattiin eristysaltaan skaalaavuutta. Kokeen eristysaltaassa oli 256 taukokonttia, ja muita kontteja käynnistettiin samanaikaisesti. Konttien käynnistyslatenssi ei kasvanut suhteellisesti, jos niitä oli alle taukokonttien. Jos käynnistettäviä kontteja ei ole yli taukokonttien, on käynnistyksien mahdollista uudelleen käyttää olemassa olevia eristyskonfiguraatioita. Kaikki ylimenevät konttien käynnistykset suoritetaan ilman tätä etua (Qin ym. 12.11.2022).

Kokeissa testattiin myös, kuinka paljon latenssia syntyy imagen vetämisestä ja pakkausten tuomisesta. Latenssia pystyttiin vähentämään keskimäärin sadoista millisekunneista noin 20:een millisekuntiin. Kontin uudelleen käyttökokeissa verrattiin Nukaa ei-optimoituihin ratkaisuihin. Vakaissa kuormissa CPU:n hyödyntämistä saatiin nostettua 20-40 %, ja flash crowdissa alustamislatenssi voitiin vähentää 50:een millisekuntiin, kun samanaikaisuuksia oli 256. Nukan tehokkuutta testattiin myös käyttämällä AWS Lambdan näytesovellusta, joka vaihtaa kuvien kokoa. Verrattuna Dockeriin, pystyttiin cold start-latenssia vähentämään kuusinkertaisesti. Samanaikaisuuskokeessa latenssia pystyttiin pienentämään 5-10-kertaisesti, ja kohottamaan suoritustehoa 3-20-kertaiseksi. Verrattuna SOCK:iin, sai Nuka samanaikaisuuskokeessa latenssia pienennettyä 3-5-kertaisesti, ja kohottamaan suoritustehoa kolminkertaisesti (Qin ym. 12.11.2022).

2.7 Tiladatan tallentaminen paikalliseen muistiin (Praas)

Koska monet sovellukset vaativat funktioiden olevan tiladatallisia, täytyy tiladata hoitaa kiertoteitse hitaiden datavarastojen kautta. Eräs ratkaisu on PraaS (Process-as-a-Service), joka mahdollistaa funktioiden tekemisen tiladatalliseksi ilman, että tiladataa täytyy hakea hitaasti ulkoisesta datavarastosta. Copikin, Calotoiun, Brunon, Böhringerin & Hoeflerin (2022) tutkimuksessa esitettävässä PraaS:sa tiladata on sessioissa. Sessioissa on muisti tilapäiselle datalle, jota funktiot voivat käyttää. Tätä muistia voivat käyttää funktiot, joita kutsutaan kyseessä olevassa sessioissa. Näin tiladataan pääsee nopeasti käsiksi. Paikallinen muisti onkin session pääkomponentti. Kutsujen sisältämä data menee suoraan muistiin, ja on kutsuttavien funktioiden käytössä. Samassa sessiossa saman aikaisesti suoritettava funktiot voivat myös kommunikoida keskenään. PraaS käyttää p2p-kommunikaatiota saadakseen nopeamman siirtonopeuden.

PraaS:sa käytetään prosessia, jossa on välillisiä funktioita, joidenka tila on ohimenevää eikä ole liitoksissa mihinkään. Prosessissa voi olla monta käyttäjää, ja tarpeen mukaan käyttäjät ovat eristettyjä toisistaan. Sessiot voivat olla yhteistyössä käyttämällä globaalisti saatavilla olevaa prosessimuistia. Globaali muisti on tehokas ja suora kommunikaatiomalli palvelimettomille sovelluksille, jolla voidaan korvata ulkoisten datavarastojen käyttö. Sessioiden avulla PraaS:sa voi käsitellä eri käyttäjien ja workflow’den kuormia yhdessä prosessissa. Funktiolla ei ole suoraa pääsyä eri sessioiden muistiin, mutta ne voivat olla yhteydessä toisten sessioiden funktioihin käyttämällä globaalia prosessimuistia. Näin sessiot voivat kommunikoida toisten sessioiden kanssa lukemalla ja kirjoittamalla muistiolioita globaaliin prosessimuistiin (Copik, Calotoiu, Bruno, Böhringer & Hoefler 2022).

Tutkimuksen kokeessa PraaS:ia verrattiin AWS:ään. Funktioiden kommunikointi käyttämällä PraaS:in session muistia, on kaksi kertaluokkaa nopeampaa kuin jos kommunikointiin käyttäisi AWS S3-datavarastoa. Verrattuna Amazon Lambda-funktioon, on nopeus PraaS:in session muistissa kymmenkertainen (Copik, Calotoiu, Bruno, Böhringer & Hoefler 2022).

2.8 Bugien korjauksen, testauksen ja monitoroinnin ongelmien ratkaisuja

2.8.1 Testaus ja monitorointi

Markkinoilla on tarjolla monia, vapaan lähdekoodin ja kaupallisia, testaustyökaluja suoritusta varten, kuten: Jmeter, Gatling, OpenSTA, Micro Focus, HP Mercury Loadrunner, IBM Performance Tester, Blaze Meter Neoload jne. Nämä työkalut pystyvät simuloimaan isoa määrää käyttäjien pyyntöjä, jolla matkitaan käyttäjien todellista käyttäytymistä tuotannossa. Palvelin vastaa jokaiselle pyynnölle, ja pyyntöön kulunut aika otetaan talteen SLA:n validointia varten. Amazon on tehnyt Cloud9 IDE:n, joka julkaistiin joulukuussa 2017. Sillä voi testata Lambda-funktioita paikallisesti. AWS SAM:lla (Serverless Application Model) voi yksinkertaisella tavalla testata funktioita paikallisesti ja ulkoisesti. Sillä voi myös generoida tapahtumia laukaisemaan Lambda-funktioita (Khatri, Khatri & Mishra 15.9.2020; Kaplunovich 19.9.2019).

Kun lähdetään mittaamaan palvelimettoman tietojenkäsittelyn suoritusta, on testausstrategia avainasemassa testaus- ja monitorointityökalujen lisäksi. Koneoppiminen, syväoppiminen ja NPL (Natural Language Processing) voivat auttaa monella tapaa testauksessa. Koneoppiminen voi parantaa tarkkuuttaa, antaa parempaa testikattavuutta ja säästää aikaa aikaisilla ennusteilla. Se myös tarjoaa tarkan näkyvyyden komponenttien ja sovelluksen suorituksesta. AWS:llä, Googlella ja Azurella on tarjolla koneoppimisen algoritmeja. Suurin osa algoritmeista jaetaan ohjattuun -ja ei-ohjattuun oppimiseen. Työkalut suorituksen monitorointiin, kuten Splunk, Nagios, Appdynamics ja Dynatrace saattavat auttaa suorituksen pullonkaulojen selvittämisessä. AWS:llä on Cloud Watch monitorointia varten. Siitä löytyy mittareita CPU:lle, muistille ja muille hyödyllisille mittauskohteille (Khatri, Khatri & Mishra 15.9.2020).

2.8.2 Muita tapoja hoitaa bugien korjaamista

Yksi tapa helpottaa FaaS-sovellusten bugien korjaamista on käyttää parempaa monitorointi- ja testaustyökaluja, jotka kattava kaikki sovelluksen palvelut. Erityisesti jaetun jäljittämisen ja- loggauksen käyttö automatisoidulla analysoinnilla voi helpottaa tätä rasitetta. Laskennallisesti intensiivisissä sovelluksissa voi helposti ylittyä funktiokäsittelijöiden suoritusajan aikaraja. Kuhlenkampin, Wernerin & Tain väittävät, että välttämättömät tavat havaita viat, jotka johtavat tähän ongelmaan, on suorituksen jatkuva testaus ja- monitorointi, luotettavuus ja kustannustehokkuus (Kuhlenkamp, Werner & Tai 19.5.2020).

2.9 Tietoturvaongelmien ratkaisuja

2.9.1 Sovelluksen tietoturvallinen arkkitehtuuri

Sovelluksen arkkitehtuuri kannattaa erotella esimerkiksi kolmitasoiseksi, jolloin siinä on kerroksen esitykselle, sovellukselle ja datalle. Nämä kerrokset täytyy määritellä selkeästi, jolloin pystyy paremmin keskittymään kerrosten välisiin rajapintoihin, sekä tietoturvan räätälöimisen jokaiselle kerrokselle erikseen. Jos rajapinnat ja vuorovaikutukset kerrosten välillä on selkeitä, on sovelluksen käyttäytyminen selkeämpää, ja epäilyttävä toiminta on helpompi havaita. Tietoturva-alueet täytyy erotella kerrosten väliin, jotta voidaan varmistaa että käyttäjä- ja palvelutilit ovat auktorisoituneita vain palveluille, jotka kuuluvat heidän rooleihinsa (O'Meara & Lennon 2020).

