Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma

# DEVOPS-JULKAISUPUTKEN RAKENTAMINEN MIKROPALVELUITA HYÖDYNTÄVÄLLE WEB-SOVELLUKSELLE

Työn tarkastaja: Apulaisprofessori Antti Knutas

# TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT

School of Engineering Science

Tietotekniikan koulutusohjelma

Joonas Ryynänen

**DevOps-julkaisuputken rakentaminen mikropalveluita hyödyntävälle web-sovellukselle**

Kandidaatintyö 2019

39 sivua, 1 kuva, 6 liitettä

Työn tarkastaja: Apulaisprofessori Antti Knutas

Hakusanat: devops, julkaisuputki, mikropalveluarkkitehtuuri

Keywords: devops, deployment pipeline, microservice architecture

Web-sovelluksien tarjoamat toiminnallisuudet paranevat vuosi vuodelta, kun ohjelmistoyritykset pyrkivät tarjoamaan kuluttajille mahdollisimman monipuolisia ratkaisuja. Samalla ohjelmistojen suorittaminen muuttuu raskaammaksi, ja monoliittista arkkitehtuuria käytettäessä kehitystyö on sidottuna tiettyihin teknologioihin. Mikropalveluarkkitehtuurin avulla web-sovelluksesta voidaan tehdä modulaarinen, jolloin tietty ohjelmiston toiminnallisuus voidaan toteuttaa sille parhaiten soveltuvalla teknologialla. Kyseinen arkkitehtuuri on rakenteeltaan monimutkainen, mutta sen tuomat hyödyt esimerkiksi sovelluksen skaalaamiseen liittyen ovat selkeitä. Tämän kandidaatintyön tavoitteena oli suunnitella ja toteuttaa automatisoitu julkaisuputki mikropalvelusovelluksen julkaisemiseen. Aluksi tehtiin vertailua monoliittisen arkkitehtuurin ja mikropalveluarkkitehtuurin välillä, ja tutkittiin miten mikropalveluarkkitehtuuri vaikuttaa sovellusten julkaisemiseen. Sen jälkeen tehtiin katsaus markkinoilla oleviin työkaluihin sekä suunniteltiin julkaisuputki. Suunnitelman pohjalta toteutettiin esimerkki julkaisuputkesta, joka julkaisee yksinkertaisen mikropalvelusovelluksen. Työn merkittävimpänä lopputuloksena voidaan pitää havaintoa siitä, että automatisoidun julkaisuputken avulla säästetään huomattava määrä sovellusten kehittäjien aikaa. Säästetty aika moninkertaistuu sovelluksen koon ja kehitystiimin kasvaessa. Voidaankin sanoa, että mikropalvelusovelluksen julkaisua kannattaa automatisoida mahdollisimman pitkälle. Työ päättyy yhteenvetoon, jossa julkaisuputken kehitysprosessia reflektoidaan.

# ABSTRACT

Lappeenranta-Lahti University of Technology LUT

School of Engineering Science

Degree Programme in Software Engineering

Joonas Ryynänen

**Implementing of a DevOps deployment pipeline for a microservice-based web application**

Bachelor's Thesis 2019

39 pages, 1 figure, 6 appendices

Examiner : Assistant Professor Antti Knutas

Keywords: devops, deployment pipeline, microservice architecture

The features of web applications become better every year, as software companies try to offer as versatile solutions to customers as possible. At the same time running the software becomes heavier, and when using monolithic architecture the development is tied to certain technologies. With microservice architecture the web application is modular and a certain part of the software can be implemented with the best suited technology. Microservice architecture is complex but the benefits it brings to scalability for example are clear. The purpose of this thesis was to plan and implement a deployment pipeline to deploy a microservice application. At first monolithic and microservice architecture were compared, and the effects of using microservices regarding the pipeline were studied. Then an overview of the tools available in the market and a tentative plan of the pipeline was made. Based on the plan an example of the deployment pipeline which deploys a simple microservices application was developed. The most significant result of the thesis is the observation that the deployment pipeline saves a lot of time of the application developers. The amount of time saved multiplies as the application size and the development team grows. It can be said that the deployment of microservices application should be automated as much as possible. The thesis ends with a summary that reflects the development process of the deployment pipeline.

# ALKUSANAT

Työ on tehty Lappeenrannassa ja Kontiolahdella syksyllä 2019. Kiitos henkilöille, jotka ovat tukeneet työn tekoa.

# SISÄLLYSLUETTELO

**1 JOHDANTO ..... 4**

**2 KIRJALLISUUSKATSAUS ALAN TEOKSIIN ..... 6**

2.1 DEVOPS JA JULKAISUPUTKET ..... 6

2.2 MIKROPALVELUARKKITEHTUURI ..... 7

**3 KOHDEOHJELMISTON MÄÄRITTELY ..... 9**

3.1 MIKROPALVELUPOHJAISEN SOVELLUKSEN PIIRTEET JA OMINAISUUDET ..... 9

3.2 EROAVAISUUDET VERRATTUNA PERINTEISEEN MONOLIITTISEEN SOVELLUKSEEN 10

3.3 MIKROPALVELUPOHJAISEN SOVELLUKSEN RAKENNE ..... 11

*3.3.1 Yksittäisen palvelun suunnittelu ..... 12*

*3.3.2 Palveluiden välinen kommunikointi ..... 12*

*3.3.3 Palveluiden integrointi ..... 13*

*3.3.4 Kohdesovelluksen varsinainen määrittely ..... 14*

**4 JULKAISUPUTKEN SUUNNITTELU ..... 15**

4.1 JULKAISUPUTKEN ERI VAIHEET ..... 16

4.2 MIKROPALVELUARKKITEHTUURIN VAIKUTUKSET JULKAISUPUTKEEN ..... 17

*4.2.1 Muutosten havaitseminen lähdekoodissa ..... 17*

*4.2.2 Mikropalvelusovelluksen testaaminen ..... 17*

*4.2.3 Mikropalvelusovelluksen julkaiseminen ..... 18*

*4.2.4 Mikropalvelusovelluksen turvallisuus ..... 20*

4.3 TOIMINNALLISET VAATIMUKSET JULKAISUPUTKELLE ..... 20

**5 KATSAUS MARKKINOILLA OLEVIIN TYÖKALUIHIN JA TYÖKALUJEN VALINTA ..... 24**

5.1 JATKUVAN INTEGRAATION PALVELIN ..... 25

5.2 TESTAAMISEEN LIITTYVÄT TYÖKALUT ..... 26

5.3 KONTITTAMISTYÖKALU SEKÄ KONTTIEN HALLINTATYÖKALU ..... 27

5.4 MIKROPALVELUJEN AJOYMPÄRISTÖ ..... 28

**6 JULKAISUPUTKEN TOTEUTUS VALITUILLA TYÖKALUILLA ..... 29**

6.1 JATKUVAN INTEGRAATION PALVELIN JA VERSIONHALLINTA ..... 29

6.2 SOVELLUKSEN YKSIKKÖTESTAUS ..... 30

6.3 Sovelluksen kontittaminen ........................................ 30
6.4 Sovelluksen integraatiotestaus ........................................ 31
6.5 Sovelluksen hyväksymistestaus ........................................ 32
6.6 Konttien hallinnointi ........................................ 32
6.7 Sovelluksen julkaiseminen ........................................ 33
6.8 Julkaisuputken vaiheiden määrittely ........................................ 34

**7 TULOSTEN ARVIOINTI ........................................ 35**

**8 YHTEENVETO ........................................ 37**

**LÄHTEET ........................................ 40**

## SYMBOLI- JA LYHENNELUETTELO

| | |
|---|---|
| API | Application Programming Interface |
| CD | Continous Delivery |
| CI | Continous Integration |
| DevOps | Development & Operations |
| HTTP | Hypertext Transfer Protocol |
| MVP | Minimum Viable Product |
| SaaS | Software as a Service |
| UML | Unified Modeling Language |

# 1 JOHDANTO

Erilaisten ohjelmistojen avulla ratkaistaan vuosi vuodelta monimutkaisempia ja laajempia ongelmia. Yritysten tavoitteena on pitää ohjelmistojensa kehittämis- ja ylläpitokulut mahdollisimman pieninä. Samaan aikaan markkinoiden tarpeisiin pitää pystyä vastaamaan tehokkaasti, eli toisin sanoen ohjelmistoon tulisi pystyä tuomaan uusia ominaisuuksia nopeasti. Suuret yritykset, kuten Netflix, Amazon ja eBay, ovat ryhtyneet käyttämään uudenlaista arkkitehtuuria web-pohjaisten sovelluksiensa rakentamiseen (Richardson ja Smith 2016, 4). Niin sanotun mikropalveluarkkitehtuurin avulla ohjelmisto jaetaan itsenäisiin yksiköihin, mikropalveluihin, jotka toteuttavat tiettyjä rajattuja toiminnallisuuksia ja yhdessä kommunikoiden ratkaisevat jonkin haastavan tehtävän (Savchenko 2019, 13). Mikropalveluarkkitehtuuri tarjoaa monia mielenkiintoisia hyötyjä verrattuna perinteiseen, yhdestä koodipohjasta koostuvaan monoliittiseen arkkitehtuuriin. Samalla mikropalveluiden itsenäinen luonne asettaa erilaisia haasteita: esimerkiksi eri palveluiden toiminnallisuuksia ei voida kutsua koodipohjaisilla funktiokutsuilla, vaan kommunikaation tulee tapahtua verkon yli (Indrasiri ja Siriwardena 2018, 63).

Nykyisin etenkin suurten yritysten ohjelmistoissa käytetään useita työkaluja ohjelmistokehityksen automatisointiin: lähdekoodin muutoksia, ohjelmiston testausta ja jopa ohjelmiston julkaisemista pyritään automatisoimaan sekä tehostamaan. Mitä enemmän ohjelmistokehityksen toistuvia ja selkeästi määriteltävissä olevia vaiheita pystytään suorittamaan automaattisesti, sitä nopeammin ja kustannustehokkaammin ohjelmistoa pystytään kehittämään. Erilaisia menetelmiä ja käytäntöjä, joilla ohjelmiston kehitys- ja ylläpitotyötä pyritään parantamaan, voidaan pitää niin sanotun DevOps-toimintakulttuurin mukaisena toimintana (Hüttermann 2012, 4). Monoliittisten sovellusten kehittämiseen liittyvät DevOps-toimintatavat ovat vakioituneet viimeisten vuosien aikana, ja tehokas tapa luoda automatisoitu prosessi ohjelmiston lähdekoodin muutoksesta ohjelmiston julkaisuun on selvillä. Tällaista tietyistä vaiheista koostuvaa prosessia voidaan nimittää termillä julkaisuputki.

Yllä esitelty mikropalveluarkkitehtuuri asettaa uusia haasteita DevOps-käytäntöjen noudattamiseen ja julkaisuputkien rakentamiseen. Sovelluksen monimutkainen rakenne ja palvelujen välinen heikko yhteys hankaloittavat ohjelmiston testaamista sekä julkaisua. Savchenko (2019, 15) esittää väitöskirjassaan, että esimerkiksi mikropalvelusovellusten testaamiseen ei ole olemassa selkeästi määriteltyä ja hyväksi todettua toimintatapaa. Koska mikropalveluarkkitehtuurin suosio on jatkuvassa kasvussa, erilaisia työkaluja niiden testaamiseen, hallinnointiin ja rakentamiseen kuitenkin kehitetään jatkuvasti. Prosessia, joka alkaa havaitusta sovelluksen lähdekoodin muutoksesta ja päättyy sovelluksen julkaisuun, ei silti ole määritelty yhtä selkeästi mikropalveluarkkitehtuurille kuin monoliittiselle arkkitehtuurille.

Tämän kandidaatintyön tavoite on suunnitella DevOps-periaatteiden mukainen julkaisuputki modernille mikropalveluarkkitehtuuria hyödyntävälle web-sovellukselle. Lisäksi tarkoituksena on tutkia, mitä eri työkaluja ja ratkaisuja markkinoilla on tarjolla julkaisuputken rakentamiseen, sekä vertailla niiden vahvuuksia ja heikkouksia. Työn kaksi keskeisintä tutkimuskysymystä voidaan määritellä seuraavasti: miten mikropalveluarkkitehtuuria hyödyntävän web-sovelluksen julkaisuputki kannattaa suunnitella ja mitä työkaluja markkinoilla on saatavilla julkaisuputkien rakentamiseen? Työ aloitetaan luomalla katsaus alaan liittyvään kirjallisuuteen, minkä jälkeen paneudutaan mikropalveluarkkitehtuurin tunnuspiirteisiin ja määritellään työssä käytettävä kohdesovellus. Seuraavaksi tutkitaan julkaisuputkiin liittyviä käytäntöjä ja perehdytään erityisesti siihen, mitä muutoksia mikropalveluarkkitehtuuri tuo sovelluksen julkaisemiseen. Samassa yhteydessä luodaan suunnitelma julkaisuputkesta, minkä pohjalta lähdetään etsimään sopivia työkaluja markkinoilta ja lopulta toteutetaan esimerkinomainen julkaisuputki kohdesovellukselle. Lopuksi arvioidaan suunnitellun julkaisuputken toimintaa ja vedetään yhteen työssä tehdyn tutkimuksen tärkeimmät lopputulokset.

# 2 KIRJALLISUUSKATSAUS ALAN TEOKSIIN

Alan kirjallisuudesta löytyy paljon tutkimuksia ja teoksia liittyen DevOps-menetelmiin ja niiden hyödyntämiseen ohjelmistokehityksessä. Suurimmassa osassa näistä teoksista perehdytään myös julkaisuputkien rakentamiseen, koska automaattiset julkaisuputket voidaan nähdä yhtenä tapana toteuttaa DevOps-toimintakulttuuria. Myös mikropalveluarkkitehtuurista löytyy paljon julkaisuja, joita käytetään apuna kohdeohjelmiston määrittelyssä sekä tutkittaessa sitä, millaisia mahdollisuuksia ja toisaalta vaatimuksia mikropalveluarkkitehtuurin käyttäminen tuo mukanaan julkaisuputkia suunniteltaessa.

