Tampereen yliopisto

Lasse Vuorinen

TULKATTAVIEN OHJELMOINTIKIELTEN SUOSIO SERVERLESS-PILVIYMPÄRISTÖISSÄ

Käytön syyt ja kustannusvaikutukset

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Kari Systä
Tarkastaja: Tiina Partanen
Joulukuu 2025

TIIVISTELMÄ

Lasse Vuorinen: Tulkattavien ohjelmointikielten suosio serverless-pilviympäristöissä: käytön syyt ja kustannusvaikutukset
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Joulukuu 2025

Serverless-arkkitehtuuri on yleistynyt pilvipohjaisissa sovelluksissa, sillä se siirtää palvelinten ylläpidon ja niiden skaalaamisen kokonaan pilvipalveluntarjoajien vastuulle. Tämä tapahtumapohjainen ja automaattisesti skaalautuva malli mahdollistaa organisaation keskittymisen puhtaasti liiketoimintalogiikan toteuttamiseen. Tulkattavat kielet kuten Python ja Node.js ovat nousseet suosituimmiksi kieliksi serverless-ympäristöissä huolimatta siitä, että käännettävät ohjelmointikielet kuten Go ja Java omaavat teoriassa paremman laskennallisen tehokkuuden.

Tämän diplomityön tavoitteena oli tutkia, mihin syihin tulkattavien ohjelmointikielten laaja suosio palvelimettomissa ympäristöissä perustuu. Lisäksi työssä tarkasteltiin ohjelmointikielen valinnan suoria ja epäsuoria vaikutuksia sovellusten kokonaiskustannuksiin ja suorituskykyyn. Tutkimus toteutettiin kirjallisuuskatsauksena, jonka aineisto kerättiin tieteellisistä julkaisuista, konferenssijulkaisuista sekä keskeisten pilvipalveluntarjoajien (AWS, Microsoft Azure ja Google Cloud) teknisestä dokumentaatiosta. Lähteiden valinnassa painotettiin julkaisuja vuosilta 2018–2025, jotta muodostettu kuva olisi mahdollisimman ajantasainen.

Analyysissa pureuduttiin erityisesti siihen, kuinka kielten suoritusympäristöt vaikuttavat kylmäkäynnistysviiveisiin sekä hinnoittelumalleihin. Serverless-ympäristöille tyypillinen kylmäkäynnistys aiheuttaa viivettä, kun funktio suoritetaan ensimmäistä kertaa tai pidemmän tauon jälkeen. Tutkimus osoittaa, että tulkattavat kielet ovat tässä suhteessa kilpailukykyisiä ja usein tehokkaampia kuin raskaammat käännettävät kielet, mikä vähentää sovellusten latenssia. Koska serverless-funktiot ovat usein lyhytkestoisia, tulkattavien kielten nopeampi käynnistymisaika kompensoi niiden hitaampaa suoritusnopeutta.

Tutkimus osoittaa, että tulkattavien kielten valinta on ensisijaisesti kehitystyön tehokkuuteen ja strategiseen ketteryyteen liittyvä päätös. Suosio selittyy niiden matalalla oppimiskynnyksellä ja laajoilla kirjastoekosysteemeillä, jotka mahdollistavat sovellusten nopeamman kehittämisen ja nopeamman siirtämisen tuotantoon. Tulkattavat kielet olivat useissa tilanteissa kilpailukykyisiä, sillä serverless-funktiot ovat usein kohtuullisen lyhytkestoisia. Tulkattavat kielet olivat erityisen tehokkaita tilanteissa, joissa tapahtui funktion kylmäkäynnistyminen. Samaan aikaan kustannusanalyysi osoitti, että vaikka käännettävien kielten resurssitehokkuus voi tuottaa säästöjä raskailla työkuormilla ja laskentaintensiivisissä funktioissa, niin silti kehitys- ja ylläpitokustannukset mitätöivät tämän hyödyn useimmissa tapauksissa. Tästä syystä tulkattavien kielten tarjoama parempi tuottavuus ja helppokäyttöisyys tarjoavat useimmissa tapauksissa taloudellisesti kannattavamman kokonaisratkaisun.

Avainsanat: serverless, FaaS, pilvipalvelut, ohjelmointikielet, suorituskyky, kustannukset

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

ABSTRACT

Lasse Vuorinen: The Popularity of Interpreted Programming Languages in Serverless Cloud Environments: Reasons for Use and Cost Implications
Master of Science Thesis
Tampere University
Master's Programme in Information Technology
December 2025

Serverless architecture has become increasingly popular in cloud-based applications, as it shifts the maintenance and scaling of servers entirely to the responsibility of cloud service providers. This event-driven and automatically scalable model allows organizations to concentrate purely on implementing business logic. Interpreted languages like Python and Node.js have emerged as the most popular languages in serverless environments, even though compiled programming languages such as Go and Java theoretically possess better computational efficiency.

The objective of this thesis was to investigate the reasons behind the extensive popularity of interpreted programming languages in serverless environments. Furthermore, the work examined the direct and indirect impacts of the chosen programming language on the total costs and performance of applications. The research was conducted as a literature review, with material collected from scientific publications, conference publications, and the technical documentation of key cloud service providers (AWS, Microsoft Azure, and Google Cloud). When selecting the sources, emphasis was placed on publications from the years 2018–2025 to ensure the overview was as up to date as possible.

The analysis focused specifically on how language runtimes affect cold start latencies and pricing models. Cold start, which is typical for serverless environments, causes latency when a function is executed for the first time or after a longer pause. The research indicates that interpreted languages are competitive in this regard and often more efficient than heavier compiled languages, which reduces application latency. Since serverless functions are often short-lived, the faster startup time of interpreted languages compensates for their slower execution speed.

The study indicates that the choice of interpreted languages is primarily a decision that is related to development efficiency and strategic agility. Their popularity is explained by a low learning curve and extensive library ecosystems, which enable faster application development and quicker deployment to production. Interpreted languages were competitive in many situations, as serverless functions are often relatively short-lived. Interpreted languages were particularly efficient in situations where a function's cold start occurred. At the same time, the cost analysis showed that although the better resource efficiency of compiled languages can yield savings in heavy workloads and compute-intensive functions, development and maintenance costs often invalidate this benefit in most cases. For this reason, the better productivity and ease of use offered by interpreted languages provide a more economically viable overall solution in most instances.

Keywords: serverless, FaaS, cloud services, programming languages, performance, costs

The originality of this thesis has been checked using the Turnitin Originality Check service.

TEKOÄLYN KÄYTTÖ OPINNÄYTTEESSÄ

Opinnäytteessäni on käytetty tekoälysovelluksia:

☐ Ei
☒ Kyllä

Ilmoitukseni mukaan olen käyttänyt opinnäytteessäni tutkielmaprosessin aikana seuraavia tekoälysovelluksia:

Tekoälysovellusten nimet ja versiot:

- ChatGPT (GPT-4o ja GPT-5)
- ScholarGPT

Käyttötarkoitus: ChatGPT:tä on käytetty työn rakenteen hahmottelemiseen sekä lähdemateriaalin tiivistämiseen. Sen avulla on myös muotoiltu lauserakenteita ja käännetty sanastoa englannista suomeksi. ScholarGPT on auttanut lähdekirjallisuuden löytämisessä sekä tiivistämällä niistä oleellisimmat käsitteet.

Osiot, joissa tekoälyä on käytetty: ScholarGPT:tä käytettäessä sitä pyydettiin etsimään työhön sopivia lähteitä IEEE Xplore-, ACM Digital Library- ja arXiv-tietokannoista. Se sai ottaa mukaan lähteitä myös noiden tietokantojen ulkopuolelta, jos löydetty lähde oli aiheeseen hyvin sopiva. Löydetyistä lähteistä muodostettiin ScholarGPT:n avulla tiivistelmät, joiden avulla tarkasteltiin lähteiden sopivuutta ennen tarkempaa tarkastelua ja läpilukua. ScholarGPT:n ehdottamia lähteitä on käytetty työn luvuissa 1–6. ChatGPT:n avustuksella tuotettuja suomennoksia ja lauserakenteiden parannuksia on käytetty läpi tutkielman. Lisäksi ChatGPT:tä on käytetty diplomityön kielentarkistukseen kieliopin ja oikeinkirjoituksen osalta.

Olen tietoinen siitä, että olen täysin vastuussa koko opinnäytteeni sisällöstä, mukaan lukien osat, joissa on hyödynnetty tekoälyä, ja hyväksyn vastuun mahdollisista eettisten ohjeiden rikkomuksista.

ALKUSANAT

Tämän työn aiheen valintaan vaikutti mielenkiintoni pilvipalveluita ja erityisesti AWS:ää kohtaan. Kiitos ohjaajalleni Kari Syställe tuesta työn tekemisessä. Hänen neuvonsa ja ohjeensa helpottivat työn etenemistä. Lisäksi haluan myös kiittää kumppaniani Lindaa tukemisesta ja positiivisesta patistamisesta.

Tampereella, 04.12.2025

Lasse Vuorinen

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. SERVERLESS-ARKKITEHTUURI 3
2.1 Serverless-arkkitehtuurin määritelmä ja periaatteet 3
2.2 Serverless-arkkitehtuurin hyödyt 5
2.3 Serverless-arkkitehtuurin haasteet 6
2.4 Suosituimmat serverless-palveluntarjoajat ja niiden erot 9
2.4.1 Amazon Web Services (AWS) Lambda 9
2.4.2 Microsoft Azure Functions 11
2.4.3 Google Cloud Functions 13
2.4.4 Palveluntarjoajien erot 14
3. TULKATTAVAT OHJELMOINTIKIELET JA NIIDEN KÄYTTÖ SERVERLESS-YMPÄRISTÖISSÄ 18
3.1 Tulkattavan ja käännettävän kielen erot 18
3.2 Staattinen vs. dynaaminen tyypitys 19
3.3 Yleisimmät tulkattavat kielet serverless-palveluissa 20
4. KÄYTTÖSYYT JA TEKNISET VALINNAT 22
4.1 Ohjelmistokehittäjien näkökulma 22
4.1.1 Ohjelmistokehittäjien mieltymykset ja kokemukset 22
4.1.2 Kehitysnopeus ja tuottavuus 23
4.1.3 Virheenkorjaus- ja testausominaisuudet 24
4.2 Ekosysteemien ja kirjastojen vaikutus 25
4.3 Suorituskyky- ja skaalautuvuustekijät 26
4.3.1 Kylmäkäynnistyksen tehokkuus 26
4.3.2 Suorituskykyominaisuudet ajonaikaisissa ympäristöissä 28
5. KUSTANNUSVAIKUTUKSET 30
5.1 Hinnoittelumallit serverless-palveluissa 30
5.1.1 Suoritusajan laskutus 30
5.1.2 Muistin allokoinnin hinnoittelu 31
5.1.3 Pyyntö-/kutsumaksu 33
5.1.4 Tiedonsiirtokustannukset 34
5.2 Kustannuserot tulkattavien ja käännettävien kielten välillä 35
5.2.1 Suoritusajan kustannusvertailu 35
5.2.2 Muistin käyttöaste ja kustannukset 37
5.2.3 Kehityksen ja ylläpidon kustannustekijät 38
5.2.4 Kielivalintojen taloudelliset kokonaisvaikutukset 39
6. KIRJALLISUUSKATSAUKSEN TULOKSET 41
7. JOHTOPÄÄTÖKSET 43
7.1 Keskeiset havainnot 43
7.2 Ohjelmointikielen valinnan vaikutukset suorituskykyyn ja kustannuksiin 43

7.3 Jatkotutkimusaiheet 44
LÄHTEET 45

1. JOHDANTO

Serverless-arkkitehtuuri on nykypäivänä erittäin suosittua pilviympäristöissä, sillä sen ansiosta kehittäjät pystyvät keskittymään ohjelmistojen kehittämiseen ilman tarvetta hallinnoida palvelininfrastruktuuria. Lisäksi serverless-pilviympäristöissä käyttäjälle kohdistuvat kustannukset perustuvat käytettyyn suorituskykyyn sekä pilvipalvelun suoritusaikaan, mikä lisää niiden houkuttavuutta.

Serverless-pilviympäristöissä käytetään sekä käännettäviä että tulkattavia ohjelmointikieliä. Erityisesti tulkattavat ohjelmointikielet, kuten Python, JavaScript (Node.js) ja Ruby ovat suosittuja serverless-ympäristöissä. Tämä voi johtua tulkattavien ohjelmointikielten helppokäyttöisyydestä ja siksi ne mahdollistavat nopean kehitystyön. Tulkattavilla ohjelmointikielillä on myös usein laajat kirjastoekosysteemit, jotka tukevat nopeaa kehitystyötä. Käännettävät ohjelmointikielet, kuten Go, Java tai C++, ovat puolestaan suorituskykyisempiä ja luotettavampia. Tulkattavilla ohjelmointikielillä suorituskyky voi olla huomattavasti heikompi kuin käännettävillä ohjelmointikielillä. Tämän takia erityisesti paljon laskentatehoa vaativissa tilanteissa kustannuseroja voi muodostua eri ohjelmointikielten välille.

Tässä työssä pyritään selvittämään syitä tulkattavien ohjelmointikielten suosion taustalla serverless-pilviympäristöissä. Selvitys on tärkeää, koska oikeanlaisella ohjelmointikielen valinnalla pystytään optimoimaan serverless-ympäristön suorituskyky ja kustannukset.

Työhön liittyy seuraavat tutkimuskysymykset:

- Mihin tulkattavien ohjelmointikielten suosio serverless-pilvipalveluissa perustuu?
- Miten käännettävien ja tulkattavien ohjelmointikielten tehokkuuserot vaikuttavat pilvipalveluiden kustannuksiin?

Työn tutkimusmenetelmänä käytettiin kirjallisuuskatsausta, jonka tavoitteena oli muodostaa mahdollisimman ajantasainen ja kokonaisvaltainen kuva tulkattavien ohjelmointikielten käytöstä serverless-ympäristöissä sekä ohjelmointikielen valintaan liittyvistä kustannusvaikutuksista. Kirjallisuuskatsaus toteutettiin keräämällä, vertailemalla ja analysoimalla sekä akateemisia julkaisuja että keskeisten pilvipalveluntarjoajien teknistä dokumentaatiota.

Lähteitä haettiin erityisesti tieteellisistä tietokannoista, kuten IEEE Xplore ja ACM Digital Library, sekä arXiv-palvelusta ja alan konferenssijulkaisuista. Lähteiden valintakriteereinä olivat julkaisuajankohta (ensisijaisesti vuodet 2018–2025), aiheen sopiminen serverless-arkkitehtuuriin, ohjelmointikielten suorituskykyyn tai kustannusmalleihin, sekä lähteen tieteellinen tai tekninen uskottavuus. Lisäksi huomioitiin lähteet, jotka esittävät vertailuja eri ohjelmointikielten suoriutumisesta serverless-ympäristöissä, analysoivat serverless-palveluiden hinnoittelumalleja tai kartoittavat serverlessin kehitystrendejä.

Lähdeaineistosta muodostettiin kokonaiskuva vertailemalla eri tutkimusten tuloksia ja etsimällä yhteneväisyyksiä ja eroavaisuuksia tulkattavien ja käännettävien ohjelmointikielten käytössä. Aineistoa ei valittu tukemaan ennalta asetettua johtopäätöstä, vaan jokaisen lähteen sisältö analysoitiin erikseen ja tämän pohjalta muodostettiin perustellut johtopäätökset.

2. SERVERLESS-ARKKITEHTUURI

Serverless-arkkitehtuuri on yksi merkittävimmistä muutoksista pilvipalveluiden kehityksessä ja se muuttaa tapaa, jolla sovelluksia suunnitellaan, toteutetaan ja ylläpidetään. Sen perusajatus on siirtää palvelininfrastruktuurin hallinta kehittäjältä pilvipalveluntarjoajalle. Kaikki resurssien hallinta ja skaalautuminen tapahtuu automaattisesti taustalla ilman kehittäjää. Tässä luvussa käsitellään serverless-arkkitehtuurin määritelmää, millaisia hyötyjä ja haasteita siihen liittyy ja miten eri palveluntarjoajien ratkaisut eroavat toisistaan.

2.1 Serverless-arkkitehtuurin määritelmä ja periaatteet

Serverless-arkkitehtuuri on pilvipalveluiden kehitysmalli, jossa sovellukset ja palvelut suoritetaan ilman, että kehittäjien tarvitsee hallita palvelininfrastruktuuria. Tämä tarkoittaa, että pilvipalveluntarjoaja huolehtii automaattisesti resurssien allokoinnista, skaalauksesta ja palvelinten ylläpidosta, jolloin kehittäjät voivat keskittyä sovelluskoodin kehittämiseen ja liiketoimintalogiikan optimointiin. [1] Serverless-laskenta on osa laajempaa pilvilaskennan kehitystä ja se voidaan nähdä jatkumona mikropalveluarkkitehtuurille. Mikropalveluarkkitehtuurissa sovellukset jaetaan pieniin, itsenäisiin osiin, jotka voidaan suorittaa erikseen.

Serverless-arkkitehtuuri koostuu pohjimmiltaan kahdesta perusmallista: Function-as-a-Service (FaaS) ja Backend-as-a-Service (BaaS). FaaS mahdollistaa yksittäisten funktioiden suorittamisen pilvessä ilman, että kehittäjien tarvitsee huolehtia palvelimista. BaaS puolestaan tarjoaa valmiita taustapalveluita, kuten tietokantoja, todennusta ja viestinvälitystä, joita voidaan käyttää ilman, että tarvitsee ylläpitää taustalla olevaa infrastruktuuria. [2] Kun sekä FaaS- että BaaS-komponentit yhdistetään, kehittäjät voivat luoda skaalautuvia ja tapahtumapohjaisia sovelluksia pienemmillä käyttökustannuksilla.

Serverless-laskenta muodostuu seuraavista ominaisuuksista, jotka määrittävät sen toimintamallin ja erot muista pilvipalvelumalleista.

1. Palvelimia ei tarvitse itse ylläpitää
2. Tapahtumalähtöisyys
3. Tilattomuus
4. Automaattinen skaalautuminen
5. Maksu käytön mukaan

Tapahtumalähtöinen suoritusmalli on yksi serverless-laskennan perusominaisuuksista. Siinä funktiot käynnistyvät vain tarvittaessa. Funktiot voivat käynnistyä monenlaisista eri tapahtumista, kuten esimerkiksi HTTP-pyynnöistä, tietokantamuutoksista, viestijonojen vastaanottamisesta tai ajastetuista tehtävistä [3]. Tapahtumapohjaisuus tekee serverless-laskennasta erittäin resurssitehokkaan. Suorituskapasiteettia käytetään vain silloin, kun on tarve. Näin ollen vältetään joutoaikaa ja siihen liittyvät kustannukset. Monimutkaisissa järjestelmissä, joissa monet eri komponentit kommunikoivat keskenään, tapahtumapohjaisuus lisää joustavuutta. Esimerkiksi yksi tapahtuma voi laukaista monta erillistä serverless-funktiota.

