Serverless-arkkitehtuurin cold start -ilmiö ja optimointimenetelmät

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Toukokuu 2026
Oskari Pessinen

TURUN YLIOPISTO
Tietotekniikan laitos

Oskari Pessinen: Serverless-arkkitehtuurin cold start -ilmiö ja optimointimenetelmät

TkK-tutkielma, 20 s.
Tietotekniikka
Toukokuu 2026

Serverless-arkkitehtuuri on pilvipalvelumalli, jossa palveluntarjoaja hallitsee sovelluksen ajonaikaisen infrastruktuurin ja resurssit varataan vain todellisen käytön mukaan. Mallin keskeinen haaste on cold start -ilmiö, jossa funktion suoritus viivästyy suoritusympäristön alustamisen vuoksi. Tämä tutkielma on kirjallisuuskatsaus, jossa tarkastellaan cold start -ilmiön syntymekanismeja sekä kolmea keskeistä menetelmää sen vähentämiseksi: warm pool -ratkaisuja, snapshot-tekniikoita ja ennakoivaa skaalausta. Tutkielmaa varten käytiin läpi kirjallisuutta IEEE Xplore- ja ACM Digital Library -tietokannoista vuosilta 2018–2025.

Tutkielman perusteella cold start -viiveet syntyvät serverless-arkkitehtuurin scale-to-zero -periaatteesta, ja niiden kestoon vaikuttavat erityisesti ohjelmointikielen ajoympäristön raskaus, riippuvuuksien määrä sekä alustan eristysteknologia. Warm pool -ratkaisut vähentävät viiveitä ylläpitämällä valmiita kontteja, mutta kuluttavat resursseja jatkuvasti. Snapshot-tekniikat ohittavat alustusvaiheen palauttamalla ympäristön tallennetusta tilasta ja tarjoavat merkittäviä parannuksia erityisesti raskaiden ajoympäristöjen kohdalla. Ennakoiva skaalaus pyrkii ennustamaan tulevia kutsuja ja valmistelemaan ympäristöjä ennakoivasti, mutta sen tehokkuus riippuu ennustemallin tarkkuudesta. Yksikään menetelmistä ei poista cold start -ilmiötä täysin, ja käytännössä tehokkain lähestymistapa yhdistää useita menetelmiä sovelluksen vaatimusten mukaan.

Asiasanat: serverless-arkkitehtuuri, FaaS, kylmäkäynnistys, pilvipalvelut, snapshot-tekniikat, ennakoiva skaalaus

Sisällys

1 Johdanto

Pilvipalvelut ovat yleistyneet nopeasti ja muuttaneet huomattavasti tapaa, miten ohjelmistoja kehitetään ja ajetaan. Yhä useammat organisaatiot ovat siirtäneet sovelluksiaan pilveen, ja pilvipalvelumarkkinat ovat kasvaneet merkittävästi viime vuosikymmenen aikana. Perinteisessä pilvipalvelumallissa organisaatiot varaavat virtuaalitietokoneita tai kontteja, joiden kapasiteetti määritellään etukäteen ja niitä hallitaan jatkuvasti käynnissä olevina. Tälläinen malli tarjoaa joustavuutta perinteiseen omaan konesaliin verrattuna, sillä resursseja voidaan lisätä ja vähentää tarpeen mukaan. Malli edellyttää kuitenkin kapasiteetin ennakointia, infrastruktuurin suunnittelua sekä jatkuvaa resurssien hallintaa [1].

Viime vuosina niin sanotut palvelimettomat arkkitehtuurit (engl. serverless) ovat nousseet esiin vaihtoehtona perinteisille pilvipalveluratkaisuille. Serverless-arkkitehtuuri lupaa kustannustehokkuutta, skaalautuvuutta sekä vähäisempää infrastruktuurin hallintaa. Serverless-arkkitehtuuri on pilvipalvelumalli, jossa pilvipalveluntarjoaja hoitaa täysin sovelluksen ajonaikaisen infrastruktuurin. Perinteisiin pilvipalveluihin verrattuna resurssit eivät ole jatkuvasti varattuina, vaan suoritus tapahtuu kutsupohjaisesti [1]. Serverless-arkkitehtuuriin kuitenkin liittyy teknisiä haasteita, joista keskeinen on kylmäkäynnistysilmiö (engl. cold start), jossa funktion suoritus viivästyy suoritusympäristön alustamisen takia [2]. Cold start -viiveet voivat olla sadoista millisekunneista useisiin sekunteihin, mikä tekee ilmiöstä

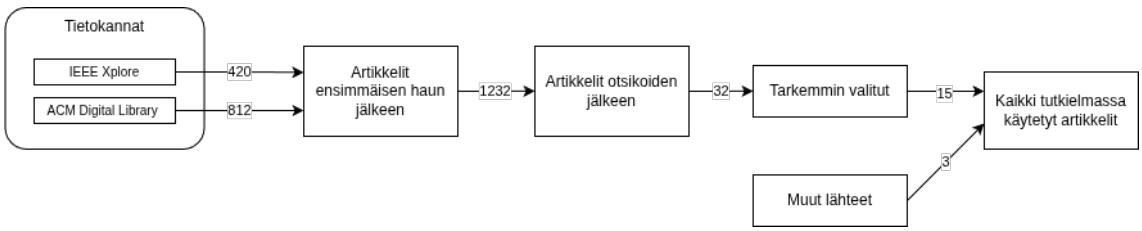
merkittävän ongelman erityisesti viiveherkissä sovelluksissa, kuten reaaliaikaisissa rajapinnoissa.

Tämä kandidaatintutkielma on toteutettu kirjallisuuskatsauksena. Tutkielmassa keskitytään tarkemmin cold start -ilmiöön serverless-arkkitehtuurissa ja menetelmiin ilmiön vähentämiseksi. Näiden tavoitteiden pohjalta on muodostettu seuraavat kaksi tutkimuskysymystä:

TK1: Mistä cold start -viiveet serverless-arkkitehtuurissa syntyvät?