Familiar (2015) käsittelee mikropalvelupohjaisten sovellusten DevOps-menetelmiä Microsoftin Azure-pilvipalvelua hyödyntäen. Tämän kandidaatintyön kontekstin huomioiden Familiarin kirja on lähteenä kokonaisvaltaisin, sillä se käsittelee sekä DevOpsia että mikropalveluarkkitehtuuria. Machiraju ja Suraj (2018) käsittelevät myös DevOps-menetelmiä Azure-pilvipalvelua hyödyntäen, mutta teoksessa käytettävät työkalut eroavat merkittävästi Familiarin käyttämiin työkaluihin verraten. Lisäksi Machirajun ja Surajin teos ei käsittele ollenkaan mikropalvelusovelluksia.

Näiden teosten lisäksi työssä käytetään hyödyksi myös muuta alan kirjallisuutta, seuraavissa kappaleissa eritellään löydettyä kirjallisuutta aihepiirin mukaan jaoteltuna.

## 2.1 DevOps ja julkaisuputket

DevOps:iin ja erityisesti julkaisuputkiin liittyvää kirjallisuutta on saatavilla yllättävän paljon ottaen huomioon, että kyseessä on suhteellisen uusi asia ohjelmistokehityksen saralla. Lwakatare ym. (2019) tuovat esille viiden eri yrityksen käytäntöjä liittyen jatkuvaan integrointiin ja julkaisuun. Lisäksi artikkeli tuo tärkeää tietoa siitä, miten yritysten DevOps-menetelmät on käytännössä toteutettu.

Hüttermann (2012) tuo kirjassaan esille DevOps:iin liittyviä asioita erityisesti kehittäjien näkökulmasta. Hän kuvailee tarkasti, miksi DevOps-toimintakulttuuri parantaa ohjelmistokehitystyötä. Lisäksi erityisen kiinnostavana osana hänen teostaan ovat kappaleet ”Automatic Releasing” ja ”Infrastructure as Code”, jossa hän perustelee ja antaa käytännön esimerkkejä nykyaikaisista menetelmistä ohjelmistojen julkaisuun ja ohjelmiston vaatimaan infrastruktuuriin liittyen.

Ravichandran ym. (2016) käsittelevät DevOps:n merkitystä organisaatioiden ja erityisesti johtotehtävissä olevien henkilöiden näkökulmasta. Kirjasta löytyy myös kokonaisvaltaista tietoa välttämättömistä DevOps-toimintatavoista sekä case-luontoisia esimerkkejä siitä, millaisia rahallisia säästöjä yritykset ovat saavuttaneet hyödyntämällä moderneja toimintatapoja ja nopeuttamalla sovellustensa julkaisua.

## 2.2 Mikropalveluarkkitehtuuri

Indrasiri ja Siriwardena (2018) käsittelevät kirjassaan mikropalveluarkkitehtuuria kokonaisvaltaisesti: kirja sisältää kattavasti tietoa mikropalveluiden suunnitteluun, kehittämiseen, integroimiseen, julkaisemiseen ja ajamiseen liittyen. Tässä työssä käytetään tätä teosta ensisijaisena lähteenä kohdesovelluksen määrittelyssä.

Savchenko (2019) puolestaan käsittelee väitöskirjassaan mikropalvelupohjaisten sovellusten testaamista. Hän esittelee aluksi mikropalveluarkkitehtuurin yleisiä piirteitä ja käsittelee sitten asioita, jotka liittyvät mikropalvelusovellusten testaamiseen. Väitöskirjassa on myös esitetty prototyyppi mikropalvelusovellusten testauspalvelusta. Tämän prototyypin pohjalta on myös tehty erillinen tutkimusprojekti (.Maintain), joka laajentaa testauspalvelun arkkitehtuuria. Kyseisen projektin tarkoituksena on luoda tuote, jonka avulla voidaan vähentää ohjelmistojen ylläpidosta aiheutuvia kuluja.

Savchenkon väitöskirjassa viitattu artikkeli (Savchenko ym. 2018) esittää yksityiskohtaisen kaavion mikropalvelusovellusten testausprosessista. Tätä kaaviota voidaan käyttää apuna suunniteltaessa tämän työn aiheena olevaa julkaisuputkea, sillä kuten Savchenkon ja muun

tutkimusryhmän (Savchenko ym. 2015, 2) artikkelissa todetaan, kohdesovelluksen validoiminen testaamalla on tärkeä osa sovellusten automatisoitua julkaisua.

# 3 KOHDEOHJELMISTON MÄÄRITTELY

Ennen kuin julkaisuputken suunnittelua ja työkalujen tutkimista on järkevää aloittaa, täytyy määritellä selkeästi kohdeohjelmisto eli se ohjelmisto, jolle julkaisuputki suunnitellaan. Aloitetaan perehtymällä ensiksi mikropalveluarkkitehtuurin tunnuspiirteisiin ja vertailemalla arkkitehtuuria perinteiseen monoliittiseen arkkitehtuuriin. Sen jälkeen siirrytään määrittelemään sopivan laajuinen kohdeohjelmisto julkaisuputkelle.

## 3.1 Mikropalvelupohjaisen sovelluksen piirteet ja ominaisuudet

Mikropalvelupohjainen sovellus tarkoittaa sovellusta, joka on implementoitu eristetyistä, itsenäisistä ja itseohjautuvista komponenteista, mikropalveluista. Nämä mikropalvelut suorittuvat omissa ajoympäristöissään, ja ne yleensä säilyttävät tilansa eristetyssä tietovarastossa. (Savchenko 2019, 14) Tämä tarkoittaa, että mikropalveluita voidaan suunnitella, kehittää, testata ja julkaista itsenäisesti erillään muista sovelluksen osista (Familiar 2015).

Mikropalvelupohjainen sovellus koostuu siis monista mikropalveluista. Tämä hajautettu rakenne mahdollistaa sen, että kutakin mikropalvelua voi olla kehittämässä oma kehitystiiminsä ja kehitystiimi voi valita itse mitä ohjelmistoteknologiaa, ohjelmointikieltä ja käyttöjärjestelmää heidän kehittämänsä mikropalvelu käyttää. Koska mikropalvelut keskittyvät toteuttamaan vain yhden sovelluksen toiminnon, voidaan sanoa, että palveluiden välillä on heikko yhteys. (Savchenko 2019, 14) Tästä seuraa monia hyötyjä:

- Yksittäisen mikropalvelun uudelleenkäyttö helpottuu (Savchenko 2019, 14).
- Mikropalvelun sisäistä toimintaa voidaan muokata vapaasti, kunhan mikropalvelu yhä toteuttaa sille asetetut ulkoiset rajapintavaatimukset (Savchenko 2019, 14).
- Mikropalvelut voidaan versioida tehokkaasti, ja migraatiot sekä yhteensopivuus taaksepäin voidaan toteuttaa esimerkiksi siten, että samasta mikropalvelusta on usea versio käynnissä samaan aikaan (Familiar 2015, 15).

- Jos yksittäisen mikropalvelun toiminnassa on ongelmia, on epätodennäköistä että koko sovelluksen toiminta häiriintyy (Ravichandran ym. 2016, 24).

Määritellään seuraavaksi, mitkä ovat suurimmat erot verrattaessa mikropalveluarkkitehtuuria perinteiseen monoliittiseen arkkitehtuuriin.

## 3.2 Eroavaisuudet verrattuna perinteiseen monoliittiseen sovellukseen

Tähän mennessä sovellusten arkkitehtuuri on ollut usein niin sanotusti monoliittinen: tämä tarkoittaa, että sovellus on yhtenäinen toimiva kokonaisuus (Ravichandran ym. 2016, 23). Erityisesti web-sovellusten tapauksessa tämä tarkoittaa, että sovellus noudattaa perinteistä asiakas-palvelin -arkkitehtuuria joka on rakennettu yksittäisen tietovaraston ympärille (Savchenko 2019, 14). Usein tällaiset web-sovellukset käyttävät myös niin sanottua kolmikerroksista arkkitehtuuria, jossa käyttöliittymä, sovelluslogiikka ja tietovaraston palvelut on eroteltu kolmeksi erilliseksi kerrokseksi (Familiar 2015, 22). Arkkitehtuurina monoliittinen rakenne on yksinkertainen, mutta etenkin suurien sovellusten tapauksessa rakenteesta aiheutuu ainakin seuraavia ongelmia:

- Jos yksikin toiminto/osa monoliittisessa sovelluksessa kaatuu, koko sovellus kaatuu. Tästä voi seurata, että kehitystiimi ei uskalla tehdä niin paljoa muutoksia sovellukseen kuin olisi mahdollista. (Ravichandran ym. 2016, 23-24)
- Kun sovellukseen tehdään muutoksia, koko sovellus on julkaistava uudestaan olipa kyseessä miten pieni muutos tahansa (Indrasiri ja Siriwardena 2018, 3).
- Sovelluksen skaalaaminen on hankalaa, koska koko sovellus ajetaan yleensä yksittäisen käyttöjärjestelmän prosessin kautta. (Savchenko 2019, 20)
- Uusien teknologioiden ja sovelluskehysten käyttöönotto on todella työlästä, sillä kaikkien toimintojen pitää olla kehitetty yhteensopivilla teknologioilla (Indrasiri ja Siriwardena 2018, 3).
- Vaikka sovellus tarjoaisikin näennäisesti monta eri palvelua, ovat palvelut vahvasti yhteydessä toisiinsa, mikä aiheuttaa kehittäjille hankaluuksia jos sovelluksen useita palveluita kehitetään samanaikaisesti. (Ravichandran ym. 2016, 24)

Niin kuin kappaleessa 3.1 on todettu, mikropalveluarkkitehtuurin käyttäminen ratkaisee yllä olevat ongelmat tehokkaasti. Mikropalveluarkkitehtuuri ei kuitenkaan ole täysin ongelmaton, ja DevOps-toimintatapojen hyödyntäminen on tietyiltä osin monimutkaisempaa kuin monoliittisten sovellusten tapauksessa. Seuraavassa asioita, jotka tulee ottaa huomioon mikropalvelusovellusten DevOps-toimintatapoja suunnitellessa:

- Sovelluksen ylläpito on haastavampaa, kun sovellus koostuu monilla eri ohjelmointikielillä, ohjelmistokehyksillä ja tietovarastoilla kehitetyistä palveluista jotka vielä kaiken lisäksi ajetaan erilaisissa ajoympäristöissä (Ravichandran ym. 2016, 25)
- Sovelluksen testaaminen sisältää useampia tasoja verrattuna monoliittiseen sovellukseen: esimerkkinä muun muassa mikropalveluiden välisen kommunikaation ja palveluiden dynaamisen skaalautumisen testaus (Familiar 2015, 18).
- Sovelluksen toiminnan seuraaminen on monimutkaisempaa, sillä mikropalveluista koostuva sovellus koostuu monista eri prosesseista jotka suorittuvat erillään toisistaan. Lisäksi kokonaan uusia seurantamenetelmiä tarvitaan, jotta sovelluksen toimivuus voidaan varmistaa. (Ravichandran ym. 2016, 25)
- Palvelujen välisen kommunikaation suunnittelu ja toteutus voi olla jopa vaikeampaa kuin itse palveluiden kehittäminen. (Indrasiri ja Siriwardena 2018, 16)

## 3.3 Mikropalvelupohjaisen sovelluksen rakenne

Mikropalveluarkkitehtuuria verrataan usein aiemmin esiteltyyn kolmikerrosarkkitehtuuriin (Savchenko 2019, 20). Nämä arkkitehtuurit eivät kuitenkaan ole toisensa poissulkevia, vaan usein sekä monoliittiset että mikropalvelupohjaiset sovellukset noudattavat kolmikerrosrakennetta. Merkittävänä erona on kuitenkin se, että mikropalvelusovelluksissa kukin mikropalvelu toteuttaa kolmikerrosrakenteen omana kokonaisuutenaan. (Familiar 2015). Koska mikropalvelut ovat toisistaan erillisiä kokonaisuuksia, on myös täysin mahdollista että jokin mikropalvelu ei toteuta kolmikerrosarkkitehtuuria, mutta yleisesti ottaen monoliittisten sovellusten tapaan myös mikropalvelut on tehokkainta toteuttaa kolmikerrosarkkitehtuuria hyödyntäen.

### 3.3.1 Yksittäisen palvelun suunnittelu

Mikropalvelut ovat siis itsenäisiä ja erillisiä toiminnallisia sovelluksen osia, jotka toteuttavat tietyn sovelluksen toiminnallisuuden (Familiar 2015, 10). Indrasirin ja Siriwardenan (2018) mukaan yksi tärkeimmistä asioista mikropalvelua suunniteltaessa on huolehtia siitä, että palvelu toteuttaa tietyn spesifin toiminnallisuuden sovelluksessa. Tällöin palvelulle voidaan määrittää selkeät vastuut, jotka se voi keskittyä toteuttamaan. Lisäksi koska mikropalveluiden tulee olla täysin riippumattomia toisistaan, tulee jokaisella palvelulla olla oma tietovarasto ja tietovaraston rakenne. Tämä tarkoittaa samalla sitä, että kullekin mikropalvelulle voidaan valita sille sopivin muoto tallentaa tietoa. (Indrasiri ja Siriwardena 2018, 12)

### 3.3.2 Palveluiden välinen kommunikointi

Mikropalveluiden välisen kommunikaation suunnittelu ja toteutus on yksi haastavimmista asioista liittyen mikropalveluiden kehittämiseen (Chawla ja Kathuria 2019, 18). Koska kukin palvelu suoriutuu omassa eristetyssä prosessissaan, mikropalvelut eivät voi kommunikoida keskenään käyttäen normaaleja kooditason funktiokutsuja (Indrasiri ja Siriwardena 2018, 63). Käytännössä katsoen mikropalvelujen pitää siis kommunikoida toistensa kanssa verkon yli. Savchenko ym. (2015, 2) painottavat, että kommunikaation tulisi silti käyttää mahdollisimman kevyttä menetelmää. Sekä Savchenko ym. (2015), Indrasiri ja Siriwardena (2018) että Chawla ja Kathuria (2019) mainitsevat HTTP-protokollan yhtenä yleisimmistä vaihtoehdoista mikropalvelujen välisen kommunikoinnin järjestämiseen.