2.9.2 AWS-käyttöoikeudet

Amazon AWS:sä käyttöoikeudet annetaan IAM-entiteeteille, johon kuuluu käyttäjät, ryhmät ja roolit. IAM:iin voi liittää linjauksen mikä määrittelee tyypin pääsylle, mitä toimintoja voidaan suorittaa ja resurssit, joille toiminnot voidaan suorittaa. Lisäksi voidaan määritellä ehdot pääsylle (Amazon). AWS:sä on tärkeää ymmärtää ja hallita IAM-käyttöoikeuksia. IAM:ssa on kaksi osaa. Ensimmäinen osa on käyttäjän tai palvelun funktioiden luomiselle, käyttöönotolle ja poistamiselle, ja toinen osa on käyttöönotetun funktion/konttipalvelun rooli/käyttöoikeus. Funktioiden IAM-roolien ja -käyttöoikeuksien täytyy olla näkyviä. Funktioilla täytyy myös olla oikeus käyttöönottoon. Chetal ym. toteaakin tutkimuksessaan, että et voi kontrolloida sitä mitä et näe. Funktioiden on käytettävä tietoturvaa varten pienimmän oikeuden mallia (Chetal ym. 2021).

2.9.3 Kubernetesin tietoturva

Kubernetes on vapaan lähdekoodin järjestelmä kontitettujen sovellusten käyttöönottoon, skaalaukseen ja hallinnointiin (Kubernetes). Järjestelmä on kokonaan API-käyttöinen. Tämän takia tietoturvan ensimmäinen kerros on rajoittaa pääsyä ja toimintojen suorittamista. Kubernetesissä kaikki API-kommunikaatio on salattu oletuksena TLS:llä. Kaikki API:n asiakkaat täytyy olla autentikoituja, jopa nekin, jotka ovat osa infrastruktuuria kuten solmut, välityspalvelimet ja aikatauluttajat. API:n päätepiste täyty pitää salassa muulta internetiltä. Siihen täytyy olla pääsy vain sen verkon sisällä, johon Kubernetes-rypäs on otettu käyttöön. Kubernetesin API täytyy ottaa käyttöön korkealla saatavuudella ja vikasietoisella käyttöönotolla. Näin täytetään saatavuusvaatimukset, jolloin varmistetaan, että rypästä voidaan hallinnoida hyökkäyksen aikana ja palvelun laatu voidaan taata (Chetal ym. 2021).

Kubernetes varastoi konfiguraatio- ja tiladatan ETCD:hen, joka on yleinen avainarvovarasto. Jos käyttäjä pystyy tallentamaan dataa ETCD:hen, voi käyttäjä myös tehokkaasti kontrolloida

Kubernetes-rypästä. Kaikki salainen tieto täytyy salata ETCD:ssä, koska tietoturvahyökkääjä saattaa saada vihjeitä murrolle vain lukemalla ETCD:n sisältöä (Chetal ym. 2021).

3 Tutkimuksen toteutus

Tein opinnäytetyöni tutkimustyyppisenä. Käytin tutkimusmenetelminä kvantitatiivista -ja kvalitatiivista menetelmää. Käytin kvantitatiivista menetelmää, jotta pystyisin laskemaan eri käsiteltyjen ongelmien määrän eri tutkimuksissa. Näin sain selville, ainakin tiedeyhteisön kannalta, mitkä olivat yleisimmät ongelmat palvelimettomassa tietojenkäsittelyssä. Kvalitatiivista menetelmää käytin tutkimaan minkälaisia yleisimmät ongelmat olivat, ja minkälaisia ratkaisuja niihin oli tarjolla.

Tutkimuksen aiheeksi valitsin palvelimettoman tietojenkäsittelyn yleisimmät ongelmat ja niihin tarjotut ratkaisut. Etsin itselleni tutkimukseen aihetta, ja törmäsin palvelimettomaan tietojenkäsittelyyn, joka vaikutti kiinnostavalta. Ongelmat ja ratkaisut valitsin sen takia, jotta tutkimuksesta olisi mahdollisimman paljon suoraa käytännön hyötyä tämän teknologian kehittäjille. Kehitystyöhän on kuitenkin jatkuvaa ongelmien selvittämistä. Lopulta vielä rajasin tutkimuksen selvittämään yleisimmät ongelmat ja niihin tarjotut ratkaisut, jotta aihe ei paisuisi liian laajaksi.

Hain aineistoa tutkimukseeni Google Scholaria käyttäen. Aineistona oli muita tutkimuksia palvelimettomasta tietojenkäsittelystä. Käytin hakusanoina ”serverless computing problems” ja ”serverless computing challenges”. Löydettyjen tutkimusten käsitellyistä aiheista sain lisää hakusanoja, mm. “cold start” ja ”security”, joiden mukaan lisäsin uusissa hauissa ”serverless computing”.

Tutkimusmenetelminä käytin kvantitatiivista -ja kvalitatiivista menetelmää. Ensin kävin läpi haetut tutkimukset, joita oli 67, ja selasin tutkimukset läpi käyttäen hakusanoja ”problem”, ”issue” ja ”challenge”, jotta löytäisin käsiteltäviä ongelmia. Tein erillisen ”tulokset”-nimisen Word-tiedoston, johon kirjasin ylös kaikki käsitellyt ongelmat, ja missä tutkimuksessa ongelmaa oli käsitelty. Sen jälkeen laskin kuinka monta kertaa eri ongelmia oli käsitelty. Lopulta listasin kaikki ongelmat listattuna suurimmasta pienimpään (tutkimuksissa käsiteltyjen määrien mukaan), ja lisäsin kaikkien ongelmien alle tutkimukset, joissa niitä oli käsitelty. Tämä oli tutkimuksen kvantitatiivinen osa.

Kvantitatiivisen osan jälkeen aloin kirjoittaa tietoperustaa siitä minkälaisia yleisimmän ongelmat ovat, ja mitä ratkaisuja niihin on tarjottu. Tämä oli tutkimuksen kvalitatiivinen osa. Tietoperustaan valitsin lähteeksi ”tulokset” Word-tiedostosta tutkimuksen kirjoitettavan aiheen mukaan. Valitsin ensisijaisesti uusimmat tutkimukset, ja lisäsinkin ”tulokset”-tiedostoon vuosiluvut yleisimpien ongelmien tutkimuksiin helpottaakseni valikointia. Kun olin löytänyt yhdestä tutkimuksesta olennaiset tiedot ja kirjoittanut ne tietoperustaan, lisäsin merkinnän ”tulokset”-tiedostoon kyseiselle tutkimukselle, jotta pystyin pitämään kirjaa siitä mitkä tutkimukset olin käynyt jo läpi.

4 Tulokset

4.1 Kvantitatiivisen tutkimuksen tulokset

Kvantitatiivisen tutkimuksen aineistona oli 67 eri tutkimusta. Cold start oli ylivoimaisesti yleisin ongelma, ja siitä kirjoitettiin 32:ssa tutkimuksessa. Toiseksi yleisimmät ongelmat olivat tiladatattomat funktiot (”stateless”), tietoturva, sekä bugien korjaus, monitorointi ja testaus. Tiladatattomista funktioista kirjoitettiin 14:sta tutkimuksessa, ja kahdesta jälkimmäisestä ongelmasta 13:sta kertaa. En nähnyt aiheelliseksi listata ongelmia yksittäisesti suuruuden perusteella, koska cold startin jälkeen tulevien kolmen yleisimmän ongelman määrät olivat niin tasaväkisiä. Muita ongelmia, jotka olivat neljän yleisimmän ongelman perässä, olivat vuorottaminen (”scheduling”), hinta, suorituskyky ja funktioiden kommunikaatio (liite 1).

