Mikael Lehmuskoski

Vertailu Golangin ja Typescriptin käytöstä mikropalveluarkkitehtuurissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja Viestintätekniikka

Insinöörityö

20.05.2024

Tiivistelmä

Tekijä:	Mikael Lehmuskoski
Otsikko:	Vertailu Golangin ja Typescriptin käytöstä mikropalveluarkkitehtuurissa
Sivumäärä:	25 sivua
Aika:	20.05.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja Viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Vesa Ollikainen

Opinnäytetyössä selvitettiin Golangin ja Typescriptin soveltuvuutta mikropalveluarkkitehtuurin kehittämiseen.

Alussa tarkasteltiin Golangin ja Typescriptin eroja ja samankaltaisuuksia. Sen jälkeen suoritettiin käytännön suorituskykyvertailu pilvilaskenta-alustalla. Vertailun perusteella laadittiin analyysi ja suositukset. Tavoitteena oli nopeuttaa tehokkaan työnkulun perustamista ja suorituskyvyn optimointia.

Opinnäytetyö saavutti asetetut tavoitteet taustaselvityksen, suorituskykyvertailun ja suositusten osalta. Ohjelmointikielten teknisten erojen syvällisempään tarkasteluun ei kuitenkaan pureuduttu, mikä heijastui suorituskykyvertailuun, analyysiin ja suosituksiin.

Vaikka työssä ei syvennytty teknisiin eroihin, sen tulokset kuitenkin mahdollistavat nopeamman ohjelmointikielen valitsemisen kehitystyön työnkulkujen perustamisen ja suorituskykyoptimoinnin osana. Työn aikana hankittiin myös ymmärrystä AWS Lambdasta ja siihen liittyvistä työnkuluista.

Vaikka taustaselvitys ja suorituskykyvertailu vahvistivat aiempia käsityksiä Golangin ja Typescriptin eroista, niiden raa'an suorituskyvyn kontrasti yllätti.

Avainsanat: Mikropalvelut, Golang, Typescript

Abstract

Author:	Mikael Lehmuskoski
Title:	Comparative Study of Golang and Typescript in Microservices Architecture
Number of Pages:	25 pages
Date:	20 May 2024
Degree:	Bachelor of Engineering (BEng)
Degree Programme:	Information and Communications Technology
Professional Major:	Software Engineering
Supervisors:	Vesa Ollikainen, Senior Lecturer

The study aimed to investigate the suitability of Golang and Typescript for developing microservices architecture.

Initially, the differences and similarities between these languages were examined. Subsequently, a practical performance comparison was conducted on a cloud computing platform. Based on the comparison, an analysis was made and recommendations suggested. The goal was to expedite the establishment of an efficient workflow and the optimization of performance.

The study partially achieved its objectives, particularly in terms of background research, performance comparison, and recommendations. However, a more in-depth examination of the technical differences between programming languages was lacking, which was reflected in the performance comparison, analysis, and recommendations.

Although the study does not delve deeply into the technical differences between the languages, its results nevertheless aid in facilitating the faster selection of a programming language as part of workflow setup and performance optimization.

While the background research and performance comparison reinforced previous understanding of the differences between Golang and Typescript, the contrast in their raw performance was surprising.

Keywords: Microservices, Golang, Typescript

Sisällys

Lyhenteet

AWS: *Amazon Web Services.* Amazon.comin tarjoama pilvipalvelualusta.

EC2: *Amazon Elastic Compute Cloud*. AWS:n virtuaalikone.

HTTP: *Hypertext Transfer Protocol.* Internetin toteutuksessa käytetty tilaton protokolla.

JWT: *JSON Web Token.* Käyttöoikeustietueiden hallinnointiin käytetty avoin standardi.

REST: *Representational State Transfer.* Yleinen arkkitehtuurityyli ohjelmointirajapintojen toteuttamiseen.

1 Johdanto

Modernin ohjelmistokehityksen dynaamisessa ympäristössä mikropalveluarkkitehtuuri on noussut paradigmaksi, joka tarjoaa skaalautuvuutta ja joustavuutta monimutkaisille tietojärjestelmille. Mikropalvelut, joille ominaista on niiden modulaarinen ja nopeasti käyttöön otettava luonne, mahdollistavat helposti skaalautuvien palveluiden kehittämisen.

Tässä työssä vertaillaan kahta johtavista ohjelmointikielistä, Golang (Go) ja Typescript, mikropalveluarkkitehtuurin kontekstissa. Golang on ytimekkään syntaksin ja hämmästyttävän suorituskyvyn ansiosta saavuttanut nopeasti suosiota palvelinpuolen ohjelmistokehityksessä. Typescript taas on lähtöisin koko pinon (engl. full stack) web-kehityksen puolelta. Javascriptin laajennuksena Typescriptin staattinen tyypitys auttaa suurten web-pohjaisten asiakas- ja palvelinohjelmistojen luomisessa. Typescriptin kattava ekosysteemi ja joustavuus tekevät siitä houkuttelevan vaihtoehdon mikropalvelujen rakentamiseen.

Työssä vertaillaan kieliä kolmen testitapauksen avulla. Testit suoritetaan palvelimettoman pilvilaskennan kiistattoman markkinajohtajan (Richter 2024), Amazon Web Servicesin (AWS), Lambda-alustalla, jonka avulla voidaan suorittaa ohjelmakoodia ilman palvelimien hallinnointia ja ylläpitämistä.

Opinnäytetyön ensisijaisena tavoitteena on tarjota vertaileva analyysi Golangista ja Typescriptistä, mikä antaa kattavan käsityksen niiden vahvuuksista ja heikkouksista, sekä yleisestä käyttäjäkokemuksesta mikropalveluarkkitehtuurin kontekstissa.

Kolmen testitapauksen tarkoituksena on arvioida kieliä skenaarioissa, jotka heijastavat todellisia mikropalveluhaasteita, kuten REST-pyyntöjen käsittelyä, resurssien käytön optimointia ja saumatonta skaalautuvuutta. Empiirisen analyysin ja suorituskykyvertailun avulla opinnäytetyö pyrkii antamaan arvokkaita oi-

valluksia kehittäjille ja organisaatioille, jotka ovat suunnittelemassa uutta mikropalveluarkkitehtuuriin pohjautuvaa projektia.

