Petri Timperi

JAVA SOVELLUKSEN MIGRAATIO YKSITYISEEN PILVEEN

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Timperi, Petri
Java sovelluksen migraatio yksityiseen pilveen
Jyväskylä: Jyväskylän yliopisto, 2020, 73 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaaja: Seppänen, Ville

Tutkimuksessa selvitettiin mitä organisaation tulee huomioida siirtyessä käyttämään yksityistä pilveä. Mitä pilveen ja mikropalveluarkkitehtuuriin siirtyminen tarkoittaa niin uuskehityksessä kuin ylläpidossa oleville järjestelmille. Miten järjestelmien käyttöönotto pilvialustalle suunnitellaan ja toteutetaan.

Tutkimus toteutettiin kaksiosaisena; mikropalvelu-arkkitehtuuriin siirtymistä selvitettiin kirjallisuus- ja artikkelikatsauksella ja yksityiseen pilveen migraatiota tapaustutkimuksella, jossa vietiin Java sovellus tilaajan OpenShiftillä toteutettuun yksityiseen pilveen. Mikropalvelu-arkkitehtuurista selvitettiin mitkä ovat sen keskeiset periaatteet, mitä suunnittelumalleja sille on olemassa ja miten sovellusten arkkitehtuurimigraatio käytännössä kannattaa tehdä. Yksityisen pilven migraatiossa koestettiin uudelleensijoitus menetelmällä Java sovelluksen vienti OpenShift alustalle, mitä toimenpiteitä se vaatii ja miten alusta palvelee sovelluksen uudelleensijoitusta.

Mikropalvelu-arkkitehtuurista tarkasteltiin palveluihin jakamista, sekä uuskehitys- että migraatioprojektissa. Miten mikropalvelujen myötä tulevat rajapinnat kannattaa suunnitella, mitkä ovat niiden tehokkaat käyttötavat ja miten rajapintojen versiointi hoidetaan. Transaktion hallintaan perehdyttiin eri suunnittelumallien kautta. Erityisesti tarkasteltiin miten ylläpidossa olevien järjestelmien migraatio mikropalvelu-arkkitehtuuriin kannattaa tehdä. Java sovelluksen migraatioon liittyen selvitettiin OpenShift alustan keskeiset resurssit ja miten kokonaisuus toimii sovellusten käyttöönotossa alustalle.

Tutkimuksessa havaittiin, että mikropalvelujen jakamisessa DDD-malli on yleinen ja se soveltuu hyvin pohjaksi, kun palveluita jaetaan pienempiin osiin. Rajapintojen kommunikaatio tyypeissä asynkroninen tapa tuottaa paremman löyhän liitoksen mutta on työläämpi toteuttaa verrattuna synkroniseen kommunikaatioon. Vanhojen järjestelmien migraatiossa painottui inkrementaalisuus ja kuristusmallin käyttö. Pilveen migraatiossa havaittiin, että mikropalvelu arkkitehtuuri on optimaalisin mutta myös uudelleensijoitus-strategialla voidaan saavuttaa konekapasiteetin käyttöaste hyötyjä. OpenShift alustan käyttö osoittautui perehtymistä vaativaksi, mutta alusta ominaisuuksiltaan kattavaksi. Kokemusten pohjalta alustan käyttöön voi suhtautua luottavaisesti, vaikkakin lisätutkimusta aiheesta tarvitaan erityisesti vikatilanteista toipumisen ja suorituskyvyn osalta.

Asiasanat: pilvipalvelut, mikropalveluarkkitehtuuri, docker, openshift

ABSTRACT

Timperi, Petri
Migration of Java application into private cloud
Jyväskylä: University of Jyväskylä, 2020, 73 pp.
Information Systems, Master's Thesis
Supervisor(s): Seppänen Ville

This study was conducted to find out what should be taken into account when taking a private cloud in use within an organization. What it means to migrate to microservices architecture both for new development and for legacy systems. What should be considered when taking into use a private cloud solution.

The study was implemented in two parts; migration to microservices architecture was elaborated by book and article review whereas the migration of a Java application into private cloud was conducted as a case study. The private cloud used in the study was deployed at customer's premises with Red Hat OpenShift technology. Microservices where studied by finding out its fundamental principles, what kind of design patterns there exists and what are the best practices to carry out an architecture migration. The migration into private cloud was experimented by rehosting a legacy Java application to OpenShift platform. It was explored what actions are needed in the process and how the platform supports rehosting.

The microservice decomposition was elaborated for both novel development and migration projects. What is the best way to design interfaces for microservices, how are those used effectively and how to handle the versioning. Transaction handling was investigated through various design patterns describing the domain area. In particular, it was explored how a legacy system should be migrated to microservices architecture. In Java application migration, it was found out what are the key components of OpenShift and how the application is deployed into the platform in practice.

The study discovered that a common model for service decomposition is DDD, which works well when dividing services into smaller domains. In communication types, the asynchronous design yields to better loose coupling, but it is laborious to implement compared to synchronous design. Incremental development and strangler fig pattern were advisable in migration of legacy applications. The microservices architecture was found out to be most optimal for cloud. Still, with plain lift-and-shift migration the savings in computing capacity can be achieved. The usage of OpenShift platform turned out to need some learning prior to utilization, but overall, the platform had extensive and working feature set. Based on the case study, the platform was trustworthy in terms of its features and basic usage. Further studies are needed on customer's domain especially in failover and performance testing areas.

Keywords: cloud, microservices, docker, openshift

KUVIOT

SISÄLLYS

KÄSITEHAKEMISTO

API	Application programming interface. Ohjelmallinen rajapinta jonka kautta palvelua tai järjestelmää voi käyttää.
CRUD	Create, read, update, delete. Tietokannassa käytettävät yleisimmät operaatiot järjestelmän resursseille.
Dockerfile	Tiedosto joka sisältää ohjeen levykuvan luomiseksi docker käännöksen yhteydessä. Ohje sisältää komentoja jolla määritellään mitä komponentteja levykuvaan sisältyy.
HTTP	Hyper-text Transfer Protocol. Verkkoprotokolla jota käytetään verkon solmujen välisessä kommunikaatiossa.
IPC	Inter-Process Communication. Tietokoneen prosessien tapa kommunikoida käyttäen jaettuja resursseja, esimerkiksi yhteistä muistialuetta.
REST	REpresentational State Transfer. Arkkitehtuuri tyyli jonka avulla luodaan ohjelmallisia rajapintoja verkon palveluille.
root käyttäjä	Superkäyttäjä Unix ja Linux systeemeissä joka voi hallita ja konfiguroida palvelimen käyttöjärjestelmän kaikkia ominaisuuksia.
RPC	Remote Procedure Call. Hajautetuissa järjestelmissä käytössä oleva tapa kutsua toisen tietokoneen ja prosessin rutiineja samaan tapaan kuin paikallisia rutiineja.
round-robin	Algoritmi joka jakaa suorituskykyä tai palvelupyyntöjä tasaisesti tietokoneen prosessien tai hajautetun järjestelmän eri osien välillä.
Saatavuus	(eng. *availability*) on ominaisuus, joka ilmentää sitä, kuinka varmasti järjestelmä, laite, ohjelma tai palvelu on tarvittaessa käytettävissä.
URL	Universal Resource Locator. URL:n avulla jokaiselle internetissä olevalla resurssille allokoidaan yksilöllinen osoite.

1 JOHDANTO

Pilvipalvelut ovat tulleet vauhdilla johtavaksi tavaksi tarjota tietojärjestelmäratkaisuja asiakkaille. Ciscon pilvipalveluiden käytön ennusteen mukaan vuonna 2021 yritysten laskentakapasiteetista 92 % tullaan suorittamaan julkisissa tai yksityisissä pilvissä. (Compton 2018).

Pilvipalveluilla tarkoitetaan hajautettua sovellusten ajoalustaa, jossa voidaan skaalata ja hallita sovellusten koko elinkaari (Erl, Puttini & Mahmood 2013, 3). Yritykset ja yhteisöt näkevät pilvipalveluista saavutettavan kustannushyötyjä joustavuuden ja skaalautuvuuden vuoksi. Pilveen voidaan kehittää sovelluksia teknologiariippumattomasti ja joustavasti uusia vanhaksi jääneitä toteutuksia. Palvelulle varattua kapasiteettia voidaan muokata joustavasti tarpeen mukaan, niin ettei sille varata tarpeettomia resursseja. Lisäksi, isojen pilvipalvelutoimijoiden myötä, asiakkaat ovat saaneet käyttöönsä koestettuja ratkaisuja tietoturvan ja muiden teknisten ratkaisujen osalta. Monen toimijan siirtyminen pilveen on kiihdyttänyt migraatiotahtia koska muiden ratkaisut ovat jo pilvessä jolloin integraatio helpottuu menemällä sinne itsekin. (Nickl & Chintalapudi 2018).

Pilveen siirtymisessä on paljon haasteita ja ongelmakohtia, vaikka etujen katsotaankin yleensä olevan suurempia. Longbottom (2017) listaa haasteiksi yhteisen laskentakapasiteetin jakamisen eri toimijoiden välillä jolloin toisen toimijan palveluissa oleva kuorma saattaa vaikuttaa toisen toimijan palvelujen suorituskykyyn. Haasteena on myös se, ettei ajoympäristö ole enää toimijan itsensä käsissä, vaan sen pitää luottaa pilvipalvelun tarjoajan takaamaan laatuun. Mikropalveluihin siirtymisen myötä tulee lisää monimutkaisuutta palvelujen välisten rajapintojen hallinnassa versioinnin ja yhteensopivuuden kannalta. Ongelmien ratkaisu saattaa hankaloitua, kun yksittäisen palvelutapahtuman tuottamiseen osallistuu useita mikropalveluita, jolloin niiden monitorointi vialisen palvelun selvittämiseksi on tärkeää. Lisäksi järjestelmien arkkitehtuurin suunnitteluun tulee kiinnittää aikaisempaa enemmän huomiota, jotta palvelujako ja vastuut ovat kestävällä pohjalla.

Pilvipalvelujen suosion kasvun myötä mikropalvelu-arkkitehtuuri on vallannut alaa järjestelmien kehityksessä. Mikropalveluilla tarkoitetaan loogisiin kokonaisuuksiin jaettuja palveluja, joilla on yksi rajattu tehtävä. Ne ovat löyhällä sidoksella riippuvia muista palveluista ja omistavat ja hallitsevat omat tietonsa. Mikropalvelujen avulla voidaan saada pilvestä enemmän käytännön hyötyjä kapasiteetin hallinnassa ja saatavuudessa. Kun palvelu pilkotaan pienempiin osiin, voidaan sen osille allokoida joustavammin kapasiteettia niin, että paljon käytetty (mikro)palvelu saa enemmän laskentaresursseja ja muistia kuin pienellä kuormalla oleva palvelu. Skaalaamalla palvelua useampaan instanssiin voidaan parantaa myös saatavuutta, kun yhden instanssin kaatuminen ei vaikuta käyttäjän palvelutasoon. Palvelujen atomisempi jako auttaa lisäksi rajaamaan virheen mahdollisuutta pienemmälle osa-alueelle järjestelmässä. Jos jollain osa-alueella järjestelmää tulee ongelma, se ei välttämättä vaikuta muiden osa-alueiden toimintaan. (Laszewski, Arora, Farr, Zonooz 2018, 16-17).

Mikropalveluilla saavutetaan myös muita etuja kuin parempi soveltuvuus pilvipalvelu-ympäristöihin. Niissä olevat suunnittelumallit ovat kehittyneet yhdessä pilvipalveluiden kanssa, kun isojen järjestelmien ylläpito on törmännyt haasteisiin. Pitkään käytössä olleet järjestelmät ovat tulleet monimutkaisiksi, jolloin niiden kehitystyö on käynyt hitaaksi ja kalliiksi. Pienikin muutos on vaatinut paljon työtä, kun järjestelmän sisäiset riippuvuudet on pitänyt ottaa huomioon ja testata monimutkaisuudesta johtuen kattavasti. Palvelujen jakaminen pienempiin osiin, mikropalveluihin on ollut vastaus monimutkaisuuden hallintaan. Kun yksittäinen selkeällä rajapinnalla varustettu palvelu vaatii muutoksia, on sovelluskehittäjän helpompi ymmärtää mitä vaikutuksia palveluun tehdyllä muutoksella käytännössä on. Palvelu voidaan testata automaattisilla testeillä helpommin, eri palvelujen kehitysvastuut voidaan jakaa joustavammin ja palvelu voidaan asentaa erillisenä järjestelmään ja ymmärtää mitä muutos aiheutti koko järjestelmässä. (Richardson 2018, 2-17).

Mikropalveluista saatavien etujen ja pilven kustannustehokkuuden ja skaalautuvuuden myötä monessa organisaatiossa on tullut ajankohtaiseksi miettiä kannattaako omat palvelut viedä pilveen ja miten se tapahtuisi. Tutkimuksessa paneudutaan edellä kuvattuihin haasteisiin, kun pitkään käytössä olleessa järjestelmässä suunnitellaan siirtymistä pilvipalveluun ja vaiheittain mikropalveluiden käyttöön. Työssä tutkitaan voisiko mikropalveluista saada apua tällä hetkellä järjestelmän ylläpidossa oleviin haasteisiin. Mitä pilvipalveluihin migraatio vaatii ja mitä hyviä käytäntöjä pilveen siirtymiseksi on olemassa. Tapaustutkimuksessa viedään yksi kokonaisjärjestelmän osa, Javalla toteutettu käyttöliittymä työn tilaajan OpenShift alustalla toteutettuun pilveen. Tutkimuksen pohjalta voidaan paremmin arvioida kustannuksia ja saavutettuja etuja pilveen siirtymisestä myös muiden tilaajan ylläpitämien palvelujen osalta.

Tutkimuskysymykset ovat:

- Mitä periaatteita ja malleja on olemassa monoliittisen arkkitehtuuriratkaisun migroimiseksi mikropalvelu arkkitehtuuriin.
- Mitä käytännön toimenpiteitä tarvitaan, että sovellus saadaan toimimaan OpenShift alustalla toimivassa yksityisessä pilvessä?

Ensimmäiseen kysymykseen pyritään vastaamaan kirjallisuus- ja artikkelikatsauksella; tutkimalla mikropalveluarkkitehtuurin periaatteita, etsimällä olemassa olevia malleja ja kokemuksia vastaavista migraatioista. Toiseen kysymykseen saadaan vastaus, kun viedään yksi järjestelmän osa OpenShift alustalle, jolloin nähdään mitä muutoksia ja työtä se käytännössä vaatii.

Tutkimuksen tuloksena on tarkoitus saada suuntaviivat tilaajalle pilveen ja mikropalveluihin siirtymisestä. Millaisiin työmääriin migraatiossa on syytä varautua ja mitkä ylipäänsä ovat hyödyt siirtymisestä. Näyttääkö siltä, että tällä hetkellä ylläpidossa kohdattavia ongelmia voitaisiin helpottaa siirtymällä mikropalveluihin. Tapaustutkimuksessa käsiteltävän järjestelmän kokemuksia on tarkoitus hyödyntää myös tilaajan muissa järjestelmissä, joissa osassa on saman tyyppisiä ylläpidon haasteita kuin nyt tarkastelussa olevassa. Toiveena on myös, että tulevaisuudessa tilaajan konesalien kapasiteettia saisi paremmin hyötykäyttöön, kun useammat järjestelmät käyttäisivät tilaajan yksityistä pilveä.

Tutkimus rakentuu niin että ensimmäisen johdantoluvun jälkeen kuvataan toisessa luvussa ensin pilvipalvelut yleisesti ja sitten tarkastellaan erityisesti yksityistä PaaS pilveä.

Kolmannessa pääluvussa siirrytään tarkastelemaan lähemmin ensimmäistä tutkimuskysymystä käymällä ensin läpi eri arkkitehtuurimalleja ja miten mikropalveluarkkitehtuuri on niistä kehittynyt. Luvussa tarkastellaan myös mikropalvelujen etuja ja haasteita verrattuna muihin malleihin. Erityisesti tarkastellaan muiden tapaustutkimusten kautta hyviä käytäntöjä monoliittisten järjestelmien migraatiolle pilvipalveluihin, mitä strategioita ja käytäntöjä on olemassa.

Neljännessä luvussa paneudutaan OpenShift alustan ominaisuuksiin, miten alustaan voidaan provisioida sovelluksia ja miten niitä hallitaan palvelussa. Mitä eri tapoja on ajaa sovelluksia alustassa ja miten sovellukset julkaistaan asiakkaiden käyttöön.

Viidennessä luvussa kuvataan järjestelmän käyttöliittymä-osuuden vienti pilvipalveluun. Miten sovelluksen kontitus tehdään ja mitä kaikkea OpenShiftille vienti vaatii ja mitä pitää ottaa huomioon.

Viimeisessä luvussa analysoidaan tutkimuksen tulokset ja vedetään yhteen johtopäätökset. Arvioidaan mitä tulokset tarkoittavat tilaajan näkökulmasta, miten pilvipalveluihin kannattaisi organisaation näkökulmasta siirtyä. Lisäksi pohditaan mitä uutta tietoa tutkimuksessa selvisi ja mitä asioita pitäisi vielä tutkia lisää.

2 PILVIPALVELUT

Tässä pääluvussa käydään läpi mitä pilvipalvelut ovat, mitä erilaisia käyttöönotto- ja palvelumalleja niissä on. Yleisen esittelyn jälkeen perehdytään erityisesti yksityisen pilven toimintamalliin ja PaaS palvelumalliin. Yksityisestä pilvestä käydään läpi tärkeimmät ratkaisuiden tarjoajat mukaan lukien tutkimuksessa käsitelty Red Hatin OpenShift. Lopuksi esitellään erilaisia tapoja ja strategioita pilvipalveluihin siirtymiseksi.

2.1 Pilvipalvelujen toimintamallit

Pilvipalvelu on hajautettua tiedonkäsittelyä tarjoava kokonaisuus, jossa voidaan provisioida, skaalata ja hallita tietojärjestelmäpalveluja. Ne eivät ole enää uusi asia informaatioteknologian alueella. Niiden käytännön toteutusten historia ulottuu vuosituhannen vaihteeseen, jolloin Salesforce esitteli ensimmäiset yrityksille suunnatut palvelut, joihin voitiin siirtää sovelluksia ajettavaksi. Vuonna 2002 Amazon esitteli niin ikään yrityksille suunnatut palvelunsa, jossa voitiin ajaa sovelluksia, tallentaa dataa ja käyttää tarjottavia palveluita. Laajamittainen kaupallinen käyttö lähti liikkeelle Amazon Web Services tuomisesta markkinoille vuonna 2006. Tänä päivänä pilvipalveluiden ja alustojen tarjonta on kattavaa. Pilvipalvelut kuten Amazon Web Services (AWS), Google Cloud ja Microsoft Azure ovat suuria toimijoita joiden palveluita käyttävät tuhannet yritykset ja yhteisöt. (Erl, Puttini & Mahmood 2013, 26-28).

Pilvipalvelujen käyttöönottomalleja on useita, kaupallisesti tärkeimpänä julkinen pilvipalvelu (public cloud) jollainen esimerkiksi AWS on. Julkinen pilvipalvelu tuotetaan sovitussa laajuudessaan (ks. palvelumallit) palvelun toimittajan toimesta. Organisaatio-asiakkaat ostavat laskutusmallista riippuen käyttöaikaa tai laskentakapasiteettia. Koko alustasta huolehtiminen jää pilvipalvelutoimittajan vastuulle. Toinen vaihtoehto käyttöönottomallille on yksityinen pilvi (private cloud) jossa organisaatio pitää itse huolen koko pilvitoteutuksesta. Se hankkii konekapasiteetin, asentaa käyttöjärjestelmät, virtualisoi ja kehittää

sovellukset. Tyypillisesti yksityiset pilvet pohjaavat johonkin kaupalliseen tai open-source toteutukseen esimerkiksi OpenStack alustaan. Yhdistelmä edellisistä on hybridi pilvi, jossa asiakas käyttää osittain omaa pilveä, ja hyödyntää osittain yhtä tai useampaa julkista pilvipalvelua. Neljäs käyttöönottomalli on yhteisöpilvi (community cloud) joka on yhdessä useamman organisaation kanssa käytettävä pilvi. Yhteisöpilvessä yleensä yksi toimijoista vastaa pilven ylläpidosta. Se on käytössä tyypillisesti julkisyhteisöissä kuten yliopistoissa. (Longbottom 2017, 15-21)

Pilvipalvelut voidaan jakaa ryhmiin myös niiden palvelumallien pohjalta. Erilaisia jakoja on paljon mutta vakiintuneet mallit ovat (Longbottom 2017, 15-21):

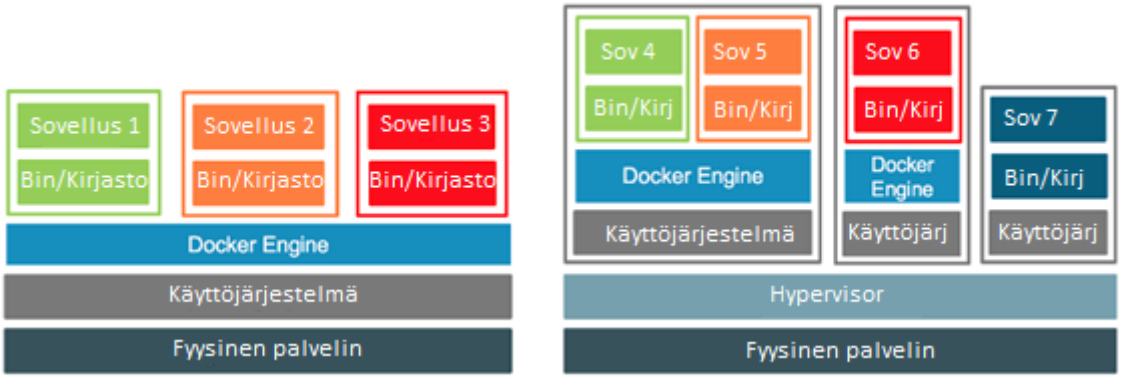
- Infrastructure as a Service (IaaS). Mallissa pilvipalvelu tuottaa laskentakapasiteetin virtuaalikoneiden avulla jota voidaan ostaa tarpeen mukaan ja joustavasti. Asiakkaan vastuulle jää käyttöjärjestelmästä alkaen ohjelmistopinon ylläpito.
- Platform as a Service (PaaS). Palvelu tuottaa alustan jonka päälle asiakas voi kehittää ja asentaa omat sovelluksensa. Alusta pitää huolen käyttöjärjestelmästä, levytilasta sekä tarvittavista apusovelluksista.
- Software as a Service (SaaS). Nopeimmin käyttöön otettavissa oleva malli jossa palvelun kaikki osat tuotetaan pilvipalvelun toimesta, käyttöjärjestelmästä asiakkaan käyttämään sovellukseen.

Tässä tutkimuksessa tarkastellaan erityisesti yksityisen pilven erityisominaisuuksia ja huomioitavia asioita, sekä PaaS mallia jota Red Hatin OpenShift alusta edustaa.

2.2 Pilven toimintaperiaatteet

Pilvipalvelujen arkkitehtuurissa ja ominaisuuksissa on joitakin eroja palvelutarjoajasta ja alustasta riippuen. Yleiset periaatteet ja tarvittavat komponentit ovat kuitenkin samankaltaisia. Pilvipalvelut perustuvat usein virtuaalikoneille joissa käyttöjärjestelmätaso on erotettu fyysisistä koneresursseista niin, että virtuaalikoneen ja sen käyttäjien näkökulmasta kyseessä on oma käyttöjärjestelmäresurssi, joka ei eroa toiminnallisesti sen isäntäkoneen resursseista. Virtualisoimalla voidaan samaa konekapasiteettia jakaa joustavammin eri tarpeisiin. Pilviratkaisuissa hyödynnetään myös vahvasti konttiteknologiaa, jolla palvelut voidaan ajaa muista sovelluksista ja ajoalustasta riippumattomassa ympäristössä. Kontti jakaa muiden konttien kanssa vain virtuaalikoneen ytimen toiminnot mutta omistaa itse käyttöjärjestelmän kirjastot, sovellukset, konfiguraation ja itse ajettavan sovelluksen. Konttien etuna on riippumattomuus ja itsenäisyys, joka takaa, että se toimii samalla tavalla riippumatta missä ympäristössä tai virtuaalikoneella se ajetaan. Konttialusta voidaan asentaa joko virtualisoituihin tai fyysisiin koneresursseihin (Erl, Puttini & Mahmood 2013, 79-112).

Alla olevassa kuviossa on esitetty virtualisoidun ympäristön ja konttien suhde kahdessa eri käyttövaihtoehdossa. Vasemmassa osiossa konttien ajoalusta on asennettu suoraan fyysiselle koneelle. Oikeassa osiossa ympäristö on virtualisoitu hypervisorilla.



Kuvio 1. Virtuaalikoneet ja konttialusta (Coleman 2016)

Muita tärkeitä periaatteita pilvipalveluissa ovat tietoverkkojen virtualisointi, pilven tallennusmedia ja resurssien replikointi. Verkon virtualisoinnilla saadaan erotettua pilvessä eri käyttäjät toisistaan niin että he näkevät vain omat resurssinsa tai ne joihin heillä on oikeus. Pilven tallennusmedia tuottaa virtualisoidun tallennuspaikan pilvessä ajettaville sovelluksille jota voidaan ottaa käyttöön tarpeen mukaan. Tallennusmedia tuottaa samankaltaisia palveluja kuin käyttöjärjestelmän levyjärjestelmä. Resurssien replikoinnilla saavutetaan ajettavalle palvelulle lisää kapasiteettia ja saatavuutta, kun samasta sovelluksesta replikoidaan useampia instansseja. Lisäksi pilvipalvelut sisältävät valmiita resursseja jotka tuottavat mahdollisimman kattavan ympäristön erilaisten sovellusten provisioimiseksi pilveen. Pilvessä voi myös olla valmiiksi asennettuna tietokantaratkaisuja, kehitysvälineitä, hallintatyökaluja sekä alustaresursseja jotka helpottavat sovellusten käyttöönottoa pilvessä. (Erl, Puttini & Mahmood 2013, 139-166).

2.3 Yksityinen pilvi

Yksityinen pilvi, eli private cloud tarkoittaa organisaation itsensä hallinnoimaa pilvipalvelua jossa ajetaan sen omia ohjelmistopalveluja omassa datakeskuksessa. Sen resursseja käyttävät tyypillisesti organisaation kaikki osat ja sen hallinta ja käyttötavat on standardoitu organisaation sisällä. Yksityinen pilvi soveltuu paremmin isommalle organisaatiolle, jolla on resursseja ja osaamista ottaa käyttöön oma pilvi. Pilven käyttöönotolla organisaatio yleensä pyrkii saamaan saatavuus-, tietoturva- ja kustannushyötyjä verrattuna julkiseen pilveen. (Longbottom 2017, 15-21).

