TURUN YLIOPISTO

Datan Mallinnus Mikropalveluissa

Suunnittelun vaiheet siirryttäessä monoliittisestä sovelluksesta mikropalveluarkkitehtuuriin

Tietotekniikan laitos
Ohjelmistotekniikka
Teknillinen tiedekunta
Diplomityö
Juho Laaksonen

Maaliskuu 2024
Turku

Diplomityö – Turun Yliopisto

Ohjelmistotekniikka
Juho Laaksonen
Datan Mallinnus Mikropalveluissa
Ohjaaja: professori Ville Leppänen
60 sivua
Maaliskuu 2024

Edelleen voimakkaana jatkuva digitalisaation kasvaminen on nopeuttanut teknologioiden kehitystä, sekä tuonut uusia innovaatioita informaatioteknologian saralla muiden alojen yritysten ulottuville. Digitalisaation avulla on suuremmalla painoarvolla tuotu oikean maailman prosesseja ja järjestelmiä osaksi digitaalista maailmaa, pyrkimyksenä helpottaa käyttäjien, kuluttajien ja toimijoiden arkea sähköisillä työkaluilla. Tämän lisäksi vanhat suuremmat ns. legacy-järjestelmät alkavat tulla elinkaarensa päähän ja uusien järjestelmien lisäksi myös vanhojen sovellusten arkkitehtuuri täytyy päivittää tukemisongelmien takia uudempiin toteutuksiin.

Digitalisaatio ei kuitenkaan aina vastaa odotuksia ja kohtuullisen usein eri viestintäkanavista välitetään tietoa erilaisten IT-projektien myöhästymisestä, kustannuksien noususta tai jopa epäonnistumisista. Syitä epäonnistumisille voidaan antaa vaikka millä mitalla. Keskeisimpiä syitä voivat olla muun muassa suunnittelun ja resurssien puute, sekä osaamisen vaihtelevuus projektihenkilöiden kesken, mutta yleensä epäonnistumiset syntyvät näiden eri painoisesta summasta. Suunnittelun merkitys korostuu ongelmien ennaltaehkäisyssä, koska suunnittelulla voidaan etukäteen koittaa puuttua projektin epäkohtiin.

Tämän työn aiheena on selvittää mahdollisuuksia datamallien suunnittelun tehostamisesta mikropalveluarkkitehtuuria varten, kun pyritään päivittämään vanhasta järjestelmästä modernimpaan muotoon. Mikropalveluarkkitehtuurilla on niin etuja, kuin myös haasteita verrattuna klassisempaan yhden ison kokonaisuuden malliin. Työssä käsitellään mikropalvelumallin taustaa, verrataan sitä ison kokonaisuuden järjestelmään, sekä pyritään löytämään tapoja tai prosesseja, joilla helpotetaan työtaakkaa niin sovellusarkkitehdeille, suunnittelijoille ja asiantuntijoille, jotta kaikki projektihenkilöt saavat kattavamman käsityksen mikropalvelujaosta.

Avainsanat: Mikropalvelu, Arkkitehtuuri, Uudistusprojekti

Master's Thesis – University of Turku

Information and Communication Technology
Juho Laaksonen
Data Modeling in Microservices
Supervisor: professor Ville Leppänen
60 pages
March 2024

Still rapidly continuing growth of digitalisation has sped up the progress of technologies and brought new innovations for the use of the other professional fields by the courtesy of the information technology area. With the help of digitalisation, more real world processes and systems have been converted to be a part of the digital world, as for the aim being to support and enhance the experiences of users, consumers and operators likewise with these digital tools. In addition, the older so called legacy systems are reaching the end of their lifecycles and along with the new upcoming applications, these architechtural decisions of older applications must be updated to ensure avoiding supporting issues.

However, digitalisation doesn't always hold up to expectations and reasonably often different medias relay information on different IT-projects being delayed, the cost of the projects rising, and even in some cases, failure of these projects. Many scenarios can be given as the reasons for these failures. The most essential reasons can be for example, the lack of planning and resources or even the varying amount of experience between the project personnel. Usually, the reason seems to not be one singular subject, but instead, the sum of many issues, where emphasis is larger in some areas than others. Significance of planning is highlighted in preventing problems arising, because with planning it is possible to preemptively intervene in to the possible upcoming issues of the project.