2 Taustaa

Pilvitekniikan kehitys on mullistanut tavan, jolla sovelluksia kehitetään, otetaan käyttöön ja hallitaan. Pilvilaskenta-alustoja käyttämällä voidaan jaetun vastuun mallin mukaisesti käyttöönottaa tismalleen halutun määrän käyttäjävastuuta ja pääomakuluja vaativa alusta, joka mahdollistaa ketterämmän ohjelmistokehityksen ja nopeamman ohjelmakoodin tuotteistamisen kuin perinteiset arkkitehtuuriratkaisut. Yksi tämän muutoksen airut on mikropalveluarkkitehtuuri (Microservices (A) 2024).

2.1 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuurissa laajat sovellukset koostuvat pienistä, löyhästi kytketyistä palveluista, joista jokainen vastaa tietystä toiminnosta (What are microservices? 2024; Microservices (B) 2024).

Tämä lähestymistapa tarjoaa ohjelmistokehittäjille merkittäviä etuja. Kun monoliittinen sovellus hajautetaan pienempiin palveluihin, niitä voidaan kehittää, ottaa käyttöön ja skaalata itsenäisesti, mikä mahdollistaa nopeamman toimituksen, hienojakoisemman ylläpidon ja paremman vikasietoisuuden. (What are the benefits of a microservices architecture? 2022.) Mainittakoon, että tämä vaatii erityistä tarkkuutta mikropalveluiden rajapintoihin ja niitä kehittävien tiimien viestintään. Parhaimmillaan mikropalveluarkkitehtuuri toimii silloin kun jokaista palvelua kehittää sille omisteinen tiimi. (Richardson 2020.)

Pilkkomalla sovellukset pienempiin ja toisistaan erillisiin yksiköihin, mikropalveluarkkitehtuuri tekee ajonaikaisten virheiden monitoroinnista, diagnosoinnista ja korjaamisesta helpompaa. Lisäksi päivitykset voidaan suorittaa yksittäisille palveluille ilman, että koko sovelluksen toiminta häiriintyy (Microservices (B) 2024). Tämä vähentää käyttökatkoja ja parantaa käyttäjäkokemusta.

Mikropalveluarkkitehtuuri parantaa sovellusten vikasietoisuutta. Jos palveluista yksi kohtaa katastrofaalisen virheen, se ei välttämättä vaikuta muihin, ihannetapauksessa täysin itsenäisesti toimiviin palveluihin. Lisäksi mikropalveluiden skaalautuvuus mahdollistaa resurssien dynaamisen jakamisen tarpeen mukaan, mikä auttaa ylläpitämään suorituskykyä, myös kuormituksen kasvaessa ja laajempien käyttökatkosten ilmetessä. (Microservices (B) 2024.)

2.2 Golang

Golang, lyhyemmin Go, on staattisesti tyypitetty ja käännettävä ohjelmointikieli, jonka Google kehitti alun perin vuonna 2007, ja julkaisi ensimmäisen kerran vuonna 2012 (Golang 2024). Sen suunnittelussa keskiössä on yksinkertaisuus ja tehokkuus, mikä tekee siitä houkuttelevan vaihtoehdon moniin sovelluskehitystarpeisiin.

Go suunniteltiin parantamaan ohjelmoinnin tuottavuutta moniytimisten, verkkoon kytkettyjen koneiden ja suurten koodikantojen aikakaudella. Suunnittelijat halusivat paikata muiden Googlen käyttämien kielien heikkouksia, mutta säilyttää niiden hyödylliset ominaisuudet. (Go (ohjelmointikieli) 2024.)

Yksi Golangin merkittävimmistä eduista on sen suorituskyky ja skaalautuvuus. Tämä tekee siitä ihanteellisen valinnan korkeaa suorituskykyä vaativien järjestelmien rakentamiseen. Suurin osa Golangin suorituskyvystä tulee sen kääntämisestä konekielelle, mikä tuo huiman määrän suorituskykyä verrattuna tulkattaviin ohjelmointikieliin (Golang 2024).

Golangin suunnittelussa on kiinnitetty erityistä huomiota myös rinnakkaisuuden toteuttamiseen, mikä on yksi syy Golangin menestykselle (Jefferson 2023). Golang tarjoaa vankan joukon sisäänrakennettuja standardikirjastoja, mukaan lukien tehokkaat rinnakkaisuusprimitiivit, kuten gorutiinit. Nämä gorutiinit tarjoavat kevyen ja tehokkaan tavan toteuttaa samanaikaisesti suorittavia komponentteja Golang-ohjelmien sisällä. Koodiesimerkissä 1 on leikelmä ohjelmasta, joka tulostaa konsoliin sanat ”world” ja ”hello” viidesti samanaikaisuutta käyttäen.

```
func say(s string) {
     for i := 0; i < 5; i++ {
          time.Sleep(100 * time.Millisecond)
          fmt.Println(s)
     }
}

func main() {
     go say("world")
     say("hello")
}
```

Esimerkkikoodi 1: Goroutinen käyttäminen (Goroutines 2024).

Gorutiinit, joita kuvaillaan usein kevyinä säikeinä, antavat kehittäjille mahdollisuuden luoda samanaikaisia toimintoja, jotka voidaan suorittaa itsenäisesti. Gorutiinit jakavat ohjelman käyttöjärjestelmän säikeiksi, joiden pinon kokoa voidaan säätää dynaamisesti, mikä vähentää säikeiden luomiseen ja hallintaan liittyviä lisäkustannuksia. Tämä suunnitteluvalinta ei ainoastaan säästä järjestelmäresursseja, vaan myös helpottaa erittäin tehokkaiden sovellusten luomista. (Ankit 2023.)

Lisäksi Golangin standardikirjasto tarjoaa lisätukea rinnakkaisuudelle rakenteiden, kuten kanavien, avulla, jotka helpottavat viestintää ja synkronointia gorutiinien välillä (Ankit 2023). Kanavat mahdollistavat turvallisen ja tehokkaan tiedonvaihdon samanaikaisten prosessien välillä, mikä parantaa samanaikaisuutta toteuttavien Golang-sovellusten yleistä kestävyyttä ja luotettavuutta.

Hyödyntämällä gorutiineja kehittäjät voivat suunnitella mikropalveluita, joilla voidaan toteuttaa samanaikaisuutta vaativia toimintoja helposti, mitkä maksimoivat suorituskyvyn ja minimoivat latenssin.

