Tampereen yliopisto

Kari Harmaahieta

Mikropalveluiden haasteet

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Marraskuu 2022

Tiivistelmä

Kari Harmaahieta: Mikropalveluiden haasteet
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Marraskuu 2022

Ohjelmistojen määrän kasvaessa ohjelmistoarkkitehtuurin merkitys kasvaa. Perinteisesti ohjelmistot ovat olleet monoliittisia, jolloin kaikki ohjelman suoritettava koodi on yhdessä suoritettavassa ohjelmassa. Ohjelmistojen koon ja määrän kasvaessa monoliittisesta ohjelmistoarkkitehtuurista on siirrytty kohti jaettua ohjelmistoarkkitehtuuria, jossa ohjelman suoritettava koodi on jaettu useampaan eri osaan. Mikropalveluarkkitehtuuri on yksi jaetun arkkitehtuurin tyyleistä joka on saavuttanut suosiota viime vuosina. Se perustuu pieniin, itsenäisiin palveluihin.

Tässä tutkielmassa tutustutaan ohjelmistoarkkitehtuuriin ja erityisesti mikropalveluarkkitehtuuriin. Tutkielmassa tehtiin systemaattinen kirjallisuuskatsaus mikropalveluarkkitehtuurin haasteisiin. Kirjallisuuskatsauksella etsittiin tietoa mikropalveluiden kehittäjien kokemista haasteista sekä ratkaisuista niihin haasteisiin. Tuloksia vertailtiin aiempiin aiheesta tehtyihin tutkimuksiin. Kirjallisuuskatsauksen aineistoon valittiin yhteensä 23 lähdettä, jotka olivat vuosilta 2017-2022. Aineistojen tutkimusmetodeina oli pääasiassa haastattelut tai kyselyt. Havaittiin, että haasteet ovat edelleen pääasiassa teknisiä ja ne olivat osittain samoja kuin aiemmissa tutkimuksissa. Haasteiden keskinäisessä tärkeysjärjestyksessä oli muutoksia: esimerkiksi mikropalveluiden suunnitteluun liittyviä haasteita ei koettu aiemmassa tutkimuksessa yhtä tärkeiksi kuin nyt. Teknisien haasteiden lisäksi esille nousi kuitenkin selkeästi erilaiset organisaatioon ja ihmisiin liittyvät haasteet, joita aiempi tutkimus ei ollut löytänyt yhtä hyvin.

Avainsanat: mikropalvelu, mikropalveluarkkitehtuuri, ohjelmistoarkkitehtuuri, ohjelmistokehitys.

Sisällys

1 Johdanto

Ohjelmistojen merkitys on kasvanut viime vuosina huomattavan suureksi. Ohjelmistoihin törmää nykyisin lähes päivittäin jossain muodossa. Älypuhelimet ja sosiaalinen media ovat tietenkin näkyvä esimerkki tästä, mutta ohjelmistot ovat taustalla lähes kaikessa muussakin toiminnassa. Pankki käsittelee rahojasi ohjelmistojen avulla, kaupan kassalla on ohjelmisto joka käsittelee myyntitapahtumat ja jopa fyysiset postilähetyksetkin voidaan nykyisin lajitella koneellisesti, ohjelmistojen avulla. Ohjelmistoarkkitehtuuri kuvaa ohjelmistopohjaisen järjestelmän ominaisuuksia ja sitä käytetään ohjelmistoja suunnitellessa (Richards ja Ford, 2020).

Mikropalveluarkkitehtuuri on ohjelmistoarkkitehtuurin tyyli, joka perustuu pieniin, itsenäisiin palveluihin. Monet johtavat organisaatiot ovat ottaneet sen käyttöönsä sen tarjoamien etujen vuoksi. Mikropalveluarkkitehtuuriin perustuvilla sovelluksilla on hyvä skaalautuvuus, saatavuus ja mahdollistaa kehitystyön rinnakkaisuuden nopeuttaen sovelluksen kehitystä huomattavasti. (Nadareishvili ja muut, 2016)

Tässä tutkielmassa tutustutaan kirjallisuuskatsauksen avulla mikropalveluarkkitehtuuriin. Kirjallisuudesta haetaan tietoa mikropalveluarkkitehtuurista, sen suunnittelusta, käytetyistä suunnittelumalleista ja teknologioista sekä mikropalveluarkkitehtuurin eduista. Lisäksi tutkielmassa tehdään tarkempi systemaattinen kirjallisuuskatsaus, jonka avulla tutustutaan mikropalveluarkkitehtuurin haasteisiin.

Systemaattisen kirjallisuuskatsauksen avulla haettiin vastausta kahteen tutkimuskysymykseen:

- Minkälaisia haasteita mikropalveluiden kehittäjät kokevat?
- Minkälaisia ratkaisuja näihin haasteisiin on?

Systemaattista kirjallisuuskatsausta varten etsittiin lähteitä kuudesta eri hakukoneesta. Katsaukseen valittiin lähdemateriaaliksi vertaisarvioituja julkaisuja, joissa on tutkittu mikropalveluiden käyttöön liittyviä haasteita. Katsauksen lähdemateriaaliin valittiin pääasiassa tutkimuksia, joiden tutkimusmenetelmänä oli haastattelu tai kysely. Systemaattisen kirjallisuuskatsauksen tuloksena löydetyt haasteet kategorisoitiin ja haasteita vertailtiin aiemman tutkimuksen kanssa. Lisäksi haasteisiin löydetyt ratkaisut esiteltiin haasteiden yhteydessä.

Toisessa luvussa käsitellään ohjelmistoarkkitehtuuria. Aluksi määritetään arkkitehtuurin ominaisuudet ja sen jälkeen esitellään muutama sekä monoliittinen että jaettu arkkitehtuurinen tyyli. Kolmannessa luvussa paneudutaan tarkemmin mikropalveluarkkitehtuuriin, sen suunnitteluun ja etuihin. Neljännessä luvussa tehdään systemaattinen kirjallisuuskatsaus mikropalveluiden haasteisiin. Viidennessä luvussa pohditaan tutkimuksen tuloksia. Kuudennessa luvussa tehdään yhteenveto ja sen jälkeen tulee vielä lähdeluettelo.

2 Ohjelmistoarkkitehtuuri

Standardi ISO/IEC/IEEE 42010 (IEEE, 2008) määrittää käsitteistön järjestelmien arkkitehtuurien kuvaamiseen. *Järjestelmät* (system) ovat tässä standardissa "järjestelmiä, jotka ovat ihmisen tekemiä ja joihin voidaan konfiguroida yksi tai useampi seuraavista: laitteisto, ohjelmisto, data, ihmiset, prosessit (esim. prosessit palvelun tarjoamiseen käyttäjille), proseduurit (esim. operaattorin ohjeet), laitokset, materiaalit ja luonnollisesti esiintyvät entiteetit". Tässä tutkielmassa keskityn *ohjelmisto-intensiivisiin järjestelmiin* (software-intensive system), jotka määritellään seuraavasti: "mikä tahansa järjestelmä, jossa ohjelmisto vaikuttaa olennaisesti suunnitteluun, rakennukseen, käyttöönottoon ja järjestelmän evoluutioon kokonaisuutena" ja ne sisältävät "yksittäiset sovellukset, järjestelmät perinteisessä tarkoituksessa, alijärjestelmät, järjestelmien järjestelmät, tuotelinjat, tuoteperheet, kokonaiset hankkeet ja muut mielenkiinnon yhdistelmät". *Arkkitehtuurin* (architecture) standardi määrittää seuraavasti: "järjestelmän keskeiset konseptit tai ominaisuudet ympäristössään ilmentyneinä sen elementeissä, suhteissa ja suunnittelun sekä kehityksen periaatteissa". (IEEE, 2008)

Ohjelmistoja suunnitellessa voidaan hyödyntää *arkkitehtuurisia tyylejä* (architecture styles). Erään määritelmän (Richards ja Ford, 2020) mukaan arkkitehtuurinen tyyli määrittää käyttöliittymän ja taustajärjestelmän lähdekoodin rakenteen organisaation sekä lähdekoodin vuorovaikutuksen tietovaraston kanssa. Kirjassaan *Fundamentals of software architecture: an engineering approach* Richards ja Ford (2020) jakavat arkkitehtuuriset tyylit kahteen luokkaan: *monoliittisiin* (monolithic), joissa koko lähdekoodilla on vain yksi *käyttöönottoyksikkö* (deployment unit) ja *jaettuihin*, joissa lähdekoodilla on monta käyttöönottoyksikköä yhdistettyinä *etäkäyttöprotokollilla* (remote access protocol).

Kohdassa 2.1 esitellään ohjelmistoarkkitehtuurien tärkeimmät ominaisuudet ja käsitteet, jotta voimme tarkastella erilaisten ohjelmistoarkkitehtuurien eroja. Tämän jälkeen käydään tarkemmin läpi monoliittista ja jaettua arkkitehtuuria.

2.1. Arkkitehtuurin ominaisuudet

Moduuli (module) voidaan määrittää "standardin mukaiseksi osaksi tai itsenäiseksi yksiköksi, jota voidaan käyttää kokoamaan monimutkaisempi rakenne". *Modulaarisuutta* (modularity) ohjelmistoarkkitehtuurissa käytetään kuvaamaan toisiinsa liittyvän koodin ryhmiä. Useimmat ohjelmointikielet tukevat modulaarisuutta jollakin tapaa, kuten lisäämällä paketeilla ja kirjastoilla ominaisuuksia. Se tukee siten koodin uudelleenkäyttöä. Modulaarisuuteen liittyy oleellisesti *liitokset* (coupling). Liitoksilla tarkoitetaan jonkin komponentin riippuvuuksia toisiin komponentteihin. Sitä voidaan mitata laskemalla komponentin sisään- ja ulostulevien kutsujen määrää toisiin komponentteihin. (Richards ja Ford, 2020) Usein puhutaan kuitenkin moduulien sijaan komponenteista, joka voidaan määrittää moduulin fyysiseksi ilmentymäksi. Komponentti voi olla jokin koodikirjasto tai esimerkiksi palvelu. Sovelluksen arkkitehtuuri luodaan käyttämällä komponentteja. Kom-

ponenttien koko ja vastuualue vaikuttaa sovelluksen liitoksien määrään. Jos liitoksia on paljon, komponenttien välisen kommunikaation määrä voi vaikuttaa sovelluksen suorityskykyyn, käyttöönoton helppouteen ja testattavuuteen. (Richards ja Ford, 2020)

Skaalautuvuus (scalability) tarkoittaa järjestelmän kykyä suoriutua käyttäjien tai pyyntöjen määrän lisääntyessä (Richards ja Ford, 2020). Pilvipalveluissa skaalautuvuus tarkoittaa esimerkiksi tallennuskapasiteetin, laskentatehon ja verkkojen lisäämistä sovelluksen käyttöön ilman käyttökatkoja. Sovelluksen ylläpitäjä voi helposti käydä lisäämässä sovellukselle lisäresursseja. *Elastisuus* (elasticity) on lähellä skaalautuvuutta oleva käsite. Pilvipalveluissa se tyypillisesti tarkoittaa sitä, että sovellus voi automaattisesti mukautua kuormitusasteen mukaan ja pilvipalvelu voi lisätä tai poistaa resursseja sovelluksen käytöstä. (VMware, 2022)

Vikasietoisuus (fault tolerance) tarkoittaa sovelluksen kykyä jatkaa toimintaansa normaalisti laitteisto- tai ohjelmistovirheen satuttua. (Richards ja Ford, 2020) Vikasietoisuus vaikuttaa olennaisesti järjestelmän saatavuuteen, jota käsitellään alakohdassa **??**.

Sovelluksen *suorituskyky* (performance) koostuu monista osista ja sitä voidaan mitata usealla tavalla. *Läpisyöttö* (throughput) määrittää, minkä määrän työtä sovellus suorittaa tietyssä ajassa. Usein tätä mitataan *tapahtumina sekunnissa* (transactions per second). *Vasteaika* (response time) määrittää sen, kuinka nopeasti sovellus vastaa käyttäjän pyyntöihin. *Aikarajat* (deadline) määrittävät kuinka kauan jokin toimenpide saa kestää. (Gorton, 2011)

Saatavuus (availability) määrittää, kuinka suuren osan ajasta järjestelmän tulee olla käytettävissä. Useimmat Internet-sivut toivovat 100% saatavuutta. Järjestelmän saatavuus voi kärsiä jonkin vian tai virheen takia. Yksi tapa varmistaa järjestelmän saatavuutta on järjestelmän komponenttien replikointi, eli jokaista järjestelmän komponenttia on käynnissä kerralla useampi kuin yksi. Kun jokin komponentti vikaantuu, sovellus voi yhä käyttää komponentin replikoita. (Gorton, 2011)

Turvallisuus (security) määrittää, kuinka turvallinen järjestelmä on (Gorton, 2011). Usein sovelluksilta vaaditaan ainakin seuraavia turvallisuuten liittyviä ominaisuuksia (Gorton, 2011):

- *autentikointi* (authentication) eli käyttäjän tunnistaminen
- *auktorisointi* (authorization) eli käyttäjän oikeudet sovelluksen resursseihin
- *salaus* (encryption) eli esimerkiksi viestien tai tietokannan salaus
- *eheys* (integrity) eli tiedon muuttumattomuuden varmistaminen
- *kiistämättömyys* (nonrepudiation) eli viestinvaihtoon osallistumista ei voi kumpikaan osapuoli jälkikäteen kiistää. Viestin lähettäjällä on todiste viestin toimituksesta ja vastaanottaja voi olla varma viestin lähettäjästä.

2.2. Monoliittinen arkkitehtuuri

Enterprise-sovellus tarkoittaa sovellusta, joka on tehty pääasiassa yritysten käyttöön helpottamaan ja automatisoimaan yritysten tehtäviä, kuten myyntiä, varaston hallintaa tai henkilöstöhallintoa (Henderson, 2021). Enterprise-sovellus koostuu tyypillisesti kolmesta osasta: web-selaimella toimiva käyttöliittymä, palvelimella suoritettava taustajärjestelmä sekä tietokanta (Fowler ja Lewis, 2014). Taustajärjestelmä on usein yksi suoritettava ohjelma, joka tekee kaiken: käsittelee HTTP-pyynnöt, suorittaa ohjelman logiikan, päivittää ja hakee tietoa tietokannasta ja lähettää tiedon selaimelle (Fowler ja Lewis, 2014). Tällaista arkkitehtuuria kutsutaan monoliittiseksi, koska kaikki ohjelman logiikka tapahtuu yhdessä suoritettavassa ohjelmassa (Fowler ja Lewis, 2014).

Seuraavaksi esitellään kolme erilaista monoliittista arkkitehtuurityyliä. Sen jälkeen pohditaan monoliittisen arkkitehtuurin haasteita.

