Tampereen yliopisto

Marjaana Laine

WEB-SOVELLUS MIKROPALVELU-ARKKITEHTUURISSA

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Toukokuu 2021

TIIVISTELMÄ

Marjaana Laine: Web-sovellus mikropalveluarkkitehtuurissa
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2021

Nykypäivän ohjelmistokehityksen tavoitteena on saada uudet liiketoiminnan ideat, muutokset ja korjaukset nopeasti ja luotettavasti kehityksestä käyttöön tuotantoon, jossa ne tuovat arvoa käyttäjille ja liiketoiminnalle. Jatkuva kehitys ja palveluiden siirtyminen verkkoon ja pilviympäristöihin ovat johtaneet mikropalveluiden ja konttiteknologian käyttöön sovellusten skaalautuvuuden parantamiseksi ja käyttöönoton nopeuttamiseksi ja joustavoittamiseksi. Kerrosarkkitehtuurin soveltaminen voi johtaa suuriin monoliittisiin ohjelmistoihin, joita voi myöhemmin olla vaikea laajentaa ja skaalata. Mikropalveluarkkitehtuurissa hajotetaan ohjelmisto pieniin, löyhästi kytkettyihin palveluihin, jotka kommunikoivat keskenään verkon yli palveluihin toteutettujen rajapintojen avulla.

Tutkielman aiheena on web-sovellus mikropalveluarkkitehtuurissa, joten aluksi tutustutaan web-arkkitehtuuriin. Sen jälkeen tarkastellaan mikropalveluarkkitehtuuria tukevia DevOps-kehitysmenetelmiä. Tutkielmassa tutustutaan kirjallisuuskatsauksen kautta mikropalveluarkkitehtuuriin ja sen keskeisiin laatutekijöihin ja niiden hallintaa tukevien taktiikoiden käyttöön hajautettujen ohjelmistojen toteuttamisessa. Mikropalveluarkkitehtuurin laatutekijöiden jälkeen tarkastellaan case-esimerkkinä web-sovelluksen kehittämistä mikropalveluarkkitehtuurissa osana matkapuhelinverkon analysointi- ja valvontaohjelmistoa. Tutkielman tutkimuskysymyksenä on: Mitkä ovat onnistuneen mikropalveluarkkitehtuurin laatutekijät?

Mikropalveluarkkitehtuurin keskeisimmiksi laatutekijöiksi tunnistettiin skaalautuvuus, suorituskyky, saatavuus, seurattavuus, turvallisuus ja testattavuus. Laatutekijöillä ja niiden hallintaa tukevilla taktiikoilla kuten konttiteknologialla ja automatisoidulla käyttöönotolla ja testauksella vastataan hajautetun ohjelmiston kompleksisuuden haasteisiin ja voidaan parantaa ohjelmiston muunneltavuutta, skaalautuvuutta ja nopeaa käyttöönottoa. Mikropalveluarkkitehtuuri ei ole aina sopivin ratkaisu ohjelmistojen toteuttamiseen. Sen soveltamista tulee arvioida ohjelmistolle asetettavien vaatimusten pohjalta eikä mikropalveluiden kautta saavutettavien mahdollisten etujen kannalta. Mikropalveluiden edut eivät ole ilmaisia. Mikropalveluarkkitehtuuri edellyttää suunnittelua, ohjelmiston osien toimivaa rajausta keskenään kommunikoiviin palveluihin, hajautettujen palveluiden hallintaa ja seurantaa sekä ohjelmiston toteuttamiseen osallistuvien toimivaa yhteistyötä.

Avainsanat: Asiakas-palvelin-arkkitehtuuri, DevOps, Docker, Konttiteknologia, Kubernetes, Mikropalvelu, Mikropalveluarkkitehtuuri, Web-arkkitehtuuri, Web-sovellus

SISÄLLYS

1 Johdanto 1

2 Web-arkkitehtuuri 3

2.1 Ohjelmistoarkkitehtuuri 3

2.2 Asiakas-palvelin-arkkitehtuuri 4

2.3 REST-arkkitehtuuri 5

2.4 Web-teknologiat 5

2.5 Palvelukeskeinen arkkitehtuuri 7

3 DevOps 8

3.1 DevOps 8

3.2 Jatkuva integraatio ja toimitus 9

4 Mikropalveluarkkitehtuuri 11

4.1 Laatutekijät 11

4.2 Skaalautuvuus 15

4.3 Suorituskyky 17

4.3.1 Konttiteknologia 19

4.3.2 Konttien orkestrointi 21

4.4 Saatavuus 22

4.5 Seurattavuus 24

4.6 Turvallisuus 25

4.7 Testattavuus 26

4.8 Yhteenveto 27

5 Kohdesovellus mikropalveluarkkitehtuurissa 29

5.1 Analysointi- ja valvontaohjelmisto 29

5.2 Hallintasovellus 30

5.2.1 Toiminnallisuus 30

5.2.2 Teknologiat 31

5.2.3 Turvallisuus 32

5.2.4 Toteutus 33

5.3 Tukipalvelut 34

5.3.1 Verkkoliikenteen välitys 34

5.3.2 Käyttäjähallinta 35

5.3.3 Metriikkaseuranta 36

5.3.4 Palvelurekisteri 36

5.4 Docker ja Kubernetes 37

5.5 Kommunikaatiokaavio 38
5.6 Arviointi 39
6 Yhteenveto 43
7 Viiteluettelo 44

1 Johdanto

Nykypäivän ohjelmistokehityksen tavoitteena on saada uudet liiketoiminnan ideat, muutokset ja korjaukset nopeasti ja luotettavasti kehityksestä käyttöön tuotantoympäristöön, jossa ne tuovat arvoa käyttäjälle ja liiketoiminnalle. Ohjelmistot ovat jatkuvassa muutoksessa ja web-teknologiat ovat hajautettujen palveluiden, rajapintojen määrän kasvun ja pilviympäristöjen yleistymisen myötä olennaisia. Hajautetuissa ohjelmistoissa on enemmän liikkuvia osia ja osien välillä tapahtuva vuorovaikutus on monipuolisempaa ja suurempaa kuin perinteisissä ohjelmistoissa. Jatkuva kehitys ja palveluiden siirtyminen verkkoon ja pilviympäristöihin ovat johtaneet mikropalveluiden ja konttiteknologian käyttöön sovellusten skaalautuvuuden parantamiseksi ja käyttöönoton nopeuttamiseksi. (Wan et al., 2018) Mikropalvelut ja kontit tarjoavat ratkaisuja sovellusten on-demand-skaalaukseen, resurssien tehokkaaseen kohdentamiseen sekä itsenäiseen kehitykseen ja käyttöönottoon.

Tutkielmassa tutustutaan kirjallisuuskatsauksen kautta mikropalveluarkkitehtuuriin ja sen olennaisiin laatutekijöihin ja niiden hallintaa tukevien taktiikoiden käyttämiseen hajautettujen ohjelmistojen toteuttamisessa. Teemana on web-sovelluksen kehittäminen mikropalveluarkkitehtuurissa, joten aluksi tutustutaan web-arkkitehtuuriin. Tutkielmassa tarkastellaan myös mikropalveluarkkitehtuurin käyttöä tukevia DevOps-kehitysmenetelmiä. Mikropalveluarkkitehtuurin laatutekijöiden jälkeen tarkastellaan case-esimerkkinä web-sovelluksen kehittämistä mikropalveluarkkitehtuurissa osana matkapuhelinverkon analysointi- ja valvontaohjelmistoa. Tutkielman tutkimuskysymyksenä on: Mitkä ovat onnistuneen mikropalveluarkkitehtuurin laatutekijät?

Kirjallisuuskatsauksen lähteitä etsittiin käyttämällä hakusanoja ja niiden yhdistelmiä: microservice, microservice architecture, container, Docker, Kubernetes, DevOps, continuous development, continuous integration, web architecture ja web application. Tieteellisiä lähteitä kerättiin tietokannoista ja hakupalveluista: Andor, Google Scholar ja ACM Digital Library. Mikropalveluarkkitehtuurin laatutekijöiden lähteinä on käytetty pääosin tieteellisiä julkaisuja. Erilaisten teknologioiden yhteydessä viitataan myös teknologioiden dokumentaatioihin ja verkkosivuihin. Englanninkielisissä lähteissä käytettyä termejä on suomennettu.

Luvussa kaksi käsitellään web-arkkitehtuuria ja siihen kuuluvia käsitteitä ja malleja web-sovelluksen kehittämisessä. Web-sovellukset on perinteisesti toteutettu asiakas-palvelin-mallin mukaisesti. Luvussa kolme tutustutaan DevOps-kehitysmenetelmiin, jotka ovat olennaisia toimivassa mikropalveluarkkitehtuurissa. Luku neljä keskittyy mikropalveluarkkitehtuurin laatutekijöihin ja taktiikoihin. Luvussa viisi esitetään case-esimerkki

mikropalveluarkkitehtuurissa kehitettävästä web-sovelluksesta. Luku kuusi sisältää yhteenvedon. Luku seitsemän toimii viiteluettelona.

2 Web-arkkitehtuuri

Web-arkkitehtuuri (engl. World Wide Web, WWW) (Fielding & Taylor, 2002, 115) on suunniteltu vastaamaan Internet-laajuisen hajautetun hypermediasovelluksen tarpeita. Nykyaikainen web-arkkitehtuuri korostaa komponenttien vuorovaikutuksen skaalautuvuutta, rajapintojen yleisyyttä, komponenttien itsenäistä käyttöönottoa, välittäjäkomponenttien käyttöä vähentämään vuorovaikutuksen viivettä (engl. latency), turvallisuutta ja vanhojen legacy-järjestelmien kapselointia. Fielding ja Taylor (2002, 147) toteavat, että web-arkkitehtuurin periaatteiden ymmärtäminen voi auttaa selittämään sen menestystä ja voi johtaa parannuksiin myös muissa hajautetuissa sovelluksissa. Verkkopohjaisissa sovelluksissa järjestelmän suorituskykyä hallitsee verkkoviestintä ja komponenttien väliset tiedonsiirrot. REST-arkkitehtuuri kehitettiin näihin tarpeisiin.

2.1 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuuri (Fielding & Taylor, 2002, 116) määrittää ohjelmiston tai järjestelmän rakenteen eli mistä komponenteista se koostuu, komponenttien vastuut, miten komponentit kommunikoivat keskenään sekä mitä rajapintoja ja protokollia viestinnässä käytetään. Ohjelmistoarkkitehtuurissa (Fielding, 2000) suunnitellaan miten parhaiten jaetaan järjestelmä osiin, miten komponentit tunnistavat toisensa ja kommunikoivat toistensa kanssa, miten tietoa välitetään komponenttien välisessä viestinnässä ja kuinka järjestelmän osat voivat kehittyä itsenäisesti sekä kuinka edellä mainitut voidaan kuvata käyttämällä virallisia ja epävirallisia notaatioita.

Ohjelmistoarkkitehtuurisuunnittelu (Koskimies & Mikkonen, 2005, 75) on riippuvuuksien määrittelemistä, ja sen ymmärtäminen on riippuvuuksien ymmärtämistä. Ohjelmiston eri osien väliset riippuvuudet ja osien riippuvuudet ulkoisista tekijöistä vaikuttavat ratkaisevasti uudelleenkäytettävyyteen, ylläpidettävyyteen ja kehitystyön hajautettavuuteen, jotka ovat tärkeimpiä ohjelmistolle asetettavia vaatimuksia. Keskeinen pyrkimys arkkitehtuurissa on vähentää ja selkeyttää riippuvuuksia esimerkiksi rajapintojen avulla.

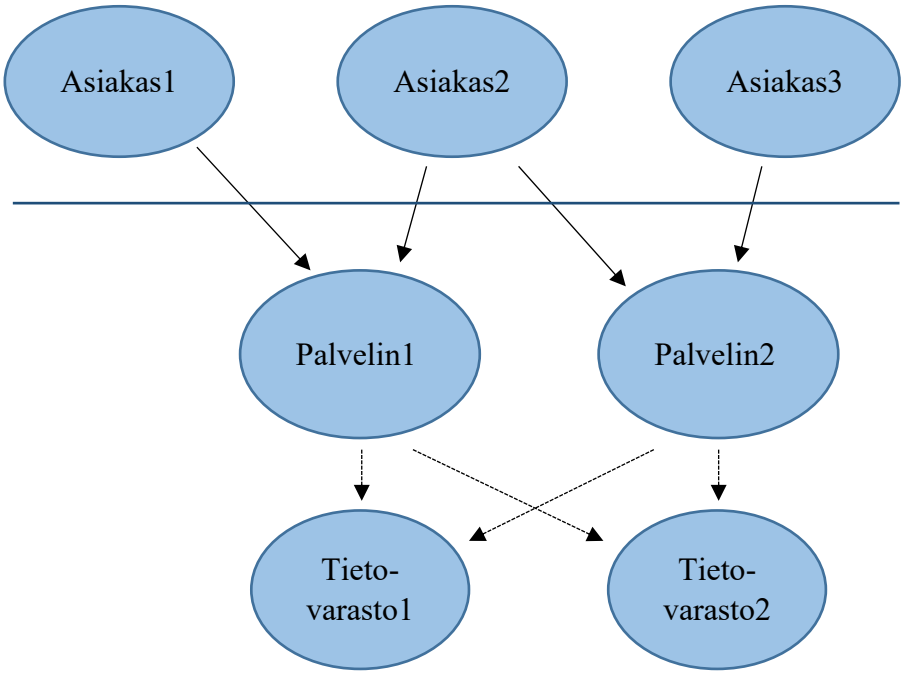
Ohjelmiston suunnittelumallit (Koskimies & Mikkonen, 2005, 102, 104) ovat tunnettujen ja käytännössä hyväksi havaittujen ratkaisujen kuvauksia yleisiin ohjelmistojen suunnittelua koskeviin ongelmiin. Ne tukevat arkkitehtuurisuunnittelua ja yksityiskohtaista suunnittelua. Suunnittelumalli voi parantaa ohjelmiston laatuominaisuutta esimerkiksi muunneltavuutta, siirrettävyyttä, ylläpidettävyyttä, uudelleenkäytettävyyttä ja suorituskykyä.

Ohjelmistoarkkitehtuurin toteuttamiseen on monia erilaisia tyylejä kuten kerrosarkkitehtuuri, asiakas-palvelin-arkkitehtuuri, palveluperusteinen arkkitehtuuri ja mikropalveluarkkitehtuuri. Kerrosarkkitehtuurissa (engl. layered architecture) (Fielding, 2000, 46)

jokainen kerros tarjoaa palveluita sen yläpuolella olevalle kerrokselle ja käyttää ainoastaan sen alapuolella olevan kerroksen palveluja. Web-sovellus voidaan rakentaa kerrostettuna asiakas-palvelin-sovelluksena, jossa ylimpänä kerroksena on käyttöliittymä ja tämän alapuolella sovelluslogiikasta vastaava kerros.

2.2 Asiakas-palvelin-arkkitehtuuri

Web-sovellus tai verkkosovellus (engl. web application) (Jazayeri, 2007) on web-selaimella käytettävä sovellus tietoverkossa tai Internetissä. Web-sovellukset on perinteisesti toteutettu asiakas-palvelin-arkkitehtuurin mukaisesti, jossa asiakkaat ja palvelimet viestivät keskenään verkon yli ja käyttävät kommunikointiin standardoituja protokollia. Palvelin (Fielding, 2000, 45-46, 78) tarjoaa joukon palveluita ja kuuntelee kyseisiä palveluita koskevia pyyntöjä. Asiakkaat lähettävät palvelimelle palvelun käyttämistä koskevan pyynnön, joka ei ole riippuvainen muista asiakkaista. Palvelin joko suorittaa pyynnön tai hylkää sen ja lähettää vastauksen takaisin asiakkaalle. Erottamalla käyttöliittymän toiminnallisuudet tietojen tallentamiseen liittyvistä toiminnallisuuksista parannetaan käyttöliittymän siirrettävyyttä käyttäjältä toiselle. Skaalautuvuutta parannetaan yksinkertaistamalla palvelinkomponentteja. Asiakas- ja palvelintoiminnallisuuksien erottaminen tukee komponenttien itsenäistä kehittymistä ja mahdollistaa hajautetut ohjelmistot.



Kuva 1. Asiakas-palvelin-arkkitehtuuri

Asiakas-palvelin-arkkitehtuurissa (Koskimies & Mikkonen, 2005, 137-138) kapseloidaan tietyn arkkitehtuuritason resurssin hallinta (palvelin) siten, että resurssin käyttäjät (asiakkaat) voivat pyytää tiettyä resurssiin liittyvää palvelua palvelimelta vapaasti riippumatta muista asiakkaista (Kuva 1). Asiakas-palvelin-arkkitehtuurin etuna on selkeä työnjako. Mallia pidetään hajautettuna järjestelmänä, koska tietyn resurssin hallinta kapseloidaan palvelimelle, joka huolehtii resurssin hallitusta käytöstä. Asiakas-palvelin-arkkitehtuuriin perustuvissa tietovarastoa hyödyntävissä hajautetuissa järjestelmissä palvelin hallitsee tietovaraston käyttöä, ja siihen pääsee käsiksi vain palvelimen kautta.

Asiakkaana toimivan web-selaimen ja palvelimen välissä voi olla verkkoliikennettä ohjaava välityspalvelin (engl. proxy server) esimerkiksi NGINX-palvelin (2021). Välityspalvelin katkaisee suoran yhteyden asiakkaan ja palvelimen välillä ja toimii palvelimena alkuperäiselle asiakkaalle ja asiakkaana alkuperäiselle palvelimelle.

2.3 REST-arkkitehtuuri

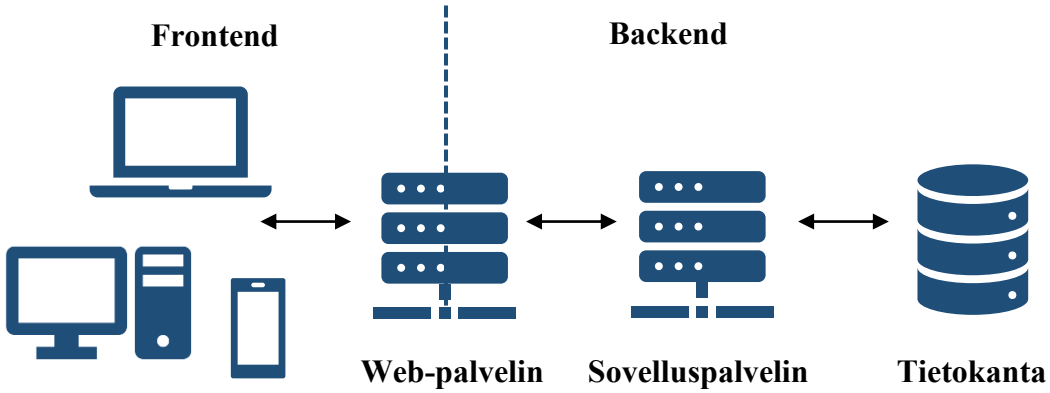
REST-arkkitehtuurityyli (engl. Representational State Transfer) on kehitetty abstraktiksi web-arkkitehtuurin malliksi ohjaamaan suunnittelua ja HTTP-protokollan (2021) (engl. Hypertext Transfer Protocol) ja URI-protokollan (2021) (engl. Uniform Resource Identifiers) käyttöä. REST on koordinoitu joukko arkkitehtonisia rajoituksia, joiden tavoitteena on minimoida verkkoviestintää ja viivettä ja samalla maksimoida komponenttien itsenäisyys ja skaalautuvuus. (Fielding & Taylor, 2002, 115-116; Fielding, 2000)

HTTP-protokollalla (2021) on keskeinen rooli web-arkkitehtuurin ominaisuuksien ja rajoitusten määrittämisessä. Web-arkkitehtuurissa asiakkaat ja palvelimet kommunikoivat käyttämällä HTTP-protokollan pyyntöjä ja vastauksia (Kuuskeri & Mikkonen, 2009, 648). REST-arkkitehtuurilla (Fielding & Taylor, 2002) on onnistuttu ohjaamaan web-arkkitehtuurin suunnittelua ja käyttöönottoa. Sen tarkoituksena on parantaa HTTP-protokollaan perustuvien web-rajapintojen (engl. Application Programming Interface, API) suorituskykyä (engl. performance), skaalautuvuutta (engl. scalability), yksinkertaisuutta (engl. simplicity), siirrettävyyttä (engl. portability) ja luotettavuutta (engl. reliability).