Golangin standardikirjasto tarjoaa monipuolisia toimintoja myös verkkopalvelimien rakentamiseen ja HTTP-pyyntöjen käsittelyyn (Golang 2024). Tämä tekee mikropalveluiden rajapintojen kehittämisestä sujuvaa ja tehokasta. Lisäksi Golangin ulkoisten kirjastojen ekosysteemi on aktiivinen ja kehittäjillä on hyödynnettävissä laajalla skaalalla erinäisiä avoimen lähdekoodin kirjastoja ja työkaluja, jotka nopeuttavat kehitystyötä ja ratkaisevat moninaisia yleisesti kohdattuja ongelmia. Golang käyttää git-versionhallintajärjestelmää sisäänrakennetussa

pakettienhallintaohjelmistossaan. Tämä helpottaa huomattavasti ulkoisten kirjastojen käyttöä ja sitä pidetään yhtenä Golangin suunnittelun parhaista päätöksistä.

Vaikka Golangilla on vahvoja etuja, sillä on myös joitain heikkouksia. Esimerkiksi sen syntaksin tiiviys ja rajoitettu perintötyyppien tuki voivat luoda haasteellisen käyttäjäkokemuksen joillekin kehittäjille, erityisesti niille, jotka ovat tottuneet muihin monasti monisanaisempiin ja eksplisiittisempiin kieliin, kuten Javascriptiin tai Pythoniin. Lisäksi Golangin ekosysteemi ei välttämättä ole yhtä laaja kuin joillakin muilla kielillä, mikä saattaa joskus rajoittaa saatavilla olevien kirjastojen ja työkalujen valikoimaa tietyissä käyttötarkoituksissa.

Golang tarjoaa tehokkaan ja yksinkertaisen lähestymistavan ohjelmistokehitykseen, erityisesti mikropalveluarkkitehtuurin alueella. Sen suorituskyky, rinnakkaisuustuki ja kattava vakiokirjasto tekevät siitä houkuttelevan vaihtoehdon monille kehittäjille, vaikka sen tiivis syntaksi ja rajoitetumpi ekosysteemi voivat näyttäytyä haasteena joillekin.

2.3 Typescript

Typescript on Microsoftin kehittämä ja vuonna 2012 julkaisema staattisesti tyypitetty JavaScriptiä laajentava ohjelmointikieli (Typescript 2024). Se syntyi tarpeesta vahvistaa JavaScriptin tyyppiturvallisuutta, erityisesti laajoissa web-pohjaisissa ohjelmistoissa. Typescriptin vahvan tyypityksen avulla kehittäjät voivat havaita virheet ohjelmistokehityksen varhaisessa vaiheessa, mikä vähentää ajonaikaisia virheitä ja parantaa ohjelmiston laatua. Tämä ominaisuus on erityisen hyödyllinen suurissa ja monimutkaisissa ohjelmistoprojekteissa, joissa virheiden havaitseminen ja korjaaminen myöhäisessä vaiheessa voi olla kallista ja aikaa vievää. (Typescript 2024.)

Typescript tarjoaa merkittäviä etuja myös mikropalveluarkkitehtuurin toteuttamisessa. Sen avulla kehitetyt sovellukset ovat helpommin ylläpidettäviä ja skaalautuvampia kuin perinteinen Javascript, sillä staattinen tyypitys auttaa välttä-

mään ajonaikaisia virheitä ja mahdollistaa paremman ohjelmistokomponenttien välisen synergian. (Typescript 2024.) Lisäksi Typescriptin tarjoamat työkalut, kuten vahva IntelliSense-integraatio ja kattava ulkoisten kirjastojen skaala, auttavat kehittäjiä kirjoittamaan koodia nopeammin ja virheettömämmin.

Node.js:n yleistyttyä palvelinpuolen sovellusten rakentamisessa Typescriptistä on tullut yksi suosituimpia laajojen ohjelmistojen kehitykseen käytettyjä Javascript-pohjaisia kieliä. Node.js:n ja Typescriptin yhdistelmä tarjoaa tehokkaan kehitysympäristön, joka hyödyntää molempien teknologioiden parhaita puolia. Typescriptin integraatio olemassa oleviin JavaScript-koodikantoihin on lähes saumaton, mikä helpottaa siirtymistä JavaScript-projektista Typescriptiin tai uuden Typescript-pohjaisen projektin aloittamista. Lisäksi Typescriptin laaja kirjastojen ja kehysten ekosysteemi tarjoaa kattavan valikoiman valmiita ratkaisuja erilaisiin kehitystarpeisiin, mikä nopeuttaa kehitystyötä ja mahdollistaa nopeamman pääsyn markkinoille.

Selaimen ulkopuolella (esim. mikropalveluna) Typescript voi Javascriptiin pohjautuvana kielenä käyttää useita eri ajonaikaisia ympäristöjä. Joitakin esimerkkejä näistä ympäristöistä on Node.js, Deno, Bun ja tuoreimpana AWS:n llrt (Low Latency Runtime). (Konik 2023; llrt 2024.) Nämä ympäristöt vaikuttavat suuresti Typescriptillä rakennettujen ohjelmistojen tehokkuuteen (llrt 2024). Tällä hetkellä yksi suosituimmista ajonaikaisista ympäristöistä on Node.js.

Node.js tarjoaa varsin ainutlaatuisen lähestymistavan samanaikaisuuden käsittelyyn. Vaikka saattaa tuntua ristiriitaiselta, että yhteen säikeeseen perustuva Node.js voi hallita useita toimintoja samanaikaisesti, Node.js saavuttaa tämän asynkronisuuden ansiosta.

Node.js:ssä samanaikaisia toimintoja ei varsinaisesti suoriteta samanaikaisesti, mutta niitä ei myöskään suoriteta peräkkäin. Sen sijaan Node.js käyttää yhtä säiettä useiden samanaikaisten pyyntöjen hallintaan. Toiminnot laitetaan Node.js:n tapahtumakäsittelijässä (event loop) pinoon ja suoritetaan aiemman toiminnon suorittamisen jälkeen. Pinossa odottavan toiminnon suorittamista voi

Node.js:ssä jäädä odottamaan async-moduulin avulla. Odottava toiminto vastaanottaa lupauksen (promise), joka täytetään pinossa olevan toiminnon suorituksen jälkeen. (Node.js 2024.)

Tämä suunnitteluvaihtoehto tarjoaa etuina lähinnä tapahtumakäsittelijän yksinkertaisen mallin ja muuttujien eheyden (vain yksi säie käsittelee muuttujia), mutta tämä lähestymistapa on kuitenkin puutteellinen useampia säikeitä vaativissa toiminnoissa.