4.2 Kvalitatiivisen tutkimuksen tulokset

4.2.1 Cold startin ongelmia ja ratkaisuja

Cold start-latenssi syntyy, kun instanssi (kontti, hiekkalaatikko ym.) funktion suorittamista varten joudutaan luomaan uudelleen ja hoitamaan sille kaikki alustustyö. Siitä syntyvä pitkä latenssi on merkittävä suhteessa palvelimettoman tietojenkäsittelyn lyhyisiin funktioihin, ja onkin huomattava ongelma. Jos funktiota kutsutaan epäsäännöllisesti, täytyy cold start käydä joka kerta läpi. Ohjelmointikielet vaikuttavat myös cold starttiin. Koodin optimointi ongelman parantamiseksi on vaikeaa, koska palvelimettomissa alustoissa koodista ei voi kerätä tehokkaasti statistiikkaa sen optimointia varten. Cold startin helpottaminen voi johtaa ratkaisuihin, jotka ovat vastoin palvelimettoman tietojenkäsittelyn ideaa. Helpottaminen voi johtaa turhaan monimutkaisuuteen, ja kontin jatkuvasti lämpimänä pitämisessä resursseja ei käytetä vain tarpeen mukaan.

Cold starttia voidaan helpottaa monella tavalla. Lee, Yoon, Yeo & Oh esittävät ratkaisuksi automatisoidun funktioiden fuusioinnin. Sen algoritmi arvioi onko fuusiointi kannattavaa, koska joskus siitä voi koitua enemmän latenssia. Vahidinia, Farahani & Aliee esittävät ratkaisun, jossa konttia ei pidetä lämpimänä tietyn kiinteän ajan verran, vaan sille lasketaan paras aika. Algoritmi oppii tämän parhaan ajan tarkkailemalla menneiden funktiokutsujen aikavälejä. LambdaLitellä voidaan helpottaa cold starttia optimoimalla koodia. Koodin määrää pienennetään, ja koodi, joka ei ole välttämätöntä pakataan kevyiksi tiedostoiksi, ja ladataan käyttöön vasta tarvittaessa. Ignitellä optimoidaan funktiokoodia runtimessa. Koodin ylimääräistä profilointia ja käännöstä vähennetään ja optimoitua koodia generoidaan tietyille funktioille, jolloin niiden kutsut ovat ”kuumia”. Igniten avulla funktiot voidaan suorittaa täysin optimoidulla koodilla alusta lähtien. HotC:ssä runtimea käytetään uudelleen, joka tapahtuu sitä mukaan, kun asiakkailta tulee pyyntöjä. Sovellukselle etsitään olemassa oleva

kontti, jolla on sille sopiva runtime. Uudelleen käytettävät vapaat kontit ovat runtime-altaassa, joka on kevyttä eikä siitä tule ylimääräistä rasitetta. HotC:ssä resurssien kulutus tulee pääasiassa sovellusten suorittamisesta. Nukassa latenssia vähennetään monella pienellä tavalla, sen sijaan että käytössä olisi yksi isompi tapa. Latenssia vähennetään uudelleen käyttämällä namespaceä ja cgroupia, pakkausvälimuisti välttää imagen vetämisen, paljon käytetyt pakkaukset asennetaan worker-koneille pysyvästi ja muut pakkaukset menevät välimuistiin, ja taukokontin daemon-tarkistaja varmistaa puhtaan namespace-ympäristön, jolloin prosessiryhmää ei häiritä. Nukassa pyyntökuormia on kahdenlaisia, vakaat ja purkautuvat kuormat. Vakaissa kuormissa kontin taukoa pidetään päällä funktion suorittamisen jälkeen, ja jatketaan seuraavassa suorituksessa. Purkautuvassa kuormassa kontteja skaalataan automattisesti.

Verrattuna uusimpiin teknologioihin, saavuttaa LambdaLite 21,25-kertaisen parannuksen kokonaisvastauslatenssiin. Ignite pystyi vähentämään kokeissa kuluja ja latenssia 1,26-5,5-kertaisesti. HotC:ssä sovelluksen suoritusaikaa saatiin vähennettyä keskimäärin 28,55 %:ia. Pyyntöpurkauksissa HotC pystyi parhaimmillaan vähentämään latenssia jopa 73 %. Tämä tosin saavutetaan vasta kun pyyntöpurkauksia on tullut enemmän, jolloin myöhemmässä vaiheessa sopivia kontteja on ehtynyt syntyä jo enemmän. Nukalle tehtyjen kokeiden mukaan, jos kontteja käynnistetään samanaikaisesti, ja niitä on vähemmän kuin taukokontteja, ei käynnistyslatenssi kasva suhteellisesti. Imagen vetämisen ja pakkausten tuomisen latenssia pystyttiin vähentämään keskimäärin sadoista millisekunneista noin 20:een millisekuntiin. Vakaissa kuormissa CPU:n hyödyntämistä saatiin nostettua keskimäärin 30% ja purkautuvissa kuormissa alustamislatenssia voitiin vähentää 50:een millisekuntiin, kun samanaikaisuuksia oli 256. Verrattuna Dockeriin, cold start vähentyi kuusinkertaisesti, samanaikaisuuskokeessa latenssi väheni 5-10-kertaisesti, ja suoritusteho kohoni 3-20-kertaiseksi. Verrattuna SOCK:iin, samanaikaisuudessa latenssi pienentyi 3-5-kertaisesti, ja suoritusteho kohosi kolminkertaisesti.

4.2.2 Funktioiden tiladatattomuuden ongelmia ja PraaS-ratkaisu

Tiladatattomat funktiot tuottavat ongelmia palvelimettomassa tietojenkäsittelyssä. Monet sovellukset vaativat tiladatan, jolloin se täytyy tallentaa hitaaseen ulkoiseen datavarastoon. Funktioiden ja näiden datavarastojen välinen yhteistyö on haavoittuvaista virhetilanteille. Tiladatattomuus myös tuottaa latenssia, vaikka funktio suoritettaisiin lämpimässä kontissa. AWS:n S3, joka on ulkoinen datavarasto, ei sisällä kansiorakennetta, kustomointimahdollisuutta eikä sillä ole monikäyttäjätietoturvallisuutta. Arkaluontoisen datan tallentaminen ulkoisiin varastoihin voi olla tietoturvariski.

PraaS:sa ratkaistaan tämä ongelma tallentamalla tiladata paikalliseen muistiin ulkoisen datavaraston sijasta. Ratkaisussa käytetään sessiota, ja funktiot, jotka ovat saman session sisällä voivat kommunikoida keskenään. PraaS käyttää p2p-kommunikaatiota saadakseen nopeamman

siirtonopeuden. Sessiot voivat olla yhteydessä toisiinsa globaalin muistin kanssa. Kokeessa verrattiin PraaS:ia AWS:sään. PraaS:in session muistia käyttämällä saatiin kaksi kertaluokkaa nopeampi tulos kuin jos kommunikointi olisi tehty AWS:n S3-datavarastosta. Verrattuna lambda-funktioon, on nopeus PraaS:in session muistissa kymmenenkertainen.

4.2.3 Bugien korjauksen, testauksen ja monitoroinnin ongelmia ja testauksen ratkaisuja

Bugien korjaamista ei ole vielä kehitetty kunnolla palvelimettomalle tietojenkäsittelylle. Teknologia onkin vielä kypsymätön. Kehittäjillä ei ole pääsyä palvelinpuolelle, joka tuottaa ongelmia bugien korjaamisessa. Bugeja täytyy korjata epäsuorin keinoin, jossa täytyy tehdä päätelmiä. Korjaamiseen voidaan käyttää dataa, jota kerätään funktiosta sen suoritusajalta. Kehittäjille täytyy myös lähettää logit funktioiden kutsuista ja tarjota tarkat stack tracet. Testausta varten on vaikeaa mallintaa aitoa pilviympäristöä paikallisiin koneisiin. Vaikeus johtuu siitä, ettei voida tietää etukäteen mitä konttia tullaan käyttämään käyttöönotossa. Suurin osa ympäristöriippuvuuksista on myös saatavilla vain ajon aikana. Testaus on myös paikallisesti erilaista, koska pilvipalvelun tarjoajat hallitsevat resursseja. Perinteisillä testaustyökaluilla on oletuksena pääsy palvelimelle, jolloin niitä ei voi käyttää palvelimettomissa alustoissa. Monitorointityökaluja ei ole vielä kunnolla kehitetty palvelimettomalle tietojenkäsittelylle.

Ratkaisuna testauksen ongelmaan, on markkinoilla paljon testaustyökaluja, joilla pystytään simuloimaan isoa määrää käyttäjien pyyntöjä. Työkalujen lisäksi myös testausstrategia on tärkeä. Koneoppiminen, syväoppiminen ja NPL voivat myös auttaa testauksessa.