2.4 Web-teknologiat

Web 2.0 ja monipuoliset web-sovellusteknologiat (Kuuskeri & Mikkonen, 2009, 647) tarjoavat keinoja dynaamisten ja vuorovaikutteisten sovellusten rakentamiseen. Web-ohjelmistokehykset ja kirjastot helpottavat kehittämistä ja ylläpitoa. Web-sovellusten toiminnallisuutta ja sovelluslogiikkaa siirretään palvelinpuolen prosessoinnista asiakaspuolelle. Web-sovellukset tarjoavat käyttäjille dynaamisesti muodostettua sisältöä ja parannettua käyttäjäkokemusta. Perinteisesti datan prosessointi ja toiminnallisuus on suoritettu palvelimella ja tarjottu käyttäjälle käyttöliittymän HTML-sivun (2021) kautta.

Web-sovelluksen (Kuva 2) käyttöliittymät, selaimessa näkyvä osuus eli niin kutsuttu frontend, luodaan käyttämällä HTML-merkintäkieltä (2021), CSS-tyyliohjeita (2021) ja JavaScript-ohjelmointikieltä (ECMAScript, 2021). Taustapalvelin eli niin kutsuttu backend tarjoaa palveluita kuuntelemalla ja käsittelemällä sille tulevat pyynnöt. Palvelin voi olla vielä yhteydessä tietokantaan (Kuva 2). API-rajapintojen kautta välitettävien tietojen siirtämiseen voidaan käyttää HTTP-prokotollan avulla erilaisia tiedonsiirtomuotoja, joita ovat JSON (2021) (engl. JavaScript Object Notation), XML (2021) (engl. Extensible Markup Language) tai HTML (2021) (engl. Hypertext Markup Language).



Kuva 2. Web-sovellusympäristö

Perinteisesti web-sovellukset (Kuuskeri & Mikkonen, 2009, 648) on toteutettu käyttämällä synkronista lähestymistapaa, jossa käyttöliittymä lähettää palvelimelle pyynnön ja odottaa kunnes data saapuu. Kun data on vastaanotettu, koko näkymä luodaan uudelleen. Web-sovelluskehitystä on vienyt eteenpäin asynkronisten tekniikoiden soveltaminen. JavaScript-ohjelmointikieli on eniten käytetty teknologia asynkronisten käyttöliittymien kehittämisessä. JavaScriptin saumaton integrointi DOM-mallin (2021) (engl. Document Object Model) kanssa on tehnyt siitä suositun. DOM-malli on dokumenttioliomalli rakenteellisen dokumentin esimerkiksi HTML-sivun esittämiseen. Sen avulla voidaan HTML-sivun olioita hakea ja muokata JavaScriptin avulla. Single page -sovelluksissa (engl. single-page application, SPA) ensimmäisellä sivulatauksella ladatuilla HTML-, CSS- ja JavaScript-tiedostoilla luodaan sovelluksen näkymät, joita täydennetään asynkronisilla kutsuilla taustapalveluiden sovellusrajapintaan. Käyttäjäkokemus ei keskeydy, koska näkymiä päivitetään ilman koko sivun uudelleen latausta. (Kuuskeri & Mikkonen, 2009, 648)

Web-selaimella toimiva asiakaspuolen koodi on upotettu HTML-sivuille. Koodin objektit voivat viitata käyttöliittymäkokonaisuuksiin kuten ikkunoihin ja valikoihin. Koodi

voi reagoida käyttäjän tapahtumiin kuten hiiren liikkeisiin tai klikkauksiin ja tarjota vuorovaikutteisia näkymiä. Palvelinpuolen koodi hoitaa datan käsittelyä palvelimella ja on usein vuorovaikutuksessa tietokannan kanssa. Asiakkaan ja palvelimen koodien yhdistelmä tarjoaa hajautettua laskentaa, joka auttaa rakentamaan räätälöityjä käyttöliittymiä Web-sovelluksiin. (Jazayeri, 2007)

2.5 Palvelukeskeinen arkkitehtuuri

Palvelukeskeinen arkkitehtuuri (engl. Service Oriented Architecture, SOA) on suunnittelutapa (Newman, 2015, 8), jossa erilliset palvelurajapinnan tarjoamat palvelut tekevät yhteistyötä tarjotakseen toiminnallisuuksia. Palveluiden välinen viestintä tapahtuu verkon yli. Palvelulla tarkoitetaan tässä yhteydessä yleensä täysin erillistä käyttöjärjestelmäprosessia. Palvelukeskeinen arkkitehtuuri on lähestymistapa, joka vastaa suurten monoliittisten sovellusten haasteisiin. Sen avulla pyritään edistämään ohjelmistojen uudelleenkäyttöä ja palveluiden korvaamista toisella palvelulla.

Newman (2015, 8) toteaa, että palvelukeskeisen arkkitehtuurin ajatus erillisistä korvattavista palveluista on järkevä. Siltä kuitenkin puuttuu ymmärrys siitä, miten arkkitehtuuria toteutetaan toimivalla tavalla, ja miten jaetaan jotain suurta pieneksi. Palvelukeskeinen arkkitehtuuri ei onnistu tarjoamaan reaalimaailmaa mallintavia käytäntöjä palveluiden toteuttamiseen. Newman (2015, 9) vertaa palvelukeskeistä arkkitehtuuria mikropalveluarkkitehtuuriin, joka perustuu todellisen maailman käyttöön ja toteuttaa palvelukeskeistä ajattelua paremmin. Mikropalvelut voidaan nähdä erityisenä lähestymistapana toteuttaa palvelukeskeistä arkkitehtuuria samalla tavalla kuin on erilaisia lähestymistapoja ketterään ohjelmistokehitykseen.

Palvelukeskeiset arkkitehtuurit ja mikropalvelut (Brandon et al., 2020) määrittelevät kaksi tapaa suunnitella ohjelmisto, jonka tavoitteena on jakaa sovellus löyhästi kytkettyihin ja keskenään kommunikoiviin palveluihin. Tämä johtaa nopeampaan kehitykseen, joka mahdollistaa palveluiden itsenäisen kehityksen ja käyttöönoton sekä uusien ominaisuuksien ja nopeasti kehittyvien sovellusten jatkuvan toimituksen. Palvelukeskeisten arkkitehtuurien taustalla oleva kompleksisuus saattaa kuitenkin aiheuttaa haasteita järjestelmän ongelmien havaittavuuteen ja ylläpitoon. Sovelluksessa havaitun poikkeaman juurisyyn tunnistaminen voi olla vaikea ja aikaa vievä tehtävä, kun taustalla on lukuisia palveluita ja niiden välisiä yhteyksiä.

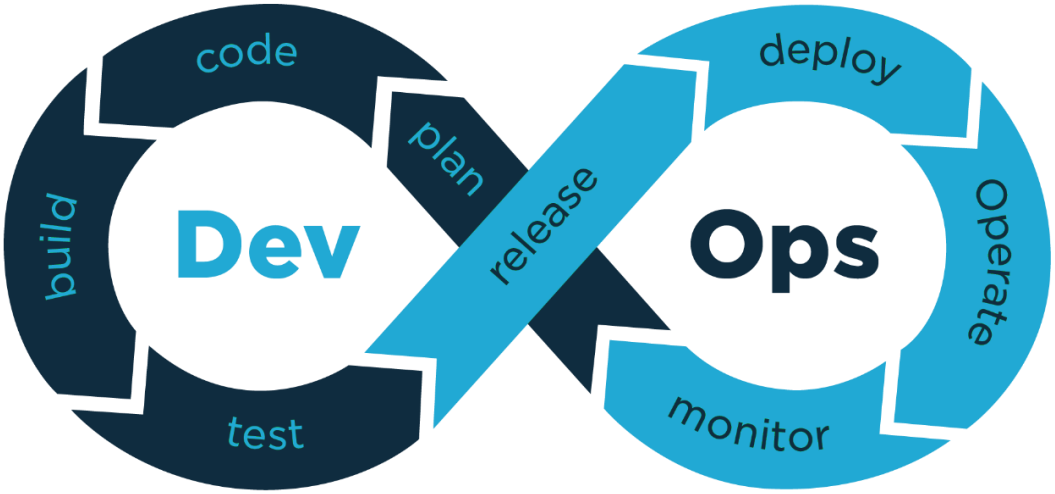
3 DevOps

DevOps kattaa joukon menetelmiä ja tekniikoita ohjelmistokehitysprosessin tehostamiseksi ja virtaviivaistamiseksi sekä kehityksen ja tuotannon integroimiseksi. DevOps-haasteisiin kuuluu pilvi-infrastruktuurien helppo käyttöönotto ja päivittäminen, koodin sujuva toimitus kehityksestä tuotantoon sekä infrastruktuurin automatisoitu skaalaus ihmisten mahdollisimman vähäisellä vaikutuksella. Toimiva DevOps on yksi pilvipohjaisten ohjelmistojen menestymisen avaimista. (Kang et al., 2016)

3.1 DevOps

DevOps (Jabbari, 2016) on yhdistelmä kehitystä (engl. development) ja tuotannon toimintoja (engl. operations), joita vaaditaan ohjelmiston toimittamiseksi asiakkaalle. DevOps on joukko käytäntöjä ohjelmistojen kehittämiseen, testaamiseen ja käyttöönottoon nopeasti ja luotettavasti sekä edistämään yhteistyötä kehittäjien, testaajien ja operatiivisista toiminnoista vastaavien henkilöiden välillä. Kaikki ovat osaltaan vastuussa kokonaisuudesta. DevOps-käytäntöjen tavoitteena on lyhentää muutoksiin ja muutosten tuotantoon siirtämiseen kuluvaa aikaa. Tavoitteena on saada uusi ominaisuus tai muutos nopeasti ja luotettavasti kehityksestä käyttöön tuotantoympäristöön, jossa se tuo arvoa käyttäjälle. DevOps on kulttuuri, joka yhdistelee uusia tai paranneltuja käytäntöjä, prosesseja, tiimirakenteita, vastuita ja työkaluja maksimoimaan organisaation kykyä toimittaa sovelluksia nopeasti. DevOps-lähestymistapa perustuu ketterään ohjelmistokehitykseen (engl. agile software development). (Waseem, 2020, 1-2; Humble & Farley, 2010, 28)

Ihannetapauksessa kaikki organisaation sisällä ovat linjassa tavoitteiden kanssa ja ihmiset työskentelevät yhdessä tavoitteiden saavuttamiseksi. Todellisuudessa kehityksen ja tuotannon välillä voi olla muuri, jonka yli julkaisun lähestyessä kehittäjät heittävät työnsä operatiivisille toiminnoille. DevOps murtaa muuria kehityksen ja tuotannon väliltä ja poistaa siiloja, jotka eristävät ihmisiä eri rooleihin. Säännöllinen kommunikointi kehityksen ja tuotannon välillä edistää toimivaa yhteistyötä. Yhteistyötä tukee myös järjestelmä, josta kaikki voivat nähdä ohjelmiston ja ympäristöjen tilan, versiot ja testitulokset. (Humble & Farley, 2010, 28)



Kuva 3. DevOps-silmukka (Oteyowo, 2018)

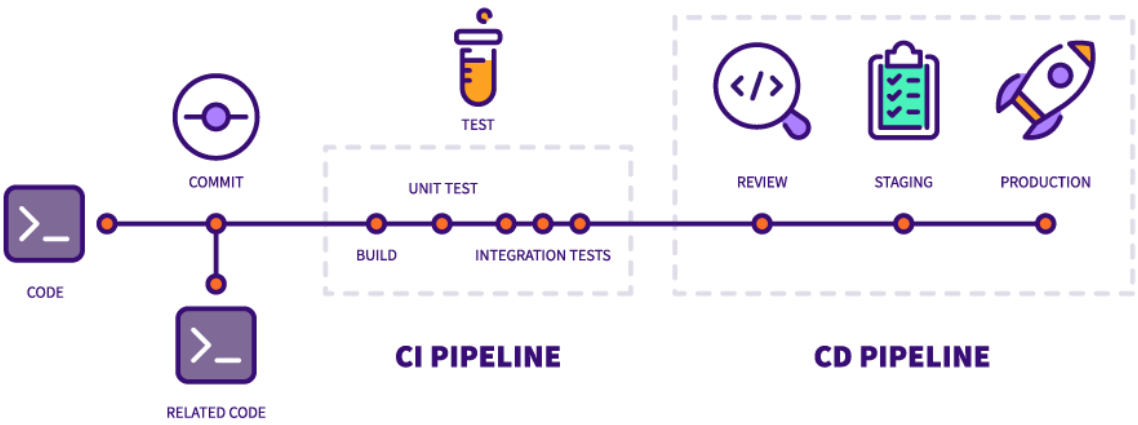
DevOps-toimintoja voidaan kuvata loputtomalla silmukalla (Kuva 3), johon kuuluu suunnittelu (engl. plan), ohjelmointi (engl. code), kokoaminen (engl. build), testaus (engl. test), julkaisu (engl. release), käyttöönotto (engl. deploy), tuotanto (engl. operate) ja seuranta (engl. monitor). DevOps-menetelmien avulla vastataan mikropalveluarkkitehtuurin haasteisiin, jotka voivat johtua hajautetuista sovelluksista, huonosti hallitusta koodi-infrastruktuurista tai puutteellisesta testiautomaatiosta. DevOps toimii prosessikehyksenä mikropalveluarkkitehtuurin kehittämisessä, käyttöönotossa ja hallinnassa. DevOps-menetelmien ja mikropalveluarkkitehtuurin rinnakkaiselo mahdollistaa uudelleen käytettävyyden, hajautettavuuden, automatisoinnin ja skaalautuvuuden. (Waseem, 2020, 1-2)

3.2 Jatkuva integraatio ja toimitus

DevOps-sovellusta voidaan toteuttaa jatkuvan integraation (engl. continuous integration, CI) ja jatkuvan toimituksen (engl. continuous delivery, CD) periaatteiden mukaisesti. DevOps-sovellus voidaan viedä astetta pidemmälle ottamalla käyttöön jatkuva käyttöönotto (engl. continuous deployment, CD). Jatkuvan integraation tavoitteena on pitää kaikki synkronoituna toistensa kanssa, mikä saavutetaan varmistamalla, että uusi koodi integroituu toimivasti olemassa olevan koodin kanssa ja läpäisee vaaditut testit. DevOps-sovellukselle voidaan rakentaa automatisoitu CI/CD-putki (engl. CI/CD pipeline), jonka vaiheiden läpi sovellus siirretään kehityksestä tuotantoon, ja jonka sisältämien testien avulla mahdolliset ongelmat voidaan havaita. Jatkuva integraatio on keskeinen käytäntö, jonka avulla tehdään nopeita muutoksia, ja jota ilman mikropalveluiden toteuttaminen on tuskallista. (Newman, 2015, 103-104)

Automatisoitu CI/CD-putki (Newman, 2015, 107, 246) auttaa prosessin etenemisen ja eri vaiheista suoriutumisen seuraamista ja saamaan käsityksen ohjelmiston laadusta. Kun muutos kulkee onnistuneesti kehityksestä käyttöönottoputken läpi tuotantoon, voidaan olla varmempia sen toimivuudesta. Jatkuvassa toimituksessa on olennaista jatkuvan palautteen saaminen tuotannon laadusta. Jatkuva käyttöönotto (Humble & Farley, 2010, 267) pakottaa tekemään asiat oikein, koska sitä ei voida toteuttaa ilman koko koonti-, käyttöönotto-, testaus- ja julkaisuprosessin automatisointia ja kattavia ja luotettavia automaattisia testejä.

CI/CD-putkessa ohjelmistoon tehtävät muutokset viedään jatkuvasti versionhallintaan ja ajetaan automatisoidun käyttöönotto- ja testausprosessin läpi, jolloin varmistetaan, että koodi on toimintakunnossa jokaisen tehdyn muutoksen jälkeen. Versionhallintaan voidaan luoda uusia toiminnallisuuksia, muutoksia ja korjauksia varten haaroja (engl. branches), jotka yhdistetään julkaisuhaaraan vasta, kun ne ovat valmiita julkaistavaksi. Versionhallintaan varastoidaan projektin sisältö: koodi, testit, tietokannan komentosarjat, luonti- ja käyttöönottokomennot sekä kaikki muu tarvittava ohjelmiston luomiseen, asentamiseen, suorittamiseen ja testaamiseen. Automaattisen testauksen kattavuus vaikuttaa ohjelmiston luotettavuuteen. CI/CD edellyttää hyvää konfiguraationhallintaa, versionhallintaa, automatisoitua kokoamista, hyväksymistestausta, julkaisuhallintaa ja tiimin sitoutumista työskentelymalliin. (Humble & Farley, 2010, 55-57, 345, 347, 406)



Kuva 4. CI/CD pipeline (Gitlab, 2021)

CI/CD -putki (Kuva 4) alkaa koodimuutoksen tekemisestä (engl. commit) versionhallintaan. Muutos versionhallinnassa käynnistää kokoamisen (engl. build), joka ajaa testit ja integroi muutoksen ohjelmaan. Onnistunut integraatio edellyttää hyväksyntää ja julkaisua tuotantoon ja käyttöön.

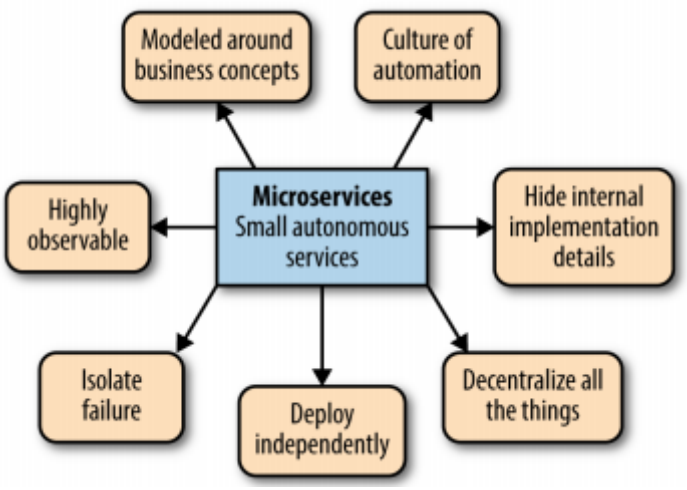
4 Mikropalveluarkkitehtuuri

Mikropalvelupohjaiset ohjelmistot ja järjestelmät mahdollistavat sovellusten itsenäisen kehittämisen, käyttöönoton ja skaalautuvuuden (Harms et al., 2017, 902). Mikropalveluarkkitehtuurin (Li et al., 2021) tarjoamien etujen innoittamina monet maailman johtavat Internet-yritykset, kuten Amazon, Netflix ja Ebay ovat ottaneet mikropalvelut käyttöön web-sovellusten arkkitehtuurityylinä. Pienet ja kohdennetut mikropalvelut itsenäisissä konteissa ovat tulleet yleiseksi tavaksi toteuttaa suuria järjestelmiä. Mitkä ovat onnistuneen mikropalveluarkkitehtuurin laatutekijät?

4.1 Laatutekijät

Mikropalveluarkkitehtuuri tarkoittaa yksinkertaisimmillaan sovelluksen tai järjestelmän hajottamista pienempiin osiin (Nadareishvili et al., 2016, 17). Mikropalvelut ovat pieniä, itsenäisiä palveluita, jotka toimivat yhdessä. Palvelut voidaan rajata liiketoiminnan rajojen mukaan, jolloin koodi toteuttaa selkeän toiminnallisuuden. Mitä pienempi palvelu on sitä enemmän maksimoidaan mikropalveluarkkitehtuurin edut, mutta myös liikkuvien osien määrän kasvusta aiheutuva kompleksisuus kasvaa. Kun kompleksisuuden hallinta toimii, voidaan tavoitella yhä pienempiä palveluita. Palvelut ovat itsenäisesti käyttöönotettavia ja voivat muuttua toisistaan riippumatta, joten niiden väliset kytkennät pidetään kevyinä. Palvelut paljastavat API-sovellusliittymät, joiden kautta palvelut kommunikoivat toistensa kanssa verkon välityksellä. (Newman, 2015, 2-3) Palveluiden itsenäinen käyttöönotto nopeuttaa myös ohjelmiston toimitusta (Nadareishvili et al., 2016, 17).