Mitä tulee luku- ja kirjoitustoimintoihin, Node.js tarjoaa kaksi vaihtoehtoa: estävä (blocking) ja ei-estävä (nonblocking). Estävä toiminto pysäyttää muiden toimintojen suorituksen tapahtumakäsittelijässä, kunnes nykyinen toiminto on valmis. Kun taas ei-estävät vastineet käsittelevät tuloksia asynkronisesti, jolloin ohjelma voi jatkaa myös muiden tapahtumien suorittamista.

Vaikka Typescript tarjoaa monia etuja, se ei ole vailla heikkouksia. Esimerkiksi Typescriptin oppimiskäyrä voi olla jyrkkä niille kehittäjille, jotka eivät ole tottuneet staattisesti kirjoitettujen kielten käyttöön. Lisäksi Typescriptin käyttö voi joissakin tapauksissa aiheuttaa tarpeetonta resurssien kuormitusta verrattuna kevyempiin kieliin, kuten Golang, vaikka tämä ero saattaakin olla hyvin pieni ja joskus jopa merkityksetön käytännön sovelluksissa. Lisäksi Typescriptin käyttöönotto vaatii jonkin verran lisäkonfigurointia olemassa oleviin Javascriptiin pohjautuviin kehitysympäristöihin ja tuotannollisiin järjestelmiin. Lisäksi Typescriptin käyttämä pakettienhallintaohjelmisto NPM tuo oman kitkansa käyttäjäkokemukseen.

Typescript tarjoaa monipuolisen ja tehokkaan työkalun modernin web- ja mikropalveluohjelmistojen kehittämiseen, sen suosio jatkaa kasvuaan kehittäjäyhteisössä, erityisesti suuryritysten isoissa IT-osastoissa.

2.4 Golang vs. Typescript

Kun verrataan Golangia ja Typescriptiä mikropalveluarkkitehtuurin viitekehyksessä, suorituskyvyn lisäksi on useita vaikuttavia tekijöitä. Asioita, joita kannattaa tarkastella ovat muun muassa suorituskyky, kehitystiimin osaaminen, koodikannan ylläpidettävyys, ekosysteemi ja käyttöönoton kitkattomuus. Molemmilla kielillä on vahvuutensa ja heikkoutensa, joten ne sopivat erilaisiin käyttötapauksiin mikropalveluarkkitehtuurin sisällä.

Yksinkertaisuudestaan ja tehokkuudestaan tunnettu Go tarjoaa vahvan tuen samanaikaisuudelle gorutiinien ja kanavien kautta. Tämä tekee siitä erittäin soveliaan korkeaa suorituskykyä ja alhaista latenssia vaativien mikropalvelujen rakentamiseen. Go:n staattinen tyypitys ja käännetty olemus takaavat paremman suorituskyvyn ja varhaisen virheiden havaitsemisen, mikä johtaa tehokkaampiin mikropalveluihin. Lisäksi sisäänrakennettu vakiokirjasto tarjoaa erinomaisen tuen rajapintojen rakentamiseen ja HTTP-pyyntöjen käsittelyyn, mikä yksinkertaistaa mikropalveluiden kehitystä.

Typescript taasen on tunnettu löyhästä tyypityksestä ja Javascriptiin pohjautuvasta ekosysteemistä. Laaja ekosysteemi on eduksi käyttökohteissa, joiden toteutuksessa halutaan tukeutua ekosysteemin tarjoamiin valmiisiin kirjastoihin. Node.js tarjoaa laajan valikoiman Typescriptiä tukevia kirjastoja ja kehyksiä, kuten ExpressJS ja Moleculer, jotka nopeuttavat mikropalvelujen kehitystä ja tarjoavat joustavuutta työkalujen valinnassa projektin vaatimusten mukaisesti.

Suorituskyvyn suhteen Golang voittaa Typescriptin puhtaassa suorituskyvyssä, kun taas Typescript loistaa skenaarioissa, joissa on jo olemassa olevaa, Typescriptiin tai Javascriptiin pohjautuvaa koodikantaa ja osaamista. Siksi valinta Go:n ja Typescriptin välillä riippuu kehitettävien mikropalvelujen erityisvaatimuksista suorituskyvykkyyden lisäksi. Typescriptille on myös verrattain helpompaa löytää kehittäjiä, koska monet kehittäjät tuntevat jo JavaScriptin, kun taas Golangin kehittäjien löytäminen voi olla haastavampaa, etenkin suurempiin projekteihin.

Molemmat kielet tarjoavat käyttöönoton työnkulkuihin erinäisiä vaihtoehtoja. Go tuottaa yhden binäärisen suoritettavan tiedoston, mikä yksinkertaistaa käyttöönottoa ja vähentää riippuvuuksia. Typescript, vaikka vaatiikin Node.js-tulkin, hyötyy lukuisista Javascript-pohjaisista alustoista ja käyttöönottotyökaluista, mikä helpottaa mikropalveluiden skaalaamista ja hallintaa tuotantoympäristöissä.

2.5 AWS Lambda

AWS Lambda, Amazon Web Servicesin palvelimeton laskentapalvelu, on mullistanut tavan, jolla ohjelmistokoodia voidaan julkaista mikropalveluksi (AWS Lambda 2024). Se tarjoaa ratkaisun koodin suorittamiseen vastimena erilaisiin tapahtumiin, kuten tietojen muutoksiin, järjestelmän tilan muutoksiin tai HTTP-pyyntöihin, ilman taustalla olevan infrastruktuurin hallintataakkaa.

AWS Lambda ilmentää palvelimettoman arkkitehtuurin ydintä, mikä antaa kehittäjille mahdollisuuden keskittyä pelkästään koodaukseen ilman infrastruktuurin hallinnan rasitteita. Tämä sopii saumattomasti mikropalveluiden paradigmaan, jossa jokainen komponentti suorittaa erillisen toiminnon. Ottamalla mikropalvelut käyttöön Lambda-funktioina kehittäjät voivat helposti skaalata ja hallita niitä itsenäisesti, mikä edistää sovellusarkkitehtuurin ketteryyttä ja modulaarisuutta.

Lambdan tapahtumavetoinen luonne resonoi mikropalvelujen periaatteiden kanssa. Mikropalvelut kommunikoivat tyypillisesti tapahtumien tai viestien kautta, ja Lambda-funktiot voidaan laukaista erilaisilla tapahtumilla, mukaan lukien muutokset eri AWS-palveluissa, kuten S3, DynamoDB tai Kinesis, sekä HTTP-pyynnöt Amazon API gatewayn kautta. Tämä tapahtumalähtöinen arkkitehtuuri helpottaa mikropalveluiden välistä löyhää kytkentää ja antaa kehittäjille mahdollisuuden reagoida muuttuviin tarpeisiin ketterästi.