HTTP-protokollan avulla mikropalvelut siis voivat kommunikoida toistensa kanssa. On kuitenkin tarpeellista määrittää protokollan lisäksi myös selkeä tapa siihen, miten palvelut tietävät toistensa tarjoamista toiminnallisuuksista. Palveluilla tulisi siis olla selkeä rajapinta. Hoffman (2016, 7) toteaa, että palveluiden suunnittelu kannattaa jopa aloittaa määrittelemällä niiden rajapinta (API). Kun ollaan tilanteessa, jossa jokaisella mikropalvelusovelluksen palvelulla on ymmärrettävä rajapinta, helpottuu mikropalveluiden

kommunikaation toteuttaminen huomattavasti. Indrasiri ja Siriwardena (2018, 66) esittelevät eri tekniikoita rajapintojen tekniseen toteuttamiseen. Näitä tekniikoita ovat esimerkiksi REST ja GraphQL.

### 3.3.3 Palveluiden integrointi

Mikropalvelujen välisen kommunikaation voi siis toteuttaa esimerkiksi REST-rajapintojen ja HTTP-pyyntöjen avulla. Tarkastellaan seuraavaksi, millaisia vaihtoehtoja koko mikropalvelusovelluksen toiminnan hallitsemiseen on. Yksittäinen mikropalvelu toteuttaa tietyn sille rajatun tehtävän, mikä tarkoittaa sitä, että monimutkaisten tehtävien suorittamiseen tarvitaan useiden mikropalvelujen yhteistyötä (Chawla ja Kathuria 2019, 22). Mikropalvelusovelluksen kokonaisuuden hallintaan on olemassa monia eri ratkaisuja. Indrasiri ja Siriwardena (2018, 176) esittävät kaksi erilaista vaihtoehtoa: toisessa vaihtoehdossa yksittäisten mikropalvelujen päälle rakennetaan ns. integraatiopalveluita, jotka yhdistelevät normaalien mikropalveluiden toimintoja suorittaen näin haastavampia tehtäviä. Integraatiopalvelujen päälle rakennetaan vielä rajapintapalvelut, jotka toimivat sovelluksen julkisena rajapintana sovelluksen kuluttaja-applikaatioille, kuten mobiilisovellukselle. Havainnollistava kuva arkkitehtuurista liitteessä 1. Toisena vaihtoehtona he näkevät ratkaisun, jossa palvelujen välinen integraatio toteutetaan tapahtumapohjaisesti ja yksittäiset palvelut kuuntelevat keskitetyn tapahtumarekisterin avulla niitä koskevia tapahtumia (liite 2).

Richardson ja Smith (2016, 15) sekä Chawla ja Kathuria (2019, 23) ehdottavat palvelujen integroinnin ratkaisuksi yksittäistä API-palvelua (engl. API Gateway), jonka tarkoituksena on olla ainoa sovelluksesta ulospäin näkyvä rajapinta. Myös Savchenko ym. (2015, 1) ovat mallintaneet mikropalveluarkkitehtuurin niin, että mikropalvelutason yläpuolella on yksittäinen API-palvelu. API-palvelu siis paketoi koko alla olevan mikropalveluiden kokonaisuuden yhdeksi rajapinnaksi, jota eri sovelluksen kuluttajasovellukset voivat käyttää (liite 3). Myös Familiar (2015, 24) ehdottaa API-palvelua mikropalvelujen ja kuluttajasovellusten väliseksi kerrokseksi, ja hänen mielestään API-palvelulla voidaan luoda yhdenmukainen ja turvallinen rajapinta koko mikropalvelusovellukselle.

Kun vertailee edellä esitettyjä kolmea tapaa integroida mikropalvelut toimivaksi kokonaisuudeksi, on jälkimmäisenä ehdotettu yksittäisen API-palvelun toteuttaminen teknisesti helpointa. Indrasirin ja Siriwardenan (2018) vaihtoehdot saattavat olla suorituskyvyltään tehokkaampia, mutta niiden toteuttaminen on monimutkaisempaa. Valitaan tässä työssä määriteltävän kohdesovelluksen ratkaisuksi yksittäisen API-palvelun toteuttaminen mikropalvelukerroksen päälle. Määritellään seuraavaksi näiden tietojen pohjalta julkaisuputkessa käytettävän kohdesovelluksen arkkitehtuuri yksittäisten palvelujen, palvelujen välisen kommunikoinnin sekä palvelujen integroinnin osalta.

### 3.3.4 Kohdesovelluksen varsinainen määrittely

Edellä on käyty läpi mikropalvelusovelluksen ominaisuuksia sekä suunnitteluperiaatteita. Määritellään seuraavaksi esiteltyjen periaatteiden mukainen mikropalvelusovellus. Käytetään Spring-sovelluskehystä sovelluslogiikkaa toteuttavien mikropalveluiden toteuttamiseen ja React-kehystä sovelluksen front-end -palvelun toteuttamiseen. Toteutetaan mikropalveluiden integrointi kappaleessa 3.3.3 esitetyn periaatteen mukaisesti Netflix Zuul -palvelulla.

Kohdesovellus koostuu siis kahdesta sovelluslogiikkaa toteuttavasta mikropalvelusta, yhdestä API-palvelusta sekä yhdestä yksinkertaisen käyttöliittymän tarjoavasta mikropalvelusta. Zuulin avulla käyttöliittymäpalvelun ei tarvitse huolehtia eri palveluiden sijainneista, sillä Zuul tarjoaa sille yksittäisen osoitteen jonka avulla käyttöliittymä voi kommunikoida sovelluslogiikan palveluille. Sovelluksen mikropalvelut kommunikoivat toistensa kanssa käyttäen kappaleessa 3.3.2 kuvattua HTTP-protokollaa, ja kukin palveluista toteuttaa REST-rajapinnan eri operaatioiden suorittamiseen.

Kohdesovelluksen arkkitehtuuri sekä eri palveluiden toteuttamiseen käytetyt teknologiat on kuvattu liitteissä 4 ja 5. Kohdesovellus pidetään suhteellisen yksinkertaisena tämän kandidaatintyön kontekstissa, sillä työn pääasiallisena tarkoituksena on julkaisuputken suunnittelu ja toteuttaminen.

# 4 JULKAISUPUTKEN SUUNNITTELU

Ohjelmistokehityksen eri vaiheista kannattaa automatisoida niin monta kuin on mahdollista. Familiarin (2015) mukaan automaatiota voidaan ottaa käyttöön ainakin seuraavilla neljällä ohjelmistokehityksen alueella: pilviympäristön resurssien luominen ja konfigurointi, ohjelmiston kääntäminen ja testaaminen, ohjelmiston julkaisu ja ohjelmiston tarvitsemien resurssien hallinta.

Indarsiri ja Siriwardena (2018) näkevät ohjelmistokehityksen automatisoinnin jakautuneen kahteen laajaan kategoriaan: jatkuvaan integraatioon (Continuous Integration, CI) ja jatkuvaan julkaisemiseen (Continuous Deployment, CD). Jatkuvalla integraatiolla tarkoitetaan käytäntöä, joka Pulkkisen (2013) mukaan käsittää ohjelmiston kääntämisen, testaamisen sekä mahdollisista ongelmista ilmoittamisen ohjelmistokehittäjälle. Käytännössä jatkuvan integraation avulla siis pyritään siihen, että mahdolliset kehitystyössä tapahtuneet virheet huomataan ja korjataan ennen kuin kyseistä ohjelmiston versiota lähdetään julkaisemaan.

Jatkuvaan julkaisuun liittyvät vaiheet eivät ole yhtä yksikäsitteisiä kuin jatkuvan integraation. Jatkuva julkaisu voidaan nähdä esimerkiksi kokonaisuutena, jossa jatkuvan integraation vaiheet läpikäynyt ohjelmakoodi julkaistaan testausympäristöön, jossa sille suoritetaan vielä manuaalinen hyväksymistestaus ennen lopullista julkaisua tuotantoympäristöön (Familiar 2015, 68). Sekä Pulkkinen (2013) että Rossel (2017) näkevät jatkuvan julkaisun täysin automatisoituna putkena, jonka lopputuloksena ohjelmisto on julkaistuna tuotantoympäristöön. Molemmat kuitenkin huomioivat myös Familiarin (2015) esittämän toimintatavan määrittelemällä sen nimellä ”jatkuva toimittaminen” (Continuous Delivery).

Tässä työssä julkaisuputkella tarkoitetaan automatisoitua, toisistaan erillisistä vaiheista koostuvaa kokonaisuutta, joka toteuttaa Rosselin (2017) määritelmän mukaisen jatkuvan julkaisun tavoitteet. Koska kohdeohjelmisto on pilvipohjainen, voidaan julkaisuputken avulla automatisoida Familiarin (2015) esittämät neljä eri osa-aluetta. Näiden asioiden

lisäksi kohdeohjelmiston käyttämä mikropalveluarkkitehtuuri otetaan huomioon julkaisuputken suunnittelussa. Määritellään seuraavaksi tarkemmin eri vaiheet, jotka julkaisuputki suorittaa.

## 4.1 Julkaisuputken eri vaiheet

Pulkkinen (2013) on kirjoituksessaan määritellyt vaiheet, jotka jatkuvan julkaisun tavoitteet täyttävän julkaisuputken tulee suorittaa:

1. Muutoksen havaitseminen lähdekoodissa
2. Ohjelmiston yksikkötestaus
3. Ohjelmiston integraatiotestaus
4. Julkaisu testausympäristöön
5. Hyväksymistestaus
6. Julkaisu tuotantoympäristöön

Kaikki vaiheet tapahtuvat automaattisesti julkaisuputken toimesta. Mikäli missä tahansa vaiheessa tapahtuu virhe, julkaisuputken toiminta keskeytyy. Vaiheet 1-3 voidaan nähdä kuuluvan ohjelmiston jatkuvaan integraatioon. Jatkuvan integraation tavoitteena on varmistaa, että ohjelmisto on julkaistavissa koko ajan (Rossel 2017, 8). Sekä Rosselin (2017) että Pulkkisen (2013) mukaan jatkuvan integraation toimivuus on esivaatimus onnistuneelle jatkuvalle julkaisulle. Näin ollen jos julkaisuputkessa ollaan päästy vaiheeseen 4 asti voidaan olettaa, että ohjelmiston versionhallinnan päähaarassa on toimiva versio ohjelmistosta. Vaiheet 4-6 koostuvat ohjelmiston julkaisemisesta ensin testaus- ja sitten tuotantoympäristöön. Vaiheessa 5 testiympäristöön julkaistulle sovellukselle tehdään automaattinen hyväksymistestaus. Sen tavoitteena on selvittää, toimiiko ohjelmisto ajoympäristössään oikealla tavalla (Hüttermann 2012, 115). Jos ohjelmisto toimii oikein testiympäristössä, jonka kannattaa olla mahdollisuuksien mukaan täysin identtinen tuotantoympäristöön nähden (Indrasiri ja Siriwardena 2018, 57), voidaan suorittaa julkaisuputken viimeinen vaihe eli julkaista ohjelmisto tuotantoympäristöön.

Kuten Hüttermann (2012) kuvailee, voidaan julkaisuputken ajatella ottavan syötteenään muuttuneen ohjelmistokoodin ja tuottavan lopputuloksena julkaistun ohjelmiston. On huomattava, että yllä esitetty kuusivaiheinen malli on melko suurpiirteinen, eikä se ole tarpeeksi tarkka tässä kandidaatintyössä toteutettavan julkaisuputken suunnittelemiseen. Mallista tehdään myöhemmin yksityiskohtaisempi, ottaen huomioon kohdeohjelmiston arkkitehtuuri, joka on esitelty liitteessä 4. Tarkastellaan kuitenkin sitä ennen, miten mikropalveluarkkitehtuuri vaikuttaa ohjelmiston jatkuvan julkaisun toteuttamiseen.

## 4.2 Mikropalveluarkkitehtuurin vaikutukset julkaisuputkeen

Kuten kappaleessa 3.2 on todettu, mikropalveluista koostuva sovellus eroaa perinteisestä monoliittisesta sovelluksesta monella tavalla. Nämä eroavaisuudet näkyvät väistämättä myös suunniteltaessa julkaisuputkea mikropalvelusovellukselle. Käsitellään seuraavaksi mikropalveluarkkitehtuurin tuomia haasteita aihealue kerrallaan.