2.2.1. Kerroksittainen arkkitehtuuri

Kerroksittainen arkkitehtuuri (layered architecture) on yksi käytetyimmistä arkkitehtuurisista tyyleistä. Se tunnetaan myös nimellä *n-tason arkkitehtuuri* (n-tier architecture). Sitä hyödyntävä sovellus koostuu horisontaalisista kerroksista. Kerroksien lukumäärää ei ole rajattu, mutta usein kerroksia on neljä: *esitys* (presentation), *bisneslogiikka* (business), *pysyvyys* (persistence) ja *tietokanta* (database). Jokaisella kerroksella on oma tehtävänsä. Esitys-kerros huolehtii tiedon näyttämisestä käyttäjälle. Bisneslogiikka-kerros käsittelee käyttäjän pyynnöt sovelluksen sääntöjen mukaan. Pysyvyys-kerros huolehtii tiedon hakemisesta ja tallentamisesta tietokantaan, ja tietokanta-kerros pitää tiedon tallessa. Vaikka kerroksittaista arkkitehtuuria hyödyntävällä sovelluksella on monia käyttöönottotapoja, osa jopa hajautettuja, sitä kutsutaan silti monoliittikseksi, koska jokaisella kerroksella on vain yksi käyttöönottoyksikkö. (Richards ja Ford, 2020)

Kerroksittaista arkkitehtuuria hyödyntävä sovellus voidaan ottaa käyttöön muutamalla eri tavalla. Kaikki kerrokset voivat sijaita samassa suoritettavassa ohjelmassa tai osa niistä voi olla erillään. Esimerkiksi voi olla, että kaikki kerrokset ovat muuten samassa suoritettavassa ohjelmassa, mutta sillä on ulkoinen tietokanta. Toisaalta sovellus voi olla myös web-sovellus, jolloin esitys-kerros on erillinen selaimella käytettävä sivu, bisneslogiikka- ja pysyvyyskerrokset sivun taustalla oleva *API* (Application Programming Interface, ohjelmointirajapinta) ja tietokanta-kerros vielä tästä rajapinnasta erillinen. (Richards ja Ford, 2020)

2.2.2. Putkistoarkkitehtuuri

Putkistoarkkitehtuuri (pipeline architecture) on arkkitehtuurinen tyyli, joka tunnetaan erityisesti Unix-tyylisten käyttöjärjestelmien komentotulkeista. Se tunnetaan myös *putki-suodatin-arkkitehtuurina* (pipes and filters architecture), mikä kuvaakin tyyliä erityisen hyvin. Putkistoarkkitehtuuria hyödyntävä sovellus koostuu suodattimista ja niiden välisistä putkista. Ne muodostavat yksisuuntaisen putkiston. Putkistoarkkitehtuuria toteute-

taan usein monoliittisena, eli sillä on vain yksi käyttönottoyksikkö. (Richards ja Ford, 2020)

Putket ovat suodattimien välisiä kommunikaatiokanavia. Ne ovat tyypillisesti yksisuuntaisia ja niillä on vain yksi sisääntulo ja yksi ulostulo. Suodattimet ovat itsenäisiä ja tilattomia komponentteja, joita putkilla yhdistellään. Suodattimia on neljää eri tyyppiä. *Tuottajat* (producer) ovat aina putken alkupäässä. Nimensä mukaisesti ne tuottavat jotain tietoa putkistolle, esimerkiksi vaikkapa lukevat tiedostosta merkkijonoja. *Muuntajat* (transformer) ovat putkiston keskellä. Ne saavat sisäänsä tietoa, muuttavat sitä omien sääntöjensä mukaan ja ohjaavat muutetun tiedon ulostuloonsa. *Testaajat* (tester) ovat myöskin putkiston keskellä. Ne saavat sisäänsä tietoa, testaavat päteekö tietoon jotkin tietyt kriteerit ja joko ohjaavat tiedon muuttumattomana ulostuloonsa tai jotain muuta, kuten vaikkapa testin tulokset. *Kuluttajat* (consumer) ovat putkiston loppupäässä. Niiden tehtävä on kuluttaa putkistossa kulkenut data esimerkiksi tallentamalla se tietokantaan tai tulostamalla se käyttöliittymälle. (Richards ja Ford, 2020)

2.2.3. Mikrokernelarkkitehtuuri

Mikrokernelarkkitehtuuri (microkernel architecture), joka tunnetaan myös nimellä *liitännäisarkkitehtuuri* (plug-in architecture), koostuu kahdesta komponentista: *ydinjärjestelmästä* (core system) ja *liitännäisistä* (plug-in). Ydinjärjestelmä sisältää sovelluksen perustoiminnallisuuden sekä järjestelmän liitännäisten käyttämiseen. Liitännäisillä voidaan sovellukseen lisätä uusia ominaisuuksia ja toiminnallisuutta. Koska kaikki liitännäiset liitetään yhteen ydinjärjestelmään, mikrokernelarkkitehtuuri on monoliittinen. (Richards ja Ford, 2020)

Mikrokernelarkkitehtuuri soveltuu hyvin esimerkiksi erilaisten ohjelmistokehitystyökalujen kehittämiseen, koska liitännäisillä voidaan helposti lisätä työkaluihin tuki esimerkiksi uusille ohjelmointikielille tai ohjelmistokehyksille. Suurin osa moderneista ohjelmistokehitystyökaluista hyödyntääkin mikrokernelarkkitehtuuria. (Richards ja Ford, 2020)

2.2.4. Monoliittisen arkkitehtuurin edut ja haasteet

Monoliittinen arkkitehtuuri on yksinkertaista. Se sopii hyvin pieniin järjestelmiin, jolloin lähdekoodi pysyy hyvin hallussa. Mitä isommaksi monoliittinen järjestelmä kasvaa, sitä vaikeampi lähdekoodia ja kehitystä on hallita. Yksinkertaisuudesta johtuen monoliittinen arkkitehtuuri on myös usein halvempaa toteuttaa, järjestelmästä riippuen. Kerroksittainen arkkitehtuuri sopii erittäin hyvin pieneen, yksinkertaiseen järjestelmään, johon ei odoteta paljon käyttäjiä eikä juurikaan uusia ominaisuuksia. Kerroksittaista sovellusta on vaikeampi ruveta laajentamaan kehityksen jälkeen. Pienenkin ominaisuuden lisääminen voi vaatia muutoksia useaan paikkaan, ja yhdenkin koodirivin muutos taas vaatii aina koko järjestelmän uudelleen testaamisen ja käyttöönoton. Jos tarvitaan hieman modulaarisempi järjestelmä, mikrokernelarkkitehtuuri soveltuu tähän hyvin. Liitännäisiä voi testata erik-

seen ja lisätä käyttöön helposti. (Richards ja Ford, 2020)

Monoliittisella arkkitehtuurilla on kaksi suurta haastetta, jotka vaikeuttavat niiden käyttämistä todella monen käyttäjän sovelluksissa: skaalautuvuus ja vikasieto (Richards ja Ford, 2020). Ensiksi, monoliittiset järjestelmät voivat skaalautua vain koko ohjelma kerrallaan. Jos vaikkapa monoliittisen web-sovelluksen jossain osiossa on paljon ruuhkaa mutta muualla ei käyttäjiä juuri ollenkaan, ainut vaihtoehto skaalautua siltikin on käynnistää toinen kokonainen taustajärjestelmä rinnalle ja jakaa kuorma näiden välillä *kuormantasaajalla* (load balancer). Toiseksi, jos monoliittisen sovelluksen jossain osassa ilmenee virhe, kaatuu silloin koko sovellus ja se on pois käytöstä kokonaan.

2.3. Jaettu arkkitehtuuri

Jaettu arkkitehtuuri tarkoittaa arkkitehtuurista tyyliä, jossa sovelluksella on monta käyttöönottoyksikköä (Richards ja Ford, 2020). Seuraavaksi esitellään kolme erilaista jaettua arkkitehtuuria ja käydään läpi jaettuun arkkitehtuuriin liittyviä haasteita.

2.3.1. Tilapohjainen arkkitehtuuri

Tilapohjainen arkkitehtuuri (space-based architecture) on arkkitehtuurinen tyyli, joka on suunniteltu ajatellen skaalautuvuutta ja suorituskykyä. Sovellus skaalautuu lisäämällä tai vähentämällä *prosessointiyksiköitä* (processing unit). Sovelluksen data on replikoituna prosessointiyksiköiden muistiin. Data on tallessa myös tietokannassa, jota prosessointiyksiköt päivittävät asynkronisesti esimerkiksi viestijonojen avulla. Prosessointiyksiköt jättävät päivitysviestin viestijonoon ja erilliden *tiedon kirjoittaja* (data writer) kirjoittaa tiedon tietokantaan. Tietokannasta data haetaan vain, kun käynnistetään uusi prosessointiyksikkö. Koska sovelluksen sisältämä data on suoraan muistissa ja tietokantaa käytetään vain asynkronisesti, se ei enää ole pullonkaula vaan sovellus voi skaalautua lähes loputtomasti. (Richards ja Ford, 2020)

2.3.2. Tapahtumapohjainen arkkitehtuuri

Tapahtumapohjainen arkkitehtuuri (event-driven architecture) on arkkitehtuurinen tyyli, jonka keskiössä ovat tapahtumat. Useimmat sovellukset perustuvat sovellukselle tehtäviin pyyntöihin, esimerkiksi pyyntö päivittää tietokantaan jokin arvo. Tapahtuma sen sijaan on jokin sovelluksessa tapahtunut asia, johon sovelluksen tulee reagoida. Se voi olla esimerkiksi huutokaupassa tapahtunut huuto, jonka jälkeen sovelluksen pitää päättää onko se suurin huuto ja kuinka tähän tulee reagoida. (Richards ja Ford, 2020)

Tapahtumapohjaisen arkkitehtuurin voi toteuttaa kahdella tavalla: *välittäjällä* (broker) tai *sovittelijalla* (mediator). Välittäjää käytettäessä tapahtumat kulkevat ketjussa palvelulta toiselle. Kun ketjun aloittava tapahtuma tapahtuu, se lähetetään *tapahtumakanavalle* (event channel) välittäjässä, joka voi olla esimerkiksi yksinkertainen viestinvälittäjä kuten RabbitMQ. Sieltä viestin lukee yksi *tapahtumankäsittelijä* (event processor), joka käsittelee tapahtuman ja lähettää tapahtumakanavalle *käsittelytapahtuman* (processing event),

jonka toiset tapahtumankäsittelijät voivat lukea. Näin tapahtuu kunnes viimeinen tapahtumankäsittelijä on käsitellyt tapahtumat. Sovittelijaa käytettäessä ketjun aloittava tapahtuma tulee tapahtumajonoon, josta sovittelija sen lukee. Sovittelija tietää kaikki tapahtuman vaatimat työvaiheet, ja luo jokaisen vaiheen tapahtumajonoille aloittavan tapahtuman käsittelyn vaatimat tapahtumat. Järjestelmän tapahtumankäsittelijät käsittelevät tapahtumat ja lähettävät sovittelijalle vastauksen, jonka perusteella se voi luoda seuraavat tarvittavat tapahtumat. Järjestelmässä voi olla useampikin sovittelija monimutkaisempia toimia vaativia tapahtumia varten. (Richards ja Ford, 2020)

2.3.3. Palvelupohjainen arkkitehtuuri

Palvelupohjainen arkkitehtuuri (service-based architecture) jakaa sovelluksen erillisiin palveluihin. Tyypillisesti siinä on erillinen käyttöliittymä, taustajärjestelmä joka on jaettu isompiin palveluihin ja monoliittinen tietokanta. Yksittäiset palvelut ovat tyypillisesti melko isoja sovelluksen osia, jotka otetaan käyttöön monoliittisesti. Käytössä voi myös olla kuormantasaaja, jolloin yksittäisiä palveluita voi käynnistää myös useamman ja sovellus voi näin skaalautua. Palveluita käytetään käyttöliittymältä tyypillisesti esimerkiksi REST-rajapintojen avulla, joista kerrotaan alakohdassa 3.4.1 lisää. (Richards ja Ford, 2020)

2.3.4. Jaetun arkkitehtuurin haasteet

Vaikka edellä esitellyt jaetun arkkitehtuurin tyylit ovat keskenään melko erilaisia, niillä on myös paljon yhteisiä piirteitä. Niillä on siten myös yhteisiä haasteita. *The 8 fallacies of distributed computing* on lista, jossa Peter Deutsch ja James Gosling esittelevät kahdeksan virhepäätelmää joita arkkitehdit ja suunnittelijat usein tekevät suunnitellessaan jaettuja järjestelmiä (Rotem-Gal-Oz, 2022). Nämä virhepäätelmät esittelevät jaetun arkkitehtuurin haasteita, jotka pätevät yhä tänä päivänä (Richards ja Ford, 2020). Taulukossa 1 on esiteltynä edellä mainitut kahdeksan virhepäätelmää selityksineen (Rotem-Gal-Oz, 2022).

Taulukko 1. Jaetun laskennan 8 virhepäätelmää.

Verkko on luotettava	Verkkolaitteet voivat hajota, sähkökatko voi sammuttaa verkkolaitteita, Internet-yhteydessä voi olla häiriöitä. Sovelluksen on varauduttava siihen, että verkko ei ole luotettava ja siihen voi tulla käyttökatkoja.
Latenssi on nolla	*Latenssi* tarkoittaa sitä, kuinka kauan aikaa kestää siirtää data paikasta toiseen. Lähiverkoissa latenssi voi olla melko pieni, mutta mitä pidemmäksi välimatka kasvaa, latenssi kasvaa myös. Kun dataa käsitellään verkon kautta, latenssi on myös aina suurempi kuin suoraan muistista luettaessa. Tämä vaikuttaa jaettujen sovellusten suorituskykyyn.
Kaista on loputon	*Kaista* (bandwidth) tarkoittaa sitä, kuinka paljon dataa voidaan siirtää tietyssä ajassa paikasta toiseen. Kaistan määrä kasvaa jatkuvasti, mutta niin kasvaa myös siirrettävän datan määrä. Jaetuissa sovelluksissa täytyy huomioida, että kaistaa ei ole loputtomasti, varsinkaan jos dataa siirretään Internetin yli.
Verkko on turvallinen	Verkot eivät ole turvallisia. Internetin kautta tehtävät hyökkäykset ovat lisääntymään päin. Arkkitehdin täytyy huomioida jaettua sovellusta suunnitellessa turvallisuus ja varautua monenlaisiin uhkiin.
Topologia ei vaihdu	Verkon topologia voi muuttua jatkuvasti. Verkkoihin liitetään uusia koneita, vaihdetaan reitittimiä, palveluiden osoitteet voivat muuttua. Jaetun sovelluksen on varauduttava siihen, että verkko ei tule pysymään samanlaisena.
On vain yksi ylläpitäjä	Isommissa organisaatioissa ja enterprise-järjestelmissä verkko sisältää monia palveluita ja käyttäjiä. Kun tähän lisätään vielä ulkoiset palveluntarjoajat, eri järjestelmillä voi olla todella paljon ylläpitäjiä. Kun jaetun sovelluksen käytössä tulee ongelmia, voi olla vaikeaa selvittää kenen ylläpitäjän vastuulla jokin on.
Siirto on ilmaista	Tämän virhepäätelmän voi ymmärtää kahdella tapaa. Ensiksi, tiedon siirtäminen sovellustasolta siirtotasolle on ilmaista. Se ei pidä kuitenkaan paikkaansa, koska tiedolle on tehtävä erilaisia muutoksia jotta se voidaan siirtää verkon ylitse. Toiseksi, verkon rakennus ja ylläpito on ilmaista. Tämäkään ei pidä paikkaansa, koska verkkolaitteet maksavat, ne kuluttavat sähköä ja lisäksi voi olla tarpeellista maksaa myös Internet-yhteydestä. Vaikka sovelluksen kehitystiimi ei näistä kuluista tiedä, joku ne joutuu maksamaan.
Verkko on homogeeninen	Jaetun sovelluksen on varauduttava siihen, että verkot eivät yleensä ole homogeenisiä. Se tarkoittaa sitä, että verkoissa sijaitsee erilaisia tietokoneita, mutta myös verkkolaitteet kuten kytkimet ja reitittimet voivat olla erilaisia. Lisäksi verkon palvelut voivat käyttää erilaisia protokollia.