Koko pilvipalveluiden markkina julkisista pilvistä yksityisiä pilviä tarjoaviin toimijoihin ja käytettäviin teknisiin ratkaisuihin muuttuu jatkuvasti. Yksityisen pilven ratkaisuja on paljon ja eri tasoisia. Osa ratkaisuista keskittyy konttien orkestrointi kerrokseen ja joidenkin lisähallintavälineiden tuottamiseen. Osa ratkaisuista on kokonaisvaltaisempia sisältäen ohjelmistokehityksen apuvälineitä, monitorointia ja verkkoratkaisuja. Yhteistä kaikille on, että ratkaisut ovat pääosin Open Source-mallilla lisensoitavia. Tärkeimpiä konttien orkestrointiratkaisuja ovat Kubernetes, Docker Swarm ja Mesos jotka keskittyvät palvelujen provisiointiin ja hallintaan pilvessä. Kokonaisvaltaisempia ratkaisuja, jotka käyttävät osanaan edellä mainittuja orkestrointiratkaisuja, ovat CloudStack, Cloud Foundry ja OpenStack. Näistä OpenStack on saavuttanut eniten tukijoita isoista teknologia-alan yrityksistä. Sen takana on yrityksiä kuten IBM, Intel, Dell, VMWare ja niin edelleen. Nykyisin IBM:n omistama Red Hat on tärkeä kumppani OpenStack Foundationille ja pohjaa OpenShift tuotteensa tälle alustalle. Edellä mainittuja Open Source alustoja kaupallistaa useampi yritys joista yhtenä Red Hat OpenShift tuotteellaan. (Longbottom 2017, 41-52).

Yksityisen pilven valintaan on yrityksissä ja yhteisöissä useita syitä. Pilveen siirtymällä voidaan paremmin standardoida yrityksen ICT infrastruktuuri tarjoamalla yhdenmukainen tapa tuottaa uusia palveluja. Yhdenmukaistamisen ansioista yrityksen henkilöresurssien allokointi sujuvoituu, kun eri sovellusten kehittäminen onnistuu samankaltaisella osaamisella. Palvelujen provisiointi ja hallinta ajoympäristöissä helpottuu, kun menettelytavat ovat yhdenmukaisia. Myös palvelimien resurssienhallinta suoraviivaistuu, kun pilvessä ajettavia sovelluksia voidaan dynaamisesti ja joustavasti allokoida ajoon eri koneresursseille. Niiden pystytys nopeutuu, kun jokaiselle uudelle palvelulle ei tarvitse räätälöidä omaa konfiguraatiota joiden hallinta pidemmällä aikavälillä olisi työlästä. Eri työvaiheiden automatisointi helpottuu, kun ne voidaan tehdä samankaltaisilla ratkaisuilla useille eri palveluille. Palvelujen koonti, testaaminen ja asentaminen eri ajoympäristöihin voidaan automatisoida helpommin. (Suzuki H., Kageyama H., Juli Y. 2017, 54).

2.4 Siirtyminen pilveen

Organisaation siirtyminen pilveen on iso ponnistus, jossa on otettava paljon asioita huomioon. Tärkeimpiä päätöksiä on pilven käyttöönottomallin valinta, käytetäänkö julkista kaupallisesti tarjolla olevaa pilveä, vai rakennetaanko pilvi yksityisenä omaan konesaliin tai käytetäänkö näiden yhdistelmää, hybridimallia. Päätös riippuu muun muassa siitä, millaiset resurssit organisaatiolla on pilven hallintaan, onko mahdollista irrottaa työntekijöitä ylläpitämään omaa yksityistä pilviratkaisua ja onko heidät mahdollista kouluttaa niin että pilvestä saadaan haluttu hyöty irti. Vai kannattaako käyttää valmista ratkaisua jossa saadaan julkisen pilven kehittäjien osaaminen omaan käyttöön. Myös tietoturva ja niihin liittyvät lainsäädäntökysymykset voivat olla merkittävä tekijä. Voidaanko esimerkiksi sallia, että organisaation omistamien

järjestelmien data voi sijaita missä maassa tahansa sijaitsevassa konesalissa. Tai onko organisaatiolla jotain erityisvaatimuksia ajoympäristölle joita ei saada julkisten pilvipalvelujen kautta täytettyä. (Laszewski ym. 2018, 8-13).

Toinen tärkeä päätös on palvelumallin valinta. Longbottom (2017, 58-64) mukaan enemmän joustoa ja räätälöintiä sallivia malleja ovat IaaS ja PaaS, joissa sovellukset kehitetään pilveen itse. IaaS:n tapauksessa myös käyttöjärjestelmätason ja hallinnointikerroksen asiat voidaan muokata halutuiksi. SaaS malli taas soveltuu lähinnä valmiiden sovellusten käyttämiseen. Valitusta pilviratkaisusta pitää lisäksi suunnitella mitä sen ominaisuuksista käytetään. Pilvialustoista hyödynnettäviä ominaisuuksia ovat muun muassa (Laszewski ym. 2018, 9-11):

- tietokannat
- tallennusratkaisut (esim. Hadoop)
- hakemistopalvelut
- kuormantasaajat
- välimuistiratkaisut
- raportointi
- versionhallinta säiliöt
- automaatiovälineet
- hakutyökalut (Elastic).

Pilvialustan valinnan lisäksi on tärkeää pohtia mitä sovelluksia viedään pilveen ja miten. Pilven palvelinresurssien hyötykäyttöön liittyvät edut voidaan saada kevyelläkin sovellusten migraatiolla, mutta mikropalveluista saatavat hyödyt vaativat laajempaa sovellusarkkitehtuurin muutosta. Laszewski ym. (2018, 61-62) esittelevät kolme eri strategiaa organisaation sovellusten viemiseksi pilveen:

- Rehost (tai lift-and-shift), sovelluksen uudelleensijoitus minimaalisilla muutoksilla pilveen. Etuna on pieni työmäärä siirtymisessä ja matala riskitaso ongelmille sovelluksen siirron jälkeen. Toisaalta mikropalveluista ja pilvestä ei saavuteta vastaavia, esimerkiksi skaalautuvuus, etuja.
- Replatform (tai lift-tinker-shift), eroaa sovelluksen suorasta uudelleensijoituksesta niin että sovelluksen lisäksi myös muita järjestelmän komponentteja viedään pilveen, esimerkiksi tiedonhallinta ratkaisut. Tämä vaatii mini-refaktoroinnin ja pieniä muutoksia arkkitehtuuriin.
- Refactor (tai re-engineer), tarkoittaa sovelluksen arkkitehtuurin muuttamista parhaiten soveltuvaksi pilveen. Migraatiosta saavutetaan pilvipalveluista saavutettavat hyödyt mutta se on myös suuritöisin ja vaatii tarkkaa harkintaa kannattaako uudelleensuunnittelua ja -toteutusta tehdä vanhoille sovelluksille.

Edellisten kategorisointien lisäksi migraatio pilveen voidaan tehdä lineaarisemman prosessin kautta niin että palveluun tehtävät muutokset tehdään mikropalvelu mallin mukaisesti jättäen perinnejärjestelmä-osuuden vielä entiselleen. Tämän strategian mukaisesti vanha monoliittinen sovellus kuristetaan hiljattain pienemmäksi, kun uudet muutokset tehdään aina mikropalvelumallin mukaiseksi. Kuristaminen voi tapahtua jopa useamman vuoden aikana

ja se vähentää merkittäväksi muutoksesta aiheutuvaa riskiä. (Richardson 2018, 430-432).

3 MIKROPALVELUARKKITEHTUURI JA PILVI

Tässä pääluvussa käydään läpi arkkitehtuurimallien kehitys ja evoluutio mikropalveluihin, sekä määritelmä mitä mikropalvelut ovat. Miten mikropalveluita suunnitellaan ja mitä suunnittelumalleja niille on kehitetty. Sen jälkeen perehdytään erityisesti monoliittisten sovellusten tekniseen migraatioon mikropalveluiksi.

3.1 Arkkitehtuurimallien kehitys ja pilvipalvelut

Ohjelmistoarkkitehtuurien kehitys on ollut jatkuvaa, kun niillä on pyritty vastaamaan uusiin liiketoiminnan vaatimuksiin tai kun uutta mahdollistavaa teknologiaa on otettu käyttöön. Ohjelmistojen noustessa yhä suurempaan rooliin kaikessa toiminnan kehityksessä, niiden koko ja monimutkaisuus on kasvanut. Alun perin yhteen tietokoneen prosessiin toteutettu ohjelmistokomponentti on saattanut vuosien kuluessa paisua niin isoksi ja monimutkaiseksi, että sen toiminnan ymmärtäminen on tullut vaikeaksi kehittäjälle. Myös uusia komponentteja on tarvittu tyydyttämään liiketoiminnan tarpeita, jolloin komponenttien väliset rajapinnat ovat tuoneet entisestään lisää kompleksisuutta ja täten vaikeuttaneet uusien ominaisuuksien kehittämistä. (Laszewski ym. 2018, 16-17).

Monimutkaisuuden hallintaan on kehitetty erilaisia arkkitehtuuri malleja ja periaatteita, joilla sitä on pyritty hallitsemaan. Mallit lähestyvät arkkitehtuuria eri näkökohdista mutta yhteistä niille on, että ne yrittävät selittää ja kuvata järjestelmän toimintaa ylemmällä abstraktiotasolla. Krutchenin (1995) 4+1 mallissa arkkitehtuuri jaetaan erilaisiin näkymiin jotka kuvaavat eri näkökulmia järjestelmän arkkitehtuuriin (Richardson 2018, 35-36):

- *Looginen näkymä (Logical view)* kuvaa ohjelmiston elementit joiden parissa kehittäjät työskentelevät. Oliopohjaisessa suunnittelussa tällä tarkoitetaan luokkia ja paketteja, joilla lähdekoodia jaotellaan loogisiin kokonaisuuksiin.

- *Kehitysnäkymä (Implementation view)* pitää sisällään moduulit ja komponentit joihin on paketoitu suoritettava koodi. Java-kielessä moduuli on yleensä paketoitu jar tiedosto ja komponentti suoritettava jar tai war paketti. Näkymässä kuvataan myös komponenttien ja moduulien suhteet toisiinsa.
- *Prosessinäkymä (Process view)* kuvaa miten komponentit suoritetaan eri prosesseissa ja mitä prosessien välistä kommunikaatiota tapahtuu.
- *Fyysinen näkymä (Deployment view)* jolla jaetaan suoritettavat prosessit eri palvelinresursseille, joko fyysisille tai virtuaalisille. Myös palvelinten välisen tietoliikenteen toteutus eri verkkojen avulla kuvataan tässä näkymässä.

Lisäksi mallissa on kuvattu skenaariot, jotka liittyvät jokaiseen malliin ja kuvaavat mallin sisäisen vuorovaikutuksen sen komponenttien välillä, esimerkiksi loogisessa näkymässä luokkien välillä tapahtuvat interaktiot.

Usein arkkitehtuuria kuvataan loogisen näkymän kautta ja uusien arkkitehtuurien kehitys on usein liittynyt loogisen rakenteen mallin muutoksiin. Yksi yleisimmistä loogisen tason arkkitehtuureista on three-tier, kolmen tason kerrosarkkitehtuuri jolla sovelluksia on jaettu loogisiin osiin. Siinä järjestelmä jaetaan esityskerrokseen (presentation), liiketoimintakerrokseen (backend/business logic) ja tietokantakerrokseen (database). Esityskerroksen tehtävä on tuottaa rajapinta loppukäyttäjälle, esimerkiksi selainkäyttöliittymä jolla käyttäjän syötteet käsitellään. Esityskerros kommunikoi liiketoimintakerroksen kanssa joka käsittelee käyttäjän pyynnöt ja toteuttaa liiketoimintalogiikan joka palvelee käyttäjän tarpeita. Liiketoimintakerros taas käyttää tietokantakerrosta joka huolehtii tiedon säilytyksestä ylemmän kerroksen ohjauksen mukaisesti. Mallissa kerrokset saavat kommunikoida vain lähimmän kerroksen kanssa, eikä esimerkiksi esityskerroksesta saa kutsua suoraan tietokantaa. Kerrosarkkitehtuurista on myös eri variaatioita joissa kerrosten määrä saattaa vaihdella esimerkiksi niin että esitys- ja liiketoimintakerrosten välissä on palvelukerros (service) joka tuottaa ulkoisen rajapinnan jolla liiketoimintakerrosta käytetään. (Richardson 2018, 37-39).

Vaikka kerrosarkkitehtuuri kuvaakin periaatteessa järjestelmän loogisen rakenteen, sen pohjalta on usein organisaatioissa asetettu myös vastuutiimejä niin että yksi tiimi vastaa käyttöliittymästä, toinen liiketoimintakerroksesta ja kolmas tietokannoista. Uusien ominaisuuksien kehityksessä tiimien mukainen vastuunjako on hankaloittanut muutoksen toteutusta, kun samaa liiketoiminnan tarvetta on tehty useampaan komponenttiin eri tiimien toimesta. Ajan kuluessa nämä komponentit ovat myös kasvaneet isoiksi monoliiteiksi joita on hankala ylläpitää ja kehittää uusia toiminnallisuuksia. (Newman 2019, 2-5).

Kerrosarkkitehtuurin etuna on, että järjestelmää voidaan jakaa pienempiin osiin, mutta ominaisuuksien lisääntyessä malli usein on johtanut isoihin vaikeasti ymmärrettäviin komponentteihin. Kovat riippuvuudet kerrosten rajapinnoissa ovat johtaneet siihen, että koko järjestelmä pitää ottaa käyttöön yhdellä kertaa, jotta muuttuneet kerrokset toimivat yhteen. Tämä on johtanut järjestelmien hitaaseen käyttöönottoon, kun kompleksin monoliitin toiminta on pitänyt

varmistaa huolellisella testauksella ennen tuotantokäyttöä. SOA-malli (Service Oriented Architecture) tuli 2000-luvun alussa helpottamaan järjestelmien jakamista modulaarisempiin, itsenäisempiin osiin joita voitaisiin ottaa käyttöön nopeammalla syklillä. SOAn tarkoituksena on purkaa kova riippuvuus komponenttien välillä ja tuottaa mahdollisimman itsenäiset komponentit jotka toteuttavat jonkin liiketoiminnan kannalta merkityksellisen palvelun. SOA palvelut otettiin käyttöön hajautetuissa järjestelmissä, joissa kukin palvelu saattoi sijaita eri virtuaalikoneella omissa prosesseissaan. Palvelut olivat löyhällä sidoksella riippuvaisia toisistaan, tarkoittaen ettei käännösaikaisia riippuvuuksia ollut, vaan rajapinnat toteutettiin web rajapintojen kautta. (Longbottom 2017, 8-9, Richardson 2018, 13-14).

SOA-palvelut toimivat tyypillisesti raskailla verkkoprotokollilla kuten SOAP ja muut web service standardit. Niiden kommunikaatio tapahtuu väliohjelmistossa (middleware) olevien älykkäiden putkien kautta joissa on liiketoiminta- ja sanomanvälityslogiikkaa. SOA palveluilla on myös usein yhteinen tietomalli ja ne jakavat käsittelemänsä datan tietokannassa. Palvelun kokoa ei ole rajattu kovin selvästi ja verrattuna mikropalveluihin palvelun koko on keskimäärin isompi. Edellä mainituista syistä myös SOA-palvelujen kehittämisessä on päädytty usein monoliittiseen, kerralla käyttöönotettavaan järjestelmään joka on ollut monimutkainen ylläpidettävä. (Newman 2015, 8-9)

Kerrosarkkitehtuuri, SOA ja muut aiemmat loogista arkkitehtuuria jakavat mallit johtivat usein monoliittisiin, paljon kovia riippuvuuksia sisältäviin toteutuksiin jotka olivat hankalia ylläpitää. Pilvipalvelujen kehittyessä 2000-luvun alusta lähtien tuli mahdolliseksi jakaa palvelut edelleen pienempiin osiin, kun pilven orkestrointipalvelut hoitivat palvelujen hallinnan. Tällöin alkoi kehittyä ajatus mikropalveluista, joilla voitaisiin jakaa järjestelmää niin pieniin osiin, että yksittäisen palvelun ylläpito olisi yksinkertaista. Lisäksi palvelujen väliset rajapinnat olivat löyhällä liitoksella riippuvaisia toisistaan REST tai jonkin muun kevyen rajapinnan kautta. Vaikka periaatteet mikropalveluissa ovat hyvin saman tyyppisiä kuin SOA-mallissa, juuri yksittäisen palvelun pienempi, paremmin ymmärrettävissä oleva toiminnallisuus ja sanomanvälityksen yksinkertaisuus ovat johtaneet mikropalvelujen suosion kasvuun. (Laszewski ym. 2018, 16-17).

3.2 Mikropalvelut

Mikropalvelut ovat kehittyneet pitkälti SOA-palveluissa käytössä olleiden periaatteiden pohjalta. Molempien perusta on etenkin Eric Evansin (2003) Domain-Driven-Design (DDD) mallissa jossa koko laaja järjestelmä jaetaan rajattuihin konteksteihin, kohdealueisiin (domain). Kohdealueen käsitteet ja periaatteet pätevät vain sen kohdealueen sisällä jossa ne pidetään ristiriidattomina ja hyvin määriteltyinä. Kohdealueella huomioidaan hyvin rajallisesti järjestelmän muiden osien käsitteet ja sisäinen toteutus, ja näin ollen keskitytään pitämään kunnossa vain oma kohdealue. Samoin toimitaan järjestelmän muissa komponen-

teissa, jolloin koko järjestelmä koostuu useista rajatuista konteksteista jotka omistavat omat käsitteensä ja datansa. Kun käsiteltävä data pidetään erillään, ei komponenttien välille synny kovia riippuvuuksia vaan palvelut voivat toimia itsenäisesti toteuttaen sen kohdealueen liiketoimintalogiikkaa. (Amundsen, McLarty, Mitra, Nadareishvili 2016).

Varsinaisesti mikropalveluista alettiin puhua vasta 2010-luvulla kun pyrittiin ketterien periaatteiden mukaisesti pienentämään työerän (batch) kokoa. Haluttiin valmiita ohjelmistopaketteja jokaisen Sprintin päätteeksi, jolloin isojen monoliittien integroiminen nopealla syklillä kävi vaikeaksi. Palvelujen pienentämisen avulla tuli mahdolliseksi kyetä nopeaan kehityssykliin. (Amundsen ym. 2016). Ensimmäisenä mikropalveluista puhui James Lewis vuonna 2012 kun hän luonnehti niitä seuraavasti (Newman 2019, 9-10):

"services should be no bigger than my head
(palvelu ei saisi olla isompi kuin minun pääni)"

Newman (2015, 2-3) taas kuvaa mikropalveluita näin:

"small, and focused on doing one thing well
(pieniä, ja keskittyneitä tekemään yhden asian hyvin")

Richardson (2018, 11) kuvaa mikropalveluja seuraavasti:

"is an architectural style that functionally decomposes
an application into a set of services
(se on arkkitehtuuri tyyli joka jakaa sovelluksen
toiminnallisuuden perusteella palveluihin)"

Määritelmiä ja kuvauksia mikropalveluille on paljon, mutta niissä toistuu säännönmukaisesti, DDD-periaatteen lisäksi, seuraavat ominaisuudet (Newman 2015, 2-8, Richardson 2018, 8-19):

- Ne ovat pieniä ja keskittyneitä tekemään yhden asian hyvin. Niillä on yksi vastuualue jonka osat muuttuvat samasta syystä, usein toiminnallisesta vaatimuksesta. Niiden koko on rajattu suosien mieluummin pienempää kuin isompaa kokoa.
- Ne ovat autonomisia ja ne voidaan provisioida itsenäisinä komponentteina esimerkiksi pilvipalveluun. Palvelu ajetaan omassa prosessissaan ja kommunikaatio tapahtuu tietoverkon kautta muihin palveluihin.
- Ne julkaisevat rajapinnan jonka kautta palvelua voidaan käyttää. Yleensä jollain kevyellä verkkoprotokollalla kuten HTTP REST tai RPC.
- Ne omistavat oman datansa ja kaikki muokkaukset palvelun tietokantaan tapahtuvat sen julkaistun rajapinnan kautta.
- Ne ovat teknologia riippumattomia. Eri mikropalvelut voidaan toteuttaa eri ohjelmointikielillä mutta ne voivat kommunikoida keskenään julkaistujen rajapintojen kautta.
- Ne ovat skaalattavissa helposti provisioimalla useita instansseja palvelusta klusteriin ja saamalla näin lisää kapasiteettia.

- Ne ovat yleensä tilattomia mahdollistaen joustavan provisioinnin klusterin eri solmuille.

Mikropalveluiden ja niiden toteuttaman arkkitehtuurin hyödyt liittyvät pitkälti palvelun tuottamaan selkeään rajapintaan ja yksittäisen mikropalvelun yksinkertaisuuteen. Richardson (2018, 14-17), luettelee eduiksi:

- Mahdollistaa jatkuvan integraation ja käyttöönoton koska yksittäisen palvelun automaattinen testaaminen API rajapinnan kautta on suhteellisen helppoa.
- Yksittäinen palvelu on kooltaan pieni ja yksinkertainen, sekä helposti ylläpidettävissä.
- Palvelut voidaan skaalata itsenäisesti ja käyttää sen vaatimaa laskentakapasiteettia.
- Huonosti toimivan palvelun eristäminen on helpompaa palvelujen välisen heikon riippuvuuden vuoksi. Yksittäinen huonosti toimiva palvelu ei kaada koko järjestelmää.
- Helpompaa kokeilla uusia teknologioita koska yksittäinen palvelu voi olla kehitetty millä tahansa tekniikalla. Palvelujen väliset riippuvuudet ovat vain API tasolla.
- Arkkitehtuuri helpottaa työn jakamista useammalle tiimille jotka voivat työskennellä riippumattomammin toisistaan.

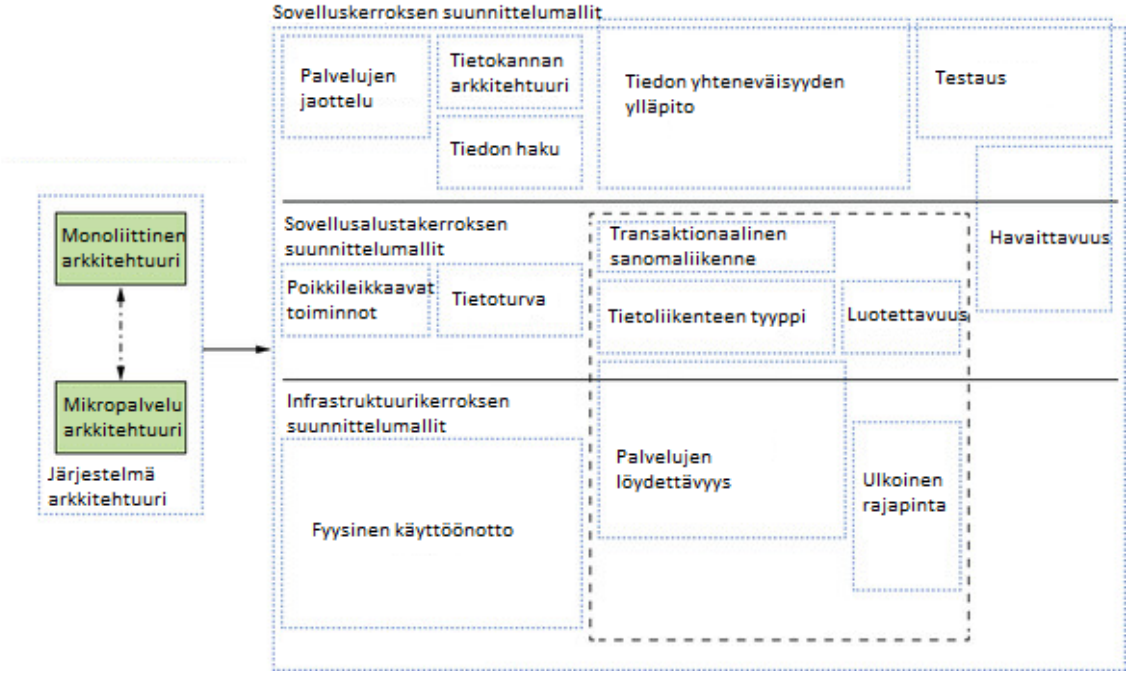
Haasteina mikropalveluissa nähdään palvelujen keskinäisten vastuiden suunnittelun vaikeus, mikä on oikea tapa jakaa järjestelmä itsenäisiksi palveluiksi, mikä on palvelun oikea koko. Hajautetun järjestelmän yleiset kehittämisen haasteet koskevat myös mikropalveluilla kehitettyjä järjestelmiä, hajautettujen ja laajojen järjestelmien kehittäminen vaatii suunnittelulta paljon. Käyttöskenaarioihin osallistuvien palvelujen määrä voi usein olla suuri. Vaaditaan koordinaatiota, jotta voidaan varmistaa, että kaikissa palveluissa tehtävät muutokset ovat linjassa keskenään. Sekin on hyvä huomioida, että mikropalvelut soveltuvat hyvin monimutkaisiin, laajassa käytössä oleviin järjestelmiin jossa suorituskyky ja kapasiteetti ovat oleellisia. Ne eivät välttämättä sovellu parhaiten palvelun ensimmäisen version tekemiseen, joka voidaan tehdä nopeammin, vaikka monoliittisella client-server toteutuksella ja katsoa miten palvelun kysyntä kehittyy. (Richardson 2018, 17-18).

3.3 Suunnittelumallit ja -periaatteet

Mikropalvelujen suunnittelussa arkkitehtuuria lähestytään eri näkökulmista aiemmin mainitun Krutchenin mallin mukaisesti, jotta kaikki arkkitehtuurin osa-alueet tulevat huomioiduksi. Looginen näkymä ohjaa suunnittelemaan palvelujen jakoa ja niiden sisäistä rakennetta, kehitysnäkymä sitä miten palvelut jaetaan käyttöönotettaviin komponentteihin, prosessinäkymä miten komponentit ajetaan eri prosesseissa ja miten ne kommunikoivat keskenään sekä fyysinen

näkymä päättämään millä palvelinresursseilla prosessit ajetaan (Richardson 2018, 35-37).