### 4.2.1 Muutosten havaitseminen lähdekoodissa

Kappaleessa 4.1 esitellyn mallin mukaan julkaisuputken ensimmäinen vaihe on tunnistaa ohjelmiston lähdekoodin muutos versionhallinnassa. Koska mikropalvelut ovat täysin erillisiä kokonaisuuksia toisistaan, ei niiden lähdekoodien tarvitsisi välttämättä olla yksittäisessä repositoriossa versionhallinnassa. Kuitenkin esimerkiksi Familiar (2015, 74) on toteuttanut mikropalvelusovelluksensa repositorion hakemistorakenteen siten, että kukin mikropalvelu on samassa repositoriossa omassa hakemistossaan. Tämä vaikuttaa ratkaisuna selkeältä, ja tällöin muutokset lähdekoodissa pystytään havaitsemaan samalla tavalla kuin monoliittisessakin sovelluksessa.

### 4.2.2 Mikropalvelusovelluksen testaaminen

Mikropalvelusovellusten testaaminen eroaa merkittävästi esimerkiksi monoliittisten sovellusten testaamisesta. Savchenkon (2019) mukaan mikropalveluarkkitehtuurin mukainen kehitystyö ei ainakaan vielä ole yhtä standardoitua verrattuna muihin yleisiin

arkkitehtuuriratkaisuihin. Samalla hän toteaa, että ei ole olemassa selkeästi määriteltyä tai hyvin tunnettua toimintatapaa testata mikropalvelusovelluksia.

Familiar (2015, 18) puolestaan ei näe suurta eroa monoliittisten ja mikropalvelusovellusten testaamisessa. Hän toteaa, että sovelluksen yksikkö- ja integraatiotestaus (jotka kuuluvat myös kappaleessa 4.1 esitettyyn kuusivaiheiseen malliin) toimivat mikropalveluiden tapauksessa lähes samalla tavalla kuin ennenkin. Familiar kuitenkin huomauttaa, että edellä mainittujen perinteisten testaustasojen lisäksi mikropalvelujen toimintaa täytyy testata julkaisuputken eri vaiheissa myös esimerkiksi yksittäisen palvelun rajapinnan, palvelujen välisen kommunikaation ja koko järjestelmän skaalautumisen osalta.

Savchenkon ja muun tutkimusryhmän (2018) artikkelissa esitetään prosessi mikropalvelujen testaukselle. Siinä testaus koostuu kuudesta eri vaiheesta, joissa voidaan havaita yhteneväisyyksiä Familiarin tekemien huomioiden kanssa. Tiivistetysti voidaan sanoa, että prosessin vaiheet 1-4 koostuvat yksittäiseen mikropalveluun liittyvistä testeistä, joissa testataan mikropalvelun lähdekoodia sekä palvelun toimintaa sen jälkeen kun siitä on tehty Docker-kontti. Vaiheessa 5 mikropalveluohjelmiston testiympäristö pystytetään, ja integraatiotestit, jotka siis testaavat mikropalvelujen välistä toimintaa, suoritetaan. Viimeisessä vaiheessa mikropalvelusovellukselle suoritetaan vielä vakaustestaus, jossa voidaan esimerkiksi testata sitä, että sovellus pystyy selviämään yksittäisen komponentin hetkittäisistä toimintahäiriöistä.

### 4.2.3 Mikropalvelusovelluksen julkaiseminen

Edelliset kappaleet ovat käsitelleet sitä, kuinka mikropalvelusovelluksen jatkuva integraatio saadaan toteutettua. Seuraavana vaiheena on selvittää, kuinka mikropalvelusovelluksen julkaiseminen tapahtuu. Indrasiri ja Siriwardena (2018, 219) toteavat mikropalvelusovelluksen julkaisemisen koostuvat useasta erillisestä ja itsenäisestä julkaisusta. Tämä johtuu siitä, että jokaisen mikropalvelun ollessa oma kokonaisuutensa täytyy jokainen palvelu myös julkaista erikseen. Koska mikropalvelujen määrä voi laajassa sovelluksessa olla suuri (jopa useampi sata eri mikropalvelua), on automaattisen julkaisujärjestelmän käyttäminen lähes pakollista (Indrasiri ja Siriwardena 2018, 219).

Myös Chawla ja Kathuria (2019) näkevät mikropalvelujen julkaisun monimutkaisena operaationa. Heidän mielestään mikropalvelusovelluksen julkaisun helpottamiseksi kannattaa käyttää jotakin mikropalvelujen hallintatyökalua. Ennen kuin näihin hallintatyökaluihin kannattaa tutustua tarkemmin, tarkastellaan millaiseen ympäristöön yksittäinen mikropalvelu kannattaa julkaista.

Kun puhutaan puhtaasti pilvipohjaisista ajoympäristöistä, on mikropalvelujen suorittaminen pääsääntöisesti mahdollista kahdella eri tavalla: virtuaalikoneissa tai konteissa. Vaikka molemmat ratkaisut ovat skaalautuvia ja pilvipohjaisia, on erillisen virtuaalikoneen pystyttäminen mikropalvelua varten usein turhan raskasta. (Chawla ja Kathuria 2019, 34) Mikropalvelujen ajaminen konteissa on kevyempää kuin virtuaalikoneissa ajaminen, sillä vaikka kontit ovatkin toisistaan eristettyjä kokonaisuuksia, voi niitä ajaa useita samalla isäntäkoneella (Indrasiri and Siriwardena 2018, 219). Savchenko (2019) esittelee kolme eri teknologiaa mikropalveluiden kontittamiseen: Vamp, Docker ja Mesosphere. Dockeria voidaan pitää tällä hetkellä alan standardina kontittamisteknologiana, esimerkiksi monet ECP-ohjelmistojen (Enterprise Container Platform) tuottajat keskittyvät tarjoamaan ainoastaan Docker-kontteihin pohjautuvia ohjelmistoratkaisuja (Bartoletti ym. 2018, 2).

Tarkastellaan vielä, miten kontitettuja mikropalveluita voidaan julkaista hallitusti varsinaiseen sovelluksen ajoympäristöön. Chawla ja Kathuria (2019) määrittelevät, että konttien hallintajärjestelmän tarkoituksena on suorittaa mikropalvelukontteja, varmistaa palvelujen saatavuus ja tasata palvelujen kuormaa silloin, kun kohdesovellus kohtaa suurta liikennettä. Indrasiri ja Siriwardena (2018, 235) tarkentavat hallintajärjestelmän vaatimuksia siltä osin, että sen tulee vastata mikropalvelukonttien toiminnasta koko niiden elinkaaren ajan. Yksittäisestä mikropalvelusta voi olla useampi instanssi toiminnassa riippuen palveluun kohdistuvasta kuormasta. Sovelluksen julkaisusta huolehtivan järjestelmän tulee siis pystyä joko lisäämään tai vähentämään tietyn mikropalvelun konttien määrää. Tätä toimintatapaa kutsutaan horisontaaliseksi skaalaamiseksi. (Indrasiri ja Siriwardena 2018, 56)

Chawla ja Kathuria (2019, 39) esittelevät neljä eri vaihtoehtoa konttien ja samalla koko mikropalvelusovelluksen hallintaan: Azure Service Fabric, Kubernetes, Docker Swarm ja

Compose sekä Mesos DC/OC. Indrasiri ja Siriwardena (2018, 235-236) toteavat Kuberneteksen ja Mesoksen olevan tämän hetken suosituimmat työkalut konttien hallinnoimiseen, joskin he keskittyvät käsittelemään vain Kuberneteksen käyttöä tarkemmin. Tarkastellaan näiden työkalujen eroja tämän työn kappaleessa 5.

### 4.2.4 Mikropalvelusovelluksen turvallisuus

Mikropalvelusovellusten turvallisuudesta huolehtiminen on monimutkaisempaa verrattuna monoliittisiin sovelluksiin. Indrasiri ja Siriwardena (2018, 60) toteavat, että turvallisuus on otettava huomioon hyvin aikaisessa vaiheessa suunnittelua. Lisäksi he lisäävät, että turvallisuudesta huolehtivien testien on syytä olla osana julkaisuputkea. Pohditaan seuraavaksi, mitkä tekijät tekevät mikropalvelujen turvallisuuden varmistamisesta vaikeaa.

Aiemmin on todettu, että mikropalvelut kommunikoivat keskenään verkon yli. Kohdesovelluksen tapauksessa tämä kommunikaatio tapahtuu HTTP-protokollalla REST-rajapintoja hyväksikäyttäen. Tämä on merkittävä ero verrattuna monoliittisovelluksiin, sillä niissä kommunikaatio toimintojen välillä tapahtuu kooditasolla. Näin ollen onkin varmistettava, että mikropalveluiden kommunikaatio on autentikoitua ja tietoturvallista. Kommunikaation turvallisuuden varmistaminen pitää myös varmistaa julkaisuputken toimesta erilaisilla testeillä.

Mikropalveluiden konteissa suorittaminen on monella tavalla hyödyllistä, kuten kappaleessa 4.2.3 on todettu. Turvallisuuden näkökulmasta konttien käyttäminen tuo kuitenkin haasteita: Jangla (2018, 7) toteaa, että konttien taustalla olevan isäntäkäyttöjärjestelmän turvallisuus vaikuttaa suoraan kaikkien käyttöjärjestelmän sisällä ajettavien konttien turvallisuuteen. On siis varmistettava, että konttien ajoympäristönä käytetään luotettavaa kolmannen osapuolen palvelua.

## 4.3 Toiminnalliset vaatimukset julkaisuputkelle

Tutkitaan seuraavaksi, mitä vaatimuksia julkaisuputken toiminnalle voidaan asettaa, kun otetaan huomioon lopullinen kohdeohjelmisto, jolle julkaisuputki suunnitellaan. Lisäksi tarkennetaan kappaleessa 4.1 esitettyä mallia julkaisuputken eri vaiheista siten, että sen pohjalta voidaan lähteä miettimään, millaisia työkaluja julkaisuputken toteuttamiseen vaaditaan.

Julkaisuputken tarkoituksena on siis automatisoida ohjelmiston julkaiseminen. Se havaitsee, kun ohjelmiston lähdekoodiin on tehty muutos ja aloittaa automaattisesti ohjelmiston julkaisun tuotantoympäristöön. Kappaleessa 4.1 esiteltiin kuusivaiheinen malli, joka on varsin toimiva etenkin monoliittisten sovellusten tapauksessa. Mikropalveluarkkitehtuuri asettaa kuitenkin uusia vaatimuksia ohjelmiston julkaisulle. Edellä esitetyt aihepiirit kuten sovellusten palveluiden kontittaminen ja konttien hallinnointi tulee ottaa huomioon julkaisuputken suunnittelussa. Lisäksi on mietittävä tarkkaan, missä vaiheessa eri testauksen tasoja suoritetaan. On eri asia testata palvelujen välistä integraatiota lokaalissa ympäristössä verrattuna kontitettuun ajoympäristöön.

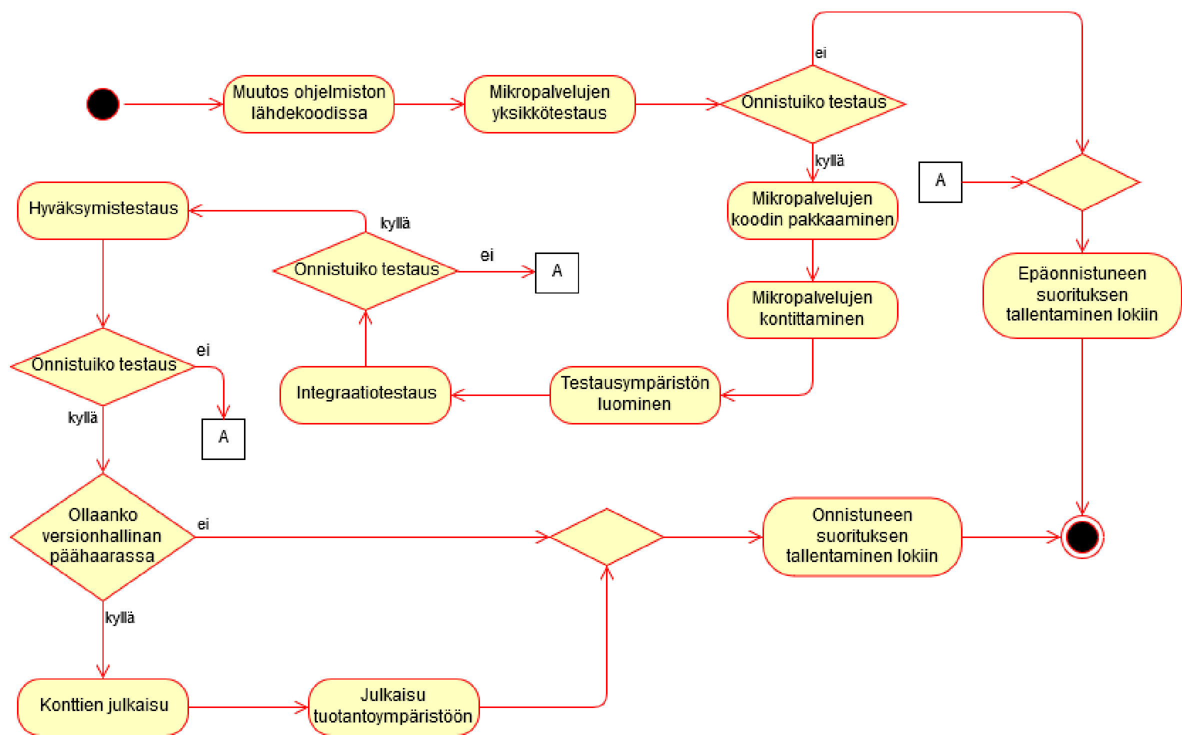
Seuraavassa on esitetty täydennetty ja tarkennettu malli mikropalveluarkkitehtuurin julkaisuputken eri vaiheista:

Vaihe 1: Muutoksen havaitseminen lähdekoodissa
Vaihe 2: Yksittäisten mikropalvelujen yksikkötestaus (jokainen palvelu testaa sisäisen toimintansa)
Vaihe 3: Mikropalvelujen lähdekoodin käsittely (pakkaaminen, koodin optimoiminen jne.)
Vaihe 4: Kontittaminen (jokainen mikropalvelu pakataan omaan konttiinsa)
Vaihe 5: Konttien pystyttäminen testausympäristöön integraatiotestausta varten
Vaihe 6: Integraatiotestaus (varmistetaan, että kontit pystyvät kommunikoimaan keskenään)
Vaihe 7: Hyväksymistestaus (varmistetaan, että sovellus toimii loppukäyttäjän näkökulmasta oikein)

Vaihe 8: Konttien julkaisu (julkaistaan kontit yleiseen rekisteriin, josta ne ovat koko kehitystiimin saatavilla)

Vaihe 9: Julkaisu tuotantoympäristöön (hyödynnetään julkaistuja kontteja, tällöin julkaisuympäristö luodaan aina ajantasaisimmista konttiversioista)

Alla olevassa kuvassa on esitetty edellä määritetyt vaiheet UML-kaaviona. Kaavion tyypiksi valittiin aktiivisuuskaavio, koska sen avulla voidaan mallintaa tehokkaasti erilaisia prosesseja (Engels ym. 2005, 3). Se siis soveltuu hyvin julkaisuputken toiminnan esittelemiseen.