The subject of this thesis is to investigate chances on how to optimize the planning of the datamodels when microservices are the chosen architechture model, when striving for updating an older system to modernized version. Microservices architecture has its own benefits, but also its challenges compared to one singular monolithic model. The work consists of researching the backgrounds of microservices, compares them to monolithic architechture, and tries to find practices and processes, which can be used to ease the workload of software architects, designers and experts. This so, that each person related to the project can have a more comprehensive outlook for the system divided into microservices.

Keywords: Microservices, Architecture, Overhaul Project

Alkusanat

Haluan kiittää tämän työn valmistumisesta perheenjäseniäni, jotka ovat aina olleet tukenani ja mahdollistaneet opiskelun kannustamalla ja suhtautumalla ymmärryksellä valintoihini, sekä auttaneet tavallisissa arkielämän haasteissa.

Tämän lisäksi haluan kiittää ystäviäni motivaation ylläpitämisestä ja johdattamisesta ylipäätään päätökseen hakea opiskelemaan maisterintutkintoa.

Ja lopuksi vielä kaikkia muita henkilöitä, joita ilman työn suorittaminen ei olisi ollut mahdollista, kuten hyviä työkavereita, esihenkilöitä ja ohjaajaa. Kiitos kaikille!

Maaliskuu 2024 – Juho Laaksonen

Sisällysluettelo

1 JOHDANTO

Jokaisella sovelluksella on oma elinkaarensa. Erilaisten sovelluksien elinkaaret vaihtelevat käyttötarkoituksiensa mukaan, jolloin elinkaari voi olla lyhyt tai pitkä riippuen tarpeesta käyttää sovellusta. Esimerkkinä yksinkertaisesta sovelluksesta voi pitää ohjelmoijan työtehtäviään varten luomaa skriptiä, eli joukkoa komentoja, jolla voidaan esimerkiksi automatisoida pienempiä tehtäviä tekemättä niitä manuaalisesti yksitellen (Lennes. 2004). Tällaisen tietokoneohjelman elinkaari voi olla erittäin lyhyt, mikäli parempi työkalu on jo saatavilla ja aika skriptin kehittämiseen on käytännössä mitätön verrattuna muihin työtehtäviin.

Vastaavasti suurten ohjelmistokokonaisuuksien elinkaari voi olla hyvinkin pitkä. Erilaiset teollisuuden prosesseja ohjaavat ohjelmistot voivat pyöriä vanhoilla alustoilla, sekä suuret asiakas- ja toiminnallisuudenhallintajärjestelmät ovat käytössä monia vuosia, jopa vuosikymmeniä. Vaikka mahdollisimman pitkään käytössä pysyvät järjestelmät ovat myös hyvä asia, niin pitkät elinkaaret aiheuttavat myös omat ongelmansa. Teknologioiden kehittyessä vanhempia järjestelmiä ja sovelluksia ajetaan alas niiden tuen ja ylläpidon ohella, jonka takia näistä riippuvien ohjelmistojärjestelmien vanhentuneet osat voivat tuottaa päänvaivaa ylläpidolle, sekä uusille kehitysideoille. Myös niin sanottu tekninen velka voi vuosien saatossa kasaantua niin suureksi, että ylläpito- ja kehitystyölle asettuvat tiukat rajat mitä kehittäjät eivät pysty purkamaan.

Kun yllä kuvatut asiat alkavat kasaantumaan, on hyvä lähteä pohtimaan mahdollista järjestelmäuudistusta. Mikäli vanhalle olemassa olevalle järjestelmälle tekohengityksen antaminen alkaa käydä raskaaksi ja kustannustehottomaksi, on hyvä lähteä suunnittelemaan vaihtoehtoja, millä toimintaa pystytään jatkamaan myös tulevaisuudessa. Vaihtoehtoina ovat olemassa olevien ratkaisujen kartoittaminen tai yrityksen tarpeiden mukaan mahdollinen ohjelmistokokonaisuuden päivittäminen nykyaikaan. Järjestelmien kokonaisuudistuksessa kannattaa kartoittaa mahdollisuus vanhan järjestelmän teknologisen velan taklaamiseen. Koska vastaavat suuret projektit tapahtuvat harvoin ja on mahdollisuus käytännössä aloittaa puhtaalta pöydältä ohjelmistokokonaisuuden rakentamisessa, on suotavaa, että ei toisteta samoja virheitä vaan arvioidaan parannuskohteita, sekä tehdään tällä tavalla järjestelmästä tehokkaampi ylläpitää ja käyttää myös tulevaisuudessa.