Richardson (2018) esittelee mikropalvelu arkkitehtuurin eri osa-alueet suunnittelumallien kautta. Suunnittelumallilla tarkoitetaan hyväksi havaittua ratkaisua tiettyyn ongelmaan, jota voidaan uudelleen käyttää toisen, saman tyyppisen ongelman ratkaisemiseksi. Alla olevassa kuviossa on esitelty mallin keskeiset osatekijät:



Kuvio 2. Mikropalvelu arkkitehtuurin suunnittelumallit (Richardson 2018, 23)

Richardsonin (2018 ja 2019) mallissa mikropalvelu arkkitehtuuri jaetaan kerroksittain eri suunnittelumallien ryhmiin jotka ratkaisevat jonkin tietyn osa-alueen arkkitehtuurissa:

Sovelluskerroksen suunnittelumallit (application patterns)
Mallit ratkaisevat ensisijaisesti sovelluskehittäjän kohtaamat ongelmat. Kerroksen suunnittelumallien ryhmiä ovat:

- Palvelujen jaotteluun (decomposition) tarkoitettu ryhmä jossa esitellään mitä eri tapoja on jakaa laaja järjestelmä eri mikropalveluihin.
- Tiedonhallinta ryhmä (data management) joka kuvaa miten järjestelmän tietoja hallitaan, onko jokaisella palvelulla oma tietokanta ja miten tiedon integriteetti varmistetaan.
- Testaus (testing) ryhmä jossa on suunnittelumalleja mikropalvelun testaamiseksi käyttäjän tai asiakassovelluksen toimesta.

Sovellusalustakerroksen suunnittelumallit (application infrastructure patterns)
Mallit ratkaisevat sekä sovelluskehityksen, että infrastruktuurin suunnittelun kohtaamat ongelmat. Kerroksen ryhmät ovat:

- Transaktionaalinen sanomaliikenne (transactional messaging) jossa on suunnittelumalleja tietokannan transaktiosta aiheutuvan sanomaliikenteen liikenteen toteuttamiseen.
- Tietoliikenteen tyypit (communication style) ryhmässä on malleja, miten prosessien välinen sanomaliikenne toteutetaan, käytetäänkö esimerkiksi REST tai SOAP pohjaisia sanomia. Miten palvelu käyttäytyy, jos se saa saman sanoman useaan kertaan ja millä tavalla sanomien välitys toteutetaan.
- Luotettavuus ja tietoturva (reliability and security) ryhmissä käsitellään miten palvelut käyttäytyvät ylikuormitustilanteessa ja miten ne voivat varmistaa, että käyttäjällä on oikeus käyttää palvelua.
- Havaittavuus ryhmässä (observability) on suunnittelumalleja palvelujen toiminnan monitorointiin. Miten saadaan tieto, jos palvelu ei toimi odotusten mukaisesti, miten palvelun käytöstä kerätään statistiikkaa tai millä tavalla havaitut virheet käsitellään.
- Poikkileikkaavat toiminnot (cross-cutting concerns) jossa on suunnittelumalleja sovellusten konfiguraatiodatan tuottamiseksi ajon aikana tai alusta-vaihtoehtoja mikropalvelujen tuottamiseksi kuten Spring boot tai Gizmo.

Infrastruktuurikerroksen suunnittelumallit (infrastructure patterns)
Infrastruktuuriin liittyvien kysymysten suunnittelumallit ratkaisevat kehitystyön ulkopuolella olevia ongelmia. Tästä kerroksessa löytyy seuraavia suunnittelumallien ryhmiä:

- Ulkoinen rajapinta (external API) jossa on ratkaisuja rajapintojen hallintaan, miten palvelut versioidaan ja muutokset hallitaan. Miten tuotetaan mikropalvelujen granulaarisista rajapinnoista eri asiakkaiden tarpeisiin soveltuvia rajapintoja.
- Fyysisen käyttöönoton (deployment) ryhmä jossa on erilaisia malleja, miten mikropalvelut provisioidaan klusteriin. Ajetaanko palveluita konteissa vai virtuaalikoneilla, tai onko käytössä palvelupyyntö perustainen (serverless) instanssien käynnistäminen.
- Palvelujen löydettävyyden (service discovery) suunnittelumallit joissa annetaan ratkaisuja, miten mikropalvelujen instansseihin pääsee käsiksi yhden päätepisteen kautta. Rekisteröikö kukin palvelu itsensä palvelurekisteriin vain löytääkö rekisteri uudet palvelut itsenäisesti.

Richardsonin lisäksi erilaisia suunnittelumalleja ja periaatteita on julkaistu eri pilvipalvelujen tarjoajien ja konsultoivien ohjelmistoarkkitehtien toimesta sekä yliopistotutkimuksissa. Hyvä kirjallisuuskatsaus löytyy Taibi, Lenarduzzi &

Pahet (2018) tutkimuksesta jossa viitataan myös Richardsonin kehittämään malliin.

Kirjallisuudessa (Picozzi, Hepburn, O'Connor 2017, Laszewski ym. 2018, Newman 2015) usein viitattu suunnittelumalli tai kokoelma mikropalvelujen periaatteita on Herokun tuottama (kirjoittaja Adam Wiggins) ”The Twelve Factor App”(12factor). Se on Richardsonin mallia yksinkertaisempi kokoelma mikropalvelujen suunnittelun periaatteita. Periaatteet löytyvät yleisesti alan kirjallisuudesta, muun muassa edellä mainituista lähteistä ja ne tuottavat kompaktin ohjeistuksen mikropalvelujen suunnitteluun (Newman 2015, 18).

12factor suunnitteluperiaatteet sisältävät nimensä mukaisesti 12 periaatetta jotka on huomioitava mikropalvelujen suunnittelussa (Wiggins 2017):

I. Koodin säilytyspaikka (codebase). Mikropalvelulla on oltava yksi ainoa koodin säilytyspaikka, jos useampi sovellus jakaa saman säilytyspaikan, palvelu ei ole yhteensopiva 12factor periaatteiden kanssa. Jos palvelulla on yhteistä koodia muiden palvelujen kanssa, pitää tämä koodi kirjastoida omaan säilytyspaikkaan ja viitata siihen riippuvuuksienhallinnan kautta. Vaikka palvelulla on vain yksi koodipohja, voidaan se provisioida useaan instanssiin ajoympäristössä.

II. Riippuvuudet (dependencies) tulee määritellä explisiittisesti ja linkata sovellukseen mukaan. 12factor periaatteiden mukainen sovellus ei koskaan luota ajoympäristössä olevien kirjastojen tai työkalujen olemassaoloon, vaan linkkaa ne mukaan ajettavaan sovellukseen. Uuden kehittäjän tarvitsee asentaa vain kehitysympäristö ja riippuvuuksien hallintatyökalu, ajaakseen sovellusta ympäristössään. Muuta konfiguraatiota ei tarvita.

III. Konfiguraatio (config) tallennetaan aina ympäristömuuttujiin, ei koskaan koodissa oleviin muuttujiin tai vakioihin. Ympäristömuuttujien arvot voivat vaihtua ympäristöstä toiseen mutta muutokset eivät vaikuta koodiin millään lailla. Periaatteen mukaisuus voidaan testata pohtimalla voiko sovelluksen julkaista avoimena lähdekoodina ilman että esimerkiksi tietokannan salasana paljastuu.

IV. Alustapalvelut (backing services) tulisi käsitellä kiinteinä, löyhän sidoksen resursseina mikropalvelun näkökulmasta. Jos esimerkiksi palvelun käyttämän tietokanta vaihtuu, sen ei tulisi aiheuttaa palveluun koodimuutoksia. Ainoastaan resurssin ympäristömuuttujan arvo muuttuu, jolla viitataan tietokannan sijaintiin.

V. Kokoa, julkaise ja aja (build, release, run). Periaatteella erotetaan sovelluksen koonti- ja ajovaiheet tiukasti toisistaan. Koontivaiheessa sovelluksen koodi ja riippuvuudet käännetään ajettavaksi binääritiedostoksi. Julkaisu-vaiheessa otetaan edellisestä vaiheesta saatu binääri ja yhdistetään se konfiguraation kanssa, jolloin se on valmiina provisioitavaksi ajoympäristöön. Ajovaiheessa sovellus provisioidaan ajettaviksi prosesseiksi ajoympäristöön. Periaatteeseen kuuluu, ettei sovelluksen koodia voi muokata enää ajoympäristössä ja jokainen julkaisu tunnistetaan uniikin tunnisteen pohjalta.

VI. Prosessit (processes). Sovellukset ovat tilattomia ja ne ajetaan yhdessä tai useammassa prosessissa. Sovelluksen käsittelemä data tallennetaan alustapal-

velun tuottamaan tallennuspaikkaan josta sovelluksen kaikki prosessit pääsevät siihen käsiksi. Periaatteiden mukainen palvelu ei koskaan oleta saavansa mitään tietoja ajoympäristön välimuistista tai levyltä, ainoastaan sen itsensä omistamasta tallennuspaikasta. Myös niin sanotut tahmeat sessiot (sticky sessions), joissa käyttäjän pyynnöt ohjautuvat aina samalle prosessille, ovat 12factor periaatteiden vastaisia.

VII. Tietoliikenneportin sidonta (port binding). 12factor sovellus käyttää samaa porttia ajoympäristöstä riippumatta. Portin sidonta ulkopuolelle näkyvään rajapintaan tehdään ajoympäristön reitityskerroksessa sovelluksesta riippumattomasti.

VIII. Skaalautuvuus (concurrency). Sovelluksen prosessien skaalautuvuus toteutetaan unix-järjestelmän prosessimallin pohjalta. Lisää kapasiteettia saadaan ajamalla sovelluksesta useampia instansseja jotka käsittelevät palvelupyyntöjä. 12factor sovellus luottaa käyttöjärjestelmän prosessinhallintaan (esim. systemd) uusien prosessi-instanssien käynnistämisessä.

IX. Kertakäyttöisyys (disposability). Palvelun prosessit ovat kertakäyttöisiä ja niitä voi käynnistää nopeasti uusia. Jos yksi sovelluksen prosessi kaatuu, uusi käynnistyy ja suorittaa kaatuneen prosessin palvelupyynnön loppuun. Prosessit pysähtyvät ja käynnistyvät hallitusti niin, ettei palvelupyyntöjä jää käsittelemättä.

X. Ajoympäristöjen yhdenmukaisuus (dev/prod parity). Kaikki ajoympäristöt pidetään samanlaisina, aina kehittäjän koneelta tuotantoon. Usein kehityksessä on halu ottaa käyttöön jotain kevyempiä, helpommin asennettavia ja konfiguroitavia tietokantoja tai välimuisti-ratkaisuja. 12factorin mukaisessa kehittäjän ympäristössä on samat ratkaisut kuin muissa testiympäristöissä ja tuotannossa.

XI. Lokitiedostot (logs). Sovelluksen lokit käsitellään aina tapahtumavirtana sovelluksen näkökulmasta, ei esimerkiksi tiedostoina. Sovelluksen tulee huolehtia ainoastaan, että lokit ohjataan järjestelmän stdout:iin. Ajoympäristö ottaa stdout:sta vastaan tapahtumavirran joka viedään esimerkiksi levylle tiedostoksi tai johonkin lokien indeksointi- ja tallennusjärjestelmään, kuten Splunk tai Graylog.

XII. Ylläpitäjän prosessit (admin processes). Sovelluksen ylläpito suoritetaan samassa ympäristössä, jossa itse sovellustakin ajetaan. Ylläpitoon tarvittavat välineet ja skriptit sijaitsevat samassa koodipohjassa jossa sovelluksen koodi ja ne kootaan julkaisuun yhdessä. Tällöin ylläpidon skriptit ovat yhteensopivia ajettavan sovelluksen kanssa ja kykenevät näin muuttamaan ajoympäristöä tai konfiguraatiota oikealla tavalla.

Edellä kuvatut suunnittelumallit ja – periaatteet antavat hyvän pohjan mikropalvelujen suunnittelulle. Osa koskettaa enemmän mikropalvelujen ajoalustaa, Richardsonin mallin mukaisesti, osa ohjaa järjestelmän palvelujen suunnittelua. Seuraavassa luvussa käydään läpi mitä järjestelmän mikropalvelujen suunnittelussa on käytännössä huomioitava.

3.4 Mikropalvelujen suunnittelu

Mikropalvelujen suunnittelu on laaja alue joka kokonaisuutena on esitetty Kuvio 2 (Richardson 2018, 23). Richardsonin kuvaamista kerroksista infrastruktuurin toteutus tulee pääasiassa pilvialustasta. Sovellusalusta-kerroksen toteutuksesta myös iso osa on ratkaistu alustassa. Tässä luvussa käydään läpi keskeisimmät sovelluskerroksen suunnittelu kysymykset sekä sovellusalusta kerroksesta niiltä osin kuin ne koskevat järjestelmän arkkitehtuurin suunnittelua, eli ovat sovelluskehittäjien ratkaistavissa.

3.4.1 Mikropalveluihin jako

Paljon keskustelua aiheuttava kysymys mikropalvelupohjaisen järjestelmän suunnittelussa on, miten järjestelmän kohdealue jaetaan palveluiksi. Kannattaako palvelut jakaa esimerkiksi toteutusteknologian, tai tiimien sijainnin mukaan. Kannattaako korkeaa suorituskykyä vaativat osat järjestelmästä erottaa omiksi palveluiksi ja toteuttaa ne C tai Go kielellä. Tai kannattaako palvelu jakaa esimerkiksi niin että Pohjois-Amerikassa oleva tiimille on omat palvelunsa ja Euroopassa sijaitsevalle tiimille omansa. (Amundsen ym. 2016, 62-66).

Edellä kuvatut tavat muodostaa mikropalveluita varmasti näyttelevät osaa lopullisen järjestelmän arkkitehtuurissa, mutta hyvin yleisesti omaksuttu pääperiaate tulee Evansin DDD mallista (mm. Newman 2015, Richardson 2018). Siinä järjestelmä jaetaan useampaan rajatun kokoiseen kohdealueeseen yhden koko järjestelmän käsittävän alueen sijasta, jolloin rajatusta kohdealueesta tulee helpommin ymmärrettävä. Evans jakaa järjestelmän rajattuihin konteksteihin, joissa pätevät yhteneväiset käsitteet ja joilla on oma rajattu vastuualueensa. Sen sijaan, että koko järjestelmä mallinnettaisiin kokonaisuutena, se mallinnetaan useaksi rajatuksi kontekstiksi jolloin kokonaisuus muodostaa joukosta alijärjestelmien malleja. Tämä ajatus osuu erityisen hyvin mikropalvelujen periaatteisiin koska sen avulla voidaan löytää koko kohdealueelta toisistaan mahdollisimman riippumattomia konteksteja, joista voidaan rakentaa omat mikropalvelut. Konteksti sisältää käsitteitä ja dataa joiden ei tarvitse näkyä sen ulkopuolelle sekä käsitteitä joiden kautta se kommunikoi palvelun ulkopuolelle. Ulkopuolelle näkyvät käsitteet ovat rajattuja ja hyvin määriteltyjä jolloin käyttäjä voi niiden avulla käyttää palvelua. Kontekstin sisäisistä käsitteistä tai rakenteesta taas käyttäjän ei tarvitse tietää mitään jolloin järjestelmän kokonaisuus on helpommin ymmärrettävissä, vain ulkopuolelle näkyvät käsitteet tulee mallintaa järjestelmän tasolla. (Newman 2015, 31-37).

Rajatun kontekstin käsite edesauttaa mikropalveluissa tärkeää periaatetta, löyhää liitosta (loose coupling), jonka avulla pyritään vähentämään mikropalvelujen riippuvuutta toisistaan. Sen keskeinen idea on, että jos yhtä palvelua joudutaan muuttamaan liiketoimintatarpeen vuoksi, sen ei pitäisi aiheuttaa muutoksia muissa palveluissa. Periaatetta auttaa rajatun kontekstin käsite. Kun palvelu toteuttaa vain yhden kontekstin, s.o. liiketoiminnan osa-alueen, on to-

dennäköistä, että sen muutos ei aiheuta muutoksia muissa palveluissa. Esimerkiksi kerrosarkkitehtuuri ei tyypillisesti täytä tätä vaatimusta vaan liiketoiminnan muutos aiheuttaa muutoksen kaikkiin arkkitehtuurin kerroksiin. (Newman 2015, 3).

Tärkeä periaate mikropalveluissa on myös pyrkiä korkeaan koheesioon. Se tarkoittaa, että samankaltainen toiminnallisuus sijaitsee samassa paikassa, mikropalvelujen tapauksessa osana palvelun lähdekoodia. Kun toiminnallisuus muuttuu, se pitää muuttaa vain yhdessä paikassa sen sijaan, että muutettaisiin samaa toimintoa useassa eri paikassa. Tällä vältetään tarvetta julkaista liiketoimintamuutoksen vuoksi useista palveluista uusia versioita joka aiheuttaa lisää integraatiotyötä ja potentiaalisia ongelmia. Rajaamalla palvelun konteksti tarkasti liiketoimintamallin osa-alueiden mukaisesti, edesautetaan myös löyhän liitoksen ja korkean koheesion periaatteiden toteutumista. (Newman 2015, 30).

Vaikka rajatun konteksti sääntö auttaa purkamaan kokonaisuutta pienempiin osiin, ei mikropalvelujen käyttö pidä olla myöskään itseisarvo. Usein uutta järjestelmää suunnitellessa liiketoimintamallin eri osa-alueet vaihtuvat tiuhaan, jolloin mikropalveluihin jakaminen aikaisessa vaiheessa kehitystä voi johtaa ongelmiin. Jos mikropalvelujen määrää ja vastuita joudutaan muuttamaan paljon, tulee ylimääräistä työtä palvelujen jakamisesta ja yhdistämisestä, aina koodauksesta automatisointiin ja automaattiseen asentamiseen. Newman esittää, että mikropalvelut eivät välttämättä ole paras valinta uuskehityksen alusta lähtien. Järkevämpää on aloittaa sisäisesti modulaarisella monoliitillä, jossa eri moduulien vastuita ja toimintaa on helpompi muokata. Kun järjestelmä saavuttaa kypsemmän vaiheen ja liiketoimintamallit stabiloituvat, voidaan moduuleja alkaa purkaa omiksi mikropalveluiksi (Newman 2015, 33-37).

Lisäksi palvelujen jaossa pitää ottaa huomioon organisaation rakenne, ja toisaalta organisaation rakenteessa palvelujen jako. Conwayn (1968) lain mukaan järjestelmän arkkitehtuuri seuraa sen rakentaneen organisaation rakennetta. Asiaa tutkittu myös tämän jälkeen ja MacCormack, Rusnak, Baldwin (2007) havaitsi että löyhästi sidotut organisaatiot jotka kehittivät tietojärjestelmiä open-source yhteisön kaltaisesti, tuottivat enemmän löyhän sidoksen periaatteita noudattavia järjestelmiä, kuin tiukemmin integroidut organisaatiot. Newman ehdottaa teoksessaan, että organisaatiorakenteet tulisi ottaa huomioon palvelujen vastuiden jakamisessa. Kahden eri lokaatiossa sijaitsevan tiimin kommunikaatio muutoksen suunnittelussa aiheuttaa ylimääräistä koordinaatiota joka heikentää tiimien tehokkuutta. Paras ratkaisu on allokoida DDD-mallin mukaan jaettuja mikropalveluja samoissa lokaatioissa oleville tiimeille. Toisaalta vaikutus on molempiin suuntiin. DDD-mallin mukaisesti jaetut mikropalvelut myötäilevät organisaation liiketoimintamallia, jolloin organisaation rakenteen tulisi tukea tätä mallia mahdollisimman hyvin. Joka tapauksessa, organisaatio tulee ottaa huomioon palveluihin jakamisessa niin että tiimi kykenisi kehittämään muutoksia mahdollisimman itsenäisesti. (Newman 2015, 191-196, Amundsen ym. 2016, 104-105).

3.4.2 Rajapintojen suunnittelu

Mikropalvelut kommunikoivat palvelun ulkopuolelle hyvin määritellyn rajapinnan kautta. Palvelujen, eli prosessien väliseen kommunikaatioon on paljon vaihtoehtoja, binääripohjaisista (esim. Avro) tekstipohjaisiin (esim. HTTP) protokolliin. Tänä päivänä yleinen uuskehityksessä on HTTP-pohjainen REST (Richardson 2018, 65-66, Newman 2015, 49-51). REST tyyliä käytetään usein JSON muodossa ja se on hyvin yhteensopiva HTTP protokollassa käytettyjen metodien (POST, PUT, GET jne) kanssa. RESTin käyttötapoja on paljon mutta sen perusperiaate pohjaa resurssien käyttöön jotka yleensä kuvaavat jonkin liiketoiminnan kohteen. Resursseihin viitataan URL-osoitteella ja niiden tilaa käsitellään käyttämällä HTTP:n metodeja. (Richardson 2018, 71-74).

Richardson L. (2010) on kuvannut REST tyylin erilaiset käyttötavat kypsyysmallin avulla:

- Taso 0: HTTP protokollaa käytetään vain siirtokehyksenä kommunikaatiossa ja eri interaktioiden käsittely tehdään palvelun sisällä. Esimerkiksi käytetään vain yhtä kontekstipolkua viittamaan palveluun:
  `POST /appointmentService HTTP/1.1`

- Taso 1: Otetaan käyttöön palvelun resurssit. Sen sijaan että viitataan aina samaan resurssiin pyynnössä, erotellaan ne liiketoiminnan pohjalta. Esimerkiksi haetaan lääkärin tiedot ja vapaiden aikojen tiedot:
  ```
  POST /doctors/mjones HTTP/1.1
  POST /slots/1234 HTTP/1.1
  ```

- Taso 2: Käytetään HTTP verbejä protokollan tarkoittamassa merkityksessä muokkaamaan ja hakemaan tietoja. Esimerkiksi GET verbiä käytetään tietojen hakemiseen ja POST verbiä tiedon luomiseen tai muokkaamiseen:
  ```
  GET /doctors/mjones/slots?date=20200604 HTTP/1.1
  POST /slots/1234 HTTP/1.1
  ```

- Taso 3: Edellisten tasojen lisäksi hyödynnetään hypermedia kontrolleita paluusanomissa niin että hakusanoman vastauksessa on linkki elementti, joka ohjeistaa miten tehdään jatkotoimenpiteitä, tässä tapauksessa varataan aika halutulta lääkäriltä. Alla esimerkki pyyntö- ja vastaus sanomasta:
  Pyyntö:
  ```
  GET /doctors/mjones/slots?date=20200604 HTTP/1.1
  ```
  Vastaus:
  ```
  HTTP/1.1 200 OK
  {
   "openSlotList": {
    "slot": {
      "id": "1234",
  ```

```
      "doctor": "mjones"
    }
    "link": {
      "rel": "/linkrels/slot/book",
      "uri": "/slots/1234"
    }
  }
}
```

Kypsyysmallin pohjalta REST rajapinnat kannattaa rakentaa käytettyjen resurssien ympärille käyttäen tason 3 tai 4 periaatteita, vaikka joissakin tapauksissa HTTP:n tarjoamat metodit eivät suoraan sovellu tarpeeseen. Lisäksi käytettyjen resurssien kontekstipolut on syytä suunnitella huolellisesti, ettei samaa polkua käytetä järjestelmässä useamman mikropalvelun toimesta. (Richardson 2018, 71-76).

3.4.3 Rajapintojen käyttötavat

Kommunikaatioprotokollan ja -tyylin lisäksi, rajapintojen käyttötavat tulee suunnitella. Richardson (2018) listaa seuraavia tapoja kommunikaatioon:

1. dimensio
- yksi-yhteen (one-to-one): jokainen asiakas pyyntö käsitellään vain yhden palvelun toimesta
- yksi-moneen (one-to-many): jokainen pyyntö käsitellään yhden tai useamman palvelun toimesta.

2. dimensio
- synkroninen: asiakas odottaa välitöntä vastausta pyyntöön jääden odottamaan sitä
- asynkroninen: asiakas ei odota välitöntä vastausta vaan jatkaa toimintaansa riippumatta saako vastauksen heti vai ei.

Synkronisissa kutsuissa (pyyntö/vastaus) asiakassovellus lähettää palvelulle pyynnön ja kommunikaatio tapahtuu vain näiden kahden osapuolen välillä. Asiakas odottaa vastausta yleensä määritellyn aikajakson verran joka estää ohjelman muun toiminnan. Jos vastausta ei tule, käsittely päättyy virheeseen. Asynkronisissa kutsuissa sitä vastoin pyynnön käsitteleviä palveluja voi olla yksi tai useampi, ja asiakas ei jää odottamaan vastausta vaan jatkaa muuta toimintaa. Asynkronisia kutsuja voi toteuttaa julkaise/tilaa (publish/subscribe) mallilla jossa pyyntö lähetetään asiakkaalta ensin sanomavälittäjälle josta palvelu käy sen käsittelemässä. Käsittelyn jälkeen palvelu lähettää välittäjälle vastaussanoman josta asiakas käy sen lukemassa. Yleisiä sanomavälittäjä toteutuksia ovat mm. ActiveMQ ja RabbitMQ, tai OpenShiftin mukana tuleva Red Hat AMQ Broker. Toinen vaihtoehto asynkronisiin kutsuihin on pisteestä-pisteeseen (point-to-point) kommunikaatio jossa ei käytetä sanomavälittäjää

vaan asiakkaan pyyntö menee kunkin palveluinstanssin omaan jonoon josta palvelu käy sen käsittelemässä. Tähän vaihtoehtoon on olemassa myös valmiita ratkaisuja kuten ZeroMQ (Richardson 2018, 90-92).