TK2: Millaisia keinoja cold start -viiveiden vähentämiseen on?

Tutkielmaa varten on käyty läpi kirjallisuutta kahdesta eri tietokannasta: IEEE Xplore ja ACM Digital Library. Haku on toteutettu englanniksi. Hakua on rajattu sisältämään tuloksia vuosilta 2018-2025. Haussa on käytetty kolmea eri hakulausetta. Ensimmäinen hakulauseke on *("serverless computing"OR "Function as a Service"OR FaaS) AND ("cold start")*, jolla on pyritty löytämään serverless-arkkitehtuurin kylmäkäynnistykseen keskittyvää kirjallisuutta. Toinen hakulauseke on *("serverless computing"OR "Function as a Service")*, jolla on haettu haettu laajemmin serverless-arkkitehtuuria käsitteleviä lähteitä taustatiedon tueksi. Kolmas hakualuseke on *("serverless computing"OR FaaS) AND ("cold start") AND ("warm pool"OR prewarming OR snapshot OR "predictive scaling")*, jolla on haettu erityisesti optimointimenetelmiä käsitteleviä julkaisuja. Tiedonhakuprosessi on kuvattu kuvassa 1.1.



Kuva 1.1: Tutkielman tiedonhakuprosessi.

Tutkielman luvussa 2 esitellään serverless-arkkitehtuurin peruskäsitteet, Function-as-a-Service -malli sekä cold start -ilmiön syntymekanismit ja taustatekijät. Luvussa 3 käsitellään kolme keskeistä optimointimenetelmää cold start -viiveiden vähentämiseksi: warm pool -ratkaisut, snapshot-tekniikat ja ennakoiva skaalaus. Lopuksi on yhteenveto luku 4, jossa kootaan tutkielman havainnot yhteen, vastataan asetettuihin tutkimuskysymyksiin ja arvioidaan eri menetelmien soveltuvuutta erilaisiin käyttötapauksiin.

2 Serverless-arkkitehtuuri ja cold start -ilmiö

2.1 Serverless-arkkitehtuurin peruskäsitteet

Serverless-arkkitehtuuri on pilvipalvelumalli, jossa infrastruktuurin hallinta siirretään kehittäjältä pilvipalveluntarjoajan hallittavaksi [1]. Toisin kuin perinteisessä infrastruktuuri palveluna (engl. Infrastructure as a Service, IaaS) -mallissa, kehittäjän ei tarvitse itse huolehtia resurssien tai käyttöjärjestelmän hallinnasta, vaan sovelluskoodi ajetaan palveluntarjoajan ylläpitämässä hallitussa ympäristössä [3].

Kaupallisista serverless-alustoista tunnetuin on vuonna 2014 julkaistu AWS Lambda, jota seurasivat muun muassa Google Cloud Functions ja Microsoft Azure Functions. Näiden lisäksi on kehitetty avoimen lähdekoodin alustoja, kuten Apache OpenWhisk ja Knative, jotka mahdollistavat serverless-arkkitehtuurin rakentamisen omaan infrastruktuuriin [4].

Vastuunjakoa havainnollistetaan kuvassa 2.1, jossa nähdään kuinka serverless-arkkitehtuurissa suurempi osa arkkitehtuurista on palveluntarjoajan vastuulla perinteiseen IaaS-malliin verrattuna. Kuvassa on eroteltu kehittäjän ja palveluntarjoajan vastuualueet eri väreillä. Tämä korostaa serverless-mallin tuomaa korkeampaa abstraktiotasoa, sillä alustan ylläpito, fyysiset palvelimet, virtualisointikerros sekä käyttöjärjestelmien päivitykset siirtyvät kokonaan pilvipalveluntarjoajan hoidetta-

Kuva 2.1: IaaS- ja Serverless-arkkitehtuurien vertailu. Mustalla on merkitty palveluntarjoajan vastuualueet ja Sinisellä kehittäjän vastuualueet.

viksi. Kehittäjän vastuulle jää tällöin vain puhtaasti sovelluslogiikan toteuttaminen ja siihen liittyvien funktioiden määrittely.

Kirjallisuudessa serverless-käsitettä käytetään laajassa ja suppeassa merkityksessä. Laajassa merkityksessä serverless viittaa pilvipalveluihin, joissa infrastruktuurin hallinta on siirretty kokonaan palveluntarjoajalle. Tällöin serverless voi kattaa useita taustapalveluita, kuten tietokannat, viestinvälitysjärjestelmät ja tallennusratkaisut (engl. Backend as a Service, BaaS). Suppeammassa merkityksessä serverless viittaa eritysesti funktio palveluna -malliin (engl. Function as a service, FaaS), jossa sovellus jaetaan pieniin yksittäisiin toisistaan erotettuihin lyhytkestoisiin funktioihin. Tässä tutkielmassa serverless-käsitettä käytetään sen suppeammassa merkityksessä, eli viitataan erityisesti FaaS-malliin ja siihen liittyvään tapahtumapohjaiseen funktioiden suoritusympäristöön. [1]

Serverless voidaan ajatella pilvipalveluiden uutena abstraktiotasona, sen limittyessä osittain alusta palveluna (engl. Platform as a Service, PaaS) -mallin kanssa. PaaS -mallissa palveluntarjoaja hallitsee käyttöjärjestelmän ja ajonaikaisen ympäristön, mutta kehittäjä vastaa edelleen sovelluksen käyttöönotosta ja skaalautumi-

sesta. Serverless vie tätä abstraktiota pidemmälle: kehittäjä työskentelee korkean tason käsitteiden parissa, kuten funktioiden ja tapahtumien, kun taas infrastruktuurin hallinta on palveluntarjoajan vastuulla. Eli kehittäjä keskittyy liiketoimintalogiikkaan ja palveluntarjoaja taas vastaa sovellusten orkesteroinnista, käyttöönotosta ja saatavuudesta. [3]

2.2 Function-as-a-Service -malli

FaaS -mallin keskeinen toimintaperiaate on tapahtumapohjainen suoritus. Funktiot eivät siis ole jatkuvasti aktiivisena, vaan käynnistyvät ulkoisen tapahtuman, kuten HTTP-pyynnön seurauksena. Resursseja ei tarvitse pitää jatkuvasti varattuina, vaan suoritusympäristöt vapautetaan, kun ne ovat käyttämättöminä. Tämä mahdollistaa skaalaaminen nollaan (engl. scale-to-zero) -mallin, jossa sovelluksen resurssien laskutus perustuu sovelluksen todelliseen käyttöön [5].