3 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri (microservices architecture) on arkkitehtuurinen tyyli, jossa sovellus koostuu monista pienistä palveluista joita suoritetaan omissa prosesseissaan (Fowler ja Lewis, 2014). Sen taustalla vaikuttaa palvelukeskeinen arkkitehtuuri (Fowler ja Lewis, 2014), mutta myös *toimialalähtöinen suunnittelu* (domain-driven design) (Richards ja Ford, 2020). Mikropalveluita on ollut olemassa jo jonkin aikaa, mutta ensimmäisiä yrityksiä määrittää se arkkitehtuurisena terminä oli vuonna 2014, kun Fowler ja Lewis julkaisivat artikkelinsa *Microservices*. (Richards ja Ford, 2020)

3.1. Kuvaus

Mikropalveluarkkitehtuurin keskiössä ovat palvelut. Sen sijaan, että lisättäisiin ohjelmistoon uusia komponentteja ja ominaisuuksia kirjastojen avulla, niitä lisätään palveluilla. Perinteisesti ohjelmistokehityksessä ohjelmisto on jaettu osiin teknologioiden perusteella: käyttöliittymä, palvelin, tietokanta. Mikropalveluarkkitehtuurissa iso ohjelmisto jaetaan osiin *toiminnallisuuden* (business capability) perusteella. Tällöin yksi mikropalvelu vastaa jonkin tietyn toiminnallisuuden kokonaisuudesta, käyttöliittymästä tietokantaan asti. (Fowler ja Lewis, 2014)

Mikropalveluarkkitehtuurin ominaispiirteisiin kuuluu myös yksinkertainen kommunikaatio palveluiden välillä. Sen sijaan että käytettäisiin monimutkaisia protokollia kuten WS-Choreographya tai BPELiä, käytetään palveluiden väliseen kommunikaatioon usein HTTP-kutsuja tai kevyttä viestinvälitystä. (Fowler ja Lewis, 2014)

Mikropalveluarkkitehtuuri on luonteeltaan hajautettu, joten usein sitä myös hallitaan hajautetusti. Monesti mikropalvelun tehnyt tiimi on vastuussa siitä kokonaan: se voi päättää itsenäisesti, millä teknologioilla palvelu toteutetaan, mutta se voi myös olla vastuussa palvelun suorittamisesta ja ylläpidosta. Mikropalveluarkkitehtuurissa myös tietoa voidaan hajauttaa. Jokaisella palvelulla voi esimerkiksi olla oma tietokantansa. (Fowler ja Lewis, 2014)

Mikropalveluarkkitehtuurin apuna hyödynnetään usein automatisoitua infrastruktuuria. Koska palveluita voi olla paljon, käyttöönoton automatisointi voi helpottaa paljonkin testausta, testiautomaatiota sekä ohjelmiston lopullista julkaisua ja käyttöönottoa. Automatisoitu infrastruktuuri auttaa myös ohjelmiston häiriötilanteissa ja se voi käynnistää kaatuneita palveluita takaisin ylös. (Fowler ja Lewis, 2014)

Kirjassaan *Microservice architecture*, Nadareishvili ja muut (2016) määrittävät mikropalveluiden tärkeimmiksi piirteiksi seuraavat:

- pienikokoinen
- käyttää viestintää
- kontekstin rajaama
- autonomisesti kehitetty

- itsenäisesti käyttöönotettava
- hajautettu
- rakennettu ja julkaistu automaattisilla prosesseilla.

Kohdassa 3.2 kerrotaan enemmän mikropalveluiden suunnittelusta sekä siihen käytetyistä suunnittelumalleista. Kohdassa 3.4 käydään läpi erilaisia teknologioita ja käytäntöjä, joita hyödynnetään mikropalveluiden kehityksessä ja jotka mahdollistavat mikropalveluille edellä kuvatut ominaisuudet ja piirteet.

3.2. Suunnittelu

Mikropalveluarkkitehtuuria hyödyntävän sovelluksen suunnittelussa yksi tärkeimmistä seikoista on mikropalvelut itse. Mikropalveluita kehitettäessä suurimmat haasteet liittyvät mikropalvelun oikeaan kokoon sekä datan tallentamiseen: mihin mikropalvelun rajat vedetään, että se kykenee toimimaan itsenäisesti mutta on silti tarpeeksi pieni hyötyäkseen mikropalveluarkkitehtuurista? Nadareishvilin ja muiden mukaan monet käyttävätkin toimialalähtöisen suunnittelun periaatteita suunnittelemaan mikropalveluita. (Nadareishvili ja muut, 2016)

Kirjassaan *Domain-Driven Design: Tackling Complexity in the Heart of Software* Evans (2003) määrittelee kaksi tärkeää konseptia, joita voidaan hyödyntää Nadareishvili ja muut (2016) mukaan mikropalvelujen suunnittelussa: *rajatun kontekstin* (bounded context) ja *kaikkialla läsnäolevan kielen* (ubiquitous language). Kun isoa järjestelmää aletaan mallintaa, tarvitaan siihen monia eri malleja. Mallintamisen apuna käytetään rajattua kontekstia. Jokainen järjestelmän malli toimii itsenäisesti ja edustaa yhtä autonomista toimialaa. Kaikkialla läsnäoleva kieli tarkoittaa periaatetta, jonka mukaan palvelusta puhuttaessa käytetään yhtenäistä kieltä kaikkien osapuolien kesken. Kolmas suunnitteluperiaate, jonka Nadareishvili ja muut (2016) määrittävät mikropalveluille, on *pienempi on parempi* (smaller is better). Heidän mukaansa koon vähentäminen on selkeä trendi esimerkiksi *ketterässä kehityksessä* (Agile Development) ja *jatkuvassa toimituksessa* (Continuous Delivery). Ongelman koon tai laajuuden pienentäminen vähentää aikaa ratkaista se, saada nopeammin palautetta ja pienentää myöskin käyttöönottoyksikön kokoa. (Nadareishvili ja muut, 2016)

Vuoden 2021b tutkimuksessaan *Design, monitoring, and testing of microservices systems: The practitioners' perspective* Waseem ja muut selvittivät kyselyn ja haastatteluiden avulla mikropalveluiden suunnittelua, monitorointia ja testaamista käytännössä. Kyselyn perusteella kaikki vastaajat hyödynsivät sovelluksen *purkamisessa* (decomposition) mikropalveluiksi joko toimialapohjaista suunnittelua tai toiminnallisuuksia. 70 prosenttia vastaajista käytti suunnittelussa apuna joko toimialapohjaista suunnittelua tai molempia. Eräs kysymys kyselyssä liittyi mikropalveluiden *laatutekijöihin* (quality attribute). Kyselyn vastausten perusteella viisi tärkeintä (vastauksista laskettu yhteen vaihtoehtojen erit-

täin tärkeä ja tärkeä osuudet) mikropalveluiden laatutekijää järjestyksessä tärkeimmästä vähiten tärkeään olivat:

1. skaalautuvuus
2. suorituskyky
3. saatavuus
4. turvallisuus
5. luotettavuus.

Kyselyssä selvitettiin myös useiten mikropalveluiden suunnittelussa käytettyjä *suunnittelumalleja* (design pattern). Viisi useiten (vastauksista laskettu yhteen vaihtoehtojen erittäin usein ja usein osuudet) käytettyä mikropalveluiden suunnittelumallia järjestyksessä useiten käytetystä vähiten käytettyyn olivat (Waseem ja muut, 2021b):

1. *API-yhdyskäytävä* (API gateway)
2. *Taustajärjestelmä käyttöliittymälle* (Backend for frontend)
3. *Käyttöoikeusvaltuus* (Access token)
4. *Tietokanta per palvelu* (Database per service)
5. *Koottu API* (API composition).

Eräässä varhaisemmassa tutkimuksessa, Soldani ja muut (2018) selvittivät kirjallisuuskatsauksen avulla "kipuja ja hyötyjä". Tutkimuksessa löydettiin seuraavat viisi mikropalveluissa käytettyä suunnittelumallia (Soldani ja muut, 2018):

1. *Tietokanta per palvelu* (Database per service)
2. *API-yhdyskäytävä* (API gateway)
3. *Kierron rikkoja* (Circuit breaker)
4. *Palveluiden löytäminen* (Service discovery)
5. *Viestinvälitys* (Message broker).

3.3. Suunnittelumallit

Seuraavaksi tutustutaan tarkemmin kohdassa 3.2 mainittuihin suunnittelumalleihin.

3.3.1. API-yhdyskäytävä

API-yhdyskäytävä on suunnittelumalli, jossa yksi API toimii useamman APIn yhdistävänä *tulokohtana* (entry point). Tämän ansiosta API-yhdyskäytävä voi palvella monia eri tarpein varustettuja asiakkaita samasta tulokohdasta. API-yhdyskäytävä voi myös tarjota erilaisia muitakin palveluita sitä kautta käytetyille palveluille, kuten palveluiden löytämisen (josta lisää alakohdassa 3.3.7), kuorman tasauksen, monitoroinnin ja turvallisuuden. (Montesi ja Weber, 2016)

Asad (2022) tutustui artikkelissaan *14 Open Source and Managed API Gateway for Modern Applications* API-yhdyskäytävään suunnittelumallina ja esitteli 14 erilaista toteutusta API-yhdyskäytävästä. Hän mainitsee API-yhdyskäytävien ominaisuuksina lisätyn turvallisuuden, käyttäjän tunnistautumisen, vikasiedon, kuorman tasauksen ja *reitityksen* (routing), *eristyksen* (isolation), *käänteisen välityksen* (reverse proxy), *välimuistin* (cache), *tiedon muuttamisen* (data transformation) sekä *protokolla-adapterin* (protocol adaptor). (Asad, 2022)

3.3.2. Taustajärjestelmä käyttöliittymälle

Sovelluksen eri asiakkailla, kuten selain- ja mobiilikäyttöliittymillä, voi olla erilaisia tarpeita taustajärjestelmiin. Taustajärjestelmä käyttöliittymälle on API-yhdyskäytävän variaatio, jossa nämä tarpeet huomioidaan. Siinä jokaiselle asiakasohjelmalle luodaan oma API-yhdyskäytävä, joka on räätälöity juuri tätä asiakasohjelmaa varten. Eri API-yhdyskäytävät voivat sisältää esimerkiksi eri palveluita tai muuta konfiguraatiota, kuten erilaisia tunnistautumistapoja, asiakkaasta riippuen. Tämä mahdollistaa sen, että yksittäisen asiakkaan API-yhdyskäytävästä tulee pienempi, koska sen ei tarvitse palvella kaikkia asiakkaita. Usein myös tietystä käyttöliittymästä vastaava tiimi vastaa myös tämän käyttöliittymän omasta API-yhdyskäytävästä. Toisaalta, koska jokaiselle asiakkaalle on oma API-yhdyskäytävänsä josta eri tiimit vastaavat, tämä voi myös lisätä samanlaisen koodin määrää, koska API-yhdyskäytävät todennäköisesti tekevät myös monia asioita samalla tavalla. (Newman, 2015)

3.3.3. Käyttöoikeusvaltuus

Käyttöoikeusvaltuus on suunnittelumalli, jota käytetään API-yhdyskäytävien kanssa. API-yhdyskäytävä autentikoi käyttäjän pyynnöt ja lähettää käyttäjän pyynnön mukana käyttöoikeusvaltuuden kun pyyntö ohjataan API-yhdyskäytävän takana oleviin palveluihin. Näin palvelut tunnistavat käyttäjän ja voivat myös käyttää samaa käyttöoikeusvaltuutta tehdessään pyyntöjä toisiin palveluihin. (Richardson, 2022a)

3.3.4. Tietokanta per palvelu

Jotta mikropalveluiden ominaisuuksista saa täyden hyödyn irti, palveluiden tulee olla mahdollisimman itsenäisiä. Tietokanta per palvelu on suunnittelumalli, jossa jokaiselle sovelluksen palvelulle tehdään oma tietokanta. Tällöin eri palvelut eivät ole niin riippuvaisia toisistaan ja palvelun tietokantaan voi tehdä muutoksia puuttumatta muihin tietokantoihin. Palvelut voivat myös käyttää toisistaan eroavia tietokantatyyppejä: yhdelle palvelulle paras tietokantatyyppi voi olla NoSQL, toiselle SQL. Palvelut voivat saada käyttöönsä vain oman tietokantansa tietoa suoraan tietokannasta. Haittapuolena tässä suunnittelumallissa on se, että useita eri tietokantoja yhdistelevien kyselyiden tekeminen vaikeutuu, ja useamman tietokantatyypin käyttö lisää sovelluksen monimutkaisuutta ja vaikeuttaa ylläpitoa. (Richardson, 2022c)

3.3.5. Koottu API

Kun käytetään alakohdassa 3.3.4 esiteltyä suunnittelumallia Tietokanta per palvelu, sovelluksen tieto on jaettu moneen tietokantaan. Kuitenkin sovellusta käytettäessä usein tulee tarve saada samaan aikaan tietoa monesta eri tietokannasta. Koottu API on suunnittelumalli, joka ratkaisee tämän ongelman. Käyttäjän tekemä pyyntö tehdään *API kokoajalle* (API composer), joka tekee tarvittavat kyselyt tiedon sisältäviin palveluihin, kokoaa vastaukset muistiin ja palauttaa tuloksen käyttäjälle. Haasteena tässä suunnittelumallissa on se, että jotkut kyselyt johtavat isojen tietojoukkojen tehottomaan yhdistelyyn muistin sisällä. (Richardson, 2022b)

3.3.6. Kierron rikkoja

Kierron rikkojan tarkoituksena on estää virheen lähteminen kiertoon. Ennemmin ja myöhemmin, mikropalveluilla toteutetussa sovelluksessa jokin mikropalvelu ylikuormittuu tai muuten lakkaa vastaamasta pyyntöihin. Tällöin on vaarana, että siitä riippuvat muut palvelut jäävät odottamaan vastausta pyyntöihin, ja pahimmassa tapauksessa myös ne voivat lakata vastaamasta pyyntöihin. (Montesi ja Weber, 2016)

Perusajatuksena kierron rikkojassa on, että jokaiselle palvelulle on määritelty kierron rikkoja, jonka kautta kaikki pyynnöt palveluun menevät ja joka tarkkailee pyyntöjen onnistumista. Kierron rikkojalla on kolme tilaa: suljettu, avoin ja puoliavoin. Kun palvelu vastaa pyyntöihin normaalisti, kierron rikkoja pysyy suljettuna. Kun epäonnistumisia tulee liian usein tai vastaukset hidastuvat, kierron rikkoja laukeaa avoimeksi ja kaikki tulevat pyynnöt palveluun epäonnistuvat välittömästi. Tämän jälkeen kierron rikkoja siirtyy puoliavoimeen tilaan, esimerkiksi vaikka tarkkailemalla palvelun tilaa ping-pyynnöillä tai tietyn ajan päästä automaattisesti. Puoliavoimessa tilassa kierron rikkoja päästää läpi rajatun määrän pyyntöjä. Jos pyynnöt onnistuvat, kierron rikkoja siirtyy takaisin suljettuun tilaan, ja jos ne epäonnistuvat, se siirtyy takaisin avoimeen tilaan. (Montesi ja Weber, 2016)

Kierron rikkoja voidaan toteuttaa *asiakas-puolen kierron rikkojana* (client-side circuit breaker), *palvelu-puolen kierron rikkojana* tai *välityspalvelimen kierron rikkojana*. Asiakas-puolen kierron rikkojassa kierron rikkoja on toteutettu asiakasohjelman lähdekoodissa. Tämän etuna on se, että palveluiden saamat pyynnöt vähenevät kun kierron rikkoja on auki. Haasteena kuitenkin on, että ilkeämielinen käyttäjä saattaa voida halutessaan kiertää kierron rikkojan koska se sijaitsee asiakasohjelman puolella. Palvelu-puolen kierron rikkojassa jokainen palvelu sisältää oman kierron rikkojansa. Tässä tavassa etuna on se, että käyttäjä on pakotettu käyttämään sitä. Tämä kierron rikkoja saa myös tiedon pyyntöjen onnistumisesta kaikilta käyttäjiltä, ei vain yhdeltä. Haittapuolena tässä kuitenkin on, että vaikka kierron rikkoja olisi auki, palvelu joutuu silti vastaamaan pyyntöihin ja se käyttää resursseja kierron rikkojaan itsessään. Välityspalvelimen kierron rikkoja toimii siten, että asiakasohjelman pyynnöt palveluille menevät välityspalvelimen kautta, jossa on oma kierron rikkojansa jokaiselle asiakkaalle ja palvelulle. Tämä tapa voi olla yksinkertaisin toteuttaa, koska tässä ei tarvitse välttämättä muokata asiakasohjelman tai palvelimen lähdekoodia ollenkaan. Tämä voi olla paras tapa toteuttaa kierron rikkoja, varsinkin jos käytössä on lisäksi API-yhdyskäytävä. (Montesi ja Weber, 2016)

3.3.7. Palveluiden löytäminen

Palveluiden löytäminen on suunnittelumalli, joka mahdollistaa mikropalveluiden käytön dynaamisissa pilviympäristöissä. Käytännössä, tietyn mikropalvelun sijainti ei välttämättä ole tiedossa sovellusta suunniteltaessa: pilvipalvelut saattavat lisätä, vähentää tai siirtää mikropalveluita ajon aikana. Palveluiden löytymiseen voidaankin käyttää *palvelurekisteriä* (service registry), johon mikropalvelut itse ilmoittavat sijainnistaan ja josta palveluiden käyttäjät voivat niitä kysyä. Palveluiden löytyminen voidaan toteuttaa kahdella eri tavalla: *asiakas-puolen palvelun löytyminen* (client-side service discovery) tai *palvelin-puolen palvelun löytyminen* (server-side service discovery). Asiakas-puolen palvelun löytymisessä, asiakasohjelma kysyy palveluiden sijaintia itse palvelurekisteristä jonka jälkeen se käyttää palveluita suoraan niiden sijainnista. Palvelin-puolen palvelun löytymisessä taas asiakasohjelma lähettää suoraan haluamansa pyynnön välissä olevalle reitittimelle (esimerkiksi API-yhdyskäytävälle), joka kysyy palvelurekisteristä halutun palvelun sijaintia ja sen jälkeen välittää asiakasohjelman pyynnön oikeaan sijaintiin. (Montesi ja Weber, 2016)

3.3.8. Viestinvälitys

Mikropalveluarkkitehtuurissa palveluiden täytyy kommunikoida keskenään. Yksi tapa tähän on viestinvälitys, tai *asynkroninen viestintä* (asynchronous messaging). Asynkronisessa viestinnässä palvelut lähettävät toisilleen viestejä jonkinlaisen viestikanavan, viestinvälittäjän kautta. Viestinvälittäjän kautta palvelu voi lähettää viestin yhdelle tai useammalle vastaanottajalle ja mahdollisesti odottaa vastausta. Tämä suunnittelumalli mahdollistaa löysän liitännän palveluiden välille, koska se erottaa viestin lähettäjän sen vastaa-

nottajasta. Se myös mahdollistaa viestien puskuroinnin vastaanottajan ollessa kiireinen. Viestinvälittäjän lisääminen sovellukseen kuitenkin monimutkaistaa sen arkkitehtuuria, ja viestinvälittäjillä toteutettuna perinteinen pyyntö/vastaus-tyylinen viestintä on vaikeampaa. (Richardson, 2022d)

3.4. Teknologiat ja käytännöt

3.4.1. REST

REST (representational state transfer) on API-rajapintojen suunnittelussa käytetty arkkitehtuurinen tyyli, jonka esitteli Fielding (2000) väitöskirjassaan *Architectural Styles and the Design of Network-based Software Architectures*. Taulukossa 2 on listattu REST-arkkitehtuurin *arkkitehtuuriset rajoitteet* (architectural constraints) Fieldingin väitöskirjasta. (Fielding, 2000)

Taulukko 2. REST-arkkitehtuurin rajoitteet.