Synkronisen ja asynkronisen sanomanvälityksen valinta on yksi tärkeistä valinnoista mikropalvelu-arkkitehtuurissa. Newmanin (2015) mukaan synkroniset kutsut soveltuvat paremmin aikakriittisiin sovelluksiin joissa vastaus pitää saada heti. Ne ovat myös yksinkertaisempi toteuttaa koska kommunikaatio tapahtuu suoraan asiakkaan ja palvelun välillä. Usein sovelluskehittäjät ovat tottuneet synkroniseen käsittelyyn esimerkiksi ohjelmallisten API rajapintojen käytössä. Toisaalta asynkroninen kutsu toteuttaa paremmin löyhän sidonnan periaatetta joka voi helpottaa palvelun uuden version julkaisua muista riippumattomasti, mutta toteutus on usein monimutkaisempi. Asynkroniset kutsut vaativat sanomavälittäjän ja vastausten käsittely saattaa hankaloitua, jos pyyntö kestää pitkään. Riskinä on myös, että sanomavälitys kerrokseen alkaa muodostua liikaa logiikkaa joka lisää monimutkaisuutta. Välityskerros tulisi siis pitää mahdollisimman yksinkertaisena. Sanomavälittäjän valinta on harkittava tarkkaan, jotta siitä ei muodostu järjestelmän heikkoa lenkkiä, asynkronisen kommunikaation edellytys, kun on että välittäjä toimii aina oikein. Asynkroninen kommunikaatio soveltuu parhaiten pitkäkestoisiin pyyntöihin tai yksi-moneen tyyppisiin tapahtumiin. Newmanin mukaan paras ratkaisu on harkita tapauskohtaisesti, kumpi tapa soveltuu tarpeeseen edellä mainitut näkökohdat huomioiden. Tosin hän varoittaa aiempiin kokemuksiinsa vedoten, että asynkronisten kutsujen käyttöä kannattaa harkita tarkkaan koska ne johtavat helposti monimutkaisuuteen ja yllättäviin ongelmiin käytännön toteutuksissa.

Richardson (2018) suosii asynkronista kommunikaatiota ensisijaisena mikropalvelu arkkitehtuurissa. Hän listaa asynkronisen kommunikaation edut:

- Löyhä liitos ei vaadi palvelun löydettävyyden toteutusta samalla tasolla kuin synkronisessa.
- Korkeat ja hetkelliset piikki kuormat saadaan purettua paremmin asynkronisen vaihtoehdon sanomajonojen kautta. Jos palvelu ruuhkautuu hetkellisesti, pyynnöt menevät jonoon josta ne voidaan käsitellä palvelun kapasiteetin mukaisesti.
- Erilaiset tavat kommunikoida ovat mahdollisia. Voidaan toteuttaa sanomien virtana tai jonototeutuksena.
- Ohjaa sovelluskehittäjiä suunnittelemaan keinot toipua virhetilanteista.

Haasteeksi Richardson listaa erityisesti sanomavälittäjä pohjaisissa ratkaisuissa, että välittäjä on altis virheille, jos sen toiminta estyy, kaikki sanomat päätyvät virheeseen. Myös sanomavälittäjän suorituskyky on kriittinen, sen on kyettävä palvelemaan kaikkia palveluja. Toisaalta pisteestä-pisteeseen tyyppisessä ratkaisussa se ei ole yksittäinen virhelähde, jolloin yhden käsittelijän vikaantuminen ei johda koko järjestelmän sanomaliikenteen pysähtymiseen. Myös lisääntynyt monimutkaisuus on haaste, kun välittäjästä muodostuu yksi arkkitehtuurin kerros lisää.

3.4.4 Transaktion hallinta

Transaktion hallinta tarkoittaa mekanismeja joilla useamman palvelukutsun käsittävän operaation virhetilanteet saadaan hallittua. Transaktioiden avulla voidaan operaatioon osallistuneiden palvelujen toiminta kompensoida niin että tilanne palautuu kaikissa operaatioon osallistuneissa palveluissa samaksi kuin se oli ennen operaation suoritusta. Transaktioiden hallinta on erityisen tärkeää mikropalveluissa, joissa periaate on, että jokainen palvelu omistaa oman datansa. Kun palvelu päivittää dataansa muista riippumattomasti, on haasteena pitää palvelujen tietokantojen data yhteneväisenä. Monoliittisissa sovelluksissa yhteneväisyyden ja ristiriidattomuuden ylläpitäminen on helpompaa, koska yhteisen tietokannan taulujen vierasavaimien avulla tietokantaratkaisu pitää huolen, ettei orpoja rivejä pääse syntymään. Mikropalvelun mukaisissa sovelluksissa taas tietokantojen yhteneväisyys pitää varmistaa muulla tavalla sovellus- tai alustatasolla. Tämän vuoksi mikropalvelu-arkkitehtuurissa datan hallintaan pitää kiinnittää erityistä huomiota. (Newman 2015, 42-58).

Hajautettu transaktion hallinta on tuettuna monessa ohjelmointikehyksessä, esimerkiksi Spring Framework käyttää `@Transactional` annotaatiota huolehtimaan koko operaation onnistumisesta kaikissa osallistuvissa palveluissa. Hajautettu transaktion hallinta toimii hyvin monoliittisissa sovelluksissa ja ylipäänsä synkronisten kutsujen kanssa. Ne vaativat toimiakseen, että kaikki kutsuketjussa käytetyt palvelut ovat käytettävissä operaation aikana. Jos yksinkin ketjuun osallistuva palvelu epäonnistuu käsittelyssä, koko operaatio epäonnistuu. Vaarana on tällöin palvelun saatavuuden aleneminen, kun kokonaissaatavuus on kaikkien operaatioon osallistuneiden palvelujen saatavuuden tulo. Välitöntä vastausta vaativat operaatiot ovat lisäksi hankalia löyhän sidoksen periaatteen kannalta, vaikka ne käytännön toteutuksessa ovatkin yksinkertaisempia toteuttaa. Hajautetut transaktiot toimivat myös huonosti asynkronisissa kutsuissa modernien sanomavälittäjien kanssa, koska ne vaativat niiltä kyseisen transaktiomallin tukea. Etuna hajautetuissa transaktioissa on, että niiden kompensointi on varmalla pohjalla ja ne varmistavat hyvin datan yhteneväisyyden. (Richardson 2018, 111-114).

Hajautettujen transaktioiden sijaan Richardson (2018, 110-132) esittelee Saga suunnittelumallin asynkronisen kommunikaation transaktioiden hallintaan. Se on mekanismi, jolla voidaan pitää eri mikropalvelujen data yhteneväisenä ilman että käytetään hajautettuja transaktioita. Saga huolehtii, että kukin palvelu päivittää tilaansa paikallisella, muista riippumattomalla transaktiolla, jonka suorittamisen jälkeen liipaistaan kutsuketjun seuraava transaktio toisessa palvelussa. Koska paikallinen transaktio päivittää aina tilansa, tarvitaan palveluun myös kompensointi transaktio, jolla tilanne peräytetään, jos jonkin pyyntö kutsuketjussa epäonnistuu. Sagan suoritusta koordinoidaan niin, että jokaisen kutsuketjun transaktion suorituksen jälkeen tiedetään, mikä palvelupyyntö liipaistaan seuraavaksi. Koordinaation pitää myös tietää mitä kaikkia palvelupyyntöjä kutsuketjuun osallistuu ja milloin koko operaatio on valmis. Richardson (2018) esittelee kaksi vaihtoehtoista koordinaatio tapaa:

- Koreografia: Hajautetaan päätöksenteko ja pyyntöjen liipaisu kutsuketjuun osallistujien kesken. Palvelut julkaisevat tapahtumia joiden perusteella tiedetään mikä on pyynnön käsittelyn tila.
- Orkestraatio: Keskitetään koordinaatio-logiikka orkestraatio tasolle. Sagan orkestroija lähettää pyyntöjä kutsuketjuun osallistuville palveluille ja koordinoi koko operaation suoritusta.

Koreografia tyyppisen koordinaation hyvänä puolena on yksittäisen palvelun kannalta yksinkertaisuus, palvelut julkaisevat tapahtumia, kun niiden tila muuttuu. Se myös toteuttaa hyvin löyhän sidoksen periaatetta koska palvelut eivät suoraan kommunikoi keskenään. Haasteena tässä tavassa on kokonaisuuden monimutkaisuus, koodissa ei ole yksittäistä paikkaa jossa sovelluslogiikan näkee vaan se on hajautettu eri palveluihin. Myös riski syklisiin riippuvuuksiin kasvaa koska palvelut liipaisevat vuorotellen toistensa transaktioita. (Richardson 2018, 118-121).

Toisena vaihtoehtona koordinaatioon on orkestraatio. Siinä orkestraatio taso koordinoi koko operaation suoritusta alusta loppuun. Se lähettää palvelupyyntöjä operaatioon osallistuville palveluille ja käsittelee niiden vastaukset, sekä päättää mikä on koko operaation tulos. Se koordinoi virhetilanteet ja tarvittaessa liipaisee kompensointi transaktiot palveluissa. Orkestraatio tyyppisen koordinaation etuna on, ettei syklisiä riippuvuuksia pääse muodostumaan. Pyynnöt lähetetään aina orkestraattorin kautta jolloin palvelut eivät kutsu orkestraattoria. Se myös erottelee selkeämmin operaation toteutuksen vastuut, yksittäisen palvelun ei tarvitse tietää mihin operaatioon se osallistuu vaan hoitaa vain yksittäisen pyynnön käsittelyn. Haasteena orkestraation käytössä on riski orkestraatiotason paisumisesta, jos yhä enemmän sovelluslogiikkaa toteutetaan orkestraatioon. Tämä voi johtaa vaikeasti ymmärrettäviin monoliittisiin toteutuksiin. (Richardson 2018, 122-125).

Newman (2015, 43-46) käsittelee myös orkestraatio ja koreografia koordinaatio toteutukset mainiten koreografia toteutusten olevan löyhemmin liitoksissa toisiinsa ja joustavammin muokattavissa. Haittapuolena on se, että palvelujen rajapintoihin pitää toteuttaa parempi monitorointi, jotta virheenselvitys onnistuu. Hänen kokemustensa pohjalta, koreografia tyyppinen koordinaatio on käyttökelpoisempi erityisesti asynkronisessa kommunikaatiossa. Toisaalta synkroninen kommunikaatio orkestrointi koordinaatiolla on nopeampi toteuttaa aluksi. Tässä hän kehottaa käyttämään omaa harkintaa ja päättämään tapauskohtaisesti kumpaa tapaa käyttää, riippuen käyttötarpeesta ja käytetystä teknologiasta.

3.4.5 Rajapintojen versiointi

Ennemmin tai myöhemmin palvelun elinkaaressa tulee tarve muuttaa sen toimintaa niin että myös sen rajapinta muuttuu. Jos palvelulla on tässä vaiheessa paljon käyttäjiä, muutos voi aiheuttaa ongelmia. Ongelma korostuu, jos palvelun käyttäjä on toisesta tiimistä tai toisesta organisaatiosta, jolloin muutoksen

koordinointi on hankalampaa. Palvelun muutoksissa kannattaa aina pyrkiä siihen, että rajapintaa muutetaan vain viimeisenä vaihtoehtona ja myöhäisimpänä mahdollisena ajankohtana. Lisäksi asiakkaiden löyhä liitos palvelun käytössä auttaa, jos asiakaspää on toteutettu hyvin, se on muutoksia sietävä niin, että rajapinnan muuttuessa sovellus on vielä käytettävissä. Hyvä periaate palvelun toteutuksessa on myös Postelin laki, ”ole konservatiivi omissa toimissasi, mutta liberaali miten suhtaudut muihin”. (Newman 2015, 63).

Kun palvelun muutos on pakko tehdä, se kannattaa tehdä niin, ettei asiakkaan päässä ole välitöntä muutostarvetta. Hyvä tapa tähän on tuottaa yhtä aikaa saataville eri versioita rajapinnoista. Tällöin asiakas voi jatkaa vanhan rajapinnan käyttöä, kunnes tekee integraation uuteen rajapintaan. Palvelun versioinnissa on syytä käyttää semanttista versiointia, MAJOR.MINOR.PATCH, jossa MAJOR muutos tarkoittaa rikkovaa muutosta joka ei ole taaksepäin yhteensopiva. MINOR muutokset ovat taaksepäin yhteensopivia ja PATCH korjaa virheen aikaisemmassa toteutuksessa. REST tyylisissä rajapinnoissa on hyvä tapa vaatia sanoman HTTP otsakkeessa versio jota asiakas käyttää, tai sisällyttää versionumero mukaan palvelun osoitteeseen, esimerkiksi /v1/tilaus ja /v2/tilaus. Jälkimmäinen tapa voi helpottaa reititystä oikeaan palveluversioon ja olla selkeämpää asiakkaan kannalta. Eri versioiden tukemisessa on harkittava tarkkaan, miten kauan vanhoja versioita pidetään saatavilla. Useiden versioiden pitkäaikainen tukeminen hankaloittaa virheenkorjauksia, kun muutos pitää tehdä kaikkiin versioihin. Lisäksi reitityskerros palveluihin monimutkaistuu, kun useampien versioiden sisäinen reititys pitää toteuttaa mahdollisesti koko operaation kutsuketjussa. Hyvä käytäntö on pitää yllä aiempaa versiota vain päivitysoperaation ajan, jonka aikana nähdään, että asiakkaiden sovellukset toimivat oikein uutta versiota vasten. Jos ongelmia ilmenee, voidaan palata käyttämään vanhaa versiota. Jos integroituvat asiakkaat vaativat enemmän aikaa, pitää vanhoista versioista luopumisesta tehdä suunnitelma hyvissä ajoin ja viestiä se asiakkaille. (Newman 2015, 64-65).

3.5 Migraatio monoliitistä mikropalveluihin

3.5.1 Perusteet migraatiolle

Usein pitkään tuotannossa olleen ohjelmistojärjestelmän koko kasvaa vuosi vuodelta, kun muutoksia ja korjauksia tehdään. Ohjelmistoille tyypillistä on, että kun versio julkaistaan, tulee toiveita lisäominaisuuksista ja muutoksista. Pitkän ajan kuluessa muutokset kasautuvat ja heikentävät koodin teknistä laatua. Ongelma pahenee heikosti modulaarisissa toteutuksissa ja erityisesti monoliiteissa, joissa järjestelmä ajetaan kokonaan tai suurelta osin saman prosessin sisällä. Tällöin nopeiden muutoksien tekeminen järjestelmässä on nopeampaa kuin jos käytettäisiin prosessien välillä olevia rajapintoja. Nopeuden haittapuolena voi olla, ettei muutoksia suunnitella ja dokumentoida samalla tarkkuudella

kuin tilanteessa jossa kutsut ylittävät prosessirajan. Kun koodin tekninen laatu heikkenee, eli syntyy teknistä velkaa, alkaa muutosten tekeminen olla työlästä ja julkaisusykli pitenee. Tällöin järjestelmään ei saada nopeasti tarvittavia liiketoimintamuutoksia ja organisaation kilpailukyky alkaa heiketä. Teknisen laadun huonontuminen johtaa myös tuotannosta löytyviin virheisiin jotka vievät kehitystiimin aikaa uusien ominaisuuksien kehittämiseltä. Järjestelmä skaalautuvuus kärsii, kun sitä ajetaan yhdessä suoritettavassa prosessissa ja lisäkapasiteetin toteuttaminen saattaa tulla kalliiksi tai olla jopa mahdotonta tietyn pisteen jälkeen. Monoliitin kasvamisella on myös vaikutus itse kehitystyön kuormitukseen, kun monimutkaisuus lisää kehittäjien kognitiivista kuormaa ja lisää riskiä virheisiin. Käytetyn teknologian vaihtaminen on usein hyvin hankalaa, koska se vaatisi koko järjestelmän uudelleen koodaamisen, toisin kuin mikropalveluissa joissa vaihdosta voi tehdä helpommin pienissä palasissa. (Richardson 2018, 2-6, 429-430).

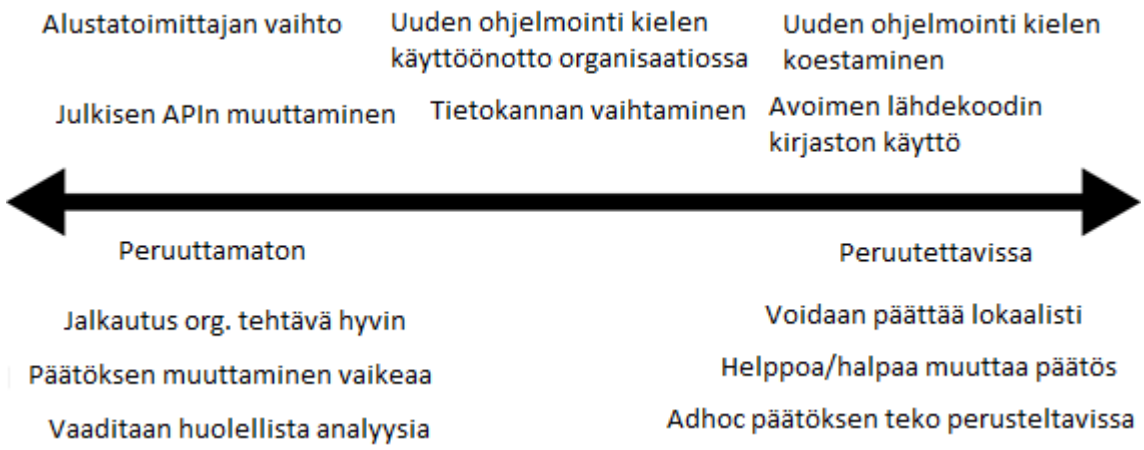
Migraatioon lähtiessä on syytä analysoida, johtuuko sen hetkiset ongelmat arkkitehtuurista vai onko syynä huono ohjelmistokehitysprosessi. Jos esimerkiksi automaattiset testit ovat tekemättä, on muutosten tekeminen hidasta pitkän regressiotesti kierroksen vuoksi. Myös skaalaus ongelmat voivat ratketa muullakin keinolla kuin mikropalveluihin siirtymällä. Lisäksi on syytä analysoida mitä olemassa olevalle järjestelmälle kannattaa tehdä, onko se niin rapistunut, että täydellinen uudelleenkoodaus on tarpeen vai kannattaako vanhaa koodia uudelleen käyttää mikropalveluissa. (Newman 2019, 76-78, Richardson 2018, 429-430).

3.5.2 Migraation suunnittelu

Monoliittisen järjestelmän migraatio mikropalveluiksi on usein työläs prosessi. Vanhaan järjestelmään on vuosien varrella tehty paljon toiminnallisuutta ja sen osiin syntynyt paljon riippuvuuksia. Tärkeä periaate migraatiossa on tehdä se inkrementaalisesti, aloittaen jostain pienestä muutoksesta mikropalveluiden suuntaan ja laajentaen tätä ajan kuluessa. Hyvin harvoin on tilanne, jossa koko järjestelmän uuskehitys voidaan laittaa syrjään ja kehittää vain uutta arkkitehtuuria. Yleensä vanhaan järjestelmään tulee jatkuvasti muutospyyntöjä ja kehitysideoita joita pitää myös vielä eteenpäin. Siksi käytännöllisin ratkaisu on laatia karkeat suuntaviivat migraatiolle ja alkaa tehdä pieni palanen kerrallaan. Inkrementaalisessa migraatiossa voidaan myös koestaa eri tapoja ja testata käytännössä toimintaa, ja palata tarvittaessa takaisin, jos jokin asia ei toiminut. Tällöin voi joutua heittämään hukkaan pienen kehitysaskeleen mutta samalla saada uusi parempi suunta johon migraatiota jatketaan. Samalla vanha järjestelmä jatkaa toiminnassa normaalisti ilman keskeytyksiä. Inkrementaalisessa etenemisessä on myös tärkeää viedä muutokset aina tuotantoon asti, jotta saadaan paras kuva, miten järjestelmä käyttäytyy todellisessa tilanteessa muutosten jälkeen. (Newman 2019, 53-56).

Migraation suunnittelussa on tunnistettava, mitkä päätökset voidaan peruuttaa ja mitkä ovat enemmän tai vähemmän lopullisia. Newman (2019, 55)

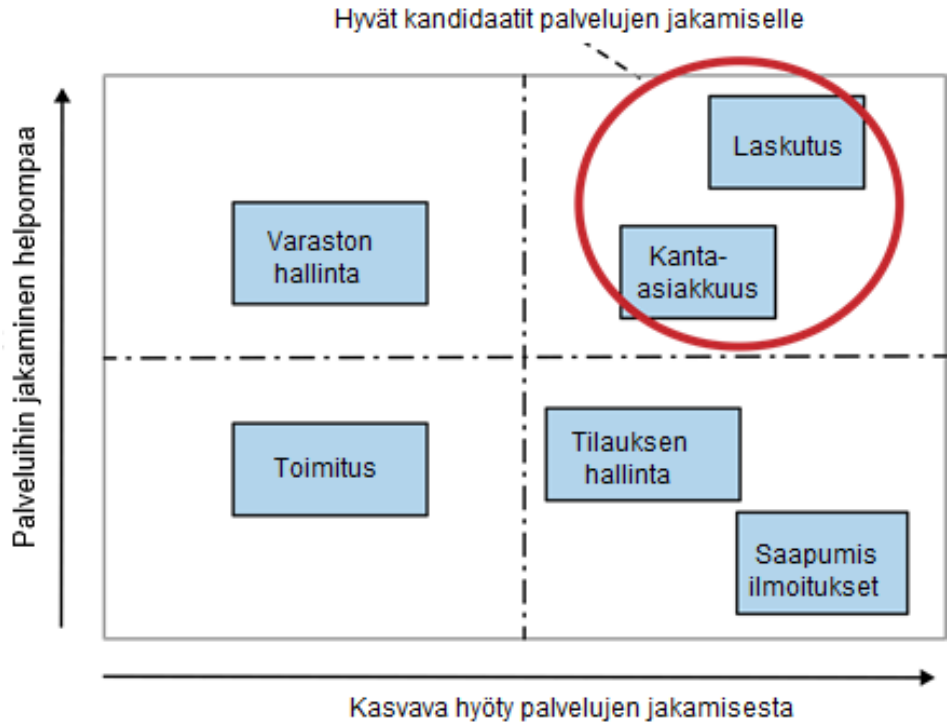
kuvaa näitä päätöksiä lineaarisella janalla joissa toisessa päässä on peruuttamattomat päätökset ja toisessa sellaiset jotka voidaan muuttaa:



Kuvio 3. Peruuttamattomat ja peruutettavissa olevat päätökset (Newman 2019, 55)

Newman painottaa, että migraatio kannattaa aloittaa niistä osista joissa epäonnistuminen aiheuttaisi vähiten haittaa tai ylimääräistä työtä. Esimerkiksi koodin siirtäminen palvelusta toiseen on verrattain helppoa ja se voidaan palauttaa helposti. Sitä vastoin tietokannan jakaminen useampaan osaan aiheuttaa enemmän työtä ja palauttaminen on hankalampaa.

Dehghani (2018) esittää että kannattaa aloittaa yksinkertaisista moduuleista joissa on mahdollisimman vähän riippuvuuksia muualle järjestelmään tai muihin sovelluksiin. Palvelun tietokanta tulisi olla yksinkertainen tai se ei mahdollisesti vaatisi sitä ollenkaan. Palvelu voi olla monoliitin rajapinnassa käytettävä osa joka on helppo irrottaa omaksi mikropalvelukseen ja se voidaan testata ja asentaa ajoympäristöön riippumatta monoliitin julkaisusyklistä. Tällaisella palvelulla olisi hyvä harjoitella mikropalvelujen kehittämistä niin että sen vaatima kehitys- ja käyttöönottoympäristö tulisi tutuiksi kehitystiimille. (Dehghani 2018). Newman (2019) painottaa vähäisen riippuvuuden muusta järjestelmästä lisäksi saavutettua hyötyä mitä migraatiolla saavutetaan. Hän ehdottaa, että kehitystiimi piirtää nelikentän johon järjestelmän toiminnot jaetaan. Nelikentän oikeassa ylänurkassa on toiminnot, jotka on helppo erottaa omiksi palveluiksi ja joista saadaan eniten hyötyä:



Kuvio 4. Palvelujen migraation priorisoinnin nelikenttä. (Newman 2019, 62)

Edellä kuvattujen kriteerien lisäksi, Dehghani (2018) listaa ominaisuuksia joilla voidaan tunnistaa potentiaalisia monoliitista irrotettavia palveluja:

- Poikkileikkaavat moduulit joilla ei ole omaa kohdealueen mallia vaan ne tuottavat toimintoja useaan liiketoiminnan kyvykkyyteen. Näiden moduulien irrottaminen omiksi palveluiksi kannattaa tehdä ensimmäisten joukossa koska muuten ne aiheuttavat migraation edetessä ongelmia.
- Toiminnallisuudet jotka voidaan kehittää ja julkaista itsenäisesti. Palvelujen irrottaminen tulee tehdä vertikaalisesti liiketoiminnan kyvykkyyksien mukaisesti, jotta ne ovat riippumattomia kokonaisuuksia.
- Liiketoiminnan kannalta tärkeät osat joihin kohdistuu paljon muutoksia. Monoliitissa näiden ominaisuuksien kehittäminen olisi työlästä, joten irrottaminen mikropalveluun tuo heti hyötyjä.

Palvelujen jakamisen lisäksi Dehghani listaa periaatteita joita migraatiossa kannattaa noudattaa. Olemassa olevaan koodiin ei pitäisi jäädä liikaa kiinni vaan tarkastella kriittisesti kannattaako sitä siirtää mikropalveluun vai kirjoittaa koodi uudelleen. Usein voi olla antoisampaa arvioida yhteistyössä liiketoiminnan kanssa onko vanhat tavat tehdä asioita parhaita mahdollisia. Samalla voidaan tarkastella järjestelmän koodin laatua, käytettävää teknologiaa ja mahdollisesti käyttää modernimpia kehitysvälineitä ja ohjelmistopinoa. Vaarana voi myös olla, että palvelujen jakamisessa mikropalveluihin mennään liiallisuuksiin ja tehdään liian pieniä palveluja, jotka on jaettu normalisoidun datan pohjalta, ei liiketoiminnan kyvykkyyksien. Tämä voi johtaa aneemisiin palveluihin jotka

tekevät vain CRUD operaatioita järjestelmän resursseihin. Tästä voi aiheutua monia ongelmia käytännön kehitystyössä; kokoonpanon hallinta hankaloituu, palvelujen asentaminen on monimutkaista, järjestelmää on vaikea debugata, sekä transaktion hallinta palvelujen välillä on työlästä toteuttaa. Dehghani ehdottaa, että palvelut kannattaa alkuvaiheessa jakaa isommiksi kokonaisuuksiksi ja myöhemmin kokemuksen kertyessä pilkkoa tarpeen mukaan pienemmiksi. Tärkeää on myös muistaa luopua kokonaan vanhoista palveluista, kun uusi valmistuu, ettei jouduta ylläpitämään useampaa ratkaisua. (Dehghani 2018).