Teknisellä tasolla yksittäiset funktiot ajetaan omissa eristetyissä ympäristöissä, kuten konteissa (engl. container) tai kevyissä virtuaalikoneissa (engl. microVM). Kontti on kevyt virtualisointiympäristö, joka sisältää sovelluksen koodin, ajonaikaisen ympäristön ja tarvittavat riippuvuudet, mutta jakaa isäntäkoneen käyttöjärjestelmän ytimen muiden konttien kanssa. Kevyet virtuaalikoneet, kuten AWS:n käyttämä Firecracker, tarjoavat vahvemman eristyksen ajamalla jokaista funktiota omassa minimaalisessa käyttöjärjestelmäytimessä, mutta ovat silti huomattavasti perinteisiä virtuaalikoneita kevyempiä [6]. Eristys on olennaista sekä turvallisuuden että resurssienhallinnan kannalta: se estää funktioita vaikuttamasta toistensa suoritukseen ja varmistaa, ettei yksittäinen funktio pääse käsiksi toisen funktion muistiin tai tiedostoihin.

Arkkitehtuurille tyypillistä on että funktiot ovat tilattomia (engl. stateless), eli ne eivät ylläpidä suoritusten välistä tilaa, ja pysyvä dataa tulee tallentaa muihin taustapalveluihin. Tilattomuus helpottaa automaattista skaalautumista, sillä uusia

instansseja voidaan käynnistää eivätkä ne ole riippuvaisia aiemmista. Tilattomuus kuitenkin rajoittaa joidenkin sovellusten ajamista jotka tarvitsevat oman tilan, kuten iteratiivisten koneoppimisalgoritmien. [1]

FaaS-mallin houkuttelevuus perustuu pitkälti sen kustannustehokkaaseen ”maksa vain käytöstä” -laskutusmalliin (engl. pay-as-you-go). Kustannukset eivät muodostu pelkästään suoritusajasta, vaan ne lasketaan yleisesti funktion suoritusajan ja sille varattujen reusurssien tulona. Laskutuksen tarkkuus on yleensä millisekuntitasolla, mikä eroaa merkittävästi perinteisten virtuaalipalvelimien minuutti- tai tuntipohjaisesta hinnoittelusta [7].

FaaS-malli ei kuitenkaan sovellu kaikkiin käyttötapauksiin. Alustoilla on tyypillisesti rajoituksia funktion suoritusajalle: esimerkiksi AWS Lambda rajoittaa yksittäisen funktion suorituksen enintään 15 minuuttiin [8]. Tämä tekee siitä sopimattoman pitkäkestoisille prosesseille, kuten suurten tietoaineistojen käsittelylle. Lisäksi funktioiden välinen kommunikaation tapahtuu ulkoisten palveluiden kautta, mikä lisää viivettä entisestään, verrattuna perinteisiin malleihin, joissa sovelluksen komponentit voivat suoraan keskenään.

2.3 Cold start -ilmiön synty ja taustatekijät

Serverless-arkkitehtuurissa sovelluslogiikka suoritetaan toisistaan eristetyissä konteissa tai virtuaalikoneissa. Kun funktiota kutsutaan, alusta tarkistaa onko funktiolle jo valmista konttia (engl. warm container). Jos ei ole valmiiksi vapaata konttia alustan on luotava uusi kontti funktiolle. Tätä uuden suoritusympäristön alustamisesta johtuvaa viivettä kutsutaan kylmäkäynnistykseksi (engl. cold start) [5]. Kylmäkäynnistysviive voi olla sadoista millisekunneista useisiin sekunteihin, mikä heikentää käyttäjäkokemusta eritysesti viiveherkissä sovelluksissa, joissa matala vasteaika on kriittistä.

Ilmiö on suora seuraus serverless-arkkitehtuurin dynaamisesta resurssienhallinnasta. Resurssien säästämiseksi alusta ajaa alas käyttämättömät kontit tietyn ajan jälkeen. Golec et al. [9] mukaan kylmäkäynnistykset johtuvat pääasiassa kolmesta tekijästä:

Valmiin suoritusympäristön puute: Kun pyyntö saapuu palvelimelle, eikä valmista ympäristöä ole, on alustan käynnistettävä uusi kontti. Tämä valmisteluprosessi, joka sisältää kontin alustamisen, sekä ajon aikaisen ympäristön ja kirjastojen lataaminen, aiheuttaa suurimman osan viiveestä.

Skaalautumisviiveet: Serverless-arkkitehtuurin dynaaminen skaalaus reagoi automaattisesti kuormitukseen. Äkilliset piikit pyyöntöjen määrässä voivat kuitenkin aiheuttaa viivettä resurssien allokoinnissa.