4.2.4 Tietoturvaongelmia ja ratkaisuja niihin

Palvelimeton tietojenkäsittely on vielä uusi teknologia, joten suojausta ei ole vielä kunnolla kehitetty. Samat asiat, jotka tuottavat palvelimettomassa tietojenkäsittelyssä latenssia ja bugien korjaamisen vaikeutta, aiheuttavat myös tietoturvaongelmia. Funktioiden tiladatattomuus aiheuttaa tietoturvaongelmia, koska hyökkäys pinta-alasta tulee laajempi. Funktioiden on myös jatkuvasti oltava vuorovaikutuksessa ulospäin, jolloin on haasteellista määritellä millä funktiolla on pääsy mihinkin. Tietoturvan kehittäminen vaikeutuu koska kehittäjillä ei ole pääsyä palvelinpuolelle. Pilvipalvelun tarjoajat pitävät myös tyypillisesti suurimman osan infrastruktuurin informaation salassa, jolloin on vaikeaa tarkastella palvelun tietoturvaa ja yksityisyydensuojaa. Hyökkäystapoja on mm. toistohyökkäys ja yksityisyydensuojahyökkäys, joka tehdään päättelemällä saatavilla olevasta datasta. Kunnollista auktorisointia ei ole vielä kehitetty, ja vankan autentikointimallin toteuttaminen voi olla monimutkainen hanke.

Sovelluksen tietoturvallinen arkkitehtuuri kannattaa erotella selkeästi kolmikerroksiseksi, jolloin on helpompi räätälöidä tietoturva jokaiselle tasolle. Rajapintojen ja vuorovaikutuksen ollessa selkeää

kerroksien välillä, voidaan epäilyttävä toiminta havaita helpommin. Tietoturva-alueet täytyy erotella kerrosten väliin. AWS:sä IAM-roolien ja -käyttöoikeuksien täytyy olla näkyviä, koska et voi kontrolloida sitä mitä et näe. Funktioiden on käytettävä tietoturvaa varten pienimmän oikeuden mallia. Kubernetesissä ensimmäinen kerros rajoittaa pääsyä ja toimintojen suorittamista. API-kommunikaatio on salattu TLS:llä, ja kaikki APIn asiakkaat täytyy olla autentikoituja, jopa nekin, jotka ovat osa infrastruktuuria kuten solmut, välityspalvelimet ja aikatauluttajat. API päätepisteeseen täytyy olla pääsy vain verkon sisällä, johon Kubernetes-rypäs on otettu käyttöön. API täytyy ottaa käyttöön korkealla saatavuudella ja vikasietoisella käyttöönotolla, jotta rypästä voidaan hallinnoida myös hyökkäyksen aikana ja palvelun laatu voidaan taata. Konfiguraatio ja tiladata varastoidaan ETCD:hen joka on yleinen avainarvovarasto. Kaikki salainen tieto täytyy salata ETCD:ssä, koska hyökkääjä saattaa saada vihjeitä murrolle vain lukemalla ETCD:n sisältöä.

4.3 Tulosten yhteenveto

Cold start oli selkeästi yleisin ongelma palvelimettomassa tietojenkäsittelyssä. Sen pituus suhteessa funktioiden lyhyeen suoritusaikaan on merkittävä ongelma. Tiladatattomat funktiot aiheuttavat latenssia, vaikka funktion kutsu tapahtuisi warm startissa. Bugien korjaamisessa, testaamisessa ja monitoroinnissa isona ongelmana on se, ettei kehittäjillä ole pääsyä palvelinpuolelle. Tietoturva on vielä puutteellinen palvelimettoman tietojenkäsittelyn kypsymättömyydestä. Palvelimettomassa tietojenkäsittelyssä moni ongelma johtuu samoista asioista. Esimerkiksi se, ettei kehittäjillä ole pääsyä palvelinpuolelle, aiheuttaa ongelmia sekä bugien korjaamisessa ja testaamisessa, kuin myös tietoturvassa. Funktioiden tiladatattomuus myös vaikuttaa latenssin lisäksi tietoturvaan.

Cold startille on useita ratkaisuja. Funktioita voidaan fuusioida, kontti voidaan pitää lämpimänä vain tarvitun ajan, koodia voidaan optimoida ja sovellukselle voidaan uudelleen käyttää sille sopivaa runtimea. Cold starttia voidaan myös helpottaa monella pienemmällä tavalla, sen sijaan että käytettäisiin yhtä isompaa ideaa. Ratkaisuilla voidaan vähentää latenssia huomattavasti. PraaS:lla tiladata voidaan tallentaa paikalliseen muistiin ulkoisen datavaraston sijasta. Testauksessa voidaan simuloida isoa määrä käyttäjien pyyntöjä markkinoilla olevien monien testaustyökalujen avulla. Tietoturvaa voidaan parantaa mm. AWS:n IAM-rooleilla, Kubernetesin tietoturvaominaisuuksilla, ja kolmikerroksisella arkkitehtuurilla, jossa kerrokset on selkeästi eroteltu toisistaan.

5 Pohdinta

5.1 Tulosten yhteenveto ja tarkastelu

Päätuloksena oli tietoa siitä mitkä ovat yleisimmät ongelmat palvelimettomassa tietojenkäsittelyssä. Tietoa tuli myös siitä, minkälaisia nämä ongelmat ovat ja mistä ne johtuvat, sekä minkälaisia ratkaisuja näihin ongelmiin on.

Tutkittavaa aineistoa oli 67, joista 32:ssa kirjoitettiin cold startista, tehden siitä yleisimmän ongelman. Kolme muuta yleisintä ongelmaa olivat funktioiden tiladatattomuus, bugien korjaus, testaus ja monitorointi, sekä tietoturva. Ongelmat ovatkin keskittyneet eniten muutamiin osa-alueisiin. Palvelimettoman tietojenkäsittelyn yksi pääideoista on suoritusajaltaan lyhyet funktiot, jolloin cold startin pitkä latenssi suhteessa niihin on merkittävä ongelma. Jotta tämä teknologia voisi käyttää potentiaalinsa mahdollisimman hyvin, on cold start-ongelmaan tartuttava kiinni. Ongelman ratkaisussa voi myös päätyä tilanteeseen, jossa ollaan vastoin palvelimetonta tietojenkäsittelyä. Näin ongelmaa onkin ratkaistava tarkoin. Tällä hetkellä tiladatalliset funktiot luodaan kiertoteitse, joka aiheuttaa latenssia ja on haavoittuvaista virhetilanteille. Bugien korjaamista ei ole vielä kunnolla kehitetty, joten kehittäjien täytyy korjata bugit epäsuorin keinoin päättelyä käyttämällä. Testauksessa on ongelmana aidon pilviympäristön mallintamine paikallisille koneille, ja testaus paikallisesti on myös erilaista verrattuna tuotantoon. Funktioiden tiladatattomuus ja se ettei kehittäjillä ole pääsyä palvelinpuolelle, tuottaa tietoturvaongelmia. Kaiken kaikkiaan, palvelimettoman tietojenkäsittelyn ollessa vielä uusi, ei perinteisiä tapoja voi aina käyttää ongelmien korjaamiseen.

Cold starttiin on tarjottu monia eri ratkaisuja. Näitä on mm. funktioiden fuusioiminen, kontin pitäminen lämpimänä oikean ajan, optimoimalla koodia pienentämällä sitä, optimoimalla koodia runtimessa, käyttämällä sopivaa runtimea uudelleen, ja monen pienemmän ratkaisun yhdistäminen. Kokeissa näillä ratkaisuille pystyttiin vähentämään cold startin latenssia jopa huomattavasti. Tiladatallisen funktion luomiseen on kehitetty ratkaisu, jossa tiladata tallennetaan paikalliseen muistiin ulkoisen datavaraston sijasta. Testausta varten voidaan simuloida isoa määrää käyttäjien pyyntöjä useilla markkinoilla olevilla testaustyökaluilla. Tietoturvaa voidaan parantaa kolmikerroksisella arkkitehtuurilla. Kaikista monista palvelimettoman tietojenkäsittelyn ongelmista huolimatta, on ratkaisuja kehitetty paljon.