3.5.3 Migraation suunnittelumallit

Monoliittisen järjestelmän migraatiossa mikropalveluiksi tärkeää on inkrementaalisuus. Muutoksia tulee tehdä yksi osa kerrallaan ja testata huolellisesti, myös tuotannossa, jotta saadaan lopullinen varmuus toimivuudesta. Tämä madaltaa kynnystä lähteä tekemään migraatiota, kun yksittäiset muutokset ovat pieniä, eivätkä vie kehitystiimin kaikkea aikaa. Tällöin voidaan rinnalla toteuttaa myös liiketoiminnan vaatimia muutoksia olemassa olevaan järjestelmään. Kantava periaate, jolla migraation ongelmia voidaan vähentää, on Martin Fowlerin tunnetuksi tekemä toteamus: ”*Ainoa asia jonka Big Bang uudelleenkoodaus takaa, on Big Bang*”. (Richardsson 2018, 430-432).

Inkrementaalisuutta tukeva suunnittelumalli on Fowlerin (2004) kuvaama kuihdutus malli (strangler fig application). Siinä perinnejärjestelmä kuihdutetaan jokaisessa muutoksessa pienemmäksi osaksi kokonaisuutta, tekemällä uudet ominaisuudet vanhan järjestelmän ulkopuolelle ja poistamalla osia perinnejärjestelmästä käytöstä. Nimi tulee viikunapuun tavasta levitä toisen puun runkoa pitkin suuremmaksi, kunnes sen oma runko pystyy kannattelemaan puun. Vanhan järjestelmän kuihdutus voi kestää vuosia, tärkeää on, että jo alusta alkaen saadaan hyötyjä migraatiosta.

Richardsson (2018, 430-442) kuvaa eri strategioita joilla vanha järjestelmä voidaan kuihduttaa:

- toteutetaan uudet ominaisuudet uusina mikropalveluina
- irrotetaan käyttöliittymä ja taustajärjestelmä toisistaan
- hajotetaan monoliitti jakamalla se mikropalveluiksi.

Uudet ominaisuudet uusina mikropalveluina
Tässä strategiassa tärkeintä on lopettaa isoksi kasvaneen monoliitin kasvattaminen entisestään. Uudet ominaisuudet toteutetaan uusiin mikropalveluihin jotka voivat hyödyntää vanhaa monoliittia sellaisenaan tai pienin muutoksin. Toteutuksen tekeminen nykyaikaisilla välineillä ja teknologia-pinolla tuovat tehokkuutta kehitykseen ja osoittavat mikropalveluihin siirtymisen hyödyn. Uuden palvelun integroimiseksi monoliittiin tarvitaan kuitenkin integraatiototeutus ja API yhdyskäytävä. Integraatiototeutuksen tehtävänä on toimia sovittimena monoliitin ja mikropalvelun välillä niin että mikropalvelu voi hyödyntää vanhassa järjestelmässä olevia toimintoja. Se rooli voi olla esimerkiksi to-

teuttaa REST tyyppinen rajapinta jolla mikropalvelu kommunikoi muiden palvelujen kanssa. API yhdyskäytävää taas tarvitaan ohjaamaan vanhaan järjestelmään tulevaa liikennettä niin että uuden mikropalvelun osalta kutsut ohjataan monoliitin sijaan sille. Uuden ominaisuuden toteuttaminen erillisenä palveluna on aina suunnittelun paikka. Joskus muutos on niin pieni, ettei uutta palvelua kannata tehdä vaan toteuttaa se monoliittiin jonka purkaminen osiin tehdään myöhemmin. (Richardsson 2018, 434-435).

Käyttöliittymän ja taustajärjestelmän irrottaminen

Kerrosarkkitehtuurissa käyttöliittymä, liiketoimintalogiikka ja tiedonhallinta ovat erotettu omiin moduuleihin, jotka kuitenkin koostetaan yhteen ajettavaan komponenttiin. Erottamalla käyttöliittymäkerroksen omaan komponenttiin ja luomalla API rajapinnan liiketoimintalogiikan toteuttavaan komponenttiin, saadaan nämä osat itsenäisesti asennettaviksi ja skaalattaviksi. Kun liiketoimintalogiikka ajetaan omassa prosessissaan ja se tuottaa prosessin ulkopuolelta kutsuttavan rajapinnan, voidaan tätä rajapintaa kutsua myöhemmin muista mikropalveluista. Hyötynä saadaan myös se, että käyttöliittymää voi kehittää riippumattomammin ja mahdollisesti tehdä toisen tiimin toimesta. Haasteeksi pelkän käyttöliittymäkerroksen erottamisessa tulee uusien ominaisuuksien kehittäminen, kun muutoksia pitää synkronoida tiiviisti käyttöliittymä- ja liiketoimintakerroksen välillä. (Richardsson 2018, 436-437).

Monoliitin hajottaminen mikropalveluiksi

Jos migraation tavoitteena on saavuttaa kaikki mikropalveluista saatavat hyödyt, tulee monoliitti loppujen lopuksi hajottaa useiksi mikropalveluiksi. Tällöin liiketoiminnan kyvykkyys irrotetaan vertikaalisesti monoliittisesta arkkitehtuurista niin että sillä on oma API rajapinta, kohdealueen liiketoimintalogiikka, tietokannan käsittely sekä tietokannan skeema. Ominaisuuksien irrottamisessa tulee huomioida saavutettavat hyödyt versus työmäärä. Irrottaminen kannattaa aloittaa niistä osista joihin kohdistuu paljon muutoksia ja se tulee tehdä inkrementaalisesti pienin askelin. Irrottaminen on usein vaikeaa ja vaatii huolellista kohdealueen mallin suunnittelua sekä tietokannan uudelleen järjestelyä. Mikropalvelun kohdealueen malli pitää irrottaa monoliitin mallista, jolloin voi tulla tilanteita joissa luokkien väliset viittaukset eivät onnistu enää metodikutsuilla. Nämä voidaan korvata DDD-mallin yhdistelmällä (aggregate) jossa toisen luokan viittaus korvataan perusavaimen tunnisteella. Tietokannan uudelleen järjestely vaatii usein tietokannan taulujen jakamista niin että uuden mikropalvelun omistama osuus viedään omaan tietokannan skeemaan ja tauluun. Tällöin monoliittiin jäävä osuus vaatisi myös merkittäviä muutoksia. Muutostarpeita ja riskiä voi vähentää Amber & Sadalage (2011) esittelemällä taulun jakaminen (split table) refaktoroinnilla. Siinä uuden skeeman tiedot replikoidaan vanhaan skeemaan tietyn ajanjakson ajan. Kun uusi mikropalvelu huolehtii sen kohdealueen tiedon käsittelystä, monoliitin käyttämästä skeemasta pitäisi irrottaa nämä osuudet. Tällöin on iso riski, että monoliitti vielä käsittelee tietoa joka ei ole enää saatavilla. Säilyttämällä monoliitin vanha skeema sellaisenaan, pienennetään huomattavasti riskiä epätoivotuilla yllätyksille. Vanhan skeeman

kentät pitää muuttaa lukutilaan ja huolehtia ettei monoliitin kautta enää tehdä päivitys operaatioita, mutta luku on edelleen mahdollista replikoinnin ansioista. (Richardsson 2018, 437-442).

Newman (2019, 79-83) kuvaa kuihdutusmallin saman tyyppisesti kuin Richardsson, mutta ei listaa anti-korruptiokerrosta erillisenä mallina. Sen sijaan hän esittelee useita muita migraation suunnittelumalleja:

- *Käyttöliittymän koostaminen (UI composition)* jossa eri web-sivut tai näkymät muodostavat oman mikropalvelunsa. Tällä mallilla voidaan erottaa koko liiketoiminnan kyvykkyys riippumattomaksi osaksi.
- *Haaroitus abstraktion avulla (branch by abstration)* mallilla tehdään uusi rajapinta olemassa olevaan toiminnallisuuteen ja vaihdetaan asiakassovellukset käyttämään tätä. Rajapinta käyttää edelleen vanhaa toiminnallisuutta joka rinnalla korvataan uudella mikropalvelulla. Kun uusi toteutus on valmis, voidaan abstraktio vaihtaa käyttämään tätä uutta mikropalvelua.
- *Rinnakkainen ajaminen (parallel run)* tarkoittaa sekä vanhan että uuden version ajamista yhtä aikaa tuotannossa. Uusi versio asennetaan tuotantoon vanhan rinnalle ja verrataan, miten se toimii suhteessa vanhaan. Tällä mallilla voidaan uusi versio ottaa käyttöön pienemmällä riskillä, kun vanhaan voidaan palata nopeasti.
- *Rajapinnan kuorrutus (decorating collaborator)* mallissa monoliitin eteen sijoitetaan älykäs välityspalvelin, joka ohjaa monoliitille tulevat kutsut ilman muutoksia. Se kuitenkin välittää tulevissa ja lähtevissä sanomissa olevaa dataa myös uudelle mikropalvelulle joka toteuttaa uutta toiminnallisuutta. Monoliitti ei tiedä muutoksesta mitään vaan toimii niin kuin ennenkin.
- *Muuttuneen tiedon käyttö (change data capture)* mallissa uusi mikropalvelu reagoi tietokannan muutokseen ja suorittaa sen pohjalta toimenpiteitä. Malli vaatii tavan, jolla datan muutokset havaitaan, esimerkiksi tietokanta-triggerin joka liipaisee mikropalvelu kutsun tiedon muuttuessa. Malli soveltuu lähinnä tilanteisiin, joissa muuttunutta tietoa vain luetaan.

Laajempi lähestyminen migraatioiden suunnittelumalleihin esitellään Balalaie, Heydarnoori, Jamshidi, Tamburri & Lynn 2018 tutkimuksessa jossa listataan mikropalvelujen arkkitehtuuri migraation lisäksi malleja myös uuteen kehitysympäristöön ja pilvipalveluun siirtymiseen. He kuvaavat 15 mallia jotka tarvitaan kokonaisuudessaan pilveen siirtymisessä. Muun muassa kuormanjakoon ja palvelujen löydettävyyteen liittyviä malleja, jotka koskettavat pilvialustan ominaisuuksia, eivätkä liity suoraan mikropalvelu-arkkitehtuuriin. Tässä käsitellään mallit jotka kuvaavat monoliittisen arkkitehtuurin muuntamista mikropalveluiksi.

Kuvaa arkkitehtuurin nykytilanne (recover the current architecture)
Mallin tavoitteena on lisätä ymmärrystä nykyisestä arkkitehtuurista, jotta sen purkaminen mikropalveluiksi olisi mahdollista. Erityisesti nykyinen kompo-

nentti ja palvelurakenne tulee olla kuvattuna ja ymmärrettynä mitä järjestelmän osat tekevät. Myös teknologia arkkitehtuuria on syytä tarkastella huolella, jotta ymmärretään jatketaanko samalla teknologia pinolla vai muutetaanko sitä mikropalveluihin siirryttäessä. Nykyiset jatkuvan toimittamisen käytännöt tulee ymmärtää ja tarkastella niitä mikropalveluihin siirtymisen kannalta. (Balalaie ym. 2018).

Monoliitin purkaminen (decompose the monolith)
Tutkimuksessa mallilla viitataan DDD-mallin soveltamiseen, kun palvelurakennetta suunnitellaan. Siinä kuvataan peruslähtökohta jossa mallia kannattaa soveltaa:

- nykyisen järjestelmä lähdekoodin yhtenäisyys on heikolla tasolla
- monoliitin eri osilla on erilaiset ei-toiminnalliset vaatimukset
- jokainen muutos aiheuttaa koko järjestelmän uudelleen provisioinnin ja sen osien muutostarpeet ovat erilaisia.

Malli ehdottaa, että koko vanha järjestelmä mallinnetaan DDD-mallin mukaisesti rajattuihin konteksteihin niin kuin kyseessä olisi uuskehitys projekti jolloin saadaan kuvattua tavoitetila, johon pyritään. Mallin toteuttaminen on sitten toinen asia, se kannattaa tehdä inkremetaalisesti niin että jokaisen muutoksen jälkeen ollaan lähempänä tavoitetilaa. (Balalaie ym. 2018).

Monoliitin purkaminen datan pohjalta (decompose the monolith based on data)
Mallissa monoliitin purkamisen lähtökohdaksi otetaan datan omistajuus. Monoliitti puretaan sen tietokannassa olevan datan pohjalta niin että ne ryhmitellään loogisiin kokonaisuuksiin ja rakennetaan mikropalvelut niiden käyttöön. Tiedon replikointi eri palvelujen tietokantojen välillä on mahdollista mutta kukin palvelu saa muokata vain omistamaansa tietoa. Tilanteet, joissa mallia sovelletaan, ovat samat kuin edellä kuvatussa mallissa. (Balalaie ym. 2018).

Riippuvuuden muuttaminen palvelukutsuksi (change code dependency to service call)
Kooditason riippuvuus voi usein olla ongelmallista tilanteissa joissa moni palvelu käyttää samaa kirjastoriippuvuutta. Jos kirjastoon tehdään muutos toisen palvelun tarpeisiin, se ei olekaan enää yhteensopiva toisen palvelutoteutuksen kanssa. Myös toiminnon skaalautuvuus tarve voi olla erilainen kuin sitä käyttävällä palvelulla. Tällöin suora kooditason riippuvuus kannattaa toteuttaa erillisenä palveluna johon on löyhempi liitos ja joka skaalautuu itsenäisesti asiakaspalvelujen tarpeiden mukaisesti. Vaarana kirjaston eriyttämisessä palveluksi voi olla palvelujen kutsuketjujen kasvu ja niistä johtuvat suorituskyky ongelmat. (Balalaie ym. 2018).

4 OPENSHIFT PILVIPALVELUALUSTANA

Pääluvussa esitellään OpenShift alustan keskeiset elementit, miten ne toimivat yhteen ja miten niistä koostuvat alustan palvelut. OpenShift alusta on asennettu yksityisenä pilviratkaisuna tutkimuksen tilaajan konesaliin.

4.1 OpenShift perusteet

OpenShift on pilvipalvelualusta, joka perustuu Kubernetes ja Docker teknologioihin joiden päälle on rakennettu kontteihin perustuva PaaS alusta helpottamaan sovellusten kehitys- ja käyttöönottoprosesseja. OpenShift on saatavilla eri vaihtoehtoina joko julkisessa pilvessä tai omaan konesaliin asennettavana yksityisenä pilvenä. Sillä voidaan ottaa käyttöön sovelluksia useammalle Linux-koneelle ja julkaista ne käyttäjien saataville tietoverkon välityksellä. Alustassa voidaan suorittaa sovelluksia, jotka on tehty millä tahansa ohjelmointikielellä tai tekniikalla. (Dumpleton 2018, 1).

OpenShift esiteltiin vuonna 2011 kun Red Hat laittoi kehittämänsä pilvipalvelu-alustan ohjelmakoodin saataville open source projektina. Projektia on kehitetty vuosien varrella ja nykymuodossaan se pohjautuu Kubernetes orkestrointi-alustaan ja Docker kontteihin. Myös muita kontitusalustoja on tuettujen listalla. Näiden lisäksi alusta pitää sisällään hallintavälineitä konttien luomiseen, virtuaaliverkkojen hallintaan ja monitorointiin. Alusta on asennettavissa CentOS pohjaisiin virtuaalikoneisiin tai suoraan fyysiselle raudalle. Sen avulla voidaan tuottaa pilvipalveluja julkiseen tai yksityiseen käyttöön. RedHat tarjoaa alustaa kolmen eri tuotteen kautta; OpenShift Origin on tarkoitettu yksityiseen pilveen, jolloin pilven hallinta on asiakasorganisaation vastuulla, OpenShift Online, julkinen pilvi josta kuka tahansa voi ostaa kapasiteettia ja viedä sovelluksiaan ajoon, sekä OpenShift Dedicated jossa Red Hat tarjoaa muista asiakkaista erillisen klusterin halutuilla resursseilla. Kahdessa viimeksi mainitussa Red Hat on vastuussa palvelun toiminnasta. (Dumpleton 2018, 3-7).

Alusta voidaan kategorisoida PaaS malliseksi koska se tuottaa tarvittavat välineet sovellusten koostamiseksi ja ajamiseksi OpenShiftin solmuissa (node). Sovelluksia voidaan koota suoraan ohjelmakoodista niin että alusta hoitaa kääntämisen, lopputuotteen (esim. Java jar) viemisen Docker levykuvaksi (image), levykuvan ajamisen Docker kontissa sekä replikoimisen halutuille isäntäkoneille. Alusta pitää sisällään käännösvälineet eri ohjelmointikielille ja mekanismit joilla voi hakea ohjelmakoodin automaattisesti esimerkiksi Git-pohjaisesta ohjelmakoodi säilöstä. (Dumpleton 2018, 5).

4.2 Konttien ajoalusta

OpenShiftin tärkeimmät komponentit ovat konttien ajon mahdollistavat Docker-pohjainen ajoympäristö ja konttien orkestrointi-ratkaisu Kubernetes. Docker tuottaa Linux palvelimelle ajonaikaisen ympäristön jossa yksittäisiä kontteja voidaan ajaa. Kubernetes orkestroi ja hallinnoi konttien ajoa klusteroidussa, useamman palvelimen ympäristössä. Konttien ajoaikaisessa ympäristössä voidaan Dockerin lisäksi käyttää myös muita Open Container Initiative (OCI) -standardia noudattavia konttien ajoympäristöjä. (Lossent, Rodriguez & Wagner 2017).

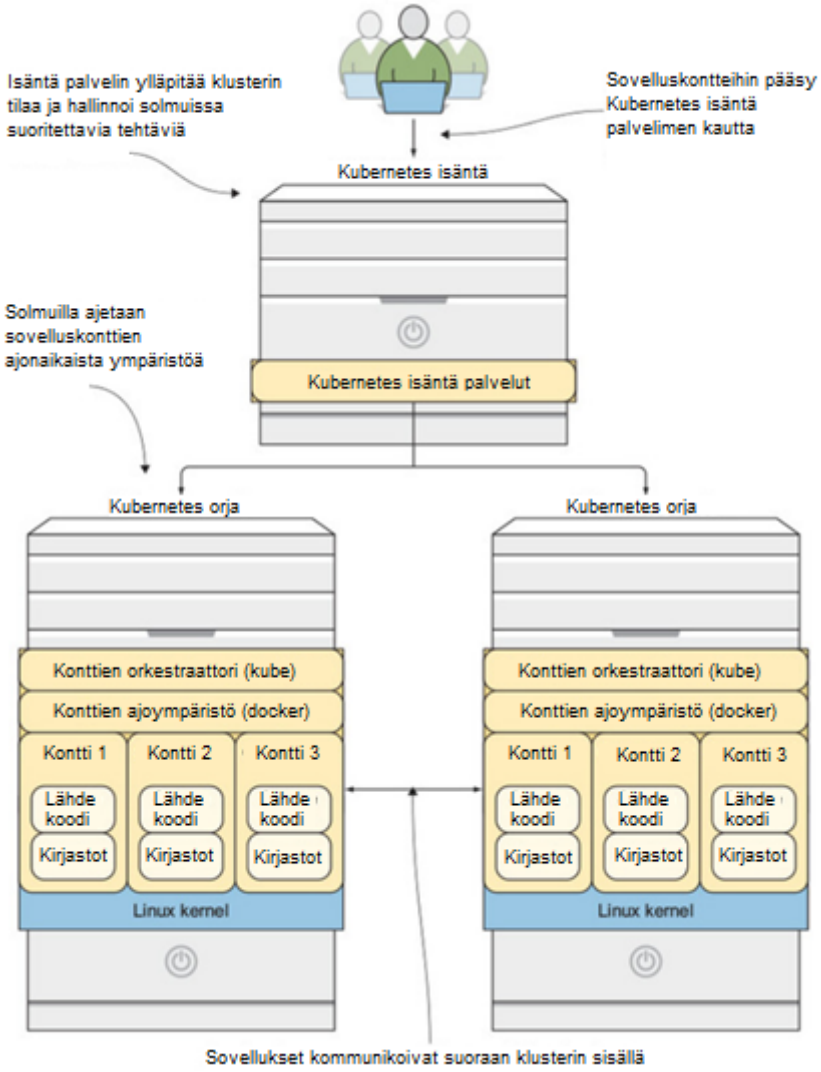
Konttien ajoympäristön tehtävä on luoda ja hallita kontteja yksittäisellä palvelimella. Se eristää kontissa ajettavat prosessit muista koneella ajettavista prosesseista niin, etteivät ne näe toistensa resursseja. Kontin näkökulmasta kaikki sen toimintaan vaikuttavat komponentit löytyvät sen sisältä niin, ettei riippuvuuksia muualle ole. Kontti tarvitsee vain ajokoneen ytimen toimiakseen, jolloin se on helposti luotavissa uudelleen ja siirrettävissä toiselle ajokoneelle. Kontilla on oma levyjärjestelmä, työmuisti, koneen nimi, verkkoresurssit sekä prosessien hallinta. Muutokset isäntäkoneella tai muissa prosesseissa eivät vaikuta kontin toimintaan. Konttien eristäminen toisistaan tehdään Linux ytimen ominaisuuksilla kuten kontrolliryhmät (cgroups), nimiavaruudet (namespaces) ja SELinux. Kontrolliryhmien avulla voidaan Linuxissa rajoittaa prosessorin ja työmuistin käyttöä kontissa. Nimiavaruudet eristävät konteissa käytettävät resurssit kuten virtuaaliverkon ja levyjärjestelmän. SELinux (Security-Enhanced) tuo lisäturvallisuutta konttien ajoon estämällä yritykset päästä kontin ulkopuolella oleviin resursseihin. Periaatteessa kontin voi tehdä Linuxissa itsekin käyttäen näitä ytimen ominaisuuksia, mutta Docker ja vastaavat konttien ajoympäristöt ovat tuoneet automatiikan ja helppokäyttöisyyden niiden luontiin. (Duncan & Osborne 2018, 38-57).

Koska konttien ajoympäristön tehtävä on rajoittunut yksittäiselle isäntäkoneelle, tarvitaan ratkaisu joka voi hallita konttien ajoa useammalla palvelimella. Tähän tarkoitukseen OpenShift alustassa on Kubernetes, konttien orkestrointi-ratkaisu, joka hallinnoi konttien ajoa ja jakaa niille kuormaa koko palvelinklusterin tasolla. Se huolehtii, että haluttu määrä sovelluksen kontteja ajetaan klusterin palvelimilla ja monitoroi niiden toimintaa niin että sovelluskontin kaatuessa se pystyttää tilalle uuden. Kubernetes voi myös automaattisesti skaa-

lata lisää kapasiteettia sovellukselle, jos annetut raja-arvot ylittyvät prosessin kuorman tai muistinkulutuksen suhteen. Kun sovellus pystytetään Kubernetes klusteriin, se luo sille replikointi-ohjaimen (replication controller), palvelun (service) sekä podin. Replikointi-ohjaimen tehtävänä on pitää huolta, että sovelluksen kontteja on ajossa koko ajan haluttu määrä. Palvelu julkaisee sovelluksen luomalla sille IP-osoitteen, jonka kautta siihen pääsee käsiksi klusterin ulkopuolelta. Pod on pienin skaalattava yksikkö klusterissa ja se pitää sisällään sovelluksen joka tyypillisesti sijaitsee yhdessä kontissa. Komplekseissa tarpeissa podissa voi olla myös useampi kontti jotka skaalautuvat yhdessä. (Lukša 2018, 16-21).

Kubernetes toimii isäntä-orja (master-slave) mallilla niin että isäntä huolehtii konttien ajamisesta ja hallinnasta orja-palvelimilla. Isäntä palvelin huolehtii klusterin tilasta ja tietää millä palvelimilla ajetaan mitäkin kontteja. Se myös monitoroi niiden kuntoa ja tekee toimenpiteitä klusterin saattamiseksi haluttuun tilaan. Liikenne orja-palvelimilla ajettaviin kontteihin menee isäntäkoneen kautta. Kaikki työkuorma ajetaan orja-palvelimilla, joissa konttien ajoympäristö huolehtii yksittäisen palvelimen sisällä konttien ajosta. Orjapalvelimella on myös sisäinen levykuvien rekisteri, jonka avulla kontti luodaan oikeasta levykuvasta ajoon. (Lukša 2018, 16-21).

Seuraavassa kuviossa on esitetty Kubernetesin isäntä-orja malli sekä sen suhde palvelimen konttien ajoympäristöön, kontteihin sekä palvelimen käyttöjärjestelmään.



Kuvio 5. Kubernetes isäntä-orja (master-slave) malli (Duncan & Osborne 2018)

4.3 OpenShiftin resurssit

OpenShift klusteri pohjautuu sisäisesti projekteihin, joilla erotetaan nimiavaruuksien (namespace) avulla eri järjestelmien ja asiakkaiden resurssit toisistaan. Nimiavaruuden avulla voidaan kerätä loogisesti toisiinsa liittyvät sovellukset yhden projektin alle ja hallita helpommin niiden riippuvuuksia. Projektiin luodaan sovelluksia, joiden koontiin, hallintaan ja julkaisuun käytetään lukuisia OpenShiftin resursseja ja komponentteja, jotka mahdollistavat jatkuvan integraation ja asentamisen käytännöt. Tässä luvussa käydään läpi OpenShiftin resurssien toiminta ja rooli kokonaisuudessa. (Duncan & Osborne 2018, 20-24).

4.3.1 Pod

Pod on Kubernetesin ja samalla OpenShiftin keskeisin resurssi jonka avulla kontteja sijoitellaan ajettavaksi klusterissa. Pod voi sisältää yhden tai useamman kontin joilla on vahva riippuvuus toisiinsa ja siksi järkevää sijoittaa samaan podiin. Vaikka yhden prosessin ja kontin malli per pod onkin ideaalisin, useamman kontin ajaminen podissa voi olla käytännön syistä tarpeen, jotta varmistetaan toisistaan riippuvien konttien sijainti samalla isäntäkoneella ja näin ollen nopea tiedonsiirto keskenään. Samassa podissa sijaitsevat kontit jakavat keskenään saman isäntänimen, verkkorajapinnan ja IPC nimiavaruuden. Podissa sijaitsevat kontit skaalautuvat aina yhdessä isäntäkoneille ja niillä on sama IP-osoite. Kontit jakavat podin sisällä saman porttiavaruuden, joten niiden on käytettävä eri portteja. Klusterin sisällä podeilla on sama verkko-osoite avaruus, jolloin podit voivat kommunikoida keskenään IP-osoitteiden avulla. (Lukša 2018, 56-58, Dumpleton 2018, 13).