Toinen serverless-laskennan perusominaisuus on tilattomuus. Tilattomuus tarkoittaa, että serverless-funktiot eivät tallenna tilaansa muistiin suoritusistuntojen välillä, vaan jokainen funktiokutsu on riippumaton aiemmista funktiokutsuista. Näin ollen serverless-funktio ei ole tietoinen tilan muutoksista, vaan kaikki serverless-funktion tarvitsemat tiedot sisältyvät joko kutsuun tai ne on haettava ulkoisesta tietolähteestä, kuten tietokannasta tai välimuistista. Serverless-funktioissa tilattomuus on tärkeää, koska FaaS-alustat eivät takaa, että serverless-funktiot suoritettaisiin joka kerta samassa instanssissa, vaan serverless-funktioiden suorittamiseen käytettävät instanssit voivat vaihdella eri suorituskertojen välillä. Tämän takia serverless-funktioiden ei olisi edes mahdollista säilyttää tilojaan muistissa suoritusistuntojen välillä. [4]

Seuraava serverless-laskennan tyypillinen ominaisuus on laskentaresurssien automaattinen skaalautuvuus käytön mukaan. Serverless-alustojen automaattinen skaalautuminen on yleensä reaktiivista, eli ne seuraavat käynnissä olevien funktioinstanssien resurssien kapasiteettia ja vertaavat sitä tarvittavaan resurssimäärään. Jos tarvittava resurssien määrä nousee suuremmaksi kuin käynnissä olevat instanssit mahdollistavat, serverless-alusta käynnistää automaattisesti riittävän määrän uusia serverless-funktioinstansseja. Jos taas tarvittavien resurssien määrä laskee, serverless-alusta sulkee automaattisesti käyttämättömät instanssit. [5]

Serverless-alustoille on tyypillistä, että niissä hinta muodostuu pay-as-you-go -mallia noudattaen vain käytön mukaan. Usein hinta perustuu suoritettuihin funktiokutsuihin, niiden käyttämään suoritusaikaan ja käytettyyn muistiin. Serverless-arkkitehtuurista muodostuu houkutteleva vaihtoehto perinteisille pilvipalvelumalleille, sillä pilvipalveluntarjoaja vastaa palvelinten ylläpidosta, tietoturvapäivityksistä ja kuormanhallinnasta.

2.2 Serverless-arkkitehtuurin hyödyt

Serverless-arkkitehtuurin käyttö tarjoaa monia etuja verrattuna perinteisiin pilvi- ja palvelinratkaisuihin. Tarkasteltaessa serverless-arkkitehtuurin sopivuutta tiettyyn käyttötarkoitukseen, on tärkeää ymmärtää sen tuomat edut ja haasteet.

Yksi serverless-laskennan suurista hyödyistä on sen perusominaisuus, automaattinen skaalautuvuus. Serverless-laskennan automaattinen skaalautuvuus ja elastisuus eroavat huomattavasti perinteisistä pilvi- ja palvelinratkaisuista. Perinteisesti skaalaaminen on manuaalinen ja paljon resursseja kuluttava prosessi, johon voi sisältyä resurssien varaamista, konfigurointia ja jatkuvaa hallintaa. Se johtaa usein resurssien varaamiseen huippukysynnän mukaan, minkä takia resursseja on varattuna liikaa hiljaisempina aikoina. Serverless-alustat sen sijaan säätävät dynaamisesti ja älykkäästi taustalla olevaa infrastruktuuria reaaliajassa, reagoiden kysynnän vaihteluihin reaktiivisesti. Serverless-alustat allokoivat lisää laskentaresursseja pyyntömäärien kasvaessa ja vähentävät laskentaresursseja aktiivisuuden vähentyessä. Tällaisella tapahtumapohjaisella horisontaalisella skaalautumismekanismilla serverless-alustat optimoivat resurssien käytön ja pystyvät varmistumaan resurssien riittävyydestä lähes kaiken kokoisella kuormituksella.

Perinteisissä pilvi- ja palvelinratkaisuissa käyttäjät maksavat tyypillisesti varatuista resursseista, kuten virtuaalikoneista tai konteista, riippumatta siitä kuinka paljon niitä todellisuudessa käytetään. Vaikka sovellukset olisivat käyttämättöminä, taustalla olevasta infrastruktuurista maksetaan silti varattujen resurssien mukaiset kustannukset. Perinteisten pilvi- ja palvelinratkaisujen skaalaaminen lisää kustannuksia, koska resursseja pitää varata lisää ja niitä täytyy konfiguroida ja ylläpitää. Joskus voi olla jopa tarve hankkia lisää fyysisiä servereitä. Serverless-laskennassa käyttäjiltä veloitetaan vain funktioiden suorittamiseen kuluneista resursseista. Automaattisesti skaalautuvassa serverless-laskennassa maksetaan vain tarvittavien resurssien verran.

Skaalautuvuus tuo kustannuseroja serverless-laskennan ja pilvi- ja palvelinratkaisuiden välille. Se ei kuitenkaan suoraan tarkoita sitä, että serverless-laskennan kokonaiskustannukset olisivat jokaisessa tilanteessa pienemmät. Serverless-laskenta soveltuu erityisen hyvin tilanteisiin, joissa työkuormat ovat ajoittaisia tai arvaamattomia. Tilanteissa, joissa kuormitus on hiljaista tai olematonta, serverless-laskennan kustannukset voivat lähestyä nollaa, koska laskentaresursseja ei kuluteta. Jos taas sovelluksen työkuormat ovat jatkuvasti korkeat ja ennustettavat, serverless-laskennan kustannukset voivat nousta korkeammiksi, kuin perinteisten pilvi- ja palvelinratkaisuiden kustannukset. Näin ollen huolellinen tapauskohtainen kustannusanalyysi on tärkeää etsittäessä

kustannustehokkainta ratkaisua. Sen lisäksi serverless-laskennassa funktioiden suoritusaikojen ja muistivarausten optimointi voi parantaa sen kustannustehokkuutta entisestään.

Serverless-arkkitehtuurin tilattomuus mahdollistaa serverless-palveluiden hajauttamisen. Hajauttamisella tarkoitetaan serverless-palveluiden sijoittamista maantieteellisesti eri sijainteihin. Hajauttamisen avulla pystytään parantamaan vikasietoisuutta, skaalautuvuutta ja suorituskykyä.

Vikasietoisuus paranee hajauttamisella monella eri tavalla. Jos ohjelman käyttämät palvelimet on sijoitettu pelkästään yhteen sijaintiin, voi koko ohjelma kaatua, jos kyseinen datakeskus kokee jonkin ulkoisen häiriön, esimerkiksi sähkökatkon, verkkohäiriön tai luonnonkatastrofin. Tämä riski pienenee huomattavasti hajauttamalla ohjelman serverless-palvelut useisiin maantieteellisiin sijainteihin. Jos yksi käytetyistä sijainneista epäonnistuu, niin muut datakeskukset jatkavat ja ohjelma pystyy jatkamaan toimintaansa. Serverless-alustat tarjoavat myös mekanismeja, jotka ohjaavat vikatilanteissa liikenteen automaattisesti eri sijainneissa toimiviin instansseihin.

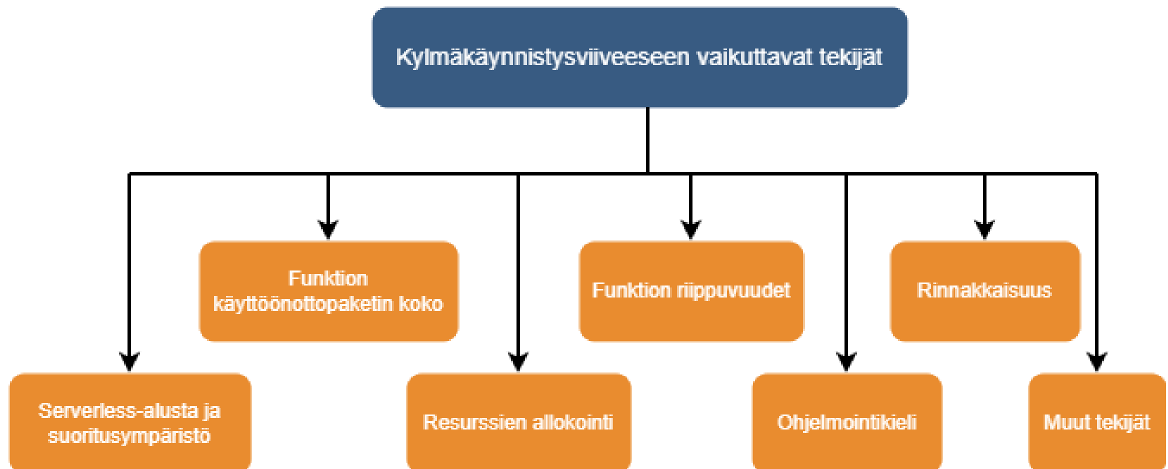
Sovellusten, joilla on käyttäjiä ympäri maailmaa, suorituskyky ja vasteajat paranevat, kun niiden osia sijoitetaan maantieteellisesti lähemmäs käyttäjiä. Käyttäjän pyynnöt ohjataan sitä lähimpänä olevaan instanssiin, minkä seurauksena verkkomatka lyhenee ja vasteajat nopeutuvat. Kuorman jakaminen useisiin sijainteihin mahdollistaa kuorman jakautumisen tasaisemmin eri infrastruktuurin osien kesken, mikä voi parantaa suorituskykyä ja skaalautuvuutta.

2.3 Serverless-arkkitehtuurin haasteet

Serverless-arkkitehtuuri ratkaisee monia ongelmia, mutta siihen liittyy myös useita haasteita. Nämä haasteet voivat estää serverless-arkkitehtuurin käytön joissakin käyttötapauksissa, tai ne on ainakin otettava huomioon sitä käytettäessä.

Eräs haasteista on latenssia kasvattava kylmäkäynnistys (cold start). Kylmäkäynnistys on ilmiö, joka syntyy, kun serverless-funktio suoritetaan ensimmäistä kertaa tai pitkän käyttämättömyysjakson jälkeen. Serverless-funktion käynnistys aiheuttaa merkittävän viiveen, koska serverless-alusta joutuu allokoimaan resurssit, käynnistämään tarvittavat kontit, alustamaan suoritusympäristön ja lataamaan tarvittavan koodin ennen suoritusta. [6] Kylmäkäynnistyksen kesto voi vaihdella useiden tekijöiden mukaan, kuten ohjelmointikielen suoritusympäristön, funktion koon, funktion riippuvuuksien ja pilvipalveluntarjoajan alustan (Kuva 1). Kylmäkäynnistys on hitaampi esimerkiksi Java- ja .NET-ohjelmointikielillä kuin Go- tai Rust-ohjelmointikielillä, koska niiden käynnistysajat ovat

pidempiä. Kylmäkäynnistyksen viive voi vaihdella millisekunneista jopa useisiin sekunteihin [6]. Kylmäkäynnistys on pääsääntöisesti ongelmallista lähinnä latenssiherkille sovelluksille, kuten esimerkiksi reaaliaikaisille API:lle tai interaktiivisille verkkosovelluksille.



Kuva 1. *Kylmäkäynnistysviiveeseen vaikuttavat tekijät [7]*

On kuitenkin tärkeää ymmärtää, että kylmäkäynnistys koetaan normaalisti vain serverless-funktion instanssin ensimmäisellä suorituskerralla. Saman funktioinstanssin myöhemmät suorituskerrat ovat yleensä paljon nopeampia, koska suoritusympäristö on jo valmiiksi alustettu ja valmis suoritettavaksi. Näitä myöhempiä suorituskertoja kutsutaan lämpimiksi käynnistyksiksi. Uusi serverless-funktion instanssi luodaan normaalisti silloin, kun funktio käynnistetään ensimmäisen kerran, funktiota ei ole käytetty pitkään aikaa, jolloin sen instanssi on poistettu käytöstä tai kuormituksen kasvaessa tarvitaan lisää instansseja kaikkien pyyntöjen toteuttamiseen. [6]

Kylmäkäynnistyksen tuottaman viiveen lieventämiseksi on kehitetty useita ratkaisuja. Yksi ratkaisu on serverless-funktioiden esilämmitystekniikat: manuaalinen esilämmitys, Provisioned Concurrency ja AI-pohjainen ennakointi. Manuaalisessa esilämmityksessä funktiota kutsutaan säännöllisesti, jotta se pysyy aktiivisena. Provisioned Concurrency on AWS Lambdan tarjoama ominaisuus, jossa tietty määrä instansseja pidetään aina valmiustilassa. AI-pohjaisessa ennakoinnissa tekoälyllä voidaan analysoida funktion käyttöä ja pyrkiä aktivoimaan se ennakkoon. [6]

Kylmäkäynnistyksen viiveen vähentäminen voi onnistua erilaisilla optimointitekniikoilla. Optimoimalla konttien kokoa kevyemmäksi voidaan vähentää käynnistysviivettä. Lisäksi nopeampien ohjelmointikielten kuten Go ja Rust käyttämisellä pystytään vaikuttamaan

viiveeseen. Näiden lisäksi voidaan optimoida funktion koodin rakennetta. Vähentämällä koodista tarpeettomia riippuvuuksia, sen latausaika pienenee. [6]

Yksi serverless-arkkitehtuurin haaste on palveluntarjoajaan lukkiutuminen eli Vendor lock-in. Palveluntarjoajaan lukkiutuminen on ilmiö, joka ilmenee, kun yritys on liian riippuvainen yhdestä tietystä pilvipalveluntarjoajasta niiden palveluiden tai ympäristöjen suhteen ja käyttää kaikkeen vain kyseisen toimittajan palveluita. Tämä voi muodostua ongelmaksi, mikäli sovellus halutaan myöhemmin siirtää osittain tai kokonaan pois kyseisen palveluntarjoajan palveluista jollekin toiselle toimittajalle. Serverless-sovelluksen siirtämisestä voi tulla monimutkainen ja aikaa vievä projekti, koska jokaisella pilvipalveluntarjoajalla on omat ainutlaatuiset integraationsa, määrittelytapansa ja palvelukohtaiset ominaisuutensa. Vaikka serverless vapauttaa kehittäjän palvelimien hallinnasta, sovelluksen ympärille tarvitaan silti runsaasti määrityksiä, kuten triggereitä, API-konfiguraatioita, käyttöoikeuksia ja muita taustapalveluihin liittyviä asetuksia. Nämä määritellään tyypillisesti infrastruktuurin määrittelyyn käytettävillä IaC-työkaluilla, jotka ovat usein sidoksissa yksittäisen palveluntarjoajan ekosysteemiin. Näin ollen siirtäminen voi vaatia osittaista sovelluksen koodin ja IaC-koodin uudelleenkirjoittamista, sekä sovelluksen uudelleenkonfiguroimista. Palveluntarjoajaa voidaan haluta vaihtaa esimerkiksi hintojen muuttuessa ja tämän seurauksena kustannuksen kasvaessa, palveluntarjoajan tietoturvan vaarantuessa, palveluntarjoajan palveluiden laadun heikentyessä tai käytetyn palvelun tuen loppuessa. Palveluntarjoajaan lukkiutumisen ongelmia voidaan lieventää toteuttamalla sovelluskoodi abstraktiotasolla niin, ettei se ole suoraan riippuvainen palveluntarjoajan serverless-rajapinnoista. Lisäksi sovelluksen eri komponenttien sijoittelu useille eri palveluntarjoajille eli multicloud-lähestymistapa vähentää palveluntarjoajaan lukkiutumista.

Serverless-arkkitehtuurin hajautetun luonteen ja tilattomuuden takia sen virheiden käsittely ja niiden lokitus voi olla joskus hankalaa ja erilaista perinteisiin palvelimiin verrattuna. Koska serverless-sovellus koostuu usein useista erillisistä funktioista ja niiden toiminta on voinut hajautua moniin eri paikkoihin, sovelluksen lokit ja virheet leviävät moniin pieniin yksiköihin, joita voi olla vaikea seurata kokonaisuutena. Serverless-funktioiden tilattomuuden seurauksena virhelokia ei voi tallentaa muistiin, vaan kaikki lokitettava tieto on lähettävä ulkoiseen lokituspalveluun. Lokituspalvelut ovat erillisiä palveluita, joiden tarkoitus on helpottaa lokitusta ja seurantaa.

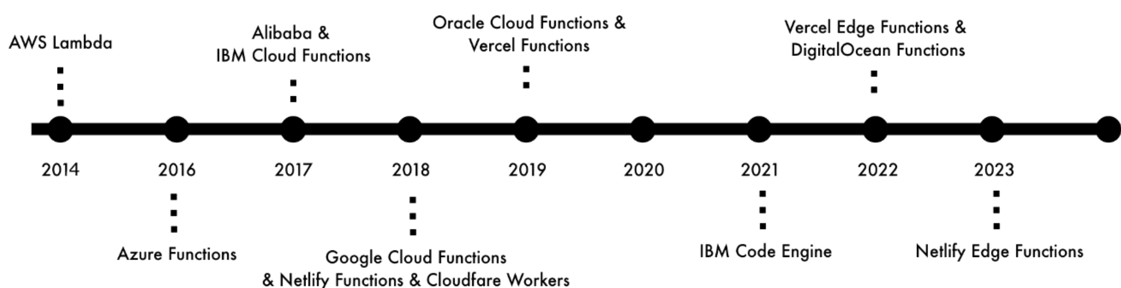
Vaikka serverless-arkkitehtuuri soveltuu moneen käyttötarkoitukseen, se ei silti ole universaali ratkaisu kaikkeen. Paljon pitkäkestoisia prosesseja tai intensiivistä laskentaa sisältäville sovelluksille serverless-arkkitehtuuri ei välttämättä ole paras mahdollinen ratkaisu. Myöskään HPC-sovellukset eli suurteholaskentasovellukset eivät sovellu hyvin

serverless-ympäristöön. HPC-sovellusten jatkuva tarve korkealle suorituskyvylle ja alhaiselle latenssille tekee niiden toteuttamisesta hankalaa serverless-ympäristössä.

2.4 Suosituimmat serverless-palveluntarjoajat ja niiden erot

Serverless-laskennan kenttää hallitsevat suurten pilvipalveluntarjoajien ratkaisut, joista jokaisella on omat erityiset ominaisuudet, vahvuudet ja rajoitteet. Näiden alustojen hyvä tuntemus on tärkeää, jotta voidaan tehdä hyvin perusteltuja päätöksiä serverless-arkkitehtuurin käyttöönotosta ja käytännön toteutuksesta.

Useat eri palveluntuottajat ovat tuoneet oman serverless-funktiototeutuksensa markkinoille. AWS Lambda ja Azure Functions olivat ensimmäiset julkaistut toteutukset. Kuvan 2 aikajana esittää eri serverless-funktioiden palveluntarjoajat ja niiden julkaisuvuodet. Työn seuraavissa osioissa perehdytään tarkemmin suosituimpien palveluntarjoajien serverless-funktioihin.



Kuva 2. *Eri palveluntarjoajien serverless-funktioiden julkaisuvuodet [8]*

2.4.1 Amazon Web Services (AWS) Lambda

AWS Lambda julkaistiin marraskuussa vuonna 2014 Amazonin re:Invent-konferenssissa Function-as-a-Service (FaaS) -mallin ensimmäisenä suurena toteutuksena, ja sitä pidetään laajasti tämän mallin edelläkävijänä [9]. Sen varhainen markkinoille tulo sekä jatkuva kehitystyö ovat tehneet siitä keskeisen ja laajasti käytetyn alustan serverless-laskennassa. Lambda on kiinteästi integroitunut osaksi laajempaa AWS-ekosysteemiä, mikä mahdollistaa sen hyödyntämisen monissa erilaisissa sovelluskonteksteissa ja on merkittävästi lisännyt sen suosiota [10].

AWS Lambda tukee monipuolisesti ohjelmointikieliä. Tällä hetkellä tarjolla on natiivisti useita suoritusympäristön (runtime) versioita Node.js:lle, Pythonille, Javalle, Rubylle sekä .NET 8:lle. Lisäksi alusta tukee myös muiden kielten, kuten Go:n, ajamista mukautettujen suoritusympäristöjen avulla tai vaihtoehtoisesti konttiratkaisuiden kautta [11] [12]. Tämä laaja ohjelmointikielten tuki ja mahdollisuus mukautettuun

suoritusympäristöön tekevät Lambdasta helposti lähestyttävän eri teknisistä taustoista tuleville kehittäjille ja tarjoavat joustavuutta moniin käyttötapauksiin.