Asiakas-palvelin (client-server)	REST-arkkitehtuurin taustalla on yksinkertainen asiakas-palvelin arkkitehtuuri, jossa asiakas pyytää tietoa palvelimelta ja palvelin vastaa asiakkaalle.
Tilaton (stateless)	REST-arkkitehtuuria noudattavan palvelimen tulee olla tilaton. Tämä tarkoittaa sitä, että palvelimelle tehtävän pyynnön tulee sisältää kaikki pyynnön käsittelemiseksi tarvittava tieto. Asiakkaan *istunnon* (session) tilan on siis oltava tallessa kokonaisuudessaan asiakkaalla.
Välimuisti (cache)	Palvelimen tulee ilmoittaa jokaisen vastauksen kohdalla, voiko pyynnön tallentaa välimuistiin uudelleenkäyttöä varten vai ei.
Yhtenäinen rajapinta (uniform interface)	Palvelimen kaikilla komponenteilla tulee olla yhtenäinen rajapinta.
Kerroksittainen järjestelmä (layered system)	Kerroksittaisuudella voidaan rajoittaa alijärjestelmien näkyvyyttä asiakkaalle, piilottaa vanhoja palveluita tai esimerkiksi lisätä skaalautuvuutta lisäämällä järjestelmään kuormantasaajan.
Koodia tilauksesta (code-on-demand)	Koodia tilauksesta on vapaaehtoinen rajoite REST-arkkitehtuuriin. Se tarkoittaa, että asiakasohjelmaan voidaan lisätä toiminnallisuutta lataamalla ja suorittamalla uutta lähdekoodia.

REST-arkkitehtuurin rajoituksista erityisesti tilattomuus, välimuisti, yhtenäinen rajapinta ja kerroksittainen järjestelmä sopii hyvin mikropalveluarkkitehtuurin toteuttamiseen. Tilattomuus varmistaa, että käyttäjän pyyntöjen mukana kulkee aina kaikki tarvittava tieto. Siten eri palveluiden välillä ei tarvitse pitää käyttäjäistuntojen tilaa tallessa. Tämä

myös helpottaa järjestelmän skaalautuvuutta, koska yksittäisen palvelun käyttöönottoyksiköitä on helpompi lisätä. Välimuistin käyttö vähentää järjestelmän kuormitusta. Yhtenäinen rajapinta helpottaa kehitystyötä, kun järjestelmän palveluiden määrä kasvaa. Kerroksittaisuuden avulla helpotetaan myös järjestelmän skaalausta esimerkiksi kuormantasaajan avulla ja lisäksi voidaan ottaa käyttöön esimerkiksi API-yhdyskäytävä.

3.4.2. JWT

JWT (JSON Web Token, JSON-verkkovaltuus) on tapa välittää tietoa turvallisesti osapuolten välillä JSON-objektina. Se koostuu kolmesta osasta: otsikko, kuorma ja allekirjoitus. Otsikko sisältää valtuuden tyypin ja allekirjoitusalgoritmin. Kuorma sisältää käyttäjän *vaatimukset* (claim). Ne voivat sisältää tietoa esimerkiksi käyttäjän nimestä tai valtuuksista käyttää joitain resursseja. Allekirjoitus sisältää otsikon ja kuorman allekirjoitettuna otsikossa mainitulla algoritmilla ja jollakin salaisuudella. Tällä voidaan varmistaa, ettei JSON-verkkovaltuus ole muuttunut matkalla. JSON-verkkovaltuus koodataan lopuksi Base64Url koodauksella. Pienen koon ja turvallisuuden vuoksi JSON-verkkovaltuus sopii hyvin web-sovellusten käyttöön. JSON-verkkovaltuutta voidaankin käyttää auktorisoimaan käyttäjän pyynnöt verkkopalveluissa, tunnistautumisen jälkeen. (auth0, 2022)

JWT on hyödyllinen teknologia esimerkiksi REST-arkkitehtuurin toteutuksessa. JSON-verkkovaltuutta voidaan käyttää, jos halutaan hyödyntää Käyttöoikeusvaltuus-suunnittelumallia.

3.4.3. Kontitus

Kontitus (containerization) on teknologia, jolla sovellukset pakataan kevyiksi suoritettaviksi ohjelmiksi, joita kutsutaan *konteiksi* (container). Kontituksen ajatuksena on, että kontteihin pakataan vain suoritettava sovellus ja sen välttämättömät riippuvuudet jotta sovellus voidaan käynnistää. Konttien mukana ei tule kokonaista käyttöjärjestelmää, vaan kontit käynnistetään isäntäpalvelimen käyttöjärjestelmään asennetulla *ajonaikaisella moottorilla* (runtime engine). Yhdellä palvelimella voidaan suorittaa useampaa konttia samanaikaisesti. Ne hyödyntävät isäntäpalvelimen käyttöjärjestelmää, mutta ovat täysin eristettyjä toisistaan. Kontit voivat myös hyödyntää keskenään yhteisiä kerroksia konteista, kuten yhteisiä kirjastoja. Taulukko 3 listaa muutamia kontitusteknologian hyötyjä kehittäjille. (IBM, 2021)

Taulukko 3. Kontitusteknologian hyötyjä.

Siirrettävyys (portability)	Kontti on suoritettava ohjelma, joka ei ole riippuvainen isäntäkäyttöjärjestelmästä, joten sitä voi suorittaa kaikkialla.
Nopeus (speed)	Kontteja kutsutaan kevyiksi, koska ne jakavat isäntäpalvelimen käyttöjärjestelmän ytimen eikä siitä tule lisäkuormaa. Tämä nostaa palvelinten tehokkuutta, mutta myös nopeuttaa sovellusten käynnistysaikoja, koska niiden ei tarvitse käynnistää käyttöjärjestelmää.
Vian eristys (fault isolation)	Kontit ovat täysin eristetty toisistaan, joten yhdessä kontissa tapahtuva virhe ei vaikuta muiden konttien toimintaan.
Tehokkuus (efficiency)	Kontit jakavat isäntäpalvelimen käyttöjärjestelmän ytimen keskenään, mutta voivat jakaa myös keskenään samoja sovelluskerroksia, kuten yhteisiä kirjastoja. Tämän vuoksi kontit ovat pienempiä kuin virtuaalipalvelimet ja käynnistyvät nopeammin, joten yhdelle isäntäpalvelimelle voidaan käynnistää useampi kontti kuin virtuaalipalvelin.
Hallinnan helppous (ease of management)	*Konttien orkestrointi* (container orchestration) tarkoittaa konttien asennuksen, skaalauksen ja hallinnan automatisointia. Konttien orkestrointijärjestelmillä voidaan helpottaa esimerkiksi sovellusten skaalautumista, uusien versioiden julkaisua ja monitorointia.
Turvallisuus (security)	Kun sovellus on eristetty konttiin, estää se haitallisen koodin vaikutukset muihin saman isäntäpalvelimen kontteihin tai itse isäntäpalvelimeen.

Kontit ja erilaiset konttien orkestrointijärjestelmät auttavat mikropalveluiden käyttöönotossa, testauksessa, skaalautuvuudessa ja vikasietoisuudessa. Konttien avulla on helppo hallita mikropalveluiden suoritusympäristöä. Jokaiselle palvelulle voi tehdä oman kontin niiden tarpeiden mukaisesti räätälöitynä. Kun palveluita on monia, konttien orkestrointi helpottaa niiden käynnistystä käyttöönoton tai testauksen yhteydessä.

3.4.4. DevOps

DevOps on tapa organisoida sovelluskehitystyötä. Se yhdistää *kehityksen* (development, Dev) ja *operoinnin* (operations, Ops). Sen sijaan, että kehitystyö ja sovelluksen operointi olisivat eri tiimeillä, DevOpsissa tämä vastuu kuuluu samalle tiimille. DevOpsin tavoitteena on nopeuttaa sovellusten kehitystä ja ongelmien ratkaisua. DevOpsiin kuuluu olennaisena osana automaatio. Kehittäjät kehittävät uusia ominaisuuksia lähdekoodiin. *Jat-*

kuvan integroinnin (continuous integration) työkalut integroivat ja testaavat lähdekoodia usein. Lopulta sovellus voidaan ottaa käyttöön automaattisesti hyödyntäen tähän tehtyjä työkaluja. (Ebert ja muut, 2016)

DevOps sopii todella hyvin mikropalveluiden kehitykseen. Se tarjoaa toimintamallin, jossa kehitys ja operointi ovat lähellä toisiaan. Tämä sopii hyvin mikropalveluiden ajatukseen siitä, että palvelut ovat mahdollisimman itsenäisiä. DevOps tarjoaa myös automaation, josta on hyötyä mikropalvelujen kehityksessä todella paljon.

3.4.5. Pilvilaskenta

Pilvilaskenta (cloud computing) tarkoittaa Internetin välityksellä käytettäviä laskentaresursseja, kuten palvelimia, sovelluksia, tietovarastoja ja muita resursseja. Niitä ylläpitää *pilvipalveluiden tarjoaja* (cloud services provider) omassa datakeskuksessaan. Pilvilaskenta voidaan jakaa julkiseen, yksityiseen ja hybridiin. Julkisessa pilvessä pilvipalveluiden tarjoaja tarjoaa laskentaresursseja kaikille asiakkaille. Samoissa datakeskuksissa voi olla siis miljoonia asiakkaita. Yksityisessä pilvessä pilvilaskentaresurssit ovat varattuja vain yhdelle asiakkaalle. Yksityisen pilven resurssit voivat sijaita asiakkaan omassa datakeskuksessa tai muualta vuokratussa datakeskuksessa. Hybridipilvi on yksityisen ja julkisen pilven yhdistelmä. Tällöin organisaatio käyttää oman datakeskuksensa lisäksi myös julkisten pilvipalveluiden tarjoajien palveluita. (IBM, 2021)

Pilvessä tarjotaan tyypillisesti kolmenlaisia palveluita. *SaaS* (Software-as-a-Service, ohjelmisto palveluna) tarkoittaa pilvessä sijaitsevan sovelluksen käyttöoikeutta, joka myydään organisaatiolle esimerkiksi kuukausittaisella hinnalla tai käytön mukaan hinnoiteltuna. *PaaS* (Platform-as-a-Service, alusta palveluna) tarkoittaa pilvipalvelua, jossa pilvipalveluiden tarjoaja tarjoaa alustan sovellusten suorittamiseen. Tämä tarkoittaa esimerkiksi palveluita konttien suorittamiseen. *IaaS* (Infrastructure-as-a-Service, infrastruktuuri palveluna) tarkoittaa pilvipalvelua, jossa pilvipalveluiden tarjoaja tarjoaa infrastruktuurin sovellusten suorittamiseen. Se sisältää laskentaresurssit, verkot ja tallennustilan: asiakas huolehtii kaikesta muusta itse. (IBM, 2021)

Pilvipalvelut helpottavat kaikenlaista sovelluskehitystä, koska se helpottaa alustan ja infrastruktuurin hankkimista sovellukselle. Eri pilvipalveluiden tarjoajien PaaS-tarjonnat helpottavat erityisesti mikropalveluiden kehittäjiä, koska ne poistavat mikropalveluiden kehittäjiltä tarpeen huolehtia tarvittavista palvelinresursseista ja käyttöjärjestelmien asentamisesta.

3.5. Edut

Mikropalveluarkkitehtuurin hyödyntämisellä voi olla useita etuja sovelluksen kehityksessä. Kirjassaan *Microservice Architecture: Aligning Principles, Practices and Culture* Nadareishvili ja muut (2016) listaavat yritysten kokemia etuja mikropalveluiden aikaisesta käyttöönotosta, joita ovat:

- palvelujen itsenäinen skaalautuminen

- korkea saatavuus
- kehitystyön nopeus
- vähemmän riippuvuuksia tiimien välillä
- kehitystyön rinnakkaisuus
- tuki useammille teknologioille
- mahdollistaa palvelun *sulavan heikentymisen* (graceful degradation)
- ison järjestelmän ymmärrettävyys parempi.

Artikkelissaan *Pattern: Microservice Architecture* Richardson (2022e) listaa mikropalveluarkkitehtuurille seuraavanlaiset hyödyt:

- pienet palvelut ovat helppoja ymmärtää, ylläpitää ja muuttaa
- pienet palvelut ovat nopeita testata
- palvelut voidaan ottaa käyttöön itsenäisesti
- kehitystyön voi organisoida itsenäisille tiimeille, jotka vastaavat kokonaisuudessaan yhdestä tai useammasta palvelusta
- pienen palvelun myötä kehitystyökalut toimivat nopeammin, jolloin kehittäjät ovat tuotteliaampia
- pienet palvelut käynnistyvät nopeammin, lisäten kehittäjien tuotteliaisuutta ja nopeuttaen koko järjestelmän käyttönottoa
- vikojen eristys palveluiden sisälle
- eri palvelut voivat käyttää eri teknologioita, joten samaan teknologiaan ei tarvitse sitoutua pitkäksi aikaa.

Lopuksi, Knoche ja Hasselbring (2019) selvittivät tutkimuksessaan *Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany* houkuttimia ja esteitä mikropalveluiden käyttöönotolle ohjelmistokehittäjien keskuudessa. Tutkimuksen mukaan viisi suurinta syytä ottaa käyttöön mikropalvelut olivat (Knoche ja Hasselbring, 2019):

- korkea skaalautuvuus ja elastisuus
- korkea ylläpidettävyys
- nopea markkinoille pääsy
- mahdollistaa jatkuvan toimituksen ja DevOpsin
- sopii yhteen pilvipalveluiden ja kontituksen kanssa.

4 Mikropalveluiden haasteet

Tässä luvussa tutustutaan tarkemmin mikropalveluiden käyttöön liittyviin haasteisiin. Aluksi esitellään käytetty tutkimusmenetelmä, joka on kirjallisuuskatsaus. Seuraavaksi tutustutaan aiheesta tehtyihin aiempiin tutkimuksiin ja esitellään tutkimuksen valittu lähdekirjallisuus. Lopuksi kirjallisuuskatsauksen tulokset analysoidaan.