Valinta sijoitetaanko kontit samaan podiin vai eri podeihin riippuu monesta tekijästä. Kuinka paljon konttien välillä on kommunikaatiota, miten paljon ne riippuvat toisistaan ja onko joku konteista päävastuussa toiminnallisuudesta? Tärkeää podien määrittelyssä on muistaa suunnitteluperiaate, jonka mukaan pod on kevyt, yhtä rajattua toiminnallisuutta tuottava kokonaisuus. Podiin ei ole tarkoitus paketoida koko monoliittista sovellusta. Lukša (2018, 57) listaa kriteerit joilla päätös konttien sijoittamisesta samaan podiin voidaan tehdä:

- Pitääkö konttien sijaita samalla palvelimella, jotta ne voivat käyttää samoja resursseja kuten jaettuja tiedostoja tai muistia?
- Onko kontti itsenäinen kokonaisuus vai tarvitseeko se aina toisen kontin toimiakseen?
- Voiko kontti skaalautua itsenäisesti vai yhdessä toisen kontin kanssa?

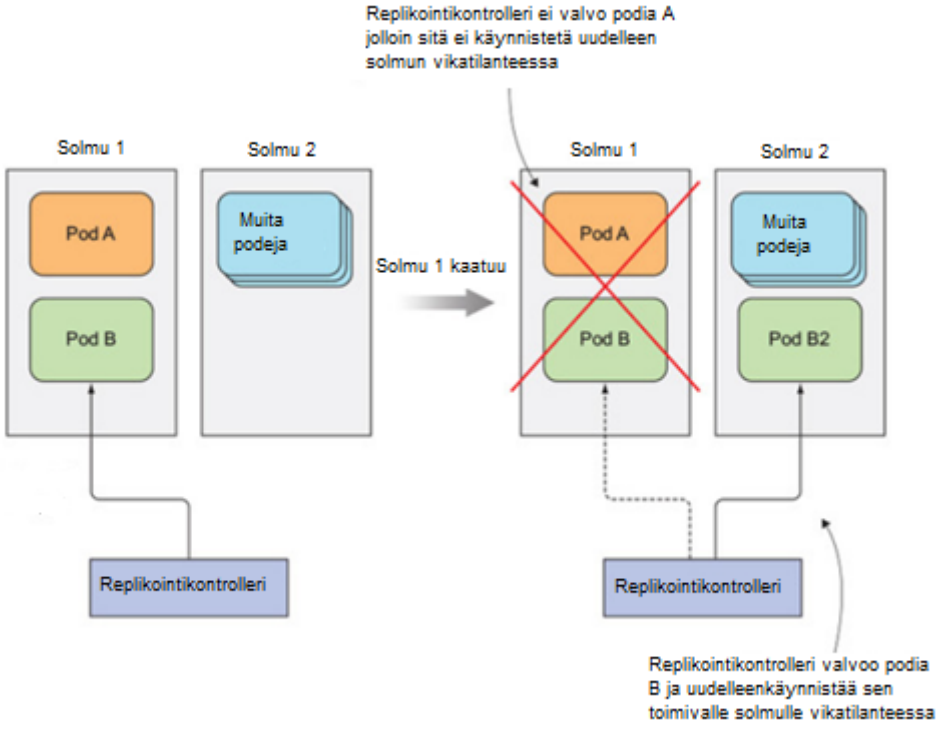
4.3.2 Replikointi kontrolleri (replication controller)

Replikointi kontrollerin tehtävä Kubernetes klusterissa on huolehtia, että haluttu määrä podeja ajetaan klusterissa ja ne toimivat oikein. OpenShiftin käyttämä Kubernetes orkestroi klusteria replikointi kontrollerien avulla niin että se monitoroi podeissa ajettavia sovelluksia ja jos löytää niistä virheen, käynnistää podin uudelleen. Se voi Kubernetesin liveness probe – ominaisuuden avulla tarkistaa, että podin sovellus vastaa HTTP GET pyyntöön onnistuneesti, podiin saadaan avattua TCP yhteys haluttuun porttiin tai suorittaa jonkin komennon podin kontissa ja tarkistaa onnistumisen. Jos kontti ei replikointi-kontrollerin mielestä toimi oikein, se käynnistää podin ja siellä olevat kontit uudelleen. Uudelleenkäynnistys voi tapahtua liveness probe:n lisäksi tilanteissa jossa kontin pääprosessi kaatuu tai koko pod ajetaan alas esimerkiksi palvelimen virhetilanteen tai uudelleenkäynnistyksen vuoksi. Kontrolleri pystyttää uuden podin haluttujen podien määrän mukaisesti, joko samalle palvelimelle jossa podin virhetilanne

havaittiin tai jollekin toiselle klusterissa olevalle palvelimelle. (Lukša 2018, 85-103).

Podin ja kontrollerin sidonta toisiinsa tapahtuu labelin avulla, jolla merkitään OpenShiftissä kaikki toisiinsa liittyvät resurssit. Kontrolleri löytää labelin avulla ne podit, joita se hallitsee ja tarkistaa jatkuvasti, että haluttu määrä podeja on ajossa. Jos podeja on jostain syystä liian vähän, tai se joudutaan uudelleenkäynnistämään, kontrolleri käyttää podin mallipohjaa uuden podin luomiseen. (Duncan & Osborne 2018, 85-89).

Seuraavassa kuvassa esitetty replikointi-kontrollerin toiminta tilanteessa jossa yksi klusterin palvelimista kaatuu virhetilanteeseen:



Kuvio 6. Palvelimen kaatuessa pod luodaan terveelle palvelimelle (Lukša 2018, 94)

4.3.3 Palvelu (service)

Kun Pod luodaan, se saa OpenShift projektin sisällä oman IP osoitteen jonka kautta muut podit voivat kommunikoida sen kanssa. Tämä IP-osoite voi kuitenkin muuttua, kun pod uudelleenkäynnistetään tai asennetaan siitä uusi versio. Tämän vuoksi Kubernetesissa on palvelu-resurssi, jolla luodaan yksittäinen piste, IP-osoite, jonka kautta liikenne ohjataan sen podeihin. IP-osoite on palvelulla muuttumaton, joten kun palvelun takana olevia podeja luodaan ja poistetaan klusterin eri palvelimilla, niihin pääsee silti käsiksi saman IP-osoitteen kautta. Palvelu tuottaa IP-osoitteen lisäksi kuormahallinnan jakaen liikennettä tasaisesti kaikkiin palvelulle kuuluviin podeihin. Se huolehtii, että liikenne ohjataan podeissa oikeaan porttiin eikä palvelun käyttäjän tarvitse huolehtia pal-

velun takana tapahtuvista muutoksista. (Lukša 2018, 121-134. Dumpleton 2018, 93-96).

4.3.4 Reitti (route)

Palvelun kautta OpenShift projektin sisällä olevat podit voivat kommunikoida keskenään muuttumattoman IP-osoitteen kautta. OpenShift luo myös automaattisesti oman kohdealueen nimipalvelun (DNS) jonka kautta IP-osoitteeseen pääsee käsiksi. Kun palvelu halutaan julkaista klusterin ulkopuolelle, luodaan OpenShiftissä reitti. Reitti tuottaa muuttumattoman IP-osoitteen ja kohdealueen nimipalvelun jonka kautta ulkoiset asiakkaat voivat käyttää palvelua. Oletuksena OpenShift nimeää osoitteen klusterin, projektin ja sovelluksen nimien mukaan mutta se voidaan muuttaa halutuksi palvelun HTTP osoitteeksi. Dumpleton 2018, 95-99).

Reitin avulla voidaan muodostaa myös suojattu HTTP tai ei-HTTP yhteys. Yhteys voidaan muodostaa kolmella eri tavalla (Dumpleton 2018, 98):

- Edge: Reititin tunnistaa asiakkaan, terminoi suojatun yhteyden sekä välittää liikenteen salaamattomana OpenShift klusteriin. Suojattu yhteys muodostetaan joko OpenShiftin generoimalla sertifikaatilla tai, käytettäessä omaa isäntäkoneen nimeä, käyttäjän generoimalla.
- Passthrough: Reititin välittää liikenteen sellaisenaan sovellukselle jolla pitää olla asianmukainen sertifikaatti suojauksen terminoimiseen. Tarvittaessa yhteys voidaan konfiguroida myös ei-HTTP liikenteelle.
- Reencrypt: Reititin terminoi suojatun yhteyden mutta salaa liikenteen uudelleen, kun se välitetään sovellukselle. Sovelluksella tulee olla sertifikaatti kunnossa, jotta se kykenee purkamaan salatun liikenteen.

Reittiä voi myös konfiguroida ingress-tyyppiseksi jolloin liikenne voidaan ohjata OpenShift projektiin saman DNS-nimen kautta, käyttäen URL-polkuja halutulle palvelulle ohjaamiseen. (Lukša 2018, 134-141, Sadeghianfar 2018)

4.3.5 Koonti konfiguraatio (build config)

Koonti konfiguraatio on OpenShiftin automaattisen asentamisen lähtöpiste joka mahdollistaa sovelluksen koonnin sen lähdekoodin pohjalta. Konfiguraatio pitää sisällään OpenShift alustasta käytettävän koonti-kontin, linkin koodin lähde-varastoon (esimerkiksi GitHubiin), luotavan levykuvan nimen ja tapahtumat jotka liipaisevat uuden koonnin. Koonti voi käynnistyä, jos lähdekoodi tai koonti kontin levykuva muuttuu. (Duncan & Osborne 2018, 20-36, Dumpleton 2018, 42-45).

4.3.6 Provisiointipohja (deployment config)

Pohja syntyy automaattisesti, kun palvelu provisioidaan, eli otetaan käyttöön OpenShift alustassa, mutta sitä voi myös räätälöidä tarpeiden mukaisesti. Pohjassa määritellään ajettavien replikoiden määrä, raja-arvot podin käyttämille resursseille kuten prosessorin käyttö ja muistin kulutus sekä provisioinnin laukaisevat tapahtumat (esimerkiksi konfiguraation tai levykuvan muutos). Lisäksi pohja määrittelee provisionti-strategian miten palvelun uuden version päivitys klusterissa tapahtuu. Strategioita ovat (Picozzi, Hepburn & O'Connor 2017, 17-22):

- Rullaava provisiointi (rolling deployment) on oletus strategia, jossa palvelun podit päivitetään yksi kerrallaan. Päivityksen jälkeen pod testataan toimivaksi ennen kuin päivitetään seuraava pod. Tällä strategialla varmistetaan katkoton päivitys koska palvelun podeista on ainoastaan yksi kerrallaan alhaalla päivityksen ajan.
- Uudelleen provisiointi (recreate deployment) strategiassa palvelu luodaan aina päivityksen yhteydessä uudelleen. Tätä käytetään, kun palvelusta on vain yksi replika tai siitä voi olla vain yksi versio kerrallaan ylhäällä. Myös pysyvä tietovarasto joka on sidottu palvelinkohtaisesti voi vaatia tämän päivitys strategian.
- Räätälöity provisiointi (custom deployment) mahdollistaa monimutkaisempien strategioiden luomisen niin että voidaan käyttää esimerkiksi blue-green päivitystä, jossa järjestelmästä on yhtä aikaa uusi ja vanha versio käytössä.

Provisiointipohjan avulla palvelu voidaan ottaa käyttöön alustassa aina samoilla periaatteilla mutta parametrisoiden provisiointia tarpeen mukaan.

4.3.7 Levykuvan virta (image stream)

Levykuvan virtoja käytetään automatisoimaan provisiointeja OpenShiftissä. Virta sisältää linkkejä konttien levykuviin joiden avulla se monitoroi levykuvien muuttumista. Kun levykuva päivittyy, virta käynnistää konttien päivitysprosessin klusterissa. Se tuo lisävarmuutta, että oikea versio kontista päivitetään jokaiseen replikaan koska levykuvan versio katsotaan sha256 hashin perusteella eikä standardilla Docker levykuvan tagilla, josta voi olla useita versioita samalla tagilla. (Duncan & Osborne 2018, 26, 93).

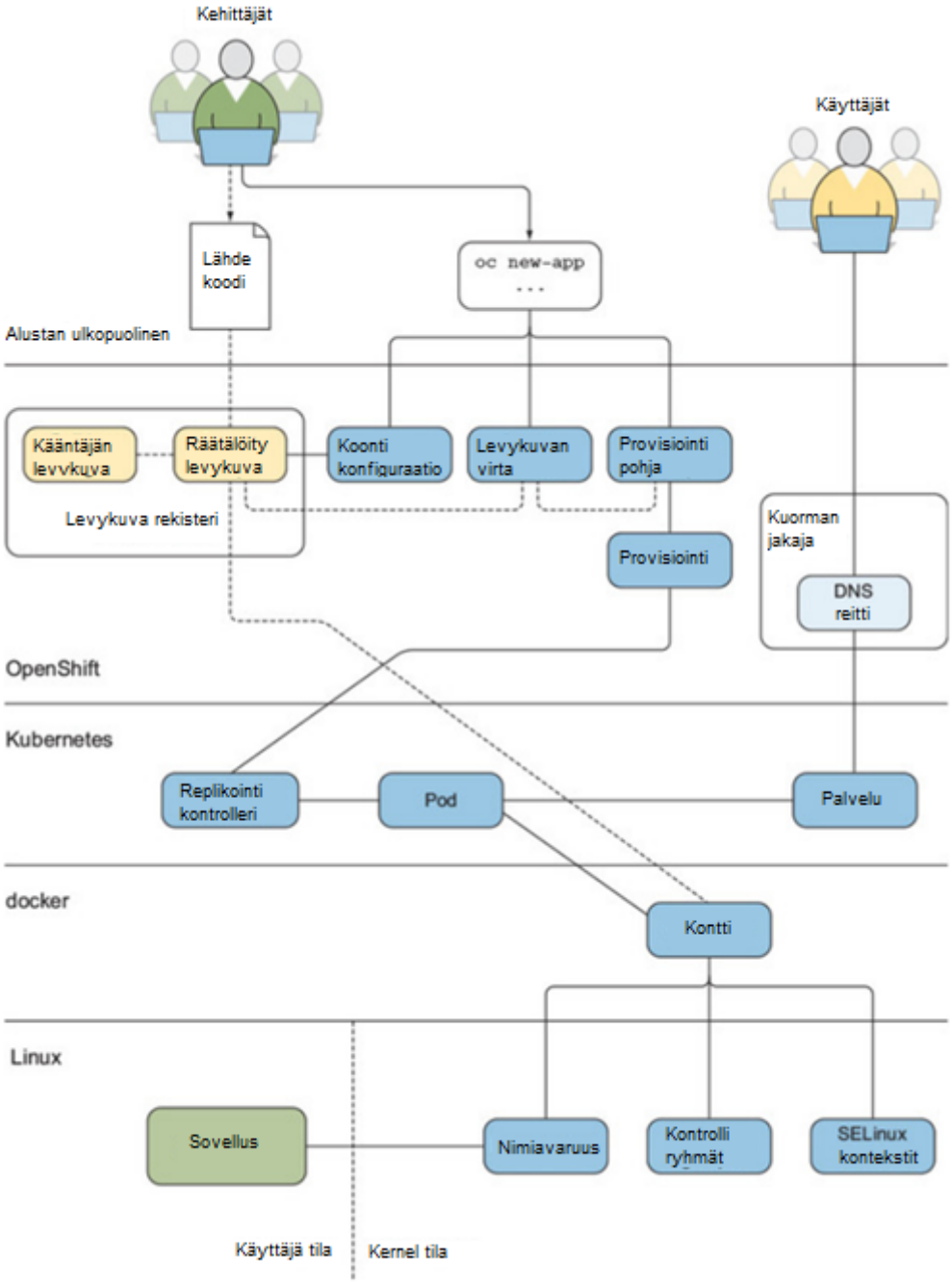
4.4 Resurssien käyttäminen alustassa

Edellisissä luvuissa kuvatuilla resursseilla OpenShift tuottaa PaaS alustan jonka avulla sovelluksen kehittämisen elinkaari voidaan hallita. Se tuottaa mekanismit sovellusten automaattiseen kääntämiseen lähdekoodista, konttien levykuvien tuottamiseen, automaattiseen testaamiseen ja sovellusten asentamiseen

klusterissa. Duncan & Osborne (2018, 38) kuvaa miten resurssit liittyvät toisiinsa ja miten niitä käytetään koko elinkaaren tuottamiseksi:

1. OpenShift luo sovelluksen kontin levykuvan kääntämällä versionhallinnassa olevan lähdekoodin ja tekemällä Docker käännöksen jonka lopputuloksena saadaan uusi levykuva.
2. Levykuva ladataan OpenShiftin konttien levykuva rekisteriin, jossa se on koko klusterin käytössä.
3. Alusta tallentaa koonti konfiguraation joka dokumentoi miten sovellus käännetään ja kootaan yhteen. Konfiguraatio pitää sisällään käytetyn ohjelmointikielen mukaisen koonti levykuvan, lähdekoodin sijainnin sekä muita kokoonpanoon liittyviä tietoja. Konfiguraatiota voi muokata tarpeita vastaavaksi.
4. Alusta luo provisiointipohjan johon tallennetaan kaikki sovelluksen asentamiseen tarvittavat tiedot. Monta replikaa siitä ajetaan, päivitysstrategia, ympäristömuuttujat, palvelinresurssien rajoitteet ja käytetyt levyresurssit.
5. Sovelluksen provisioinnissa tallennetaan asennuksen tiedot, millä parametreilla ja ympäristömuuttujilla sovellus asennettiin klusteriin.
6. Kontit luodaan levykuvan pohjalta käyttäen sisäisesti konttien ajoalustan tuottamia rutiineja jotka erottavat kontit Linux ytimen nimiavaruuksilla ja kontrolliryhmillä.
7. Alusta nostaa ylös palvelun joka huolehtii replikointi-kontrollerin kautta podien luomisesta klusterin palvelimille. Lisäksi palvelu tuottaa kuormanjaon ja DNS-osoitteen jonka kautta palvelun podeihin pääsee käsiksi.
8. OpenShift luo levykuva virran joka monitoroi konttien levykuvien muutoksia ja liipaisee automaattisesti uusien versioiden asennuksen.
9. Klusterin ulkopuolelta käytettäville palveluille luodaan reitti, joka tuottaa muuttumattoman DNS-nimen jonka kautta palvelua voidaan käyttää.

Seuraavassa kuvassa on esitetty OpenShiftin tärkeimmät resurssit ja miten ne liittyvät toisiinsa.



Kuvio 7. Sovelluksen käyttöönoton OpenShift resurssit (Duncan & Osborne 2018, 43)

4.5 Sovellusten käyttöönottomallit

Sovellukset voidaan ottaa käyttöön OpenShiftissä eri tavoilla riippuen missä muodossa sovellus on lähderekisterissä ja onko lähderekisteri verkossa olevalla palvelimella vai OpenShift klusterin omassa rekisterissä. Päätavat ottaa sovellus käyttöön ovat (Dumpleton 2018, s. 23)

- sovelluksen provisiointi olemassa olevasta levykuvasta
- sovelluksen provisiointi lähdekoodin pohjalta
- sovelluksen provisiointi Dockerfile määrityksestä.

4.5.1 Sovelluksen provisiointi olemassa olevasta levykuvasta

OpenShiftillä voidaan ajaa kontin levykuvan perusteella kontteja. Levykuva voi olla tallennettuna mihin tahansa lähderekisteriin johon OpenShift klusterista on pääsy. Levykuvia voidaan hakea esimerkiksi Docker Hubista tai Red Hatin omasta levykuva katalogista, joissa on tuhansia valmiita levykuvia eri sovelluksista. Palveluihin voi myös tallentaa omia levykuvia tai vaihtoehtoisesti ylläpitää niitä omassa rekisterissä johon OpenShift klusterista on pääsy.

Sovelluksen ajaminen onnistuu OpenShiftin oc työkalulla, jolle annetaan sovelluksen luontikomento ja parametrina levykuvan osoite. Tämän pohjalta OpenShift luo podin, levykuvan virran, provisiointipohjan, palvelun sekä replikointi kontrollerin. Levykuva haetaan ulkoisesta rekisteristä OpenShift klusterin omaan rekisteriin, ja levykuvan virta (imagestream) huolehtii, että uuden version ilmestyessä ulkoisen rekisteriin, se päivittää oman paikallisen rekisterin ja aloittaa podin päivitysprosessin. (Dumpleton 2018, s. 29-39).

4.5.2 Sovelluksen provisiointi lähdekoodin pohjalta

Edellisessä levykuvaan pohjautuvassa provisioinnissa, sovelluksen lähdekoodin kääntäminen ja levykuvan luonti on pitänyt tehdä erillisessä kehitysympäristössä, jossa valmis levykuva on koostettu. Lähdekoodi-pohjainen provisiointi automatisoi nämä vaiheet OpenShift alustan sisälle. Se käyttää Source-to-Image (S2I) kääntäjiä joita on olemassa useimpiin sovelluskehitys ympäristöihin. S2I kääntäjä on käytännössä OpenShiftin rekisterissä oleva levykuva joka sisältää kyseisen kehitysympäristön kirjastot ja ohjelmointikielen (esim. Java) kääntäjän. Lähdekoodi-pohjaisessa provisioinnissa annetaan versionhallinta säilön osoite, josta koodi löytyy, ja S2I levykuvan nimi. Provisiointi hakee ensin lähdekoodin versionhallinnasta, usein GitHubista, injektoi sen S2I kontille joka kääntää koodin. Käännöksestä syntyvä ajettava lopputuote (esim. Java jar) viedään levykuvan koostamiseen, jonka tuloksena syntyy sovelluksen uusi levykuva. Tämä tallennetaan levykuva virran määrittelemään paikkaan, josta se otetaan käyttöön sovelluksen asennuksessa klusterin palvelimille. (Dumpleton 2018, s. 41-49, Picozzi, Hepburn & O'Connor 2017, 75-81).

Sovelluksen provisioinnissa syntyy samat resurssit kuin levykuvasta provisioinnin yhteydessä, mukaan lukien koonti konfiguraatio jonka merkitys korostuu. Konfiguraation pohjalta tehdään automaattisesti uusi koonti, kun lähdekoodi tai joku muu konfiguraatioon liittyvä resurssi muuttuu. Konfiguraatiota voi myös räätälöidä omiin tarpeisiin esittelemällä ympäristömuuttujia joita käytetään lopputuotteiden kääntämisessä tai levykuvan kokoamisessa.

Tarvittaessa on myös mahdollista tehdä omia S2I kääntäjien levykuvia tekemällä Dockerfilen, jossa pohjana olevaa S2I räätälöidään. Omalla kääntäjällä

voidaan räätälöidä koontiprosessia asentamalla tarvittavia kirjastoja, korvaamalla oletus skriptit (assemle, run ja usage) paketin kokoamiseen ja ajamiseen tai lisäämällä jotain muuta käännöksessä tarvittavaa konfiguraatiota. Räätälöity S2I kääntäjä pitää olla yhteensopiva OpenShiftin S2I määrityksien kanssa, jotta sitä voidaan käyttää standardiin tapaan provisioinneissa. Levykuva tulee myös tallentaa klusterin käyttämään levykuva säilöön, josta se voidaan ladata provisioinneissa. (Dumpleton 2018, 65-71).

4.5.3 Sovelluksen provisiointi Dockerfile määrityksestä

Kolmas päätapa provisioida sovelluksia on tehdä se Dockerfile määrityksen avulla. Vaikka S2I käännöstä ja koottavaa levykuvaa voidaan räätälöidä monella tapaa, koostamista rajoittaa kuitenkin S2I levykuvan ominaisuudet ja itse käännöksen prosessi. Jos halutaan saada täysi kontrolli, miten levykuva luodaan, tarvitaan oma Dockerfile. Dockerfile on konttialustan käyttämä määritystiedosto, jolla se kokoaa halutun levykuvan docker build -komennolla. Dockerfilen tärkein käsky on pohjana käytettävän levykuvan nimi, jonka päälle kootaan sovelluksen tarvitsema konfiguraatio ja itse sovellus, esimerkiksi Javan jartiedosto. (Duncan & Osborne 2018, 3.2.3, Dumpleton 2018, 51).

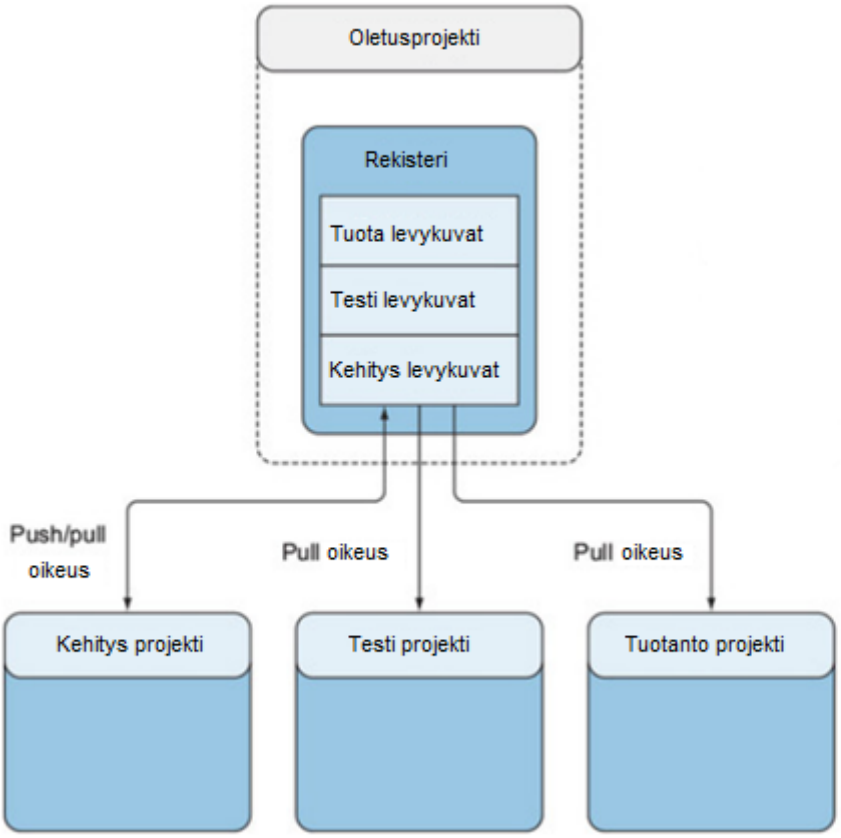
Sovellus provisioidaan samaan tapaan kuin olemassa olevasta levykuvasta tai S2I kääntäjällä, mutta koonti strategiaksi valitaan docker. Provisoinnissa annetaan Dockerfilen sisältävän GitHub säilön osoite, jonka avulla versionhallinnassa oleva Dockerfile injektoidaan prosessiin ja tehdään koonti. Provisioinnissa syntyy samat resurssit kuin muillakin provisiointitavoilla. Luotu levykuvan virta resurssi huolehtii saadun sovelluksen levykuvan viemisestä levykuva säilöön ja päivittämisestä uuden provisioinnin yhteydessä. (Dumpleton 2018, 51-54).