AWS Lambdan suoritusmalli mahdollistaa funktioiden ajamisen enintään 15 minuutin ajan yhdellä kutsulla. Käyttäjät voivat määritellä muistiallokaatioita 128 megatavusta 10 gigatavuun asti yhden megatavun tarkkuudella. Suorituskyky skaalautuu muistin mukana, mikä tarkoittaa, että suurempi muistivaraus tuo käyttöön suhteessa enemmän laskentatehoa. [13] Lisäksi Lambdassa voidaan hyödyntää väliaikaista tallennustilaa (/tmp), jonka koko on oletuksena 512 MB, mutta se on käyttäjän konfiguroitavissa aina 10 GB asti [14]. Funktioiden käyttöönotto onnistuu joko zip-paketteina tai konttikuvina. Zip-paketin enimmäiskoko on 250 MB purettuna, kun taas konttikuvan enimmäiskoko voi olla jopa 10 GB [15]. Nämä rajat mahdollistavat varsin laajojenkin riippuvuuksien ja sovellusten suorittamisen, mutta voivat toisaalta asettaa rajoituksia tietyntyyppisten työkuormien toteuttamiselle.

Yksi AWS Lambdan keskeisimmistä vahvuuksista on sen laaja integraatio muiden AWS-palveluiden kanssa. Lambda-funktio voidaan laukaista yli 200 eri AWS-palvelun tai SaaS-sovelluksen tapahtumasta [16]. Tämä tekee mahdolliseksi hyvin monipuolisten, tapahtumapohjaisten sovellusten rakentamisen. Tyypillisiä esimerkkejä ovat API Gatewayn kautta vastaanotettavat HTTP-pyynnöt [17], Amazon S3 -palvelun tiedostojen käsittely [18], DynamoDB Streams -taulujen muutosoperaatioihin liittyvät käynnistimet [19], SQS-viestijonojen käsittely [20] sekä EventBridgen avulla toteutettu monimutkainen tapahtumien reititys [21]. Näiden integraatioiden ansiosta AWS Lambda soveltuu laajasti erilaisten järjestelmien ja sovellusten osaksi.

AWS on kehittänyt Lambda-alustan tueksi useita täydentäviä palveluita, joiden avulla voidaan hallita monimutkaisempia sovelluskokonaisuuksia. Step Functions mahdollistaa useiden Lambda-funktioiden orkestroinnin hallituiksi, tilallisiksi työnkuluiksi, joilla voidaan toteuttaa loogisesti jäsenneltyjä ja vikasietoisia prosesseja [22]. Lambda Layers tarjoaa mekanismin, jonka avulla yhteisiä kirjastoja ja koodikomponentteja voidaan jakaa useiden funktioiden kesken, mikä edistää koodin uudelleenkäyttöä ja modulaarisuutta [23]. Amazon API Gateway tarjoaa kattavat API-hallintaominaisuudet, kuten päätepisteiden määrittelyn, suojaamisen ja valvonnan Lambda-pohjaisille rajapinnoille [17]. Lisäksi Amazon CloudWatch tarjoaa integroidun ratkaisun Lambda-sovellusten seurantaan, sisältäen lokituksen, metriikat ja hälytysmekanismit [24].

AWS Lambdan hinnoittelumalli edustaa tyypillistä serverless-laskennan käyttöperusteista lähestymistapaa. Kustannukset muodostuvat funktiokutsujen määrästä, niiden suoritusajasta (gigatavusekunteina) sekä tiedonsiirrosta. AWS tarjoaa myös ilmaisen

aloitustason, johon sisältyy miljoona ilmaista funktiokutsua ja 400 000 gigatavusekuntia laskenta-aikaa kuukaudessa [25]. Tämä tekee Lambdasta houkuttelevan vaihtoehdon pienimuotoisiin projekteihin ja kokeiluihin.

AWS Lambdan laaja käyttäjäkunta ja pitkä kehityshistoria ovat edesauttaneet laajan ekosysteemin syntymistä. Kolmannen osapuolen työkalut, kuten Serverless Framework, sekä AWS:n omat työkalut, kuten Serverless Application Model (SAM) ja Cloud Development Kit (CDK), tarjoavat infrastruktuurin määrittelyn koodina ja helpottavat Lambdan käyttöönottoa. Niiden lisäksi kehittäjien tueksi on tarjolla kattava dokumentaatio, esimerkkisovelluksia sekä aktiivinen käyttäjäyhteisö, mikä madaltaa käyttöönoton kynnystä ja nopeuttaa kehitystyötä.

Vahvuuksista huolimatta Lambdalla on myös tiettyjä rajoituksia, jotka tulee huomioida arkkitehtuurisia päätöksiä tehtäessä. Yksi merkittävimmistä haasteista on kylmäkäynnistys (cold start). Se voi aiheuttaa huomattavia viiveitä erityisesti Java- ja .NET-pohjaisissa sovelluksissa [6]. AWS on kuitenkin pyrkinyt vähentämään tätä ongelmaa muun muassa SnapStart-ominaisuudella Javalle [26] sekä .NET 8:n Native AOT -tekniikalla, joka lyhentää käynnistysaikoja merkittävästi [27]. Myös Lambdan liittäminen VPC-resursseihin on aiemmin lisännyt kylmäkäynnistysten viivettä, mutta AWS paransi tilannetta vuonna 2019 julkaisemalla uuden arkkitehtuurin, jossa hyödynnetään ennakkoon luotuja verkkoliittymiä (ENI), mikä vähensi viivettä huomattavasti [28]. Lisäksi Lambdan 15 minuutin enimmäissuoritusaika rajoittaa pitkäkestoisten prosessien toteutusta ilman erillisiä arkkitehtuurisia ratkaisuja. Resurssirajoitteet, kuten muistin määrä, väliaikaisen tallennustilan kapasiteetti ja käyttöönotettavan paketin koko, voivat niin ikään vaikuttaa joidenkin työkuormien toteutettavuuteen [13][14] [15]. Näiden rajoitusten huomioiminen on tärkeää Lambda-pohjaisten sovellusten suunnittelussa, jotta toteutus on sekä teknisesti että taloudellisesti kestävä.

2.4.2 Microsoft Azure Functions

Microsoftin serverless-laskentaratkaisu Azure Functions julkaistiin AWS Lambdan jälkeen, mutta se on kehittynyt nopeasti monipuoliseksi ja kilpailukykyiseksi alustaksi. Azure Functions hyödyntää Microsoftin vahvaa asemaa yritysmarkkinoilla sekä .NET-ekosysteemin keskeisiä vahvuuksia [29].

Alusta tukee useita ohjelmointikieliä, kuten C#, JavaScript/TypeScript, Python, Java ja PowerShell [30]. Laaja kielivalikoima soveltuu monien erilaisten kehittäjien tarpeisiin, ja erityisen vahva tuki .NET-kielille on seurausta Microsoftin pitkäaikaisesta sitoutumisesta omaan teknologiaekosysteemiinsä [31]. Azure Functions -ympäristö tarjoaa

muistiallokaatioita 128 MB:sta aina 14 GB:iin asti [32]. Azure Functions tarjoaa useita erilaisia hosting-suunnitelmia, jotka tuovat joustavuutta erilaisiin käyttötarpeisiin. Kulutusperusteisessa (Consumption) suunnitelmassa funktioiden enimmäissuoritusaika on 10 minuuttia, mutta Premium- ja Dedicated-suunnitelmat mahdollistavat tätä pidemmät suoritukset, jopa 60 minuuttiin saakka [33].

Consumption-suunnitelma noudattaa klassista serverless-mallia, jossa skaalautuminen tapahtuu automaattisesti ja kustannukset määräytyvät todellisen käytön perusteella. Premium-suunnitelma lisää ominaisuuksia, kuten VNET-integraation, pidemmät suorituskestot ja esilämmitetyt instanssit, jotka vähentävät kylmäkäynnistyksistä johtuvia viiveitä [32]. Dedicated-suunnitelmassa funktiot suoritetaan pysyvästi varatuilla App Service -palvelinresursseilla, joita ei jaeta muiden käyttäjien kanssa. Tämä takaa ennustettavan suorituskyvyn ja kiinteän kustannustason, mikä on hyödyllistä sovelluksille, jotka vaativat jatkuvaa käytettävyyttä ja lyhyitä vasteaikoja [34]. Näiden eri suunnitelmien avulla organisaatiot voivat sovittaa serverless-laskentamallin tarkasti omiin teknisiin ja toiminnallisiin vaatimuksiinsa.

Yksi Azure Functionsin suurimmista vahvuuksista on sen tiivis integraatio muuhun Azure-ekosysteemiin. Käynnistimet ja sidokset mahdollistavat saumattoman yhteyden palveluihin, kuten Azure Storage, Azure Cosmos DB, Azure Service Bus ja Azure Event Grid [35]. Lisäksi Azure Logic Apps tarjoaa työnkulkujen orkestrointiominaisuuksia, jotka täydentävät Functionsia monimutkaisissa prosesseissa. Azure API Management tarjoaa API-yhdyskäytävätoiminnallisuudet Functions-pohjaisille rajapinnoille, mukaan lukien autentikointi, suojaus ja liikenteen hallinta [36]. Tämä vahva ekosysteeminen integraatio tuo erityistä lisäarvoa organisaatioille, jotka ovat jo rakentaneet ratkaisujaan Microsoftin pilvipalveluiden varaan.

Azure Functions sisältää myös laajennuksen nimeltä Durable Functions, jonka avulla voidaan toteuttaa tilallisia toimintoja ja orkestrointeja. Durable Functions mahdollistaa monimutkaisten työnkulkujen, pitkäkestoisten prosessien ja ihmisen vuorovaikutusta vaativien toimintamallien rakentamisen. Se tukee useita yleisiä toteutusmalleja, kuten funktioiden ketjutusta, rinnakkaista hajautusta ja kokoamista (fan-out/fan-in) sekä asynkronisia HTTP-rajapintoja. Tämä laajentaa serverless-arkkitehtuurin soveltamismahdollisuuksia huomattavasti ja tekee siitä käyttökelpoisen myös vaativissa liiketoimintaprosesseissa. [37]

Kehityskokemus Azure Functionsin kanssa on tiiviisti sidoksissa Microsoftin tarjoamiin työkaluihin. Visual Studio ja Visual Studio Code tarjoavat monipuoliset kehitysympäristöt sisäänrakennetuilla virheenkorjausominaisuuksilla [38]. GitHub Actions ja Azure DevOps

tarjoavat CI/CD-integraation, mikä mahdollistaa modernien ohjelmistokehitysprosessien toteuttamisen [39]. Lisäksi lokaaliin kehitykseen ja testaukseen tarkoitetut työkalut, kuten Azure Functions Core Tools, helpottavat siirtymistä kehitys- ja tuotantoympäristöjen välillä [40].

Azure Functionsin hinnoittelumalli noudattaa serverless-arkkitehtuurin kulutusperusteista laskutustapaa. Kulutussuunnitelmassa kustannukset määräytyvät funktiokutsujen määrän, niiden suoritusajan sekä käytetyn muistin perusteella. Ilmainen käyttö sisältää miljoona suoritusta ja 400 000 gigatavusekuntia kuukaudessa, mikä vastaa pitkälti AWS Lambdan tarjoamaa ilmaistasoa [41]. Premium-suunnitelma lisää kiinteitä kustannuksia varattujen instanssien ylläpidosta, mutta tarjoaa vastineeksi ennustettavamman suorituskyvyn ja lisäominaisuuksia [32].

Azure Functionsilla on myös rajoitteita, jotka tulee huomioida arkkitehtuurisia päätöksiä tehtäessä. Kulutussuunnitelman 10 minuutin enimmäissuoritusaika voi olla este pitkäkestoisille prosesseille, ja tällöin tarvitaan vaihtoehtoisia ratkaisuja, kuten Durable Functions -mallien hyödyntämistä [33]. Lisäksi muiden kuin Microsoftin teknologioiden integrointi voi olla monimutkaisempaa kuin joissain kilpailevissa alustoissa [42]. Näiden rajoitusten huolellinen arviointi suhteessa sovelluksen tarpeisiin ja olemassa olevaan teknologiaympäristöön on tärkeää suunnitteluvaiheessa.

2.4.3 Google Cloud Functions

Google Cloud Functions on Googlen tarjoama serverless-laskentaratkaisu, joka on suunniteltu korostamaan helppokäyttöisyyttä, skaalautuvuutta ja tiivistä integraatiota Googlen muihin pilvipalveluihin. Erityisesti tekoäly- ja datankäsittelyalustat, kuten BigQuery, Dataflow ja Vertex AI tekevät siitä houkuttelevan vaihtoehdon dataintensiivisiin sovelluksiin [29]. Vaikka Google tuli serverless-markkinoille AWS:n ja Microsoft Azuren jälkeen, se on onnistunut erottumaan kilpailijoista etenkin suurten tietomassojen käsittelyssä ja koneoppimisen tukemisessa [42].

Google Cloud Functions tukee useita ohjelmointikieliä, kuten Node.js, Python, Go, Java, Ruby, PHP ja .NET [43]. Suoritusympäristössä voidaan varata muistia 128 megatavusta 8 gigatavuun asti, ja varatun muistin määrään suhteutetaan automaattisesti myös käytettävissä oleva suorittimen määrä [44]. Vakiofunktioiden enimmäissuoritusaika on yhdeksän minuuttia [43]. Nämä rajat tekevät alustasta joustavan monenlaisiin käyttötapauksiin, mutta pitkäkestoiset laskennalliset prosessit saattavat edellyttää vaihtoehtoisia ratkaisuja.

Alustan suunnittelufilosofia painottaa yksinkertaisuutta. Google Cloud Functions pyrkii toteuttamaan yhden asian tehokkaasti: suorittamaan koodia ulkoisiin tapahtumiin reagoiden. Tämä lähestymistapa minimoi monimutkaisten infrastruktuurimääritysten ja konfiguraatioiden tarpeen [45]. Kehitystyötä tukevat myös valmiit integraatiot Google Cloud Buildin ja Cloud Source Repositoriesin kanssa, mikä nopeuttaa CI/CD-työnkulkuja ja käyttöönottoa.

Yksi Google Cloud Functionsin keskeisistä vahvuuksista on sen laajasti skaalautuva integraatio Googlen muihin pilvipalveluihin. Funktioita voidaan laukaista monista eri tapahtumista, kuten HTTP-pyynnöistä, Cloud Storage -objektien muutoksista, Pub/Sub-viesteistä sekä Firestore-tietokannan päivityksistä [46]. Tämä mahdollistaa tehokkaat tiedonkäsittelyn työnkulut, joissa hyödynnetään Googlen analytiikkapalveluita. Yhdessä BigQueryn, Dataflown ja Vertex AI:n kanssa Google Cloud Functions tarjoaa erityisen kilpailuedun sovelluksille, jotka tarvitsevat reaaliaikaista datan prosessointia tai älykästä päätöksentekoa.

Vuonna 2021 julkaistu toinen sukupolvi (Gen2) toi Google Cloud Functionsiin merkittäviä parannuksia. Gen2 pohjautuu Cloud Run -palvelun konttipohjaiseen infrastruktuuriin ja tarjoaa paremman suorituskyvyn, lyhyemmät käynnistysajat sekä aiempaa joustavamman skaalautuvuuden [47]. Integrointi Cloud Runiin mahdollistaa sen, että sovelluksia voidaan tarpeen mukaan siirtää tai laajentaa Cloud Functions -alustalta Cloud Runiin. Tämä on hyödyllistä erityisesti silloin, kun Cloud Functionsin rajoitukset, kuten yhdeksän minuutin maksimisuoritusaika tai tilattomuus, estäisivät sovelluksen toteutuksen. Cloud Run säilyttää serverless-arkkitehtuurin hyödyt, kuten automaattisen skaalautuvuuden ja hallinnoidun infrastruktuurin, mutta tarjoaa samalla laajempaa hallittavuutta konttipohjaisille sovelluksille.

Hinnoittelumalli noudattaa serverless-arkkitehtuurien vakiokäytäntöjä. Maksu perustuu kutsujen määrään, suoritusaikaan ja varattuun muistiin. Google tarjoaa ilmaisen tason, johon sisältyy 2 miljoonaa kutsua ja 400 000 gigatavusekuntia laskenta-aikaa kuukaudessa [48]. Tämä tekee alustasta erityisen houkuttelevan pienimuotoisiin sovelluksiin ja kokeiluihin, mutta samalla se skaalautuu myös suurten organisaatioiden tarpeisiin.

2.4.4 Palveluntarjoajien erot

Serverless-alustojen vertailussa on syytä tarkastella useita tekijöitä, jotka erottavat palveluntarjoajien tarjonnan toisistaan ja voivat merkittävästi vaikuttaa käyttöönoton tehokkuuteen, skaalautuvuuteen ja pitkän aikavälin ylläpidettävyyteen. Näitä tekijöitä ovat

kielituki, integrointiominaisuudet, suorituskykyominaisuudet, hinnoittelumallit ja ekosysteemin kypsyys.

Kielituki vaihtelee palveluntarjoajien välillä (Kuva 3). Kaikki kolme suurta alustaa, AWS Lambda, Azure Functions ja Google Cloud Functions, tukevat suosittuja kieliä kuten JavaScript/Node.js, Python ja Java [29]. AWS Lambda tarjoaa laajimman natiivin kielituen sekä mahdollisuuden ajaa käytännössä mitä tahansa ohjelmointikieltä mukautetun suoritusympäristön mallin avulla [49]. Google Cloud Functions tukee useita kieliä, mutta sillä ei ole vastaavanlaista mukautettujen suoritusympäristöjen käyttömahdollisuutta, mikä tekee sen kielivalikoimasta rajatumman [43]. Azure Functions tarjoaa kattavan kielituen ja on erityisesti optimoitu .NET-kielille, mikä tukee Microsoftin omaa ekosysteemiä ja on hyödyllistä organisaatioille, jotka hyödyntävät laajasti Microsoftin teknologioita [30]. Oikean ohjelmointikielen ja alustan yhteensopivuus vaikuttaa käytettävissä oleviin ominaisuuksiin, suorituskyvyn optimointiin ja kehittäjäkokemukseen.

	Alibaba	AWS	Azure	Cloudflare	Digital Ocean	Google	IBM	Netlify	Oracle	Vercel
Node.js	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Java	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗
Python	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓
.NET/C#	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗
Powershell	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
Ruby	✗	✓	✗	✗	✗	✓	✗	✗	✓	✓
Go	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
PHP	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗
Custom Runtime	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓

Kuva 3. *Eri palveluntarjoajien tukemat suoritusympäristöt [8]*

Integrointiominaisuudet ovat keskeinen erottava tekijä, sillä serverless-funktiot toimivat harvoin yksinään. AWS Lambda tarjoaa laajimmat integrointivaihtoehdot, ja se yhdistyy yli 200 AWS-palveluun sekä useisiin SaaS-sovelluksiin hyvin dokumentoiduilla laukaisimilla [13]. Azure Functions tarjoaa syvän integraation Microsoftin ekosysteemiin, mikä on arvokasta organisaatioille, jotka jo hyödyntävät Microsoftin palveluita kuten Azure Storagea, Cosmos DB:tä ja Service Busia [35]. Google Cloud Functions on vahvimmillaan Googlen data- ja tekoälypalveluihin integroitumisessa, kuten BigQueryyn, Dataflow'hun ja Vertex AI:hin [50]. Valittu alusta ja sen tarjoamat integraatiomahdollisuudet vaikuttavat merkittävästi siihen, millaiseksi sovelluksen arkkitehtuuri muodostuu.