Newman (2015) on määrittänyt mikropalveluarkkitehtuurille seitsemän pääperiaatetta, jotka auttavat luomaan pieniä itsenäisiä palveluita, jotka toimivat hyvin yhdessä (Kuva 5). Periaatteet ovat mallintaminen liiketoimintojen ympärille (engl. modeled around business concepts), automaatiokulttuuri (engl. culture of automation), sisäisten toteutusten piilottaminen (engl. hide internal implementation details), asioiden hajauttaminen (engl. decentralize all things), itsenäinen käyttöönotto (deploy independently), virheiden eristäminen (engl. isolate failure) ja hyvä havaittavuus (engl. highly observable). Newman korostaa, että periaatteiden arvo syntyy niiden yhdistelmästä. Jos päätetään pudottaa yksi periaatteista pois, on tärkeää ymmärtää mistä luovutaan.



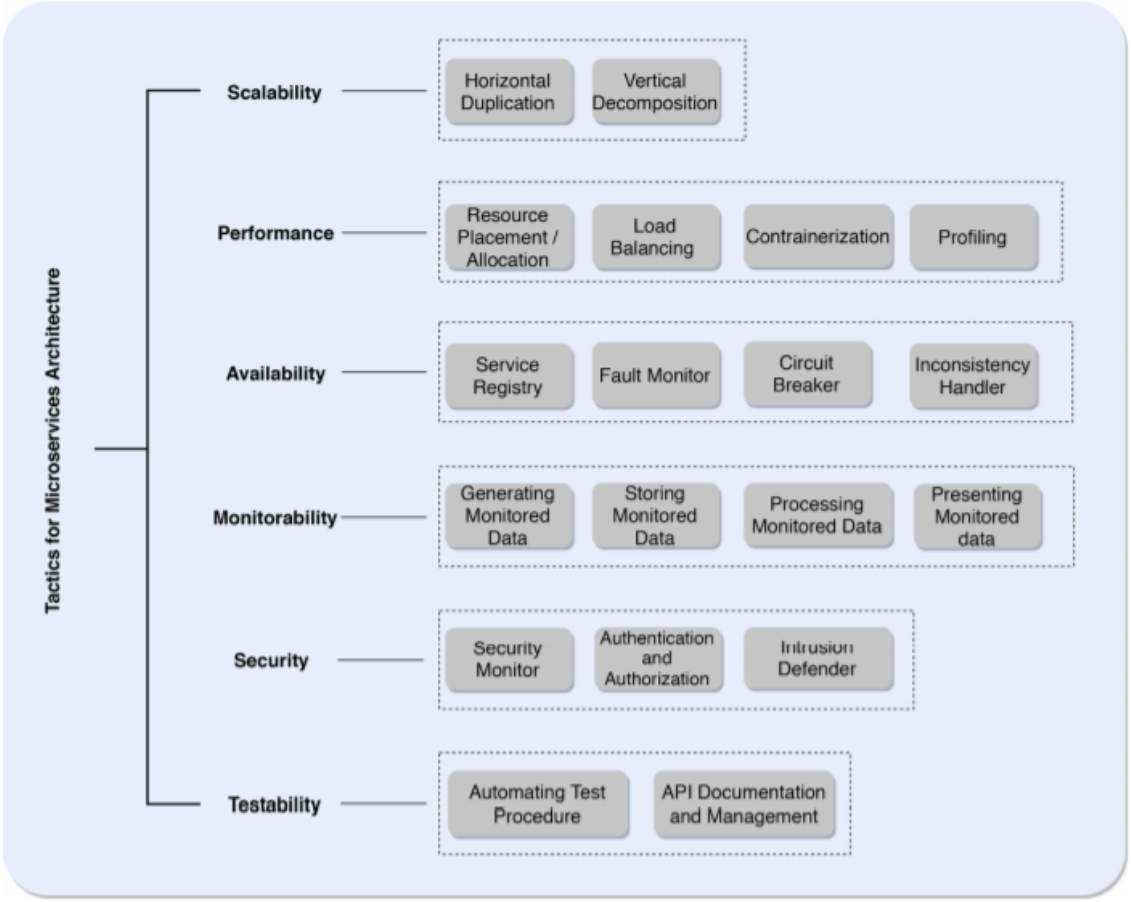
Kuva 5. Mikropalveluiden periaatteet (Newman, 2015, 246)

Lewis ja Fowler (2014) ovat esittäneet yhdeksän mikropalveluarkkitehtuuria kuvaavaa ominaisuutta verkkoartikkelissaan, johon myös monet tieteelliset artikkelit viittaavat. Yhdeksän ominaisuutta ovat komponentointi palveluiden kautta (engl. componentization via services), organisointi liiketoimintojen mukaan (engl. organized around business capabilities), tuotteet projektien sijaan (engl. products not projects), älykkäät päätepisteet ja typerät putket (engl. smart endpoints and dumb pipes), hajautettu hallinto (engl. decentralized governance), hajautettu tiedonhallinta (engl. decentralized data management), infrastruktuurin automatisointi (engl. infrastructure automation), virheisiin varautuminen (engl. design for failure) ja evoluutiosuunnittelu (engl. evolutionary design).

Palveluiden käyttäminen komponentteina (Lewis & Folder, 2014) johtaa selkeämpiin komponenttien rajapintoihin. Jotkut palvelut otetaan käyttöön aina yhdessä, kuten sovelluskomponentti ja tietokanta, jota vain kyseinen palvelu käyttää. Evoluutiosuunnittelulla tarkoitetaan, että palveluiden hajauttamisella voidaan hallita sovelluksen muutoksia hidastamatta muutosta. Muutosten hallinta ei tarkoita muutosten vähentämistä, vaan oikeilla työkaluilla voidaan tehdä usein, nopeasti ja hyvin hallittuja muutoksia ohjelmistoihin. Sovelluksen pilkkomisessa olennaista on itsenäinen korvattavuus ja päivitettävyys, mikä tarkoittaa että etsitään pisteitä, joissa komponentti voidaan kirjoittaa uudelleen vaikuttamatta sen yhteistyökumppaneihin. Voi olla myös tarpeen pitää samanaikaisesti muuttuvat asiat samassa paketissa. Jos toistuvasti huomataan kahden palvelun toimivan tai muutosten tapahtuvan yhdessä, voi ne olla tarpeen yhdistää. Sovellus voi olla myös suunniteltu ja rakennettu alun perin monoliittina, mutta sitä kehitetään jatkossa mikropalveluiden suuntaan. Ohjelmiston ytimenä säilyvään monoliittiin lisätään uusia ominaisuuksia mikropalveluina, jotka käyttävät monoliitin sovellusliittymää.

Nadareishvili ja muiden (2016, 89) mukaan mikropalveluiden keskeisiin käsitteisiin kuuluvat itsenäinen käyttöönotto (engl. independent deployability), konttien rooli kustannustehokkaassa käyttöönotossa ja erityisesti Dockerin rooli mikropalveluiden käytössä, palveluiden löydettävyys ja tunnistettavuus (engl. service discovery), turvallisuus, reititys, muutos ja orkestrointi (engl. orchestration). Näiden todetaan yhdessä antavan vankan pohjan mikropalveluarkkitehtuurin operatiivisten tarpeiden ymmärtämiseen, suunnitteluun ja toteuttamiseen. Onnistuneella mikropalveluarkkitehtuurilla oletetaan myös olevan tietty infrastruktuurin automatisoinnin taso ja toiminnallista kypsyyttä (engl. operational maturity).

Li ja muiden (2021) toteuttamassa systemaattisessa kirjallisuuskatsauksessa tunnistettiin mikropalveluarkkitehtuurin kuusi tärkeintä laatuominaisuutta (engl. quality attributes, QA), jotka ovat skaalautuvuus (engl. scalability), suorituskyky (engl. performance), saatavuus (engl. availability), seurattavuus (engl. monitorability), turvallisuus (engl. security) ja testattavuus (engl. testability). Näitä laatuominaisuuksia arkkitehtonisesti vastaamaan tunnistettiin 19 taktiikkaa, joista kaksi liittyy skaalautuvuuteen, neljä suorituskykyyn, neljä saatavuuteen, neljä seurattavuuteen, kolme turvallisuuteen ja kaksi testaukseen (Kuva 6). Kirjallisuuskatsauksessa oli mukana 72 tutkimusta vuosilta 2015-2018. Johdonmukaisuuden varmistamiseksi kirjallisuuskatsauksessa käytettiin yleisesti hyväksyttyjä standardeja kuten ISO/IEC 25,010 sekä Bass ja muiden (2013) määrittelyitä ominaisuuksista.



Kuva 6. Mikropalveluarkkitehtuurin laatutekijät ja taktiikat (Li et al., 2021, 8)

Li ja muiden (2021) kirjallisuuskatsauksessa seurataan myös edellä esitettyjä Lewis ja Folderin (2014) yhdeksää ominaisuutta ja Newmanin (2015) seitsemää periaatetta. Mikropalveluarkkitehtuurin etuihin ja ominaisuuksiin tunnistetaan komponentointi palveluiden kautta, mikä parantaa ohjelmiston muunneltavuutta, skaalautuvuutta ja käyttöönottoa, organisointi liiketoimintojen mukaan, ymmärrettävä ja ylläpidettävä koodi, infrastruktuurin automatisointi jatkuvan toimituksen ja DevOps-menetelmien mukaisesti sekä hajautettu hallinto ja tiedonhallinta. Li ja muiden (2021, 20) tutkimuksessa todetaan, että mikropalveluarkkitehtuurityyli on kasvavassa määrin hyväksytty ja omaksuttu ohjelmistokehityksen arkkitehtoninen tyyli, joka voittaa perinteisen monoliittisen arkkitehtuurityylin rajoitukset.

4.2 Skaalautuvuus

Skaalautuvuus (Li et al., 2021, 7) tarkoittaa järjestelmän kykyä lisätä resursseja käsittelemään vaihtelevaa määrää palvelupyyntöjä. Horisontaalinen ja vertikaalinen skaalautuvuus ovat erilaiset tavat resurssien lisäämiseen. Horisontaalinen skaalautuvuus on mikropalveluarkkitehtuurin tärkein ominaisuus, koska se mahdollistaa mikropalveluiden instanssien skaalaamisen. Horisontaalisen skaalautuvuuden haasteena on automaattisesti ratkaista kuinka monta instanssia tietystä mikropalvelusta on oltava käynnissä halutun palvelun laadun (engl. quality of service, QOS) saavuttamiseksi. Li ja muiden (2021, 7) tutkimuksessa tunnistettiin skaalautuvuuden taktiikoiksi horisontaalinen kopiointi (engl. horizontal dublication) ja vertikaalinen hajottaminen (engl. vertical decomposition).

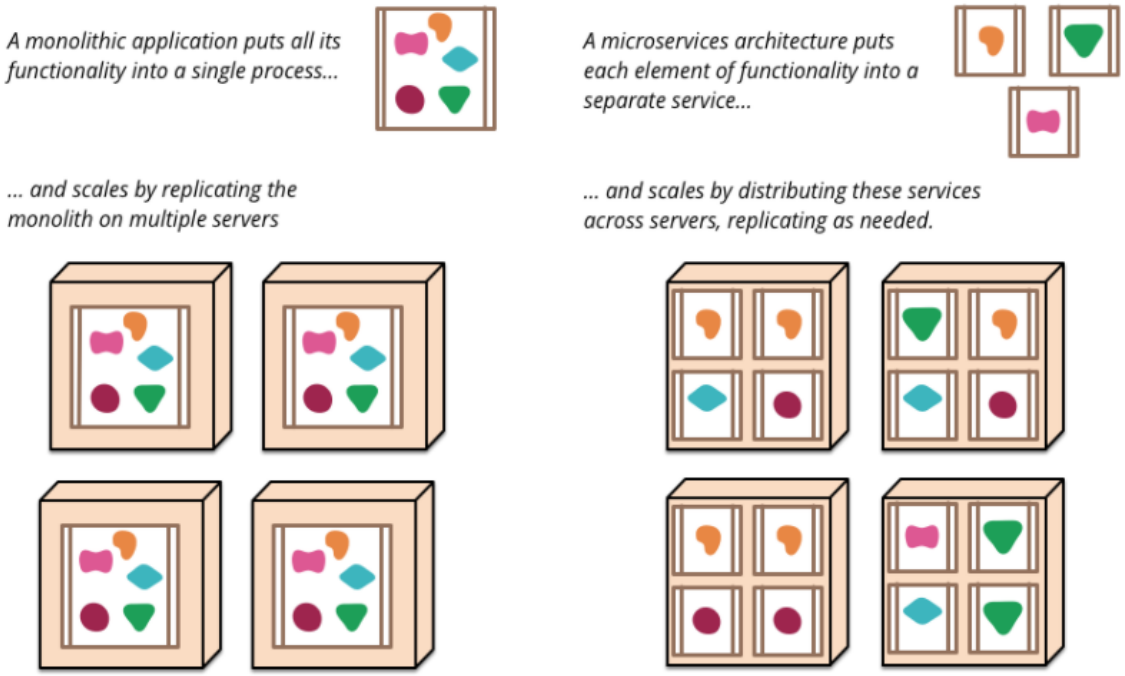
Horisontaalisessa kopioinnissa (Li et al., 2021, 8) mikropalveluiden instanssien määrää kasvatetaan tai vähennetään vastaamaan palvelupyyntöjen määrää ja saavuttamaan haluttu palvelun laadun taso. Mikropalveluarkkitehtuurissa on reagoitava nopeasti ja sovitettava mikropalveluiden instanssien tarjonta mahdollisimman lähelle niihin kohdistuvaa kysyntää, samalla varmistaa palvelun laatu ja toisaalta minimoida kustannukset. Horisontaalista kopiointia voidaan toteuttaa reaktiivisella automaattisella skaalauksella, jossa käytetään ennalta määritettyjä raja-arvoja (engl. threshold) instanssien lisäämisessä ja vähentämisessä, kuten CPU-käyttöä, HTTP-viivettä, HTTP-suoritustehoa tai muita metriikoita. Jos seurattava mittari ylittää ennalta määritetyn raja-arvon, skaalain päättää joko lisätä tai vähentää palveluiden instanssien määrää. Horisontaalinen kopiointi tukeutuu myös seurattavuuden taktiikoihin mittareiden raja-arvojen saamiseksi. Proaktiivisessa eli ennakoivassa automaattisessa skaalauksessa ennustetaan mikropalveluiden työmäärää palvelupyyntöjen historiatietojen perusteella.

Vertikaalinen hajottaminen (Li et al., 2021, 9) ohjaa monoliitin hajottamista sopiviksi mikropalveluiksi korkeamman ja itsenäisemmän skaalautuvuuden saavuttamiseksi. Skaalautuvuutta ja resurssien tehokasta käyttöä voidaan parantaa erottamalla vastuita, toimintoja ja dataa mikropalveluihin. Järjestelmän sopiva hajottaminen palveluihin on edellytys skaalautuvuudelle mikropalveluarkkitehtuurissa. Palveluita voidaan rajata esimerkiksi noudattamalla yhden vastuun periaatetta, jossa samasta syystä muuttuvat ohjelmiston elementit tulisi koota yhteen. Vertikaalisen hajottamisen taktiikassa päätökset palveluiden rajaamisessa tulisi tehdä huolellisesti, jotta löydetään skaalautuvuuden ja suorituskyvyn tasapaino. Korkean koheesion ja löyhän kytkennän periaatteilla voidaan ratkaista mikropalveluiden koko, kun tasapainotellaan itsenäisen skaalautuvuuden ja palveluiden vuorovaikutuksen suorituskyvyn välillä. Mitä pienempi mikropalvelu ei ole aina parempi. Liian pienet mikropalvelut kasvattavat palveluiden määrää ja niiden välistä vuorovaikutusta verkossa, mikä voi laskea ohjelmiston suorituskykyä.

Toimivan hajauttamisen (Newman, 2015, 2-3, 30) edellytyksenä on palveluiden ja sovellusliittymien oikea mallintaminen ja toteuttaminen, jotka tukevat palveluiden integroimista. Hyvän mikropalvelun kaksi keskeistä käsitettä ovat korkea koheesio (engl. high cohesion) ja löyhät kytkökset (engl. loose coupling). Korkealla koheesiolla tarkoitetaan kuinka hyvin yksittäinen mikropalvelu keskittyy toteuttamaan tietyn toiminnallisuuden. Kun palvelut ovat löyhästi kytkettyinä toisiinsa, yhtä palvelua pitäisi voida muuttaa ilman toiseen palveluun tehtävää muutosta. Löyhästi kytketty palvelu tietää muista palveluista vain sen verran, jonka se tarvitsee yhteistyöhön niiden kanssa. Jos mikropalvelua ei voida muuttaa ja ottaa käyttöön itsenäisesti, monet mikropalveluarkkitehtuurin mahdollisuudet ovat vaikeita saavuttaa. Palveluiden sisältöjen (Newman, 2015, 246, 249) mallintaminen ja rajaaminen on sitä vaikeampaa mitä huonommin toimialue (engl. domain) ja ohjelmiston vaatimukset tunnetaan ja ymmärretään. Toimintojen epäonnistunut rajaaminen erillisiin palveluihin voi heikentää mikropalveluiden itsenäisyyttä ja skaalautuvuutta. Newman toteaa, että liiketoimintojen mukaan mallinnetut palvelut ovat vakaampia kuin teknisten syiden mukaan tehdyt rajaukset. Tällöin voidaan paremmin vastata liiketoiminnan muutoksiin.

Mikropalveluarkkitehtuurissa (Lewis & Folder, 2014) ohjelmisto on jaettu löyhästi linkitettyihin komponentteihin, joista jokainen voidaan suorittaa itsenäisesti mikropalveluna ja korvata tai päivittää ilman muiden komponenttien osallistumista. Ohjelmistokehityksessä on aina ollut halu rakentaa järjestelmiä kytkemällä komponentteja toisiinsa. Komponentti voidaan määritellä ohjelmistoyksiköksi, joka on itsenäisesti vaihdettavissa ja päivitettävissä. Palveluiden itsenäinen käyttöönotto ei ole kuitenkaan ehdotonta, koska osa muutoksista muuttaa palvelurajapintoja ja vaatii jonkin verran palveluiden yhteensovittamista. Hyvän mikropalveluarkkitehtuurin tavoitteena on kuitenkin minimoida useisiin palveluiden kohdistuvat muutokset oikein rajatuilla palveluilla ja kevyillä kytköksillä. Itsenäiset ja löyhästi kytketyt mikropalvelut voivat tehostaa resurssien käyttöä ja vähentää kustannuksia skaalaamalla tiettyjä palveluita koko järjestelmän sijaan.

Kontti- ja mikropalveluarkkitehtuuri (Wan et al., 2018) mahdollistavat paremman skaalautuvuuden ja joustavamman käyttöönoton. Sovellukset ovat perinteisesti monoliittisia, joissa toiminnallisuudet ovat yhdessä kokonaisuudessa. Vaikka monoliittisia sovelluksia on helppo ottaa käyttöön, niitä on vaikea päivittää ja ne ovat vähemmän skaalautuvia. Pienet muutokset aiheuttavat päivityksen ja käyttöönoton koko järjestelmään, mikä väistämättä viivästyttää sovelluksen julkaisusykliä. Monoliitin skaalaus (Lewis & Fowler, 2014; Nadareishvili et al., 2016, 89) useammalle palvelimelle vaatii koko ohjelmiston kopioinnin sen sijaan, että skaalattaisiin vain osia, jotka vaativat enemmän resursseja (Kuva 7). Mikropalveluarkkitehtuuriin kuuluva toisista mikropalveluista riippumaton käyttöönotto mahdollistaa valikoivan ja tarpeisiin perustuvan skaalauksen.