4.6 Jatkuva testaus ja käyttöönotto

Edellisissä luvuissa kuvatut resurssit ja provisiointi-prosessit mahdollistavat jatkuvan testauksen ja käyttöönoton käytännöt niin että sovellus voidaan viedä tuotantoon asti automaattisen kehitysprosessin läpi. Kun versionhallintaan viedään koodimuutos, laukaisee se automaattisen koonnin, asentamisen ja testauksen prosessit, jonka päätteeksi sovellus asennetaan tuotantoon asti. (Duncan & Osborne 2018, 108-110).

OpenShiftissä on tavat, joilla jatkuvan integraation ja käyttöönoton prosessit voidaan automatisoida. Sovelluksen provisiointiin voidaan injektoida ympäristömuuttujia tai monimutkaisempien rakenteiden ollessa kyseessä, konfiguraatiokuvauksia (config map). Näiden avulla podit voidaan konfiguroida eri testiympäristöihin niin, ettei uutta käännöstä tarvita. Asentaminen ylempään testiympäristöön voidaan liipaista käyttämällä levykuvan tageja; kun tietty versio on testattu onnistuneesti esimerkiksi kehitysympäristössä, voidaan se

asentaa testiympäristöön merkitsemällä levykuvaan tag, jonka perusteella levykuvan virta havaitsee uuden version ja liipaisee asennuksen. Samalla tavalla, kun levykuva on testattu onnistuneesti testiympäristössä, voidaan se merkitä tagilla asennettavaksi tuotantoympäristöön. Kun asentamisen automatisoinnin lisäksi yksikkö-, toiminnalliset- ja suorituskyky-testit tehdään automaattisesti, saadaan koko käyttöönottoprosessista automaattinen. Alla olevassa kuvassa on esitetty ympäristöjen jakaantuminen OpenShift projekteihin ja miten ne saavat haettua uuden version levykuvasta.



Kuvio 8. Levykuvien käyttö ajoympäristöissä (Duncan & Osborne 2018, 104)

Uusien versioiden koostamista, testaamista ja merkitsemistä seuraaviin ympäristöihin hallitaan OpenShiftissä Jenkins työkalulla. Jenkins on prosessimoottori, jolla voidaan automatisoida sovelluksen kääntäminen, testaaminen ja asentaminen. Se hyödyntää plugin tekniikkaa jolla voidaan joustavasti lisätä siihen uusia ominaisuuksia. Jenkinsille määritellään käyttöönotto prosessi työjonoiksi joissa yksittäisen työn, esimerkiksi toiminnallisen testin, onnistumisen jälkeen liipaistaan seuraava työ. Jos työ taas epäonnistuu, esimerkiksi toiminnallinen testi menee virheeseen, työjono katkeaa eikä uutta versiota viedä enää eteenpäin seuraavaan ympäristöön. (Sadeghianfar 2016).

5 JAVA KÄYTTÖLIITTYMÄN MIGRAATIO OPENSHIFTILLE

Pääluvussa kuvataan valitun sovelluksen migraatio OpenShift alustalle. Miksi sovellus valittiin migroitavaksi, ja mikä migraatio-strategia valittiin. Mitkä ovat migraation vaiheet, käytännön toimenpiteet ja miten sovellus käyttäytyy uudessa ympäristössä.

5.1 Migroitavan järjestelmän valinta

Tutkimus sijoittuu terveydenhuolto-alalla toimivaan organisaatioon, jossa tuotetaan sähköisiä palveluja sosiaali- ja terveydenhuollon toimijoille. Sähköisten palvelujen kehitys on lähtenyt liikkeelle lähes 15 vuotta sitten ja tarkastelun alla oleva kokonaisuus on otettu käyttöön noin 10 vuotta sitten. Käyttöönoton jälkeen järjestelmä on ollut säännöllisen muutoksen kohteena, kun olemassa oleviin palveluihin on lisätty ominaisuuksia ja uusia palveluja on kehitetty.

Tarkastelun alla olevan kokonaisuus on kehitetty pääsääntöisesti 2000-luvun valtavirta-ratkaisuilla vastaamaan sen hetkisiä suorituskyky ja saatavuus vaatimuksia. Palvelut ovat klusteroituja ja jaettuna useampaan putkeen niin että yksittäiset pyynnöt suoritetaan aina samassa putkessa alusta loppuun. Käytössä on client-server malli jossa palvelusovellukset tuottavat palvelinpään toteutuksen ja asiakassovellukset käyttöliittymän. Vuosien varrella tehtyjen useiden muutosten jäljiltä palvelusovellukset ovat kasvaneet kokoa ja monimutkaisuutta niin että uusien muutosten tekeminen on jatkuvasti haasteellisempaa. Palvelukehityksessä voidaan puhua normaalista tietojärjestelmän vanhenemisesta joka johtaa haasteisiin ylläpidossa.

Organisaation ylläpitämät palvelusovellukset ovat laajasti käytössä terveydenhuollossa ja niiden merkitys toimijoiden liiketoiminnalle on tärkeä. Palvelujen saatavuusvaatimukset ovat korkealla tasolla ja ne on myös saavutettu hyvin tähän mennessä. Tarve muutokselle tulee lähinnä yleisestä teknologiakehityksestä, jossa halutaan pysyä mukana toisaalta työvoiman saatavuuden kan-

nalta ja toisaalta tehokkaan toiminnan. Jos käytettävät teknologiat pääsevät vanhentumaan, ei uusia sovelluskehittäjiä ole helppo löytää, jos vanhoja osaajia siirtyy muualle. Uudet teknologiat ja arkkitehtuurit tuovat myös parannuksia sekä työntekijä- että palvelinresurssien tehokkaampaan käyttöön. Tämä tutkimus on osa esiselvitystä, jossa pyritään löytämään migraatio-polku uuteen teknologiaan, jotta kriittisten palvelusovellusten migraatio menee jouhevasti ilman että palvelujen saatavuus kärsii. Työn tilaajan tarkoitus oli koestaa pilviteknologiaa ensin vähemmän kriittisellä järjestelmällä, jotta kriittisimpien palvelusovellusten migraatio onnistuisi kokemusten pohjalta hyvin. Tähän tarpeeseen valittiin nykyisten palvelusovellusten päälle rakennettu käyttöliittymä, jonka käyttäjäkunta ja käyttötarve on rajallinen. Vastaavia palveluita käyttäviä järjestelmiä on useita muitakin terveydenhuollon toimijoilla. Tavoitteena migraatiossa oli koestaa sovellusta testiympäristöstä tuotantoon asti.

5.2 Järjestelmän kuvaus

Migroitava järjestelmä on web-käyttöliittymä jonka käyttäjiä ovat terveydenhuollon ammattihenkilöt. Se on otettu käyttöön alle viisi vuotta sitten, jonka jälkeen siihen on kohdistunut pieniä muutostarpeita. Järjestelmä on tehty Java 8 ympäristöön IBM Websphere Liberty Profile (WLP) sovelluspalvelimelle ja sen käyttöliittymä osuudet on toteutettu Java Server Faces (JSF) ja Prime Faces komponenteilla. Sovellus on kooltaan keskikokoinen, sisältäen useita kymmeniä tuhansia koodirivejä ja se on paketoitu yhteen ajettavaan war-komponenttiin ja muutamaan jar-kirjastoon. Sovellus käyttää SOAP-rajapintojen kautta useampaa liiketoimintapalvelua toiminnallisuuden tuottamiseksi. Yksi liiketoimintapalveluista tuottaa käyttöliittymän käsittelemän tiedon ja muut ovat tunnistautumiseen, ammattioikeuteen, henkilötietoihin, asiakastietoihin sekä kooditettuihin tietoihin liittyviä palveluja. Palveluista kooditettujen tietojen palvelu ajetaan samalla WLP sovelluspalvelimella web-käyttöliittymän kanssa.

5.3 Migraatiostrategian valinta

Pilvipalveluihin siirtymisessä tulee kysymykseen olemassa olevan järjestelmän arkkitehtuurin muokkaaminen mikropalveluiksi. Mikropalvelut käsitetään usein pilvipalvelujen natiiviksi arkkitehtuuriksi, jolla saadaan pilvestä suurin hyöty skaalauksen, suorituskyvyn ja saatavuuden näkökulmasta. (Laszewski ym. 2018, 16-17).

Migroitavan järjestelmän arkkitehtuuria pohdittiin sovelluskehitys-tiimin työpajassa, jossa koko tiimin kehittämän tuotealueen arkkitehtuurin tiekarttaa hahmoteltiin. Järjestelmän osalta todettiin, että sen nykyistä kerrosarkkitehtuuria voisi yksinkertaistaa vähentämällä kerroksia jolloin ylläpitotyö voisi nopeu-

tua. Myös muita parannusideoita tuli esiin, muun muassa käyttöliittymäkirjastoa voisi harkita vaihdettavaksi koska nykyisen kirjaston käyttö lisää monimutkaisuutta koodiin ja hidastaa sovelluksen käyttöä. Myös kehitysympäristön hitaus aiheuttaa viivettä sovelluksen koontiin ja ajamiseen lokaalisti. Edellä kuvatut parannustarpeet todettiin kuitenkin vähemmän kriittisiksi, erityisesti pilvipalveluun siirtymisen näkökulmasta koska ne eivät vaikuttaisi järjestelmän arkkitehtuuriin tai ajettavien sovellusten määrään. Toisin sanoen järjestelmän prosessinäkymä (4+1 malli) ei muuttuisi, vaikka muutokset tehtäisiin, jolloin järjestelmän provisiointi pilvipalveluun olisi samankaltainen. Tästä syystä päätettiin priorisoida tuotealueen muita kehityskohteita ensin ja tehdä migroitavan järjestelmän muutokset mahdollisesti myöhemmin. Näin ollen migroitava käyttöliittymä päätettiin viedä OpenShiftille rehost-menetelmällä (Laszewski ym. 2018), jossa migroitavaan järjestelmään ei tehdä muutoksia, ainoastaan ajoalusta vaihtuu. Tällä tavalla saataisiin mahdollisimman nopeasti kokemuksia pilvipalvelu-alustasta, joita voitaisiin hyödyntää tuotealueen muiden migraatioiden tekemisessä.

5.4 OpenShift provisiointitavan valinta

Järjestelmää ajetaan WLP sovelluspalvelimella jolle provisioidaan web-käyttöliittymä ja kooditettujen tietojen palvelu. Muut järjestelmän käyttämät palvelut sijaitsevat muilla sovelluspalvelimilla ja niitä käytetään SOAP-rajapintojen kautta. Migraatio päätettiin tehdä rehost-strategialla (kts. 5.3) jolloin järjestelmän arkkitehtuuriin ei kosketa. Järjestelmä pitää sisällään paljon konfiguraatiota mm. käytettäviin palveluihin ja sovelluspalvelimeen liittyen sekä xml-muotoista dataa liiketoiminnallisuuden lähtötietoja varten. Edellä mainituista syistä johtuen valmista S2I koonti levykuvaa ei OpenShiftin levykuva katalogista löydy, joten sovellusten provisiointi lähdekoodin pohjalta vaatisi omaa S2I levykuvan tekemistä. Toisaalta useamman sovelluksen ajaminen samassa kontissa on epäsuunnittelumalli pilvialustoissa (Richardson 2018, 40, Wiggins 2017), joten tähän harjoitukseen ei haluttu lähteä. S2I tyyppinen provisiointi vaatisi molempien sovellusten kontitusta erikseen ja jonkin verran konfiguraation muutosta. Tässä vaiheessa katsottiin kuitenkin järkeväksi, että tehdään puhdas rehost strategian mukainen migraatio, jolloin OpenShift alustasta saadaan migraatiota tehdessä mahdollisimman nopeasti kokemuksia.

5.5 Sovellusten kontitus

OpenShift alustaan voidaan asentaa OCI standardin mukaisia kontteja, joita voidaan tehdä tässä migraatiossa käytetyllä Docker välineellä (Lossent ym. 2017). Dockerille määritellään kontin rakenneosat Dockerfile tiedostolla, johon

kuvataan web-käyttöliittymän vaatimat resurssit ajoympäristössä. Näitä ovat sovelluksen tarvitsemat konfiguraatiotiedostot, hakemistorakenteet, tietokanta-ajuri, käyttöjärjestelmän apuohjelmistot, pohjalla oleva levykuva sekä ajettavat binääritiedostot. Migroitavalle web-käyttöliittymällä oli tehty jo aiemmin alustava Dockerfile määritys, jota tässä työssä arvioitiin ja muokattiin tarvittavilta osin.

Kontin muodostamisessa tärkeä vaihe on valita tarvittava levykuvan pohja (base image). Käyttöliittymä toimii WebSphere Liberty Profile (Liberty) sovelluspalvelimella Linux ympäristössä, joten käytettävässä pohja levykuvassa tulee nämä osat olla mukana. Valmiita levykuvia löytyy muun muassa Docker Hubista, joka on Docker yhtiön ylläpitämä vapaasti käytettävissä oleva levykuva rekisteri. Sieltä löytyy myös käyttöliittymän vaatima levykuva, jossa on Ubuntu Linux, Java 8 JRE, Java EE kirjastot sekä Libertyn uusin versio joka tukee edellä listattua teknologia pinoa. Koska migraatiostrategiaksi valittiin sovelluksen uudelleensijoitus pilveen ilman arkkitehtuuri-muutoksia tätä pohja levykuvaa ei lähdetty vaihtamaan olemassa olevasta Dockerfilestä. Sen sijaan määritystä pyrittiin parantamaan OpenShift Documentation (2020) ja Docker Documentation (2020) lähteiden mukaisilla toimenpiteillä joilla levykuvan kokoa pyrittiin pienentämään sekä tietoturvaa parantamaan. Levykuvan optimoimiseksi hyödynnettiin seuraavia periaatteita:

- RUN, COPY ja ADD käskyt luovat levykuvaan aina uuden kerroksen jotka vievät tilaa levykuvassa. Optimoitiin kerrosten määrää ketjuttamalla Dockerfile käskyjen ajamia Linux komentoja.
- Linuxin paketti-managerilla tapahtuvissa asennuksissa syntyy yleensä väliaikaistiedostoja jotka kannattaa poistaa apt-get clean komennolla.
- ADD käskyn hyödyntäminen pakattujen tiedostojen purkamisessa levykuvaan.
- COPY käskyn käyttäminen ADD käskyn sijaan tiedostojen ja hakemistojen kopioinnissa.
- Docker koonnin kontekstihakemistojen optimointi ja turhien tiedostojen poistaminen.
- Kontin käyttäjä vaihdetaan koonnin jälkeen ei-rootiksi, jolloin kontin väärinkäyttö mahdollisen tietomurron yhteydessä vaikeutuu.

Edellä mainituissa lähteissä on paljon muitakin keinoja levykuvan koon pienentämiseen, mutta niitä ei tämän tutkimuksen puitteissa kokeiltu. Dockerfilen optimoinnin lisäksi Libertyn ominaisuuksia tutkimalla voisi olla mahdollista pienentää pohja levykuvan kokoa käyttämällä Libertyn kernel pohja levykuvaa ja lisäämällä siihen vain tarvittavat ominaisuudet.

Alla olevassa listauksessa on optimoitu Dockerfile, johon edellä kuvattuja optimointiohjeita on sovellettu. Listauksessa osa tiedoista on obfuskoitu.

```
FROM nexus.repo.fi:18444/websphere-liberty:javaee7

ENV LANG en_US.UTF-8
ENV LICENSE accept

USER root

RUN apt-get update -y \
 && apt install -y unzip \
 && /opt/ibm/wlp/bin/installUtility install defaultServer \
 && apt-get clean

COPY target/webui-config*.zip /tmp/
COPY target/data*.zip /opt/webui/data/lib/

RUN mkdir -p /opt/webui-wlp/webui/config \
   && cd /opt/webui-wlp/webui/config \
   && unzip /tmp/webui-config*.zip \
   && cd /opt/webui-wlp/datapalvelu/lib \
   && unzip *.zip \
   && rm *.zip \
   && cp /opt/webui-wlp/webui/config/libertyServer/* \
         /opt/ibm/wlp/usr/servers/defaultServer/ \
   && mkdir -p /opt/oracle \
   && mkdir -p /opt/webui-wlp/datapalvelu/lib \
   && mkdir -p /opt/ibm/wlp/etc


COPY target/ojdbc*.jar /opt/oracle/
COPY target/webui*.ear \
     /opt/ibm/wlp/usr/servers/defaultServer/apps/webui.ear
COPY target/koodisto*.ear \
     /opt/ibm/wlp/usr/servers/defaultServer/apps/koodisto.ear

ADD kpalvelu /opt/

ADD *.properties /opt/webui-wlp/webui/config/
ADD *.jks /opt/ibm/wlp/output/defaultServer/resources/security/

RUN chgrp -R 0 /opt && \
    chmod -R g=u /opt

USER 1001
```

Uusi versio kontista tehdään docker enginen shellin avulla ja viedään levykuva rekisteriin seuraavilla komennoilla:

```
[sh]$ docker build –t webui-wlp –file=webui-wlp/Dockerfile .
[sh]$ docker tag webui-wlp nexus.repo.fi:18445/webui-wlp
[sh]$ docker push nexus.repo.fi:18445/webui-wlp
```

5.6 Käyttöönotto OpenShift alustassa

Tutkimuksen tilaajalla on asennettu OpenShift v.4 klusteri omaan konesaliin, johon järjestelmän käyttöönotto tehtiin. Luvun 5.5 mukainen levykuva otettiin käyttöön järjestelmälle tehdyssä OpenShift projektissa, joka erottelee järjestelmän vaatimat sovellukset ja konfiguraation muista järjestelmistä.

5.6.1 Projektin luonti ja sovelluksen provisiointi

Uuden sovelluksen luonti levykuvasta vaatii oikein konfiguroidun projektin, josta löytyy muun muassa; ulkoisen levykuva rekisterin tunnukset (secret) ja oikeudet sen käyttöön. Projekti ja tarvittava konfiguraatio luotiin OpenShiftin oc työkalulla:

```
[sh]$ oc new-project webui-wlp
[sh]$ oc create secret docker-registry nexus-dockercfg-secret \
      > --docker-server nexus.repo.fi \
      > --docker-username user \
      > --docker-password password
[sh]$ oc policy add-role-to-user system:image-puller \
      > user_name -n project_name
```

Projektiin provisioidaan uusia sovelluksia new-app komennolla jota tässä käytettiin asentamaan sovellukset olemassa olevan levykuvan pohjalta. Webui ja koodisto sovellukset sisältävä levykuva asennettiin projektiin komennolla:

```
[sh]$ oc new-app --docker-image=nexus.repo.fi:18444/webui-wlp:latest \
      > --name=webui
```

Komento hakee sovelluksen levykuvan ulkoisesta levykuvarekisteristä, luo provisiointipohjan projektiin (deployment config), provisiointipohjan käyttämän replikointikontrollerin sekä laittaa levykuvan ajoon yhteen podiin. Sovellukselle myös luodaan palvelu, jonka kautta kaikkiin sen podeihin pääsee käsiksi. Luotuja resursseja voi tarkastella oc työkalun avulla:

```
[sh]$ oc get all -l app=webui
NAME                 READY   STATUS    RESTARTS   AGE
pod/webui-23-kdp7q   1/1     Running   0          8d

NAME                              DESIRED   CURRENT   READY   AGE
replicationcontroller/webui-23    1         1         1       8d

NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
service/webui    ClusterIP   172.30.65.228   <none>        9080/TCP,9443/TCP

NAME                                       REV.  DESIRED CURRENT TRIGGERED BY ...
deploymentconfig.apps.openshift.io/webui 23   1       1       config,image(web

NAME                                    IMAGE REPOSITORY                    ...
imagestream.image.openshift.io/webui    image-registry.openshift-image-
                                        registry.svc:5000/webui-wlp/webui
```

Luoduista resursseista sovelluksen konfiguraatio tallentui provisointipohjaan (deploymentconfig) ja jonka avulla se voidaan provisoida uudelleen täsmälleen samalla konfiguraatiolla. Alla listattu provisiointipohjaan luotuja määrityksiä.

```
[sh]$ oc describe dc/webui
Name:      webui
Namespace: resepti-te
Created:   10 minutes ago
Labels:    app=webui
Annotations: openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Selector:  app=webui,deploymentconfig=webui
Replicas:  1
Triggers:  Config, Image(webui@latest, auto=true)
Strategy:  Rolling
Template:
Pod Template:
  Labels:  app=webui
           deploymentconfig=webui
  Annotations: openshift.io/generated-by: OpenShiftNewApp
  Containers:
   webui:
    Image: nexus.repo.fi:18444/webui
      @sha256:e8d38876c2df00b5cbc5650d8514ec6f3a854a54a5b845d2d5468977b8466997
    Ports:            9080/TCP, 9443/TCP
    Host Ports:       0/TCP, 0/TCP
    Environment:      <none>
    Mounts:           <none>
  Volumes:            <none>
...
```

5.6.2 Resurssien määritys ja palvelun skaalaus

Käytetty sovelluksen levykuva provisioitui klusteriin mutta sovellusten lokilta huomattiin, etteivät palvelut nousseet ylös liian vähäisen muistin vuoksi. OpenShiftin allokoima muisti selvitettiin komennolla:

```
[sh]$ oc describe pod webui-1-kzqkz
Name:       webui-1-kzqkz
Namespace:  project-te
...
    Limits:
      cpu:     600m
      memory:  200Mi
    Requests:
      cpu:     50m
      memory:  200Mi
```

Näitä arvoja verrattiin Docker kehitysympäristössä ajettuun konttiin, jonka käyttämä muisti saatiin komennolla:

```
[sh]$ docker stats
CONTAINER ID   NAME   CPU %  MEM USAGE / LIMIT   MEM %  NET I/O              ...
90ce750181fd   webui  0.42%  676.5MiB / 7.79GiB  8.48%  9.28kB / 6.91kB
```

Tästä voitiin päätellä, että OpenShift allokoi oletuksena liian vähän muistia podille. Koska sovellus vei ilman kuormaa 676 MT muistia, päätettiin allokoida 1 GT podin käynnistyksessä ja rajata maksimi allokoitava muisti 2 GT:uun. Arvoja tarkastellaan myöhemmin pitkäkestoisten suorituskykytestien yhteydessä.

Edellä kuvattujen toimenpiteiden jälkeen sovellus oli käynnissä OpenShift klusterissa ja sen aloitussivulle pääsi käsiksi kirjautumalla oc työkalun avulla klusterissa olevaan podiin ja testaamalla pääsivua lokaalisti:

```
[sh]$ oc rsh webui-23-kdp7q bash
1000830000@webui-23-kdp7q:/$ curl http://localhost:9080
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head id="j_id_4                ...
...
```

Tässä vaiheessa podiin ei päässyt vielä käsiksi klusterin ulkopuolelta vaan sille oli allokoitu OpenShift palvelun kautta yksi IP osoite, jonka kautta sen podeihin pääsi sisäisesti kiinni. Palvelun julkaisu OpenShift klusterin ulkopuolelle tehtiin komennolla:

```
[sh]$ oc create route edge --service=webui --insecure-policy=Redirect \
      --hostname=webui-te.project.apps-testi.x.y.fi
route.route.openshift.io/webui created
```

Komento luo yhteyspisteen jossa on edge-tyyppinen suojatun TLS yhteyden terminointi ja uudelleen ohjaus kuormanjakajalla HTTP (9080) portista suojattuun HTTPS (9443) porttiin. Luotua reitti resurssia voi tarkastella seuraavalla komennolla:

```
[sh]$ oc get route -l app=webui
NAME   HOST/PORT                            SERVICES   PORT       TERMINATION
webui  webui-te.project.apps-testi.x.y.fi   webui      9080-tcp   edge/Redirect
```

Edellä kuvattujen toimenpiteiden jälkeen sovellus oli käytettävissä OpenShiftillä muuttumattoman osoitteen kautta yhdessä podissa. Seuraavaksi hyödynnettiin OpenShiftin replikointi ominaisuutta lisäämällä useampi pod ajoon siltä varalta, että yksittäisessä podissa olisi ongelmia. Tällä saadaan parempi saatavuus palvelulle, kun sen käyttö ei ole riippuvainen yhden

instanssin olemassaolosta, sekä lisää suorituskykyä, kun palvelupyyntöihin vastaa useampi instanssi. Replikoiden määrä asetettiin koontikonfiguraatioon muokkaamalla replicas-asetusta:

```
[sh]$ oc scale dc webui --replicas=2
deploymentconfig.apps.openshift.io/webui scaled

[sh]$ oc describe dc/webui
Name:      webui
Labels:    app=webui
...
Selector:  app=webui,deploymentconfig=webui
Replicas:  2
Triggers:  Config, Image(webui@latest, auto=true)
...

[sh]$ oc get pods -l app=webui
NAME            READY   STATUS    RESTARTS   AGE
webui-2-b9z7x   1/1     Running   0          47h
webui-2-zttxx   1/1     Running   0          7m5s
```

Koontikonfiguraation muokkaus liipaisi automaattisesti palvelun replikointikontrollerin käyntiin, joka nosti toisen podin ylös ja päivitti kuormanjakajalle tiedon toisen podin olemassaolosta. Kuormanjakaja välittää palvelupyynnöt round-robin menetelmällä, eli käytännössä kaikille podeille tasapuolisesti. Koska kyseessä on käyttöliittymä-sovellus joka ei ole tilaton, tuottaa OpenShiftin kuormanjakaja lisäksi evästeen, jonka avulla web-selain istunnon pyynnöt menevät aina samalle podille jolloin istunto säilyy käyttöliittymässä navigoinnin ajan (Picozzi ym. 2017, 30).