Jotkin asiat palvelimettomassa tietojenkäsittelyssä johtavat moniin ongelmiin. Esim. se, ettei kehittäjillä ole pääsyä palvelinpuolelle, tuottaa ongelmia tietoturvassa, sekä-että bugien korjaamisessa ja testauksessa. Funktioiden tiladatattomuus ei aiheuta pelkästään latenssia, vaan myös tietoturvaongelmia. Suljetun palvelimen lisäksi muita rajoituksia on vaikeus kerätä statistiikkaa koodista, jolloin koodia voitaisiin optimoida cold starttia varten. Palvelimeton tietojenkäsittely on teknologiana

vielä kypsymätön, joka näkyy mm. siinä, että bugien korjaus ja tiladatallisuus funktiolle täytyy tehdä kiertoteitse. Myöskään tietoturvaa ei ole vielä kunnolla kehitetty.

Palvelimettoman tietojenkäsittelyn kehittäjät saavat tämän tutkimuksen avulla tietoa joistakin ratkaisuista yleisimpiin ongelmiin. Kehittäjät saavat myös tietoa eri syistä, mistä yleisimmät ongelmat johtuvat. Tätä tuotettua tietoa voidaan hyödyntää ja soveltaa uusien ratkaisujen kehittämisessä.

Tutkimuksessa tutkittiin yleisimpiä ongelmia määrällisesti. Tutkimusta voisi jatkokehittää niin, että voitaisiin myös tutkia määrällisesti yleisimmistä ongelmista sen, mitkä ovat yleisimmät syyt niiden aiheuttajille. Ratkaisuista voitaisiin tutkia laadullisesti mitkä ovat kaikkein tehokkaimmat ratkaisut.

5.2 Johtopäätökset

Koska ongelmat ovat keskittyneet eniten neljään eri osa-alueeseen, niin ratkaisemalla näitä yleisiä ongelmia saadaan paljon aikaiseksi. Myös yhden ongelman tai puutteen hoitaminen voi ratkaista monta asiaa. Esim. jos kehittäjillä olisi pääsy palvelinpuolelle, voitaisiin sillä ratkaista tietoturvaan sekä myös bugien korjaamiseen, testaukseen ja monitorointiin liittyviä ongelmia. Tiladatattomien funktioiden ongelman ratkaisemisessa saataisiin parannettua latenssia, sekä-että tietoturvaa. Ratkaisuja alkaa olemaan jo aika hyvin, mutta tietoisuus ei luultavasti niistä ole vielä kovin suurta. Cold startin helpottamiseksi tehty ratkaisu Nuka, on yhdistelmä pienempiä ratkaisuja sen sijaan että siinä olisi yksi iso pääidea. Tämä voi myös olla yksi lähestymistapa ongelmiin, jossa monella pienellä asialla saadaan isompi tulos aikaiseksi. Tosin, kun lähdetään puuttumaan palvelimettoman tietojenkäsittelyn moniin eri osa-alueiden ongelmiin, vaatii se yksittäisten ratkaisujen käyttöä, joka voi aiheuttaa tarpeetonta monimutkaisuutta. Olisikin hyvä, jos tulevaisuudessa nämä yksittäiset ratkaisut saataisiin osaksi palvelimettomia alustoja.

5.3 Tutkimuksen luotettavuus ja eettisyys

Tutkimuksen luotettavuudessa täytyy ottaa huomioon, että palvelimettoman tietojenkäsittelyn ongelmien yleisyys tehtiin sen perusteella, kuinka paljon niistä on puhuttu muissa tutkimuksissa. Tämän tutkimuksen näkökulma siitä, mitkä ovat yleisimmät ongelmat, onkin varsin akateeminen näkemys.

Määrällisessä tutkimuksessa on myös epätarkkuutta. Esimerkiksi ongelma ”function communication” voi myös olla tiladataton ongelma, koska palvelimettomassa tietojenkäsittelyssä ei ole suoranaisesti tiladataa funktioille. Myös ongelmat ”performance” ja ”latency” saattavat viitata cold starttiin, tai tiladatattomuudesta johtuvaan latenssiin. Tietoturva -, bugien korjaus -, testaus -ja monitorointi ongelmat johtuvat mm. siitä, ettei kehittäjillä ole pääsyä palvelinpuolelle, jolloin nämä ongelmat voitaisiin myös laskea ongelmaan ”users can’t control the server” (liite 1).

En ollut mitenkään esteellinen tutkimusta tehdessäni. Minulla ei ollut mitään yhteyksiä palvelimettomiin alustoihin tai kehittäjiin tutkimusta tehdessäni, eikä tutkimukselle ollut rahoitusta tai sidonnaisuuksia. Opinnäytetyöni aihe oli omalta alaltani, johon opiskelen. Aloitin työn tekemisen aivan opiskelujeni loppupuolella, joten tietojenkäsittelystä minulla oli jo riittävästi tietoa aiheen tutkintaa varten. Resurssit työhön olivat riittävät, vaikka aikataulultaan tiukat. En käsitellyt kenenkään henkilötietoja tutkimusta tehdessäni, eikä tässä työssä ole salassa pidettävää tietoa tai aineistoa. Työlle ei ollut toimeksiantajaa eikä yhteistyökumppaneja, enkä tarvinnut eettistä ennakkoarviointia tai tutkimuslupaa työlleni. Olen kertonut lähteet kaikkiin tutkimuksiin, joihin ollut viitannut omassa tekstissäni.

5.4 Opinnäytetyöprosessin ja oman oppimisen arviointi

Työn tavoitteet olivat tutkia määrällisesti mitkä olivat palvelimettoman tietojenkäsittelyn yleisimmät ongelmat. Tavoitteena oli myös tutkia laadullisesti, minkälaisia nämä ongelmat olivat, ja minkälaisia ratkaisuja niihin on olemassa. Saavutin tavoitteeni määrällisesti hyvin. Tutkimusainestoa määrälliseen tutkimukseen oli 67 tutkimusta, joka oli hyvä määrä. Laadullisen tutkimuksen olisin voinut tehdä paremmin niin, että olisin etsinyt kaikkein painavimmat syyt ongelmille, ja kaikkein tehokkaimmat ratkaisut niille. Onnistuin määrällisen tutkimuksen aineiston keruussa. Ainestoa oli paljon (67 tutkimusta), verrattuna siihen, että kyse oli tutkimuksista.

Opin hyvin paljon tietojenkäsittelystä. En oppinut pelkästään palvelimettomasta tietojenkäsittelystä, vaan myös ohjelmistoarkkitehtuurista yleensä. Tiladatallisten funktioiden luominen kiertoteitse antoi lisää ymmärrystä, kuinka sovelluskehitys toimii. Kyse on kuitenkin ongelmien ratkaisusta loppujen lopuksi. Tietokantoihin voi yhtä hyvin tallentaa välillistä dataa, kuin ”oikeaa” dataa. Vaikkei tämä olisikaan mikään paras ratkaisu, on tässä hoidettu jonkinlainen ratkaisu funktioiden tiladatattomuuteen. Muutenkin, jos on aivan pakko tehdä väliaikaisia ”purkkaratkaisuja”, on kehittämisessä olemassa erilaisia kiertoteitä.

Tein opinnäytetyöni kovassa kiireessä, joten jouduin tinkimään laadusta. Omaan ohjaajaani olisin voinut olla enemmän yhteydessä. Kaikkia lähdeluettelon merkintöjä ja tekstiviitteitä en saanut oikein tehtyä, mutta opin ainakin kirjoittamaan niitä nopeasti.

Lähteet

Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H., Jin, X. & Liu, X. 2022. LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing. Cornell University. Ithaca. https://doi.org/10.48550/arXiv.2207.08175. Luettu: 18.11.2022.

Shafiei, H., Khonsari, A. & Mousavi, P. 18.2.2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. ACM. Elektroninen tietoaineisto. https://doi.org/10.1145/3510611. Luettu: 18.11.2022.

Vahidinia, P., Farahani, B. & Shams Aliee, F. 2020. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. IEEE. Luettu: 18.11.2022.

Zhang, H., Cardoza, A., Baile Chen, P., Angel, S. & Liu, V. 2020. Fault-tolerant and transactional stateful serverless workflows. USENIX Association. Berkeley. https://dl.acm.org/doi/pdf/10.5555/3488766.3488833. Luettu: 18.11.2022.

Kuhlenkamp, J., Werner, S. & Tai, S. 2020. The Ifs and Buts of Less is More: A Serverless Computing Reality Check. IEEE. https://doi.ieeecomputersociety.org/10.1109/IC2E48712.2020.00023. Luettu: 18.11.2022.

Amazon. AWS Lambda Features. https://aws.amazon.com/lambda/features/. Luettu: 18.11.2022.