Lisäksi Lambda tarjoaa luontaisen skaalautuvuuden, joka mukautuu automaattisesti käsittelemään vaihtelevia työkuormia. Jokainen toiminto skaalautuu itsenäisesti saapuvien pyyntöjen tai tapahtumien perusteella, jolloin ei tarvita manuaalisia toimia palvelimien hallintaan tai hallintaan. Tämä skaalautuva ominai-

suus yksinkertaistaa yksittäisten mikropalveluiden hallintaa ja varmistaa optimaalisen suorituskyvyn tinkimättä resurssien käytöstä, aiheuttamatta kuitenkaan tarpeettomia kustannuksia.

Kustannusten näkökulmasta AWS Lambdan käyttöperusteinen hinnoittelumalli osoittautuu erittäin edulliseksi sovelluksissa, joissa työtaakka vaihtelee. Lambdan avulla käyttäjät maksavat vain toimintojensa käyttämästä laskenta-ajasta, mikä eliminoi käyttämättömiin resursseihin liittyvät kulut. Tämä kustannustehokkuus yhdistettynä automaattiseen skaalautuvuuteen tekee AWS Lambdasta houkuttelevan vaihtoehdon organisaatioille, jotka haluavat optimoida kustannustehokkuuden säilyttäen samalla skaalautuvuuden ja saavutettavuuden.

Lisäksi AWS Lambda integroituu saumattomasti laajempaan AWS-ekosysteemiin, jolloin kehittäjät voivat hyödyntää AWS-palveluiden täyden tehon kestävien ja monipuolisten sovellusten rakentamisessa. Tämä integraatio virtaviivaistaa kehittämiseen ja käyttöönottamiseen liittyviä työnkulkuja, mikä helpottaa mikropalvelupohjaisten sovellusten luomista ja hallintaa AWS:ssä.

AWS Lambda tarjoaa joustavan, skaalautuvan ja kustannustehokkaan alustan mikropalvelupohjaisten sovellusten kehittämiseen pilvessä. Sen palvelimeton arkkitehtuuri, tapahtumalähtöinen malli, skaalautuvuus, kustannustehokkuus ja saumaton integrointi AWS-palveluihin tekevät siitä keskeisen työkalun mikropalvelujen pilvipalveluiden alalla, mikä antaa kehittäjille mahdollisuuden rakentaa joustavia, ketteriä ja innovatiivisia sovelluksia verrattoman tehokkaasti ja helposti. Muun muassa näistä syistä johtuen Amazon Web Services on markkinajohtaja pilvilaskennassa ja siksi sitä käytettiin myös tämän työn laskenta-alustana.

2.6 Kehitysympäristö

AWS Cloud9 -kehitysympäristöä käytettiin testitapausten toteutukseen (AWS Cloud9 2024). AWS Cloud9 tarjoaa monipuolisen kehitysympäristön, joka mahdollistaa eri kielten, kuten Golangin ja Typescriptin, kehittämisen ja

testaamisen samassa ympäristössä, ilman työkalujen asentamista ja konfigurointia.

AWS Cloud9 integroituu saumattomasti muihin AWS-palveluihin, kuten AWS Lambdaan, mikä tekee kehitys- ja testausprosessista sujuvaa. Tämä mahdollistaa Lambda-funktioiden hallinnan ja testaamisen suoraan kehitysympäristöstä.

Riippuen Cloud9:n käyttämän EC2-instanssin koosta saatetaan työnkuluissa kohdata suorituskykyongelmia erityisesti suurten projektien kanssa. Tästä esimerkkinä laajemman Typescript-projektin riippuvuuksien rakentaminen, kun käytetään useampaa isoa ulkoista kirjastoa.

3 Testitapaukset

Tässä luvussa käydään läpi vertailevan tutkimuksen tueksi valikoidut kolme erillistä testitapausta. Testitapauksista eritellään niiden tarkoitus, toteutukset ja tulokset kullakin kielellä.

Ensimmäisessä testitapauksessa tarkastellaan Golangin ja Typescriptin suoritusaikoja ja muistinkäyttöä yksinkertaisessa metodissa, joka ei sisällä ulkoisia kirjastoja. Toinen testitapaus keskittyi HTTP-kutsun käsittelyyn ja paluuarvojen muodostamiseen eri kielillä. Kolmannessa testitapauksessa tutkittiin suorituskykyä JSON Web Token -standardin mukaisen tunnisteen luomisessa.

Toteutuksia tarkastellessa on tärkeää huomata, että kummallekkin kielelle saatavilla olevat AWS Lambda -kirjastot toimivat tärkeinä komponenteina sekä Golangin että Typescriptin toteutuksia. Nämä kirjastot kapseloivat AWS Lambdan toiminnallisuuksia ja sisältävät työkaluja, jotka on räätälöity kielten tarpeisiin, mikä helpottaa ja virtaviivaistaa kehitykseen ja käyttöönottoon liittyviä työnkulkuja.

Golangille kehitetty aws-lambda-go tarjoaa kattavan tuen Lambda-funktioiden toteuttamiseksi ja tapahtumapohjaisten työnkulkujen käsittelyyn. Golang-ohjelmat suoritetaan AWS Lambdassa ajonaikaisessa ympäristössä nimeltä Provided.al2. Provided.al2 ei sisällä lainkaan Lambda-funktiota tukevaa ohjelmistokoodia, vaan se on tarkoitettu käännetyille kielille, kuten Golangille. Käännetyn binääritiedoston tulee siis sisältää muun muassa ajonaikainen rajapinta, joka on sisällytetty aws-lambda-go-kirjastoon.

Typescriptiä käytettäessä lähestymistapa on erilainen. AWS Lambdan tässä työssä käytetty ajonaikainen ympäristö Node.js18.x ei suoraan tue Typescriptiä, vaan se tulee kääntää Javascriptiksi ennen käyttöönottoa. Node.js18.x sisältää Javascriptin suorittamiseen vaaditut ohjelmistokomponentit, kuten Node.js-tulkin sekä ajonaikaisen rajapinnan, joka kutsuu käyttöönotossa määriteltyä Lambda-funktiota. Typescriptin kanssa työskentelyyn vaaditaan lisäksi AWS Lambda-

funktiolle oleelliset tyypitykset, jotka ovat saatavilla Noden pakettihallinnasta (NPM).