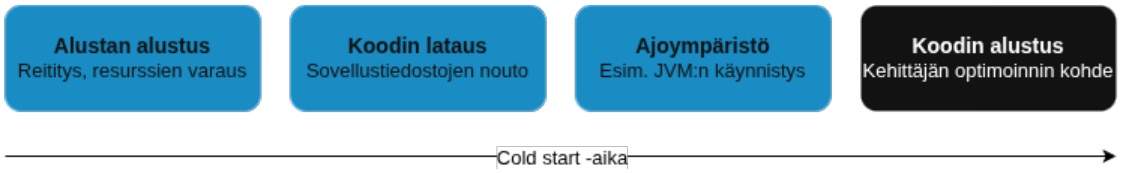
Resurssikilpailu: Serverless-alustoilla fyysiset resurssit, kuten prosessoriaika (CPU) ja keskusmuisti (RAM), on jaettu useiden eri funktioiden kesken. Kuormistutilanteissa syntyvä resusrssikilpailu voi hidastaa konttien käynnistymistä ja lisätä viivettä entisestään.

Kylmäkäynnistys ei myöskään rajoitu pelkästään funktion ensimmäiseen kutsuun, vaan kun samalle funktiolle saapuu useita samanaikaisia pyyntöjä, jokainen rinnakkainen kutsuu vaatii oman suoritusympäristönsä [2]. Cold start -esiintymistiheys riippuu siis paljon sovelluksen kuormitusprofiilista: tasaisesti kuormitetut sovellukset kokevat niitä harvemmin, kun taas harvoin kutsutut tai äkillisiä kuormituspiikkeihin altistuvat kärsivät niistä huomattavasti enemmän [10].

Kylmäkäynnityksen kesto vaihtelee myös huomattavasti eri alustojen välillä. Eri palveluntarjoajat käyttävät erilaisia eristysteknologioita suoritusympäristöjen toteutukseen. Esimerkiksi AWS Lambda käyttää Firecracker-nimistä kevyttä mikrovirtuaalikoneteknologiaa, jossa jokainen suoritusympäristö ajaa omaa minimaalista käyttöjärjestelmäydintä [6], [8]. Avoimen lähdekoodin alustoissa, kuten Knativessa,

funktiot ajetaan tyypillisesti Kubernetes-konteissa [4]. Eristysteknologia vaikuttaa suoraan kylmäkäynnistysviiveeseen, sillä eri teknologiat vaativat erilaisia alustus- ja käynnistysvaiheita.

Cold start -ilmiön yleisyys vaihtelee huomattavasti eri alustojen ja sovellusten välillä. Wang ym. [11] tutkivat kaupallisten alustojen käyttäytymistä ja havaitsivat merkittäviä eroja konttien säilytysajoissa ja cold start -viiveissä AWS Lambdan, Azure Functionsin ja Google Cloud Functionsin välillä. Tuotantoympäristöissä cold startien osuus kaikista kutsuista on tyypillisesti pieni mutta merkittävä: AWS on esimerkiksi raportoinut, että noin yksi prosentti kutsuista kohtaa cold startin [9]. Vaikka osuus vaikuttaa vähäiseltä, suurten sovellusten mittakaavassa tämä tarkoittaa tuhansia hidastettuja pyyntöjä päivittäin. Teknisesti kylmäkäynnistys ei ole yksittäinen tapahtuma, vaan se voidaan jakaa neljään peräkkäiseen vaiheeseen, kuten kuvassa 2.2 on esitetty.



Kuva 2.2: Cold start -ilmiön osatekijät ja vaiheet.

Aluksi alusta varaa tarvittavat resurssit. Tämän jälkeen funktion koodi ladataan palvelimelle ja puretaan. Kolmannessa vaiheessa käynnistetään valitun ohjelemointikielen ajonaikainen ympäristö (esim. JVM- tai Python-tulkki). Lopuksi suoritetaan käyttäjän koodin alustus, kuten tietokantayhteyksien luominen ja kirjastojen lataus. Näiden jälkeen varsinainen funktiologiikka voidaan suorittaa [8].

Kylmäkäynnistysviiveeseen vaikuttaa merkittävästi valitut teknologiat. Eristyisesti ohjelmointikielen valinta: tulkattavat kielet, kuten Python ja JavaScript käynnistyvät tyypillisesti nopeammin. Kun taas käännettävät tai raskaampia virtuaalikoneita käyttävät kielet, kuten Java ja C# käynnistyvät hitaammin [11]. Myös

sovelluksen koodipaketin koolla on suora yhteys latausvaiheen kestoon, suuret riippuvuudet ja kirjastoot pidentävät huomattavasti alustusaikaa [7].

Tulee myös huomata, että cold start -ilmiö kytkeytyy suoraan luvussa 2.2 käsiteltyyn tilattomuuteen. Koska funktioinstanssit eivät säilytä tilaa suoritusten välillä, alusta voi vapauttaa ne milloin tahansa ilman tiedon menetyksen riskiä. Tämä madaltaa kynnystä käyttämättömien konttien alas ajamiseen ja lisää kylmäkäynnistysten todennäköisyyttä [1]. Tilattomuus on se sama piirre, joka mahdollistaa serverless-arkkitehtuurin joustavan skaalautumisen, mutta aiheuttaa samalla myös kylmäkäynnistysviiveet.

3 Optimointimenetelmät

3.1 Warm pool -ratkaisut

Warm pool -ratkaisut ovat yksi keskeisimmistä tavoista serverless-arkkitehtuurin cold start -ilmiön vähentämiseksi. Menetelmät perustuvat siihen, että suoritusympäristöjä pidetään valmiiksi alustettuina, vaikka niissä ei suoriteta funktioita. Kun uusi pyyntö saapuu, järjestelmä ohjaa sen valmiina käynnissä olevaan ympäristöön [4].

Kaksi keskeisintä warm pool -ratkaisua ovat esialustetut kontit (engl. pre-warmed containers) ja konttien uudelleenkäyttö (engl. keep-alive) [10]. Esialustetut kontit -menetelmässä alusta luo ennakoivasti geneerisiä kontteja, joihin on valmiiksi asennettu ajonaikainen ympäristö. Pyynnön saapuessa konttiin ladataan ainoastaan funktion koodi. Lin ja Glikson [4] toteuttivat tämän lähestymistavan Knative-alustalle ja osoittivat, että jo yhden valmiiksi alustetun instanssin pool vähensi P99-vastausaikaa 85 %. Konttien uudelleenkäyttö -menetelmässä taas funktion suorituksen jälkeen konttia ei tuhota, vaan se pidetään valmiina tietyn ajan saman funktion tulevia kutsuja varten. Eli toisin kuin esialustetuissa konteissa, keep-alive-kontti on sidottu tiettyyn funktioon [10].