Tämä työ käsittelee järjestelmäuudistuksen ja sen datamallien suunnittelua, kun halutaan uudistuksen yhteydessä vaihtaa ohjelmistoarkkitehtuuria, monoliittisestä sovelluksesta mikropalveluarkkitehtuurimalliin. Ohjelmistoarkkitehtuurimalleja on useita, mutta käyttäjälle voivat erilaiset sovellukset näyttäytyä yhtenäisenä isona kokonaisuutena. Tämä työ pyrkii avaamaan eroja monoliittisen mallin ja mikropalvelumallin välillä, kuinka jo olemassa olevia datamalleja voi hyödyntää suunnittelussa uutta järjestelmää varten, sekä selvittämään mahdollisuutta jalostaa prosessi datamallien suunnitteluun tai vastaavasti tuottaa askeleet suunnittelun helpottamiseen jatkossa samanlaisia projekteja varten. Työ pyrkii vastaamaan seuraaviin tutkimuskysymyksiin:

- **TK1: Milloin vaihtaminen mikropalvelumalliarkkitehtuuriin on paras vaihtoehto vanhan monoliittisen järjestelmän uudistusprojektissa?**
- **TK2: Miten suunnitella uudistustyön prosessi vaihdettaessa monoliittisesta mallista mikropalvelumalliin?**
- **TK3: Millä prosesseilla sekä toimenpiteillä saadaan olemassa olevat datamallit muutettua mikropalveluarkkitehtuuria tukeviksi?**

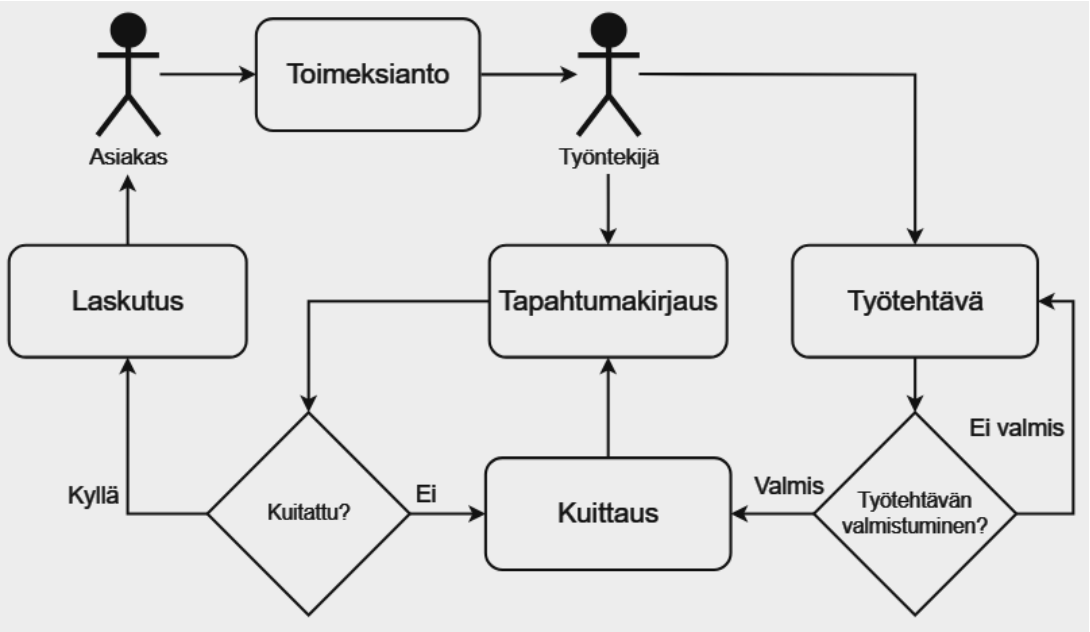
Vaikka jokaisen ohjelman käyttäjän ei tarvitsekaan olla tietoinen siitä, miten ohjelmistokokonaisuudet käytännössä rakentuvat, on silti etu projektin kannalta jakaa tietoa myös muille sidosryhmille kuin kehittäjille. Tämä korostuu projekteissa, jossa järjestelmää käyttävä yritys ei itse esimerkiksi ole suoraan vastuussa järjestelmän ylläpidosta. Yrityksen on hyvä olla tietoinen omista järjestelmäympäristöistään ja niiden toiminnasta, esimerkiksi siihen liittyvistä mahdollisista rajoitteista, kun tehdään suunnittelutyötä, sekä tiedon ja dokumentaation määrä helpottaa pahimmassa tapauksessa ratkomaan järjestelmään liittyviä ongelmia tulevaisuuden tilanteissa.