AWS tarjoaa vankan sovellusliittymien ja apuohjelmien joukon palvelimattomien sovellusten rakentamiseen niin Golangilla kuin Typescriptilläkin. Molemmille kielille saatavat kirjastot abstrahoivat matalan tason monimutkaisuudet, jolloin kehittäjät voivat keskittyä liiketoiminnallisen logiikan kirjoittamiseen varmistaen samalla saumattoman synergian erinäisten AWS-palveluiden kanssa.

Käyttöönotto

Lambda-funktioiden käyttöönotossa käytettiin AWS:n Serverless Application Model (SAM) -työkalua, joka on valmiiksi konfiguroitu Cloud9-kehitys-ympäristöön. Sen avulla käyttöönottotyönkuluista tulee virtaviivaisia ja toistettavia.

Esimerkkikoodi 2:sessä määritellään SAM:ille käyttöönottotyönkulkua varten ajonaikaisen ympäristön tietoja sekä se mistä, ja millä nimellä Lambda-funktio löytyy.

```
Properties:
      CodeUri: src/
      Handler: app.ping
      Runtime: Node.js18.x
      Architectures:
        - x86_64
```

Esimerkkikoodi 2: Leikelmä yaml-muodossa olevasta SAM-sapluunasta.

Esimerkkikoodi 2 havainnollistaa Golangille ominaisen kääntämistyönkulun osan, jossa määritellään muun muassa se, miten käännetyn binääritiedoston ei tule tukeutua ympäristön C-kielen kirjastoihin.

```
build-PingFunction:
     GOOS=linux CGO_ENABLED=0 go build -o bootstrap main.go
     cp ./bootstrap $(ARTIFACTS_DIR)/.
```

Esimerkkikoodi 3: Yksinkertaisen, Golangilla kirjoitetun, Lambda-funktion kääntämistä määrittelevä makefile.

Cloud9-palvelun avulla käyttöönottotyönkulkujen luonti oli varsin triviaalia.

Tuloksista

Toteutuksien yhteydessä esitellään lisäksi tulokset, jotka saatiin suorittamalla testitapauksia AWS Lambda -palvelussa. Testit suoritettiin hyödyntäen palvelun sisäänrakennettuja metriikka- ja monitorointityökaluja. Tuloksissa keskityttiin vertaamaan Golangin ja Typescriptin suorituskykyä suoritusaikojen ja muistinkäytön kannalta eri testitapauksissa.

3.1 Testitapaus 1

Ensimmäisessä testitapauksessa tavoitteena oli tutustua tarvittaviin työnkulkuihin ja saada osviittaa siitä, miten kielet suoriutuvat yksinkertaisimmasta mahdollisesta tehtävästä.

Tuloksena haluttiin, että funktio yksinkertaisesti palauttaa merkkijonon.

Golang

Golangilla käytettiin AWS:n aws-lambda-go-kirjastoa toteuttamaan rajapinta ajonaikaisen ympäristön ja ohjelmakoodin välille, itse ohjelma on hyvin ytimekäs.

```
func ping() (string, error) {
     return "pong", nil
}
```

Esimerkkikoodi 4: Go-metodi.

Metodi yksinkertaisesti palauttaa merkkijonon. API Gatewaytä käytettäessä tämän palvelun vastauksessa olisi HTTP-status 200 ja sisältönä merkkijono ”pong”.

Typescript

Typescriptillä ajonaikainen rajapinta on sisäänrakennettu ympäristöön, jolloin itse koodissa riittää, kun käytetään oikeaa tyypitystä, joka varmistaa metodin oikeellisuuden.

```
export const ping: Handler = async (event, context) => {
    return "pong";
};
```

Esimerkkikoodi 5: Typescript-metodi.

Metodin tyypityksenä käytetty ”Handler” on tuotu aws-lambda-kirjastosta.

Typescriptillä joudutaan toteuttamaan vietynä (export) async-metodina, sillä ajonaikainen ympäristö kutsuu Lambda-funktiota asynkronisesti.

Tulokset

Taulukon 1 tulokset osoittavat Golangin huomattavasti suuremman suorituskyvyn Typescriptiin verrattuna sekä suoritusajan että muistin käytön suhteen. Golang on ylivoimainen 1,37 ms:n suoritusajalla verrattuna Typescriptin 18,91 ms:iin.

Taulukko 1: Testitapaus 1 tulokset

	Keski-määräinen suoritusaika	**Käynnistysaika**	**Laskutettava laskenta-aika**	**Muistin-käyttö**
Golang	1.37 ms	60.38 ms	62 ms	16 MB
Typescript	18.91 ms	175.24 ms	194.15 ms	67 MB

Lisäksi Golang käyttää huomattavasti vähemmän muistia, vain 16 megatavua Typescriptin 67 megatavuun verrattuna, mikä korostaa Golangin optimoidumpaa ajonaikaista ympäristöä ja kykyä käyttää resursseja huomattavasti tehokkaammin.

3.2 Testitapaus 2

Toisessa testitapauksessa tarkoituksena oli tarkastella ulkoisen rajapinnan käyttämistä Lambda-funktiossa. Funktio kutsuu Typicoden jsonplaceholder-rajapintaa ja palauttaa vastauksen JSON-muodossa.

Ulkoista rajapintaa kutsuttaessa huolenaiheena oli ulkoisen palvelun latenssin vaikutus testin tuloksiin. Tämä huoli osoittautui käytännössä mitättömäksi vastausaikojen vaihdellessa vain vähän suhteessa kymmeniin kutsuihin pohjautuviin keskimääräisten suoritusaikojen kokonaisuuteen.

Golang

```
func call() (string, error) {
     res, _ := http.Get("https://jsonplaceholder.typicode.com/todos/1")
     defer res.Body.Close()
     body, _ := io.ReadAll(res.Body)
     data := string(body)
     return data, nil
}
```

Esimerkkikoodi 6: Go-metodi.

Esimerkkikoodi 6:sessa havainnollistetussa metodissa käytetään AWS:n aws-lambda-go-kirjaston lisäksi Go:n sisäänrakennettuja io- ja net/http-kirjastoja. Io-kirjaston avulla käsitellään tietovirtoja (HTTP-vastauksen sisältö) ja net/http-kirjastolla kutsuttiin ulkoista verkossa olevaa HTTP-rajapintaa. Metodissa näkyy myös Go:n implisiittinen tyypitys (:=), jossa muuttujan tyyppi määrittyy sille asetetun tietotyypin mukaisesti.