Kuva 7. Monoliitin ja mikropalveluiden skaalautuvuus (Lewis & Folder, 2014)

Valikoiva skaalaus voi tuoda olennaista joustavuutta ja taloudellisia säästöjä yrityksille. Jos osa ohjelmistosta, esimerkiksi yksittäinen mikropalvelu on liian kuormittunut, voidaan ottaa käyttöön uusi instanssi tai siirtää mikropalvelu ympäristöön, jossa on enemmän resursseja ilman, että tarvitsee laajentaa laitteistokapasiteettia koko ohjelmistolle. Tarvittaessa yhdestä palvelusta käynnistetään useampi instanssi palveluun kohdistuvan kuorman tasaamiseksi ja ohjelmiston tai järjestelmän vakauttamiseksi. (Lewis & Fowler, 2014; Nadareishvili et al., 2016, 89)

4.3 Suorituskyky

Suorituskyky on (Li et al., 2021, 9) mittari sille, kuinka nopeasti järjestelmä vastaa tapahtumiin ja palvelupyyntöihin eli suoriutuu sille asetetuista aikavaatimuksista (engl. timing requirements). Hajautettujen mikropalveluiden toinen kriittisin laatutekijä on suorituskyky, johon vuorovaikutuksen kompleksisuus vaikuttaa. Resurssien allokointi esimerkiksi konttien avulla voi vaikuttaa palvelupyyntöjen siirtokykyyn ja vastausaikaan. Skaalautuvuuden ja suorituskyvyn välillä voidaan joutua tekemään kompromissi. Pienemmät mikropalvelut voivat parantaa skaalautuvuutta, mutta heikentää suorituskykyä palveluiden välisen vuorovaikutusten kasvun vuoksi. Suorituskykyä voidaan parantaa yhdistämällä kaksi mikropalvelua skaalautuvuuden kustannuksella. (Li et al., 2021, 9)

Suorituskykyyn liittyvät taktiikat (Li et al., 2021, 9-10) on jaettu neljään ryhmään: resurssien hallinta ja allokointi (engl. resource management and allocation), kuormituksen tasoitus (engl. load balancing), konttiteknologia (engl. containerization) ja profilointi

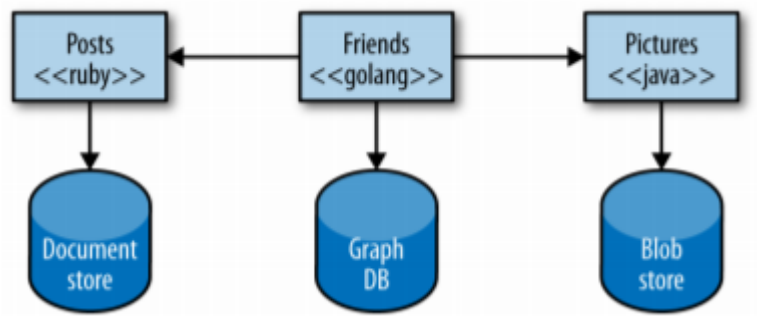
(engl. profiling). Optimoidulla resurssien hallinnalla ja allokoinnilla voidaan tehostaa resurssien käyttöä palvelupyyntöjen käsittelyssä ja saavuttaa parempi suorituskyky kuten korkeampi siirtokyky ja lyhyempi vastausaika. Resurssien hallinnan ja allokoinnin taktiikka luottaa horisontaalisen kopioinnin taktiikkaan resurssien skaalaamiseksi ja sovittamiseksi. Mikropalveluiden orkestrointityökalua käytetään resurssien kuten konttien keskitettyyn hallintaan. Orkestroinnin käyttämisen edellytyksenä on myös suorituskykymittareiden seuranta.

Kuormituksen tasauksessa (Li et al., 2021, 10) yksittäiselle mikropalvelulle tuleva liikenne jaetaan sen instanssien kesken, jolloin palvelupyynnöt voidaan suorittaa maksimoimalla nopeutta ja kapasiteetin käyttöä. Sen avulla varmistetaan, ettei yksittäinen palveluinstanssi ole merkittävästi ylityöllistetty verrattuna muihin, mikä voi heikentää järjestelmän suorituskykyä. Kuormituksen tasauksen toteuttamiseksi palvelun pitää pystyä jakamaan siihen tuleva liikenne instanssiensa kesken.

Profilointi (Li et al., 2021, 11) ohjaa suorituskykyyn vaikuttavien asioiden tunnistamista ja suorituskyvyn optimointia. Profiloinnilla analysoidaan mikropalveluista muodostuvaa järjestelmää kuten vaadittua muistia, CPU-käyttöä ja kaistanleveyttä sekä selvitetään mitkä asiat tarvitsevat optimointia. Muistin profilointia voidaan käyttää muistivuotojen löytämiseen muistin käytön optimoimiseksi.

Useista yhteistyössä toimivista palveluista koostuvassa järjestelmässä voidaan käyttää erilaisia teknologioita (Kuva 8). Palvelulle voidaan valita sille sopiva teknologia, joka auttaa edellytetyn suorituskyvyn saavuttamisessa. Mikropalvelut mahdollistavat uusien teknologioiden käyttöönoton nopeammin kuin perinteisissä monoliittisissa ohjelmistoissa, jotka rakennetaan yhden kasvavan koodipohjan päälle. Uuden teknologian (ohjelmointikieli, tietokanta, ohjelmistokehys) käyttöön liittyvät riskit voidaan rajata yhteen palveluun, jolloin uusien teknologioiden kokeileminen on helpompaa. (Newman, 2015, 4)

Datan tallentaminen voidaan myös toteuttaa eri lailla eri palveluissa. Jokainen palvelu voi käyttää ja hallita omaa itsenäistä tietokantaa ja sopivaa tietojen tallennusmenetelmää (tietokanta-per-palvelu-suunnittelumalli), mikä tukee myös palveluiden skaalautuvuutta, itsenäisyyttä ja turvallisuutta. Jos tietokannan skeema eli taustalla oleva rakenne muuttuu, se ei vaikuta muihin mikropalveluihin tai tietokantoihin. Muiden palveluiden pääsy tietokantaan ja tietojen suojaus voidaan varmistaa. (Taibi et al., 2018)

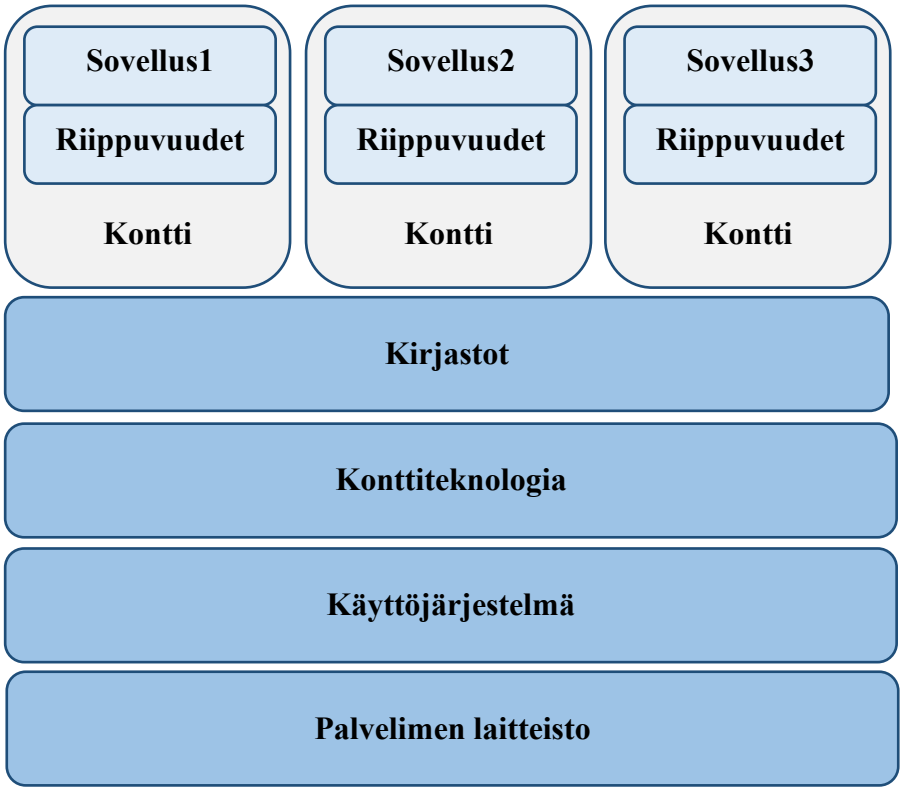


Kuva 8. Mikropalvelut voivat hyödyntää eri teknologioita (Newman, 2015, 4)

4.3.1 Konttiteknologia

Konttiteknologia (Li et al., 2021, 11) on taktiikka virtualisoinnin toteuttamiseen konttien avulla. Virtualisointi on ollut avaintekijä hajautettujen sovellusten suorituskyvyn parantamisessa. Virtualisointi (Bui, 2015, 1) mahdollistaa sovelluksen skaalautuvan, toistettavan ja turvallisen ajoympäristön, kun sovelluksesta tai koneesta voidaan tehdä virtuaalinen versio. Konttiteknologia eli konttipohjainen käyttöjärjestelmän virtualisointi (engl. operating system level virtualization) ja hypervisor-pohjainen laitteiston virtualisointi (engl. hardware level virtualization) ovat keskeisimmät virtualisointitekniikat. (Li et al., 2021, 11) Konttiteknologiat kuten Docker tarjoavat ketteryyttä sovellusten kehittämisessä ja käytössä erityisesti yhdistettynä mikropalveluarkkitehtuuriin (Kang et al., 2016, 202).

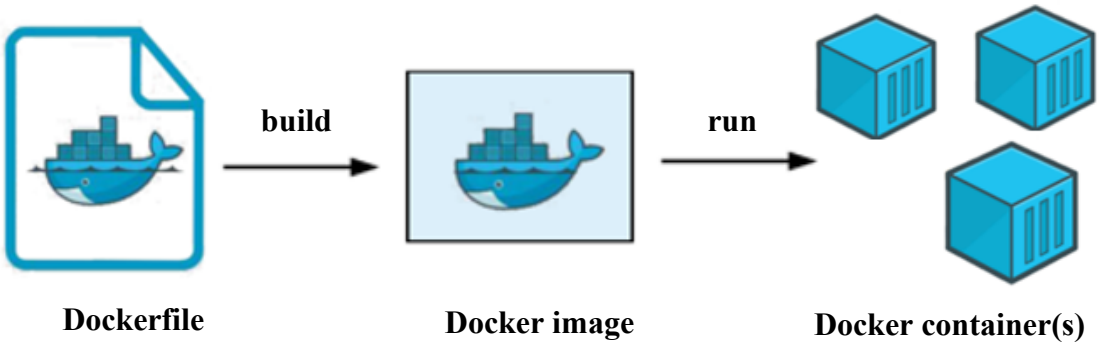
Docker on yleisin konttiteknologia. Kevyiden Docker-konttien avulla mikropalveluiden instansseja voidaan luoda ja ajaa enemmän, mikä johtaa korkeampaan resurssien käyttöasteeseen. Docker-kontit jakavat isäntäkäyttöjärjestelmän (engl. operating system, OS) ja tukevat kirjastojen käyttöä, mikä tekee konttien välisestä kommunikoinnista kevyempää ja tehokkaampaa ja voi johtaa parempaan suorituskykyyn, kuluttaa vähemmän muistia ja vähentää infrastruktuurin kustannuksia. Konttiteknologia sallii useiden eristettyjen ja itsenäisten ajoympäristöjen suorittamisen samalla isäntäkoneella (Kuva 9). Virtuaalikoneilla on usein oma käyttöjärjestelmänsä, joka voi kasvattaa yleiskustannuksia verrattuna konttiteknologiaan. (Li et al., 2021, 11; Wan et al., 2018, 97; Nadareishvili et al., 2016, 92; Nickoloff, 2016, 4-5)



Kuva 9. Konttipohjainen virtualisointi

Docker (2021) piilottaa taustalla olevaa teknologiaa. Docker-konttiteknologian (Kocher, 2018, 49) avulla sovelluksen koodi ja sen tarvitsemat riippuvuudet pakataan konttiin, jonka avulla palvelu voidaan suorittaa samalla tavalla esimerkiksi siirrettäessä kehitysympäristöstä testausympäristöön, testausympäristöstä tuotantoon tai fyysisestä koneesta pilvessä sijaitsevaan virtuaalikoneeseen. Kontti tarjoaa mikropalvelulle standardoidun ajoympäristön, johon voidaan valita sille sopivat teknologiat. Konttien avulla testit voidaan suorittaa luotettavasti tuotantoympäristöä mallintavassa testiympäristössä. Konttien avulla voidaan ajaa ja toistaa sovelluksia tuotantoympäristöä mallintavalla tavalla. Docker (Bui, 2015, 2) tukee myös hyvin kolmansien osapuolten työkalujen käyttöä, jotka yksinkertaistavat sovellusten hallintaa ja käyttöönottoa. DevOps-työkaluja voidaan integroida Dockerin kanssa.

Dockerin (2021) avulla sovelluksista luodaan Docker-kuvia (engl. Docker image), joita säilötään Docker-varastoon (engl. Docker registry). Docker-kuva rakennetaan Dockerfile-nimisellä ohjetiedostolla. Docker-kuva on sarja tietokerroksia pohjakuvan (engl. base image) päällä. Docker-kuvasta käynnistetään Docker-kontteja (engl. Docker container) eli niin monta instanssia samasta mikropalvelusta kuin halutaan (Kuva 10). Dockerin dokumentaatio sisältää kymmeniä Docker-komentoja, joita voidaan käyttää konttien luomisessa, käynnistämisessä, sammuttamisessa, poistamisessa ja valvonnassa. (Newman, 2015, 126; Nickoloff, 2016, 4-5; Bui, 2015, 3)



Kuva 10. Docker-kontin käyttöönotto

Monista konteista koostuvan sovelluksen käyttöä samalla palvelimella, samalla isäntäkoneella voidaan helpottaa Docker Composen avulla, joka käyttää yaml-tiedostoa konfiguraatioon. Docker Compose on suunniteltu yksinkertaistamaan useista Docker-konteista koostuvan sovelluksen suorittamista yhdellä komennolla. Docker Composessa määritellään useista konteista muodostuva sovellus: konteissa ajettavat mikropalvelut ja niiden väliset riippuvuudet, konttien tarvitsemat ympäristömuuttujat, tarvittavien tiedostojen sijainnit ja sisäinen verkko. Docker Compose voi skaalata palvelun ajamaan useita instansseja. Docker Composen avulla voidaan käynnistää siinä määritellyt palvelut ja yhdistää palvelut samaan verkkoon, jossa kontit voivat viitata toisiinsa nimillään ja näkyvät vain toisilleen. (Newman, 2015, 121, 127; Kocher, 2018)

Pilvipohjaisten ohjelmistojen jatkuvan integraation ja toimituksen mahdollistamiseksi suositaan koodin elinkaaren hallintaa käyttämällä kuvapohjaisia lähestymistapoja kuten versioituja Docker-kuvia. Jos ongelma ilmenee uuden julkaisun yhteydessä, voidaan palata edelliseen toimivaan konttikuvaan ja odottaa kunnes uusi korjattu versio on saatavilla. (Kang et al., 2016, 204)

4.3.2 Konttien orkestrointi

Orkestrointityökalut kuten Kubernetes (2021) tukevat Docker-kontteja ja tarjoavat abstraktin kerroksen resurssien hallintaan ja ajoitukseen (Bui, 2015, 2). Useille palvelimille hajautettujen konttien ajamiseen ja hallintaan voidaan käyttää Kubernes-orkestrointijärjestelmän klusteritekniikkaa. Googlen kehittämä Kubernetes on avoimen lähdekoodin alusta konttisovellusten käyttöönoton, skaalauksen ja hallinnan automatisointiin. Konttien hallinnan ja käyttöönoton automatisoinnilla voidaan ottaa kontteja käyttöön tarpeen mukaan. Lisäksi Kubernetes tarjoaa verkostoitumista konttien välillä ja tilan seurantaa. Kubernetes huolehtii sovelluksen kaaduttua sen käynnistämisestä uudelleen automaattisesti ilman, että kontti joudutaan manuaalisesti laittamaan pystyyn. Tehtävänä on pitää sovellus automaattisesti ajossa. (Newman, 2015, 121, 127; Kocher, 2018)

Kubernetes-järjestelmää ajetaan virtuaalisten tai fyysisten (engl. bare-metal) palvelimien päällä. Kubernetes yhdistää palvelimia klustereiksi, joita käytetään konttien ajamiseen. Kubernetekseen kuuluvalla Podilla (engl. Pod) ryhmitellään kontteja. Pod on yksikkö yhdelle tai useamman kontin ryhmälle, jota käytetään konttiryhmän ajoittamiseen, sijoittamiseen, käyttöönottoon, skaalaukseen ja ajonaikaiseen eristämiseen. Pod-kontit ajoitetaan, sijoitetaan samalle isäntäkoneelle ja otetaan käyttöön ja skaalataan yhdessä. (Ibryam & Huss, 2019, 5-6) Suurimmat pilvipalveluiden tarjoajat kuten Amazon Web Services, Google Cloud ja Microsoft Azure tarjoavat Kubernetes-klustereita palveluina, jotka yksinkertaistavat ja helpottavat Kuberneteksen käyttöönottoa ja hallintaa.

Konttien ajoitusratkaisut helpottavat konttien käytön automatisointia ja abstraktointia. Kubernetes huolehtii konttien tasapainottamisesta palvelimien välillä, kun pitää päättää mikä kontti käynnistetään milläkin palvelimella ja kuinka suuri osa palvelimen resursseista tulee olla tietyn palvelun käytössä. Mikropalveluarkkitehtuuriin kuuluu päätösten teko siitä, kuinka paljon automatisointia käytetään hallintaan ja kuinka paljon hallintaa pidetään itsellä, ja onko Kubernetes siihen sopiva vaihtoehto. Kubernetes saattaa hyvin todennäköisesti sopia mikropalveluarkkitehtuuriin, mutta voidaan myös päättää, että mikropalveluita hallitaan tukipalvelun, esimerkiksi HashiCorpin Consul-palvelun (2021), avulla. (Nadareishvili et al., 2016, 97)

4.4 Saatavuus

Saatavuus (Li et al., 2021, 12) tarkoittaa järjestelmän kykyä toipua vikatilanteista niin, että ei saatavilla olevan palvelun tila ei ylitä vaadittua arvoa. Saatavuus kattaa tässä myös luotettavuuden vaatimuksen. Saatavuuden merkitys korostuu mikropalveluarkkitehtuurissa, koska hajautetut mikropalvelut ovat alttiita häiriöille ja ongelman juurisyyn löytäminen voi olla haastavaa kompleksisessa vuorovaikutuksessa. Saatavuuden hallintaan Li ja muut (2021, 12) tunnistivat neljä taktiikkaa: vikojen valvonta (engl. fault monitor), palvelurekisteri (engl. service registry), katkaisin (engl. circuit breaker) ja epäjohdonmukaisuuden käsittelijä (engl. inconsistency handler).

Vikojen valvonnassa tunnistetaan mikropalveluiden virhetilanteita jatkuvan oikean toiminnan tarkkailun (engl. health monitoring) avulla käyttämällä tiettyä komponenttia. Mikropalvelujen saatavuuden ja luotettavuuden kannalta on välttämätöntä havaita häiriö ennen kuin järjestelmä voi ryhtyä toimenpiteisiin virhetilanteista toipumiseksi. Korkean saatavuuden saavuttamiseksi mikropalvelupohjaiset sovellukset edellyttävät jatkuvaa seurantaa, jotta niiden tilaa voidaan analysoida automaattisesti ja reagoida mahdollisimman vähäisellä ihmisen puuttumisella. Mikropalveluiden valvontaan voidaan käyttää keskitettyä valvontaa (engl. centralized monitor), symmetristä valvontaa (engl. symmetric monitor) ja välillistä valvontaa (engl. arbitral monitor). Keskitetystä valvonnasta vastaava palvelu päivittää palveluiden instanssien tilatiedot (engl. health status) palvelurekisteriin.