2 DATAMALLIT

2.1 Datan mallinnus ja datamallit

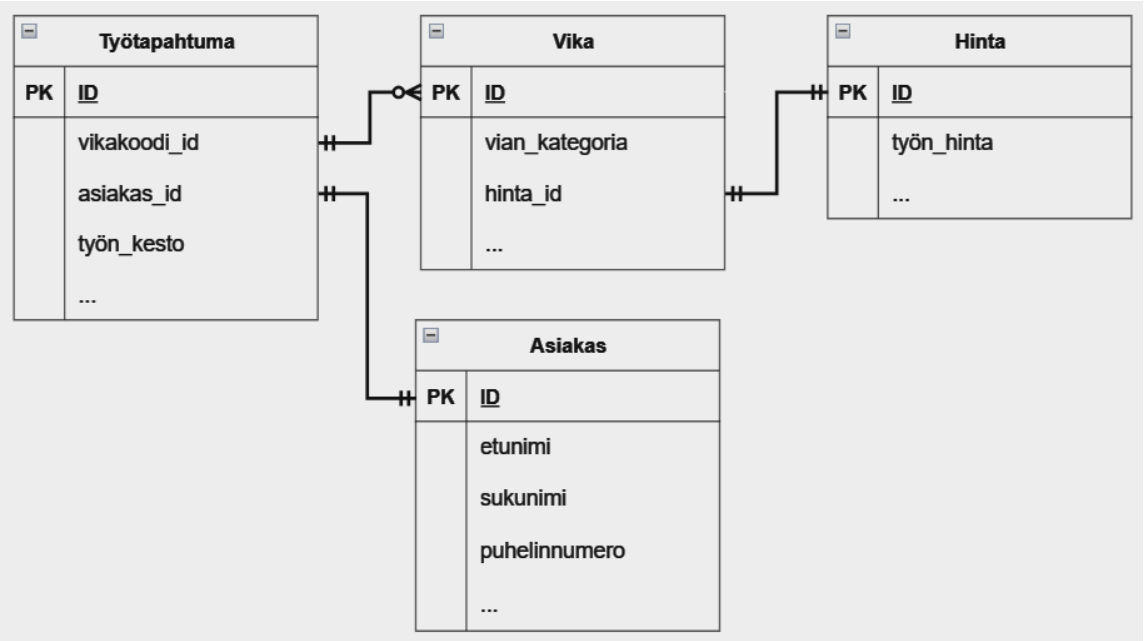
Datan mallinnus ja datamallit ovat tärkeä kulmakivi sovellusten toiminnassa. Käytännössä datan mallinnus tarkoittaa prosessia, jossa luodaan kuvaajia datan suhteista järjestelmän sisällä. Tarkemmin kuvattuna, mallintaja selvittää atomiset elementit ja niiden yhteydet keskenään, jotta niitä pystytään esittämään myöhemmin. Datan mallinnuskaaviot ovat hyvä tapa dokumentoida, mitä järjestelmän sisällä tapahtuu, kun tiettyä tallennettua tietoa muokataan ja mihin muutos vaikuttaa isommassa kuvassa. Mallintamisesta syntyvällä dokumentilla halutaan päästä tilanteeseen, jossa tallennettavia perusdataelementtejä voidaan käyttää sovelluksen tapahtumien kuvaamiseen, kuten uusien tietueiden luomiseen. Sovelluksien toiminnallisuudet yleensä muokkaavat tietokantoihin tallennettua dataa sovelluksen eri osissa. Jos kyseisten muokkausoperaatioiden tekijä on oikean maailman henkilö, on operaatioiden hyvä olla erotettu käyttäjien suoraan käyttämästä käyttöliittymästä. Yleinen käytäntö on välittää muokkauspyyntö palvelimelle rajapinnan kautta. (Allen & Terry. 2008. Luku 1, s. 7–8.)