Typescript

```
export const call: Handler = async (event, context) => {
 const res = await fetch(url);
 return {
  statusCode: res.status,
  body: JSON.stringify({
   message: await res.text(),
  }),
 };
};
```

Esimerkkikoodi 7: Typescript-metodi.

Typescriptillä käytettiin Noden sisäänrakennettuja fetch- ja json-metodeja. Vastauksena on olio, jossa vastauksen statuskoodi ja sisältö määrittyy ulkoisen palvelun vastauksen mukaan. Huomionarvoista lienee se, miten Typescriptiä käytettäessä vastausta ei käsitellä tietovirtana, mikä yksinkertaistaa muuttujien käsittelyä. Handler-tyypitys on jälleen aws-lambda-kirjastosta.

Tulokset

Taulukon 2 perusteella voidaan tulkita ulkoisen rajapinnan kutsumisen hidastavan kumpaakin toteutusta huomattavasti, mutta Typescriptin suorituskyky on edelleen huomattavasti alhaisempi kuin Golangin.

Taulukko 2: Testitapaus 2:n tulokset

	Keski-määräinen suoritusaika	**Käynnistys-aika**	**Laskutettava laskenta-aika**	**Muistinkäyttö**
Golang	715.73 ms	60.47 ms	777 ms	23 MB
Typescript	2365.13 ms	174.83 ms	2366 ms	90 MB

Typescriptin muistin käyttö on jälleen noin kolminkertainen verrattuna Golangiin.

3.3 Testitapaus 3

Kolmannessa testitapauksessa luotiin metodi, joka luo JSON Web Token-standardin mukaisen tunnisteen ja lähettää sen vastauksena. Molemmissa toteutuksissa käytettiin HS256-algoritmia tunnisteen luomiseen. Metodi ei toteuta autentikointia.

Golang

Metodissa käytetään Node-ekosysteemissä laajasti käytetyn erinomaisen jsonwebtoken-kirjaston Golangille käännettyä toteutusta.

```
func token() (string, error) {
     return jwt.NewWithClaims(jwt.SigningMethodHS256,
          jwt.MapClaims{"foo": "bar"}).SignedString([]byte("secret"))
}
```

Esimerkkikoodi 8: Go-metodi.

Esimerkkikoodi 7:n token-metodi palauttaa JWT-standardin tunnisteen. Metodissa määritellään eksplisiittisesti, millä algoritmilla (HS256) tunniste kirjataan.

Typescript

Metodissa käytetään jsonwebtoken-kirjastoa (Jsonwebtoken 2024) JWT-tunnisteen luomiseen. Metodi on jälleen asynkroninen ajonaikaista ympäristöä varten.

```
export const token: Handler = async (event, context) => {
    return jwt.sign({ foo: 'bar' }, 'secret');
};
```

Esimerkkikoodi 9: Typescript-metodi.

Esimerkkikoodi 8:ssa näkyy jälleen tietotyyppien käsittelyn yksinkertaisuus verrattuna Golang-toteutukseen Esimerkkikoodissa 7. Toisaalta toteutuksessa on myös enemmän implisiittisesti määräytyviä komponentteja (esim. tunnisteen kirjaamisessa käytetty algoritmi). Handler-tyyppi on tuotu aws-lambda-kirjastosta.

Tulokset

Kolmannen testitapauksen tuloksissa (taulukko 3) ilmenee Golangin merkittävä ylivertaisuus tunnisteen laskennan tehokkuudessa.

Taulukko 3: Testitapaus 3 tulokset

	Keskimääräinen suoritusaika	**Käynnistysaika**	**Laskutettava laskenta-aika**	**Muistinkäyttö**
Golang	1.44 ms	58.15 ms	60 ms	20 MB
Typescript	41.91 ms	170.38 ms	212.29 ms	69 MB

Typescriptin suoritusaika on merkittävästi pidempi kuin Golangin. Jos toteutukseen liittyisi muita raskasta laskentaa vaativia toimintoja tai tietokantaoperaatioita, olisi suoritusaikojen ero tätäkin suurempi.

4 Analyysi ja suositukset

Ohjelmointikielten suorituskyvyn arvioinnin osalta empiirinen näyttö toimii perustana tietoiselle päätöksenteolle. Kun verrataan Golangilla kehitettyjen sovellusten suorituskykymittareita Typescriptin vastaaviin, tulosten tarkastelu paljastaa huomattavan eron resurssien käytössä. Testitapausten avulla on selvää, että Typescript osoittaa huomattavaa taipumusta resurssien tarpeettomalle kulutukselle, erityisesti muistin varaamisessa ja suorittimen käytössä, verrattuna Golangiin.

Muistin käytön tarkka analyysi paljastaa jyrkän kontrastin Golang- ja Typescript-sovellusten välillä. Testitapauksen tuloksissa on johdonmukaisia viitteitä siihen, että Typescript-sovellukset edellyttävät suurempaa muistiresurssien allokointia, jotka järjestään ylittävät Golangin tarpeen kolminkertaisesti. Tämä muistijalanjäljen kasvu ei ainoastaan aiheuta potentiaalisesti suurempia infrastruktuurivaatimuksia, mutta myös mahdollisia skaalautuvuushaasteita, erityisesti resurssirajoitteisissa projekteissa. Mikäli muistiresurssien kulutus on laskenta-alustan käytön laskutusperusteena, on syytä harkita muistia tehokkaammin käyttävän kielen valitsemista. Monessa käyttötapauksessa voidaan kuitenkin sallia muistin korkeampi käyttöaste, varsinkin jos Typescript on jo organisaation käytössä ja käyttötapaukset ovat laskennallisesti keveitä.

Typescriptin osoittamaa resurssien ahnasta käyttöä koskeva näyttö peilautuu myös suorittimen käyttöaikaan, joka on suorituskyvyn arvioinnissa tärkeä mittari. Typescriptin resurssien käytön tarkastelu testitapausten osalta paljastaa huomattavan eron myös suoritusajassa verrattuna vastaaviin Golang-toteutuksiin. Typescriptin pitkittynyt suoritusaika saattaa rasittaa laskentaresursseja huomattavissakin määrin, mikä saattaa vaikuttaa sovellusten reagointikykyyn ja järjestelmän yleiseen tehokkuuteen. Mahdollisen laskutusperusteen lisäksi suoritusaika saattaa myös vaikuttaa käyttäjäkokemukseen, esimerkiksi jos jokin tapahtu-

ma vaatii useamman mikropalvelun kutsumista tai sisältää raskaampia työkuormia ja siten käyttäjän näkökulmasta hidastaa sovelluksen toimintaa.