4.1. Tutkimusmenetelmä

Tutkimusmenetelmän pohjana toimi systemaattinen kirjallisuuskatsaus. Systemaattinen kirjallisuuskatsaus on tutkimustapa, jossa on tarkoituksena löytää, arvioida ja tulkita kaikki saatavilla oleva tutkimus tiettyyn aiheeseen liittyen. Systemaattisessa kirjallisuuskatsauksessa tutkimusmetodi dokumentoidaan huolellisesti, jotta lukija voi arvioida myös sen. Systemaattinen kirjallisuuskatsaus koostuu kolmesta vaiheesta, joita ovat katsauksen suunnittelu, katsauksen suoritus ja katsauksen raportointi. Suunnitteluvaihe koostuu kahdesta vaiheesta, joista ensimmäinen on katsauksen tarpeen tunnistus ja toinen katsauksen protokollan suunnittelu. Toteutusvaihe koostuu viidesta vaiheesta, joita ovat tutkimuksien löytäminen, tutkimuksien valinta, laadun arviointi, tiedon hankinta ja tiedon yhdistely. Raportointivaihe on yksivaiheinen. (Kitchenham, 2004)

Tämän tutkimuksen tarkoituksena on löytää vastaus tutkimuskysymyksiin, jotka ovat:

- *Minkälaisia haasteita mikropalveluiden kehittäjät kokevat?*
- *Minkälaisia ratkaisuja näihin haasteisiin on?*

Katsauksen lähdemateriaaliksi haluttiin pääasiassa tutkimuksia, joissa mikropalveluiden kehittäjien ääni kuuluu, jotta tulokset olisivat mahdollisimman lähellä oikeaa tilannetta. Kehittäjä on henkilö, joka on jollain tavalla mukana mikropalveluiden kehityksessä esimerkiksi ohjelmistokehittäjänä, ohjelmistoarkkitehtina tai projektipäällikkönä. Tällaiselle katsaukselle oli tarvetta, koska kohdassa 4.2 mainituissa aiemmissa tutkimuksissa ei löytynyt tutkimusta vastaavalla näkökulmalla.

Kirjallisuutta etsittiin kuudella eri tieteellisen kirjallisuuden hakukoneella. Hakukoneesta ja tuloksien määrästä riippuen hakuja tehtiin hieman erilaisilla hakusanoilla. Hakutuloksista otettiin tarkempaan tarkasteluun tapaustutkimukset, haastattelut, kyselyt ja empiiriset tutkimukset, jotka olivat vertaisarvioituja ja englanninkielisiä. Näistä valittiin lähdeaineistoon ne, joissa käsiteltiin riittävästi mikropalvelujen käytön haasteita, vaikeuksia tai ongelmia. Taulukossa 4 on lueteltuna kaikki tehdyt haut.

Taulukko 4. Kirjallisuuskatsauksen haut.

Hakukone	Hakusana	Tuloksia	Valittu tutkimukseen
Andor	(microservice OR micro-service) AND (survey OR study)	534	14
ACM	(microservice OR micro-service) AND (survey OR study)	192241	tarkennettu hakusanaa
ACM	(microservice* OR "micro-service*") AND (survey OR study)	1488	tarkennettu hakusanaa
ACM	(microservice* OR "micro-service*") AND (survey OR study) AND (challenge* OR problem*)	1417	4
ProQuest Computer Science Database	(microservice* OR "micro-service*") AND (survey OR study)	960	0
IEEEXplore	(microservice* OR "micro-service*") AND (survey OR study)	510	4
ScienceDirect	(microservice OR "micro-service") AND (survey OR study) AND (challenge OR problem)	1349	1
SpringerLink	(microservice* OR "micro-service*") AND (survey OR study) AND (challenge* OR problem*)	457	0

4.2. Aiemmat tutkimukset

Mikropalveluiden käyttöön liittyvistä haasteista on tehty muutamia aiempia kirjallisuuskatsauksia.

Alshuqayran ja muut (2016) tutkivat kirjallisuuskatsauksessaan *A Systematic Mapping Study in Microservice Architecture* mikropalveluiden arkkitehtuurisia haasteita, arkkitehtuurin kuvaamiseen käytettyjä diagrammeja tai näkymiä sekä arkkitehtuurin laatutekijöitä. Katsauksen lähteinä käytettiin konferenssijulkaisuja, tieteellisiä artikkeleita ja työpajajulkaisuja (Alshuqayran ja muut, 2016). Tutkimuksen mukaan mikropalveluiden haasteet liittyivät seuraaviin seikkoihin, useiten mainitusta haasteesta vähiten mainittuun (Alshuqayran ja muut, 2016):

1. palveluiden välinen kommunikaatio ja integrointi
2. käyttöönotto
3. palveluiden löytyminen
4. suorituskyky
5. vikasietoisuus
6. sovelluksen suorituskyvyn tarkkailu

7. turvallisuus
8. jäljitys ja lokitus.

Kalske ja muut (2018) tutkivat kirjallisuuskatsauksessaan *Challenges When Moving from Monolith to Microservice Architecture* erityisesti yritysten mahdollisesti kohtaamiin haasteisiin siirtyessä monoliittisesta arkkitehtuurista mikropalveluihin. Katsaus käyttää lähteenään mikropalveluista kirjoitettua kirjallisuutta ja mikropalveluihin siirtymisestä tehtyjä tapaustutkimuksia (Kalske ja muut, 2018). He jakoivat mikropalveluihin siirtymiseen liittyvät haasteet kahteen kategoriaan: tekniset haasteet ja organisaatioon liittyvät haasteet (Kalske ja muut, 2018). Heidän mukaan tekniset haasteet olivat (Kalske ja muut, 2018):

- monoliittisen sovelluksen jakaminen palveluihin
- mikropalveluiden välinen integraatio
- mikropalveluiden käyttöönotto tuotantoympäristössä
- mikropalveluiden lokitus ja monitorointi
- mikropalveluiden vikasietoisuus
- mahdollinen erilaisten tietokantojen määrä.

Organisaatioon liittyviä haasteita olivat (Kalske ja muut, 2018):

- organisaatiorakenteen muutos
- kokonaisvastuun antaminen palvelusta vastaavalle tiimille
- tiimit voivat tarvita lisäosaamista.

Soldani ja muut (2018) tutkivat kirjallisuuskatsauksessaan *The pains and gains of microservices: A Systematic grey literature review* mikropalvelujen hyötyjä ja haittoja. Tutkimuksen lähteenä käytettiin harmaata kirjallisuutta (Soldani ja muut, 2018). Heidän mukaan mikropalveluiden suurimmat haasteet ovat (Soldani ja muut, 2018):

- mikropalveluiden monimutkaisuus
- resurssien kulutus
- palvelujen koon määritys
- turvallisuus
- tiedon yhtenäisyys
- testaus.

Waseem ja muut (2020) tutkivat kirjallisuuskatsauksessaan *A Systemic Mapping Study on Microservices Architecture in DevOps* mikropalveluiden käyttöä DevOpsin kanssa. Katsauksen lähteenä toimivat pääasiassa tieteelliset konferenssijulkaisut ja artikkelit, mutta myös muut kirjat ja julkaisut. He luokittelivat katsauksessaan kirjallisuudesta löydetyt haasteet kategorioihin ja etsivät niihin myös ratkaisut (Waseem ja muut, 2020). Heidän tutkimuksessaan jäi ainoastaan kolme haastetta ilman ratkaisua (Waseem ja muut, 2020):

- suorituskykyongelmat johtuen liiallisesta mikropalveluiden välisestä kommunikaatiosta
- turvallisuusongelmat
- ajonaikaisten arkkitehtuuristen mallien generointi.

Hassan ja muut (2020) tutkivat kirjallisuuskatsauksessaan *Microservice transition and its granularity problem: A systematic mapping study* erityisesti mikropalveluiden käyttöönottoa ja mikropalveluiden *rakeisuutta* (granularity). Katsauksessa käytettiin lähteenä pääasiassa konferenssiartikkeleita, mutta myös muita lähteitä laidasta laitaan, kuten blogitekstejä, videoita, tieteellisiä artikkeleita ja tutkielmia. Rakeisuudella tarkoitetaan palvelun kokoa ja toimialaa. Sen avulla voidaan määrittää, pystyykö palvelua pilkkomaan vielä pienempiin osiin. Mikropalvelun oikean koon määrittäminen on heidän mukaansa yksi haaste joka tarvitsee lisätutkimusta. (Hassan ja muut, 2020)

Lopuksi, Söylemez ja muut (2022) tutkivat kirjallisuuskatsauksessaan *Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review* mikropalveluarkkitehtuurin haasteita ja ratkaisuja niihin. Katsaus on vuodelta 2022 ja aiemmasta tutkimuksesta uusin. Tutkimuksen lähteenä toimivat tieteelliset artikkelit, konferenssijulkaisut, työpajajulkaisut ja kirjat (Söylemez ja muut, 2022). Lähteiden tutkimusmenetelminä oli pääasiassa tapaustutkimukset (Söylemez ja muut, 2022). He löysivät tutkimuksessaan seuraavat haasteet, järjestyksessä useiten mainitusta vähiten mainittuun (Söylemez ja muut, 2022):

1. palveluiden orkestrointi
2. monitorointi, jäljitys ja lokitus
3. suorituskyvyn ennustus, mittaus ja optimointi
4. testaus
5. turvallisuus
6. palveluiden löytyminen
7. sovelluksen jakaminen mikropalveluihin
8. tiedon hallinta ja yhtenäisyys
9. kommunikaatio ja integrointi.

4.3. Aineiston esittely

Kirjallisuuskatsauksen aineistoon valittiin yhteensä 23 lähdettä, jotka ovat esiteltyinä taulukossa 5.

Taulukko 5. Kirjallisuuskatsauksen lähteet.

Nro	Vuosi	Tekijät	Nimi	Tutkimusmetodit
1	2017a	Taibi ja muut	*Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation*	haastattelu
2	2017b	Taibi ja muut	*Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages*	työpaja
3	2018	Bucchiarone ja muut	*From Monolithic to Microservices: An Experience Report from the Banking Domain*	tapaustutkimus
4	2018	Di Francesco ja muut	*Migrating Towards Microservice Architectures: An Industrial Survey*	haastattelu, kysely
5	2018	Haselbock ja muut	*An Expert Interview Study on Areas of Microservice Design*	haastattelu
6	2018	Luz ja muut	*An experience report on the adoption of microservices in three Brazilian government institutions*	itsetarkkailu, kysely
7	2019	da Silva ja muut	*An Experience Report from the Migration of Legacy Software Systems to Microservice Based Architecture*	tapaustutkimus
8	2019	Fritzsch ja muut	*Microservices Migration in Industry: Intentions, Strategies, and Challenges*	haastattelu
9	2019	Ghofrani ja Bozorgmehr	*Migration to Microservices: Barriers and Solutions*	haastattelu
10	2019	Knoche ja Hasselbring	*Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany*	kysely
11	2019	Niedermaier ja muut	*On Observability and Monitoring of Distributed Systems – An Industry Interview Study*	haastattelu
12	2019	Zhang ja muut	*Microservice Architecture in Reality: An Industrial Inquiry*	haastattelu
13	2021	Bogner ja muut	*Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review*	haastattelu, kirjallisuuskatsaus
14	2021	de Toledo ja muut	*Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study*	moni-tapaustutkimus
15	2021	Laigner ja muut	*Data management in microservices: State of the practice, challenges, and research directions*	kirjallisuuskatsaus, kysely, tapaustutkimus
16	2021a	Waseem ja muut	*On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study*	empiirinen tutkimus

17	2021b	Waseem ja muut	*Design, monitoring, and testing of microservices systems: The practitioners' perspective*	haastattelu, kysely
18	2021	Wang ja muut	*Promises and challenges of microservices: an exploratory study*	kysely, haastattelu
19	2021	Zhou ja muut	*Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study*	empiirinen tutkimus, kysely
20	2022	de Toledo ja muut	*Accumulation and Prioritization of Architectural Debt in Three Companies Migrating to Microservices*	moni-tapaustutkimus
21	2022	Li ja muut	*Enjoy your observability: an industrial survey of microservice tracing and analysis*	kysely
22	2022a	Zhou ja muut	*A cross-company ethnographic study on software teams for DevOps and microservices: organization, benefits, and issues*	haastattelu
23	2022b	Zhou ja muut	*Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry*	haastattelu

Kirjallisuuskatsauksen aineistossa on käytetty tutkimusmetodeina pääasiassa haastatteluita ja kyselyitä: 87 prosenttia lähteistä käytti tutkimusmetodina näistä ainakin toista. Taulukko 6 sisältää listan kaikista käytetyistä tutkimusmetodeista ja tiedon kuinka monessa lähteessä niitä on käytetty. Iältään aineisto on melkoisen tuoretta. Määrällisesti eniten lähteitä on vuodelta 2021, ja myös vuodelta 2022 on mukana 4 lähdettä. Taulukko 7 sisältää listan aineiston julkaisuvuosista ja niiden määristä. Aiempiin tutkimuksiin verrattuna tämän kirjallisuuskatsauksen painopiste on selkeästi enemmän haastatteluissa ja kyselyissä, mutta se sisältää myös uudempaa lähdemateriaalia.

Taulukko 6. Aineiston tutkimusmetodit.

Tutkimusmetodi	Käytössä tutkimuksissa (kpl)
haastattelu	12
kysely	8
tapaustutkimus	3
empiirinen tutkimus	2
kirjallisuuskatsaus	2
moni-tapaustutkimus	2
itsetarkkailu	1
työpaja	1

Taulukko 7. Aineiston julkaisuvuodet.

Vuosi	Tutkimuksia (kpl)
2017	2
2018	4
2019	6
2021	7
2022	4

4.4. Löydetyt haasteet

Kirjallisuuskatsauksen aineisto luettiin läpi ja niistä etsittiin mikropalveluihin liittyvät haasteet. Löydetyt haasteet jaettiin kategorioihin kahdessa tasossa. Ylemmällä tasolla haasteet jaettiin kahteen kategoriaan sen mukaan, liittyivätkö ne erityisesti vanhan sovelluksen migraatioon mikropalveluihin vai ei. Alemmalla tasolla toisiinsa liittyvät haasteet jaettiin samoihin kategorioihin.

Taulukossa 8 on listattuna yleiset mikropalveluihin liittyvät haasteiden kategoriat. Taulukosta löytyy myös tieto, missä tutkimuksissa löytyy maininta kyseisestä haasteesta sekä tutkimuksien lukumäärä, joista maininta löytyi. Taulukko on lajiteltu tutkimuksien lukumäärän mukaan laskevasti. Taulukkoa tarkastelemalla voidaan havaita, että suurin osa haasteista on teknisiä haasteita. Mikropalveluiden suunnitteluun liittyvät haasteet olivat eniten esillä, mikä löytyy myös aiemmista tutkimuksista. Kuitenkin, verrattuna uusimpaan aiempaan tutkimukseen, tulokset eroavat jonkin verran. Taulukon 8 kolme tärkeintä haastetta ovat mikropalveluiden suunnittelu, palveluiden välinen kommunikaatio ja integrointi sekä käyttönotto ja DevOps. Söylemezin ja muiden tutkimuksessa mikropalveluiden suunnittelu oli vasta seitsemäntenä, palveluiden välinen kommunikaatio ja integrointi yhdeksäntenä ja käyttöönotto ja DevOps ensimmäisenä (Söylemez ja muut, 2022). Lisäksi on huomioitava, että aiemmista tutkimuksista vain Kalske ja muut (2018) mainitsivat erikseen organisaatioon ja ihmisiin liittyviä haasteita. Tässä kirjallisuuskatsauksessa niitä on taulukon 8 mukaan löytynyt jopa seitsemästä eri tutkimuksesta. Taulukossa 8 mainittuja haasteita käydään tarkemmin läpi kohdan 4.4 alakohdissa.

Taulukko 8. Mikropalveluiden haasteet tutkimuksissa.