Esimerkiksi AWS Lambda -alustalla keep-alive-mekanismi toimii siten, että funktion suorituksen jälkeen suoritusympäristö jäädytetään ja säilytetään uudelleenkäyttöä varten. Jos uusi pyyntö saapuu samalle funktiolle, ympäristö voidaan ottaa käyttöön ilman alustusta. Ympäristön säilytysaika on kuitenkin rajallinen, ja Lambda

voi tuhota ympäristön muutaman tunnin jälkeen, vaikka funktiota kutsuttaisiin jatkuvasti [8].

Warm pool -ratkaisuihin kuitenkin liittyy merkittäviä haasteita. Valmiina ylläpidetut suoritusympäristöt kuluttavat muistia ja prosessoriaikaa, vaikka niissä ei suoriteta pyyntöjä. Tämä on suoraan ristiriidassa serverless-arkkitehtuurin kustannustehokkuuden kanssa. Koska warm pool -kapasiteetti on rajallinen, tarvitaan poistopolitiikka hallitsemaan, mitkä kontit pidetään valmiina. Poistopolitiikan valinnalla on suora vaikutus sekä cold start -viiveiden esiintymistiheyteen että resurssien kulutukseen. Liian aggressiivinen poistopolitiikka lisää kylmäkäynnistyksiä, kun taas liian konservatiivinen politiikka kasvattaa resurssien käyttöä ja heikentää kustannustehokkuutta. Yksinkertainen ja yleisesti käytetty menetelmä on LRU-strategia (engl. Least Recently Used), jossa pisimpään käyttämättä ollut kontti poistetaan poolista ensimmäisenä kapasiteetin täyttyessä [10]. Keep-alive-menetelmässä kiinteä säilytysaika ei myöskään ole optimaalinen, sillä se ei ota huomioon funktiokohtaisia kutsuvälejä [10]. Lisäksi keep-alive ei auta tilanteessa, joissa samalle funktiolle saapuu samaan aikaan useita pyyntöjä, sillä jokainen rinnakkainen pyyntö vaatii oman suoritusympäristönsä [8]. Warm pool -ratkaisut siis vähentävät tehokkaasti kylmäkäynnistysviivettä, mutta eivät poista ongelmaa kokonaan, ja samalla ne heikentävät serverless-arkkitehtuurin keskeisiä etuja.

Warm pool -ratkaisut edustavat yksinkertaisinta lähestymistapaa cold start -ongelmaan, ja niiden vahvuus on toteutuksen suoraviivaisuus. Menetelmät eivät vaadi muutoksia itse funktioiden koodiin, vaan toimivat alustatasolla. Warm pool -ratkaisut eivät siis puutu itse alustusprosessin kestoon, vaan ainoastaan pyrkivät välttämään sen. Vaihtoehtoinen lähestymistapa on pyrkiä tekemään itse alustusprosessista niin nopea, ettei valmiiden konttien jatkuva ylläpito ole välttämätöntä. Tähän ajatukseen perustuvat snapshot-tekniikat, joita käsitellään seuraavassa luvussa.

3.2 Snapshot-tekniikat

Toinen lähestymistapa kylmäkäynnistyviiveiden vähetämiseen on snapshot-tekniikat, jossa alustetun suoritusympäristön tila tallennetaan levylle ja palautetaan myöhemmin uuden instanssin käynnistämisen sijaan [12]. Tämä ohittaa alustusvaiheen raskaimmat osat, kuten ajonaikaisen ympäristön käynnistämisen, riippuvuuksien lataamisen ja sovelluskoodin alustamisen.

Snapshot-tekniikat voidaan jakaa karkeasti kahteen eri kategoriaan eristysteknologian mukaan: prosessitason ja virtuaalikonetason ratkaisuihin. Prosessitason snapshotit perustuvat tyypillisesti CRIU-työkaluun (Checkpoint/Restore In Userspace), joka tallentaa käynnissä olevan Linux-prosessin tilan, ja palauttaa sen myöhemmin kloonattuun prosessiin [13]. Silva ym. [12] esittivät Prebaking-tekniikan, jossa funktion suoritusympäristö alustetaan etukäteen ja sen tila tallennetaan konttikuvan sisälle CRIU:n avulla. Nimitys tulee siitä ajatuksesta, että alustustyö tehdään valmiiksi jo rakennusvaiheessa ja tallennetaan osaksi konttikuvan. Kun alusta myöhemmin tarvitsee uuden kopion, CRIU palauttaa prosessin suoraan tallennetusta snapshotista sen sijaan, että koko alustusprosessi toistettaisiin. Silva ym. [13] osoittivat, että tekniikka paransi funktion käynnistysaikaa kaikissa testatuissa ajoympäristöissä: Node.js-ympäristössä parannus oli 3,5-kertainen, kun taas CPython-ympäristössä se oli jopa 12-kertainen [13]. CPythonin suurempi hyöty selittyy sillä, että Pythonin ajoympäristön alustus on huomattavasti raskaampi kuin esimerkiksi Node.js:n, jolloin snapshotin tuoma aikasäästö on suhteessa suurempi.

Virtiaalikonetason snapshot-tekniikat hyödyntävät mikrovirtaalitietokoneiden, kuten Firecrackerin tilanhallintamekanismeja. Firecrackerin snapshot-toteutuksessa mikrovirtuaalikoneen muistin ja laitteiston tila tallennetaan tiedostoiksi levylle. Palautusvaiheessa Firecracker käyttää muistinkartoitusta, jossa muistisivut ladataan vasta kun niitä tarvitaan sen sijaan että koko muistisisältö luettaisiin kerralla [14].