Typescript on luonnostaan dynaaminen komentokieli, vaikka se saattaa joissain tilanteissa parantaa kehittäjien tuottavuutta ja helpottaa koodin ylläpidettävyyttä, lisää se merkittävästi muistin ja suoritinajan kulutusta. Sitä vastoin Golangin tehokas ajonaikainen malli mahdollistaa resurssien tehokkaamman käytön.

Suorituskykyvertailun perusteella Golang olisi parempi vaihtoehto mikropalveluarkkitehtuurin kehittämiseen, mutta ohjelmointikieltä valittaessa on otettava huomioon muitakin seikkoja. Olemassa olevat resurssit, kuten aiempi koodikanta ja kehittäjien Typescriptiin tai Javascriptiin liittyvä osaaminen, saattavat olla riittävä peruste pysyä jatkossakin Typescriptissä, mutta jos kehitystä ollaan aloittamassa puhtaalta pöydältä ja käyttötapaus vaatii suorituskykyä, on suositeltavissa harkita Golangin käyttämistä.

Nämä havainnot ja päätelmät korostavat kuhunkin käyttötapaukseen ja organisaation strategisiin tavoitteisiin perustuvien tietoisten päätösten merkitystä ohjelmointikieleen liittyvissä valinnoissa kehitystyötä aloitettaessa sekä mikropalveluarkkitehtuurin jatkokehityksessä.

5 Yhteenveto

Opinnäytetyössä pyrittiin selvittämään, miten Golang ja Typescript soveltuvat mikropalveluarkkitehtuurin kehittämiseen.

Työ alkoi selvityksellä Golangin ja Typescriptin eroista ja yhtäläisyyksistä. Seuraavaksi toteutettiin käytännön suorituskykyvertailu pilvilaskenta-alustalla. Vertailusta laadittiin analyysi ja lopuksi läpikäytiin havaintoihin perustuvat suositukset. Analyysin ja suositusten oli tarkoitus nopeuttaa tehokkaan työnkulun perustamista ja suorituskyvyn optimoinnin aloittamista.

Opinnäytetyö saavutti sille asetetut tavoitteet niiltä osin kuin valituista kielistä suoritettiin taustaselvitys, pilvilaskenta-alustalla toteutettiin suorituskykyvertailu sekä selvityksen ja vertailun pohjalta laadittiin suositukset. Selvityksessä ei kuitenkaan pureuduttu ohjelmointikielten teknisiin eroihin kovinkaan syvällisesti, tämä näkyi myös niin suorituskykyvertailussa, kuten myös analyysissä ja suosituksissa. Lopputuloksen voidaan silti katsoa nopeuttavan tehokkaan työnkulun perustamista ja suorituskyvyn optimoinnin aloittamista ainakin jossain määrin.

Työn aikana opin AWS Lambdasta ja siihen liittyvistä työnkuluista. Vaikka taustaselvitys ja suorituskykyvertailu vahvisti aiempaa käsitystä Golangin ja Typescriptin eroista, kontrasti näiden kielten raa’an suorituskyvyn välillä yllätti silti.

Lähteet

Ankit, Malik. 2023. What is Goroutine. Verkkoaineisto. <https://dev.to/ankitmalikg/what-is-goroutine-1j6d/> Luettu 3.5.2024.

AWS Lambda. 2024. Verkkoaineisto. <https://aws.amazon.com/lambda/> Luettu 3.5.2024.

AWS Cloud9. 2024. Verkkoaineisto. <https://aws.amazon.com/cloud9/> Luettu 3.5.2024.

Richter, Felix. 2024. Amazon Maintains Cloud Lead as Microsoft Edges Closer. Verkkoaineisto. <https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/> Luettu 3.5.2024.

Go (ohjelmointikieli). 2024. Verkkoaineisto. <https://fi.wikipedia.org/wiki/Go_(ohjelmointikieli)> Luettu 3.5.2024.

Golang. 2024. Verkkoaineisto. <https://go.dev/> Luettu 3.5.2024

Golang-JWT. 2024. Verkkoaineisto. <https://github.com/golang-jwt/jwt/> Luettu 3.5.2024.

Goroutines. 2024. Verkkoaineisto. <https://go.dev/tour/concurrency/1/> Luettu 3.5.2024.

Jefferson, Lima. 2023. A little about Goroutines in Go! Verkkoaineisto. <https://dev.to/jeffotoni/a-little-about-goroutines-in-go-2f0f/> Luettu 3.5.2024.

Jsonwebtoken. 2024. Verkkoaineisto. <https://github.com/auth0/node-jsonwebtoken/> Luettu 3.5.2024.

Konik, James. 2023. Node.js vs. Deno vs. Bun: JavaScript runtime comparison. Verkkoaineisto. <https://snyk.io/blog/javascript-runtime-compare-node-deno-bun/> Luettu 3.5.2024.

Microservices (A). 2024. Verkkoaineisto. <https://en.wikipedia.org/wiki/Microservices/> Luettu 3.5.2024.

Microservices (B). 2024. Verkkoaineisto. <https://aws.amazon.com/microservices/> Luettu 3.5.2024.

What are microservices? 2024. Verkkoaineisto. <https://www.ibm.com/topics/microservices/> Luettu 16.5.2024.

Node.js. 2024. Verkkoaineisto. <https://fi.wikipedia.org/wiki/Node.js/> Luettu 3.5.2024.

Ostrowski, Rafał. 2023. A simple guide to JavaScript concurrency in Node.js and a few traps that come with it. <https://tsh.io/blog/simple-guide-concurrency-node-js/> Luettu 3.5.2024.

Richardson, Chris. 2020. Service per team. Verkkoaineisto. <https://microservices.io/patterns/decomposition/service-per-team.html/> Luettu 3.5.2024.

Typescript. 2024. Verkkoaineisto. <https://www.Typescriptlang.org/> Luettu 3.5.2024.

What are the benefits of a microservices architecture? 2022. Verkkoaineisto. <https://about.gitlab.com/blog/2022/09/29/what-are-the-benefits-of-a-microservices-architecture/> Luettu 3.5.2024.