Dantas, J., Khazaei, H. & Litoiu, M. 2022. Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda. York University. Toronto. Luettavissa: https://www.researchgate.net/publication/362166735_Application_Deployment_Strategies_for_Reducing_the_Cold_Start_Delay_of_AWS_Lambda. Luettu 13.10.2022.

Shin, W., Kim, W. & Min, C. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot. EuroSys ’22. Rennes. Luettavissa: https://dl.acm.org/doi/10.1145/3492321.3519581. Luettu: 13.10.2022.

Pan, L., Wang, L., Chen, S. & Liu, F. 2022. Retention-Aware Container Caching for Serverless Edge Computing. IEEE INFOCOM 2022. Luettavissa: https://ieeexplore.ieee.org/document/9796705. Luettu: 13.10.2022.

Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q. & Chen, H. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. ASPLOS’20. Luettavissa: https://ipads.se.sjtu.edu.cn/_media/publications/catalyzer-asplos20.pdf. Luettu: 13.10.2022.

Ustigov, D., Amariucai, T. & Grot, B. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. IEEE International Symposium on Workload Characterization (IISWC). Luettavissa: https://ieeexplore.ieee.org/document/9668286. Luettu: 13.10.2022.

Mikhail Shilkov 2021. Cold Starts in AWS Lambda. Luettavissa: https://mikhail.io/serverless/coldstarts/aws/. Luettu 19.10.2022.

Mikhail Shilkov 2021. Cold Starts in Azure Functions. Luettavissa: https://mikhail.io/serverless/coldstarts/azure/. Luettu: 19.10.2022.

Li, Y., Lin, Y., Wang, Y., Ye, K. & Xu, C., 2022. Serverless Computing: State-of-the-Art, Challenges and Opportunities. IEEE. Luettavissa: https://doi.org/10.1109/TSC.2022.3166553. Luettu: 20.10.2022.

Xu, Z., Zhang, H., Geng, X., Wu, Q. & Ma, H., 2019. Adaptive Function Launching Acceleration in Serverless Computing Platforms. IEEE 25th International Conference on Parallel and Distributed Systems. Tianjin. Luettavissa: https://ieeexplore.ieee.org/document/8975850. Luettu: 20.10.2022.

Shahrad, M., Fonseca, R., Goiri, Í., Chaudry, G., Barum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M. & Bianchini, R., 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. 2020 USENIX Annual Technical Conference. Boston. Luettavissa: https://www.usenix.org/conference/atc20/presentation/shahrad. Luettu: 22.10.2022.

Klimovic, A., Wang, Y., Trivedi, A., Pfefferle, J. & Kozyrakis, C. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. OSDI ’18. Carlsbad. Luettavissa: https://www.usenix.org/conference/osdi18/presentation/klimovic. Luettu: 26.10.2022.

Li, Z., Guo, L., Cheng, J., Chen, Q., He, B. & Guo, M. 2022. The Serverless Computing Survey: A Technical Primer for Design Architecture. ACM Computing SurveysVolume 54Issue 10s. Luettavissa: https://doi.org/10.48550/arXiv.2112.12921. Luettu: 26.10.2022.

Basu Roy, R., Patel, T. & Tiwari, D. 2021. Characterizing and Mitigating the I/O Scalability Challenges for Serverless Applications. IEEE International Symposium on Workload Characterization (IISWC). Austin. Luettavissa: https://ieeexplore.ieee.org/document/9668299. Luettu: 27.10.2022.

Denninnart, C. & Amini Salehi, M. 2021. Efficiency in the Serverless Cloud Computing Paradigm: A Survey Study. arXiv:2110.06508v1. Ithaca. Luettavissa: https://arxiv.org/abs/2110.06508. Luettu: 27.10.2022.

Pu, Q., Venkataraman, S. & Stoica, I. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’19). Boston. Luettavissa: https://www.usenix.org/system/files/nsdi19-pu.pdf. Luettu: 27.10.2022.

Jindal, A., Gerndt, M., Chadha, M., Podolskiy, V. & Chen, P. 2021. Function Delivery Network: Extending Serverless Computing for Heterogeneous Platforms. IEEE 41st International Conference on Distributed Computing Systems (ICDCS). Luettavissa: https://ieeexplore.ieee.org/document/9546403. Luettu: 28.10.2022.

Chard, K. & Foster, I. 2021. Toward a definition for serverless computing. Serverless Computing: Report from Dagstuhl Seminar 21201. Schloss Dagstuhl -- Leibniz-Zentrum fur Informatik. Luettavissa: https://orca.cardiff.ac.uk/id/eprint/144409/. Luettu: 28.10.2022.

Wu, M., Mi, Z. & Xia, Y. 1.9.2020. A Survey on Serverless Computing and its Implications for Joint-Cloud Computing. 19913982. IEEE. Elektronin tietoaineisto. https://doi.org/10.1109/JCC49151.2020.00023. Luettu: 31.10.2022.

Lenarduzzi, V. & Panichella, A. 13.10.2020. Serverless Testing: Tool Vendors’ and Experts’ Points of View. 20321359. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/MS.2020.3030803. Luettu: 31.10.2022.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. & Suter, P. 10.6.2017. Serverless Computing: Current Trends and Open Problems. arXiv:1706.03178v1. Cornell University. Elektroninen tietoaineisto. https://doi.org/10.48550/arXiv.1706.03178. Luettu: 31.10.2022.

Hassan, H., Barakat, S. & Sarhan, Q. 12.7.2021. Survey on serverless computing. Journal Of Cloud Computing. Elektroninen tietoaineisto. https://doi.org/10.1186/s13677-021-00253-7. Luettu: 1.11.2022.

Castro, P., Ishakian, V., Muthusamy, V. & Slominski, A. 7.6.2019. The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry. arXiv:1906.02888v1. Cornell University. Elektroninen tietoaineisto. https://doi.org/10.48550/arXiv.1906.02888. Luettu: 1.11.2022.

Lin, W., Krintz, C., Wolski, R. & Zhang, M. 17.5.2018. Tracking Causal Order in AWS Lambda Applications. 17767624. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/IC2E.2018.00027. Luettu: 1.11.2022.

Wen, J., Chen, Z. & Liu, X. 27.7.2022. Software Engineering for Serverless Computing. arXiv:2207.13263v1. Cornell University. Elektroninen tietoaineisto. https://doi.org/10.48550/arXiv.2207.13263. Luettu: 2.11.2022.

Patnayakuni, R., Patnayakuni, N. 3.5.2019. Securing Serverless Computing. AIS eLibrary. Eletroninen tietoaineisto. https://aisel.aisnet.org/wisp2018/15/. Luettu: 5.11.2022.

Marin, E., Perino, D. & Di Pietro, R. 8.8.2021. Serverless Computing: A Security Perspective. arXiv:2107.03832v2. Cornell University. Elektroninen tietoaineisto. https://doi.org/10.48550/arXiv.2107.03832. Luettu: 5.11.2022.

Lee, S., Yoon, D., Yeo, S. & Oh, S. 2021. Mitigating Cold Start Problem in Serverless Computing with Function Fusion. MDPI. Basel. https://doi.org/10.3390/s21248416. Luettu: 7.11.2022.

Vahidinia, P., Farahani, B. & Aliee, F. 5.4.2022. Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/JIOT.2022.3165127. Luettu: 9.11.2022.

Carreira, J., Kohli, S., Bruno, R. & Fonseca, P. 3.6.2021. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. ACM. Elektroninen tietoaineisto. https://doi.org/10.1145/3458336.3465305. Luettu: 10.11.2022.

Qin, S., Wu, H., Wu, Y., Yan, B., Xu, Y. & Zhang, W. 1.9.2020. Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing. 19913981. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/JCC49151.2020.00021. Luettu: 12.11.2022.

Suo, K., Son, J., Cheng, D., Chen, W. & Baidya, S. 13.10.2021. Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing. 21226734. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/Cluster48925.2021.00018. Luettu: 11.11.2022.

Copik, M., Calotoiu, A., Bruno, R., & Böhringer, R. & Hoefler, T. 2022. Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes. Scalable Parallel Computing Lab. Zürich. http://spcl.ethz.ch/Publications/.pdf/2022_copik_praas_report.pdf. Luettu: 17.11.2022.

Khatri, D., Khatri, S. & Mishra, D. 15.9.2020. Potential Bottleneck and Measuring Performance of Serverless Computing: A Literature Study. 19988092. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/ICRITO48877.2020.9197837. Luettu: 14.11.2022.