Mallinnuskaavioiden tekemiseen on tarjolla monia eri vaihtoehtoja mallinnuskielien tarjonnasta. Mallinnuskielen määritelmänä toimii mikä tahansa keinotekoinen kieli, joka kuvaa tietoa, informaatiota tai järjestelmää, joka toimii tiettyjen sääntöjen mukaan. Periaatteessa mallinnuskielien joukosta, jotka jakautuvat kaavioihin (engl. graphical types), tekstipohjaisiin (engl. textual types), sekä erikoistyyppeihin (engl. specific types) voi valita myös muita tyyppejä mallinnettaessa, mutta alan standardiksi on yleisesti vakiintunut UML-malli (Unified Modeling Language), joka on graafinen tapa visualisoida järjestelmän suunniteltua kokonaisuutta tai sen osaa (Satklichov, A. 2008., Amit. 2018). Kuvassa 1 on yksinkertaistettu esimerkki UML:n käytöstä prosessin hallinnassa. Kuvassa on esitetty asiakas, joka tuo laitteensa huoltoon. Vastaava työntekijä hoitaa tapauksen ja käyttää toiminnanohjausjärjestelmää tapahtuman tietojen tallentamiseen, minkä takia tapahtumasta syntyy jälkikäteen lasku asiakkaalle.



Kuva 1. Yksinkertaistettu UML-prosessikaavio

UML:n käyttöä voidaan johtaa eteenpäin esimerkiksi visualisoimalla järjestelmän tietokannan rakennetta entiteettien suhteita kuvaavalla mallilla, eli ER-kaaviolla (engl. Entity Relationship diagram). Kuvassa 2 tuodaan supistetusti esille, mitä dataa ja sen välisiä suhteita tarvitaan kuvan 1 laskutustapahtuman muodostamiseen. Työtapahtumaan on tässä tapauksessa linkitetty asiakkaan tiedot, jotta tiedetään laskun vastaanottaja ja millainen vika on ollut kyseessä, johon hinta pohjautuu. Käytännössä vikoja voi olla useita tai ei yhtään per työtapahtuma, jolloin standardoitu merkistö kuvaa heti kaavion käyttäjälle yhteyden entiteettien välillä. Hyvin pohjustettu suunnittelutyö auttaa ratkaisemaan järjestelmään liittyviä haasteita ja asettaa tietyt säännöt kehitysvaiheeseen. Vaihdettaessa arkkitehtuurimallia uudelleenkehittämisprojektissa, alkuperäiset kaavioit voivat muuttua, mutta mikäli tietorakenteesta on olemassa visuaalinen kuvaus, tekee se mallintajan työtä helpommaksi. (Dobing & Parsons, 2006.)



Kuva 2. Työtapahtuman entiteettien ja suhteiden esittäminen ER-kaaviolla

Itsessään datamalli tarkoittaa rakennetta, jonka muodon mukaan tiedot tallentuvat tietokantaan. Datamalleista voi ymmärtää relaatiosuhteita järjestelmän välillä. Edellä mainituissa muokkausoperaatioissa tietyn datan tallentaminen voi aiheuttaa pitkän tapahtumasarjan, missä muokattu tieto laukaisee joukon toimintoja, mitkä vaikuttavat järjestelmän muihin tietueisiin. Nämä tietoa muokkaavat tapahtumat voivat olla synkronisia tai epäsynkronisia, eli toisin sanoen järjestelmän pitää osata hallita tapahtumia, jotka tapahtuvat saman- tai eriaikaisesti. Mikäli järjestelmässä sattuva virhe aiheuttaa ongelmia tallennettavaan dataan, niin tämänkaltaisessa tapauksessa datamalli antaa järjestelmän hallinnoijalle tietoa siitä, mihin kaikkeen virhe on voinut vaikuttaa ja se helpottaa mahdollisten virhetilanteiden korjausta.