Virtuaalikonetason snapshotit ovat yleisempiä kaupallisissa serverless-alustoissa kuin prosessitason ratkaisut. Tämä johtuu turvallisuussyistä, sillä virtuaalikone eristää funktion kokonaan omaan ympäristöönsä käyttöjärjestelmän ytimen tasolla, kun taas prosessitason ratkaisut jakaa saman ytimen usean funktion kesken. Prosessitason toteutukset soveltuvat kuitenkin hyvin yksityisiin pilviympäristöihin ja organisaatioiden sisäisiin hallittuihin alustoihin, joissa tiukat monen käyttäjän eristysvaatimukset ovat vähäisempiä. [12].

Kaupallisella puolella AWS on integroinut snapshot-tekniikat Lambda-alustaansa SnapStart-ominaisuutena. SnapStart toimii siten, että kun funktion uusi versio julkaistaan, Lambda alustaa suoritusympäristön ja ottaa Firecracker-mikrovirtuaalikoneen muistista ja levystä tilannevedoksen, joka salataan ja tallennetaan välimuistiin. Myöhemmillä kutsuilla Lambda palauttaa suoritusympäristön tallennetusta snapshotista alustusvaiheen toistamisen sijaan [15]. SnapStart hyödyntää monikerroksista välimuistiarkkitehtuuria, usein kutsuttujen funktioiden snapshotit säilytetään lähellä suoritusympäristöä nopeassa L1-välimuistissa, kun taas harvemmin käytetyt osat haetaan hitaammasta L2-kerroksesta [16].

Snapshot-tekniikalla on myös myönteinen vaikutus muistin käyttöön. Ustiugov ym. [14] havaitsivat, että snapshotista palautetun funktion muistijälki oli vain 8–99 megatavua, kun taas saman funktion käynnistäminen alusta alkaen vaati 148–256 megatavua. Tämä vähennys muistinkäytössä johtuu siitä, että snapshotista ladataan vain ne muistisivut, joita funktio oikeasti käyttää. Pienempi muistijälki mahdollistaa useamman funktion samanaikaisen ajamisen samalla palvelimella, mikä on merkittävä etu palveluntarjoajan näkökulmasta.

Snapshot-tekniikoihin liittyy myös haasteita. Koska palautettu ympäristö on kopio aiemmin alustetusta tilasta, alustusvaiheen aikana luodut arvot, kuten aikaleimat ja verkkoyhteydet, voivat toistua eri instansseissa. Tämä edellyttää kehittäjiltä erityistä huomiota tilan hallintaan palautuksen jälkeen, esimerkiksi ajonaikaisten-

koukkujen (engl. runtime hooks) avulla, jotka alustavat nämä arvot uudelleen palautuksen yhteydessä [15].

Snapshot-tekniikoiden ja warm pool -ratkaisujen lähestymistavat ovat pohjimmiltaan erilaisia. Warm pool -ratkaisut pitävät kontteja käynnissä ja kuluttavat jatkuvasti resursseja, kun taas snapshot-tekniikat tallentavat tilan levylle ja vapauttavat resurssit. Tämä tekee snapshotista erityisen hyödyllisen harvoin kutsutuille funktioille, joille jatkuva kontin ylläpitäminen olisi kustannustehotonta. Toisaalta snapshot-palautus ei ole yhtä nopeaa kuin valmiin kontin käyttöönotto, joten usein käytetyille funktioille warm pool voi olla edelleen parempi vaihtoehto.

3.3 Ennakoiva skaalaus

Kolmas lähestymistapa cold start -viiveiden vähentämiseen on ennakoiva skaalaus (engl. predictive scaling), jossa tulevia funktiokutsuja pyritään ennustamaan ja suoritusympäristöjä valmistelemaan etukäteen ennen varsinaisten pyyntöjen saapumista. Toisin kuin warm pool -ratkaisuissa, joissa valmiita kontteja ylläpidetään reaktiivisesti aiemman käytön perusteella, ennakoiva skaalaus pyrkii varautumaan tulevaan kuormitukseen [9].

Ennakoivan skaalauksen perusajatus on, että monien serverless-sovellusten kuormitusprofiileissa esiintyy toistuvia kaavoja, kuten vuorokaudenaikaan sidottuja piikkejä tai kausittaista vaihtelua. Näitä kaavoja voidaan mallintaa ja käyttää ennusteita konttien esilämmittämiseen juuri ennen kuormituksen kasvua. Jos ennuste osuu oikein, pyynnöt kohtaavat valmiita kontteja ja cold start -viiveet vältetään.

IceBreaker on yksi tunnetuimmista ennakoivan skaalauksen toteutuksista. Se käyttää Fourier-pohjaista ennustemallia funktiokutsujen kaavojen tunnistamiseen ja ennustaa tulevien kutsujen ajoitusta [17]. Malli perustuu siihen, että lähes 98 prosenttia Microsoft Azuren serverless-funktioista osoittaa jonkinlaista jaksollisuutta kutsukaavoissaan. Fourier-muunnoksen avulla kutsuprofiilit voidaan hajottaa yksin-

kertaisiin sinimuotoisiin komponentteihin, joiden perusteella tulevien kutsujen ajoitus voidaan ennustaa [17]. Ennusteiden perusteella järjestelmä lämmittää kontteja etukäteen. Tutkimuksen tulokset osoittivat, että IceBreaker vähensi kokonaissuoritusaikaa 27 prosenttia ja ylläpitokustannuksia 45 prosenttia verrattuna perinteisiin menetelmiin [17].