Haaste	Tutkimuksissa (tutkimus nro:t)	Tutkimuksia (kpl)
Mikropalveluiden suunnittelu	2, 3, 4, 5, 6, 8, 11, 12, 13, 14, 15, 16, 17, 20, 22, 23	16
Palveluiden välinen kommunikaatio ja integrointi	1, 5, 8, 9, 12, 13, 14, 16, 17, 20, 23	11
Käyttöönotto ja DevOps	1, 4, 5, 6, 10, 12, 13, 16, 22, 23	10
Monitorointi, logitus ja debuggaus	2, 3, 4, 5, 9, 11, 12, 17, 19, 21, 23	10
Tiedonhallinta	5, 9, 10, 12, 14, 15, 16, 17, 20, 23	10
Mikropalveluiden itsenäisyys	4, 5, 11, 12, 13, 14, 15, 20, 23	9
Testaus	2, 5, 5, 9, 12, 13, 16, 17, 23	8
Organisaatio ja ihmiset	2, 4, 11, 12, 13, 17, 23	7
Turvallisuus	5, 6, 8, 9, 16, 17	6
Muutoshallinta	2, 5, 12, 13, 18, 23	6
Koodin hallinta	13, 14, 16, 18, 20	5
Suorituskyky	22, 23	2
Palveluiden löytyvyys	5	1

Taulukossa 9 on listattuna migraatioon liittyvien haasteiden kategoriat. Taulukko sisältää ainoastaan haasteet, jotka koskevat migraatiota. Osa migraation aikana koetuista haasteista koskee myös yleisesti mikropalveluiden käyttöä. Nämä haasteet löytyvätkin muiden haasteiden joukosta taulukosta 8. Taulukosta 9 löytyy myös tieto, missä tutkimuksissa löytyy maininta kyseisestä haasteesta sekä tutkimuksien lukumäärä, joista maininta löytyi. Taulukko on lajiteltu tutkimuksien lukumäärän mukaan laskevasti. Taulukkoa tarkastelemalla voidaan havaita, että migraatioon liittyvistä haasteista noin puolet oli muita kuin teknisiä haasteita. Organisaatioon ja ihmisiin liittyviä haasteita löytyi 11 tutkimuksesta, ja hintaan liittyviä haasteita löytyi kolmesta tutkimuksesta. Aiemmissa tutkimuksissa ei eritelty mitkä haasteista liittyivät juuri migraatioon ja mitkä eivät. Taulukossa 9 mainittuja haasteita käydään tarkemmin läpi kohdan 4.4 alakohdissa.

Taulukko 9. Migraatioon liittyvät haasteet tutkimuksissa.

Haaste	Tutkimuksissa (tutkimus nro:t)	Tutkimuksia (kpl)
Organisaatio ja ihmiset migraatiossa	1, 2, 4, 5, 6, 8, 9, 10, 12, 22, 23	11
Vanhan järjestelmän haasteet migraatiossa	2, 4, 6, 7, 8, 9	6
Uuden järjestelmän haasteet migraatiossa	1, 2, 8, 9, 10	5
Migraation hinta	1, 6, 22	3
Tekninen migraatio	1, 9	2

4.4.1. Mikropalveluiden suunnittelu

Mikropalveluiden suunnitteluun liittyviä haasteita löytyi yhteensä 16 tutkimuksesta. Kategoria on laaja, ja se sisältää kaikki mikropalveluarkkitehtuurin tekniseen suunnitteluun liittyvät haasteet, joille ei annettu omaa kategoriaansa. Suurimmat haasteet olivat palveluiden rajojen ja koon määritys (mainittu 9 tutkimuksessa), järjestelmän monimutkaisuus (mainittu 6 tutkimuksessa), palveluiden väliset liitokset ja riippuvuudet (mainittu 5 tutkimuksessa) sekä vikasietoisuus (mainittu 5 tutkimuksessa). Tekninen velka mainittiin haasteena kolmessa tutkimuksessa, puutteellinen APIen suunnittelu ja rinnakkaisuuden hallinta mainittiin haasteena kahdessa tutkimuksessa kumpikin. Lisäksi haasteiksi mainittiin yksittäisissä tutkimuksissa mikropalveluiden käyttö väärässä paikassa, epäselvät ei-funktionaaliset vaatimukset, skaalautuvuus, palveluiden mallinnus, toiminnallisuuden ja operoinnin erotus sekä tilanhallinta.

Tutkimuksista löytyi myös ehdotuksia ongelmien ratkaisuun. Tutkimus 17 löysi palveluiden rajojen ja koon määrityksen helpottamiseksi kuusi ehdotusta toimenpiteiksi (Waseem ja muut, 2021b):

- syvennä ymmärrystä vaatimuksista, toimialasta, teknologioista ja liiketoiminnan sekä kehityksen prosesseista
- määritä mikropalveluiden rajat *minipalveluiden* (miniservice) avulla
- hyödynnä toimialapohjaista suunnittelua kontekstien ja alitoimialojen määrityksessä
- löydä jokaisen alitoimialan yhteiset ja eristetyt toiminnallisuudet
- löydä sopiva rakeisuustaso mikropalveluille
- varmista mikropalveluiden rajojen näkyvyys.

Samassa tutkimuksessa ehdotettiin ratkaisuja myös järjestelmän monimutkaisuuden hallintaan. Tutkimuksessa löydettiin seuraavat neljä ehdotusta toimenpiteiksi (Waseem ja muut, 2021b):

- älä suunnittele tai kehitä mikropalvelua jokaisesta järjestelmän ominaisuudesta
- määritä hallittavissa oleva joukko vastuita jokaiselle mikropalvelulle
- ymmärrä mikropalveluiden toimintaympäristö
- määritä selkeästi mikropalveluiden rajat.

Tutkimus 14 löysi ehdotuksia useammankin haasteen ratkaisemiseksi tästä kategoriasta. Mikropalveluiden liitoksiin liittyviin haasteisiin tutkimus löysi seuraavat ehdotukset toimenpiteiksi (de Toledo ja muut, 2021):

- käytä aikaa suunnitellessa geneerisiä ja itsenäisiä palveluita
- kehittäjien koulutus API-rajapintojen kehityksestä
- "API ensin" -lähestymistapa palveluita suunnitellessa
- kehityksen aikana varattu aikaa pelkästään jatkuvalle APIn parantamiselle.

Vikasietoisuuden parantamiseksi tutkimus 14 ehdotti kolmannen osapuolen tekemien tuotteiden, jotka tarjoavat vikasietoisuuteen liittyviä ominaisuuksia, käyttämistä, kuten esimerkiksi kierron rikkojia tai *palveluverkkoa* (service mesh) (de Toledo ja muut, 2021).

4.4.2. Palveluiden välinen kommunikaatio ja integrointi

Palveluiden väliseen kommunikointiin ja integrointiin liittyviä haasteita löytyi yhteensä 11 tutkimuksesta.

Tutkimus 14 mainitsi useita haasteita liittyen palveluiden väliseen kommunikaatioon ja esitti näille myös ratkaisuja. Viestien jäljitettävyyteen ja *kuolleen kirjeen jonon* (dead letter queue) hallitsemattomaan kasvuun se ehdotti seuraavia toimenpiteitä (de Toledo ja muut, 2021):

- lisää viestin metadataan palvelun omistajuustiedot jotta viestin lähde selviää
- toteuta *kanoninen datamalli* (canonical data model)
- kuolleen kirjeen jonon poisto ja vastuun siirto viestin välityksestä päätepisteille
- kuolleen kirjeen jonon jako useampaan pieneen jonoon jotka ovat eri tiimien hallinnoimia.

Palveluiden välisen kommunikaation standardointiin se ehdotti ratkaisuksi kanonisen datamallin toteutusta (de Toledo ja muut, 2021). API-rajapintojen suunnitteluun tutkimus ehdotti seuraavia ratkaisuja (de Toledo ja muut, 2021):

- vakauta API-rajapinta ja vältä muutoksia
- API-rajapintojen versionhallinta, rajapintojen käyttäjien seuranta ja selkeä *deprekoinnin* (deprecation, "ominaisuuden merkitseminen niin, että se käyttöä tulisi välttää, koska se poistetaan käytöstä myöhemmin"(Techopedia, 2022)) strategia
- määrittele standardit rajapinnoille
- liian monimutkaiset API-kutsut voi korvata viestinvälityksellä.

4.4.3. Käyttöönotto ja DevOps

Käyttöönottoon ja DevOpsiin liittyvät haasteet saivat yhteisen kategorian, koska ne liittyvät hyvin paljon toisiinsa. Alakohdassa 3.4.4 kerrotaan, kuinka DevOpsista on apua mikropalveluarkkitehtuurissa. Haasteita löytyi yhteensä 10 tutkimuksesta. Suurin osa haasteista tässä kategoriassa liittyi joko palveluiden käyttöönottoon (mainittu 6 tutkimuksessa) tai automaatioon ja DevOpsiin (mainittu 5 tutkimuksessa). Infrastruktuuri mainittiin haasteena kahdessa tutkimuksessa, palveluiden orkestrointi ja pilvipalveluiden käyttö kumpikin yhdessä tutkimuksessa.

Tutkimuksessa 5 havaittiin, että osa haastatelluista kertoi suuren, usean sadan mikropalvelun järjestelmän käynnistyksen olevan käytännössä mahdotonta ilman automaatiota. Tutkimuksen mukaan tärkeintä onkin tehdä käyttöönottoputki, joka tekee kaiken automaattisesti uuden koodin latauksesta uuden mikropalvelun käynnistykseen tuotannossa. Tutkimuksessa huomattiin myös, että pilvipalveluiden käyttökin on lähes pakollista mikropalveluita käytettäessä, koska niihin kuuluuu monia hyödyllisiä ominaisuuksia kuten palveluiden orkestrointi, kuorman tasaus ja palveluiden löytyvyys. (Haselbock ja muut, 2018)

4.4.4. Monitorointi, logitus ja debuggaus

Monitorointi, logitus ja debuggaus ovat toimintoja, jotka liittyvät kaikki jollain tavalla toisiinsa, joten myös ne saivat yhteisen kategorian. Tähän kategoriaan liittyviä haasteita löytyi yhteensä 11 tutkimuksesta. Palveluiden monitorointi mainittiin haasteena yhteensä 9 tutkimuksessa. Seuraavaksi eniten mainitut haasteet olivat palveluiden debuggaus (mainittu 5 tutkimuksessa) ja palveluiden lokitus (mainittu 4 tutkimuksessa). Yksittäisissä tutkimuksissa mainittuja haasteita olivat jäljitysdatan analyysi ja monitoroinnin toteutus vasta kun tulee vaikeuksia.

Tutkimuksessa 11 ehdotettiin monitorointiin liittyviin haasteisiin ratkaisuksi seuraavia ratkaisuja (Niedermaier ja muut, 2019):

- tapahtumien ja topologian hallinta
- *hajautettu jäljitys* (distributed tracing)
- *SRE* (Site Reliability Engineering, paikan luotettavuuden rakentaminen)
- *synteettinen tunnustelu* (synthetic probing)
- *APH* (Application Performance Management, sovelluksen suorituskyvyn hallinta)
- metadata
- *SLO* (Service Level Objective, palvelutason tavoite)*SLI* (Service Level Indicator, palvelutason indikoija)
- erillinen asiantuntijoista koostuva monitorointitiimi

- praktiikkayhteisöt parhaiden käytäntöjen jakamiseen
- poikkeuskäytäntöjen ja taksonomioden luonti poikkeamille
- valmiiden ja standardoitujen komponenttien käyttö
- standardoidun APIn tarjoaminen erilaisia monitorointiteknologioita varten
- tekoälyn käyttö analysoinnissa
- bottien ja agenttien käyttö.

Tutkimuksessa 17 ehdotettiin monitorointiin liittyviin haasteisiin ratkaisuksi seuraavia työkaluja (Waseem ja muut, 2021b):

- ELK (Elasticsearch, Logstash ja Kibana)
- Prometheus ja Sprint Boot Actuator
- Grafana
- Zipkin
- Raygun APM
- *API-rajapinta terveystarkastukseen* (health check API)
- sisäisesti kehitetty monitorointikehys.

4.4.5. Tiedonhallinta

Tiedonhallintaan liittyviä haasteita löytyi yhteensä 10 eri tutkimuksesta. Tähän kategoriaan kuuluu myös yhdessä tutkimuksessa erikseen haasteena mainittu tiedon tai pyyntöjen käsittely, joka kattaa useamman mikropalvelun toimialueen.

Tutkimus 14 ehdottaa tiedonhallinnan helpottamiseksi seuraavia toimenpiteitä (de Toledo ja muut, 2021):

- jokaiselle palvelulle oma tietokanta TAI
- yhden tietokannan järjestelmissä jokaiselle palvelulle oma tietokantaskeema
- tietokannan suoran käytön estäminen
- tiedon synkronisaation varmistamiseksi voidaan käyttää jaettua tietokantaa.

Tutkimuksen 15 mukaan mikropalveluiden tiedonhallinnan haasteet liittyvät samanaikaisuuteen, tiedon jakautumiseen, tiedon replikointiin, *siirtoihin* (transaction) ja tiedon *johdonmukaisuuteen* (consistency). Se ehdottaa mikropalveluiden tiedonhallinnan vaikeuksien helpottamiseksi uudenlaista tietokantajärjestelmää, joka olisi tehty mikropalveluita ajatellen ja helpottaisi muunmuassa tiedon johdonmukaisuuden hallitsemisessa ja monen palvelun tiedon välisiä kyselyitä. (Laigner ja muut, 2021)

4.4.6. Mikropalveluiden itsenäisyys

Mikropalveluiden itsenäisyys nostettiin omaksi kategoriakseen, koska siihen löytyi myös useamman tyyppisiä haasteita. Palveluiden heterogeenisuus ja liiallinen teknologioiden määrä mainittiin haasteena yhteensä 7 tutkimuksessa. Yksittäisissä tutkimuksissa mainittuja haasteita olivat palveluiden yhtenäisyyden varmistaminen sekä yhtenäisen käyttöliittymän tekeminen monille eri palveluille.

Tutkimus 14 ehdotti ratkaisuksi liialliselle teknologioiden määrälle sallittujen teknologioiden rajaamista sekä ohjelmointikielikohtaisten repositorioiden käyttöönottoa (de Toledo ja muut, 2021).

4.4.7. Testaus

Mikropalveluiden testaus mainittiin haasteena yhteensä 9 tutkimuksessa. Tutkimuksen 14 mukaan suurimmat haasteet mikropalveluiden testauksessa ovat manuaalisten testien luominen ja suoritus, mikropalveluiden integraatiotestaus ja mikropalveluiden debuggaus kontitusalustoilla (de Toledo ja muut, 2021). Tutkimuksessa muita havaittuja haasteita olivat automaattisten testien luominen ja suoritus, mikropalveluiden itsenäisyys, monimutkaisuus ja koko, jokaisen testattavan mikropalvelun ymmärtäminen, suorituskykytestaus, monta itsenäistä tiimiä sekä mikropalveluissa käytetty teknologioiden määrä (de Toledo ja muut, 2021).

Tutkimus 14 ehdotti seuraavat ratkaisut helpottamaan mikropalveluiden testaamisen haasteita (de Toledo ja muut, 2021):

- testiautomaation ja DevOpsin käyttöönotto
- testienhallintaohjelmiston käyttöönotto
- *BDD:n* (Behavior-Driven Development, käytöslähtöinen kehitys) käyttäminen kehityksessä helpottaa automaattisten testien ja integraatiotestauksen toteutusta
- projektille voi antaa erillisen vaatimuksen, että testitapausten tulee olla yhtenäisiä ja testipeiton tulee olla tarpeeksi suuri
- aloita testaus yksittäisestä mikropalvelusta osaavan henkilökunnan kanssa
- jokaisella mikropalvelulla tulisi olla oma tietokanta
- testaa mikropalvelujärjestelmät perinpohjaisesti yksikkö-, integraatio-, komponentti- ja *E2E* (End-to-end, päästä päähän)-testistrategioiden avulla.

4.4.8. Organisaatio ja ihmiset

Teknisten haasteiden lisäksi myös organisaatioon ja ihmisiin liittyviä haasteita löytyi lähdemateriaalista peräti 7 eri tutkimuksesta. Kehittäjien ja tiimien välinen kommunikaatio

mainittiin haasteena yhteensä 3 eri tutkimuksessa. Tähän liittyen, myöskin 3 eri tutkimuksessa mainittiin haasteena se, että organisaatio ja sovelluksen arkkitehtuuri ei vastaa toisiaan. Mikropalveluiden vaatima osaaminen nähtiin myös haasteena: 3 tutkimuksessa mainittiin haasteena tarve kokeneille kehittäjille, 2 tutkimuksessa mainittiin haasteena osaamisen puute, vaikeus oppia teknologiat mainittiin haasteena 1 tutkimuksessa ja vaikeus rekrytoida DevOps-osaajia mainittiin haasteena yhdessä tutkimuksessa. Lopuksi, ihmisten turhautuminen työkaluihin mainittiin haasteena yhdessä tutkimuksessa.