2.2 Datamallien tärkeys

Miksi datamallit sitten ovat sovelluskehityksen kulmakivi? Oli kyse kokonaan uuden järjestelmän luomisesta tai vanhan järjestelmän uudistamisesta, niin näissä tapauksissa datamalli toimii konkreettisena suunnitelmana, miten järjestelmä käyttäytyy. Tavalliselle käyttäjälle järjestelmän toiminnot voivat vaikuttaa siltä, että syöttämällä asioita erilaisiin kenttiin kaikkitietävä entiteetti pystyy päättelemään mitä käyttäjä haluaa saada aikaiseksi, mutta totuudessa nämä toiminnot voivat olla enemmän tai vähemmän kompleksisia käsittelysääntöjä, joiden pohjalla datamalli vaikuttaa. Malli siis antaa selvät rajat sille, mitä sovellusarkkitehtien ja kehittäjien suunnittelemat säännöt pystyvät toteuttamaan, missä

muodossa data käsitellään, sekä tallennetaan. Datamalli määrää miten tietokantarakenne toimii ja se auttaa järjestelmän kehittäjiä pysymään tietyllä uralla, jolloin taataan datan eheys ja järjestelmän virheettömyys.

Datamallin ymmärtäminen ei auta pelkästään sovelluskehittäjiä, vaan sillä voidaan vahvistaa myös sovelluksen käyttäjien osaamista, sekä mahdollisesti johtaa päätöksiä ja auttaa oikean maailman suunnittelussa. Tämän takia hyvä mallintaja ei pyri tekemään mallia pelkästään itseään varten, vaan pystyy myös huomioimaan asiakkaansa tarpeet kehittävien osapuolien ohella. Mitä paremmin on saatavilla informaatiota, kuvauksia, sekä selvityksiä järjestelmän tietorakenteesta, sitä paremmin projektien eri toimijat pystyvät vaikuttamaan ja antamaan päätöksiä järjestelmän kehityksestä (Allen & Terry. 2008. S. 16). Datamallit pyrkivät kääntämään oikean maailman prosessit tallennettavaan ja toistettavaan muotoon. Yksittäinen dataolio ei välttämättä kerro kovin paljon oikeassa maailmassa tapahtuvista asioista, mutta kasvussa olevat data-analytiikan keinot voivat käyttää hyödykseen suurta joukkoa tallennettua dataa ja tallennetusta raakadatasta voidaan mahdollisesti ennustaa käyttäytymistä, luoda erilaisia kaavioita ja löytää järjestelmän kipupisteitä. Analytiikasta saatavia johtopäätöksiä voi käyttää mallin parantamiseen näin parantaa tehokkuutta oikeassa maailmassa (SAP. Päiväämätön). Tosin, on hyvä tiedostaa analytiikan ohella siihen liittyvät rajoitukset. Vaikka datamalli tarjoaakin mahdollisesti erittäin hyvän lähtökohdan analyyttisiin keinoihin, datan määrä (raskasta ja kallista käsitellä), laatu (puuttuvat tiedot hankaloittavat käsittelyä) ja erinäiset ulkopuoliset pakotteet kuten lait ja asetukset (yksityisyys) voivat estää tai esittää esteen analyyttisten keinojen käytölle.

2.3 Tyypit ja prosessit

Datamallityyppejä on useita ja mallin tyyppi käytännössä kertoo minkälaisen rakenteen malli omaa. SAP on verkkoartikkelissaan (SAP. Päiväämätön) kuvannut kolme erilaista tyyppiä: relationaalisen- (relational), dimensionaalisen- (dimensional), sekä entiteetti-rikkaanmallin (entity-rich).