Suorituskykyominaisuudet vaihtelevat palveluntarjoajien välillä erityisesti kylmäkäynnistyksen, enimmäissuorituskeston ja skaalautuvuuden osalta. AWS Lambda on ottanut

merkittäviä edistysaskeleita kylmäkäynnistysten hallinnassa erityisesti Node.js- ja Python-funktioiden kohdalla [10]. Azure Functions tarjoaa Premium-suunnitelmassaan esilämmitettyjä instansseja, joiden avulla voidaan minimoida kylmäkäynnistyksen viive lähes kokonaan [32]. Google Cloud Functions Gen2 on parantanut kylmäkäynnistysten suorituskykyä hyödyntämällä Cloud Run -pohjaista infrastruktuuria [47]. Enimmäissuoritusaika eroaa alustoittain: Google Cloud Functions rajoittaa funktion yhdeksään minuuttiin, AWS Lambda sallii 15 minuutin suorituksen ja Azure Functions tarjoaa Premium- ja Dedicated-suunnitelmissaan jopa 60 minuutin suorituksia [13][30][32][43]. Kaikki kolme alustaa tukevat automaattista horisontaalista skaalautuvuutta, mutta skaalausrajojen hallinta ja säätömahdollisuudet vaihtelevat alustoittain.

Hinnoittelumallit noudattavat samaa perusperiaatetta, sillä maksut määräytyvät kutsujen, suoritusaikojen ja muistin perusteella. Kaikki kolme palveluntarjoajaa tarjoavat laajat ilmaistasot, jotka mahdollistavat pienten sovellusten ja kokeilujen toteuttamisen ilman kustannuksia. AWS ja Azure tarjoavat miljoona ilmaista kutsua kuukaudessa, kun taas Google tarjoaa kaksi miljoonaa [25][41][51]. Erot tulevat esiin laskutustarkkuudessa ja minimiveloituksissa. AWS Lambda laskuttaa millisekunnin tarkkuudella ilman vähimmäisveloitusta [25]. Azure Functions laskuttaa millisekunnin tarkkuudella, mutta vähintään 100 millisekuntia per kutsu [41]. Google Cloud Functions puolestaan pyöristää suorituskeston ylöspäin 100 millisekunnin eriin [51]. Näillä eroilla voi olla merkitystä erityisesti sovelluksissa, joissa funktiot ovat hyvin lyhytkestoisia ja kutsumäärät suuria.

Ekosysteemin kypsyys vaihtelee palveluntarjoajien välillä. AWS Lambda hyötyy asemastaan markkinoiden ensimmäisenä FaaS-palveluna, ja sen ympärille on muodostunut laaja kolmansien osapuolten työkalujen, yhteisöresurssien ja parhaiden käytäntöjen ekosysteemi [42]. Azure Functions puolestaan hyödyntää Microsoftin kehittäjätyökaluja ja laajaa yritysasiakaskuntaa, mikä tekee siitä houkuttelevan erityisesti organisaatioille, joilla on jo olemassa olevia Microsoft-sidonnaisuuksia [30][35]. Google Cloud Functionsilla on suhteellisesti pienempi, mutta jatkuvasti kasvava ekosysteemi, jota tukee Googlen vahva panostus data- ja koneoppimisratkaisuihin [50].

Muita erottavia tekijöitä ovat kehitystyökalut, valvontaominaisuudet, tietoturvaominaisuudet ja vaatimustenmukaisuussertifikaatit. AWS tarjoaa kattavat työkalut esimerkiksi AWS SAM:n ja CloudFormationin kautta [52]. Azure Functions tarjoaa vahvan Visual Studio -integraation, mikä tekee siitä houkuttelevan .NET- ja C#-kehittäjille [38]. Google Cloud Functions puolestaan painottaa yksinkertaisuutta ja keveyttä kehitystyökalujen lähestymistavassa [35][50]. Valvontaominaisuuksissa on eroja: AWS:n CloudWatch tarjoaa tarkkaa lokitusta ja diagnostiikkaa, Azure Monitor integroituu laajasti muihin Azure-palveluihin ja Google Cloud tarjoaa integroidut työkalut Cloud Monitoringin kautta

[24][53][54]. Tietoturva- ja vaatimustenmukaisuusominaisuudet ovat kaikilla kolmella alustalla laajoja, mutta niiden painotukset ja sertifikaatit vaihtelevat alueittain ja asiakassegmenteittäin.

3. TULKATTAVAT OHJELMOINTIKIELET JA NIIDEN KÄYTTÖ SERVERLESS-YMPÄRISTÖISSÄ

Ohjelmointikieli on keskeinen osa serverless-sovelluksen suoritusympäristöä ja sillä on suora vaikutus sekä suorituskykyyn että kustannuksiin. Serverless-kehityksessä käytetään useita kieliä, mutta erityisesti tulkattavat kielet ovat saavuttaneet ison aseman. Niiden joustavuus, helppokäyttöisyys ja laajat kirjastoekosysteemit tekevät niistä houkuttelevan vaihtoehdon monenlaisille projekteille. Tässä luvussa tarkastellaan ensin kielten perustavia eroja ja niiden vaikutuksia serverless-ympäristöissä.

3.1 Tulkattavan ja käännettävän kielen erot

Ohjelmointikielen valinnalla on merkittävä vaikutus serverless-sovellusten kehitykseen ja toimintaan. Oikean kielen valinta vaikuttaa niin suorituskykyyn, kustannuksiin kuin kehitystyön tehokkuuteen. On siis tärkeää ymmärtää keskeiset erot käännettävien ja tulkattavien ohjelmointikielten välillä, jotta voidaan valita käyttötapaukseen parhaiten sopiva ratkaisu.

Käännetyt kielet, kuten Go, Rust ja C++, kääntävät lähdekoodin suoraan konekieleksi ennen suoritusta. Tämä tuottaa optimoituja binääritiedostoja, jotka voidaan ajaa tehokkaasti ilman erillistä tulkkia. Käännetyt kielet ovat usein suorituskyvyltään parempia, koska käännösvaiheessa voidaan tehdä laajoja optimointeja, kuten muistinhallinnan tehostamista ja koodin rakenteen analysointiin perustuvia parannuksia. Haittapuolena on käännösaikojen pituus ja monimutkaisempi build-prosessi, mikä voi hidastaa kehitystä erityisesti nopeissa kokeiluissa.

Tulkattavat kielet, kuten JavaScript, Python ja Ruby, suorittavat koodia riviltä riville ajonaikaisen tulkin kautta. Tämä tekee kehityksestä joustavampaa ja nopeampaa, koska käännösvaihetta ei tarvita ja muutokset voidaan testata välittömästi. Toisaalta tulkkipohjainen suoritus johtaa yleensä hitaampaan suoritukseen, koska koodi ei ole valmiiksi optimoitua konekieltä. Tulkattavat kielet hyötyvät kuitenkin usein laajoista kirjastoekosysteemeistä ja kehittyneistä ajonaikaisista optimoinneista, kuten just-in-time (JIT) -kääntäjistä, jotka parantavat suorituskykyä käytännössä merkittävästi [55].

Serverless-ympäristöissä ero käännettyjen ja tulkattavien kielten välillä ei ole yksiselitteinen. Artikkelissa ”Serverless performance comparison: Does the language matter?” havaittiin, että käännetyt kielet tarjoavat selvästi paremman suorituskyvyn

laskentaintensiivisissä funktioissa, joissa suuri osa suorituksesta on puhdasta laskentaa [56]. Tulkattavien kielten joustavuus ja kehitysnopeus voivat kuitenkin olla kokonaisuutena edullisempia erityisesti I/O-sidonnaisissa funktioissa, joissa verkkoviive hallitsee suoritusaikaa. Tällöin kielen raakasuorituskyvyllä on vähäisempi merkitys, sillä suurin osa ajasta kuluu ulkoisten palveluiden, kuten tietokantojen tai API-kutsujen odottamiseen [57].

3.2 Staattinen vs. dynaaminen tyypitys

Toinen merkittävä ero ohjelmointikielten välillä on niiden tyypitysjärjestelmä. Staattisesti tyypitetyissä kielissä, kuten Javassa, C++:ssa ja Go:ssa, tyyppitarkastus suoritetaan käännösaikana. Tämä tarkoittaa, että muuttujien ja funktioiden tyypit määritellään etukäteen, ja virheet havaitaan jo ennen ohjelman ajamista [58]. Staattinen tyypitys lisää luotettavuutta ja helpottaa virheiden löytämistä varhaisessa vaiheessa, mikä voi olla tärkeää erityisesti suurissa tai pitkäikäisissä serverless-järjestelmissä, joissa koodin ylläpito ja refaktorointi ovat toistuvia tehtäviä.

Dynaamisesti tyypitetyissä kielissä, kuten JavaScriptissä, Pythonissa ja Rubyssa, tyyppitarkastus suoritetaan vasta ajon aikana. Tämä tekee ohjelmoinnista joustavampaa ja mahdollistaa nopean kehityksen, koska tyyppejä ei tarvitse määritellä etukäteen [59]. Haittapuolena on, että virheitä voidaan havaita vasta ohjelman suorituksen aikana, mikä voi johtaa ajonaikaisiin poikkeuksiin. Serverless-ympäristöissä tämä voi olla ongelmallista, koska funktiot ovat usein osa hajautettuja ja monimutkaisia järjestelmiä, joissa virheen toistaminen ja debuggaaminen voi olla hankalaa.

Dynaaminen tyypitys voi kuitenkin nopeuttaa kehitystyötä merkittävästi, mikä on usein kriittistä erityisesti kokeiluluontoisissa serverless-projekteissa ja prototyypeissä. Tyypitykseen liittyviä riskejä voidaan hallita käyttämällä valinnaisia tyypitysjärjestelmiä. Python tukee tyyppivihjeitä (type hints), jotka voidaan tarkastaa staattisesti työkaluilla kuten mypy, ja JavaScriptin rinnalla voidaan käyttää TypeScriptiä, joka lisää siihen staattisen tyyppijärjestelmän [60]. Näiden avulla voidaan yhdistää dynaamisen tyypityksen kehitysnopeus ja staattisen tyypityksen tarjoama turvallisuus ja ennustettavuus.

Serverless-kontekstissa valinta staattisen ja dynaamisen tyypityksen välillä riippuu usein projektin luonteesta. Nopeat kokeilut, datankäsittely ja prototypointi hyötyvät dynaamisesta tyypityksestä. Laajat, pitkäikäiset ja monimutkaiset järjestelmät puolestaan saattavat hyötyä enemmän staattisesta tyypityksestä, joka vähentää virheiden riskiä ja parantaa ylläpidettävyyttä.

3.3 Yleisimmät tulkattavat kielet serverless-palveluissa

Serverless-arkkitehtuurien kehittyessä on käynyt selväksi, että muutama tulkattava ohjelmointikieli on saanut siinä vahvan aseman. Erityisesti Node.js ja Python ovat nousseet käytetyimmiksi vaihtoehdoiksi serverless-funktioiden toteutuksessa, ja ne muodostavat valtaosan kaikista julkisesti julkaistuista serverless-sovelluksista. Laajan GitHub-aineiston perusteella tehdyssä tutkimuksessa havaittiin, että Node.js- ja TypeScript-ympäristöt kattavat noin kolme neljäsosaa kaikista serverless-toteutuksista, ja Pythonin osuus on hieman yli 15 prosenttia. Muiden kielten, kuten Rubyn, PHP:n ja R:n käyttö jää selvästi vähäisemmäksi. [61]

JavaScriptin asemaa selittää ennen kaikkea sen pitkä historia web-kehityksen yleiskielenä. Sen ympärille on kasvanut laaja kehittäjäyhteisö ja valtava määrä valmiita kirjastoja, mikä tekee siirtymisen serverless-maailmaan luontevaksi. Node.js:n suorituskyky on osoittautunut yllättävänkin hyväksi ottaen huomioon sen tulkattavan luonteen. V8-moottorin tehokas just-in-time-käännös ja ei-estetty, tapahtumapohjainen arkkitehtuuri sopivat hyvin yhteen serverless-funktioiden rinnakkaisten pyyntöjen käsittelymallin kanssa. Lisäksi suurimmat pilvialustat, kuten AWS Lambda, Azure Functions ja Google Cloud Functions, ovat optimoineet Node.js-suoritusympäristönsä erityisesti nopeaa käynnistymistä ja suoritusta varten. Näiden ominaisuuksien ansiosta Node.js on säilyttänyt asemansa kehittäjien ensisijaisena valintana, etenkin kun se on monille jo valmiiksi tuttu osa web-sovellusten kehitystyötä.

Python on toinen merkittävä tulkattava kieli, jonka suosio serverless-ympäristöissä on kasvanut nopeasti. Sen vahvuuksia ovat laajat standardikirjastot, erinomainen tuki data-analytiikalle ja koneoppimiselle sekä helppo omaksuttavuus. Python soveltuu erityisen hyvin tilanteisiin, joissa serverless-funktio toimii osana laajempaa tietovirtaa tai datankäsittelyketjua. Kielen CPython-toteutus mahdollistaa myös tehokkaan yhteistoiminnan muiden kielten, kuten C:n ja C++:n, kanssa. Laskentaintensiiviset osat voidaan siten suorittaa natiivikoodissa ilman merkittävää suorituskyvyn heikkenemistä. Tämä tekee Pythonista monipuolisen valinnan: sitä voidaan käyttää sekä nopeaan prototypointiin että raskaampiin datankäsittelytehtäviin.

Vaikka JavaScript ja Python hallitsevat selvästi serverless-kenttää, myös muita tulkattavia kieliä esiintyy, joskin huomattavasti harvemmin. Ruby, PHP ja R ovat tuettuja joillakin alustoilla, mutta niiden käyttö jää usein erityistapauksiin tai perinteisten sovellusten migraatioihin. Niiden vähäisempää suosiota selittävät rajallinen ekosysteemi, pienempi kehittäjäkunta ja se, että pilvialustat eivät ole optimoineet niitä yhtä pitkälle kuin Node.js:ää

tai Pythonia. Niiden yhteenlaskettu osuus serverless-funktioista jää alle viiden prosentin. [61]

4. KÄYTTÖSYYT JA TEKNISET VALINNAT

Tulkattavien ohjelmointikielten suosioon serverless-ympäristöissä vaikuttavat monet tekijät kehittäjien kokemuksista aina ohjelmointikielten teknisiin ominaisuuksiin asti. Tässä luvussa käsitellään ohjelmointikielen valinnan taustalla olevia syitä serverless-laskennassa. Tarkasteltavia näkökulmia on mm. ohjelmistokehittäjien näkökulmat, ekosysteeminäkökohdat, suorituskykytekijät ja integrointikyvyt.

4.1 Ohjelmistokehittäjien näkökulma

Serverless-sovellusten kehittäminen ei ole pelkästään tekninen kysymys, vaan myös ihmisten tekemää työtä. Kehittäjien kokemukset, mieltymykset ja käytännön arjen sujuvuus vaikuttavat paljon siihen, mitä teknologioita ja kieliä he valitsevat. Usein päätöksiin vaikuttaa se, mikä tuntuu tutulta ja nopealta käyttää, ei niinkään se, mikä olisi teoreettisesti tehokkain.

4.1.1 Ohjelmistokehittäjien mieltymykset ja kokemukset

Ohjelmistokehittäjien mieltymykset ja kokemukset vaikuttavat merkittävästi ohjelmointikielen valintaan serverless-ympäristöissä. Usein tuottavuus ja tuttuus asetetaan etusijalle raakasuorituskyvyn sijaan. Inhimilliset tekijät painavat päätöksenteossa usein enemmän kuin tekniset ominaisuudet, ja siksi ne vaikuttavat merkittävästi siihen, mitä teknologioita organisaatiot ottavat käyttöön [62]. Kehittäjille entuudestaan tutut ohjelmointikielet mahdollistavat osaamisen hyödyntämisen ilman uusien kielten ja työkalujen opettelua. Tämä lyhentää perehtymisaikaa, nopeuttaa kehitystä ja helpottaa aikaisemmin kertyneen kokemuksen hyödyntämistä [63].

Organisaatiot asettavat usein nykyisten tiimiensä taidot etusijalle ohjelmointikielten teoreettisiin etuihin verrattuna. Tutulla ohjelmointikielellä kirjoitettu koodi voi olla kokonaisuudessaan kustannustehokkaampaa kuin teknisesti optimoidulla, mutta kehittäjille vieraalla kielellä toteutettu ratkaisu. Tämä johtuu erityisesti kehitys- ja ylläpitokustannuksista, sillä kehittäjien aika on usein merkittävämpi kustannustekijä kuin laskentaresurssit. Tämä pätee erityisesti organisaatioissa, joissa nopea innovointi ja markkinoille pääsy ovat strategisia prioriteetteja. [62] [64]

Dynaamisesti tyypitetyt ja tulkattavat kielet, kuten JavaScript ja Python, koetaan usein helpommiksi oppia kuin kielet, jotka vaativat monimutkaisempien tyyppijärjestelmien tai muistinhallinnan hallintaa [65]. Tämä tekee niistä houkuttelevia kasvaville tai vaihtuville

kehitystiimeille, koska yksinkertaisempi syntaksi helpottaa uusien jäsenten perehdyttämistä. Lisäksi näiden kielten laaja käyttäjäkunta tarkoittaa, että osaajia on enemmän tarjolla, mikä helpottaa rekrytointia [66].

Serverless-funktiot ovat tyypillisesti pieniä ja tarkasti kohdennettuja verrattuna monoliittisiin sovelluksiin. Tämä tekee niistä käyttökelpoisia myös uusien ohjelmointikielten tai mallien kokeiluun ja oppimiseen. Yksittäisten funktioiden rajattu näkyvyys ja selkeät rajat vähentävät uusien teknologioiden kokeiluun liittyviä riskejä. Näin organisaatiot voivat ottaa käyttöön uusia teknologioita asteittain ilman, että koko sovellus joudutaan rakentamaan uudelleen. [57]

4.1.2 Kehitysnopeus ja tuottavuus

Tulkattavat ohjelmointikielet tarjoavat usein merkittäviä etuja kehitysnopeudessa ja tuottavuudessa. Nämä hyödyt sopivat hyvin serverless-ympäristön iteratiiviseen ja funktiokeskeiseen luonteeseen, jossa uusia toiminnallisuuksia rakennetaan ja otetaan käyttöön nopeissa sykleissä. Usein kehityksen nopeus ja tiimien tuottavuus nousevatkin merkittävämmiksi tekijöiksi kuin yksittäisten funktioiden raakasuorituskyky, sillä serverless-sovelluksissa painopiste on nopeassa kokeilussa ja markkinoille pääsyssä. [57][64]

Yksi keskeisistä tuottavuuteen vaikuttavista tekijöistä on se, että tulkattavilla kielillä ei ole käännösvaihetta. Kehittäjä voi suorittaa koodin välittömästi muutosten jälkeen, mikä mahdollistaa tiheät iteraatiot ilman odotusta. Tämä ominaisuus tukee erityisesti palvelimettoman arkkitehtuurin kehitystapaa, jossa funktiot ovat pieniä ja rajattuja, ja niiden nopea muokkaaminen ja testaaminen on olennainen osa työnkulkua. Monissa tulkattavissa kielissä käytössä oleva dynaaminen tyypitys vahvistaa tätä nopeutta edelleen, sillä ohjelmoijan ei tarvitse käyttää aikaa yksityiskohtaisiin tyyppimäärityksiin. Toisaalta tämä etu tuo mukanaan myös varjopuolen: virheitä voi löytyä vasta ajon aikana, mikä voi nostaa testauksen ja laadunvarmistuksen merkitystä erityisesti tuotantokäytössä. [59]

Tuottavuutta ei kuitenkaan määritä ainoastaan käännösvaiheen puuttuminen. Tulkattaville kielille, kuten Pythonille ja JavaScriptille, on tyypillistä yksinkertainen ja ytimekäs syntaksi sekä laajat standardikirjastot, jotka tarjoavat korkean tason abstraktioita ja valmiita komponentteja. Monimutkainen toiminnallisuus voidaan usein kapseloida muutamaan koodiriviin, mikä vähentää tarvittavan koodin määrää ja parantaa luettavuutta. [63] Tämä yksinkertaistaa kehittäjien työtä ja ohjaa huomion siihen, mikä on serverless-sovelluksissa olennaista: liiketoimintalogiikkaan ja sen ketterään toteutukseen, ei ohjelmointikielen teknisiin yksityiskohtiin. Näin kehittäjät voivat käyttää enemmän aikaa sovelluksen arvon tuottamiseen ja vähemmän aikaa kielen tai ympäristön hallintaan.