**Kuva 1.** Julkaisuputken toiminta UML-aktiviteettikaaviona.

Vaikka tämän työn kohdesovelluksena on varsin yksinkertainen mikropalvelusovellus, voidaan yllä oleva malli julkaisuputkesta yleistää myös monimutkaisemmille sovelluksille. Toki käytetyt arkkitehtuurivalinnat, kuten esimerkiksi kappaleessa 3.3.3 esitellyt vaihtoehdot palvelujen integroinnille voivat eri tapauksissa muuttaa optimaalisen julkaisuputken mallia. Lisäksi esimerkiksi testaus- ja tuotantoympäristön vaihtelevuus eri tapauksissa vaikuttaa väistämättä siihen, miten julkaisuputki kannattaa kussakin tilanteessa rakentaa. Tutkitaan seuraavaksi, mitä työkaluja markkinoilla on tarjolla liittyen

julkaisuputkien rakentamiseen. Käytetään katsauksen pohjana yllä esitettyä mallia mikropalvelusovelluksen julkaisuputkesta.

# 5 KATSAUS MARKKINOILLA OLEVIIN TYÖKALUIHIN JA TYÖKALUJEN VALINTA

Onnistuneen julkaisuputken rakentamiseen tarvitaan monia erilaisia työkaluja. Pulkkinen (2013, 47-48) määrittelee, että kaiken keskipisteenä on jatkuvan integroinnin palvelin (engl. CI Server), joka kääntää lähdekoodin suoritettavaksi ohjelmaksi, testaa ohjelmaa eri testitasoilla sekä julkaisee toimivaksi todetun version varsinaiseen ajoympäristöön. Jatkuvan integroinnin palvelin voidaan siis nähdä julkaisuputken välttämättömänä osana, joka hallinnoi julkaisuputken etenemistä siirtyen tietyn vaiheen onnistuessa seuraavaan vaiheeseen. Tämän takia eri työkaluja valittaessa on järkevää päättää ensimmäisenä, minkä palveluntarjoajan jatkuvan integroinnin palvelinta julkaisuputkessa käytetään.

Kohdeohjelmiston testaaminen on merkittävä osa julkaisuputken toimintaa. Kuten aiemmin on todettu, mikropalveluarkkitehtuuri asettaa testaamiselle suurempia vaatimuksia verrattuna perinteisiin monoliittisiin web-sovelluksiin. Erityisesti palvelujen välisen kommunikaation testaaminen niin toiminnallisuuden kuin tietoturvallisuuden kannalta on suuressa osassa. Nämä vaatimukset otetaan huomioon, kun markkinoilla olevia testauksen työkaluja vertaillaan.

Tarkastellaan tämän osion lopussa, mitä työkaluja mikropalvelusovelluksen ajoympäristöön liittyy. Koska kohdesovelluksen mikropalveluita ajetaan eristetyissä konteissa, täytyy tarkastella mitä kontittamistyökalua sekä konttien hallintatyökalua kannattaa käyttää. Huomiota täytyy kiinnittää erityisesti siihen, että valitut työkalut toimivat koko julkaisuputken keskipisteen eli jatkuvan integroinnin palvelimen kanssa sujuvasti yhteen. Viimeisenä asiana ajoympäristöön liittyen täytyy vielä päättää, minkä palveluntarjoajan pilvipalveluun mikropalvelusovelluksen sisältämät kontit lopulta julkaistaan hallintatyökalun avulla. Luodaan seuraavaksi katsaus edellä esitettyjen työkalujen tarjontaan markkinoilla.

## 5.1 Jatkuvan integraation palvelin

Condo ym. (2017) tutkivat Forresterin julkaisemassa raportissa kymmentä eri jatkuvan integraation palvelinta. Myös Pulkkinen (2013, 49) esittelee eri työkaluja jatkuvaan integraatioon. Kun otetaan huomioon määritelty kohdesovellus sekä suunniteltu julkaisuputki, erityisesti seuraavat ominaisuudet ovat tärkeitä vaatimuksia liittyen jatkuvaan integraatioon:

- Mahdollisuus monipuoliseen testaukseen (mikropalvelujen testaaminen tapahtuu useammalla tasolla kuin monoliittisen sovelluksen)
- Tuki konteille sekä konttien hallintatyökaluille (mikropalvelut ajetaan konteissa hallintatyökalun avulla)
- Mahdollisuus julkaista sovellus haluttuun pilvipalveluun (kontitettu sovellus halutaan julkaista pilvipalveluun)

Condon ym. (2017) tekemässä raportissa johtaviksi jatkuvan integraation palveluiksi on todettu GitLab, CircleCI, Microsoft sekä CloudBees. Näistä palveluista kaikki ovat pilvipohjaisia palveluita, eli niiden käyttöönottoon ei vaadita omaa palvelinympäristöä. Osa edellä mainituista palveluntarjoajista kuitenkin tarjoaa myös mahdollisuuden käyttää palvelua omalla palvelinympäristöllä. Lwakataren ym. (2019, 222) case-pohjaisessa tutkimuksessa selvitettiin viiden eri yrityksen työkaluja ja toimintatapoja julkaisuputken rakentamiseen, ja jokainen viidestä yrityksestä käytti avoimen lähdekoodin Jenkins-palvelua jatkuvan integraation palvelimena. Kun Condon ym. (2017) esittelemiä työkaluja tutkii tarkemmin, käy ilmi että CloudBees käyttää myös omassa palvelussaan pohjana edellä mainittua Jenkins-ohjelmistoa.

Kun otetaan huomioon ylempänä esitetyt vaatimukset liittyen jatkuvan integraation palvelimeen ja tarkastellaan Condon ym. (2017, 7) taulukkoa eri palveluiden ominaisuuksista, voidaan todeta että useat palveluntarjoajat tarjoavat mikropalvelusovelluksen julkaisemiseen sopivia työkaluja. Kuitenkin GitLabin tarjoama palvelu, jossa on yhdistettynä ohjelmiston versionhallinta, jatkuva integraatio sekä myös

jatkuva julkaisu, on perusteltua käyttää tässä kandidaatintyössä GitLabin tarjoamaa palvelua julkaisuputken pohjana.

## 5.2 Testaamiseen liittyvät työkalut

Mikropalvelusovelluksen testaamiseen liittyvät työkalut riippuvat siitä, mitä ohjelmistokehyksiä palvelujen kehittämiseen on käytetty. Tämän työn tapauksessa ohjelmistokehyksinä palveluille olivat Spring Boot ja React.js. Molemmille kehyksille on olemassa monipuolisia testaustyökaluja. Sillä, mikä testaustyökalu otetaan käyttöön ei ole tämän työn kontekstissa suurta merkitystä, joten valitaan Spring Boot -palvelujen testaamiseen jUnit-työkalu ja React.js-palvelun testaamiseen Jest-työkalu. Näiden työkalujen avulla toteutetaan kohdesovellukseen yksinkertaiset yksikkötestit, joiden suorittaminen lisätään GitLabiin määritettävään julkaisuputkeen.

Kuten kappaleessa 4.2.2 on todettu, mikropalveluarkkitehtuuri asettaa korkeampia vaatimuksia sovelluksen testaamiselle verrattuna monoliittiseen arkkitehtuuriin. Tarvitaan siis myös testaustyökalu, joka varmistaa että eri konteissa suoritettavat mikropalvelut pystyvät kommunikoida toistensa kanssa. Koska kohdesovelluksen käyttöliittymäpalvelu hyödyntää Zuulilla toteutettua API-palvelua, täytyy varmistaa että käyttöliittymäpalvelu saa yhteyden API-palveluun ja että API-palvelu saa yhteyden sovelluslogiikan palveluihin. Lisäksi pitää huomioida, että integraatiotestausvaiheessa mikropalvelut ovat jo omissa konteissaan. Mikropalvelujen väliset yhteydet hyödyntävät HTTP-protokollaa, kuten kappaleessa 3.3.4 on määritelty. Käytetään integraatiotestauksessa Pythonille tehtyä Tavern-kirjastoa. Kyseisen kirjaston avulla saadaan määritettyä selkeät ja yksinkertaiset testit palvelujen välisen kommunikaation testaamiseen.

Pulkkinen (2013) oli määritellyt julkaisuputken yhdeksi osaksi myös hyväksymistestauksen, ja myös tämän työn tarkennetussa julkaisputken mallissa se on yksi suoritettavista testauksen tasoista. Hyväksymistestauksen tavoitteena on varmistaa, että julkaistava sovellus toimii loppukäyttäjän näkökulmasta halutulla tavalla. Toteutetaan tässä työssä kohdesovelluksen hyväksymistestaus Robot Framework -työkalun avulla. Kyseisen työkalun avulla voidaan

simuloida loppukäyttäjän toimintaa hänen käyttäessään käyttöliittymäpalvelua internet-selaimen kautta.

### 5.3 Kontittamistyökalu sekä konttien hallintatyökalu

Aiemmin on jo todettu, että Dockeria voidaan pitää alan standardina kontittamisteknologiana. Savchenko (2019, 18) esitteli myös vaihtoehtoja Dockerille (Vamp ja Mesosphere), mutta koska Bartolettin ym. (2018) tekemä tutkimus sekä monet muut tämän työn lähteet käyttävät ja suosittelevat Dockeria lähes poikkeuksetta, käytetään myös tässä työssä mikropalvelusovelluksen konttiteknologiana Dockeria.

Seuraavaksi voidaan lähteä miettimään työkalua Docker-konttien suorittamiseen ja hallinnoimiseen. Jangla (2018, 77-78) esittelee ratkaisuksi Dockerin oman Docker Compose-palvelun, joka mahdollistaa useiden konttien ajamisen ja näin mikropalveluarkkitehtuurin onnistuneen toteuttamisen. (Vohra 2016, 3) puolestaan käyttää Googlen kehittämää avoimen lähdekoodin Kubernetes-työkalua. Hänen mukaansa Kuberneteksen avulla useita eri kontteja voidaan paketoida ns. Podeihin, joiden avulla esimerkiksi konttien välisiä yhteyksiä voidaan käsitellä tarvittaessa Podin tasolla pelkän konttien välisen tason sijaan. Kubernetes on suurien mikropalvelusovellusten tapauksessa parempi ratkaisu kuin Docker Compose, mutta tämän työn kohdesovelluksen tapauksessa Kubernetes toisi tarpeetonta monimutkaisuutta sovelluksen julkaisemiseen.

Markkinoilla on myös tarjolla useita ECP-palveluita, jotka tarjoavat ratkaisuja konttipohjaisten sovellusten ajamiseen, hallinnoimiseen ja integroimiseen. Nämä palvelut usein hyödyntävät Docker-konttiteknologiaa ja Kubernetes-työkalua ohjelmiston pohjana. (Bartoletti ym. 2018, 2) Tämän työn tapauksessa erillisen ECP-palvelun käyttöönotto ei ole tarpeellista, sillä Docker ja Docker Compose ovat riittävät työkalut kohdesovelluksen konttien hallintaan.

### 5.4 Mikropalvelujen ajoympäristö

Aiemmin on päätetty, että kohdeohjelmiston kontittamiseen käytetään Dockeria ja Docker Composea. Dockerin avulla mikropalvelut voidaan paketoida kontteihin, joita sitten hallitaan Docker Composen avulla. Seuraavaksi pitää valita varsinainen ympäristö, jonne kohdesovellus julkaistaan. Useat pilvipalvelut tukevat Docker-pohjaisia ratkaisuja. Suurimmilta pilvipalveluiden tuottajilta kuten Googlelta, Microsoftilta, Amazonilta ja DigitalOceanilta löytyy omia räätälöityjä palveluita kontitettujen sovellusten julkaisemiseen. Näin ollen ei ole suurta merkitystä, minkä palveluntarjoajan ratkaisua päättää ruveta käyttämään. Käytetään tässä työssä DigitalOcean-pilvipalvelua sovelluksen lopullisena ajoympäristönä.

# 6 JULKAISUPUTKEN TOTEUTUS VALITUILLA TYÖKALUILLA

Kappaleessa 5 on tehty katsaus markkinoiden tarjoamiin työkaluihin sekä valittu sopivimmat työkalut kohdesovelluksen julkaisuputken toteuttamiseen. Toteutetaan seuraavaksi julkaisuputki valituilla työkaluilla.