Keskitetty valvonta edellyttää lisäpalvelun käyttöönottoa, joka osaltaan vie resursseja ja lisää yhden mahdollisen epäonnistumispisteen järjestelmään. (Li et al., 2021, 12)

Hajautetuista mikropalveluista (Newman, 2015, 236) muodostuvassa järjestelmässä haasteena on palveluiden tunnistaminen ja löydettävyys (engl. service registration and discovery). Palveluiden on löydettävä toisensa hajautetussa ympäristössä. Kun kokonaisuuteen lisätään uusi instanssi, on sen kanssa yhteistyötä tekevien palveluiden saatava tieto uuden instanssin sijainnista. On myös hyvä tietää mitä palveluita on ajossa ja missä sijainnissa, jotta tiedetään mitä palveluita tulee monitoroida. Sovelluskehittäjien tiedossa on hyvä olla saatavilla olevat sovellusliittymät, joita uudet palvelut voivat hyödyntää. Mikropalveluarkkitehtuurissa on palveluiden tunnistamiseen liittyviä ratkaisuja, jotka tarjoavat mikropalvelusta luodulle instanssille mahdollisuuden rekisteröidä itsensä ja ilmoittaa itsensä ja sijaintinsa. Ratkaisut tarjoavat myös keinon löytää palvelu sen jälkeen, kun se on luotu ja toiminnassa. Palveluiden löytäminen on monimutkaisempaa ympäristössä, jossa jatkuvasti otetaan käyttöön ja poistetaan käytöstä palveluita.

Palvelurekisteri tallentaa ajossa olevien palveluiden sijainnin keskitettyyn paikkaan, jota palvelut voivat käyttää muita palveluita koskevien tietojen haussa. Palvelurekisterin käyttöä puoltaa se, että mikropalveluilla voi olla ajossa useita instansseja, joiden määrä voi muuttua. Palveluiden sijainnin paikallistaminen häiriöiden havaitsemiseksi on tärkeää. Palvelurekisterinä toimiva palvelu voi automaattisesti rekisteröidä mikropalveluiden instanssit ja niiden sijainnit. Palvelurekisteri voi myös poistaa instanssin luettelosta, jos palvelusta seurattavaa sykettä ei enää saada. Palvelurekisteri voidaan toteuttaa myös niin, että jokainen palvelu rekisteröi itsensä käynnistymisen jälkeen palvelurekisteriin. Keskitetyn palvelurekisterin käyttö lisää järjestelmään mahdollisen epäonnistumispisteen, jos palvelurekisteri ei toimi oikein. (Li et al., 2021, 12)

DNS-nimipalvelun (engl. Domain Name Services) avulla voidaan yhdistää nimiä IP-osoitteisiin. IP-osoitteiden sijasta voidaan käyttää helpommin muistettavia nimiä ja löytää palveluiden sijainti verkossa. DNS on yleisesti käytössä oleva ja eri teknologioiden tukema standardi, mutta tietojen päivittäminen voi olla haasteellista, kun ollaan tekemisissä vaihtuvien ja kertakäyttöisten mikropalveluiden ilmentymien kanssa. DNS:n haasteet löytää palvelut dynaamisessa ja muuttuvassa ympäristössä ovat johtaneet vaihtoehtoisiin tapoihin, jotka tarjoavat palvelulle mahdollisuuden rekisteröidä itsensä keskusrekisteriin ja mahdollisuuden etsiä ja löytää nämä palvelut. (Newman, 2015, 237-238)

Mikropalveluiden hallinnan ja näkyvyyden parantamiseen on tarjolla avoimen teknologian ratkaisuja, kuten HashiCorpin Consul (2021) ja CoreOS:n ETCD (2021). Palvelunetsintäohjelmat kuten Consul ja ETCD valvovat mikropalveluiden instansseja ja seuraavat mitkä IP-osoitteet ja portit ovat kunkin mikropalvelun käytössä. (Nadareishvili et al., 2016, 97, 100) Tukipalveluiden avulla voidaan keskittää sekä konfiguraatiohallintaa

että palveluiden löydettävyyttä ja suorittaa palveluiden tilatarkistuksia (Newman, 2015, 239).

Katkaisin on taktiikka, joka estää virheelliseen palveluun kohdistuvat pyynnöt. Mikropalveluiden vikasietoisuus tulisi ottaa huomioon korkean saatavuuden varmistamiseksi, koska palvelut ovat riippuvaisia yhteistyöstä toisten palveluiden kanssa ja yhden palvelun epäonnistuminen voi vaikuttaa useisiin palveluiden ja johtaa jopa koko järjestelmän epäonnistumiseen. Epäjohdonmukaisuuden käsittelijä käsittelee kompromissia johdonmukaisuuden ja saatavuuden välillä. Sen tehtävänä on varmistaa järjestelmän johdonmukaisuus samalla, kun huolehditaan palveluiden saatavuudesta. Hajautetuissa järjestelmissä johdonmukaisuudella tarkoitetaan, että käyttäjät saavat saman vastauksen riippumatta siitä, mitä samasta palvelusta ajossa olevaa instanssia käytetään. Samasta palvelusta ajossa olevat instanssit voivat olla rinnakkaisesti vuorovaikutuksessa tietokannan kanssa, mikä voi aiheuttaa haasteita tietojen johdonmukaisuuteen. (Li et al., 2021, 13)

4.5 Seurattavuus

Seurattavuus (Li et al., 2021, 13-14) tarkoittaa järjestelmän kykyä tukea sen seurantaa. Seuranta on tärkeää mikropalveluiden dynaamisen rakenteen ja käyttäytymisen vuoksi. Seuranta voi koskea infrastruktuuria, sovelluksia tai ympäristöä kuten kontteja, sovellusten vasteaikaa tai verkkoa. Seurattavuuden parantamisella voi olla vaikutuksia myös muihin laatutekijöihin kuten skaalautuvuuteen, suorituskykyyn ja saatavuuteen. Seurattavuuden taktiikat jaetaan seurattavan tiedon luomiseen, keräykseen, tallentamiseen, käsittelyyn ja esittämiseen.

Newman (2015, 157) nostaa esiin kysymykset: Kuinka löydetään ohjelmiston toimintaan vaikuttava vika useille palvelimille hajautettujen palveluiden ja tuhansien lokirivien joukosta? Miten vika jäljitetään mikropalveluiden ketjussa ongelmalliseen palvelimeen tai palveluun? Seuraamalla (Li et al., 2021, 14) palvelin-, alusta- ja palvelukohtaisia mittareita saadaan suorituksen aikaista tietoa erilaisista näkökulmista esimerkiksi käytettävissä olevista palvelimista, palveluiden vasteajoista, vikatasoista ja resurssien kulutuksesta. Tietojen avulla voidaan seurata palveluiden tilaa ja tehdä suorituskykyyn liittyvää aikataulutusta.

Palvelukohtaisen näkymän lisäksi (Newman, 2015, 21, 249) on olennaista saada yhtenäinen näkymä mikropalveluiden muodostaman kokonaisuuden tilasta (engl. health). Järjestelmän luotettavuuden seurantaa ei voida jättää yksittäisten mikropalvelujen, konttien, ajonaikaisen käyttäytymisen ja tilan tarkkailuun. Seurantaa helpottaa, jos kaikki palvelut tuottavat tilan ja yleisen seurannan mittarit samalla tavalla. Palveluiden tilan ja mittareiden seurantaa varten voidaan ottaa käyttöön keskitetty tukipalvelu, johon palvelut ilmoittavat tietonsa. Sovellusten toiminnan ja suorituskyvyn seuranta (Lewis & Fowler,

2014) on haastavaa, jos ne eivät anna tietoja itsestään ulos. Mikropalveluiden toteutuksessa on jatkuvasti otettava huomioon, kuinka palvelun epäonnistuminen vaikuttaa järjestelmän toimintaan ja käyttäjäkokemukseen. Ohjelmiston sietokykyä testataan ja valvotaan. Palvelun epäonnistuminen on tärkeää havaita nopeasti ja epäonnistumisesta toipuminen toteuttaa automaattisesti, jos mahdollista.

Lokien keräyksellä tallennetaan mikropalvelun instanssia koskevat tulevat ja lähtevät palvelupyynnöt levylle aikaleimoittain (Li et al., 2021, 14). Lokien ja mittareiden keräämisellä ja seurannalla (Newman, 2015, 158) voidaan ymmärtää mikropalveluiden suorituskäyttäytymistä ja löytää ongelman lähde. Lokien keräämiseen ja siirtämiseen keskitettyyn tallennustilaan, indeksointiin ja visualisointiin on tarjolla tukipalveluita kuten ELK Stack (2021).

Palveluiden tunnistamisessa ja löydettävyydessä apuna käytettävät työkalut ja tukipalvelut (Nadareishvili et al., 2016, 101) tarjoavat ratkaisuja myös palveluiden seurantaan ja vikasietoisuuden parantamiseen. Consul-palvelun (2021) avulla voidaan seurata kuinka monta mikropalvelun instanssia on käynnissä ja seurata niiden tilaa. Consulin avulla voidaan seurata, että mikropalvelu vastaa tietyssä osoitteessa tai että palvelu palauttaa määrätyn vastauksen. Palveluiden ja järjestelmien käyttäytymisen ymmärtämiseen voidaan käyttää metriikkaraportointia (Newman, 2015, 159). Mikropalveluilla (Lewis & Fowler, 2014) toteutetuissa ohjelmistoissa korostetaan reaaliaikaista valvontaa, jota voidaan toteuttaa erilaisilla mittareilla (engl. metrics). Voidaan seurata kuinka paljon pyyntöjä palvelulle tulee sekunnissa ja kuinka kuormittunut palvelu on. Seurannan työkaluilla kuten Prometheus-ohjelmalla (2021) ja Grafana-ohjelmalla (2021) voidaan seurata operatiivisia ja liiketoimintaan liittyviä mittareita, joiden avulla saadaan tietoja palveluiden tilasta, suoritustehosta, läpisyötöstä (engl. throughput) ja viiveestä (engl. latency). Metriikkaa tarjoava palvelu voidaan integroida palvelun etsintäohjelmisto Consulin (2021) kanssa.

Seurannalla voidaan parantaa mikropalveluarkkitehtuuriin perustuvan järjestelmän laatua ja toimintavarmuutta, mikä liittyy myös ohjelmiston skaalautuvuuteen. Palveluiden tuottamien lokitietojen ja metriikan avulla luodaan kuva järjestelmään kohdistuvasta skaalautumistarpeesta, joka välitetään skaalautumista hoitavalle palvelulle. Kuormittunut tai kaatunut palvelu ei selviä sen vastuulla olevista tehtävistä.

4.6 Turvallisuus

Turvallisuus (Li et al., 2021, 15) tarkoittaa järjestelmän kykyä suojata tiedot ja estää luvaton pääsy järjestelmään samalla, kun mahdollistetaan pääsy valtuutetuille käyttäjille. Hajautetut palvelut ja palveluiden välinen vuorovaikutus rajapintojen välityksellä verkossa altistavat järjestelmän palveluihin kohdistuville hyökkäyksille. Palveluiden välinen

vuorovaikutus ja sovellusliittymät on suojattava. Yhteyden osapuolten keskinäisellä todennuksella varmistetaan, että osapuolilla on oikeus päästä tietoihin käsiksi. (Newman, 2015, 169; Soldani et al., 2018, 227)

Li ja muiden (2021, 15) toteamat turvallisuuden taktiikat ovat turvallisuusmonitorointi (engl. security monitor), todennus ja valtuutus (engl. authentication and authorization) ja tunkeutujien torjunta (engl. intrusion defender). Turvallisuusmonitoroinnissa valvotaan palveluita epänormaalin toiminnan ja hyökkäysten varalta. Turvallisuusmonitorointia voidaan tehdä paikallisesti palveluiden sisällä valvomalla verkkotapahtumia tai ulkoisesti valvomalla koko järjestelmää. Palvelut voivat käyttää omaa tai jaettua palvelua tai tietokantaa tunnistetietojen säilyttämiseen ja käyttäjän todentamiseen. Avainperusteisella todennuksella voidaan varmentaa palveluiden vuorovaikutus ja osapuolet. Asiakasvarmenteita (asiakassertifikaatteja) voidaan käyttää osapuolten välisen tiedonsiirron suojaamiseen. Tunkeutujien torjunnassa tunnistetaan sovelluksen suojaustila ja reagoidaan haavoittuvuuksiin. (Li et al., 2021, 16-17)

Mikropalveluiden välisen yhteyden osapuolten todentaminen ja niiden välisen liikenteen salaus voidaan toteuttaa SSL-protokollan seuraajan TLS-protokollan (engl. Transport Layer Security) avulla asiakasvarmenteiden (sertifikaattien) muodossa. Palvelin voi tarkistaa asiakassertifikaatin aitouden. Yksisuuntaisessa TLS-todennuksessa palvelin esittää varmenteensa asiakkaalle todentaakseen itsensä. Keskinäinen TLS-todennus laajentaa asiakas-palvelin-mallia molempien osapuolten todennukseen. Kahdensuuntaisen eli keskinäisen todennuksen (engl. mutual TLS, mTLS) avulla todennetaan yhteyden osapuolet ja salataan niiden välinen liikenne. Keskinäinen TLS-todennus käyttää X.509-varmenteita mikropalveluiden tunnistamiseen ja todentamiseen. Varmenne sisältää julkisen salausavaimen ja identiteetin. Varmenteen allekirjoittanut luotettava varmentaja todistaa, että varmenne esittää sitä edustavaa tahoa. (Newman, 2015, 175; Gupta, 2021)

4.7 Testattavuus

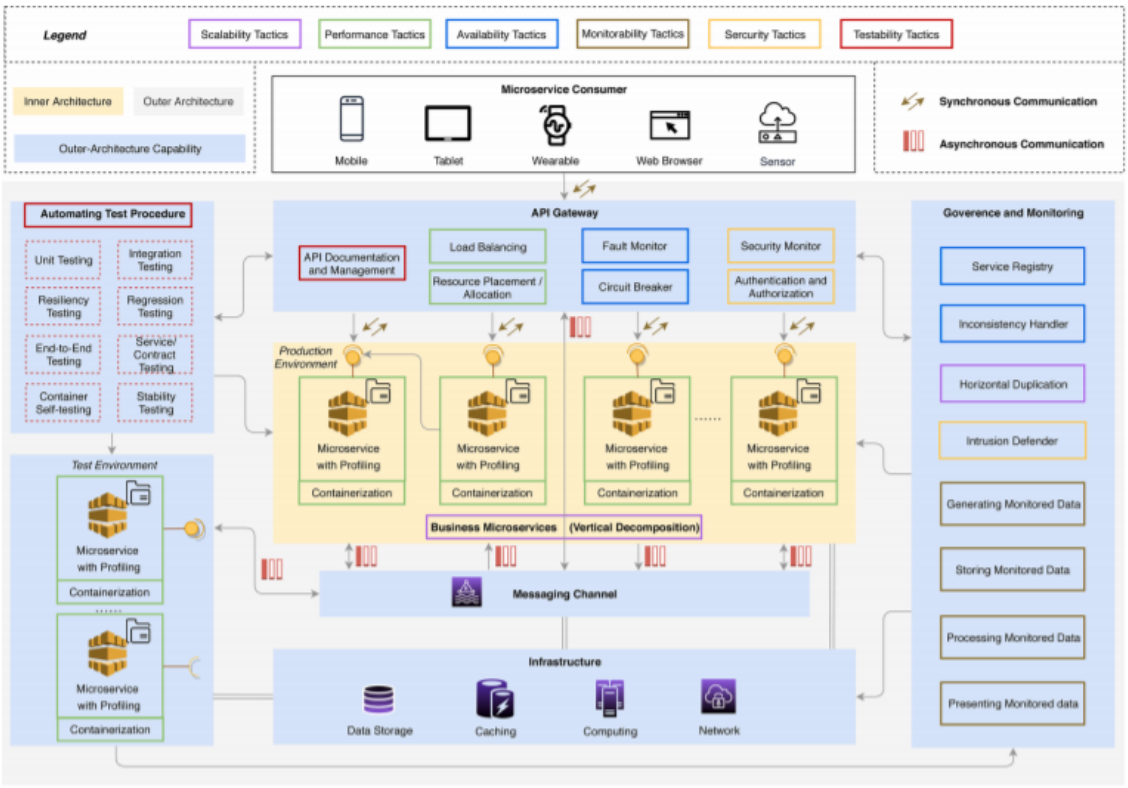
Testattavuus (Li et al., 2021, 17) mittaa järjestelmän kykyä osoittaa viat testauksen kautta. Mikropalveluiden välisen vuorovaikutuksen kompleksisuuden ja jatkuvan muutoksen vuoksi testattavuus on otettava huomioon järjestelmän suorituskyvyn, saatavuuden ja turvallisuuden varmistamiseksi. Mikropalveluarkkitehtuurissa testausprosessi on tärkeässä osassa palveluiden integroinnissa. Palveluiden suuren määrän manuaalinen testaus on aikaa vievää ja miltei mahdotonta. DevOps-menetelmien kannustamana testauksen automatisointi (engl. automating test procedure) on välttämätöntä tehokkuuden parantamiseksi. Automaattisen testauksen lisäksi testattavuus edellyttää rajapintojen dokumentointia ja hallintaa (engl. API documentation and management). Testien toteuttamiseksi on tunnettava rajapintojen kuvaukset, syntaksi ja rajapinnassa liikkuvien muuttujien tyypit.

Infrastruktuurin hallinnan automatisointi on edellytys toimivalle mikropalveluarkkitehtuurille. Automatisoinnilla hallitaan hajautetun ohjelmiston kompleksisuutta. Mikropalveluiden uusien ominaisuuksien, muutosten ja korjausten julkaisu- ja toimitussykliä nopeutetaan ja yhtenäistetään automatisoidulla käyttöönotto- ja julkaisuputkella sekä jatkuvan integraation ja toimituksen menetelmillä. Mikropalveluiden itsenäisyyden säilyttämiseksi on suositeltavaa, että mikropalveluilla on omat versionhallinnat. Käyttöönoton yhtenäistämisellä varmistetaan sen toimivuus kaikkialla. (Lewis & Fowler, 2014; Newman, 2015)

4.8 Yhteenveto

Mikropalveluarkkitehtuurin käyttöönotto (Li et al., 2021, 18) voi auttaa vähentämään monoliittisen järjestelmän sisäistä kompleksisuutta ja tarjota monia etuja kuten itsenäistä ja jatkuvaa käyttöönottoa, skaalautuvuutta ja teknologioiden heterogeenisuutta. Edut eivät ole kuitenkaan ilmaisia, koska hajauttamisesta seuraa uusia haasteita ja palveluiden vuorovaikutuksen kompleksisuutta, jota voidaan hallita laatutekijöihin liitetyillä taktiikoilla.

Kuvassa 11 on Li ja muiden (2021, 19) tutkimuksessa esitetty mikropalvelupohjaisen järjestelmän viitearkkitehtuuri. Vasemmassa reunassa on testauksen automatisointi ja kontitetut mikropalvelut testiympäristössä. Oikeassa reunassa on hallinto ja seuranta, johon kuuluu esimerkiksi palvelurekisteri ja seurannan datat. Mikropalveluiden asiakkaita voivat olla esimerkiksi web-selaimet, sensorit ja mobiililaitteet. Keskellä ovat tuotantoympäristössä konteissa suoritettavat mikropalvelut, joiden käyttöä tuetaan laatutekijöiden taktiikoilla.