Kokonaisuutena tulkattavien kielten tuottavuusedut muodostavat vahvan perustelun niiden käytölle serverless-ympäristöissä, vaikka ne saattavatkin joissain tilanteissa tarkoittaa kompromisseja suorituskyvyssä. Käytännön ohjelmistokehityksessä on kuitenkin usein tärkeämpää, että tiimit voivat työskennellä nopeasti ja tehokkaasti, kuin että yksittäisten funktioiden suoritus on maksimaalisesti optimoitu [57][64]. Tämä tekee tulkattavista kielistä monelle organisaatiolle houkuttelevan valinnan.

4.1.3 Virheenkorjaus- ja testausominaisuudet

Virheenkorjaus- ja testauskäytännöt eroavat huomattavasti tulkattavien ja käännettävien ohjelmointikielten välillä [57]. Näillä eroilla voi olla käytännön merkitystä serverless-sovellusten kehityksessä, sillä palvelimettomat järjestelmät ovat hajautettuja ja tapahtumapohjaisia. Tämä hajautettu luonne vaikeuttaa virheiden jäljittämistä: yksittäinen funktio voi käynnistyä monesta eri lähteestä ja suorittua vain lyhyen aikaa, minkä vuoksi virheen toistaminen on usein haastavaa [67].

Tulkattavat ohjelmointikielet tarjoavat usein välittömämmän ja luettavamman palautteen virhetilanteissa. Niiden ajonaikaiset ympäristöt tuottavat tyypillisesti selkeitä pinojälkiä, jotka osoittavat suoraan virheen sijainnin lähdekoodissa ja antavat kontekstia sen syistä. Tämä helpottaa ongelmien paikantamista ja lyhentää korjaamiseen kuluvaa aikaa. Esimerkiksi JavaScriptin virheilmoitukset voivat kertoa tarkalleen, missä funktiossa virhe syntyi, mikä sopii hyvin serverless-ympäristöön, jossa kehittäjät tarvitsevat nopeaa palautetta. [68]

Käännetyt kielet tarjoavat erilaisen virheenkorjauskokemuksen. Koska ohjelma muunnetaan konekieleksi, virheiden tarkka paikantaminen lähdekoodiin ei onnistu pelkästään pinojälkeä tutkimalla. Tätä varten käytetään symboleita, lähdekoodikarttoja ja erillisiä työkaluja, kuten gdb (GNU Debugger) tai vastaavia IDE-integraatioita. Vaikka nämä työkalut ovat tehokkaita, ne vaativat kehittäjiltä enemmän osaamista ja konfiguraatiota, mikä voi hidastaa virheiden paikantamista verrattuna tulkattaviin kieliin. Toisaalta vahva tyypitys ja käännösvaiheen tarkistukset vähentävät usein niiden virheiden määrää, jotka päätyvät ajoon.

Serverless-ympäristöissä debuggausta vaikeuttaa lisäksi se, että funktiot ovat luonteeltaan tilattomia ja usein lyhytikäisiä. Tämä tekee perinteisestä interaktiivisesta virheenkorjauksesta hankalaa. Tulkattavat kielet voivat tällöin tarjota kehittäjälle enemmän joustavuutta, sillä virheitä voidaan paikantaa ajon aikana myös ilman erillisiä käännös- ja kartoitusvaiheita. Toisaalta modernit käännetyt kielet, kuten Go ja Java, tarjoavat

kehittyneitä työkaluja, joilla voidaan seurata tapahtumia hajautetuissa ympäristöissä, mutta näiden hyödyntäminen voi edellyttää syvempää perehtymistä ja infrastruktuuritukea.

4.2 Ekosysteemien ja kirjastojen vaikutus

Merkittävä käytännön etu suosituimmille tulkattaville ohjelmointikielille, kuten JavaScriptille ja Pythonille on se, että niille on tarjolla runsaasti serverless-kehitykseen erikoistuneita kehyksiä ja työkaluja. Nämä työkalut on suunniteltu vastaamaan serverless-arkkitehtuurien erityisiin haasteisiin, ja ne yksinkertaistavat funktiopohjaisten sovellusten kehitys-, käyttöönotto- ja hallintaprosesseja. Tämä vaikuttaa suoraan ohjelmointikielten valintaan projekteissa ja tukee niiden asemaa serverless-ekosysteemeissä. [69]

Yksi alan tunnetuimmista ja laajimmin käytetyistä työkaluista on Serverless Framework. Se sai alkunsa Node.js-ekosysteemistä, mikä liittyi JavaScriptin hallitsevaan asemaan serverless-sovellusten varhaisessa kehityksessä, ja laajentui myöhemmin tukemaan useita muita kieliä. [69] Framework tarjoaa yhtenäisen komentorivikäyttöliittymän ja määrittelytavan serverless-sovellusten rakentamiseen eri pilvialustoilla. Se poistaa palveluntarjoajakohtaisia yksityiskohtia ja helpottaa siirrettävyyttä. Frameworkin ympärille on syntynyt laaja ekosysteemi lisäosia, jotka tarjoavat ratkaisuja erityisiin tarpeisiin, kuten paikalliseen testaukseen, valvontaan ja tietoturvaan.

AWS on kehittänyt myös omia työkalujaan, kuten Serverless Application Model (SAM) ja Cloud Development Kit (CDK). SAM laajentaa CloudFormationia serverless-ohjelmien abstraktioilla ja tarjoaa erityisen vahvan tuen Node.js- ja Python-funktioille [52]. CDK puolestaan mahdollistaa infrastruktuurin määrittelyn ohjelmointikielillä mallinnuskielen sijaan, ja sen ensimmäinen versio julkaistiin TypeScript-toteutuksella, minkä jälkeen se laajennettiin tukemaan Pythonia, Javaa ja muita kieliä [70]. Nämä työkalut yksinkertaistavat serverless-resurssien määrittelyä ja käyttöönottoa, ja niiden painotus tulkattaviin kieliin on vahvistanut näiden kielten käyttöä AWS Lambda-sovelluksissa.

Myös muut palveluntarjoajat ovat panostaneet tulkattavien kielten tukeen. Azure Functions tarjoaa vahvan integraation JavaScriptin ja Pythonin kanssa, ja niille on kattavasti dokumentaatiota, esimerkkejä ja työkaluja [72]. Google Cloud Functions on samoin painottanut Node.js- ja Python-ympäristöjä, joille se tarjoaa räätälöityjä käyttöönottotyökaluja ja valvontaratkaisuja [45]. Näiden investointien myötä tulkattavat kielet ovat vahvistaneet asemaansa serverless-kehityksessä.

Tulkattavat kielet ovat myös hyvin tuettuja lokaaleissa kehitys- ja testaustyökaluissa. Esimerkkejä ovat AWS SAM CLI, Azure Functions Core Tools ja Serverless Frameworkin

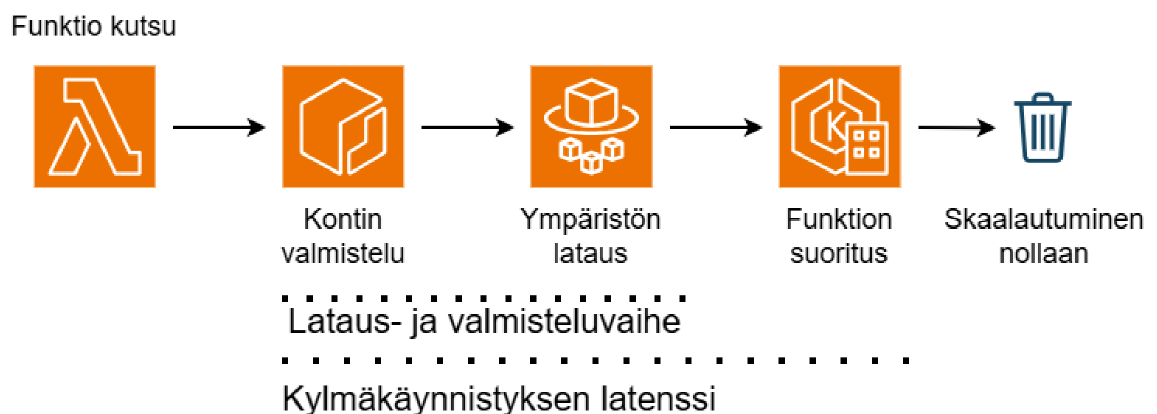
offline-laajennus. Nämä työkalut mahdollistavat funktioiden suorittamisen paikallisesti ilman käyttöönottoa, mikä nopeuttaa kehitystä ja vähentää testaukseen liittyviä kustannuksia. [40][71] Tulkattavat kielet soveltuvat erityisen hyvin tällaisiin emulaattoreihin, koska koodia voidaan ajaa suoraan ilman käännösvaihetta. Käännettävien kielten kohdalla sen sijaan tarvitaan usein build-prosesseja, riippuvuuksien paketointia ja joissain tapauksissa konttipohjaisia ratkaisuja yhteensopivuuden varmistamiseksi. Tämä ei tee niiden käyttöä mahdottomaksi, mutta lisää kehitysprosessin monimutkaisuutta verrattuna tulkattaviin kieliin.

4.3 Suorituskyky- ja skaalautuvuustekijät

Vaikka tulkattavat ohjelmointikielet eivät yleensä pärjää käännetyille kielille raa'assa laskentatehossa, niiden useat suorituskyky- ja skaalautuvuusominaisuudet tekevät niistä yllättävän sopivia monille serverless-työkuormille. Näiden pienien suorituskykytekijöiden ymmärtäminen auttaa selittämään, miksi tulkattavat kielet ovat säilyttäneet hallitsevan asemansa teoreettisista suorituskykyhaitoista huolimatta.

4.3.1 Kylmäkäynnistyksen tehokkuus

Kylmäkäynnistyksen latenssi on yksi merkittävimmistä suorituskykyyn vaikuttavista tekijöistä serverless-ympäristöissä [57]. Kun funktio suoritetaan ensimmäistä kertaa tai sen jälkeen, kun instanssi on vapautettu, alustan on alustettava ajonaikainen ympäristö ennen suorituksen aloittamista (Kuva 4). Tämä voi lisätä viivettä erityisesti latenssiherkissä sovelluksissa.



Kuva 4. *Kylmäkäynnistyksen latenssiprosessi [7]*

Tulkattavat kielet ovat osoittautuneet monissa tutkimuksissa kilpailukykyisiksi kylmäkäynnistyksen suhteen. Node.js on usein suoriutunut erittäin hyvin verrattuna moniin muihin kieliin, myös joihinkin käännettyihin vaihtoehtoihin. Sen taustalla on V8 JavaScript -moottori, joka on suunniteltu selainympäristöihin, joissa nopea käynnistyminen on välttämätöntä hyvän käyttökokemuksen takaamiseksi. Sama optimointi näkyy serverless-ympäristöissä, ja Node.js:n kylmäkäynnistys voi olla alle 100 millisekuntia yksinkertaisissa funktioissa, joissa riippuvuuksia on vähän. [73]

Python on sekin monissa raporteissa sijoittunut hyvin kylmäkäynnistystesteissä. Datadogin State of Serverless 2023 -raportin mukaan AWS Lambdassa Pythonin kylmäkäynnistys oli jopa lähes kolme kertaa nopeampi kuin Javalla, ja myös Node.js:ää nopeampi monissa tapauksissa [10]. Tämä johtuu osittain siitä, että Pythonin alustukset on optimoitu pilvipalveluntarjoajien toimesta ja sen laaja vakiokirjasto kattaa suuren osan yleisistä käyttötapauksista, jolloin erillisten ulkoisten riippuvuuksien tarve voi olla vähäisempi.

Java puolestaan on perinteisesti ollut kylmäkäynnistyksessä hitain vaihtoehto. Tämä selittyy muun muassa virtuaalikoneen ja laajan suoritusympäristön alustamiseen tarvittavalla ajalla. Vaikka Java tarjoaa vahvaa suorituskykyä pitkäkestoisissa prosesseissa, sen kylmäkäynnistyslatenssi on useissa tutkimuksissa jäänyt selvästi jälkeen Node.js:stä ja Pythonista. [73][74]

Kylmäkäynnistyksen kestoon vaikuttavat kuitenkin ohjelmointikielen lisäksi monet tekijät. Sovelluspaketin koko ja siihen sisällytettyjen riippuvuuksien määrä ovat keskeisiä: mitä enemmän paketteja ja kirjastoja alustalla täytyy ladata, sitä kauemmin ympäristön käynnistäminen kestää [73]. Muistin määrä on toinen merkittävä tekijä, sillä suuremmat muistimäärät tuovat käyttöön enemmän prosessointitehoa, mikä voi nopeuttaa alustusta [74]. Lisäksi ajonaikaisen ympäristön versioilla on merkitystä, sillä uudemmat versiot sisältävät usein optimointeja, jotka parantavat käynnistysaikaa.

Pilvipalveluntarjoajat ovat viime vuosina kehittäneet ratkaisuja, joilla kylmäkäynnistykseen liittyvää ongelmaa voidaan pienentää tai poistaa kokonaan. AWS Lambda tarjoaa provisioned concurrency-ominaisuuden, joka pitää ennalta määritellyn määrän funktioinstansseja jatkuvasti valmiina suoritukseen [75]. Azure Functions Premium-suunnitelma ja vastaavat ratkaisut muilla alustoilla tarjoavat samankaltaisia esilämmitettyjä instansseja [32]. Tällaiset ratkaisut poistavat käytännössä kylmäkäynnistysongelman, mutta lisäävät kustannuksia. Näiden alustatason optimointien myötä suorituskykyerot tulkattavien ja käännettyjen kielten välillä ovat pienentyneet, ja kylmäkäynnistys ei enää ole yhtä merkittävä erottava tekijä kuin varhaisissa serverless-toteutuksissa.

4.3.2 Suorituskykyominaisuudet ajonaikaisissa ympäristöissä

Ajonaikaisen ympäristön suorituskyky on alustuksen ja käynnistyksen jälkeen toinen keskeinen tekijä ohjelmointikielen valinnassa. Sen käytännön merkitys riippuu kuitenkin suuresti työkuorman ominaisuuksista ja suoritustavoista. Tulkattavat kielet eroavat tyypillisesti suoritusteholtaan käännettävistä kielistä, mikä vaikuttaa sekä funktioiden toimintaan että niiden kustannustehokkuuteen serverless-ympäristöissä. [57][64]

JavaScript on hyvä esimerkki tulkattavasta kielestä, jonka suorituskyky on Node.js:n kautta osoittautunut monissa serverless-työkuormissa yllättävän kilpailukykyiseksi. V8-moottorin just-in-time (JIT) -kääntäjä mahdollistaa koodin dynaamisen optimoinnin: usein suoritettavat funktiot käännetään ajon aikana natiivikoodiksi, ja suoritusprofiilin perusteella voidaan tehdä lisäoptimointeja, kuten inline-kutsujen tai silmukoiden optimointia [55][76]. Kun koodi stabiloituu, V8 kykenee tekemään yhä aggressiivisempia optimointeja, mutta se voi myös purkaa niitä (deoptimointi), jos aiemmat oletukset esimerkiksi muuttujatyypeistä eivät enää pidä paikkaansa. Tämä mekanismi luo tasapainon suorituskyvyn ja joustavuuden välille. Vaikka JIT ei tee Node.js:stä systemaattisesti yhtä nopeaa kuin käännetyt kielet, tietyissä työkuormissa se voi lähestyä niiden tasoa.

Node.js:n tapahtumapohjainen ja ei-estetty (non-blocking) arkkitehtuuri on lisäksi luonteva ratkaisu I/O-painotteisiin funktioihin. Se mahdollistaa useiden yhtäaikaisten toimintojen käsittelyn ilman raskasta säikeidenhallintaa. Tällöin suorituskyky on usein enemmän kuin riittävä, ja kehittäjä saa käyttöönsä yksinkertaisen ohjelmointimallin, joka soveltuu erityisen hyvin palveluiden väliseen orkestrointiin. [57]

Pythonin suorituskyky riippuu pitkälti sen ekosysteemistä ja tavasta hyödyntää valmiita kirjastoja. Yleisin toteutus on CPython, joka tarjoaa tyydyttävän suorituskyvyn yleiskäyttöön, mutta sen todellinen vahvuus on siinä, että raskas laskenta voidaan delegoida tehokkaammilla kielillä kirjoitetuille kirjastoille. Esimerkiksi NumPy ja TensorFlow suorittavat suuren osan operaatioista C- tai CUDA-toteutuksina, jolloin funktiot pysyvät suorituskykyisinä, vaikka pääkoodi on Pythonilla kirjoitettua. Tämä tekee Pythonista erityisen käyttökelpoisen datakeskeisiin serverless-funktioihin, joissa laskenta voidaan keskittää optimoituihin kirjastoihin.

Pythonista on olemassa myös PyPy-toteutus, joka sisältää JIT-kääntäjän ja se voi nopeuttaa puhtaan Python-koodin suorituskykyä jopa 2–10-kertaiseksi verrattuna CPythoniin [77]. Rajoitteena on kuitenkin se, että se ei tue laajasti C-laajennuskirjastoja, mikä heikentää sen soveltuvuutta raskaaseen laskentaan ja koneoppimiseen. Lisäksi useimmat serverless-alustat eivät tue PyPyä natiivisti, vaan sen käyttö vaatii konttipohjaisia

ratkaisuja, kuten AWS Lambdan mukautettua suoritusympäristöä tai Google Cloud Runia.

Suorituskyky ei rajoitu pelkästään prosessoritehoon, vaan myös muistinkäyttö on ratkaisevaa. Tulkattavat kielet, kuten Python ja JavaScript, kuluttavat usein enemmän muistia tulkin, ajonaikaisten tyyppitietojen ja roskienkeruun vuoksi. Serverless-alustoilla suorituskyky skaalautuu usein varatun muistin mukaan, joten suuremman muistimäärän varaaminen voi nopeuttaa funktioiden suoritusta. Tämä voi olla kustannustehokasta, jos suoritusaikojen lyhentyminen kompensoi resurssikustannusten kasvun. [73] Hyvin mitoitettu muistin allokointi onkin käytännössä tärkeämpää kuin pelkkä minimiresurssien tavoittelu.

Lopulta ohjelmointikielten suorituskykyerot näkyvät käytännössä vaihtelevasti. Suurin osa serverless-funktioista on lyhytkestoisia ja selvästi alustan aikarajojen sisällä, jolloin kielten väliset erot ovat usein vain millisekuntien luokkaa eivätkä vaikuta merkittävästi kustannuksiin tai käyttäjäkokemukseen. Erot korostuvat vasta silloin, kun kyseessä on laskennallisesti raskaita funktioita, jotka lähestyvät aikarajoja. Näissä tapauksissa käännettävät kielet voivat tarjota selkeän edun, sillä ne mahdollistavat suorituksia, jotka tulkatuilla kielillä ylittäisivät aikarajat ja johtaisivat aikakatkaisuihin. [64]

5. KUSTANNUSVAIKUTUKSET

Serverless-ratkaisujen kustannusrakenne poikkeaa perinteisistä malleista, mikä tekee taloudellisesta näkökulmasta erityisen tärkeän. Tässä luvussa tarkastellaan tärkeimpien serverless-palveluntarjoajien hinnoittelumalleja, analysoidaan tulkattavien ja käännettävien ohjelmointikielien välisiä kustannuseroja ja tutkitaan strategioita suorituskyvyn ja kustannusten optimoimiseksi serverless-ympäristöissä.