## 6.1 Jatkuvan integraation palvelin ja versionhallinta

Kuten kappaleessa 5.1 on todettu, GitLabin avoimeen lähdekoodiin perustuva ratkaisu toimii samanaikaisesti sekä versionhallintajärjestelmänä että jatkuvan integraation palvelimena. GitLabia voi käyttää joko pilvipalveluna tai SaaS-palveluna. Tässä työssä on valittu, että GitLabia ajetaan itse lokaalisti kontitettuna ratkaisuna. GitLabin voi asentaa Dockerin ja Docker Composen avulla lataamalla verkosta GitLabin virallisen docker-compose.yml-tiedoston, ja suorittamalla tiedosto halutussa ajoympäristössä komennolla ”docker-compose up”. Konttiteknologiaa voi siis hyödyntää paitsi mikropalvelusovelluksien tapauksessa, niin myös monoliittisen sovelluksen kuten GitLabin asennuksessa. GitLab pitää asentaa johonkin domain-nimeen, joten lokaalia ajoympäristöä käytettäessä tulee ajoympäristöön konfiguroida jokin domain-nimi, mikä osoittaa ip-osoitteeseen 127.0.0.1 (localhost).

GitLabin asennus jatkuu luomalla pääkäyttäjätunnukset, jonka jälkeen sen käyttämisen voi aloittaa. Kappaleessa 4.2.6 on todettu, että mikropalvelusovelluksen lähdekoodi kannattaa sijoittaa yksittäiseen repositorioon siten, että eri palvelut ovat omissa alikansioissaan. Näin ollen GitLab-järjestelmään lisätään vain yksi uusi repositorio, ja kappaleessa 3.3.4 määritellyn kohdesovelluksen lähdekoodi lisätään kyseiseen repositorioon.

Seuraavana vaiheena on konfiguroida GitLabiin jatkuvaan integraatioon ja jatkuvaan julkaisuun liittyvät toiminnallisuudet. GitLab tarjoaa monipuolisia vaihtoehtoja liittyen julkaisuputkiin. Ensimmäiseksi tulee määrittää julkaisuputkille niin sanottu ”GitLab Runner”. GitLab Runner on julkaisuputken ”suorittaja”, eli sovellus joka varsinaisesti

suorittaa määritellyn julkaisuputken. Se on täysin erillinen osa verrattuna varsinaiseen versionhallintajärjestelmään. GitLab Runner voidaan asentaa monella eri tavalla, asennetaan tämän työn tapauksessa GitLab Runner lokaaliin Windows-ympäristöön. Asennuksen jälkeen Runner pitää rekisteröidä versionhallintajärjestelmässä. Tämä tapahtuu token-pohjaisen autentikaation avulla. Rekisteröinnin ja asetuksien määrittämisen jälkeen runner on valmiina käytettäväksi.

## 6.2 Sovelluksen yksikkötestaus

Mikropalvelusovelluksen tapauksessa yksikkötestaaminen tarkoittaa sitä, että jokaisen mikropalvelun sisäinen toiminnallisuus varmistetaan tietyllä työkalulla toteutetuilla yksikkötesteillä. Kappaleessa 5.2 on valittu yksikkötestaamisen työkaluiksi jUnit ja Jest, ja itse testien toteuttaminen mikropalvelujen lähdekoodiin on melko suoraviivaista. Spring Boot -palvelujen tapauksessa testit kirjoitetaan jUnit:lla, kun taas React-palvelun testit tehdään Jestillä. Yksikkötestit sijoitetaan erillisiin kansioihin itse sovelluslogiikan lähdekoodista.

Kohdesovelluksen sovelluslogiikan palvelut hyödyntävät Maven-työkalua sovelluksen testaamiseen ja pakkaamiseen. Näin ollen yksikkötestit voidaan suorittaa komennolla ”mvn test”. Mavenia käytetään hyväksi myöhemmin GitLabiin konfiguroitavassa julkaisuputkessa sekä sovelluksen pakkaamisessa että testaamisessa. Käyttöliittymäpalvelun yksikkötestaaminen voidaan myös suorittaa tietyllä komennolla, tässä tapauksessa testaaminen tapahtuu komennolla ”yarn test”.

## 6.3 Sovelluksen kontittaminen

Tarkastellaan seuraavaksi, miten mikropalveluiden kontittaminen käytännössä tapahtuu. Kappaleessa 5.3 on valittu käytettäväksi konttiteknologiaksi Docker. Yksittäisistä mikropalveluista luodaan Docker-kuvat, jonne konfiguroidaan mikropalvelun lähdekoodi, mikropalvelun vaatima ajoympäristö sekä mikropalvelun suorittamiseen vaadittavat

komennot. Docker-kuvan avulla voidaan luoda varsinaisia Docker-kontteja. Samasta kuvasta voi käynnistää niin monta konttia kuin haluaa. Spring Boot –mikropalvelujen tapauksessa Docker-kuvaan lisätään ajoympäristöksi Java Development Kit 8: Alpine. Mikropalvelun suorittamiseen tarvitaan vain Maven-työkalulla mikropalvelusovelluksen lähdekoodista luotu suoritettava jar-tiedosto, joten Docker-kuvassa kopioidaan ainoastaan kyseinen jar-tiedosto kontin sisään lähdekoodista. Viimeisenä vaiheena on suorittaa komento, jolla mikropalvelu saadaan käynnistettyä kontin sisällä. Tässä tapauksessa komento java –jar [suoritettava tiedosto] käynnistää mikropalvelut.

Samanlaista logiikkaa voidaan käyttää myös käyttöliittymäpalvelun kontittamiseen. Tavoitteena on julkaista React-sovelluksen tuotantoversio, eli versio jossa lähdekooditiedostoista on muodostettu staattisia ja pakattuja html, css ja javascript –tiedostoja. Tämä toiminnallisuus voidaan toteuttaa esimerkiksi nginx-palvelimen avulla. Julkaisuputki rakentaa lähdekoodista staattiset tiedostot putken ensimmäisessä vaiheessa. Kontittamisvaiheessa nämä staattiset tiedostot kopioidaan kontin /usr/share/nginx/html-hakemistoon, joka on nginx-palvelimen oletushakemisto staattiselle web-sisällölle. Tämän jälkeen Docker-kuvaan määritetään vielä komento, jolla web-palvelin käynnistetään konttiin.

## 6.4 Sovelluksen integraatiotestaus

Integraatiotestauksen tarkoituksena on varmistaa, että mikropalvelut pystyvät kommunikoimaan keskenään. Monoliittisen sovelluksen tapauksessa sovelluksen eri palvelut sisältyvät samaan koodipohjaan, jolloin integraatiotestaus voidaan tehdä pääsääntöisesti samoilla työkaluilla kuin yksikkötestaaminenkin. Mikropalvelusovelluksen tapauksessa pitää ottaa huomioon se, että integraatiotestausvaiheessa kukin mikropalvelu on omassa kontissaan. Tämän työn julkaisuputken tapauksessa integraatiotestausta varten kunkin mikropalvelun kontti pystytetään samaan Dockerin verkkoon Docker Compose -työkalua hyödyntäen. Docker Composen konfiguraatiotiedostoon (docker-compose.yml) määritetään kunkin kontin nimi, ja tämä määritetty nimi toimii myös kontin tunnuksena Dockerin verkossa.

Integraatiotestausta aloittaessa tilanne on siis se, että kunkin mikropalvelun kontti on pystytetty samaan Docker-verkkoon. Jotta integraatiotestauksen tulos olisi luotettava, täytyy testaus suorittaa kyseisen Docker-verkon sisällä. Tätä varten luodaan uusi väliaikainen integraatiotestauskontti, joka liitetään samaan verkkoon muiden mikropalvelujen kanssa. Tämän kontin sisällä ajetaan sitten Tavern-kirjastolla tehdyt testit, jotka varmistavat, että Docker-verkon sisäiset HTTP-yhteydet toimivat palvelujen välillä. Kun testaaminen on suoritettu, voidaan ylimääräinen kontti poistaa verkosta.

## 6.5 Sovelluksen hyväksymistestaus

Sovelluksen hyväksymistestaus varmistaa, että sovellus toimii loppukäyttäjän näkökulmasta oikealla tavalla. Tässä tapauksessa loppukäyttäjä käyttää kohdesovelluksen käyttöliittymäpalvelua internet-selaimella. Robot Frameworkin sekä Selenium-kirjaston avulla voidaan toteuttaa testaus, jossa käyttöliittymäpalvelun toimivuus testataan selaintasolla. Robot Framework siis simuloi selaimen käyttämistä, ja testiskriptiin voidaankin esimerkiksi määrittää tapahtumia, jossa hakukenttään kirjoitetaan tekstiä ja varmistetaan, että haun mukainen tuote ilmestyy käyttöliittymäpalveluun näkyviin. Hyväksymistestaus suoritetaan kohdesovellukselle suunnitellussa julkaisuputkessa heti integraatiotestauksen jälkeen, sillä jo integraatiotestausvaiheessa sovellukselle pystytettiin testiympäristö Docker Composen avulla. Erona integraatiotestaukseen on kuitenkin se, että hyväksymistestauksessa ei enää huomioida konttien sisäistä verkkoa tai kommunikaatiota millään tavalla. Hyväksymistestauksessa keskitytään ainoastaan siihen, että loppukäyttäjän näkökulmasta sovellus toimii oikein.

## 6.6 Konttien hallinnointi

Kappaleessa 5.3 on päätetty, että mikropalvelusovelluksen konttien hallintaan käytetään Docker Composea. Sen avulla voidaan määritellä ja suorittaa monista Docker-konteista koostuvia sovelluksia. Sovelluksen kontit määritellään konfiguraatiotiedostoon omiin

kohtiinsa, ja konteille voi määrittää erilaisia ominaisuuksia jotka helpottavat sovelluksen julkaisua. Tämän työn tapauksessa esimerkiksi konttien nimeäminen vakioiduilla nimillä helpottaa konttien välisen kommunikaation toteuttamista, sillä kontit näkyvät Docker-verkossa toisilleen omilla nimillään. Tätä käytetään hyväksi muun muassa kappaleessa 6.4 käsitellyssä integraatiotestauksessa.

## 6.7 Sovelluksen julkaiseminen

Kuten aiemmin on todettu, monet palveluntarjoajat mahdollistavat Docker-pohjaisten sovellusten julkaisemisen. Integraatio- ja hyväksymistestausvaiheessa sovellus julkaistiin GitLab-runnerin ajoympäristöön Docker Composen avulla testauksen ajaksi. Kontit rakennettiin sovelluksen muuttuneen lähdekoodin perusteella julkaisuputken aiemmassa vaiheessa. Mikäli integraatio- ja hyväksymistestaus suoriutuu onnistuneesti, julkaistaan konttien kuvat verkossa sijaitsevaan repositorioon, tässä tapauksessa Docker Hubiin. Verkossa olevassa repositoriossa on siis aina saatavilla uusimmat testatut Docker-kuvat mikropalvelusovelluksen eri palveluista. Verkossa olevaa repositoriota voidaan käyttää hyväksi sovelluksen julkaisemisessa tuotantoympäristöön.

Koska mikropalvelusovellus voidaan julkaista kokonaan Dockerin ja Docker Composen avulla, ei tuotantoympäristöön tarvitse konfiguroida juuri mitään muuta kuin toimiva Docker-asennus. Kohdesovellus voidaan laittaa saatavilla julkiseen domain-nimeen konfiguroimalla DNS-asetukset niin, että domain osoittaa tuotantoympäristön ip-osoitteeseen, jossa käyttöliittymäpalvelu ottaa vastaan domainiin tulevat pyynnöt. Julkaisuputken ja tuotantoympäristön pitää myös pystyä kommunikoimaan toistensa kanssa siten, että onnistunut testaus laukaisee tuotantoympäristössä prosessin, jossa Docker Hubista haetaan uusimmat konttikuvat ja sovellus uudelleenjulkaistaan uusimmilla konttiversioilla. Kommunikoinnin voi toteuttaa esimerkiksi ssh-yhteyden avulla.

## 6.8 Julkaisuputken vaiheiden määrittely

Kappaleissa 6.1-6.7 on kuvattu julkaisuputken eri vaiheet ja niiden käytännön toteutus. Seuraavaksi eri vaiheet pitää yhdistää yhdeksi kokonaisuudeksi eli julkaisuputkeksi. GitLabissa ohjelmiston julkaisuputki voidaan kuvata .gitlab-ci.yml-tiedoston avulla. Tiedosto sisältää siis koko julkaisuputken määrittelyn. GitLab Runner suorittaa tiedoston määrittämät vaiheet ja niihin sisältyvät komennot järjestyksessä. Näin ollen tiedostossa pitää ottaa huomioon käytettävän GitLab Runnerin ajoympäristö sekä Runnerin tyyppi. Tämän työn tapauksessa GitLab Runnerin ympäristönä käytettiin paikallista linux-ympäristöä ja tyyppinä komentorivisuorittajaa.

GitLabin julkaisuputken määrittämisessä pitää ottaa huomioon itse julkaisuputken eri vaiheiden toteuttamisen lisäksi myös muita asioita: on esimerkiksi tärkeää, että testiympäristöön luodut Docker-kontit poistetaan aina kun uusi suoritus julkaisuputkesta aloitetaan. Tämä voidaan toteuttaa GitLabin CI-tiedostoon monella tavalla, mutta testauksen tuloksena parhaaksi tavaksi osoittautui puhdistaa kaikki aikaisemman suorituksen resurssit uuden suorituskerran alussa eli ennen yksikkötestien ajamista. Tällä tavalla on täysin varmaa, että testausympäristö jossa GitLab Runner suorittaa julkaisuputkea on puhdas uudella suorituskerralla.