Kuva 11. Mikropalveluiden viitearkkitehtuuri (Li et al., 2021, 19)

Kevyet ja ketterät kontit yhdistettynä mikropalveluarkkitehtuuriin ovat osoittautuneet lupaaviksi halutun tehokkuustason ja nopean käyttöönoton saavuttamisessa (Kang et al., 2016, 2002). Mikropalveluarkkitehtuurilla ja DevOps-menetelmillä todetaan olevan monia yhteisiä ominaisuuksia, jotka tekevät niistä sopivat toisilleen. Yhdessä ne edistävät isojen ongelmien hajottamista pienemmiksi paloiksi ja niiden käsittelyä poikkitoiminnallisten tiimien avulla. Konteissa ajettavat mikropalvelut voidaan toteuttaa itsenäisesti, koska DevOps mahdollistaa jatkuvan integraation ja käyttöönoton. (Waseem, 2020, 1-2)

Mikropalvelut ja kontit ovat rakennuspalikoita pilvipohjaisilla alustoilla. Pilvipohjaiset ympäristöt ovat skaalautuvia, ketteriä ja kustannustehokkaita, joten enemmän sovelluksia siirretään yksityisistä infrastruktuureista pilvipohjaisiin datakeskuksiin. Kohdentamalla pilven fyysiset resurssit oikein pilvisovellusten käyttöön, hyödynnetään laskentaresursseja (engl. computing resources) tehokkaasti ja pienennetään sovellusten käyttöönotosta ja käytöstä aiheutuvia kustannuksia. Konttipohjainen virtualisointi esimerkiksi Dockerin avulla on suosittu tapa pilvisovellusten käyttöönottoon. Konttipohjainen virtualisointi Dockerin mahdollistamana ja mikropalveluarkkitehtuurin tekniikat tarjoavat mahdollisuuksia parantaa sovelluksen skaalautuvuutta ja joustavaa kehitystä. (Wan et al., 2018, 97)

5 Kohdesovellus mikropalveluarkkitehtuurissa

Seuraavaksi tarkastellaan case-esimerkkinä uuden web-sovelluksen kehittämistä mikropalveluarkkitehtuurissa ja DevOps-tiimissä osana monista mikropalveluista koostuvaa matkapuhelinverkon analysointi- ja valvontaohjelmistoa. Ohjelmisto kerää verkossa toimivilta tukiasemilta reaaliajassa dataa, jonka avulla analysoidaan verkon tilaa. Tietoliikenne- ja radioverkon luotettava ja vakaa toiminta on verkossa toimiville osapuolille ja verkko-operaattoreille tärkeää.

5.1 Analysointi- ja valvontaohjelmisto

Ohjelmisto rakentuu useista Docker-konteissa ajettavista mikropalveluista, joita suoritetaan hajautetuilla Linux-palvelimilla, keskuspalvelimella ja keräilypalvelimilla, joihin tukiasemat ovat yhteydessä. Toiminnallisuudet on hajautettu pienempiin mikropalveluihin, jotka kommunikoivat keskenään REST-pohjaisten rajapintojen ja TCP-yhteyksien avulla. Mikropalveluarkkitehtuuri mahdollistaa sopivien teknologioiden ja työkalujen valitsemisen komponenttikohtaisesti. Ohjelmiston web-käyttöliittymä muodostuu useista itsenäisistä käyttöliittymistä, jotka tarjotaan käyttäjälle portaalikäyttöliittymän kautta. Portaalikäyttöliittymä mahdollistaa pääsyn useisiin itsenäisiin käyttöliittymiin yhteisen valikon kautta ja tarjoaa mahdollisuuden myös tiedon siirtoon käyttöliittymien välillä. Olennaista on, että käyttäjälle ohjelmisto näkyy yhtenäisenä, vaikka taustalla oleva toiminnallisuus on jaettu pienempiin osiin, helpommin hallittaviin taustaohjelmiin ja käyttöliittymiin.

Verkossa toimivat tukiasemat ohjataan lähettämään dataa hajautetuilla palvelimilla oleville datakerääjille, jotka kuuntelevat tukiasemilta tulevia yhteyksiä ja vastaanottavat niiltä dataa. Datakerääjät ovat itsenäisiä Docker-konteissa ajettavia mikropalveluita. Datakerääjän liitännäisohjelmat analysoivat, muokkaavat ja suodattavat dataa eri käyttötarkoituksia varten ja lähettävät sitä eteenpäin ulkopuolisille asiakkaille sekä keskuspalvelimella oleville mikropalveluille. Datakerääjä tarjoaa REST/JSON-pohjaisen API-rajapinnan, jonka kautta voidaan pyytää datakerääjän tiedot, tilastolaskurit (engl. statistics counters), hälytykset (engl. alarms), suorituskykymittarit (engl. key performance indicators, KPI), tukiasemayhteydet, konfigurointiparametrit ja lisenssitiedot. Rajapinnan kautta voidaan hallita liitännäisohjelmia (engl. plugins). Datakerääjä tarjoaa myös WebSocket-rajapinnan ilmoitusten vastaanottamiseen. Tietoja käytetään verkon analysointiin ja valvontaan.

Datakerääjille on aiemmin toteutettu käyttöliittymä, jonka koodipohjaan uusien ominaisuuksien ja muutosten tekeminen on osoittautunut haasteelliseksi. Käyttöliittymä ei ole enää linjassa organisaation käyttöliittymien suunnittelumallien ja teknologioiden kanssa, eikä se tue hyvin datakerääjien eli itsenäisten mikropalveluiden instanssien automaattista käyttöönottoa ja skaalausta. Datakerääjiä on tarpeen ottaa käyttöön ja poistaa

käytöstä niihin kohdistuvan liikenteen ja kuormituksen mukaan, joten käyttöliittymän tulisi pysyä ajan tasalla ajossa olevista datakerääjien instansseista.

Ohjelmisto asennetaan ja otetaan käyttöön erillisellä asennusohjelmalla. Ohjelmistoon kuuluvat mikropalvelut voidaan ottaa myös itsenäisesti käyttöön, mutta yhteisen asennusohjelman tarkoituksena on yksinkertaistaa ja helpottaa useista konteista koostuvan ohjelmiston käyttöönottoa ja konfigurointia. Docker-konttien käyttö mahdollistaa ohjelmiston käyttöönoton erilaisissa ajoympäristöissä. Ohjelmistoa, siihen kuuluvia mikropalveluita, infrastruktuurin automatisointia ja yhtenäistä asennusohjelmaa kehitetään jatkuvan integraation periaatteiden mukaisesti. Tavoitteena on ohjelmiston jatkuva toimitus, jossa muutokset viedään automatisoidun käyttöönotto- ja testausputken kautta tuotantoon.

Ohjelmistoa ei ole alun perin suunniteltu mikropalveluarkkitehtuuriin tai pilvipohjaisille alustoille. Osa komponenteista on toteutettu ennen mikropalveluiden ja Dockerin käyttöönottoa, mikä vaikuttaa siihen miten ne hyödyntävät mikropalveluiden ja konttien tarjoamia etuja. Ohjelmiston arkkitehtuuria kehitetään kohti skaalautuvia mikropalveluita. Ohjelmiston uusi sukupolvi on rakennettu Linux- ja pilvi-pohjaisista (engl. cloud-enabled) ja Docker-konteissa toimitettavista mikropalveluista. Käytössä oleva projektinhallintamenetelmä on Jira (2021). Ohjelmiston komponenttien versionhallinta on Gitlab-versionhallintavarastossa (2021), jossa komponenteilla on omat CI/CD-käyttöönottoputket. Ohjelmiston komponenttiin tehtävä muutos kulkee CI-/CD-putken läpi, jonka seurauksena on uusi Docker-kuva, joka julkaistaan JFrog-artefaktivarastoon (2021). Ohjelmistoa rakennetaan useiden tiimien yhteistyönä DevOps-menetelmien mukaisesti.

5.2 Hallintasovellus

Ohjelmistossa toimivien datakerääjien hallintaan toteutetaan uusi web-sovellus, hallintasovellus, jolla on selaimella käytettävä käyttöliittymä. Tutkielman tekijän rooli analysointi- ja valvontaohjelmistossa on uuden datakerääjien hallintasovelluksen kehittäminen ja suunnittelu.

5.2.1 Toiminnallisuus

Datakerääjien hallintasovelluksen toiminnallisuudet toteutetaan määriteltyjen käyttötapausten (engl. use case) mukaan. Käyttöliittymän toiminnallisuuksien suunnittelussa on kyse käyttäjäkokemuksesta. Tavoitteena on tarjota käyttäjille yksinkertainen ja selkeä käyttöliittymä datakerääjien rajapinnan kautta saatavien tietojen tarkasteluun ja datakerääjien hallintaan. Hallintasovelluksen käyttöliittymässä halutaan nähdä ajossa olevien datakerääjien instanssit, niiden status eli onko datakerääjä tavoitettavissa tarjoamansa rajapinnan kautta ja olennaiset tiedot datakerääjien seurantaa varten. Datakerääjän tuottamat hälytykset kertovat erilaisista ongelmatilanteista. Suorituskykyyn liittyvät mittarit ja

tilastolaskurit antavat tietoa saatavan datan määrästä, ohjelman kuormituksesta tai muistinkäytöstä. Mittareita käytetään valvontaan ja vianmääritykseen. Datakerääjä voi olla liian kuormittunut, jolloin se ei selviä tukiasemilta tulevan datan käsittelystä ja dataa häviää. Ohjelmistossa toimivia datakerääjiä skaalataan tarpeen ja kuormituksen mukaan, joten käyttöliittymän halutaan tukevan datakerääjien instanssien automaattista skaalausta ja käyttöönottoa. Käyttöliittymän kautta voidaan tarkastella ja hallita yhtä tai mahdollisesti satoja datakerääjiä.

Käyttöliittymässä halutaan nähdä datakerääjiin yhteydessä olevat tukiasemat: tukiasemien tiedot ja aikaleimat, jolloin yhteydet tukiaseman ja datakerääjän välillä on muodostettu. Yhteystapahtumat kertovat uusista ja päättyneistä tukiasemayhteyksistä ja epäonnistuneista yhteysyrityksistä. Käyttöliittymä kertoo myös paljonko tukiasemayhteyksiä on lisenssin sallimissa rajoissa vielä mahdollista lisätä. Verkon tilasta ja ongelmista voidaan tehdä päätelmiä vertaamalla mittareita ja epäonnistuneita tukiasemien yhteysyrityksiä keskenään.

5.2.2 Teknologiat

Datakerääjien hallintasovellus jaetaan käyttöliittymään, palvelinohjelmaan ja tietokantaan. Hallintasovelluksen tulee olla helposti käytettävä ja päivitettävä ja tukea uusien ominaisuuksien lisäämistä. Käyttöliittymän halutaan toteuttavan organisaation yhtenäinen ”Look & Feel”, jolla itsenäiset käyttöliittymät saadaan näkymään ja tuntumaan loppukäyttäjälle yhtenäisiltä. Olennaista on, että käyttöliittymä piilottaa taustalla olevan mikropalveluarkkitehtuurin kompleksisuuden.

Hallintasovelluksen käyttöliittymässä käytettäväksi JavaScript-kirjastoksi valittiin React (2021). Organisaation yhteisenä käyttöliittymien suunnittelumallina on React-kehitykseen perustuva joustava ja sujuva käyttökokemus verkko- ja palvelusovelluksissa. React-kirjaston avulla kehitetään moderneja verkkopohjaisia sovelluksia, jotka on suunniteltu täyttämään organisaation käyttäjäkokemuksen ja tuotemerkin standardit ja vaatimukset. React-kirjaston avulla käyttöliittymään voidaan toteuttaa selkeä arkkitehtuurimalli, jossa sovelluksen osat on jaettu uudelleenkäytettäviin React-komponentteihin. Käytössä on myös organisaation sisäinen komponenttikirjasto, joka tarjoaa valmiita React-kirjastolla rakennettuja käyttöliittymäkomponentteja ja ratkaisuja näkymien, navigoinnin, datataulukoiden ja lomakkeiden toteuttamiseen.

React-komponentti voi sisältää tietoja sovelluksen tilasta. Komponentin sisältämää tilaa ja tilaa käsitteleviä funktioita voidaan välittää muille komponenteille. Suuren määrän komponentteja sisältävissä sovelluksissa komponentteihin hajautettu tila voi hämärtää sovelluksen selkeyttä ja vaikeuttaa ylläpitoa. React-käyttöliittymän tilanhallintaan otettiin käyttöön Redux (2021), jossa käyttöliittymän tila tallennetaan React-komponenttien ulkopuoliseen varastoon eli Redux-storeen. Storessa olevaa tilaa ei muuteta suoraan vaan

action-tapahtumien avulla. Kun tapahtuma muuttaa sovelluksen tilaa, luodaan tarvittavat näkymät ja komponentit uudelleen. Tapahtumien vaikutus sovelluksen tilaan määritellään reducer-toimintojen avulla. Redux-storen tila välitetään sovelluksen komponenteille käyttämällä React Redux Hooks API -rajapintaa (2021), joka tarjoaa toimintoja Redux-storen tilan hallintaan.

Uuden hallintasovelluksen palvelinohjelman toteuttamiseen valittiin Spring Boot -ohjelmistokehys (2021), jolla voidaan rakentaa itsenäisiä Java-sovelluksia. Ohjelmiston useiden web-sovellusten toteutuksissa on käytössä Spring Boot, joten siitä on ennestään kokemusta. Myös Node.js on käytössä osassa mikropalveluita. Spring Boot on hyvin tuettu ja laajasti käytössä oleva ohjelmistokehys, joka tukee SQL-pohjaisia tietokantoja ja integraatioita muihin ohjelmiin. Spring Boot sisältää oletuksena Apache Tomcat -web-palvelimen. Spring Boot -sovelluksen saa myös hyvin integroitua tukipalveluiden esimerkiksi metriikan seurantaan käytettävän Prometheus-palvelun (2021) kanssa. Spring Boot -projekti luo konfiguraatiotiedoston projektiin valittujen ohjelmistokirjastojen pohjalta.

Hallintasovelluksen tarvitseman datan tallennukseen otettiin käyttöön PostgreSQL-relaatiotietokanta. Tietokanta on oma mikropalvelunsa, jota vain hallintasovellus käyttää. Analysointi- ja valvontaohjelmiston mikropalveluilla on omat tietokantansa, mikä tukee palveluiden itsenäistä kehitystä ja käyttöönottoa. PostgreSQL on ohjelmistossa ennestään käytössä osassa mikropalveluita. Relaatiotietokannan käsittelyyn ja olioiden tallentamiseen relaatiotietokantaan käytetään JPA-rajapintaa (engl. Java Persistence API) ja JPA:n toteuttavaa Hibernate-kirjastoa, joka helpottaa tietokantakyselyiden tekemistä suoraan Spring Boot -ohjelmakoodista.

5.2.3 Turvallisuus

Ohjelmiston mikropalvelut toimivat hajautetussa palvelinympäristössä, jossa on tärkeää todentaa vuorovaikutuksen osapuolet ja salata osapuolten välinen liikenne. Datakerääjien REST- ja WebSocket-rajapintojen suojaus perustuu HTTPS- ja TLS-pohjaiseen käyttäjien todennukseen. Mikropalvelut voivat olla yhteydessä toisiinsa sertifikaattien avulla, jotka sama CA-sertifikaatti on allekirjoittanut.

Ohjelmiston kehityksessä tavoitellaan sitä, että kaikki komponentit käyttävät sertifikaatteja samalla tavalla konfiguroinnin, käyttöönoton ja ylläpidon yksinkertaistamiseksi ja helpottamiseksi. Datakerääjien hallintasovellus lataa salausavaimen ja sertifikaatit levyhakemistosta ja luo dynaamisesti niiden avulla Keystoren ja Truststoren, joita se käyttää suojatussa viestinnässä datakerääjien kanssa.

5.2.4 Toteutus

Hallintasovelluksen Spring Boot -projekti luotiin Spring Initializr -sivun avulla, jossa projektin käyttämäksi ohjelmointikieleksi valittiin Java ja käännöksessä käytettäväksi työkaluksi Maven (2021). Mavenia käytetään valmiiden ohjelmointikirjastojen hakemiseen ja projektin hallintaan. Spring Boot -palvelinohjelman toiminnallisuudet jaettiin omiksi luokikseen, uudelleen käytettäviin osiin ja metodeihin uusien ominaisuuksien, muutosten ja koodin ylläpidon helpottamiseksi sekä turhan koodin toiston välttämiseksi.

Spring Boot -sovellus kommunikoi datakerääjien tarjoamien REST-pohjaisten ja WebSocket-pohjaisten rajapintojen kanssa sekä hoitaa datan käsittelyä ja tallentamista tietokantaan. Spring Boot -sovellukseen toteutettiin REST/JSON-pohjainen pyyntöihin reagoiva API-rajapinta, jonka päätepisteiden kautta käyttöliittymä voi pyytää datakerääjiltä kerättyjä ja käsiteltyjä tietoja sekä välittää komentoja datakerääjille. API-rajapinnan jokaiselle päätepisteelle toteutettiin Controller-toiminto, joka hoitaa kyseiseen päätepisteeseen tulevat pyynnöt ja välittää ne oikealle palvelulle hoidettavaksi Spring Boot -sovelluksen sisällä.

Hallintasovelluksen käyttöliittymä on React-kirjastolla toteutettu dynaaminen single page -sovellus, joka kommunikoi taustalla toimivan Spring Boot -sovelluksen API-rajapinnan kanssa. Hallintasovelluksen käyttöliittymän ja palvelinohjelman rajapinnan välisessä viestinnässä käytetään JSON-merkintätapaa (2021). Käyttöliittymän näkymät muodostetaan dynaamisesti JSON-muotoisen datan perusteella, joka saadaan API-rajapinnan päätepisteisiin tehtyjen HTTP-pyyntöjen vastauksina. Käyttöliittymän toiminnallisuus jaettiin React-koodissa pienempiin, helpommin hallittaviin ja uudelleenkäytettäviin komponentteihin, joilla edistetään sovelluksen ylläpidettävyyttä, modulaarisuutta ja vältetään turhaa koodin toistamista. React-koodiin luotiin kansiorakenne, johon eri komponentit jaoteltiin, ja josta ne ovat selkeämmin löydettävissä. Kun käyttöliittymä saa API-rajapinnasta pyytämänsä tiedot, ne tallennetaan Redux-storen tilaan. Kun storen tila muuttuu, luodaan tarvittavat näkymät uudelleen.

Hallintasovelluksen React-käyttöliittymää ja Spring Boot -sovellusta kehitetään omissa versionhallintavarastoissaan. Kehitysvaiheessa React-koodia suoritetaan kehitysversiona, jossa sovellus antaa havainnollisia virheilmoituksia ja päivittää koodiin tehdyt muutokset reaaliaikaisesti selaimeen. Testaus- ja tuotantokäyttöä varten React-käyttöliittymästä tehdään tuotantoversio, jonka build-hakemistoon lähdekooditiedostoista pakataan staattiset HTML-, CSS- ja JavaScript-tiedostot. Hallintasovelluksen toteutuksessa käyttöliittymän tuotantoversion build-hakemiston sisältö kopioidaan Spring Boot -sovelluksen hakemistoon src/main/resources/static, jonka sisältämä index.html tarjotaan käyttäjälle selaimessa. Hallintasovelluksen päätepisteisiin /api tai /config tulevat pyynnöt ohjataan Controller-toimintojen hoidettaviksi. Muut pyynnöt ohjataan käyttöliittymään. Käyttöliittymän staattiset resurssit tarjotaan käyttäjälle Tomcat-palvelimen kautta.

Hallintasovellus pakattiin JAR-pakettiin, joka luotiin Maven-työkalun avulla. Tässä toteutuksessa hallintasovelluksen React-käyttöliittymän tarvitsemat tiedostot sisällytetään Spring Boot -sovelluksesta luotavaan JAR-pakettiin ja Docker-konttiin. Vaihtoehtoisessa toteutuksessa React-käyttöliittymä voidaan tarjota erillisen web-palvelimen esimerkiksi NGINX-palvelimen (2021) kautta käyttäjälle, jolloin käyttöliittymä toimii omassa itsenäisessä kontissa. Osa käyttöliittymistä on toteutettu tällä tavalla niin, että käyttöliittymä ja palvelinohjelma ovat omissa konteissaan. Ylimääräisen välityspalvelimen ja tässä tapauksessa turhan ketjuttamisen välttämiseksi hallintasovelluksen käyttöliittymä tarjotaan Spring Boot -sovelluksen kautta. Tarpeiden muuttuessa React-käyttöliittymä on mahdollista toimittaa itsenäisesti omassa kontissaan.

Yksikkötestit (engl. unit test) toteutetaan osaksi sovellusta ja uusien toiminnallisuuksien toteuttaminen on perusteltua aloittaa yksikkötestien toteuttamisella. Yksikkötestauksen tavoitteena on havaita muutosten aiheuttamat mahdolliset virheet ohjelman toiminnassa ennen käyttöönottoa. Yksikkötesteillä varmistetaan esimerkiksi, ettei uusi ominaisuus riko jo olemassa olevaa toiminnallisuutta. Yksikkötestit ajetaan jokaisen muutoksen yhteydessä.