5.1 Hinnoittelumallit serverless-palveluissa

Serverless-laskenta tuo mukanaan erilaisen hinnoittelumallin verrattuna perinteisiin infrastruktuurilähestymistapoihin, sillä siinä useat keskeiset komponentit vaikuttavat suoraan sovellusten kokonaiskustannuksiin. Näiden hinnoittelukomponenttien ymmärtäminen on tärkeää tietoon perustuvia kielivalintoja tehtäessä ja kustannustehokkaiden serverless-ratkaisujen suunnittelussa ja toteutuksessa.

5.1.1 Suoritusajan laskutus

Suoritusaika on ensisijainen osa serverless-palveluiden hinnoittelussa, ja se luo suoran taloudellisen yhteyden koodin tehokkuuden ja toimintakustannusten välille. Tämä ”maksa käytöstä” -malli muuttaa perusteellisesti tapaa, jolla organisaatioiden tulisi suhtautua ohjelmointikielen suorituskykyyn ja optimointistrategioihin.

Suoritusajan laskutuksen tarkkuus vaihtelee tärkeimpien serverless-palveluntuottajien välillä, mikä luo erilaisia kustannusvaikutuksia funktioille, joilla on vaihtelevat suoritusprofiilit. AWS Lambda tarjoaa hienojakoisimman hinnoittelun veloittamalla suorituksesta millisekunnin tarkkuudella ilman kiinteää vähimmäisveloitusta [25], mikä mahdollistaa tarkan kustannusten kohdentamisen lyhyille suorituksille. Google Cloud Functions käyttää 100 millisekunnin tarkkuutta [51], mikä muodostaa kompromissin laskutustarkkuuden ja yksinkertaisuuden välille. Azure Functionsin kulutusperusteinen malli veloittaa myös 100 ms tarkkuudella [41], mutta Premium- ja Dedicated-vaihtoehdoissa voidaan veloittaa 1 ms tarkkuudella. Nämä erot suoritusaikojen laskennassa voivat muodostaa huomattavia kustannusvaikutuksia, erityisesti sovelluksissa, joiden toiminnot ovat lyhyitä tai skaalautuvat voimakkaasti.

Suoritusajan laskentamenetelmä luo tärkeitä näkökohtia ohjelmointikielen valinnalle ja optimoinnille. Palveluntarjoajat mittaavat aikaa funktion kutsumisesta valmistumiseen tai aikakatkaisuun, mukaan lukien kriittinen alustusaika kylmäkäynnistysten aikana. Tämä

alustusajan sisällyttäminen tarkoittaa, että nopeamman käynnistyksen omaavat kielet voivat tarjota kustannusetuja, erityisesti harvoin kutsutuille funktioille, jotka kokevat useita kylmäkäynnistyksiä. Useimmat palveluntarjoajat pyöristävät suoritusajan ylöspäin omaan laskutusjaksoonsa, olipa se millisekunti, 100 millisekuntia tai sekunti, mikä aiheuttaa ns. kynnysefektejä. Pienetkin suorituskyvyn parannukset voivat tällöin johtaa kustannussäästöihin, jos suoritusajan lyhentyminen pudottaa funktion alempaan laskutusrajaan.

Hinnoittelutasot tuovat kustannuslaskelmaan uuden ulottuvuuden, erityisesti suurten käyttömäärien sovelluksissa, jotka voivat hyötyä volyymialennuksista tai vaihtoehtoisista hinnoittelumalleista. Useimmat palveluntarjoajat tarjoavat ilmaisia käyttökiintiöitä, jotka voivat kattaa kehitysympäristöjen ja pienten työkuormien tarpeet kokonaan. AWS Lambda tarjoaa 1 miljoonan kutsun ja 400 tuhannen GB-sekunnin ilmaisen kuukausikiintiön [25], Google Cloud Functions 2 miljoonaa kutsua ja vastaavat resurssit [51], ja Azure Functions 1 miljoonan kutsun ilmaisen tason [41].

Hinnoittelurakenteet vaihtelevat: Azure Functions tarjoaa eri hintatasoja (Consumption, Premium, Dedicated), joista jokainen vastaa erilaisia suorituskyky- ja skaalautuvuustarpeita [41]. Google ja AWS eivät tarjoa kiinteitä hinnoittelutasoja, mutta molemmat mahdollistavat neuvoteltavat alennukset suurille yritysasiakkaille. Lisäksi aluekohtaiset hinnat luovat lisähaasteita kustannusten ennustamiseen, sillä suoritusmaksut voivat vaihdella merkittävästi eri maantieteellisillä alueilla, esimerkiksi Euroopan ja Yhdysvaltojen välillä.

Suurasiakkaille tarjottavat Enterprise-sopimukset voivat edelleen muokata kokonaiskustannuksia, jolloin hinnoittelu ei enää riipu pelkästään funktioiden määristä ja kestosta, vaan myös strategisista volyymisopimuksista. Tämä voi vaikuttaa merkittävästi siihen, mitä ohjelmointikieltä tai alustan ominaisuuksia organisaatio pitää taloudellisesti optimaalisena pitkällä aikavälillä.

5.1.2 Muistin allokoinnin hinnoittelu

Muistin allokointi on toinen keskeinen osa serverless-laskennan hinnoittelussa. Sen ja suorituskustannusten välinen suhde ei ole suoraviivainen. Allokoidun muistin määrä vaikuttaa sekä kustannuksiin että suorituskykyyn, ja näiden välinen tasapaino muodostaa tärkeän optimointikohdan. Useimpien serverless-alustojen hinnoittelumalli perustuu gigatavu-sekuntien laskentaan eli muistimäärän ja funktion suorituskeston tuloon. Näin ollen suurempi muistiallokaatio kasvattaa sekuntikohtaista kustannusta. Kuitenkin muistiallokaatio vaikuttaa myös suoritintehoon. Useimmilla alustoilla suurempi muistimäärä

tuo käyttöön enemmän CPU-resursseja, mikä voi nopeuttaa funktion suoritusta ja siten vähentää kokonaiskustannusta, erityisesti lyhytaikaisissa tai suorituskykykriittisissä tehtävissä. Tämän vuoksi muistimäärän optimointi ei ole pelkästään kustannusten minimointia vaan myös suorituksen tehokkuuden maksimointia. [25][41][51]

Optimointistrategiat voivat vaihdella myös ohjelmointikielen mukaan. Joissakin tutkimuksissa ja käytännön mittauksissa on havaittu, että tulkattavat kielet, kuten Python ja JavaScript, hyötyvät usein suhteellisesti suuremmista muistiallokaatioista suorituskyvyn parantamiseksi, kun taas käännetyt kielet, kuten Go tai Rust, voivat saavuttaa optimaaliset tulokset pienemmällä resurssimäärällä [2][56]. Näin ollen muistin allokoinnin valinta on keskeinen osa kielivalintaa ja sovellusarkkitehtuuria, erityisesti silloin kun pyritään kustannustehokkaaseen ja skaalautuvaan toteutukseen.

Palveluntarjoajakohtaiset lähestymistavat muistin allokointiin luovat erilaisia rajoituksia ja mahdollisuuksia serverless-ympäristöissä. AWS Lambda tarjoaa tällä hetkellä joustavimman mallin, jossa muisti voidaan allokoida 128 megatavusta aina 10 gigatavuun saakka, yhden megatavun tarkkuudella [13]. Tämä mahdollistaa tarkan optimoinnin työkuormien luonteen ja käytetyn ohjelmointikielen vaatimusten mukaan.

Azure Functions toimii eri tavalla. Consumption-suunnitelmassa muisti ei ole suoraan konfiguroitavissa, vaan se on sidottu host-instanssiin, jonka enimmäismuisti on noin 1,5 gigatavua. Premium- ja Dedicated-suunnitelmissa voidaan käyttää suurempia määriä muistia, jopa 14 gigatavua per instanssi. [32] Muistin hallinta ei kuitenkaan ole yhtä hienojakoista kuin AWS:llä.

Google Cloud Functions tarjoaa muistiallokoinnin 128 megatavusta 8 gigatavuun, porrastettuna ennalta määriteltyihin arvoihin [44]. Vaikka tämä ei ole yhtä tarkasti säädettävissä kuin AWS:n ratkaisu, se tarjoaa silti merkittävää joustavuutta eri kokoisille työkuormille.

Viimeaikainen tutkimus on osoittanut, että muistin hallinta on yksi keskeisimmistä tekijöistä serverless-ympäristöjen kustannustehokkuudessa. Wang ym. havaitsivat, että palvelimettomissa funktioissa jopa 50 % suoritusajasta voi kulua muistinhallintaan ja että suurin osa allokaatioista on hyvin pieniä ja lyhytikäisiä, mikä korostaa muistinhallinnan optimoinnin merkitystä [79]. Xu ym. esittävät FaaSMem-mekanismin, joka hyödyntää muistiallasarkkitehtuuria ja siirtää harvoin käytetyt sivut etämuistiin [80]. Tulokset osoittavat, että suurin osa muistin tehottomuudesta johtuu konteista, jotka pidetään aktiivisina niin sanotussa keep-alive-vaiheessa. Ehdotettu ratkaisu vähentää paikallista muistinkäyttöä jopa 80 % ilman merkittävää vaikutusta vasteaikoihin.

Tzenetopoulos kollegoineen ovat puolestaan kehittäneet elastisen muistiallokoinnin mallin, jossa muistia voidaan skaalata dynaamisesti kuormituksen mukaan [81]. Bilal ja tutkimusryhmä osoittivat, että CPU:n ja muistin erottaminen toisistaan resurssien hallinnassa voi parantaa resurssien käyttöastetta ja alentaa kokonaiskustannuksia huomattavasti [82]. Eismann ja tutkimusryhmä täydentävät kokonaiskuvaa mallillaan, joka ennustaa optimaalisen muistimäärän serverless-funktiolle ja osoittaa, että automaattinen allokointiennuste voi parantaa kustannustehokkuutta ja suorituskykyä verrattuna staattisiin asetuksiin [83].

Muistin allokointi on siis sekä tekninen että taloudellinen optimointiongelma. Tuoreet tutkimukset vahvistavat, että muistinhallinta vaikuttaa ratkaisevasti serverless-arkkitehtuurien kustannusrakenteeseen ja että uudet arkkitehtuuriset ja algoritmiset lähestymistavat voivat tuoda huomattavia hyötyjä palvelimettomien järjestelmien tehokkuuteen. [79] [83]

5.1.3 Pyyntö-/kutsumaksu

Funktion kutsujen määrä muodostaa kolmannen keskeisen osan serverless-laskennan hinnoittelussa. Vaikka yksittäisen kutsun hinta ei riipu käytetystä ohjelmointikielestä, eri kielten mahdollistamat arkkitehtuurivalinnat voivat merkittävästi muuttaa kutsumääriä ja siten kokonaiskustannuksia.

Palveluntarjoajien hinnoittelu vaihtelee hieman, mutta perustuu yleisesti kiinteään hintaan kutakin kutsua kohden. AWS Lambda veloittaa 0,20 Yhdysvaltain dollaria miljoonalta kutsulta ilmaiskynnyksen ylityttyä [25]. Azure Functions ja Google Cloud Funtions veloittavat 0,40 Yhdysvaltain dollaria per miljoona kutsua [41][51]. Näiden erojen takia erityisesti suurivolyymiset sovellukset voivat kokea merkittäviä kustannuseroja alustavalinnan perusteella.

Useimmat palveluntarjoajat tarjoavat kuitenkin laajat ilmaiset käyttörajat, jotka poistavat kustannuksia kehitysvaiheessa ja pienimuotoisessa käytössä. AWS Lambda ja Azure Functions tarjoavat kumpikin 1 miljoona ilmaista kutsua kuukaudessa, kun taas Google Cloud Functions tarjoaa 2 miljoonaa ilmaista kutsua [25][41][51].

Kutsujen lähteet, kuten HTTP-pyynnöt, pilvipalveluiden tapahtumat, ajastetut tehtävät ja sovelluksen sisäiset funktiokutsut vaikuttavat suoraan kutsumääriin. Arkkitehtuuriratkaisut, kuten kuinka hienojakoisia funktiot ovat tai kuinka tehokkaasti tapahtumia suodatetaan, vaikuttavat kutsujen volyymiin. Samoin valittu kieli voi tukea tai rajoittaa eri optimointitekniikoiden, kuten batch-käsittelyn tai välimuistin käytön hyödyntämistä [56][78].

Vaikka ohjelmointikieli ei siis vaikuta suoraan kutsukohtaisiin maksuihin, se voi muokata kutsumääriä epäsuorasti mahdollistamiensa arkkitehtuuristen ratkaisujen ja

optimointimallien kautta. Näin ollen myös kielivalinta voi vaikuttaa merkittävästi serverless-ratkaisujen kokonaiskustannuksiin, erityisesti suurivolyymisissa sovelluksissa.

5.1.4 Tiedonsiirtokustannukset

Tiedonsiirron kustannukset jäävät usein vähälle huomiolle serverless-laskennassa, vaikka niillä voi olla merkittävä vaikutus kokonaiskustannuksiin. Tiedonsiirto liittyy läheisesti sekä ohjelmointikieliin että arkkitehtuuriratkaisuihin. Vaikka siirtomaksut eivät riipu suoraan kielestä, kielikohtaiset ekosysteemit ja kehitysmallit voivat vaikuttaa siihen, kuinka paljon ja missä muodossa dataa siirretään ja siten siihen, kuinka paljon siirrosta lopulta maksetaan.

Suurimmat pilvipalveluntarjoajat noudattavat samankaltaista hinnoittelua ulkoisen tiedonsiirron osalta. Internetiin lähtevä liikenne maksaa tyypillisesti noin 0,05–0,12 Yhdysvaltain dollaria gigatavulta, riippuen palveluntarjoajasta, alueesta ja kuukausittaisesta siirtomäärästä [25][41][51]. Nämä kustannukset kohdistuvat dataan, joka lähtee serverless-funktioista ulkoisiin järjestelmiin, käyttäjille tai asiakassovelluksiin. Sitä vastoin internetistä tuleva liikenne funktioihin on pääsääntöisesti maksutonta, mikä luo epäsymmetrisen kustannusrakenteen ja kannustaa minimoimaan ulospäin suuntautuvaa tiedonsiirtoa.

Alueen sisäinen liikenne, kuten palveluiden välinen tiedonsiirto samassa datakeskuksessa, on useimmiten maksutonta tai huomattavasti edullisempaa. Sen sijaan alueiden välinen tiedonsiirto hinnoitellaan usein samoin kuin ulkoinen siirto, mikä tekee monialue-arkkitehtuureista tiedonsiirron näkökulmasta kalliimpia. [25][41][51]

Funktiotason siirtotarpeet lisäävät malliin vielä yhden ulottuvuuden. Pyyntöjen ja vastausten koko vaikuttaa suoraan siirtokustannuksiin, erityisesti dataintensiivisissä sovelluksissa. Eri ohjelmointikielet tarjoavat erilaisia ratkaisuja datan pakkaamiseen ja sarjallistamiseen, mitkä vaikuttavat suoraan siirrettävän datan määrään. Kielet, jotka tukevat tehokasta muistinhallintaa ja suorituskykyistä tiedonkäsittelyä, voivat vähentää siirrettävän datan määrää merkittävästi. [56][78][84]

Arkkitehtuuriset ratkaisut, kuten rajapintamalli, välimuistien käyttö ja datan sijainti, vaikuttavat suoraan tiedonsiirron määrään ja kustannuksiin. Esimerkiksi GraphQL voi vähentää siirrettävän datan määrää verrattuna perinteisiin REST-rajapintoihin. Eri kielet tarjoavat vaihtelevan tasoista tukea tehokkaille GraphQL-toteutuksille. Välimuistien hyödyntäminen sovellustasolla tai sisällönjakeluverkkojen (CDN) kautta voi pienentää toistuvien tiedonsiirtojen määrää. Lisäksi reunalaskenta, jossa dataa käsitellään lähellä sen syntypaikkaa, vähentää ulkoisen siirron tarvetta ja voi parantaa suorituskykyä. Tämän

lähestymistavan tuki vaihtelee kuitenkin palvelualustojen ja kieliekosysteemien välillä. [85][86][87]

5.2 Kustannuserot tulkattavien ja käännettävien kielten välillä

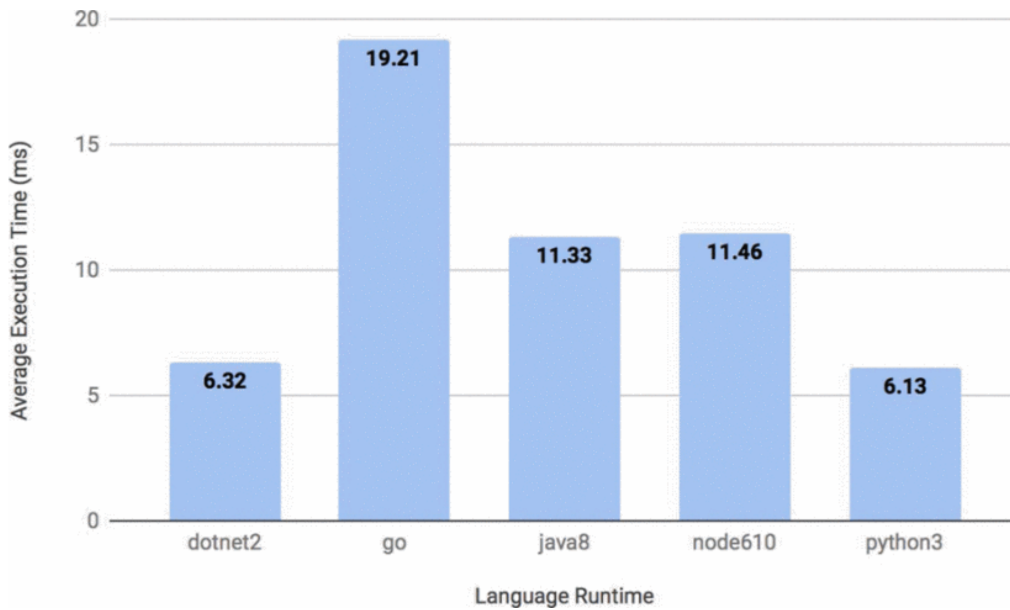
Tulkattavien ja käännettävien kielten erot suoritustavassa, muistinhallinnassa ja käynnistysnopeudessa heijastuvat suoraan laskutettavaan aikaan ja resurssien käyttöön. Näiden teknisten tekijöiden ohella myös kehityksen ja ylläpidon kustannukset vaikuttavat kokonaisuuteen. Kustannuseroja ei siten voida arvioida yksittäisen mittarin perusteella, vaan ne muodostuvat useista toisiinsa liittyvistä tekijöistä.