Hu ym. [18] esittivät koneoppimiseen perustuvan lähestymistavan, jossa takaisinkytkettyä neuroverkkoa (engl. Recurrent Neural Network, RNN) käytetään tulevien funktiokutsujen ennustamiseen historiallisen datan perusteella. RNN soveltuu tähän tehtävään hyvin, sillä se pystyy mallintamaan aikasarjadatan ajallisia riippuvuuksia, eli oppimaan miten aiemmat kutsuprofiilit ennustavat tulevia kutsuja. Ennusteen pohjalta esilämmitysajoitin päättää, kuinka monta konttia valmistellaan etukäteen. Menetelmä paransi suorituskykyä 49,5 prosenttia verrattuna Apache OpenWhiskin oletuskäytäntöön [18].

Ennakoivan skaalauksen hyöty perustuu pitkälti ennustusmallin tarkkuuteen: jos ennuste aliarvioi kuormituksen, osa pyynnöistä kohtaa silti cold startin, ja jos se taas yliarvioi, resursseja kulutetaan turhaan [9]. Lisäksi äkilliset kuormituspiikit ovat vaikea ennustaa, jos ne eivät nooudata historiallisia kaavoja.

Koneoppimismallit vaativat toimiakseen myös riittävän määrän historiallista dataa, mikä voi olla haaste harvoin kutsutuille funktioille [17]. Monimutkaiset mallit, kuten LSTM-neuroverkot, voivat saavuttaa paremman tarkkuuden, mutta niiden laskentakustannus voi olla serverless-ympäristössä liian korkea suhteessa saavutettuun hyötyyn [17]. Esimerkiksi IceBreaker-tutkimuksessa LSTM-neuroverkon korvaaminen Fourier-mallilla tuotti vain marginaalisen parannuksen ennustetarkkuuteen, mutta sen laskentakustannus oli 243-kertainen [17]. Tämä osoittaa, että ennakoivassa skaalauksessa on löydettävä tasapaino ennustemallin tarkkuuden ja sen aiheuttaman laskentakuorman välillä.

Ennakoiva skaalaus pyrkii ratkaisemaan cold start -ongelman ennen kuin se ehtii ilmetä. Parhaimmillaan se voi lähes kokonaan poistaa cold start -viiveet sovelluksilta, joiden kuormitusprofiilit ovat ennustettavia. Menetelmä soveltuu hyvin tasaisesti kuormitetuille ja jaksollisille sovelluksille, kun taas satunnaisesti kutsutuille funktioille sen tuoma arvo voi jäädä vähäiseksi.

4 Yhteenveto

Tässä tutkielmassa tarkasteltiin cold start -ilmiötä serverless-arkkitehtuurissa ja menetelmiä sen vähentämiseksi. Tutkielman tavoitteena oli vastata kahteen tutkimuskysymykseen: mistä cold start -viiveet syntyvät ja millaisia keinoja niiden vähentämiseen on olemassa.

Ensimmäiseen tutkimuskysymykseen TK1 ”Mistä cold start -viiveet serverless-arkkitehtuurissa syntyvät?” voidaan vastata seuraavasti. Cold start -viiveet ovat suora seuraus serverless-arkkitehtuurin scale-to-zero -periaatteesta, jossa käyttämättömät suoritusympäristöt vapautetaan resurssien säästämiseksi ja uudet ympäristöt luodaan vasta kutsujen saapuessa. Viive koostuu useista peräkkäisistä vaiheista, kuten resurssien varaamisesta, kontin alustamisesta, ajoympäristön käynnistämisestä ja sovelluskoodin lataamisesta. Viiveen suuruuteen vaikuttavat erityisesti ohjelmointikielen ajoympäristön raskaus, riippuvuuksien määrä ja alustan eristysteknologia.

Toiseen tutkimuskysymykseen TK2 ”Millaisia keinoja cold start -viiveiden vähentämiseen on?” liittyen tutkielmassa käsiteltiin kolme optimointimenetelmää: warm pool -ratkaisut, snapshot-tekniikat ja ennakoiva skaalaus. Warm pool -ratkaisut pyrkivät välttämään cold startin ylläpitämällä valmiita kontteja. Snapshot-tekniikat ohittavat alustusvaiheen palauttamalla suoritusympäristön tallennetusta tilasta. Ennakoiva skaalaus taas pyrkii ennustamaan tulevia kuormituspiikkejä ja valmistelemaan ympäristöjä sen mukaan.

Tutkielman perusteella voidaan todeta ettei mikään näistä menetelmistä poista täysin cold start -ilmiötä, ja kunkin soveltuvuus riippuu käyttötarkoituksesta. Menetelmät eroavat toisistaan paljon siinä, mihin ongelman osaan ne puuttuvat. Warm pool -ratkaisut eivät nopeuta itse alustusprosessia, vaan pyrkivät välttämään sen. Tämä tekee niistä yksinkertaisia mutta resurssitehottomia, sillä kontteja on pidettävä käynnissä myös hiljaisina aikoina. Snapshot-tekniikat puuttuvat suoraan alustusprosessin kestoon, mutta samalla siirtävät osan monimutkaisuudesta kehittäjälle, jonka on huolehdittava tilan oikeellisuudesta palautuksen jälkeen. Ennakoiva skaalaus yrittää poistaa ongelman täysin, mutta on samalla herkin virheille: ennustemallin epätarkkuus johtaa joko turhaan resurssien kulutukseen tai cold start -viiveisiin.

Cold start -ilmiön merkitys riippuu paljon sovelluksen luonteesta. Viiveherkkien sovellusten, kuten reaaliaikaisten rajapintojen, kohdalla jo yksittäinen cold start voi heikentää käyttäjäkokemusta merkittävästi, kun taas eräajo (engl. Batch job) sovelluksissa sadankaan millisekunnin viive ei ole haitaksi. Harvoin kutsuttavat funktiot kärsivät ilmiöstä huomattavasti enemmän kuin tasaisesti kuormitetut sovellukset, joissa warm pool -ratkaisut riittävät usein pitämään kontit valmiina. Ei ole siis olemassa yhtä ainutta optimaalista ratkaisua, vaan menetelmän valinnassa tulee ottaa huomioon sovelluksen viiverajoitteet, kuormitusprofiili ja kustannusrajoitteet.