Alakohdassa 4.4.14 käsitellään organisaatioon ja ihmisiin liittyviä haasteita erityisesti migraation näkökulmasta. Siinä esitetyt haasteiden ratkaisuehdotukset sopivat ratkaisuksi myös yleisesti organisaatioon ja ihmisiin liittyviin haasteisiin.

4.4.9. Turvallisuus

Mikropalveluiden turvallisuus mainittiin haasteena 6 tutkimuksessa.

Tutkimus 14 mainitsi seuraavat ehdotukset turvallisuuden parantamiseksi mikropalveluissa (de Toledo ja muut, 2021):

- mikropalvelujärjestelmän voi turvata ostamalla päivitetyn turvallisuusratkaisun pilvipalvelujen tarjoajalta
- loppukäyttäjien pitäisi säännöllisesti päivittää palomuurinsa ja noudattaa muitakin turvallisuusstandardeja
- *DevSecOpsin* (Development Security Operations, kehitys turvallisuus operointi) käyttöönotto.

4.4.10. Muutoshallinta

Muutoshallintaan liittyviä haasteita mainittiin yhteensä 6 tutkimuksessa. Muutoshallintaan liittyvistä haasteista suurin oli mikropalveluiden versionhallinta (mainittu 4 tutkimuksessa). Suurena haasteena koettiin myös sovelluksen toiminnallisuuden rikkovat API-rajapintamuutokset, joka mainittiin 3 tutkimuksessa. Lopuksi, yhdessä tutkimuksessa mainittiin haasteena uuden toiminnallisuuden lisäämisen hitaus.

Tutkimus 18 ehdottaa API-rajapintojen muutoshallintaa ja versionhallintaa helpottamaan seuraavat ratkaisut (Wang ja muut, 2021):

- rajapinnan muutokset deprekoimalla ominaisuuksia poistamisen sijaan
- API-yhdyskäytävän tai asiakaskirjaston käyttö.

4.4.11. Koodin hallinta

Koodin hallintaan liittyviä haasteita löytyi useita, mutta yhteensä vain 5 eri tutkimuksesta. Puutteellinen dokumentaatio mainittiin haasteena 2 tutkimuksessa, kuten myös yhteisen koodin hallinta. Yksittäisissä tutkimuksissa mainittuja haasteita oli enemmän: koo-

din duplikointi, hajautetut repositoriot, ongelmien ratkaisun pitkä kesto, ulkoisten riippuvuuksien suuri määrä, mikropalveluiden suuri määrä, vanhan koodin integrointi ja varianttien tuki.

Tutkimus 18 ehdottaa koodin hallintaan liittyvien haasteiden ratkaisuksi seuraavia toimia (Wang ja muut, 2021):

- yhteisen koodin pakkaus yhteen jaettuun kirjastoon, joiden eri versioita palvelut voivat ottaa käyttöön
- yhteisen koodin sisällyttäminen yhteen erilliseen mikropalveluun
- *sivuvaunu-suunnittelumallin* (sidecar) käyttö, jossa kaikki yhteinen koodi on erillisessä mikropalvelussa ja se yhdistetään dynaamisesti mikropalveluun
- variantteja voidaan tukea *ominaisuuslipuilla* (feature flag), *roolipohjaisella pääsynhallinnalla* (role-based access control) tai erillisillä käyttöönotoilla.

4.4.12. Suorituskyky

Suorituskykyyn liittyviä haasteita mainittiin kahdessa eri tutkimuksessa. Yhdessä tutkimuksessa haasteeksi nimettiin mikropalveluiden suuri resurssien kulutus ja toisessa suorituskykyongelmat.

Tutkimus 23 mainitsee, palveluiden välinen kommunikaatio vaikuttaa koko järjestelmän suorituskykyyn olennaisesti. Se suositteleekin erityisesti suuren viestimäärän järjestelmiä käyttämään *RPC-pohjaista* (remote procedure call) kommunikaatiomenetelmää palveluiden välillä HTTP-kutsujen sijaan, koska RPC-pyyntöjen siirtämä tiedon määrä on pienempi. Tutkimus mainitsee myös mikropalveluiden koon vaikuttavan suorituskykyyn. Mitä pienempi toimiala mikropalveluilla on, sitä enemmän ne joutuvat kommunikoimaan keskenään ja se vaikuttaa suorituskykyyn. (Zhou ja muut, 2022b)

4.4.13. Palveluiden löytyvyys

Palveluiden löytyvyys mainittiin haasteena vain tutkimuksessa 5. Sitä ei kuitenkaan koettu kovin suureksi haasteeksi, koska palveluiden löytyvyyttä varten on olemassa useita teknologioita ja esimerkiksi pilvialustoissa palveluiden löytyvyys on integroitu toiminnallisuus (Haselbock ja muut, 2018).

4.4.14. Organisaatio ja ihmiset migraatiossa

Yleisistä mikropalveluiden haasteista poiketen, migraatiossa organisaatioon ja ihmisiin liittyvät haasteet koettiin suurimmaksi. Niitä löytyi yhteensä 11 tutkimuksesta. Suurimmat haasteet olivat organisaation muutokset (mainittu 5 tutkimuksessa), DevOpsin käyttöönotto (mainittu 4 tutkimuksessa) sekä kehittäjien asenteet, asenteiden muutos ja vastustus (mainittu 4 tutkimuksessa). Osaamisen puute ja työtapojen muutokset mainittiin

kumpikin 3 tutkimuksessa. Työmäärän arviointi ja migraation perustelu päättäjille mainittiin haasteena 2 tutkimuksessa. Lopuksi, rekrytointivaikeudet ja keskushallinnon puuttuminen projektiin mainittiin kumpikin yhdessä tutkimuksessa.

Tutkimus 4 ehdottaa migraation organisaatioon ja ihmisiin liittyviin haasteisiin ratkaisuksi tiedon jakamisen onnistuneista migraatioprojekteista (Di Francesco ja muut, 2018). Tutkimus 8 havaitsi, että organisaatioon ja ihmisiin liittyvät haasteet koettiin osittain jopa tärkeämmiksi kuin tekniset haasteet, joten organisaation kannattaa kiinnittää niihin erityisesti huomiota migraatiossa (Fritzsch ja muut, 2019). Tutkimus 23 ehdottaa, että DevOpsin ja mikropalveluiden käyttöönotossa tulisi olla mukana yksittäisten tiimien sijaan koko organisaation, jotta voitaisiin varmistaa myös ylemmän johdon tuki toteutukselle (Zhou ja muut, 2022b).

Tutkimus 9 ehdottaa migraation organisaatioon ja ihmisiin liittyviin haasteisiin useita ratkaisuja (Ghofrani ja Bozorgmehr, 2019):

- mikropalveluiden hyödyt on tehtävä näkyviksi
- autetaan tiimien kulttuurin muutoksissa ja parannetaan hallintoa
- tehdään ohjeistuksia mikropalveluiden käyttöön liittyen
- tarjotaan koulutusta.

4.4.15. Vanhan järjestelmän haasteet migraatiossa

Migraatiossa toiseksi eniten haasteita liittyi migratoitavaan, vanhaan järjestelmään. Tähän liittyviä haasteita mainittiin yhteensä 6 tutkimuksessa. Suurin haaste oli monoliitin purkaminen mikropalveluihin (mainittu 6 tutkimuksessa). Korkea liitosten määrä vanhassa järjestelmässä mainittiin haasteena 2 tutkimuksessa. Vanhan järjestelmän puutteellinen dokumentaatio, monimutkaisuus, omituisuudet ja vanhentuneet teknologiat saivat kukin yhden maininnan.

Tutkimus 4 ehdottaa, että migraation oikea aika on silloin, kun uusien ominaisuuksien lisääminen vanhaan järjestelmään kestää liikaa ja vanha järjestelmä on liian suuri (Di Francesco ja muut, 2018). Tutkimus 7 kertoo, että palvelun jakaminen mikropalveluihin vaatii palvelun toimialan tietämystä, mutta melko usein se tehdään teknologia edellä (da Silva ja muut, 2019). Se ehdottaakin, että palveluiden jakaminen tehtäisiinkin teknologiasta riippumatta (da Silva ja muut, 2019). Tutkimus 9 ehdottaa ratkaisuksi monoliitin purkamisen haasteisiin tutkimustiedon tarjoamista mallinnuksesta sekä toimialapohjaisen suunnittelun ja *mallipohjaisen kehityksen* (MDD, Model Driven Development) käyttämistä (Ghofrani ja Bozorgmehr, 2019).

4.4.16. Uuden järjestelmän haasteet migraatiossa

Uuteen järjestelmään liittyviä haasteita mainittiin monia erilaisia, mutta kukin haaste sai maininnan vain yhdessä tutkimuksessa. Niitä olivat uuden järjestelmän monimutkaisuus, yhteensopivuusongelmat, uusien teknologioiden maturiteetti, varmuuskopioinnin

vaikeus, jatkuvat arkkitehtuurin muutokset, muuttuvat vaatimukset ja monoliitista eroaminen.

Tutkimus 9 ehdotti migraation uuden järjestelmän haasteisiin seuraavia ratkaisuja (Ghofrani ja Bozorgmehr, 2019):

- tehdään ohjeistuksia
- tarjotaan parempia prototyyppejä, dokumentaatiota ja *nopean aloituksen oppaita* (quick start guides)
- parempi automatisoitu tuki alustavan mikropalveluarkkitehtuurin luomiselle
- tarjotaan koulutusta
- tarjotaan vertailukehys eri tyyppisille mikropalveluissa käytetyille teknologioille.

4.4.17. Migraation hinta

Migraation hintaan liittyviä haasteita mainittiin yhteensä 3 tutkimuksessa. Kahdessa tutkimuksessa mainittiin migraation kalleus haasteeksi. Yhdessä tutkimuksessa mainittiin haasteeksi migraation hidas *ROI* (return of investment, sijoitetun pääoman tuotto). Tutkimus 1 mainitsi haasteeksi migraation hitaan ROI:n, mutta mainitsi, että pitkällä aikavälillä mikropalvelujärjestelmät ovat kuitenkin halvempia, koska niiden ylläpitokustannukset ovat halvempia kuin monoliittisillä järjestelmillä (Taibi ja muut, 2017a).

4.4.18. Tekninen migraatio

Migraation tekniseen toteutukseen liittyviä haasteita löytyi yhteensä 2 tutkimuksesta. Tietokannan migraatio mainittiin haasteena 2 tutkimuksessa. Kirjastojen muuttaminen mainittiin haasteena 1 tutkimuksessa. Tutkimus 9 ehdottaa tietokannan migraatioon liittyviin haasteisiin ratkaisuksi jaettujen siirtojen ja *monikielisten tietokantojen* (polyglot databases) käyttöönottoa (Ghofrani ja Bozorgmehr, 2019).

5 Pohdinta

Tutkimuksen lähdemateriaali ajoittui aikavälille 2017-2022. Tutkimuksen hakuehdoissa julkaisuille ei oltu määritelty minkäänlaista aikaväliä, joten tutkimuksen lähdemateriaali oli melko tuoretta luonnostaan. Luvun 3 alussa mainittiin, että mikropalvelu terminä määriteltiin ensimmäisiä kertoja vasta vuonna 2014. Todennäköisesti juuri tästä syystä myös tutkimuksen materiaali alkoi vasta vuodesta 2017, sillä mikropalveluiden haasteita ei ole ehkä tätä ennen tutkittu juurikaan.

Aiempien tutkimuksien tuloksia mukaillen myös tässä tutkimuksessa havaittiin, että suurin osa haasteista oli luonteeltaan teknisiä. Haasteet ovat pääasiassa samoja, mutta niiden tärkeysjärjestys on muuttunut. Kohdassa 4.2 esiteltiin aiempia tutkimuksia ja listattiin niiden löytämiä haasteita tärkeysjärjestyksessä, jos se oli mainittuna tutkimuksessa.

Tämän tutkimuksen viisi tärkeintä yleistä mikropalveluihin liittyvää haastetta olivat järjestyksessä tärkeimmästä vähiten tärkeään mikropalveluiden suunnittelu, palveluiden välinen kommunikaatio ja integrointi, käyttöönotto ja DevOps, monitorointi, logitus ja debuggaus sekä tiedonhallinta. Yleisiä haasteita löytyi yhteensä 13. Lista haasteista tärkeysjärjestyksessä löytyy taulukosta 8.

Aiemmassa tutkimuksessa mikropalveluiden suunnitteluun liittyviä haasteita löytyi sijoilta 5 (yhteensä 8 haastetta) (Alshuqayran ja muut, 2016) ja 7 (yhteensä 9 haastetta) (Söylemez ja muut, 2022). Lisäksi niitä löytyi myös kahdesta muusta aiemmasta tutkimuksesta, joissa haasteille ei määritelty järjestystä. Tässä tutkimuksessa mikropalveluiden suunnitteluun liittyvät haasteet saivat paljon suuremman painoarvon kuin aiemmissa. Koska sijoitus oli melko alhainen aiemman tutkimuksen sekä varhaisimmassa että uusimmassa tutkimuksessa, ero tämän tutkimuksen sijoitukseen ei selity tutkimuksen ajankohdalla. Ero voi selittyä tutkimuksissa käytettyjen lähteiden tutkimusmenetelmillä. Tässä tutkimuksessa on käytetty pääasiassa kyselyitä ja haastatteluita sisältäviä lähteitä, joita aiemmassa tutkimuksessa ei käytetty juuri ollenkaan.

Aiemmassa tutkimuksessa palveluiden väliseen kommunikaatioon ja integrointiin liittyviä haasteita löytyi sijoilta 1 (yhteensä 8 haastetta) (Alshuqayran ja muut, 2016) ja 9 (yhteensä 9 haastetta) (Söylemez ja muut, 2022). Lisäksi niitä löytyi myös yhdestä aiemmasta tutkimuksesta, joissa haasteille ei määritelty järjestystä. Tässä tutkimuksessa sijoitus (2) oli paljon lähempänä vanhinta aiempaa tutkimusta, jossa sijoitus oli ensimmäinen, kuin uusinta, jossa sijoitus oli viimeinen. Ero ei voi selittyä tutkimusvuodella, koska tämän tutkimuksen sijoitus vastaa paremmin vanhinta tutkimusta. Ero voi selittyä tutkimuksissa käytettyjen lähteiden tutkimusmenetelmillä. Tässä tutkimuksessa on käytetty pääasiassa kyselyitä ja haastatteluita sisältäviä lähteitä, joita aiemmassa tutkimuksessa ei käytetty juuri ollenkaan.

Aiemmassa tutkimuksessa käyttöönottoon ja DevOpsiin liittyviä haasteita löytyi sijoilta 2 (yhteensä 8 haastetta) (Alshuqayran ja muut, 2016) ja 1 (yhteensä 9 haastetta) (Söylemez ja muut, 2022). Lisäksi niitä löytyi myös yhdestä aiemmasta tutkimuksesta, joissa haasteille ei määritelty järjestystä. Sijoitus oli melko korkealla sekä vanhimmassa (2) että uusimmassa (1) aiemmassa tutkimuksessa, kuten myös tässä tutkimuksessa (jaettu 3). Käyttöönotto ja DevOps on siten edelleen mikropalveluiden kehittäjille suurimpia haasteita.

Aiemmassa tutkimuksessa monitorointiin, logitukseen ja debuggaukseen liittyviä haasteita löytyi sijoilta 6 ja 8 (yhteensä 8 haastetta) (Alshuqayran ja muut, 2016) ja 2 (yhteensä 9 haastetta) (Söylemez ja muut, 2022). Lisäksi niitä löytyi myös yhdestä aiemmasta tutkimuksesta, joissa haasteille ei määritelty järjestystä. Sijoitus on noussut vanhimmasta tutkimuksesta (8) melko korkealle uusimpaan (2). Tässä tutkimuksessa sijoitus oli jaettu 3, joten se noudattelee melko hyvin uusimman aiemman tutkimuksen tuloksia. Ero voi johtua siitä, että monitoroinnin, loggauksen ja debuggauksen merkitys on kasvanut vuosien kuluessa. Voi myös olla, että ajan kuluessa mikropalveluista on tullut monimutkai-

sempia kuin vanhimman aiemman tutkimuksen aikaan ja niiden monitorointi, logitus ja debuggaus on vaikeutunut.