5.2.1 Suoritusajan kustannusvertailu

Suoritusajan kustannukset ovat yksi keskeinen erotteleva tekijä tulkattujen ja käännettyjen ohjelmointikielten välillä serverless-ympäristöissä. Kustannukset määräytyvät sekä varsinaisen suorituksen kestosta että alustuksen eli kylmäkäynnistyksen viiveestä, ja nämä yhdessä muodostavat laskutettavan ajan. Kirjallisuudessa korostuu, ettei yhtä yleispätevää voittajaa ole. Erot vaihtelevat alustan, suoritusympäristön toteutuksen, muistiasetusten ja mittausmenetelmän mukaan. [88]

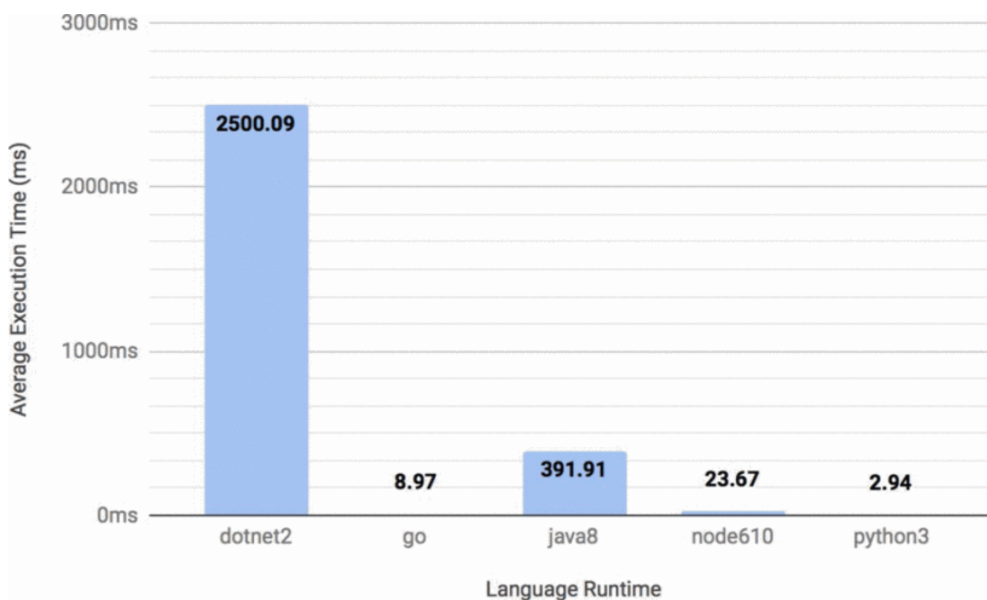
AWS Lambdassa tehty vertailututkimus osoittaa, että ohjelmointikielten suoritusympäristöjen suoriutumisprofiilit poikkeavat selvästi toisistaan. Jackson ja Clynch mittasivat viiden suoritusympäristön (.NET Core 2, Java 8, Python 3, Node.js 6.10, Go) suoritusta ja kustannusvaikutusta hyödyntäen asetuksia, joissa eroteltiin kylmät ja lämpimät käynnistykset sekä vakioitiin muistikonfiguraatio.

He raportoivat, että lämpimissä käynnistyksissä keskimääräiset laskutettavat suoritusajat (execution duration) olivat Python 3:lla 6,13 ms, .NET Core 2:lla 6,32 ms, Java 8:lla 11,33 ms, Node.js:lla 11,46 ms ja Go:lla 19,21 ms (Kuva 5). Tässä skenaariossa Python ja .NET Core 2 olivat nopeimpia, kun taas Go oli selkeästi hitain.



Kuva 5. *Keskimääräinen suoritusaika (ms) kaikissa AWS lambdan lämminkäynnistystesteissä [78].*

Kylmissä käynnistyksissä keskimääräiset laskutettavat suoritusajat puolestaan olivat Python 3:lla 2,94 ms, Go:lla 8,97 ms, Node.js:lla 23,67 ms, Java 8:lla 391,91 ms ja .NET Core 2:lla 2500,09 ms (Kuva 6). On kuitenkin huomioitava, että nämä tulokset kuvaavat funktioiden sisäisiä suoritusaikoja, eivätkä niiden kokonaisviiveitä. Tulokset osoittavat, että Pythonilla ja Node.js:llä suoritusympäristön alustusaika ei sisälly laskutettavaan suoritusaikaan samalla tavalla kuin C#:lla ja Javalla, joissa raskas alustus, kuten käännös, tapahtuu funktion suorituksen aikana.



Kuva 6. *Keskimääräinen suoritusaika (ms) kaikissa kylmäkäynnistystesteissä AWS lambdassa [78].*

Kylmäkäynnistyksessä Python oli mitatulla suoritusajalla selvästi nopein, ja Go toimi myös suhteellisen nopeasti. Tutkimuksessa havaittiin, että jostain syystä Python ja Go olivat nopeampia kylmäkäynnistyksessä kuin lämpimässä käynnistyksessä, mutta syytä siihen ei tutkittu sen tarkemmin. Sen sijaan kylmäkäynnistystesteissä Java ja erityisesti .NET Core 2 kärsivät huomattavasti pidemmistä alustusajoista. .NET Core 2:n suoritusaika kasvoi yli 39 000 % lämpimään käynnistykseen verrattuna. Tekijät korostivat lisäksi, että päätelmät ovat alusta- ja asetussidonnaisia ja ympäristö kehittyy nopeasti. [78] Mittaustavan vuoksi tutkimuksesta ei kuitenkaan selviä suoritusympäristöjen keskinäinen järjestys todellisen kokonaisviiveen perusteella.

Kylmäkäynnistyksillä on selkeä taloudellinen vaikutus, koska alustus pidentää laskutettavaa kestoa erityisesti harvoin kutsutuissa funktioissa. Järjestelmällinen katsaus kokoaa näyttöä siitä, että suoritusympäristöt eroavat alustuksen raskaudessa ja että alustuksen viive voi muodostua merkittäväksi kustannustekijäksi vähäisellä kutsutiheydellä. Tällöin alustuksen tehokkuus voi vaikuttaa kokonaistaloudellisuuteen enemmän kuin suorituksen steady state -vaiheen laskentateho. [7][88]

Käytännön työkuormat ovat usein I/O-sidonnaisia. Tällöin kokonaiskesto määräytyy pääosin ulkoisten palveluiden ja tietokantayhteyksien viiveistä eikä paikallisen laskennan tehokkuudesta. Kielen valinnan vaikutus laskutettavaan kestoon jää tällaisissa tehtävissä vähäisemmäksi, ja toteutusmallin soveltuvuus kuormitusprofiiliin nousee ratkaisevaksi. FaaS- ja stream-prosessoinnin kustannusvertailu osoittaa, että CPU-sidonnaisissa kuormissa suoritustehokkuus korostuu, kun taas I/O-sidonnaisissa kuormissa keskeistä on odotusajan hallinta ja orkestrointi [89].

5.2.2 Muistin käyttöaste ja kustannukset

Muistin käyttöön liittyvät kustannukset muodostavat toisen keskeisen eron tulkattavien ja käännettyjen kielten välillä palvelimettomissa ympäristöissä. Koska palveluntarjoajat veloittavat muistia allokoidun, eivätkä käytetyn määrän perusteella, erot kielten muistin käytön tehokkuudessa näkyvät suoraan kustannusrakenteessa [13][32][44].

Käännetyt kielet hyötyvät usein pienemmästä perusmuistijalanjäljestä, mikä mahdollistaa funktioiden suorittamisen pienemmillä muistiasetuksilla [78]. Tämä voi tuoda säästöjä etenkin yksinkertaisissa funktioissa, joissa riippuvuuksia on vähän. Tulkattavilla kielillä muistivaatimuksia kasvattaa yleensä tulkin ylläpito suorituksen aikana, mikä voi nostaa minimikokoonpanon tasoa [73][74]. Erojen merkitys kuitenkin vaihtelee kieli- ja toteutuskohtaisesti: modernit tulkit voivat olla optimoituja, ja toisaalta jotkin käännettyjen

kielten ekosysteemit sisältävät raskaita kirjastoja, jotka kumoavat lähtökohtaisen etulyöntiaseman [76][77][78].

Muistin käyttöön vaikuttavat myös roskienkeruun ja tietorakenteiden ominaisuudet [76][80]. Tulkattavien kielten dynaaminen luonne ja useammin tapahtuva roskienkeruu voivat johtaa suurempiin muistin vaihteluihin, mikä pakottaa kehittäjän usein varaamaan ylimääräistä muistia luotettavuuden varmistamiseksi. Staattisesti tyypitetyissä kielissä muistin käyttö on usein ennustettavampaa ja rakenteet kompaktimpia, mikä voi alentaa kustannuksia erityisesti dataintensiivisissä funktioissa [78][80].

Serverless-alustojen hinnoittelumallit monimutkaistavat vertailua, sillä muistin määrä vaikuttaa myös laskentatehoon: suurempi muistivaraus tuo suhteessa enemmän suorittimen resursseja ja voi lyhentää suoritusaikaa [13][44]. Näin ollen muistitehokkaat kielet eivät välttämättä ole edullisimpia kaikkein pienimmällä konfiguraatiolla, vaan optimaalinen kustannustaso löytyy usein kokeilemalla eri muisti-suorituskykytasapainoja [82][83].

Kokonaisuutena muistin käyttöasteen ja kustannusten vertailu paljastaa, että kielityypin vaikutus ei rajoitu pelkkään tekniseen tehokkuuteen, vaan liittyy työkuormien ominaisuuksiin, riippuvuuksiin ja alustan hinnoittelulogiikkaan. Tästä syystä muistitehokkuus harvoin yksin määrää kielivalintaa, mutta sillä voi olla ratkaiseva rooli tietyissä sovelluksissa, joissa pienetkin erot skaalautuvat merkittäviksi kustannuksiksi [42][89].

5.2.3 Kehityksen ja ylläpidon kustannustekijät

Kehityksen ja ylläpidon kustannustekijät muodostavat useimmissa serverless-sovelluksissa suuremman kokonaisuuden kuin ajonaikaiset suorituskyky- ja resurssikustannukset [42][57]. Tämä auttaa selittämään, miksi tulkattavat kielet säilyttävät asemansa suosittuina vaihtoehtoina, vaikka ne saattavat hävitä käännetyille kielille teknisessä tehokkuudessa [62]. Ohjelmistoprojektien kustannusrakenne määräytyy usein ensisijaisesti inhimillisten ja organisatoristen tekijöiden, ei niinkään teknisten resurssierojen perusteella [63].

Kehitysnopeus on yksi keskeisistä taloudellisista ulottuvuuksista. Tulkattavilla kielillä koodia voidaan ajaa ja testata suoraan ilman erillistä käännösvaihetta, mikä nopeuttaa paikallista kehitystyötä ja mahdollistaa tiheämmät kokeilu- ja testaussyklit. [57]. Tämä nopeuttaa erityisesti alkuvaiheen toteutuksia ja prototypointia ja voi tuoda merkittäviä säästöjä henkilöstökuluissa tilanteissa, joissa markkinoille pääsyn nopeus on ratkaisevaa [63].

Tiimien osaaminen ja kehittäjämarkkinoiden tilanne vaikuttavat myös suoraan kokonaiskustannuksiin. Laajalle levinneitä kieliä, kuten JavaScriptiä ja Pythonia, tukee laaja

kehittäjäyhteisö, mikä helpottaa rekrytointia ja lyhentää perehdytysaikoja [65]. Sen sijaan vähemmän käytettyjen kielten hyödyntäminen edellyttää usein koulutusta ja siirtymävaiheen tuottavuuden laskua. Näin ollen henkilöstöön liittyvät tekijät painavat käytännön päätöksenteossa usein enemmän kuin suorituskykyhyödyt [42].

Ylläpitokustannuksiin vaikuttavat puolestaan virheenkorjaus ja vianmääritys. Tulkattavissa kielissä virheilmoitukset ja ajonaikaisen tilan tarkastelu ovat usein suoraviivaisempia, mikä helpottaa ongelmien paikantamista ja lyhentää korjausaikoja [67]. Tämä on erityisen merkittävää palvelimettomissa ympäristöissä, joissa yksittäiset funktiot toimivat osana laajoja ja hajautettuja järjestelmiä.

Koodin elinkaaren aikana erot kielityyppien välillä tulevat esiin ylläpidettävyydessä. Dynaaminen tyypitys tukee joustavuutta ja nopeaa etenemistä, mutta voi edellyttää enemmän testausta ja dokumentointia muutosten yhteydessä [59]. Staattinen tyypitys puolestaan tarjoaa vahvempia turvatakuita ja helpottaa refaktorointia, mikä voi pienentää kustannuksia erityisesti pitkäikäisissä ja usein päivitettävissä järjestelmissä [60].

Kokonaisuutta täydentää ekosysteemien ja työkalujen merkitys. Suosittujen kielten ympärillä olevat kattavat kirjasto- ja työkalupaketit vähentävät tarvetta rakentaa ratkaisuja alusta ja siten alentavat kehityskustannuksia [69][72]. Tämä etu kasvaa sovelluksen monimutkaistuessa ja integraatiovaatimusten kasvaessa.

Kaiken kaikkiaan kehityksen ja ylläpidon taloudelliset tekijät painavat useimmissa tapauksissa enemmän kuin suoritusajan kustannuserot [42][57]. Vaikka teoreettiset suorituskykyedut voivat tuoda säästöjä suorituskustannuksissa, ne kalpenevat usein viikkojen tai kuukausien säästöihin kehitys- ja ylläpitotyössä [63].

5.2.4 Kielivalintojen taloudelliset kokonaisvaikutukset

Palvelimettomien sovellusten kielivalintojen arviointi edellyttää kokonaisvaltaista taloudellista näkökulmaa, jossa huomioidaan sekä suorat resurssikulut että kehityksen, ylläpidon ja organisaation toiminnan kustannukset [42][57]. Pelkkä suorituskykyyn perustuva vertailu ei riitä, sillä kielivalintojen taloudellinen merkitys rakentuu monista eri ulottuvuuksista, jotka painottuvat eri tavoin sovelluksen käyttötarkoituksesta ja kontekstista riippuen.

Suorat pilviresurssikustannukset, kuten laskutettava suoritus- ja muistiaika, kutsumaksut ja tiedonsiirto, ovat näkyvin mutta usein pienin osa kokonaiskuluista [78][89]. Käännettyjen kielten etu resurssitehokkuudessa voi näkyä säästöinä, mutta erot jäävät monissa sovelluksissa suhteellisen vähäisiksi, etenkin jos kuorma on maltillinen tai toiminta painottuu ulkoisten palvelujen hyödyntämiseen.

Selvästi suurempi osuus kokonaiskustannuksista liittyy sovelluksen kehitykseen, käyttöönottoon ja ylläpitoon. Kehittäjien työaika, suunnittelu, testaus ja jatkuvat muutokset muodostavat huomattavan taloudellisen kokonaisuuden, jossa tulkattavat kielten etu kehitysnopeudessa ja joustavuudessa korostuu [57]. Tämä voi olla ratkaisevaa organisaatioille, joilla resurssit ovat rajalliset tai joiden kilpailukyky perustuu nopeaan markkinoille pääsyyn. Samalla ylläpitovaiheessa tulkattavien kielten tarjoama ajonaikainen läpinäkyvyys ja suoraviivaisempi virheenkorjaus voivat pienentää operointikustannuksia erityisesti hajautetuissa ja usein muuttuvissa järjestelmissä [42]. Käännetyt kielet puolestaan tarjoavat etuja pitkäikäisten järjestelmien vakaudessa ja refaktoroinnissa, mikä voi pidemmällä aikavälillä pienentää ylläpitotaakkaa [60].

Kehitysaikaan liittyvät vaihtoehtoiskustannukset muodostavat oman taloudellisen ulottuvuutensa. Viivästynyt markkinoille pääsy voi vähentää potentiaalisia tuloja, kun taas kehittäjäystävälliset kielet mahdollistavat nopean reagoinnin muuttuviin olosuhteisiin. Tämä näkökulma on erityisen merkittävä aloilla, joilla kilpailuetu perustuu kykyyn mukautua nopeasti.

Riskiin liittyvät tekijät vaikuttavat kokonaiskuvaan. Suosituilla kielillä on laajat yhteisöt, vakiintuneet käytännöt ja runsaasti tuotantokokemusta, mikä pienentää epäonnistumisen, haavoittuvuuksien ja ylläpito-ongelmien todennäköisyyttä [65]. Näiden tekijöiden rahallinen arvo on vaikeasti mitattava, mutta niiden merkitys kasvaa sovelluksissa, joissa liiketoiminnan jatkuvuus on kriittistä.

Myös skaalaustarpeet vaikuttavat. Suorituskykykriittisissä ja massiivisesti kasvavissa sovelluksissa käännettyjen kielten tehokkuusetu voi pitkällä aikavälillä tuottaa huomattavia säästöjä, vaikka kehitys olisi aluksi kalliimpaa [78]. Toisaalta epävarman tai maltillisen skaalan hankkeissa tulkattavien kielten kehitystehokkuus ja alhaisempi alkuinvestointi tuottavat usein paremman kokonaisarvon [42].

Kokonaisvaltainen taloudellinen tarkastelu osoittaa, miksi kielivalinnat vaihtelevat organisaatioiden kontekstien ja prioriteettien mukaan [42][57]. Startup-yrityksille nopeus ja joustavuus voivat olla tärkeämpiä kuin resurssitehokkuus, kun taas suurissa yrityksissä, joilla on raskaita ja suorituskykykriittisiä kuormia, resurssikustannusten optimointi voi nousta keskiöön [89]. Yhtä yleispätevää ratkaisua ei ole, vaan kielivalinnat heijastavat tasapainoa suorituskyvyn, kehityksen, ylläpidon, riskien ja skaalautumisen taloudellisten näkökulmien välillä.

6. KIRJALLISUUSKATSAUKSEN TULOKSET

Tässä kirjallisuuskatsauksessa tarkasteltiin serverless-arkkitehtuurien kehitystä, eri ohjelmointikielten käytön jakautumista serverless-ympäristöissä sekä kielivalinnan vaikutusta kustannuksiin ja suorituskykyyn. Kirjallisuuskatsauksen tulokset osoittavat, että tulkattavien kielten, kuten JavaScriptin ja Pythonin osuus on suuri nykyaikaisissa serverless-toteutuksissa. [78][8]

Yksi keskeinen havainto on, että kehitysnopeus ja helppokäyttöisyys ovat ratkaisevia tekijöitä ohjelmointikielten valinnassa. Tulkattavien kielten, erityisesti JavaScriptin (Node.js) ja Pythonin, suosio perustuu vahvasti niiden laajoihin kirjastoekosysteemeihin, matalaan oppimiskynnykseen ja nopeaan prototypointiin. [90][8] Nämä ominaisuudet tukevat hyvin serverless-arkkitehtuurin tavoitteita, kuten nopeaa käyttöönottoa, pienikokoisia funktioita ja iteratiivista kehitystä. Toisaalta käännetyt kielet, kuten Go ja Java, osoittautuivat kustannustehokkaammiksi erityisesti laskentaintensiivisissä työkuormissa. Käännetyt kielet mahdollistavat lyhyemmät suoritukset ja pienemmän muistinkulutuksen, mikä voi tuoda säästöjä suurivolyymisissa palveluissa, joissa jokaisen funktion kustannuksella on merkitystä. [78][88]

Työssä havaittiin, etteivät kustannusvaikutukset johdu pelkästään kielivalinnasta, vaan kokonaisuuteen vaikuttavat myös pilvipalveluiden hinnoittelumallit, käynnistysviiveet ja mahdollisuus hyödyntää esilämmitystekniikoita, kuten AWS Provisioned Concurrency tai Azure Premium Plan [7]. Tulkattavien kielten heikompi suorituskyky voidaan usein kompensoida hyvällä arkkitehtuurilla, tehokkaalla funktioiden jaettavuudella sekä optimoidulla resurssien käytöllä [91].

Palveluntarjoajien tarjoama kielituki ja ekosysteemin kypsyys vaikuttavat merkittävästi myös kielivalintoihin. AWS Lambda tukee laajasti kieliä ja mahdollistaa omien suoritusympäristöjen määrittämisen, kun taas Azure Functions on optimoitu erityisesti .NET-ympäristölle. Google Cloud Functions tarjoaa puolestaan vahvat integraatiot Googlen data- ja tekoälypalveluihin, mikä voi vaikuttaa valintaan sovelluskohtaisesti. [8]

Lisäksi työssä havaittiin, että kielivalintoihin vaikuttavat myös erilaiset kehittäjäyhteisöjen trendit. OpenLambdaVerse-analyysi osoittaa, että Go ja Rust ovat kasvattaneet suosiotaan infrastruktuurilähtöisissä projekteissa, mutta silti JavaScript ja Python säilyttävät edelleen selvästi hallitsevan aseman [61]. Tämä kehitys korostaa, että ohjelmointikielen valintaan vaikuttavat teknisten ominaisuuksien ohella myös yhteisöjen koko, kypsyys ja ekosysteemin tuki.