Kaplunovich, A. 19.9.2019. ToLambda-Automatic Path to Serverless Architectures. 18994858. IEEE. Elektroninen tietoaineisto. https://doi.org/10.1109/IWoR.2019.00008. Luettu: 12.11.2022.

Chetal, A., Manral, V., Bregkou, M., Ferreira, R., Hadas, D., Murthy, V., Vasquez, E. & Wrobel, J. 2021. How to Design a Secure Serverless Architecture. Cloud Security Alliance. https://cloudsecurityalliance.org/artifacts/serverless-computing-security-in-2021/. Luettu: 16.11.2022.

Amazon. Manage IAM Permissions. https://aws.amazon.com/iam/features/manage-permissions/. Luettu: 16.11.2022.

Kubernetes. https://kubernetes.io/. Luettu: 16.11.2022.

O'Meara, W. & Lennon, R. 2020. Serverless Computing Security: Protecting Application Logic. IEEE. https://doi.org/10.1109/ISSC49989.2020.9180214. Luettu: 16.11.2022.

Liitteet

Liite 1. Kvantitatiivisen tutkimuksen tulokset

COLD START (32)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing: State-of-the-Art, Challenges and Opportunities

Survey on serverless computing

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

A Survey on Serverless Computing and its Implications for JointCloud Computing

Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda

COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms

Cold Start in Serverless Computing: Current Trends and Mitigation Strategies

Cold Start Influencing Factors in Function as a Service

ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot

FnSched: An Efficient Scheduler for Serverless Functions

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era

In Search of a Fast and Efficient Serverless DAG Engine

LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing

Mitigating Cold Start Problem in Serverless Computing with Function Fusion

Mitigating Cold Starts in Serverless Platforms

Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach

Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing

OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING

Optimizing and Extending Serverless Platforms: A Survey

Retention-Aware Container Caching for Serverless Edge Computing

SAND: Towards High-Performance Serverless Computing

Sequoia: Enabling !ality-of-Service in Serverless Computing

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

Serverless Performance and Optimization Strategies

Serverless Computing: A Security Perspective

Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing

WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing

Software Engineering for Serverless Computing

STATELESS (14)

Serverless Computing Current Trends and

Survey on serverless computing

The Serverless Computing Survey: A Technical Primer for Design Architecture

Characterizing and Mitigating the I/O Scalability Challenges for Serverless Applications

Efficiency in the Serverless Cloud Computing Paradigm: A Survey Study

Exploiting Serverless Runtimes for Large-Scale Optimization

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

Harnessing the Power of Serverless Runtimes for Large-Scale Optimization

Narrowing the Gap Between Serverless and its State with Storage Functions

Optimizing and Extending Serverless Platforms: A Survey

Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Function delivery network: Extending serverless computing for heterogeneous platforms

SECURITY (13)

Serverless Computing A Survey of Opportunities Challenges and

Serverless Computing: State-of-the-Art, Challenges and Opportunities

Survey on serverless computing

A Survey on Serverless Computing and its Implications for JointCloud Computing

Compiler-Assisted Semantic-Aware Encryption for Efficient and Secure Serverless Computing

Efficiency in the Serverless Cloud Computing Paradigm: A Survey Study

How to Design a Secure Serverless Architecture

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

Serverless Computing Security: Protecting Application Logic

Serverless Computing: A Security Perspective

Software Engineering for Serverless Computing

Status of Serverless Computingand Function-as-a-Service (FaaS) inIndustry and Research

DEBUGGING / MONITORING / TESTING (13)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing A Survey of Opportunities Challenges and

Serverless Computing Current Trends and

(2021) Survey on serverless computing

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Tracing Function Dependencies Across Cloud

Software Engineering for Serverless Computing

Status of Serverless Computingand Function-as-a-Service (FaaS) inIndustry and Research

Serverless Testing: Tool Vendors' and Experts' Points of View

Survey on serverless computing

Software Engineering for Serverless Computing

SCHEDULING (9)

Serverless Computing: State-of-the-Art, Challenges and Opportunities

Survey on serverless computing

Automated Fine-Grained CPU Cap Control in Serverless Computing Platform

FnSched: An Efficient Scheduler for Serverless Functions

Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit

Practical Scheduling for Real-World Serverless Computing

Sequoia: Enabling !ality-of-Service in Serverless Computing

Function delivery network: Extending serverless computing for heterogeneous platforms

Serverless Computing

COST (8)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing Current Trends and

Survey on serverless computing

Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement

FnSched: An Efficient Scheduler for Serverless Functions

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

Software Engineering for Serverless Computing

Status of Serverless Computingand Function-as-a-Service (FaaS) inIndustry and Research

PERFORMANCE (8)

Serverless Computing Current Trends and

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

A Survey on Serverless Computing and its Implications for JointCloud Computing

Caching Techniques to Improve Latency in Serverless Architectures

Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit

Serverless Computing

FUNCTION COMMUNICATION (7)

Serverless Computing: State-of-the-Art, Challenges and Opportunities

A Survey on Serverless Computing and its Implications for JointCloud Computing

Optimizing Stateful Serverless Computing

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Optimizing and Extending Serverless Platforms: A Survey

Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

SAND: Towards High-Performance Serverless Computing

CONCURRENCY (7)

Survey on serverless computing

The Serverless Computing Survey: A Technical Primer for Design Architecture

Formal Foundations of Serverless Computing

Formal Foundations of Serverless Computing

Kappa: A Programming Framework for Serverless Computing

Optimizing and Extending Serverless Platforms: A Survey

Sequoia: Enabling !ality-of-Service in Serverless Computing

SCALING (5)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing Current Trends and

Survey on serverless computing

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

LATENCY (5)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Caching Techniques to Improve Latency in Serverless Architectures

Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Status of Serverless Computingand Function-as-a-Service (FaaS) inIndustry and Research

USERS CAN'T CONTROL THE SERVER (5)

Optimizing Stateful Serverless Computing

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing

Serverless Computing

DEVELOPER TOOLS (4)

Serverless Computing A Survey of Opportunities Challenges and

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

Securing Serverless Computing

Tracing Function Dependencies Across Cloud

VENDOR LOCK-IN (4)

Survey on serverless computing

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

Software Engineering for Serverless Computing

EPHEMERAL (4)

Formal Foundations of Serverless Computing

Harnessing the Power of Serverless Runtimes for Large-Scale Optimization

OverSketched Newton: Fast Convex Optimization for Serverless Systems

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

CACHING (3)

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing A Survey of Opportunities Challenges and

INFINICACHE: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

HARDWARE (3)

Serverless Computing Current Trends and

Survey on serverless computing

Optimizing Stateful Serverless Computing

FAULT TOLERANCE (3)

Serverless Computing: State-of-the-Art, Challenges and Opportunities

Survey on serverless computing

Serverless Workflows with Durable Functions and Netherite

LACK OR QUALITY OF SERVICE (QOS) SUPPORT (3)

Survey on serverless computing

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

In Search of a Fast and Efficient Serverless DAG Engine

CONTAINERS (3)

The Serverless Computing Survey: A Technical Primer for Design Architecture

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Software Engineering for Serverless Computing

WORKFLOW (3)

Modeling and Optimization of Performance and Cost of Serverless Applications

Serverless Workflows with Durable Functions and Netherite

Triggerflow: Trigger-based orchestration of serverless workflows

INCONSISTENT PERFORMANCE BETWEEN CLOUD PROVIDERS

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

PORTABILITY

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

WARM QUEUE

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

QUALITY OF CODE

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

PERFORMANCE ANALYSIS

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

NETWORKING

Serverless Computing A Survey of Opportunities Challenges and

PACKING

Serverless Computing A Survey of Opportunities Challenges and

IDE

Serverless Computing Current Trends and

LONG RUNNING (2)

Serverless Computing Current Trends and

Survey on serverless computing

DATA MODEL (2)

Serverless Computing Current Trends and

Function delivery network: Extending serverless computing for heterogeneous platforms

OPEN SOURCE

Serverless Computing Current Trends and

NETWORK

Serverless Computing Current Trends and

RESOURCE LIMITS

Survey on serverless computing

FUNCTION COMPOSITION

Survey on serverless computing

RESOURCE SHARING

Survey on serverless computing

NAMING AND ADDRESSING SYSTEM

Survey on serverless computing

LEGACY SYSTEMS

Survey on serverless computing

MANAGING HYBRID CLOUDA

Survey on serverless computing

ARCHITECTURE (2)