Menetelmät eivät ole toisiaan poissulkevia, vaan ne voivat täydentää toisiaan. Esimerkiksi ennakoiva skaalaus voi hyödyntää snapshot-tekniikoita konttien nopeaan valmisteluun, ja warm pool voi toimia varajärjestelmänä silloin kun ennusteet ei osu oikein. Tälläisten hybridi ratkaisujen tutkiminen on kuitenkin melko varhaisessa vaiheessa, ja useimmat tutkimukset keskittyvät vielä menetelmiin erillään. Jatkotutkimuksen kannalta erityisen kiinnostavaa olisikin hybridiratkaisujen tutkiminen, joissa eri menetelmiä yhdistetään.

Tutkielman rajoitteena on, että käsitellyt tulokset perustuvat eri tutkimusten omiin koeasetelmiin ja työkuormiin, mikä tekee menetelmien suorasta vertailusta haastavaa. Yhteinäinen testiympäristö, jossa kaikkia mentelmiä testattaisiin samoilla työkuormilla ja alustoilla, antaisi paremman kuvan menetelmien heikkouksista ja vahvuuksista. Lisäksi tutkielma rajattiin kolmeen optimointimenetelmään, vaikka kirjallisuudessa on esitelty paljon muitakin menetelmiä, kuten koodin optimointi ja konttien jakaminen funktioiden välillä.

Lähdeluettelo

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

[2] J. Manner, M. Endreß, T. Heckel ja G. Wirtz, "Cold Start Influencing Factors in Function as a Service", teoksessa *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, s. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[3] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță ja A. Iosup, "Serverless is More: From PaaS to Present Cloud Computing", *IEEE Internet Computing*, vol. 22, nro 5, s. 8–17, 2018. DOI: 10.1109/MIC.2018.053681358.

[4] P. Lin ja A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach", *CoRR*, vol. abs/1903.12221, 2019. arXiv: 1903.12221. url: http://arxiv.org/abs/1903.12221.

[5] P. Vahidinia, B. Farahani ja F. S. Aliee, "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach", *IEEE Internet of Things Journal*, vol. 10, nro 5, s. 3917–3927, 2023. DOI: 10.1109/JIOT.2022.3165127.

[6] A. Agache et al., "Firecracker: lightweight virtualization for serverless applications", teoksessa *Proceedings of the 17th Usenix Conference on Networked Sys-*

tems Design and Implementation, sarja NSDI'20, Santa Clara, CA, USA: USENIX Association, 2020, s. 419–434, ISBN: 9781939133137.

[7] T. Yu et al., "Characterizing serverless platforms with serverlessbench", teoksessa *Proceedings of the 11th ACM Symposium on Cloud Computing*, sarja SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, s. 30–44, ISBN: 9781450381376. DOI: 10.1145/3419111.3421280.

[8] Amazon Web Services. "AWS Lambda Developer Guide: Execution environment", viitattu 26. toukokuuta 2026. url: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtime-environment.html.

[9] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill ja S. Uhlig, "Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions", *ACM Comput. Surv.*, vol. 57, nro 3, marraskuu 2024, ISSN: 0360-0300. DOI: 10.1145/3700875.

[10] A. C. Zhou, R. Huang, Z. Ke, Y. Li, Y. Wang ja R. Mao, "Tackling Cold Start in Serverless Computing with Multi-Level Container Reuse", teoksessa *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, s. 89–99. DOI: 10.1109/IPDPS57955.2024.00017.

[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart ja M. M. Swift, "Peeking Behind the Curtains of Serverless Platforms", teoksessa *Proceedings of the 2018 Acm Usenix International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, s. 133–145. DOI: 10.1145/3173162.3173172.

[12] P. Silva, D. Fireman ja T. E. Pereira, "Prebaking Functions to Warm the Serverless Cold Start", teoksessa *Proceedings of the 21st International Middleware Conference*, sarja Middleware '20, Delft, Netherlands: Association for Compu-

ting Machinery, 2020, s. 1–13, ISBN: 9781450381536. DOI: 10.1145/3423211.3425682.

[13] D. Fireman, P. Silva, T. E. Pereira, L. Mafra ja D. Valadares, ”Prebaking runtime environments to improve the FaaS cold start latency”, *Future Generation Computer Systems*, vol. 155, s. 287–299, 2024, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2024.01.019.

[14] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion ja B. Grot, ”Benchmarking, analysis, and optimization of serverless function snapshots”, teoksessa *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, sarja ASPLOS ’21, Virtual, USA: Association for Computing Machinery, 2021, s. 559–572, ISBN: 9781450383172. DOI: 10.1145/3445814.3446714.

[15] Amazon Web Services. ”Improving startup performance with Lambda SnapStart”, Amazon Web Services, Inc., viitattu 26. toukokuuta 2026. url: https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html.

[16] A. Kulkarni ja E. Heinz. ”Under the hood: how AWS Lambda SnapStart optimizes function startup latency”, viitattu 26. toukokuuta 2026. url: https://aws.amazon.com/blogs/compute/under-the-hood-how-aws-lambda-snapstart-optimizes-function-startup-latency/.

[17] R. B. Roy, T. Patel ja D. Tiwari, ”IceBreaker: warming serverless functions better with heterogeneity”, teoksessa *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, sarja ASPLOS ’22, Lausanne, Switzerland: Association for Computing Machinery, 2022, s. 753–767, ISBN: 9781450392051. DOI: 10.1145/3503222.3507750.

[18] Q. Hu, H. Li ja E. Nikougoftar, ”Mitigating cold start problem in serverless computing using predictive pre-warming with machine learning: Mitigating cold start problem in serverless...”, *Computing*, vol. 107, nro 1, joulukuu 2024, ISSN: 0010-485X. DOI: 10.1007/s00607-024-01382-y.