Kustannustehokkuuden näkökulmasta kirjallisuus tuo esiin, että optimaalinen ratkaisu riippuu sovelluksen kuormitusprofiilista ja toteutusmallista. Pfandzelter ja Bermbach vertasivat FaaS- ja stream-prosessointimalleja ja havaitsivat, että kustannustehokkuus määräytyy kuormituksen vaihtelun ja toimintaympäristön mukaan. [89] Näin ollen sekä tekniset että taloudelliset tekijät on otettava huomioon samanaikaisesti, kun arvioidaan ohjelmointikielen sopivuutta serverless-toteutuksiin.

Kaiken kaikkiaan kirjallisuus osoittaa, että ohjelmointikielen valinta palvelimettomissa ympäristöissä muodostuu useiden tekijöiden yhteisvaikutuksesta. Teknisten ominaisuuksien lisäksi ratkaisevia ovat kehittäjäyhteisön tuki, alustakohtainen ekosysteemi ja palveluntarjoajien hinnoittelumallit.

7. JOHTOPÄÄTÖKSET

7.1 Keskeiset havainnot

Tämän diplomityön tavoitteena oli selvittää, miksi tulkattavat ohjelmointikielet ovat saavuttaneet laajan suosion serverless-pilviympäristöissä ja millaisia kustannusvaikutuksia kielivalinnoilla on. Työn perusteella voidaan todeta, että tulkattavien kielten suosio perustuu ennen kaikkea kehitystyön nopeuteen, matalaan oppimiskynnykseen ja laajaan kirjastotukeen. Nämä ominaisuudet tukevat serverless-kehityksen luonnetta, jossa korostuvat nopea iterointi, pienet funktiot ja jatkuva käyttöönotto.

Vaikka käännetyt kielet tarjoavat parempaa suorituskykyä ja resurssitehokkuutta, ei tämä yksin riitä syrjäyttämään tulkattavia kieliä. Käytännössä kehittäjien tuottavuus, valmiiden työkalujen saatavuus ja tuttuus teknologian kanssa painavat usein enemmän kuin yksittäisten funktioiden raakasuorituskyky. Tämän seurauksena tulkattavat kielet ovat säilyttäneet vahvan aseman erityisesti sovelluksissa, joissa kehityksen ketteryys ja laaja yhteisötuki ovat keskeisiä.

7.2 Ohjelmointikielen valinnan vaikutukset suorituskykyyn ja kustannuksiin

Ohjelmointikielen vaikutus serverless-sovellusten suorituskykyyn ja kustannuksiin korostuu erityisesti silloin, kun funktiot ovat pitkiä, laskentaintensiivisiä tai niitä suoritetaan suurivolyymisesti. Käännetyt kielet hyötyvät paremmasta suorituskyvystään ja voivat tarjota merkittäviä säästöjä suurissa käyttövolyymeissä. Tulkattavat kielet sen sijaan mahdollistavat nopeamman kehityksen ja pienemmät ylläpitokustannukset, jotka voivat kompensoida korkeampia ajonaikaisia kustannuksia.

Kustannuksiin vaikuttavat myös palveluntarjoajan ominaisuudet, kuten kylmäkäynnistyksen hallinta, kielikohtainen suoritusympäristö tuki ja skaalausmekanismit. Esimerkiksi AWS Lambda, Azure Functions ja Google Cloud Functions eroavat merkittävästi siinä, miten ne optimoivat eri kielten suoritusta. Näin ollen ohjelmointikielen taloudellista ja suorituskykyvaikutusta ei voida tarkastella irrallaan alustan teknisistä ratkaisuista.

Serverless-ympäristöjen jatkuva kehitys ja hinnoittelumallien monimutkaistuminen viittaavat siihen, että kielivalinnoista tulee tulevaisuudessa entistä dynaamisempia. Kielen valinta ei ole pysyvä päätös, vaan siihen voidaan vaikuttaa sovelluksen elinkaaren eri vaiheissa esimerkiksi muuttamalla suoritusympäristöä tai optimoimalla funktiorakennetta.

7.3 Jatkotutkimusaiheet

Tämä työ avaa useita jatkotutkimusmahdollisuuksia, jotka voivat syventää ymmärrystä serverless-kielivalintojen vaikutuksista.

Ensinnäkin tarvitaan tarkempaa empiiristä mittausta eri kielten kustannus- ja suorituskykyeroista nykyaikaisilla serverless-alustoilla, kuten AWS Lambda, Azure Functions ja Google Cloud Functions. Tämä auttaisi validoimaan kirjallisuuskatsauksen havaintoja käytännön tasolla.

Jatkotutkimuksessa voitaisiin myös tarkastella, miten työkuorman tyyppi (esimerkiksi I/O-sidonnaisuus tai CPU-sidonnaisuus) vaikuttaa optimaaliseen kielivalintaan ja millaisia kompromisseja eri tilanteissa syntyy.

Lisäksi olisi mahdollista tutkia, miten konttipohjaiset serverless-ratkaisut, kuten AWS Lambda Docker Images tai Kubernetes-pohjaiset FaaS-ympäristöt muuttavat kielivalintojen joustavuutta ja siirrettävyyttä eri pilvialustojen välillä.

LÄHTEET

[1] Amazon Web Services, "Serverless Architectures," [Internet]. [viitattu 3.3.2025]. Saatavilla: https://aws.amazon.com/lambda/serverless-architectures-learn-more/

[2] Koschel A, Klassen S. Cloud Computing: Serverless. *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE; 2021. s. 1–8. doi:10.1109/IISA52424.2021.9555534

[3] E. Marin, D. Perino, R. Di Pietro, Serverless computing: a security perspective, Journal of Cloud Computing, 2022, https://doi.org/10.1186/s13677-022-00347-w

[4] Lichtenthäler R, Winzinger S, Manner J, Wirtz G. When to use FaaS? - Influencing Technical Factors for and against using Serverless Functions. CEUR-WS.org; 2020. [viitattu 10.12.2025]. Saatavissa: https://ceur-ws.org/Vol-2575/paper7.pdf

[5] Ship H, Shindin E, Wang C, Arroyo D, Tantawi A. Optimizing simultaneous autoscaling for serverless cloud computing. Proc. 2024 IEEE 17th Int. Conf. Cloud Comput. (CLOUD); 2024. doi:10.1109/CLOUD62652.2024.00022

[6] Vahidinia, P., Farahani, B. and Aliee, F. S. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. 2020 International Conference on Omnilayer Intelligent Systems (COINS). 2020, pp. 1–7. doi:10.1109/COINS49042.2020.9191377.

[7] Golec M, Walia GK, Kumar M, Cuadrado F, Gill SS, Uhlig S. Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions. ACM Comput. Surv. 57, 3, Article 65 (March 2025), 36 pages. doi:10.1145/3700875.

[8] Ekwe-Ekwe N, Amos L. The state of FaaS: an analysis of public Functions-as-a-Service providers. *arXiv preprint* arXiv:2408.03021. 2024

[9] Amazon Web Services. Introducing AWS Lambda. re:Invent 2014. [viitattu 10.12.2025]. Saatavilla: https://aws.amazon.com/lambda/

[10] Datadog. State of Serverless 2023. [viitattu 10.12.2025]. Saatavilla: https://www.datadoghq.com/state-of-serverless/

[11] Amazon Web Services. Lambda runtimes. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[12] Amazon Web Services. Go support deprecation and custom runtimes. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/lambda-golang.html

[13] Amazon Web Services. Lambda quotas. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[14] Amazon Web Services. Configuring AWS Lambda functions. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-ephemeral-storage

[15] Amazon Web Services. Lambda deployment packages. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-package.html

[16] Amazon Web Services. Invoking Lambda with events from other AWS services. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

[17] Amazon Web Services. Using API Gateway with Lambda. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-lambda-api.html

[18] Amazon Web Services. Enabling Amazon EventBridge. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/AmazonS3/latest/userguide/enable-event-notifications-eventbridge.html

[19] Amazon Web Services. DynamoDB Streams and Lambda. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.Lambda.html

[20] Amazon Web Services. Amazon SQS and Lambda. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html

[21] Amazon Web Services. Event bus targets in Amazon EventBridge. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/eventbridge/latest/userguide/eb-targets.html

[22] Amazon Web Services. Invoke an AWS Lambda function with Step Functions. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/step-functions/latest/dg/connect-lambda.html

[23] Amazon Web Services. Managing Lambda dependencies with layers. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html

[24] Amazon Web Services. Monitoring, debugging, and troubleshooting Lambda functions. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/lambda-monitoring.html

[25] Amazon Web Services. AWS Lambda pricing. [viitattu 10.12.2025]. Saatavilla: aws.amazon.com/lambda/pricing

[26] Amazon Web Services. Improving startup performance with Lambda SnapStart. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/lambda/latest/dg/snapstart.html

[27] Amazon Web Services. Compile .NET Lambda function code to a native runtime format. [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/lambda/latest/dg/dotnet-native-aot.html

[28] Amazon Web Services. Announcing improved VPC networking for AWS Lambda functions. AWS Compute Blog. 2019. [viitattu 10.12.2025]. Saatavilla:

aws.amazon.com/blogs/compute/announcing-improved-vpc-networking-for-aws-lambda-functions

[29] Castro P, Ishakian V, Muthusamy V, Slominski A. The rise of serverless computing. Commun ACM. 2019;62(12):44–54. doi:10.1145/3368454

[30] Microsoft. Supported languages in Azure Functions. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/supported-languages

[31] Manner J, Endreß M, Peuster M, Karl H. Cold start influencing factors in Function as a Service. In: IEEE CLOUD; 2018. doi:10.1109/UCC-Companion.2018.00054

[32] Microsoft. Azure Functions Premium plan. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan

[33] Microsoft. Azure Functions hosting options. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-scale

[34] Microsoft. Dedicated hosting plans for Azure Functions. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/dedicated-plan

[35] Microsoft. Azure Functions triggers and bindings. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings

[36] Microsoft. What is Azure API Management? [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/api-management/api-management-key-concepts

[37] Microsoft. What are Durable Functions? [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[38] Microsoft. Develop Azure Functions using Visual Studio. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-develop-vs

[39] Microsoft. Continuous delivery by using GitHub Actions. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-how-to-github-actions

[40] Microsoft. Develop Azure Functions locally using Core Tools. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-run-local

[41] Microsoft. Azure Functions pricing. [viitattu 10.12.2025]. Saatavilla: azure.microsoft.com/en-us/pricing/details/functions

[42] Eismann S, Scheuner J, van Eyk E, Schwinger M, Herbst N, Kounev S. Serverless applications: Why, when, and how? IEEE Software. 2021;38(1):32–39. doi:10.1109/MS.2020.3023302

[43] Google Cloud. Cloud Functions runtimes. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/functions/docs/concepts/execution-environment

[44] Google Cloud. Configure memory limits for services. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/functions/docs/configuring/memory

[45] Google Cloud. Functions overview. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/functions/docs/concepts/overview

[46] Google Cloud. Cloud Functions triggers. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/functions/docs/calling

[47] Google Cloud. Cloud Functions 2nd generation general availability. Cloud Blog. 9 Aug 2022. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/blog/products/serverless/cloud-functions-2nd-generation-now-generally-available

[48] Google Cloud. What is Cloud Run. [viitattu 10.12.2025]. Saatavilla: https://docs.cloud.google.com/run/docs/overview/what-is-cloud-run

[49] Amazon Web Services. Building a custom runtime for AWS Lambda. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html

[50] Google Cloud. Integrate with Cloud databases. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/run/docs/integrate-functions-with-cloud-databases

[51] Google Cloud. Cloud Functions pricing. [viitattu 10.12.2025]. Saatavilla: https://cloud.google.com/run/pricing?hl=fi

[52] Amazon Web Services. What is the AWS Serverless Application Model (AWS SAM)? [viitattu 10.12.2025]. Saatavilla: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html

[53] Microsoft. Azure Monitor overview. [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-monitor/overview

[54] Google Cloud. Cloud Monitoring. [viitattu 10.12.2025]. Saatavilla: cloud.google.com/monitoring

[55] Gal A, Eich A, Shaver M, Anderson D, Mandelin D, Haghighat MR, et al. Trace-based just-in-time type specialization for dynamic languages. ACM SIGPLAN Notices. 2009;44(6):465–78. doi:10.1145/1543135.1542528

[56] Blaquiere G. Serverless performance comparison: Does the language matter? [Internet]. Medium; 2021 Feb 11 [viitattu 12.04.2025]. Saatavilla: https://medium.com/google-cloud/serverless-performance-comparison-does-the-language-matter-c72a7191c799

[57] Jonas E, Schleier-Smith J, Sreekanti V, Tsai C-C, Khandelwal A, Pu Q, et al. Cloud programming simplified: A Berkeley view on serverless computing. arXiv preprint arXiv:1902.03383. 2019.

[58] Pierce BC. Types and programming languages. Cambridge (MA): MIT Press; 2002.

[59] Meijer E, Drayton P. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In: ACM OOPSLA Companion. 2004. p. 2–26. Saatavilla: researchgate.net/publication/213886116

[60] Schreiber A, Schiller T, Cambronero JM, Erich F, Proksch S. To type or not to type? A systematic comparison of the software quality of JavaScript and TypeScript applications on GitHub. ACM; 2022. doi:10.1145/3524842.3528454

[61] Chávez-Moreno ÁC, Abad CL. OpenLambdaVerse: A dataset and analysis of open-source serverless applications. In: 2025 IEEE International Conference on Cloud Engineering (IC2E). IEEE; 2025. doi:10.1109/IC2E65552.2025.00032

[62] Meyerovich L, Rabkin A. Empirical analysis of programming language adoption. In: ACM SIGPLAN; OOPSLA 2013. doi:10.1145/2509136.2509515

[63] Pano A, Graziotin D, Abrahamsson P. Factors and actors leading to the adoption of a JavaScript framework. Empir Softw Eng. 2018;23(6):3503–34. doi:10.1007/s10664-018-9613-x

[64] Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, et al. Serverless computing: Current trends and open problems. arXiv:1706.03178

[65] Bissyandé TF, Thung F, Lo D, Jiang L, Réveillère L, Klein J, Le Traon Y. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: 2013 IEEE 37th Annual Computer Software and Applications Conference; 2013. p. 303–12. doi:10.1109/COMPSAC.2013.55

[66] Stack Overflow. Developer Survey 2024. [viitattu 10.12.2025]. Saatavilla: survey.stackoverflow.co/2024

[67] Maffeis S, Mitchell JC, Taly A. An operational semantics for JavaScript. In: Proceedings of the 6th Asian Symposium on Programming Languages and Systems (APLAS). Springer; 2008. p. 307–25. doi:10.1007/978-3-540-89330-1_22

[68] Richards G, Hammer C, Burg B, Vitek J. The eval that men do: A large-scale study of the use of eval in JavaScript applications. ACM SIGPLAN Notices. 2011;46(6):1–12. doi:10.5555/2032497.2032503

[69] Serverless Inc. Serverless Framework Documentation. [viitattu 10.12.2025]. Saatavilla: serverless.com/framework/docs

[70] Amazon Web Services. AWS Cloud Development Kit (CDK). [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/cdk

[71] Amazon Web Services. What is the AWS Serverless Application Model (AWS SAM)? [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli.html

[72] Microsoft. What is Azure Functions? [viitattu 10.12.2025]. Saatavilla: learn.microsoft.com/en-us/azure/azure-functions/functions-overview

[73] Wang L, Li M, Zhang Y, Ristenpart T, Swift M. Peeking behind the curtains of serverless platforms. 2018. doi:10.5555/3277355.3277369

[74] Yu T, Liu Q, Du D, Xia Y, Zang B, Lu Z, et al. Characterizing serverless platforms with ServerlessBench. In: Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20). ACM; 2020. p. 30–44. doi:10.1145/3419111.3421280

[75] Amazon Web Services. Configuring provisioned concurrency for a function. [viitattu 10.12.2025]. Saatavilla: docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

[76] Wimmer C, Franz M. Linear scan register allocation on SSA form. ACM SIGPLAN Notices. 2010;45(6):170–180. doi:10.1145/1772954.1772979

[77] PyPy. What is PyPy? [viitattu 10.12.2025]. Saatavilla: pypy.org/features.html

[78] Jackson D, Clynch G. An investigation of the impact of language runtime on the performance and cost of serverless functions. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE; 2018. doi:10.1109/UCC-Companion.2018.00050

[79] Wang Z, Skarlatos D, Kim H, et al. Memento: Architectural support for ephemeral memory management in serverless environments. In: Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO). 2023. doi:10.1145/3613424.3623795

[80] Xu C, Liu Y, Li Z, et al. FaaSMem: Improving memory efficiency of serverless computing with memory pool architecture. In: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2024. doi:10.1145/3620666.3651355

[81] Tzenetopoulos A, Psomakelis E, Sakkopoulos E, et al. Towards elastic memory allocation of serverless functions. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). 2025. doi:10.1145/3723851.3723855

[82] Bilal M, Rodrigues R, Pietzuch P. Rethinking resource allocation for serverless functions. In: Proceedings of the ACM European Conference on Computer Systems (EuroSys). 2023. doi:10.1145/3552326.3567506

[83] Eismann S, Grohmann J, Herbst N, Kounev S. Sizeless: Predicting the optimal size of serverless functions. arXiv preprint. 2020. arXiv:2010.15162

[84] Ustiugov D, Jesalpura S, Alper MB, Baczun M, Feyzkhanov R, Bugnion E, Grot B, Kogias M. Shattering the Ephemeral Storage Cost Barrier for Data-Intensive Serverless Workflows. arXiv preprint. 2023. arXiv:2309.14821.

[85] Mahgoub A, Shankar K, Mitra S, Klimovic A, Chaterji S, Bagchi S. Application-aware Data Passing for Chained Serverless. Proc USENIX Annual Technical Conference (USENIX ATC). 2021. [viitattu 10.12.2025]. Saatavilla: https://www.usenix.org/system/files/atc21-mahgoub.pdf

[86] Saha D, Talluri S, Jansen M, Trivedi A. Survey of Serverless Workflows [preprint]. AtLarge Research; 2024. [viitattu 10.12.2025]. Saatavilla: https://atlarge-research.com/pdfs/2024-dsaha-litsurvey.pdf

[87] Sreekumar N, Chandra A, Weissman JB. Locality, Latency and Spatial-Aware Data Placement Strategies at the Edge. arXiv preprint. 2022. arXiv:2212.01984.

[88] J. Scheuner, P. Leitner, Function-as-a-Service performance evaluation: A multivocal literature review, Journal of Systems and Software, 2020, https://doi.org/10.1016/j.jss.2020.110708.

[89] T. Pfandzelter, S. Henning, T. Schirmer, W. Hasselbring and D. Bermbach, "Streaming vs. Functions: A Cost Perspective on Cloud Event Processing", 2022 IEEE International Conference on Cloud Engineering (IC2E), 2022, doi: 10.1109/IC2E55432.2022.00015.

[90] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. ACM Trans. Softw. Eng. Methodol. 32, 5, Article 131 (September 2023), 61 pages. https://doi.org/10.1145/3579643

[91] Eismann S, Grohmann J, van Eyk E, Herbst N, Kounev S. Predicting the costs of serverless workflows. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE). New York (NY): Association for Computing Machinery; 2020. doi:10.1145/3358960.3379133.