Survey on serverless computing

Automated Fine-Grained CPU Cap Control in Serverless Computing Platform

BENCHMARKING

Survey on serverless computing

LACK OF STANDRADS (2)

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

Software Engineering for Serverless Computing

LEGACY CODE

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

EDGE COMPUTING (2)

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

Optimized container scheduling for data-intensive serverless edge computing

API AND BENCHMARK LOCK-IN WITHIN COORDINATION LAYER

The Serverless Computing Survey: A Technical Primer for Design Architecture

DE-PROVISIONING

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

OPERATIONAL TASKS

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

DEPENDENCY

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

CO-LOCATING DATA AND COMPUTATION

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

MIGRATION

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

Software Engineering for Serverless Computing

JOINT COMPUTING

A Survey on Serverless Computing and its Implications for JointCloud Computing

WORKER

Automated Fine-Grained CPU Cap Control in Serverless Computing Platform

I/O

Characterizing and Mitigating the I/O Scalability Challenges for Serverless Applications

FUNCTION ALLOCATION

Efficiency in the Serverless Cloud Computing Paradigm: A Survey Study

DIVERSE APPLICATIONS (2)

ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

FnSched: An Efficient Scheduler for Serverless Functions

DESIGNED FOR EVENT-DRIVE

Exploiting Serverless Runtimes for Large-Scale Optimization

JIT COMPILING

Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot

FUNCTION ORCHESTRATION

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

NO BEST PRACTICES

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

DEVELOPMENT (2)

Formal Foundations of Serverless Computing

Kappa: A Programming Framework for Serverless Computing

DATA LOSS FROM TERMINATED FUNCTIONS

Formal Foundations of Serverless Computing

INTERACTING WITH EXTERNAL SERVICES

Formal Foundations of Serverless Computing

AT-LEAST ONCE EXECUTION

Formal Foundations of Serverless Computing

IDEMPOTENT FUNCTIONS

Formal Foundations of Serverless Computing

JSON

Formal Foundations of Serverless Computing

RUNTIME OPTIMIZATION

From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era

LIMITS OF CLOUD RESOURCES

In Search of a Fast and Efficient Serverless DAG Engine

FUNCTION/EXECUTION TIME LIMIT (2)

Kappa: A Programming Framework for Serverless Computing

Mitigating Cold Start Problem in Serverless Computing with Function Fusion

DUPLICATE DATA (2)

Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit

OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING

PERFORMANCE MODELLING

Modeling and Optimization of Performance and Cost of Serverless Applications

DUPLICATION IN SERVERLESS

OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING

LIMITED FUNCTION LIFETIME (2)

Optimizing Stateful Serverless Computing

Optimizing and Extending Serverless Platforms: A Survey

COMMUNICATION (2)

Optimizing Stateful Serverless Computing

Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

ISOLATION (3)

Optimizing Stateful Serverless Computing

Securing Serverless Computing: Challenges, Solutions, and Opportunities

Serverless Computing

PERFORMANCE PREDICTING

Optimizing Stateful Serverless Computing

HIGH BANDWIDTH USAGE

Optimizing and Extending Serverless Platforms: A Survey

SYSTEM NOISE / STRAGGLERS

OverSketched Newton: Fast Convex Optimization for Serverless Systems

FUNCTION INTERACTION (2)

SAND: Towards High-Performance Serverless Computing

Serverless Performance and Optimization Strategies

LOCAL / GLOBAL MESSAGE COORDINATION

SAND: Towards High-Performance Serverless Computing

MANAGEMENT PRACTICES

Sequoia: Enabling !ality-of-Service in Serverless Computing

RESOURCE ALLOCATION (2)

Sequoia: Enabling !ality-of-Service in Serverless Computing

Software Engineering for Serverless Computing

NEW TECHNOLOGY

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

EXECUTION LIMIT

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

LIMITED DB CONFIGURATION

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

MONITORING

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

STATEFUL

Serverless Workflows with Durable Functions and Netherite

SYNCHRONIZATION

Serverless Workflows with Durable Functions and Netherite

FUNCTION CALL LIMIT

Serverless Performance and Optimization Strategies

“POLLUTING” CODE

Transactions across serverless functions leveraging stateful dataflows

TRANSACTIONAL

Transactions across serverless functions leveraging stateful dataflows

TRIGGER

Triggerflow: Trigger-based orchestration of serverless workflows

FUNCTION DOUBLE EXECUTION

Function delivery network: Extending serverless computing for heterogeneous platforms

APPLICATION IMPLEMENTATION

Software Engineering for Serverless Computing

CONFIGURATION

Software Engineering for Serverless Computing

EVALUATION REPRODUCIBLE (2)

Software Engineering for Serverless Computing

Serverless Computing

PERFORMANCE OF VMs

Software Engineering for Serverless Computing

FUNCTION PARTITION

Serverless Computing

WHICH APPLICATION AND DOMAIN TO CHOOSE

Liite 2. Tutkimusaineisto

Potential_Bottleneck_and_Measuring_Performance_of_Serverless_Computing_A_Literature_Study

Serverless Computing: State-of-the-Art, Challenges and Opportunities

Survey on serverless computing

The server is dead, long live the server: Riseof Serverless Computing, Overview of Current State and Future Trends in Research and Industry

A Survey on Serverless Computing and its Implications for JointCloud Computing

Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda

COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms

Cold Start in Serverless Computing: Current Trends and Mitigation Strategies

Cold Start Influencing Factors in Function as a Service

ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Fireworks: A Fast, Efficient, and Safe Serverless Framework using VM-level post-JIT Snapshot

FnSched: An Efficient Scheduler for Serverless Functions

Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing

From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era

In Search of a Fast and Efficient Serverless DAG Engine

LambdaLite: Application-Level Optimization for Cold Start Latency in Serverless Computing

Mitigating Cold Start Problem in Serverless Computing with Function Fusion

Mitigating Cold Starts in Serverless Platforms

Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach

Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing

OPTIMIZING COLD START LATENCY IN SERVERLESS COMPUTING

Optimizing and Extending Serverless Platforms: A Survey

Retention-Aware Container Caching for Serverless Edge Computing

SAND: Towards High-Performance Serverless Computing

Sequoia: Enabling !ality-of-Service in Serverless Computing

Serverless Computing Architecture Security and Quality Analysis for Back-end Development

Serverless Performance and Optimization Strategies

Serverless Computing: A Security Perspective

Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing

WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing

Software Engineering for Serverless Computing

Serverless Computing Current Trends and

The Serverless Computing Survey: A Technical Primer for Design Architecture

Characterizing and Mitigating the I/O Scalability Challenges for Serverless Applications

Efficiency in the Serverless Cloud Computing Paradigm: A Survey Study

Exploiting Serverless Runtimes for Large-Scale Optimization

Harnessing the Power of Serverless Runtimes for Large-Scale Optimization

Narrowing the Gap Between Serverless and its State with Storage Functions

Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

Serverless Architectures Review, Future Trend and the Solutions to Open Problems

Function delivery network: Extending serverless computing for heterogeneous platforms

Serverless Computing A Survey of Opportunities Challenges and

Compiler-Assisted Semantic-Aware Encryption for Efficient and Secure Serverless Computing

How to Design a Secure Serverless Architecture

Serverless Computing Security: Protecting Application Logic

Status of Serverless Computingand Function-as-a-Service (FaaS) inIndustry and Research

The Ifs and Buts of Less is More: A Serverless Computing Reality Check

Tracing Function Dependencies Across Cloud

Serverless Testing: Tool Vendors’ and Experts’ Points of View

Automated Fine-Grained CPU Cap Control in Serverless Computing Platform

Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit

Practical Scheduling for Real-World Serverless Computing

Serverless Computing

Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement

Caching Techniques to Improve Latency in Serverless Architectures

Optimizing Stateful Serverless Computing

Formal Foundations of Serverless Computing

Kappa: A Programming Framework for Serverless Computing

Securing Serverless Computing

OverSketched Newton: Fast Convex Optimization for Serverless Systems

INFINICACHE: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Serverless Workflows with Durable Functions and Netherite

Modeling and Optimization of Performance and Cost of Serverless Applications

Triggerflow: Trigger-based orchestration of serverless workflows

Optimized container scheduling for data-intensive serverless edge computing

Securing Serverless Computing: Challenges, Solutions, and Opportunities

Transactions across serverless functions leveraging stateful dataflows