Julkaisuputken eri vaiheiden määrittäminen GitLabin CI-tiedostoon on melko suoraviivaista: tiedoston alussa määritellään selkeästi, mitä eri vaiheita julkaisuputkeen sisällytetään ja sen jälkeen yml-tiedostoon aletaan vaihe vaiheelta määrittelemään käskyjä, jotka GitLab Runnerin on määrä suorittaa. Myös vaihekohtaisia määritelmiä on mahdollista tehdä: esimerkiksi mikäli julkaisuputkea halutaan suorittaa muissakin repositorion haaroissa kuin päähaarassa, voidaan tietyt vaiheet kuten tuotantoympäristöön julkaisu jättää välistä repositorion sivuhaaroissa. Tässä työssä tätä ominaisuutta ei kuitenkaan hyödynnetä, sillä se ei kuulunut julkaisuputken toiminnallisiin vaatimuksiin.

# 7 TULOSTEN ARVIOINTI

Toteutettu julkaisuputki toteuttaa sille asetetut vaatimukset kohdesovelluksen tapauksessa. GitLabin tilastoista voidaan nähdä, että julkaisuputken suorittaminen alusta loppuun kestää noin kolme minuuttia. Jos vertailee tätä aikaa siihen aikaan, joka kehittäjältä kuluisi vaiheiden suorittamiseen käsin, voidaan todeta että julkaisuputki tuo merkittäviä ajallisia säästöjä. Automaattisen julkaisuputken avulla kehittäjät voivat keskittyä luovaan kehitystyöhön ja ongelmien ratkaisemiseen toistuvien ja rutiininomaisten julkaisuun liittyvien tehtävien sijaan.

Kohdesovelluksen arkkitehtuuri ja etenkin sen sisältämien mikropalveluiden lukumäärä vaikuttaa julkaisuputken suoritusaikaan kasvattavasti. Sadan mikropalvelun sovelluksen julkaisemisessa kestää väistämättä pidempään verrattuna neljän mikropalvelun sovelluksen julkaisemiseen. Julkaisuputkesta voidaan kuitenkin tunnistaa sellaisia vaiheita, joita voidaan suorittaa yhtäaikaisesti kaikille mikropalveluille. Esimerkiksi sovellusten yksikkötestaaminen ja pakkaaminen ovat sellaisia julkaisuputken vaiheita, jotka eivät ole riippuvaisia muista mikropalveluista. Lisäksi suuren mikropalvelusovelluksen tapauksessa kannattaa ottaa huomioon se, että mikäli vain yhden mikropalvelun lähdekoodia on muutettu, ei ole tarpeellista yksikkötestata tai pakata jokaista mikropalvelua uudestaan. Myös esimerkiksi kontittamisvaiheessa kontin rakentaminen uudestaan on tällöin tarpeellista vain muutetulle mikropalvelulle. Tällaisilla julkaisuputken optimoinneilla voidaan rajoittaa julkaisuputken kestoaikaa isojen sovellusten tapauksissa.

Kappaleen 4 alussa esitettiin, että ohjelmistokehityksen eri vaiheista kannattaa automatisoida niin monta kuin mahdollista. Julkaisuputken tavoitteena oli automatisoida ohjelmistokehityksen prosesseja ja näin ollen se on yksi DevOps-toimintakulttuurin mukainen työkalu. Familiar (2015) esitti neljä eri osa-aluetta, joilla automaatiota voidaan ottaa käyttöön. Tämän työn julkaisuputken avulla onnistuttiin automatisoimaan muut osa-alueet lukuunottamatta lopullisen tuotantoympäristön luomista ja sen resurssien hallintaa. Pilvipohjaisten resurssien luominen on täysin mahdollista automatisoida, joten se on yksi kehityskohde julkaisuputkelle tulevaisuutta ajatellen.

Yksi tärkeimmistä lopputuloksista julkaisuputken toimintaa tarkastellessa on se havainto, että julkaisuputken avulla pystytään tehokkaasti paikallistamaan mahdolliset virhetilanteet ohjelmistossa. Kohdeohjelmistoa kehittäessä tuli esimerkiksi vastaan tilanne, jossa sovelluksen hyväksymistesti ei mennyt enää julkaisuputkessa läpi. Tieto siitä, missä julkaisuputken vaiheessa virhe on tapahtunut on hyvin arvokas ohjelmistokehittäjälle, koska sen avulla voidaan helpottaa ja nopeuttaa virheen korjaamista. Esimerkiksi hyväksymistestauksessa tapahtunut virhe tarkoittaa, että sovelluksen yksikkö- ja integraatiotestit ovat onnistuneet, ja virhettä voidaan lähteä heti etsimään selaintasolta jossa hyväksymistestaus tämän työn kontekstissa suoritetaan. Kaiken kaikkiaan voidaan todeta, että automaattinen julkaisuputki nopeuttaa ja tehostaa kehitystyötä, ja julkaisuputkesta saatu hyöty moninkertaistuu kohdesovelluksen laajentuessa tai kehitystiimin kasvaessa.

# 8 YHTEENVETO

Kappaleessa 4.3 esitetyn mallin mukaisen julkaisuputken laatiminen työn kohdesovellukselle voidaan katsoa onnistuneen hyvin. Markkinoille tehdyn katsauksen perusteella valitut työkalut osoittautuivat yhteensopiviksi keskenään, ja lopputuloksena saatiin tuotettua julkaisuputki, joka alkaa ohjelmiston lähdekoodin muutoksesta ja päättyy tuotantoympäristöön julkaisemiseen. Julkaisuputken käytännön toteutuksessa ilmeni kehitystyön edetessä monia haasteita:

- GitLab Runnerin ajoympäristön alkuperäinen valinta Windows-ympäristöön osoittautui huonoksi valinnaksi
- Kontitetun ympäristön oman sisäisen verkon olemassaolo vaikeutti integraatiotestauksen suorittamista
- Hyväksymistestausta varten pystytettävän testausympäristön luominen ja resurssien poistaminen osoittautui haastavaksi suunnitella

Oikeastaan ainoa selkeä muutos alkuperäiseen suunnitelmaan julkaisuputken toteutuksessa oli se, että GitLab ja GitLab Runner oli tarkoituksena alun perin ajaa Windowsilla, mutta yhteensopivuusongelmien takia oli parempi ratkaisu siirtyä käyttämään ympäristönä Linuxia. Tämän aihepiirin tutkiminen ei sisältynyt markkinoille tehtyyn katsaukseen, mikä osaltaan aiheutti tehdyn väärän valinnan. Muuten voidaan sanoa, että valitut työkalut olivat yhteensopivia keskenään ja teknistä estettä julkaisuputken kehitystyölle ei syntynyt.

Työ alkoi kohdesovelluksen määrittelemisellä ja toteuttamisella. Työssä olisi ollut mahdollista käyttää myös valmista ja olemassaolevaa mikropalvelusovellusta, mutta toisaalta sovelluksen toteuttaminen työn ohessa antoi tärkeää kokemusta ja osaamista mikropalveluarkkitehtuurista. Julkaisuputken rakentamiseen olisi varmasti ollut haastavampaa ryhtyä valmiin kohdesovelluksen tapauksessa, joten kohdesovelluksen määritteleminen ja tekeminen itse osoittautui hyväksi ratkaisuksi jälkeenpäin katsottuna.

Ennen julkaisuputken rakentamista tehtiin monipuolinen katsaus alan kirjallisuuteen. Kirjallisuutta tarkasteltiin sekä mikropalveluarkkitehtuuriin että DevOps-menetelmiin liittyen. Kirjallisuuskatsauksesta oli monia hyötyjä käytännön toteutukseen liittyen: mikropalveluarkkitehtuurin teknologiavalinnat sekä kontittamiseen liittyneet valinnat tehtiin pääosin kirjallisuuden perusteella. Lisäksi markkinoille tehty katsaus oli helpompaa toteuttaa, kun eri lähteistä saatiin kerättyä monipuolisesti pohjatietoja liittyen eri teknologioihin ja palveluihin. Vaikka monet kirjallisuuden esittelemät aihepiirit ja toiminnallisuudet saatiinkin toteutettua käytännön toteutukseen julkaisputkesta, niin joitakin kirjallisuudessa laajastikin käsiteltyjä aihepiirejä jouduttiin jättämään tämän työn yhteydessä toteuttamatta. Esimerkiksi mikropalvelujen tietovarastoja ei huomioitu julkaisuputkessa ollenkaan, ja lisäksi tiettyjä tietoturvaan liittyviä näkökulmia jätettiin tietoisesti toteuttamatta julkaisuputkeen. Kaiken kaikkiaan voidaan kuitenkin todeta, että kirjallisuudessa esiteltyjä huomioita ja teemoja liittyen mikropalveluarkkitehtuuriin ja julkaisuputkien rakentamiseen on hyödynnetty työn käytännön toteutuksessa kattavasti.

Työssä toteutettu DevOps-periaatteiden mukainen julkaisuputki toteuttaa sille ennalta asetetut vaatimukset. Tämä ei kuitenkaan tarkoita, että julkaisuputkea ei voisi jatkokehittää monella eri osa-alueella: muun muassa parempien lokitietojen saaminen virheellisillä suorituksilla ja versionhallinnan haarojen hyväksikäyttö ovat asioita, jotka on jätetty tämän työn kontekstissa kokonaan huomiotta. Julkaisuputkea voidaan pitää tehokkaana yhden henkilön kehitystiimissä, mutta isommalla kehitystiimillä tietyt parannukset kuten julkisen testiympäristön luominen ja paremmat lokitiedot ovat lähes välttämättömiä.

Mikropalvelusovelluksien monimutkainen rakenne ja monipuolisen testauksen tarve tuovat haasteita sovelluksen luotettavaan julkaisemiseen. Toteutettu yhdeksän vaiheen julkaisuputki säästää kehittäjien aikaa huomattavasti, jos vertaillaan tilannetta jossa kehittäjän pitäisi itse testata ja julkaista sovellus. Monet julkaisuputken vaiheista ovat toistuvia ja rutiininomaisia (esimerkiksi yksikkötestaus ja kontittaminen), joten niiden automatisoiminen julkaisuputken avulla on ehdottomasti järkevää. Kun tarkastellaan tässä työssä toteutettua julkaisuputkea, voidaan todeta että ainoastaan hyväksymistestaus on sellainen vaihe jonka automatisointi julkaisuputkella ei välttämättä ole kannattavaa. Tämä

johtuu siitä, että loppukäyttäjän toimintaa sovelluksessa on hankalaa simuloida ohjelmallisesti. Yksinkertaisten sovellusten, kuten tämän työn kohdesovelluksen tapauksessa kuitenkin myös hyväksymistestaus voidaan nähdä vaiheena joka voidaan automatisoida.

Kootusti voidaan todeta, että julkaisuputkien avulla monimutkainen mikropalvelusovellus voidaan pakata, testata ja julkaista tuotantoympäristöön minuuteissa. Ohjelmiston automaattinen julkaisu mahdollistaa sen, että kehittäjät voivat keskittyä paremmin sovelluksen kehitystyöhön rutiininomaisten testaus- ja julkaisutoimenpiteiden suorittamisen sijaan. Julkaisuputken avulla säästettävä aika moninkertaistuu sovelluksen koon ja kehitystiimin kasvaessa.

# LÄHTEET


Bartoletti, D., Dai, C., O'Donnel, G., Lipson, A., Giron, F., Bao, H., Nagel, B., 2018. The Forrester New Wave™: Enterprise Container Platform Software Suites, Q4 2018.

Chawla, H., Kathuria, H., 2019. Building Microservices Applications on Microsoft Azure : Designing, Developing, Deploying, and Monitoring, 1st ed. 2019. ed. Apress.

Condo, C., LeClair, A., Mines, C., Homan, A., Reese, A., 2017. The Forrester Wave™: Continuous Integration Tools, Q3 2017.

Engels, G., Förster, A., Heckel, R., Thöne, S., 2005. Process Modeling using UML, in: Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M. (Eds.), Process-Aware Information Systems. John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 83–117. https://doi.org/10.1002/0471741442.ch5

Familiar, B., 2015. Microservices, IoT, and Azure : Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions. Apress.

Hoffman, K., 2016. Beyond the Twelve-Factor App: Exploring the DNA of Highly Scalable,Resilient Cloud Applications 72.

Hüttermann, M., 2012. DevOps for Developers. Apress.

Indrasiri, K. kirjoittaja, Siriwardena, P., 2018. Microservices for the Enterprise : Designing, Developing, and Deploying. Apress.

Jangla, K. kirjoittaja, 2018. Accelerating Development Velocity Using Docker : Docker Across Microservices. Apress.

Lwakatare, L.E., Kilamo, T., Karvonen, T., Sauvola, T., Heikkilä, V., Itkonen, J., Kuvaja, P., Mikkonen, T., Oivo, M., Lassenius, C., 2019. DevOps in practice: A multiple case study of five companies. Inf. Softw. Technol. 114, 217–230. https://doi.org/10.1016/j.infsof.2019.06.010

Machiraju, S., Suraj, G., 2018. DevOps for Azure Applications : Deploy Web Applications on Azure. Apress.

Pulkkinen, V., 2013. Continous Deployment of Software. Cloud-Based Softw. Eng.

Ravichandran, A., Taylor, K., Waterhouse, P., 2016. DevOps for Digital Leaders. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-1842-6

Richardson, C., Smith, F., 2016. Microservices - From Design to Deployment. NGINX, Inc.

Rossel, S., 2017. Continuous Integration, Delivery, and Deployment. Packt Publishing.