5.6.3 Levyn käyttö ja palvelun valvonta

Seuraavaksi määritettiin sovellusten lokien tallennus pysyvään tallennuspaikkaan. Koska kontti on määritelmänsä mukaisesti ephemeraalinen, eli se ei talleta mitään tietoja pysyvästi, tuli sovellukselle määrittää lokien tallennuspaikka, joka säilyy myös podin uudelleenkäynnistyksen jälkeen. Tähän tarpeeseen OpenShiftissä määriteltiin levyn käyttö volume-määrityksellä. Koska sovelluspalvelimelle provisiointiin kaksi sovellusta, molempien lokitiedostojen sijainnit persistoitiin:

```
[sh]$ oc set volume dc/webui --add --type=pvc --claim-size=512Mi \
      --claim-mode=ReadWriteMany --claim-name=webui-loki-claim \
      --name=webui-lokit --mount-path=/opt/webui-wlp/webui/logs
deploymentconfig.apps.openshift.io/webui volume updated

[sh]$ oc set volume dc/webui --add --type=pvc --claim-size=256Mi \
      --claim-mode=ReadWriteMany --claim-name=koodisto-loki-claim \
      --name=koodisto-lokit --mount-path=/opt/koodisto/log
deploymentconfig.apps.openshift.io/webui volume updated
```

Komennolla luodaan ensin pysyvä tallennuspaikka varaus (Persistent Volume Claim) OpenShiftin levyresursseihin, tehdään uusi tallennuspaikka, joka liitetään varaukseen sekä linkitetään kontin sisäinen hakemisto osoittamaan tähän tallennuspaikkaan. Tämän jälkeen hakemistoon tallentuvat lokitiedostot ovat saatavilla myös kontin uudelleenkäynnistyksen jälkeen esimerkiksi virhetilanteiden selvitystä varten. Sovellusten tallennuspaikka varaus (pvc) tehtiin ReadWriteMany tyyppiseksi jolloin useampi pod instanssi voi yhtäaikaisesti käyttää varausta. Luotuja tallennusresursseja voitiin tarkastella seuraavilla komennoilla:

```
[sh]$ oc set volume dc/webui
deploymentconfigs/webui
  pvc/webui-loki-claim (allocated 512MiB) as webui-lokit
    mounted at /opt/webui-wlp/webui/logs
  pvc/koodisto-loki-claim (allocated 256MiB) as koodisto-lokit
    mounted at /opt/koodisto/log

[sh]$ oc get pvc
NAME                STATUS   VOLUME                      CAPACITY   ACCESS MODES
koodisto-loki-claim Bound    pvc-c1b95e9a-9d0f-44d5...   256Mi      RWX     ...
webui-loki-claim    Bound    pvc-d97f45c8-b8fc-42df...   512Mi      RWX     ...
```

Pysyvän tallennuksen lisäksi sovellukselle määritettiin valvonta OpenShiftin koetin-ominaisuudella. Koetin (probe) tyyppejä on kaksi, valmius (readiness) ja valvonta (liveness), joista ensimmäinen tarkastelee sovelluksen valmiutta ottaa kutsuja vastaan. Toinen taas sitä kykeneekö se vastaamaan esimerkiksi HTTP pyyntöihin. Jos koetin epäonnistuu, eli sovellus ei vastaa palvelupyyntöön, OpenShift alusta käynnistää kontin uudelleen automaattisesti jotta palvelu palautuisi normaalitilaan. Sovellukselle päätettiin asettaa vain valvonta-tyyppinen koetin koska se tarkastelee myös sovelluksen valmiutta käynnistyksen jälkeen. Valvonta asetettiin komennolla:

```
[sh]$ oc set probe dc/webui --liveness --get-url=http://:9080/ \
     --initial-delay-seconds=100 --timeout-seconds=5
deploymentconfig.apps.openshift.io/webui probes updated
```

Koettimen toimintaa testattiin asettamalla kontissa oleva WLP palvelin pause-tilaan, jolloin se ei enää vastannut HTTP pyyntöihin. Podin tapahtumista (events) voitiin havaita koettimen aktivointi ja kontin uudelleenkäynnistys:

```
[sh]$ oc describe pod/webui-21-dlbf7 | grep -A10 Events:
Events:
  Type     Reason    Age    Message
  ----     ------    ----   -------
  Warning  Unhealthy 2m23s  Liveness probe failed: Get http:// 10.130.4.75:9...
  Normal   Killing   2m23s  Container webui failed liveness probe, will be r...
  Normal   Pulling   2m18s  Pulling image "nexus.repo.fi:18444/webui@sha256:...
  Normal   Pulled    2m18s Successfully pulled image "nexus.repo.fi:18444/we...
  Normal   Created   2m18s Created container webui
```

Edellä kuvattujen toimenpiteiden jälkeen sovellus oli provisoitu ja konfiguroitu onnistuneesti OpenShift klusteriin.

6 YHTEENVETO JA POHDINTA

Tutkimuksessa selvitettiin kirjallisuuskatsauksen ja tapaustutkimuksen avulla mitä tulee ottaa huomioon, kun ohjelmistojärjestelmiä siirretään yksityiseen pilveen ja mikropalvelu-arkkitehtuuriin. Lähestymistapa tuki työn tilaajan tarpeita selvittää miten olemassa olevat järjestelmät voitaisiin migroida ja uudet järjestelmät kehittää yksityiseen pilveen ja näin saavuttaa hyötyjä sovellusten ylläpidon ja palvelinresurssien tehokkaan käytön näkökulmista. Tutkimus kosketti, ei ainoastaan järjestelmän viemistä pilveen, mutta myös laajemmin mikropalvelu-arkkitehtuurin periaatteita joita noudattamalla tehdään sovelluksia jotka ovat helposti ylläpidettävissä, skaalattavissa ja ajettavissa joustavasti erilaisissa ajoympäristöissä.

6.1 Mikropalvelu-arkkitehtuuriin siirtyminen

Mikropalvelu-arkkitehtuurista selvitettiin kirjallisuuskatsauksella teknisiä suunnitteluperiaatteita, joita kannattaa ottaa huomioon niin uusien kuin ylläpidossa olevien sovellusten osalta. Katsauksessa käytiin läpi muun muassa 12factor mallia, joka listaa keskeiset suunnitteluperiaatteet joita noudattamalla saavutetaan hyvä mikropalvelu-arkkitehtuuri. Palvelujen jakamisessa mikropalveluiksi kantavana periaatteena oli DDD-malli, jossa koko järjestelmä jaetaan toisistaan mahdollisimman riippumattomiksi kohdealueiksi, joiden suunnittelu on itsenäistä ja muista palveluista eriytettyä. Kohdealueiden väliset sidokset ovat löyhiä ja tavoite on, ettei muutos yhdessä palvelussa riko toisen palvelun toimintaa.

Palvelujen rajapintojen suunnittelussa tutustuttiin tutkimuskirjallisuudessa kuvattuihin parhaisiin käytäntöihin, joista katsauksen perusteella HTTP-pohjainen REST-tekniikka JSON-muodossa oli yleinen suositeltu vaihtoehto. REST tyylistä käytiin läpi Richardsonin kypsyysmalli, joka määrittelee HTTP operaatioiden käyttötavan resurssien manipuloinnissa. Mallin suositusten perusteella päädyttiin, että kannattaa pyrkiä tason 2 tai 3 rajapintoihin jossa hyö-

dynnetään eri HTTP operaatioita resurssien käsittelyyn. Myös rajapintojen versioinnin käytäntöjä tutkittiin. Lähteiden perusteella havaittiin, että se tulisi ottaa huomioon suunnittelussa, jotta järjestelmät voisivat olla paremmin yhteensopivia tukemalla yhtä aikaa useampaa versiota rajapinnasta. Versiointiin löydettiin kaksi yleistä tapaa; joko versionumerointi HTTP otsakkeessa tai palvelun kutsuosoitteessa. Näistä jälkimmäisen etu on helpompi reititys, jos palvelusta on saatavilla yhtä aikaa useampi versio.

Kommunikaatio tyypin valintaa tutkittiin eri käyttötarpeiden kautta. Lähteiden mukaan aikakriittisissä sovelluksissa synkroninen kutsu on suositeltavampi, kun taas asynkroninen toteuttaa paremmin mikropalveluiden löyhän liitoksen periaatteen. Toisaalta asynkroninen kutsu on hankalampi toteuttaa koska se vaatii sanomajonon ja sovelluslogiikkaa epäonnistuneiden kutsujen kompensoinnissa. Kommunikaatio tyypin valinta kannattaa siis tehdä tapauskohtaisesti; yksi-yhteen rajapinnoissa synkroninen kutsu on usein helpompi, yksi-moneen kutsuissa taas asynkroninen on usein tarpeen.

Transaktion hallinnasta tutustuttiin miten hajautetut ja ei-hajautetut transaktiot toimivat eri kommunikaatio tyypeillä. Hajautettu transaktio on yleinen synkronisissa kutsuissa mutta vaatii asynkronisissa tukea sanomavälittäjältä. Vaihtoehtona hajautetulle transaktiolle käytiin läpi Saga suunnittelumallia, joka esittelee kaksi vaihtoehtoa koordinaatioon; koreografian jossa kukin kutsuketjuun osallistuja liipaisee seuraavan kutsun liikkeelle, sekä orkestrointi jossa yksi keskitetty piste koordinoi kutsuketjua. Koreografia tyyli voi toimia hyvin asynkronisten kutsujen kanssa, kun taas orkestrointi on nopea toteuttaa synkronisten kutsujen kanssa.

Mikropalveluarkkitehtuuriin siirtymisestä käytiin läpi tutkimuskirjallisuutta, josta selvitettiin kriteerejä migraatiolle, migraation suunnittelua ja parhaita käytäntöjä teknisestä näkökulmasta. Kriteereissä painottui juurisyiden selvittäminen miksi migraatio halutaan tehdä. Onko järjestelmän ylläpidossa sellaisia vaikeuksia joihin mikropalveluarkkitehtuuri toisi helpotusta, vai voisiko ongelmat ratketa esimerkiksi parantamalla jatkuvaa integraatiota. Migraation suunnittelussa painottui komponenttien valinta niin, että migroidaan ensimmäisenä ne joista saavutetaan eniten hyötyä pienimmällä työmäärällä. Jos sovellukseen tulee jatkuvasti muutoksia, siinä on laatu ongelmia ja se on kohtuullisen helposti irrotettavissa omaksi palvelukseen, on se hyvä kandidaatti migraatiolle. Lähteissä painottui myös inkrementaalisuus muutosten tekemisessä; kannattaa lähteä liikkeelle jostain pienestä muutoksesta ja viedä se tuotantoon asti, jotta koestetaan toiminta todenmukaisesti. Vanha palvelu kannattaa pitää rinnalla mukana, jolloin siihen voidaan tarpeen tullessa palata. Peruuttamattomia muutoksia kannattaa välttää.

Migraation suunnittelumalleista tutustuttiin Fowlerin (2004) kuihdutusmalliin jossa ideana on hitaasti, jopa usean vuoden kuluessa hankkiutua eroon vanhasta toteutuksesta. Mallissa muutokset tehdään aina uusiin palveluihin ja jätetään vanhat palvelut entiselleen. Ajan kuluessa vanhan toteutuksen osuus kokonaisuudessa kuihtuu pienemmäksi, kunnes jossain vaiheessa se on niin

pieni, että sen jäljellä oleva osuus voidaan korvata kokonaan uudella toteutuksella.

6.2 Sovelluksen käyttöönotto OpenShiftillä

Tutkimuksen empiirisessä osassa selvitettiin, miten olemassa oleva Java käyttöliittymäsovellus otetaan käyttöön tilaajan OpenShiftillä toteutetussa yksityisessä pilvessä. Aluksi tutustuttiin teknologiakirjallisuuden avulla alustan ominaisuuksiin ja selvitettiin sen keskeiset toimintaperiaatteet. Alusta perustuu Docker ja Kubernetes teknologioille jonka päälle on rakennettu jatkuvan käyttöönoton mahdollistavia työkaluja. Sillä voidaan ajaa millä tahansa teknologialla kehitettyjä sovelluksia jotka saadaan paketoitua OCI standardin mukaisiin kontteihin. Alustassa on käännös- ja koonti-työkalut, joiden avulla sovellukset voidaan koostaa ja provisioida suoraan lähdekoodista, Dockerfile määrityksestä tai valmiista levykuvasta. Alusta kykenee valvomaan ja skaalaamaan sovelluksia automaattisesti ja ottamaan käyttöön valmiita monitorointi, sovellusloki ja muita palveluita jotka on paketoitu Docker levykuviksi. Se mahdollistaa palvelinresurssien tehokkaan käytön jakamalla kapasiteettia dynaamisesti eri järjestelmien välillä, jolloin resurssien kokonaistarvetta voi organisaatiotasolla optimoida helposti.

Alusta ajaa sovellusten kontteja podeissa joihin voi provisioida yhden tai useamman toisistaan riippuvan kontin. Podin resursseja voi säätää koontikonfiguraation avulla manuaalisesti tai määritellä raja-arvot joiden täyttyessä resursseja allokoidaan automaattisesti lisää. Sovellukset voidaan julkaista klusterista ulospäin muuttumattoman IP-osoitteen avulla jolloin palvelukutsut menevät kuormanjakajan kautta tasaisesti sovelluksen instansseille podeissa.

Sovellusten päivitykset klusterissa tapahtuvat automaattisesti, kun ennalta määritellyt resurssit muuttuvat. Esimerkiksi Git koodisäilössä tapahtuva muutos liipaisee sovelluksen kääntämisen, koonnin ja uuden version provisioinnin. Tai muutokset kontin pohjalla olevassa levykuvassa käynnistävät uuden levykuvan koostamisen ja provisioinnin automaattisesti.

Tuoteohjeistuksen pohjalta voidaan sanoa, että alusta tuottaa kattavasti työkaluja sovellusten hallintaan ja provisiointiin klusterissa. Lisäksi alustalla on referenssejä toimivista yksityisen pilven toteutuksista. Sillä voidaan tuottaa tuotantovalmis koonti- ja ajoalusta isonkin organisaation tarpeisiin. Tässä tutkimuksessa ei kuitenkaan koestettu alustan toimintaa vikatilanteista toipumisessa. Huomioitavaa on, että koska referenssitoteutuksen mukaan kaikkia tuotantojärjestelmiä ajetaan yhdessä klusterissa, sen vikaantuminen voisi tarkoittaa kaikkien organisaation palvelujen keskeytystä yhtäaikaisesti. Koko klusterin toipumista vikatilanteissa on siis hyvä selvittää jatkotutkimuksissa.

Migraatiossa tarkasteltiin ensin OpenShiftille siirrettävän sovelluksen arkkitehtuuria ja ylläpidon haasteita. Tarkastelu tehtiin sovelluskehitystiimin työpajassa, jossa arvioitiin koko tuotealueen järjestelmien teknisiä kehitystarpeita. Migroitavassa sovelluksessa tunnistettiin työpajassa tarvetta arkkitehtuurin

yksinkertaistamiselle, vähentämällä monikerrosarkkitehtuurin kerroksia. Koska sovellus on kuitenkin kohtuullisen tuore, ei siinä ollut ikääntymisestä johtuvia koodin uudelleenkirjoitustarpeita. Toisaalta toimivan sovelluksen muuttamiselle ei tässä kohtaa löytynyt myöskään aikaa, joten migraatio päätettiin tehdä uudelleensijoitus-menetelmällä ilman arkkitehtuuri-muutoksia. Migraation tavoitteeksi asetettiin kokemusten saaminen alustasta jota voisi hyödyntää tuotealueen muiden sovellusten pilvimigraatiossa myöhemmin.

Migraatio aloitettiin tarkastelemalla aiemmin tehtyä Dockerfile määritystä, jota pyrittiin optimoimaan niin että sovelluksen levykuvan koko olisi mahdollisimman pieni, jolloin sen ajaminen alustan podissa veisi mahdollisimman vähän muistia. Optimoimalla Docker käännöksen tekemiä kerroksia, saatiin levykuvan koko hieman pienemmäksi. Lisäksi Dockerfile määritykseen lisättiin OpenShift ohjeistuksen mukainen käyttäjä ja sille oikeudet tarvittaviin hakemistoihin. Levykuvan koon pienentäminen edelleen vaatisi jatkotutkimusta, esimerkiksi pohjalla olevan levykuvan sekä WLP sovelluspalvelimen lisäosien osalta. Dockerfile määrityksen tekemisessä havaittiin, että tarvittavien konfiguraatiotiedostojen kokoaminen Docker käännöksen tekemiseksi aiheutti jonkin verran ylimääräistä työtä. Parempi vaihtoehto olisi paketoida konfiguraatio ajettavaan war-pakettiin jolloin sovelluksen lähdekoodi ja konfiguraatio olisi kiinteämmin yhdessä vähentäen koontivirheiden mahdollisuutta.

Saadun levykuvan provisisointi alustaan oli pääsääntöisesti suoraviivaista. Sovelluksen ajo vaati OpenShift projektin luomista ja hieman konfiguraatiota salasanojen ja oikeuksien suhteen. Sen jälkeen sovellus saatiin podiin yhdellä komennolla. Lisäksi sovellus vaati vielä podille annettavan muistin kasvattamista, sillä klusterin oletukset eivät riittäneet WLP sovelluspalvelimen käynnistämiseen. OpenShiftin tarjoaman oc komentorivityökalun avulla voitiin sovellukselle luotuja resursseja ja tapahtumia tarkastella ja havaita muistiongelma.

Sovelluksen julkaisu klusterin ulkopuolelle tehtiin luomalla reitti, joka tuotti kiinteän IP osoitteen jonka kautta palvelukutsut ohjautuvat kuormanjakajalle. Palvelu skaalattiin useammaksi instanssiksi, jotta voitiin koestaa tuotantovaatimuksia vastaava konfiguratio. Järjestelmän podien alas ajamisella testattiin alustan kykyä nostaa kaatunut sovellus takaisin ylös. Virhetilanteita simuloitiin ajamalla kontin pääprosessi alas sekä laittamalla sovelluspalvelin tauko-tilaan. Jälkimmäisen tilanteen havainnointiin tehtiin valvontakoetin, joka tarkkaili palvelun kykyä vastata HTTP-pyyntöihin. Molemmissa virhetilanteissa alusta toimi katkotta niin, että kunnossa oleva pod vastasi pyyntöihin, sillä välin kun kaatunutta podia uudelleen käynnistettiin. Lisäksi järjestelmän sovelluksille tehtiin pysyvät tallennuspaikat klusterissa niin, että niiden sovelluslokit säilyivät myös podin uudelleenkäynnistyksen jälkeen.

Näiden toimenpiteiden jälkeen järjestelmä toimi oikein OpenShiftin testiympäristössä. Tuotantoympäristöön asentamista ei tehty vielä aikataulusyistä, joten tämän koestuksen tulokset tuotantokäytössä jäivät vielä uupumaan. Myös suorituskykytestit jäivät vielä tekemättä eli jatkotutkimukselle on tarvetta ennen kuin järjestelmä voidaan viedä tuotantoon OpenShift alustalla.

Tilaajan pilvimigraatiostrategian kannalta tämän koestuksen lopputulokset ovat positiiviset. Sovelluksen kontitus on suoraviivaista ja Docker ajoalustan yleisyyden vuoksi käyttö riskitöntä. OpenShift alustan käyttö vaatii jonkin verran opettelua sekä tuotannon ylläpitäjiltä, että sovelluskehittäjiltä, mutta kun osaaminen on kunnossa, järjestelmien vienti alustaan on suoraviivaista ja tuotteen prosessit ovat kunnossa. Lisää tutkimusta vaaditaan mikropalvelu migraation osalta. Miten olemassa olevat sovellukset kannattaa muuntaa mikropalvelu arkkitehtuuriin. Miten kuristus malli toimii käytännössä, kun mennään kohti mikropalveluita. Miten tietojen integriteetti varmistetaan migraatiota tehdessä, kun palvelujen vastuita muutetaan. Lisäksi aiemmin mainittu alustan toipuminen vikatilanteista klusteritasolla tulee selvittää ja tarvittaessa testata huolellisesti.

LÄHTEET

Alagarason V. (2015). Seven Microservices Anti-patterns. Haettu 21.5.2020 osoitteesta https://www.infoq.com/articles/seven-uservices-antipatterns/

Ambler S.W., Pramod J., Sadalage P. (2011). Refactoring Databases. Addison-Wesley

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

Atchison L. (2018). Preparing to Adopt the Cloud: A 10-Step Cloud Migration Checklist. New Relic. Haettu 2.2.2020 osoitteesta https://blog.newrelic.com/engineering/cloud-migration-checklist/

Conway M. (1968). How Do Committees Invent. Datamation magazine, April 1968

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

Fitzsimons P., Carlson B., Potter B. (2019). AWS Well Architected Framework White Paper. Haettu 23.2.2020 osoitteesta https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

Coleman M. (2016). Containers and VMs Together. Haettu 9.4.2020 osoitteesta https://www.docker.com/blog/containers-and-vms-together/

Cisco Systems (2010). Planning the Migration of Enterprise Applications to the Cloud. Haettu 1.2.2020 https://www.cisco.com/en/US/services/ps2961/ps10364/ps10370/ps11104/Migration_of_Enterprise_Apps_to_Cloud_White_Paper.pdf

Compton K., Cisco Systems (2018). Cisco's Global Cloud Index Study: Acceleration of the Multicloud Era. Haettu 28.2.2020 osoitteesta https://blogs.cisco.com/news/acceleration-of-multicloud-era

Docker Documentation (2020). Best practises for writing Dockerfiles. Haettu 8.6.2020 osoitteesta https://docs.docker.com/develop/develop-images/dockerfile_best-practices

Dumpleton G. (2018). Deploying to Openshift. Sebastopol: O'Reilly Media Inc.

Duncan J., Osborne J. (2018). OpenShift in Action. Manning Publications

Erl T., Mahmood Z., Puttini R. (2013) Cloud Computing: Concepts, Technology & Architecture. New Jersey: Prentice Hall

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

Fowler M. (2004). StranglerFigApplication. Haettu 5.6.2020 osoitteesta https://martinfowler.com/bliki/StranglerFigApplication.html

Fowler M. (2015). Presentation Domain Data Layering. Haettu 5.5.2020 osoitteesta https://martinfowler.com/bliki/PresentationDomainDataLayering.html

Dehghani Z. (2018). How to break a Monolith into Microservices. Haettu 5.2.2020 osoitteesta https://martinfowler.com/articles/break-monolith-into-microservices.html

Jadeja Y., Modi K. (2012). Cloud computing - concepts, architecture and challenges. 2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET), Kumaracoil, pp. 877-880.

Jamshidi P., Ahmad A., Pahl C. (2013) Cloud Migration Research: A Systematic Review. IEEE Transactions on Cloud Computing, vol. 1, no. 2, pp. 142-157, July-December 2013.

Jamshidi P., Pahl C., Mendonça N.C., Lewis J., Tilkov S. (2018). Microservices: The Journey So Far and Challenges Ahead. IEEE Software. 35 (3): 24–35.

Krutchen P. (1995). Architectural Blueprints—The '4+1' View Model of Software Architecture. IEEE Software 12.Nov 1995.

Larsson M. (2019). Hands-On Microservices with Spring Boot and Spring Cloud. Packt Publishing

Laszewski T., Arora K., Farr E., Zonooz P. (2018). Cloud Native Architectures: Design High-availability and Cost-effective Applications for the Cloud. Birmingham: Packt Publishing Ltd.

Longbottom C. (2017). The Evolution of Cloud Computing: How to plan for change. Swindon: BCS Learning & Development Limited

Lossent A., Rodriguez Peon A., Wagner A. (2017). PaaS for web applications with OpenShift Origin. IOP Conf. Series: Journal of Physics: Conf. Series 898

Lukša M. (2018) Kubernetes in Action. New York: Manning Publications

Ahmad N., Naveed Q., Hoda N. (2018.) Strategy and procedures for Migration to the Cloud Computing, 2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS), Bangkok, Thailand, pp. 1-5.

MacCormack A., Rusnak J., Baldwin C. (2007). Exploring the Duality Between Product and Organizational Architectures. Harvard Business School.

Newman S. (2015). Building Microservices. Sebastopol: O'Reilly Media

Newman S. (2019). Monolith to Microservices. Sebastopol: O'Reilly Media

Nickl T., Chintalapudi R. (2018). 8 common reasons why enterprises migrate to the cloud. Google Cloud. Haettu 28.2.2020 osoitteesta https://cloud.google.com/blog/products/storage-data-transfer/8-common-reasons-why-enterprises-migrate-to-the-cloud

OpenShift. (2020). Creating images. Haettu 8.6.2020 osoitteesta https://docs.openshift.com/container-platform/4.4/openshift_images/create-images.html

OpenShift. (2019) Pod Autoscaling. Haettu 30.10.2020 osoitteesta https://docs.openshift.com/container-platform/3.9/dev_guide/pod_autoscaling.html

Picozzi S., Hepburn M., O'Connor N. (2017). DevOps with OpenShift. Sebastopol: O'Reilly Media

Plankers B. (2012) Building a Private Cloud in 10 Steps. Haettu 29.3.2020 osoitteesta http://docs.media.bitpipe.com/io_10x/io_104804/item_544777/Handbook_BuildingaPrivateCloudin10steps_final.pdf?apsws.customField=607cb08e2065a22c772066b0c00c6164

Rafaels R.J. (2015). Cloud Computing: From Beginning to End. CreateSpace Independent Publishing Platform

Richardson L. (2010). Richardson Maturity Model: Steps toward the glory of REST. Haettu 21.5.2020 osoitteesta http://martinfowler.com/articles/richardsonMaturityModel.html

Richardson C. (2018). Microservices Patterns. New York: Manning Publications

Richardson C. (2019). A pattern language for microservices. Haettu 17.5.2020 osoitteesta https://microservices.io/patterns/index.html

Sadeghianfar S. (2016). CI and CD With OpenShift. Haettu 26.4.2020 osoitteesta https://www.openshift.com/blog/cicd-with-openshift.

Sadeghianfar S. (2018). Kubernetes Ingress vs OpenShift Route. Haettu 24.4.2020 osoitteesta https://www.openshift.com/blog/kubernetes-ingress-vs-openshift-route

Spears J. (2018). A Guide to Migrating Applications to the Cloud. Haettu 19.1.2020 https://blog.netapp.com/a-guide-to-migrating-application-to-the-cloud/

Srinivas J., Venkata K., Reddy S., Qyser M. (2012). Cloud Computing Basics. International Journal of Advanced Research in Computer and Communication Engineering, Vol. 1, Issue 5

Suzuki H., Kageyama H., Juli Y. (2017). Private Cloud Platform Based on Open Source Technology. Fujitsu Scientific and Technical Journal, Vol 53, No. 1

Taibi D., Lenarduzzi V., Pahl C. 2018. Architectural Patterns for Microservices: a Systematic Mapping Study

Wei Y., Blake MB. (2010). Service-oriented computing and cloud computing: Challenges and opportunities. IEEE Internet Computing

Wiggins, A. (2017). The Twelve Factor App. Haettu 20.2.2020 osoitteesta https://12factor.net/

Wright D., Smith D., Bala R., Gill B. (2019). Magic Quadrant for Cloud Infrastructure as a Service, Worldwide. Haettu 13.2.2020 osoitteesta https://www.gartner.com/en/documents/3947472/magic-quadrant-for-cloud-infrastructure-as-a-service-wor