Hallintasovelluksen käyttöliittymä tarjoaa käyttäjälle nopealla vilkaisulla tiedon datakerääjien tilasta ja hälytyksistä, joten se helpottaa hajautettujen datakerääjäpalvelimien saatavuutta ja seurantaa. Käyttöliittymän kautta nähdään datakerääjät, niiden liitännäisohjelmat, hälytykset, olennaiset mittarit ja tilastolaskurit sekä tukiasemayhteydet. Käyttöliittymän etusivun valvontanäkymän kautta voidaan siirtyä tarkastelemaan yksityiskohtaisempia tietoja.

5.3 Tukipalvelut

Kaikkea datakerääjien hallintasovelluksen tarvitsemaa toiminnallisuutta ei toteuteta uuteen web-sovellukseen, vaan mikropalveluiden periaatteiden mukaisesti osa toiminnoista saadaan muiden mikropalveluiden avulla, esimerkiksi käyttäjien hallinta ja todentaminen. Hallintasovellus kommunikoi ja tekee yhteistyötä datakerääjien lisäksi myös muiden mikropalveluiden kanssa.

5.3.1 Verkkoliikenteen välitys

Kohdeohjelmiston käyttöliittymien ja palvelinsovellusten välissä toimii Kong-välityspalvelin (2021), joka ohjaa verkkoliikennettä ja yhteyksiä käyttöliittymiin ja muihin järjestelmän mikropalveluihin. Kong on skaalautuva, avoimen lähdekoodin API-kerros, API Gateway, joka tarjoaa abstraktikerroksen asiakkaiden ja mikropalveluiden väliseen viestintään. Kong toimii yhdyskäytävänä mikropalveluiden pyynnöille ja tarjoaa ratkaisuja kuormituksen tasapainottamiseen. Kong hoitaa yleistä toiminnallisuutta, jolloin muut palvelut voivat keskittyä oman vastuualueensa hoitamiseen.

Hallintasovellus tarjotaan Kongin välityksellä käyttäjälle. Kong-palveluun määritellään reitit (engl. routes) ja säännöt (engl. rules) vastaamaan asiakkailta tulevia pyyntöjä. Kong välittää reitille osuvan pyynnön siihen liittyvälle palvelulle. Hallintasovelluksen React-käyttöliittymästä tehtävät pyynnöt API-rajapintaan kulkevat Kongin kautta.

5.3.2 Käyttäjähallinta

Kohdeohjelmiston käyttäjähallinta toteutetaan Keycloak-ohjelman (2021) avulla, joka on avoimen lähdekoodin ratkaisu käyttäjien todentamiseen ja pääsyn hallintaan. Keycloakin avulla suojataan käyttöliittymäsovelluksia ja palveluita. Keycloak-palvelu toimii omassa Docker-kontissaan ja muut mikropalvelut kommunikoivat sen tarjoaman rajapinnan kanssa. Hallintasovellus käyttää Keycloak-palvelua käyttäjähallintaan ja API-rajapinnan suojaamiseen. Tunnistamaton käyttäjä ohjataan ensin Keycloakin kirjautumissivulle. Onnistuneen kirjautumisen jälkeen käyttäjä ohjataan hallintasovelluksen käyttöliittymään valtuutustunnuksen kanssa (engl. authorization token). Kun käyttöliittymä tekee pyyntöjä taustalla olevaan API-rajapintaan, pyynnön mukana on oltava voimassa oleva valtuutustunnus, jonka voimassaolo tarkistetaan Keycloak-palvelusta. Voimassaolevalla valtuutustunnuksella varmistetaan, että käyttäjällä on oikeus saada tietoja API-rajapinnan päätepisteistä.

Keycloak-palveluun luodaan Keycloak Realm (alue), joka hallinnoi käyttäjien joukkoa, tunnistetietoja (engl. credentials), rooleja ja ryhmiä. Käyttäjä kuuluu ja kirjautuu realmiin. Realmit ovat eristettyjä toisistaan ja voivat hallinnoida ja todentaa vain omia käyttäjiään. Keycloak-palveluun luodaan myös asiakkaita (engl. client), jotka pyytävät Keycloakia todentamaan käyttäjän. Usein asiakkaat ovat sovelluksia tai palveluita, jotka käyttävät Keycloakia itsensä suojaamiseen ja sisäänkirjautumiseen. Keycloak-palveluun luodaan lisäksi roolit ja ryhmät sekä käyttäjä. Ryhmälle voidaan lisätä halutut roolit. Käyttäjä voidaan lisätä kyseiseen ryhmään, jolloin käyttäjä perii ryhmässä olevat roolit.

Hallintasovelluksen React-käyttöliittymälle luotiin Keycloak-palveluun julkinen asiakas (engl. public client), joka sallii uudelleen ohjauksen kirjautumissivun kautta käyttöliittymään. Taustalla toimivan Spring Boot -sovelluksen API-rajapintaa varten Keycloakiin luotiin luottamuksellinen, kahdenkeskinen asiakas (engl. confidental client), jolla on kommunikointia varten Keycloak-salatunnus (engl. secret). Spring Boot -sovelluksen API-rajapinnan salaamista varten Keycloak-palveluun luotiin kaksi roolia, ylläpitäjärooli (engl. admin role) ja käyttäjärooli (engl. user role). API-rajapinnan tietyt päätepisteet sallitaan vain käyttäjälle, jolla on ylläpitäjärooli. Esimerkiksi päätepisteet, joilla voidaan muuttaa tietokannassa olevia tietoja tai lähettää komentoja datakerääjille, on sallittuja vain ylläpitäjäroolin sisältäville käyttäjille.

Käyttäjähallintaan ja käyttäjien todentamiseen käytettävä itsenäinen mikropalvelu mahdollistaa sen, että kyseiset toiminnallisuudet voidaan pitää erillään muista ohjelmiston toiminnallisuuksista ja muiden palveluiden yhteisessä käytössä. Muiden mikropalveluiden tulee tietää Keycloak-mikropalvelun sijainti, jotta kommunikointi suoraan sen tarjoaman rajapinnan kautta onnistuu. Hallintasovellukselle annetaan Keycloak-palvelun osoite ja tarvittavat parametrit ympäristömuuttujina.

5.3.3 Metriikkaseuranta

Hallintasovellus integroidaan mittareiden (metriikan) valvontajärjestelmä Prometheuksen (2021) kanssa sekä graafisia näkymiä tarjoavan Grafanan (2021) kanssa. Prometheus on SoundCloudin rakentama avoimen lähdekoodin seurantajärjestelmä sovellusten monitorointiin ja hälytyksiin. Prometheus hakee määritellyillä aikaväleillä mittaustietoja sovellusten tarjoamien rajapintojen kautta. Prometheus tarjoaa mittaritietojen tallentamiseen aikasarjatietokantaa (engl. time-series database). Prometheuksen yksinkertaisen käyttöliittymän avulla voidaan visualisoida, kysellä ja seurata hallintasovelluksen tuottamia mittareita. (Singh, 2018)

Spring Boot Actuator -moduuli auttaa seuraamaan hallintasovellusta tarjoamalla ratkaisuja seurantaan ja mittareiden keräämiseen. Spring Boot käyttää Micrometer-sovellusta mittareiden integroimisessa ulkoisiin valvontajärjestelmiin ja tukee muun muassa Prometheuksen käyttöä. Kun micrometer-registry-prometheus ja spring-boot-starter-actuator -riippuvuudet (engl. dependency) lisätään Spring Boot -sovelluksen konfiguraatiotiedostoon (pom.xml), sovellus määrittää sen jälkeen automaattisesti PrometheusMeterRegistry:n ja CollectorRegistry:n keräämään ja viemään mittarit Prometheuksen ymmärtämässä muodossa. Kaikki sovelluksen mittarit, metriikka on saatavilla aktuaattorin päätepisteessä /prometheus. Prometheus voi käyttää tätä päätepistettä metriikan säännölliseen hakemiseen. Prometheus- ja Grafana-ohjelmien avulla voidaan visualisoida ja seurata Spring Boot -sovelluksen tuottamia tietoja. Tietoja voidaan käyttää apua ohjelmiston mikropalveluiden kehittämisessä ja sisäisessä seurannassa.

5.3.4 Palvelurekisteri

Palveluiden löydettävyys on keskeinen mikropalveluihin liittyvä haaste hajautetussa ympäristössä. Uuden datakerääjien hallintasovelluksen on tiedettävä datakerääjien sijainti eli niiden etähallintarajapintojen osoitteet, joiden kautta se voi löytää datakerääjät ja kommunikoida niiden kanssa. Datakerääjät sijaitsevat eri palvelimilla kuin niiden hallintasovellus eivätkä ne nykytoteutuksessa ilmoita itseään ja sijaintiaan. Datakerääjien toiminnallisuus ja rajapintojen päätepisteet on hyvin dokumentoitu, mikä auttaa ja tukee niiden testaamista ja myös hallintasovelluksen kehittämistä.

Käyttäjä voi antaa hallintasovellukselle datakerääjien tavoittamiseen tarvittavat tiedot joko käyttöliittymän lomakkeella tai lataamalla käyttöliittymään tiedot sisältävän CSV-tiedoston. Tietojen syöttäminen käyttöliittymän kautta vaatii käyttäjältä manuaalista työtä ja käyttäjän tiedossa tulee tällöin olla datakerääjien sijainti. Datakerääjien automaattisessa skaalauksessa tarpeen ja kuormituksen mukaan manuaalinen konfigurointi käyttöliittymän kautta käyttäjän tekemänä ei ole tarkoituksenmukaista. Muutaman datakerääjän manuaalinen hallinta on vielä mahdollista, mutta kymmenien tai satojen datakerääjien manuaalinen tietojen lisäys tai muokkaus käyttöliittymän kautta on haasteellista ellei mahdotonta.

Ohjelmiston mikropalveluiden ja erityisesti datakerääjien automaattista skaalausta ja käyttöönottoa, saatavuutta ja seurattavuutta voidaan parantaa palvelurekisterinä toimivan Consulin (2021) avulla, jonka tiedossa on ajossa olevien mikropalveluiden instanssit. Consulia voidaan käyttää myös ohjelmiston tilallisen tiedon hallintaan. Mikropalveluiden saatavuutta ja vikasietoisuutta voidaan parantaa hyödyntämällä palvelurekisterin mahdollisuuksia. Consulin tarjoaman rajapinnan avulla mikropalvelut voivat löytää toisensa tallentamalla sijaintitietonsa kuten IP-osoitteet keskitettyyn rekisteriin.

Consulin tarjoaman rajapinnan kautta uusi datakerääjien hallintasovellus voisi pyytää tiedot ajossa olevista datakerääjistä. Hallintasovelluksen kautta voitaisiin myös päivittää datakerääjien tiedot Consuliin. Tämä parantaisi käyttäjäkokemusta ja vähentäisi käyttäjältä tarvittavaa manuaalista työtä. Palvelurekisteri voisi myös tarjota datakeräilijöille keskitetyn konfiguraatiohallinnan, josta datakerääjät voisivat hakea konfiguraatiotiedot.

5.4 Docker ja Kubernetes

Ohjelmiston muiden mikropalveluiden tapaan uusi hallintasovellus toimitetaan Docker-kontissa, jonka avulla sovellus voidaan ottaa käyttöön erilaisissa testaus- ja tuotantoympäristöissä. Hallintasovellukselle luotiin Dockerfile-tiedosto, jossa määriteltiin Docker-kontin sisältö. Docker-konttiin sisällytetään sovelluksesta luotu JAR-tiedosto. Docker-kuvan pohjana käytetään ohjelmiston mikropalveluille luotua yhteistä pohjakuvaa (engl. base image), jota päivitetään ja tarkistetaan haavoittuvuuksien varalta. Dockerfilen avulla rakennetaan Docker-kuva, josta luodaan Docker-kontti. Hallintasovellukselle rakennettiin Gitlab-versionhallintaan CI/CD-putki, joka käynnistyy, kun uusi muutos lisätään versionhallintaan. Onnistuneen CI/CD-putken vaiheiden seurauksena on Docker-kuva, joka löytyy organisaation artefaktivarastosta. Hallintasovelluksen käytössä oleva PostgreSQL-tietokanta toimii omassa Docker-kontissaan.

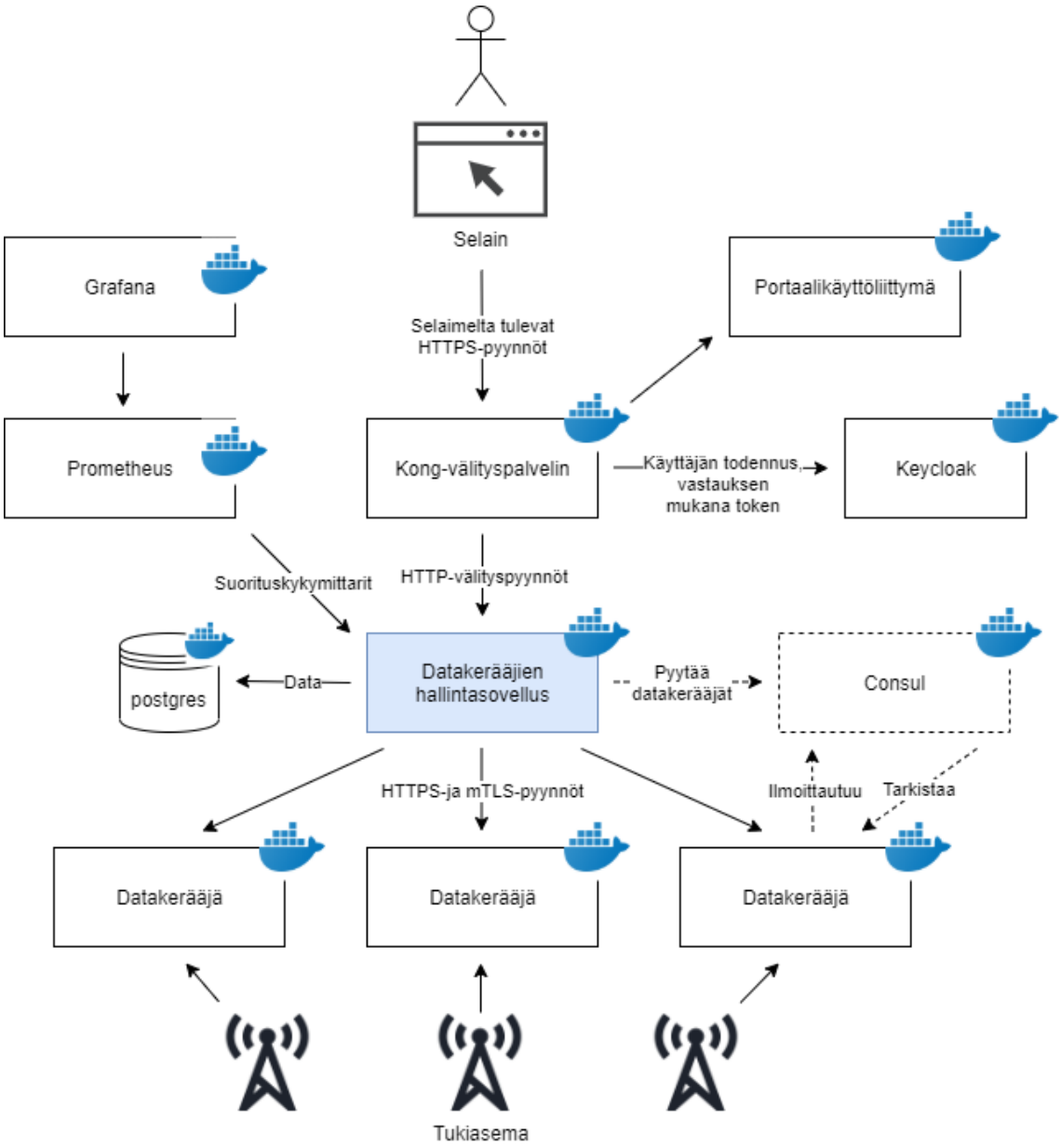
Hallintasovelluksen käyttöä ja hallintaa helpottamaan otettiin käyttöön Docker Compose, jonka yaml-tiedostoon määritettiin hallintasovelluksen tarvitsemat ympäristömuuttujat ja tiedostosijainnit. Docker Composen ympäristömuuttujien avulla hallintasovelluk-

selle annetaan sen tarvitsemat tiedot, kuten PostgreSQL-tietokantayhteys, käyttäjähallinnasta ja käyttäjän todentamisesta vastavan mikropalvelun osoite sekä datakerääjien kanssa tapahtuvan viestinnän salaukseen tarvittavien salausavaimen ja sertifikaattien sijainti.

Ohjelmiston konttien hallintaan otetaan käyttöön Kubernetes-orkestrointijärjestelmä (2021). Uutta hallintasovellusta testataan konseptitodistuksena (engl. proof-of-concept, POC) palvelinklusterissa käyttämällä Rancher-hallintajärjestelmää (2021). Myös käytössä oleva Helm-hallintajärjestelmä (2021) auttaa Kubernetes-konttien hallinnassa. Helm-kaavioilla määritellään, asennetaan ja päivitetään käytetään Kubernetes-sovellusta.

5.5 Kommunikaatiokaavio

Kuvassa 12 esitetään kommunikaatiokaavio, joka kuvaa kohdeohjelmiston mikropalveluiden ja prosessien välistä vuorovaikutusta ja miten mikropalveluiden yhteistyö perustuu palveluiden välisiin kytkentöihin. Kommunikaatiokaaviossa on kuvattu ne mikropalvelut, joiden kanssa hallintasovellus tekee yhteistyötä. Analysointi- ja valvontaohjelmistoon kuuluu kokonaisuudessaan enemmän vuorovaikutuksessa olevia komponentteja kuin kuvassa näkyy.



Kuva 12. Mikropalveluiden kommunikointi

5.6 Arviointi

Tutkielmassa tarkasteltiin case-esimerkkinä web-sovelluksen kehittämistä mikropalveluarkkitehtuurissa osana matkapuhelinverkon analysointi- ja valvontaohjelmistoa. Ohjelmistoa kehitetään mikropalveluina, jotka vastaavat tietystä osasta toimintaa, ja jotka toimitetaan ja otetaan käyttöön Docker-konteissa. Tässä luvussa arvioidaan miten ohjelmisto ja uusi datakerääjien hallintasovellus toteuttavat mikropalveluarkkitehtuurin keskeiset laatutekijät, joita ovat skaalautuvuus, suorituskyky, saatavuus, seurattavuus, turvallisuus ja testattavuus.

Skaalautuvuuteen ja suorituskykyyn vaikuttaa miten toiminnallisuudet jaetaan palveluihin eli miten palvelut rajataan ja kuinka niiden välinen kommunikointi toteutetaan. Skaalautuvuuteen vaikuttaa, miten ohjelmisto toimii käyttäjäkuorman tai käsiteltävän datan määrän kasvaessa. Vertikaalista hajottamista toteutetaan hajottamalla ohjelmisto sopiviksi mikropalveluiksi korkeamman ja itsenäisemmän skaalautuvuuden saavuttamiseksi. Ohjelmiston skaalautuvuutta ja resurssien tehokasta käyttöä parannetaan, kun vastuut, toiminnot ja data on ohjelmistossa erotettu omiin mikropalveluihin. Tällöin voidaan skaalata eli kopioida tarvittavia mikropalveluita koko ohjelmiston sijaan. Docker-konttien käytöllä myös edistetään ohjelmiston skaalautuvuutta.

Horisontaalisen skaalautuvuuden haasteena on automaattisesti ratkaista kuinka monta instanssia tietystä mikropalvelusta on oltava käynnissä halutun palvelun laadun saavuttamiseksi. Horisontaalinen skaalautuvuus ja horisontaalisen kopioinnin toteuttaminen on ohjelmiston haasteena, koska nykytilanteessa ei hyödynnetä automaattista skaalausta eli instanssien automaattista lisäystä ja poistamista palveluun kohdistuvan tarpeen mukaan. Ohjelmiston kannalta datakerääjien skaalautuvuus ja niiden ajossa olevien instanssien automaattinen lisäys ja poistaminen kuormituksen ja vikatilanteiden mukaan on olennaista. Nykytilanteessa datakerääjien käyttöönotto, konfigurointi ja datakerääjien instanssien horisontaalinen skaalaus tapahtuu manuaalisesti. Erillinen asennusohjelma helpottaa ja yhtenäistää kuitenkin käyttöönottoa ja konfigurointia.