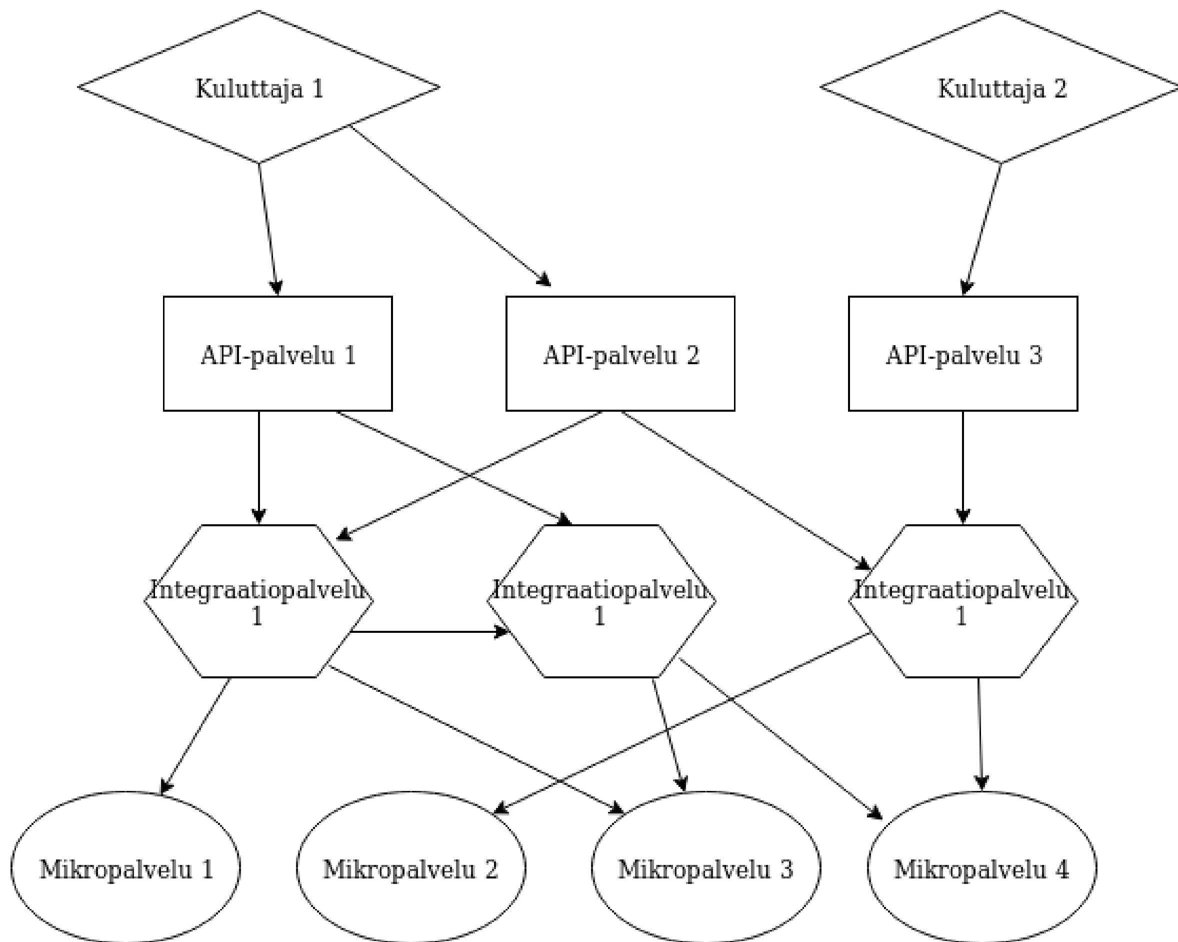
Savchenko, D., 2019. Testing microservice applications. Lappeenranta-Lahti University of Technology LUT.

Savchenko, D., Radchenko, G., Hynninen, T., Taipale, O., 2018. Microservice Test Process: Design and Implementation - CONVERIS Research Information System - LUT University [WWW Document]. URL https://research.lut.fi/converis/portal/Publication/11536257?auxfun=&lang=en_GB (accessed 10.20.19).

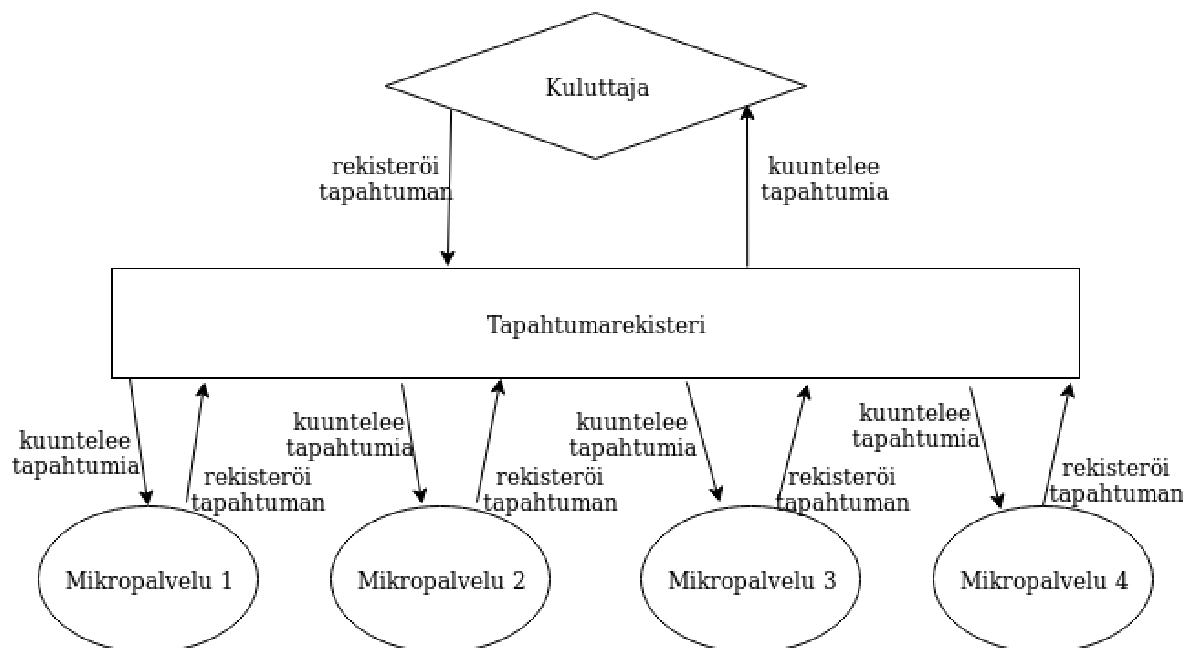
Savchenko, D., Radchenko, G., Taipale, O., 2015. Microservices validation: Mjolnirr platform case study. https://doi.org/10.1109/MIPRO.2015.7160271

Vohra, D., 2016. Kubernetes Microservices with Docker. Apress.

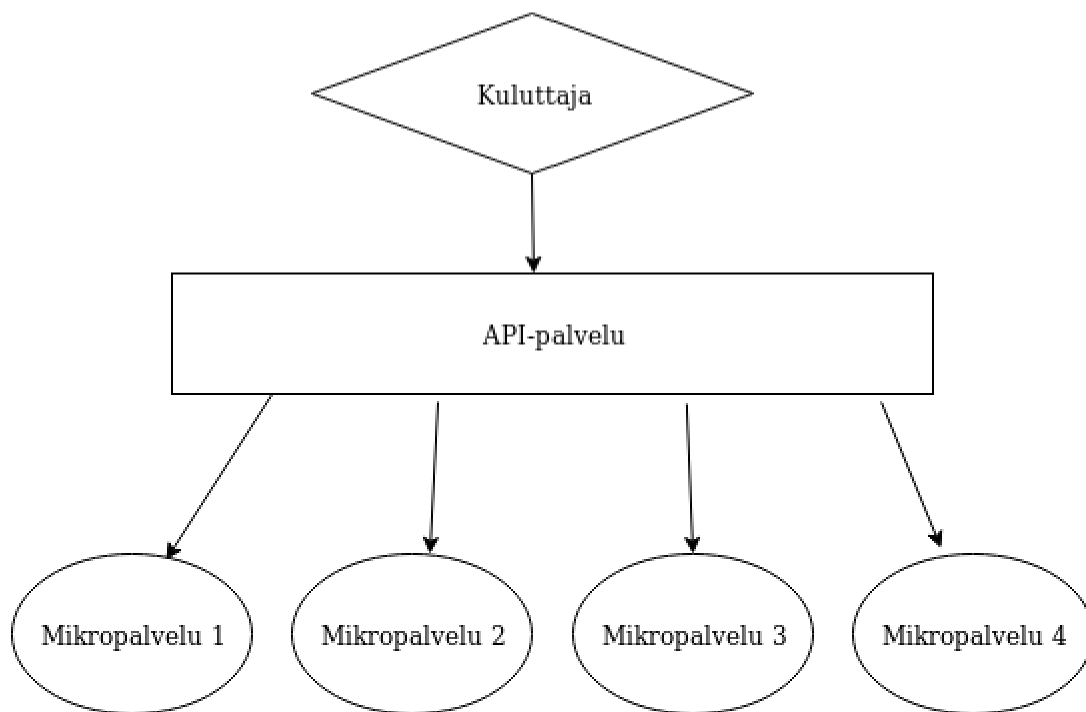
# LIITE 1. Integraatiopalvelu-arkkitehtuuri



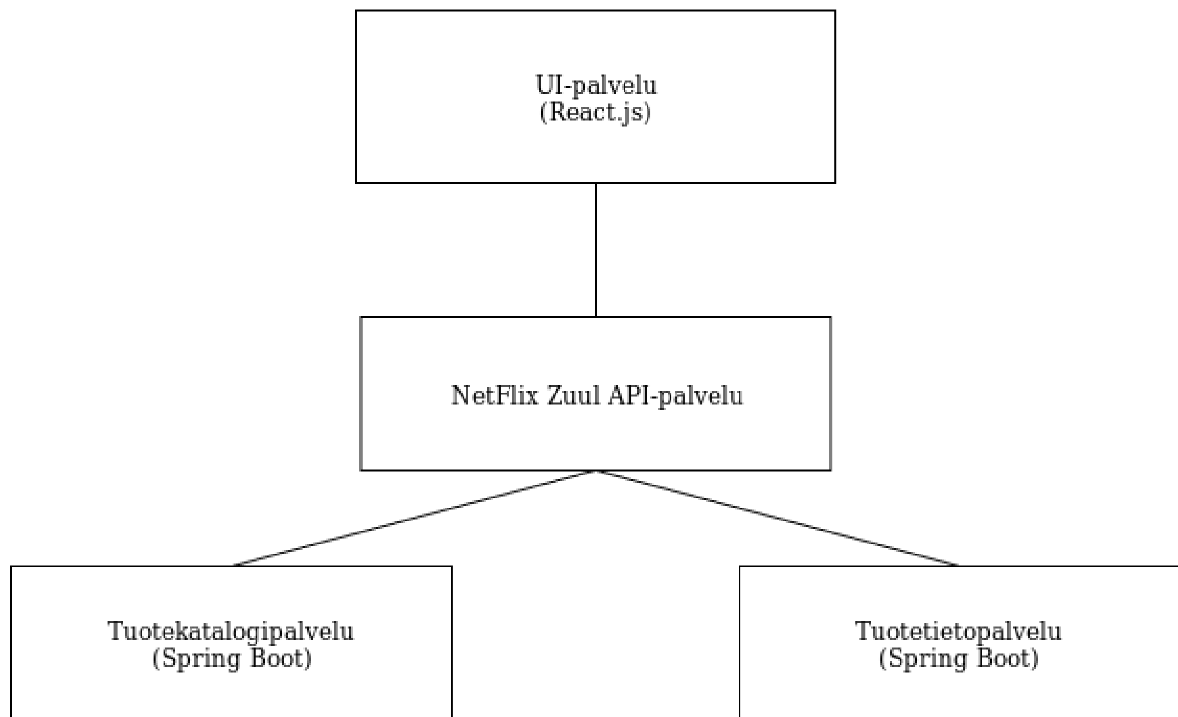
## LIITE 2. Tapahtumapohjainen arkkitehtuuri



**LIITE 3. API-palvelu arkkitehtuuri**



## LIITE 4. Kohdesovelluksen arkkitehtuuri



# LIITE 5. Kohdesovelluksen mikropalvelujen kuvaukset

**Käyttöliittymäpalvelu:**

nimi: ui-service

käytetyt teknologiat: React.js, HTML, CSS, Jest

tehtävä: tarjoaa loppukäyttäjälle selainpohjaisen käyttöliittymän, jolla sovellusta voi käyttää

**API-palvelu:**

nimi: zuul-service

käytetyt teknologiat: Spring Boot, NetFlix Zuul

tehtävä: ohjaa käyttöliittymäpalvelulta tulleet pyynnön varsinaisille mikropalveluille

**Tuotekatalogipalvelu:**

nimi: catalog-service

käytetyt teknologiat: Spring Boot

tehtävä: tarjoaa REST-rajapinnan, josta voi hakea tuotteiden katalogin

**Tuotetietopalvelu:**

nimi: detail-service

käytetyt teknologiat: Spring Boot

tehtävä: tarjoaa REST-rajapinnan, josta voi hakea yksittäisen tuotteen tarkemmat tiedot

**LIITE 6. Julkaisuputken eri vaiheissa käytetyt työkalut**

Vaihe 1 (Muutosten havaitseminen lähdekoodissa): GitLab

Vaihe 2 (Palvelujen yksikkötestaus): jUnit, Jest, Maven, Yarn

Vaihe 3 (Palvelujen lähdekoodin käsittely): Maven, Yarn

Vaihe 4 (Kontittaminen): Docker

Vaihe 5 (Testiympäristön luominen): Docker Compose

Vaihe 6 (Integraatiotestaus): Docker, Tavern, Pytest

Vaihe 7 (Hyväksymistestaus): Robot Framework, Selenium

Vaihe 8 (Konttien julkaisu): Docker, Docker Hub

Vaihe 9 (Julkaisu tuotantoympäristöön): Docker Compose, DigitalOcean