Aiemmassa tutkimuksessa tiedonhallintaan liittyviä haasteita löytyi sijalta 8 (yhteensä 9 haastetta) (Söylemez ja muut, 2022). Lisäksi niitä löytyi myös kahdesta aiemmasta tutkimuksesta, joissa haasteille ei määritelty järjestystä. Sijoitus eroaa melko paljon tämän tutkimuksen sijoituksesta (jaettu 3). Ero voi selittyä tutkimuksissa käytettyjen lähteiden tutkimusmenetelmillä. Tässä tutkimuksessa on käytetty pääasiassa kyselyitä ja haastatteluita sisältäviä lähteitä, joita aiemmassa tutkimuksessa ei käytetty juuri ollenkaan.

Tämän tutkimuksen kolme tärkeintä mikropalveluiden migraatioon liittyvää haastetta olivat järjestyksessä tärkeimmästä vähiten tärkeään organisaatio ja ihmiset, vanhan järjestelmän haasteet ja uuden järjestelmän haasteet. Migraatioon liittyviä haasteita löytyi yhteensä 5. Lista haasteista tärkeysjärjestyksessä löytyy taulukosta 9. Aiemmassa tutkimuksessa migraatioon liittyviä haasteita löytyi jokaisesta kategoriasta. Haasteita ei kuitenkaan oltu järjestelty tärkeysjärjestykseen.

Vaikka suurin osa löydetyistä haasteista olikin teknisiä, tutkimuksista nousi esiin myös paljon organisaatioon ja ihmisiin liittyviä haasteita. Aiemmassa tutkimuksessa ihmisiin ja organisaatioon liittyviä haasteita löytyi vain yhdestä tutkimuksesta. Ero voi selittyä tutkimuksissa käytettyjen lähteiden tutkimusmenetelmillä. Tässä tutkimuksessa on käytetty pääasiassa kyselyitä ja haastatteluita sisältäviä lähteitä, joita aiemmassa tutkimuksessa ei käytetty juuri ollenkaan. Haastatteluissa ja kyselyissä on ollut pääasiassa osallistujina henkilöitä, jotka työskentelevät mikropalveluiden kehityksen parissa oikeissa organisaatioissa, joten heidän vastauksistaan myös näkyy se, että ohjelmistokehitys ei ole pelkästään tekninen suoritus vaan siihen liittyy myös organisatorinen ja sosiaalinen ulottuvuus.

Tutkimuksen lähdemateriaalista löytyi myös paljon ehdotuksia näitä haasteita ratkaisemaan. Osaan haasteista ei kuitenkaan välttämättä löytynyt kovin kattavia ratkaisuja. Käyttöönotto ja DevOps on haastekategoriana jo hieman erikoinen: DevOps auttaa ratkaisemaan käyttöönoton ongelmia. DevOps vaatii kuitenkin erityistä osaamista, jotta organisaatio voi lisätä joko kouluttamalla vanhoja työntekijöitä tai palkkaamalla uusia. DevOpsin suhteen yksi ongelmista on ollut tutkimuksessa 11 mainittu osaamisen puute ja vaikeus rekrytoida DevOps-osaajia, jotka luokiteltiin kuitenkin tässä tutkimuksessa ihmisiin ja organisaatioon liittyviksi haasteiksi (Niedermaier ja muut, 2019). Tutkimus 9 löysi hyviä ratkaisuja migraation organisaatioon ja ihmisiin liittyviin haasteisiin, joita voi hyödyntää myös vaikka ei olisikaan kyse migraatiosta.

6 Yhteenveto

Tässä tutkielmassa esiteltiin ohjelmistoarkkitehtuuria ja erityisesti mikropalveluarkkitehtuuria. Mikropalveluarkkitehtuuri on jaettuun arkkitehtuuriin kuuluva arkkitehtuurinen tyyli, jossa sovelluksen toimiala on jaettu pieniin palveluihin. Sen etuja ovat muun muassa hyvä skaalautuvuus ja elastisuus, helppo ylläpidettävyys, korkea saatavuus ja kehitystyön rinnakkaisuus. Mikropalvelut saavuttavat nämä edut hyödyntämällä tehokkaasti erilaisia teknologioita kuten pilvilaskentaa, DevOpsia, kontitusta, JSON-verkkovaltuuksia ja REST-arkkitehtuuria.

Tutkielmassa tehtiin myös systemaattinen kirjallisuuskatsaus, jolla selvitettiin mikropalveluiden kehittäjien kokemia haasteita ja ratkaisuja niihin. Kirjallisuuskatsauksen tuloksena havaittiin, että suurin osa mikropalveluiden kehitykseen liittyvistä haasteista on teknisiä haasteita, mutta aiemmista tutkimuksista poiketen kirjallisuuskatsauksessa löydettiin myös organisaatioon ja ihmisiin liittyviä haasteita. Kun tarkasteltiin erityisesti mikropalveluiden migraatioon liittyviä haasteita, organisaatioon ja ihmisiin liittyvät haasteet nousivat yleisimmäksi. Lähes jokaiseen kirjallisuuskatsauksessa löydettyyn haasteeseen löydettiin myös ratkaisuehdotus kirjallisuuskatsauksen lähdemateriaalista. Kirjallisuuskatsauksen tulokset erosivat jonkin verran aiemmasta tutkimuksesta. Osa eroista voi selittyä tutkimusvuodella, osa taas tutkimuksen lähdemateriaalin tutkimusmenetelmillä.

Tutkielmassa esitelty kirjallisuus keskittyi pääasiassa mikropalveluihin liittyviin teknisiin haasteisiin. Siitä selvisi kuitenkin, että mikropalveluiden kehittäjät kokevat myös ihmisiin ja organisaatioon liittyviä haasteita. Nämä haasteet koettiin osittain jopa tärkeämmiksi kuin tekniset haasteet (Fritzsch ja muut, 2019). *"30 vuoden työkokemuksella voin vakuuttaa, että teknologia saadaan kyllä kohdilleen, mutta täytyy saada myös ihmiset mukaan"*, totesi eräs haastateltavista tutkimuksessa 8 (Fritzsch ja muut, 2019). Sopiva jatkotutkimuksen aihe olisikin tutkia organisaatioiden kokemia haasteita organisaatiorakenteen muutoksissa ja ihmisten osaamisessa ja asenteissa sekä ratkaisuja niihin kun organisaatio siirtyy kehittämään mikropalveluja. Tutkimusmenetelmänä voisi olla haastattelut ja kyselyt joihin osallistuisi kehittäjien lisäksi myös organisaation ylempää johtoa, henkilöstöhallintoa ja rekrytoijia, jotta saataisiin kattava näkemys organisaation ja ihmisten kokemista haasteista mikropalveluihin siirryttäessä.

Lähdeluettelo

Alshuqayran, N., Ali, N., & Evans, R. (2016). *A Systematic Mapping Study in Microservice Architecture*. In Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 44–51. https://doi.org/10.1109/SOCA.2016.15

Asad, A. (2022). *14 Open Source and Managed API Gateway for Modern Applications*. Haettu 08.10.2022 osoitteesta https://geekflare.com/api-gateway/.

Bogner, J., Fritzsch, J., Wagner, S., & Zimmermann, A. (2021). *Industry practices and challenges for the evolvability assurance of microservices: An interview study and systematic grey literature review*. Empirical Software Engineering: an International Journal, 26(5). https://doi.org/10.1007/s10664-021-09999-9

Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T., & Mazzara, M. (2018). *From Monolithic to Microservices: An Experience Report from the Banking Domain*. IEEE Software, 35(3), 50–55. https://doi.org/10.1109/MS.2018.2141026

Containerization. (2021). IBM. Haettu 15.10.2022 osoitteesta https://www.ibm.com/cloud/learn/containerization.

da Silva, H., de F. Carneiro, G., & Monteiro, M. P. (2019). *An Experience Report from the Migration of Legacy Software Systems to Microservice Based Architecture*. In Proceedings of the 16th International Conference on Information Technology-New Generations (ITNG 2019), 800, 183–189. https://doi.org/10.1007/978-3-030-14070-0_26

Deprecation. Techopedia. Haettu 30.10.2022 osoitteesta https://www.techopedia.com/definition/28957/deprecation.

de Toledo, S. Martini, A., & Sjøberg, D. I. K. (2021). *Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study*. The Journal of Systems and Software, 177, 110968–. https://doi.org/10.1016/j.jss.2021.110968

de Toledo, S., Martini, A., Nguyen, P. H., & Sjoberg, D. I. K. (2022). Accumulation and Prioritization of Architectural Debt in Three Companies Migrating to Microservices. IEEE Access, 10, 37422–37445. https://doi.org/10.1109/ACCESS.2022.3158648

Di Francesco, P., Lago, P., & Malavolta, I. (2018). *Migrating Towards Microservice Architectures: An Industrial Survey*. In Proceedings of the 2018 IEEE International Conference on Software Architecture (ICSA), 29–2909. https://doi.org/10.1109/ICSA.2018.00012

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). *DevOps*. IEEE Software, 33(3), 94–100. https://doi.org/10.1109/MS.2016.68

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Fielding, R. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine.

Fowler, M. & Lewis, J. (2014). *Microservices*. Haettu 10.09.2022 osoitteesta https://www.martinfowler.com/articles/microservices.html.

Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). *Microservices Migration in Industry: Intentions, Strategies, and Challenges*. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 481–490. https://doi.org/10.1109/ICSME.2019.00081

Ghofrani, J., & Bozorgmehr, A. (2019). *Migration to Microservices: Barriers and Solutions*. In Applied Informatics (Vol. 1051, pp. 269–281). Springer International Publishing. https://doi.org/10.1007/978-3-030-32475-9_20

Gorton, I. (2011). *Essential Software Architecture* (2nd ed. 2011.). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-19176-3

Haselbock, S., Weinreich, R., & Buchgeher, G. (2018). *An Expert Interview Study on Areas of Microservice Design*. In Proceedings of the 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), 137–144. https://doi.org/10.1109/SOCA.2018.00028

Hassan, S., Bahsoon, R., & Kazman, R. (2020). *Microservice transition and its granularity problem: A systematic mapping study*. Software, Practice & Experience, 50(9), 1651–1681. https://doi.org/10.1002/spe.2869

Henderson, C. (2021). *What Is An Enterprise Application? Definition, Examples and Trends*. Haettu 10.09.2022 osoitteesta https://anyconnector.com/enterprise-application.html.

ISO/IEC/IEEE Systems and software engineering – Architecture description (pp. 1–46). (2011). IEEE. https://doi.org/10.1109/IEEESTD.2011.6129467

Introduction to JSON Web Tokens. auth0. Haettu 15.10.2022 osoitteesta https://jwt.io/introduction.

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2018). *Challenges When Moving from Monolith to Microservice Architecture*. In Proceedings of the Current Trends in Web Engineering (ICWE 2017), 10544, 32–47. https://doi.org/10.1007/978-3-319-74433-9_3

Kitchenham, B. (2004). *Procedures for Performing Systematic Reviews*. Keele University.

Knoche, H. & Hasselbring, W. (2019). *Drivers and Barriers for Microservice Adoption - A Survey among Professionals in Germany*. Enterprise Modelling and Information Systems Architectures, 14. https://doi.org/10.18417/emisa.14.1

Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). *Data management in microservices: State of the practice, challenges, and research directions*. In Proceedings of the VLDB Endowment, 14(13), 3348–3361. https://doi.org/10.14778/3484224.3484232

Li, B., Peng, X., Xiang, Q., Wang, H., Xie, T., Sun, J., & Liu, X. (2022). *Enjoy your observability: an industrial survey of microservice tracing and analysis*. Empirical Software Engineering: an International Journal, 27(1), 25–25. https://doi.org/10.1007/s10664-021-10063-9

Luz, W., Agilar, E., de Oliveira, M., de Melo, C., Pinto, G., & Bonifácio, R. (2018). *An experience report on the adoption of microservices in three Brazilian government institutions*. ACM International Conference Proceeding Series, 32–41. https://doi.org/10.1145/3266237.3266262

Montesi, F. & Weber, J. (2016). *Circuit Breakers, Discovery, and API Gateways in Microservices*. Cornell University, arXiv.org. https://doi.org/10.48550/arXiv.1609.05830

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Incorporated.

Newman, S. (2015). *Pattern: Backends For Frontends*. Haettu 09.10.2022 osoitteesta https://samnewman.io/patterns/architectural/bff/.

Niedermaier, S., Koetter, F., Freymann, A., & Wagner, S. (2019). *On Observability and Monitoring of Distributed Systems – An Industry Interview Study*. Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11895, 36–52. https://doi.org/10.1007/978-3-030-33702-5_3

Richards, M. & Ford, N. (2020). *Fundamentals of software architecture: an engineering approach* (First edition.). O'Reilly.

Richarson, C. *Pattern: Access token*. Haettu 09.10.2022a osoitteesta https://microservices.io/patterns/security/access-token.html.

Richarson, C. *Pattern: API composition*. Haettu 09.10.2022b osoitteesta https://microservices.io/patterns/data/api-composition.html.

Richarson, C. *Pattern: Database per service*. Haettu 09.10.2022c osoitteesta https://microservices.io/patterns/data/database-per-service.html.

Richarson, C. *Pattern: Messaging*. Haettu 09.10.2022d osoitteesta https://microservices.io/patterns/communication-style/messaging.html.

Richarson, C. *Pattern: Microservice Architecture*. Haettu 19.10.2022e osoitteesta https://microservices.io/patterns/microservices.html.

Rotem-Gal-Oz, A. *Fallacies of Distributed Computing Explained*. Haettu 16.10.2022 osoitteesta https://arnon.me/wp-content/uploads/Files/fallacies.pdf.

Soldani, J., Tamburri, D. A., & Van Den Heuvel, W.-J. (2018). *The pains and gains of microservices: A Systematic grey literature review*. The Journal of Systems and Software, 146, 215–232. https://doi.org/10.1016/j.jss.2018.09.082

Söylemez, M., Tekinerdogan, B., & Tarhan, A. K. (2022). *Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review*. Applied Sciences, 12(11), 5507–. https://doi.org/10.3390/app12115507

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). *Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation*. IEEE Cloud Computing, 4(5), 22–32. https://doi.org/10.1109/MCC.2017.4250931

Taibi, D., Lenarduzzi, V., Pahl, C., & Janes, A. (2017). *Microservices in agile software development: A workshop-based study into issues, advantages, and disadvantages*. In Proceedings of the ACM International Conference Proceeding Series, 129907. https://doi.org/10.1145/3120459.3120483

Wang, Y., Kadiyala, H., & Rubin, J. (2021). *Promises and challenges of microservices: an exploratory study*. Empirical Software Engineering: an International Journal, 26(4). https://doi.org/10.1007/s10664-020-09910-y

Waseem, M., Liang, P., & Shahin, M. (2020). *A Systematic Mapping Study on Microservices Architecture in DevOps*. The Journal of Systems and Software, 170, 110798–. https://doi.org/10.1016/j.jss.2020.110798

Waseem, M., Liang, P., Shahin, M., Ahmad, A., & Nassab, A. R. (2021a). *On the nature of issues in five open source microservices systems: An empirical study*. In Proceedings of the ACM International Conference Proceeding Series, 201–210. https://doi.org/10.1145/3463274.3463337

Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021b). *Design, monitoring, and testing of microservices systems: The practitioners' perspective*. The Journal of Systems and Software, 182, 111061–. https://doi.org/10.1016/j.jss.2021.111061

What is Cloud Scalability?. VMware. Haettu 19.10.2022 osoitteesta https://www.vmware.com/topics/glossary/content/cloud-scalability.html.

Zhang, H., Li, S., Jia, Z., Zhong, C., & Zhang, C. (2019). *Microservice Architecture in Reality: An Industrial Inquiry*. In Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA), 51–60. https://doi.org/10.1109/ICSA.2019.00014

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., & Ding, D. (2021). *Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study*. IEEE Transactions on Software Engineering, 47(2), 243–260. https://doi.org/10.1109/TSE.2018.2887384

Zhou, X., Huang, H., Zhang, H., Huang, X., Shao, D., & Zhong, C. (2022a). *A Cross-Company Ethnographic Study on Software Teams for DevOps and Microservices: Organization, Benefits, and Issues*. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 1–10. https://doi.org/10.1145/3510457.3513054

Zhou, X., Li, S., Cao, L., Zhang, H., Jia, Z., Zhong, C., Shan, Z. & Babar, M. A. (2022b). *Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry*. Journal of Systems and Software. 111521. https://doi.org/10.1016/j.jss.2022.111521