Kuberneteksen käyttöönotolla parannetaan datakerääjien skaalautuvuutta ja mikropalveluarkkitehtuurin mahdollisuuksien hyödyntämistä. Kuberneteksen tehtävänä on pitää ohjelmisto automaattisesti toimintakunnossa. Yksittäisen palvelun kaaduttua Kubernetes huolehtii uuden palvelun instanssin käynnistämisestä automaattisesti.

Ohjelmistoa käytetään erilaisiin käyttötarkoituksiin ja erilaisissa ympäristöissä, joissa kaikkia ohjelmistoon kuuluvia mikropalveluita ja kontteja ei aina oteta käyttöön. Itsenäiset ja konteissa toimitettavat mikropalvelut mahdollistavat erilaisten kokoonpanojen käyttöönoton ja konfiguroinnin, mikä lisää ohjelmiston skaalautuvuutta ja joustavuutta. Datan tallentamista tarvitsevilla mikropalveluilla on käytössään omat itsenäiset tietokannat, millä tuetaan myös palveluiden itsenäisyyttä ja datan turvallisuutta.

Suorituskykyyn vaikuttaa kuinka nopeasti ohjelmisto vastaa palvelupyyntöihin ja suoriutuu sille asetetuista vaatimuksista. Mikropalveluarkkitehtuuri on mahdollistanut kehittäjille vapauden valita komponenteille sopivat teknologiat, millä edistetään myös komponenttien suorituskykyä. Raskaampaa laskentaa suorittavat komponentit toteutetaan eri teknologioilla kuin käyttöliittymäkomponentit. Docker-konttiteknologian käyttö on keskeinen taktiikka suorituskyvyn parantamisessa ja virtualisoinnin toteuttamisessa konttien avulla. Virtualisoinnilla parannetaan hajautettujen sovellusten suorituskykyä. Kevyiden Docker-konttien avulla mikropalveluiden instansseja voidaan luoda ja ajaa enemmän, mikä johtaa korkeampaan resurssien käyttöasteeseen. Suorituskykyä edistetään osassa

ohjelmistoa myös kuormituksen tasauksen avulla, jolla jaetaan palveluun kohdistuvaa liikennettä tasaisemmin ajossa oleville instansseille.

Saatavuuden kannalta datakerääjät ovat kriittisiä mikropalveluita, joiden saatavuus, oikea toiminta ja vikasietoisuus on olennaista olla kunnossa. Datakerääjien häiriöt ja virhetilanteet vaikuttavat ohjelmiston toimivuuteen, koska ne keräävät, suodattavat ja välittävät tukiasemien dataa koko ohjelmiston ja kolmansien osapuolten käyttöön. Loppukäyttäjälle, esimerkiksi matkapuhelinoperaattorille, datakerääjien häiriöt ja virhetilanteet aiheuttavat ongelmia. Saatavuutta voidaan parantaa palvelurekisterinä toimivan Consulin (2021) avulla, jonka tiedossa on ajossa olevien mikropalveluiden instanssit. Consulia voidaan käyttää myös ohjelmiston tilallisen tiedon hallintaan. Consulin avulla uusi datakerääjien hallintasovellus voi hakea ja päivittää datakerääjien tietoja ilman käyttäjän manuaalista työtä.

Saatavuuden näkökulmasta uuden datakerääjien hallintasovelluksen häiriöt ja virhetilanteet eivät ole koko ohjelmiston kannalta erityisen kriittisiä. Uusi hallintasovellus on tarkoitettu parantamaan ja tukemaan datakerääjien seurantaa ja käyttäjäkokemusta käyttöliittymän avulla. Datakerääjät eivät kuitenkaan tarvitse hallintasovellusta toimiakseen. Hallintasovelluksen ja muiden koko ohjelmistoon kuuluvien itsenäisten käyttöliittymien saatavuutta voidaan tarkastella myös niin, että hallintasovellus ja muut itsenäiset käyttöliittymät integroidaan portaalikäyttöliittymään, jonka valikon kautta käyttäjä pääsee siirtymään useampaan käyttöliittymään. Pääsy itsenäisiin käyttöliittymiin ja datan siirto näiden välillä on mahdollista ohjelmistoon toteutetun portaalikäyttöliittymän kautta.

Ohjelmiston mikropalveluiden saatavuutta parannetaan ottamalla käyttöön useamman palvelimen Kubernetes-klusteri, joka tarjoaa mahdollisuuksia vikasietoiselle arkkitehtuurille. Esimerkiksi yhden palvelimen virhetilanteessa voidaan sillä olevat mikropalvelut siirtää automaattisesti toiselle toimivalle palvelimelle.

Seurattavuus voi koskea infrastruktuuria, sovelluksia tai ympäristöä ja sillä voidaan edistää skaalautuvuutta, suorituskykyä ja saatavuutta. Ohjelmiston mikropalvelut tuottavat niiden toiminnasta ja palvelupyynnöistä lokeja, joita käytetään suorituskäyttäytymisen ymmärtämiseen ja vikojen selvittämiseen. Uusi datakerääjien hallintasovellus tuottaa myös lokeja sen käsittelemistä palvelupyynnöistä ja virhetilanteista, joiden avulla saadaan näkyvyyttä sovelluksen toimintaan. Mikropalveluiden reaaliaikaista valvontaa toteutetaan metriikkaraportoinnin avulla, jossa sovellusten tuottamaa metriikkaa seurataan Prometheus-ohjelman (2021) ja Grafana-ohjelman (2021) avulla. Prometheus-ohjelman avulla saadaan hallintasovelluksesta määritellyillä aikaväleillä mittaustietoja, joita voidaan tarkastella Grafanaan rakennettujen näkymien kautta.

Ohjelmiston turvallisuus otetaan huomioon todentamalla palveluiden välisen kommunikoinnin osapuolet ja salaamalla osapuolten välinen liikenne HTTPS-protokollan ja

kahdensuuntaisen TLS-suojauksen avulla. Mikropalvelut kommunikoivat asiakasvarmenteiden (sertifikaattien) avulla. Ohjelmiston komponentit käyttävät mahdollisuuksien mukaan yhteistä Docker-pohjakuvaa, joka pidetään ajan tasalla haavoittuvuusskannausten ja tietoturvapäivitysten avulla. Mikropalveluissa kuten uudessa hallintasovelluksessa käytetyt kirjastot pidetään ajan tasalla haavoittuvuuksien välttämiseksi. Käyttäjien hallinta ja todentaminen sekä API-rajapinnan päätepisteiden suojaus toteutetaan Keycloak-ohjelman (2021) avulla, jolloin estetään valtuuttamattomien käyttäjien pääsy järjestelmään.

Docker-konteissa käyttöönotettavat ja toimitettavat mikropalvelut mahdollistavat sen, että muutosten ja uusien versioiden testaaminen on nopeaa. Mikropalveluiden yksikkötesteillä ja jatkuvalla integraatiolla varmistetaan, että muutokset eivät riko toiminnallisuutta ja muutokset integroituvat toimivasti olemassa olevaan toteutukseen. Mikropalveluiden kokonaisuuden ja palveluiden välisen vuorovaikutuksen end-to-end-testaus on monimutkaista ja edellyttää testauksen automatisointia ja DevOps-menetelmiä kuten jatkuvaa integrointia. Mikropalveluihin tehtävät muutokset ajetaan Gitlab-versionhallintavarastoon rakennetun CI/CD-julkaisuputken läpi, jonka seurauksena sovelluksesta julkaistaan uusi Docker-kuva. Docker-kuva on artefaktivarastosta ladattavissa käyttöön.

Mikropalveluiden testaus on nykytilanteessa osittain manuaalisen testauksen varassa, joten testauksen automatisoinnin ja CI/CD-prosessin kehittäminen on tulevaisuuden haasteena. Uuden datakerääjien hallintasovelluksen toimivuutta testataan tukiasemien ja tukiasemien liikennettä simuloivien ohjelmien avulla. Testiympäristöissä hallintasovellus on toiminut toivotulla tavalla. Uuden hallintasovelluksen kehittämisessä otetaan huomioon DevOps-menetelmien korostama jatkuvan palautteen saaminen järjestelmällä kehitysvaiheessa demoja palautteiden saamiseksi. Koko ohjelmiston testattavuutta parannetaan myös rajapintojen dokumentoinnilla. Kuten luvussa neljä todettiin, testien toteuttamiseksi on tunnettava rajapintojen kuvaukset, syntaksi ja rajapinnassa liikkuvien muuttujien tyypit.

Kokonaisuudessaan ohjelmisto on laajamittainen monista komponenteista koostuva järjestelmä, jossa on paljon osia, niiden välistä kompleksista vuorovaikutusta ja liikkuvaa dataa. Ohjelmiston kehittäminen ja toimittaminen vaatii kymmenien ihmisten työpanosta, yhteistyötä, DevOps-menetelmiä ja mikropalveluarkkitehtuurin laatutekijöihin liittyviä taktiikoita hajautetun ohjelmiston kompleksisuuden hallinnassa. Tulevaisuuden mielenkiintoisena kehitys- ja tutkimuskohteena on Kubernetes-orkestrointialustan käyttöönotto ja konttien automatisoitu hallinta modernissa pilviympäristössä, jossa hyödynnetään konttien automaattista skaalausta. Mikropalveluiden käyttöönotosta ja hallinnasta Kubernetes-ympäristössä saisi oman mielenkiintoisen tutkimuksen.

6 Yhteenveto

Palvelukeskeiset arkkitehtuurit ja mikropalvelut jakavat sovelluksen pieniin, löyhästi kytkettyihin ja keskenään kommunikoiviin palveluihin, joilla tavoitellaan nopeaa kehitystä, palveluiden itsenäistä käyttöönottoa sekä jatkuvassa muutoksessa olevien sovellusten jatkuvaa toimitusta. Kehitettävän ohjelmiston vaatimusten tunnistaminen ja ymmärtäminen helpottaa mikropalveluiden toteuttamista ja hallintaa. Hajautettujen palveluiden taustalla oleva kompleksisuus, lukuisat palvelut ja niiden väliset yhteydet saattavat hämärtää järjestelmän ongelmien havaittavuutta, ylläpitoa ja kokonaisuuden hallintaa. Mikropalveluarkkitehtuuri ei ole aina sopivin ratkaisu sovellusten ja ohjelmistojen toteuttamiseen. Se vaatii huolellista suunnittelua ja järjestelmän vaatimusten tuntemista, jotta toiminta voidaan hajauttaa toimiviin palveluihin ja rajapintoihin. Mikropalveluarkkitehtuuri tarvitsee DevOps-menetelmiä kuten jatkuvaa integraatiota varmistamaan, että uusi koodi integroituu toimivasti olemassa olevaan toteutukseen. Jatkuva toimitus auttaa jatkuvan palautteen saamista.

Ohjelmistokehityksen tavoitteena on tuoda arvoa käyttäjälle ja liiketoiminnalle. Sovelluksen loppukäyttäjän ei tarvitse olla tietoinen taustalla olevien hajautettujen palveluiden määrästä tai kompleksisuudesta. Verkossa olevan sisällön, resurssin tai palvelun löytäminen on web-sovellusympäristöön kuuluva keskeinen periaate, joka on olennainen haaste myös mikropalveluarkkitehtuurissa palveluiden hajautetun luonteen vuoksi. Kompleksisuutta pyritään hallitsemaan tunnistamalla onnistuneen mikropalveluarkkitehtuurin laatutekijät ja niihin liittyvät taktiikat mikropalveluiden toteuttamisessa. Laatutekijät kuten skaalautuvuus, suorituskyky, saatavuus, seurattavuus, turvallisuus ja testattavuus sekä taktiikat kuten konttiteknologian hyödyntäminen, horisontaalinen kopiointi, palvelurekisteri, testiautomaatio, API-rajapintojen dokumentointi sekä lokien ja metriikan seuranta auttavat toimivan mikropalveluarkkitehtuurin toteuttamisessa ja hallitsemisessa.

Kevyet ja ketterät kontit yhdistettynä mikropalveluarkkitehtuurin mahdollisuuksiin ja DevOps-kulttuuriin sopivat yhteen, tukevat toisiaan ja mahdollistavat jatkuvan käyttöönoton mukaisen DevOps-sovelluksen. Konttien avulla mikropalvelua skaalataan eli siitä muodostetaan useita instansseja palveluun kohdistuvan tarpeen mukaan. Valikoiva skaalaus tuo joustavuutta ja säästää resursseja. DevOps-käytännöt ja mikropalveluarkkitehtuuri mahdollistavat isojen ongelmien hajottamisen pienemmiksi paloiksi ja niiden käsittelyn poikkitoiminnallisissa tiimeissä, joissa tarvitaan kaikkien ohjelmiston toteutukseen osallistuvien ihmisten toimivaa yhteistyötä ja jatkuvaa kommunikointia.

7 Viiteluettelo

Brandon, A., Sole, M., Huelamo, A., Solans, D., Perez, M., S., & Muntes-Mulero, V. 2020. Graph-based root cause analysis for service-oriented and microservice architectures. *The Journal of systems and software*, 159(1), 110432.

Bui, T. 2015. Analysis of Docker security. *Proceedings of the Aalto University T-110.5291 Seminar on Network Security*.

Consul. 2021. Service mesh solution for service discovery, configuration, and health checking. Saatavilla: https://www.consul.io/. (Haettu 30.4.2021)

CSS. 2021. Cascading Style Sheets. Saatavilla: https://www.w3.org/TR/CSS/#css. (Haettu 30.4.2021)

Docker. 2021. Docker container technologies. Saatavilla: https://www.docker.com/ (Haettu 30.4.2021):

DOM. 2021. Document Object Model. Saatavilla: https://dom.spec.whatwg.org/. (Haettu 30.4.2021)

ECMAScript. 2021. ECMAScript language specification. Saatavilla: https://262.ecma-international.org/5.1/. (Haettu 30.4.2021)

ELK Stack. 2021. Solutions for enterprise search, analytics, data processing and visualizing data. Saatavilla: https://www.elastic.co/what-is/elk-stack. (Haettu 30.4.2021)

ETCD. 2021. A distributed key-value store for storing data that needs to be accessed by a distributed system or cluster of machines. Saatavilla: https://etcd.io. (Haettu 30.4.2021)

Fielding, R., T. 2000. Architectural styles and the design of network-based software architectures. Doctoral Dissertation. University of California, Irvine.

Fielding, R., T. & Taylor, R., N. 2002. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2), 115-150.

Gitlab. 2021. GitLab is the open DevOps platform. Saatavilla: https://about.gitlab.com/. (Haettu 30.4.2021)

Grafana. 2021. Observability dashboard. Saatavilla: https://grafana.com/. (Haettu 30.4.2021)

Gupta, P. 2021. Mutual TLS: Securing microservices in service mesh. Saatavilla: https://thenewstack.io/mutual-tls-microservices-encryption-for-service-mesh/ (Haettu 29.3.2021)

Harms, H., Rogowski, C., & Iacono, L. 2017. Guidelines for adopting frontend architectures and patterns in microservices-based systems. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 902-907.

Helm. 2021. A package manager for Kubernetes. Saatavilla: https://helm.sh/. (Haettu 30.4.2021)

HTML. 2021. Hypertext markup language. Saatavilla: https://html.spec.whatwg.org/. (Haettu 30.4.2021)

HTTP. 2021. Hypertext transfer protocol. Saatavilla: https://tools.ietf.org/html/rfc7235. (Haettu 30.4.2021)

Humble, J. & Farley, D. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

Ibryam, B. & Huss, R. 2019. *Kubernetes Patterns. Reusable Elements for Designing Cloud-Native Applications*. O'Reilly Media, Inc.

Jabbari, R., Ali, N. B., Petersen, K. & Tanveer B. 2016. What is DevOps? A Systematic Mapping Study on Definitions and Practices. *Proceedings of the Scientific Workshop Proceedings of XP2016*, 12, 1-11.

Jira. 2021. Software development tool for project and issue tracking. Saatavilla: https://www.atlassian.com/software/jira. (Haettu 30.4.2021)

JFrog. 2021. Artifact repository. Saatavilla: https://jfrog.com/. (Haettu 30.4.2021)

JSON. 2021. JavaScript object notation is a lightweight data-interchange format. Saatavilla: https://www.json.org/json-en.html. (Haettu 30.4.2021)

Jazayeri, M. 2007. Some trends in web application development. *Future of Software Engineering*, FOSE ’07, 199-213.

Kang, H., Le, M., & Tao, S. 2016. Container and microservice driven design for cloud infrastructure DevOps. *Proceedings of the 2016 IEEE International Conference on Cloud Engineering (IC2E)*, 202-211.

Keycloak. 2021. Open source identity and access management for applications and services. Saatavilla: https://www.keycloak.org/index.html (Haettu 30.4.2021)

Kocher, P., S. 2018. *Microservices and Containers*. Pearson Education, Inc.

Kong. 2021. Service connectivity for modern architectures. Saatavilla: https://konghq.com/. (Haettu 30.4.2021)

Koskimies, K. & Mikkonen T. 2005. *Ohjelmistoarkkitehtuurit*. Talentum.

Kubernetes. 2021. Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services. Saatavilla: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (Haettu 30.4.2021)

Kuuskeri J. & Mikkonen T. 2009. Partitioning web applications between the server and the client. *Proceedings of the 2009 ACM symposium on applied computing*, 647-652.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. 2021. Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131, 1-23.

Lewis, J. & Fowler M. 2014. Microservices: a definition of this new architectural term. Saatavilla: https://martinfowler.com/articles/microservices.html. (Haettu 30.4.2021)

Maven. 2021. Apache Maven is a software management and comprehension tool. Saatavilla: https://maven.apache.org/. (Haettu 30.4.2021)

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

Newman, S. 2015. *Building Microservices – Designing Fine-Grained Systems.* O'Reilly.

Nickoloff, J. 2016. *Docker in Action.* Manning Publications Co.

NGINX. 2021. Open source web server and API gateway. Saatavilla: https://www.nginx.com/. (Haettu 30.4.2021)

Oteyowo T. 2018. DevOps in a scaling environment. Saatavilla: https://medium.com/tech-tajawal/devops-in-a-scaling-environment-9d5416ecb928. (Haettu 30.4.2021)

Prometheus. 2021. An open-source systems monitoring and alerting toolkit. Saatavilla: https://prometheus.io/. (Haettu 30.4.2021)

Rancher. 2021. Enterprise Kubernetes management. Saatavilla: https://rancher.com/ (Haettu 30.4.2021)

React. 2021. A JavaScript library for building user interfaces. Saatavilla: https://reactjs.org/. (Haettu 30.4.2021)

React Redux Hooks. 2021. React Redux Hooks API interface. Saatavilla: https://react-redux.js.org/api/hooks. (Haettu 30.4.2021)

Redux. 2021. A state container for JavaScript applications. Saatavilla: https://redux.js.org/. (Haettu 30.4.2021)

Singh, R. 2018. Spring Boot actuator metrics monitoring with Prometheus and Grafana. Saatavilla: https://www.callicoder.com/spring-boot-actuator-metrics-monitoring-dashboard-prometheus-grafana/. (Haettu 30.4.2021)

Soldani, J., Tamburri, D. A., & Heuvel, W.-J. V. D. 2018. The pains and gains of microservices: a systematic grey literature review. *The Journal of systems and software*, 146, 215-232.

Spring Boot. 2021. Spring Boot for creating stand-alone Spring applications. Saatavilla: https://spring.io/projects/spring-boot. (Haettu 30.4.2021)

Taibi, D., Lenarduzzi V., and Pahl C. 2018. Architectural patterns for microservices: a systematic mapping study. *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 221-232.

URI. 2021. Uniform resource identifier. Saatavilla: https://tools.ietf.org/html/rfc3986. (Haettu 30.4.2021)

Wan, X., Guan, X., Wang, T., Bai, G., & Choi, B. 2018. Application deployment using microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications*, 119, 97-109.

Waseem, M., Liang, P., & Shahin, M. 2020. A systematic mapping study on microservices architecture in DevOps. *The Journal of systems and software*, 170, 30.

XML. 2021. Extensible markup language. Saatavilla: https://www.w3.org/XML/. (Haettu 30.4.2021)