Luultavasti helpoin ymmärtää ja usein käytetty relationaalinen malli käyttää hyväksi tietokantataulujen olioiden keskinäisiä yhteyksiä. Data on jaettuna tietokannassa tauluihin ja jokaisella taulun rivillä on oma uniikki tunnisteensa, joka voi muodostua yhdestä tai useammasta kentän arvosta. Tätä kutsutaan ensisijaiseksi avaimeksi (primary key). Tauluja voidaan linkittää tarvittaessa yhteen tai moneen muuhun tietokannan tauluun viiteavaimien (foreiqn key) avulla. Taulut ovat jaettuina riveihin ja sarakkeisiin, jossa rivit pitävät sisällään

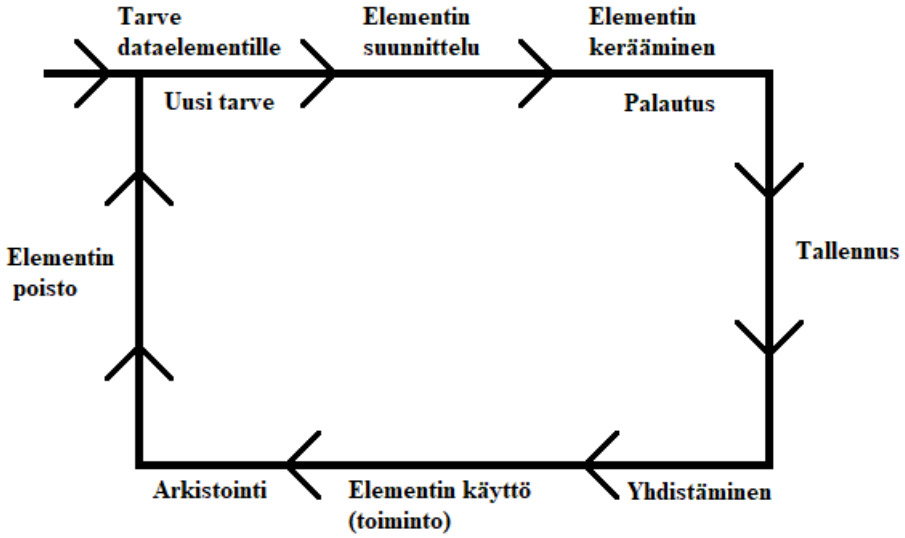
tunnistetiedon ja rivien sarakkeet pitävät sisällään yksittäisiä rivin tietoja eli attribuutteja. Relationaalisen mallin vahvuus piilee datan esittämisen helppoudessa ja standardisoimisessa. Esimerkiksi relaatiotietokannan taulut ovat aina samanlaisia keskenään, jolloin taataan parempi datan käsittely, sekä hakuoperaatiot helpottuvat tietokannassa. Vaikka datan hakemiseen käytetäänkin kyselyitä (SQL), relaatiot eli suhteet mahdollistavat sen, että tulkitsemalla suhteita taulujen välillä voidaan muodostaa loogisia kokonaisuuksia ja nähdä miten kokonaisuus muodostuu erilaisesta datasta. Tällöin on mahdollista myös ihmissilmällä tarkastella tauluja ja nähdä taulujen yhteyksiä ja mahdollisesti tulkita sovelluksen logiikkaa tietokantatasolta (Oracle. Päiväämätön).

Mallien tarkkuutta voi parantaa lähestymällä mallinnettaessa dataelementtien elinkaarta. Alla oleva yksinkertaistettu kuvaaja (Allen & Terry. 2008. S. 9) esittää toistuvaa kiertoa, jossa määriteltyyn asiayhteyteen tarvitaan dataelementti. Tarpeen vaatiessa mallintaja kysyy kysymyksiä, kuten miksi dataa tarvitaan ja ketä tarvitsee kyseistä elementtiä. Käytännössä elementin täytyy tuoda jotain arvoa järjestelmän ja käyttäjän tarpeisiin tai elementti on turha. Seuraavaksi voi mallintaja keskittyä yksityiskohtiin, kuten mitä tallennetaan, minkälaisia rajoituksia tarvitaan ja millaisia yhteyksiä muihin elementteihin löytyy. Kerätessä tarkastetaan, toimiiko suunniteltu elementti, tarvitaanko lisätietoja tai epäonnistuuko elementin tarkoitus käytännössä, jolloin tarvitaan eri lähestymistapa.

Seuraavissa vaiheissa määritetään, miten data tallennetaan, miten sitä voidaan käyttää jatkossa, sekä mitä elinkaaren päässä tapahtuu. Dataelementeille voi syntyä uusia käyttötarkoituksia ja on hyvä käytäntö uusiokäyttää elementtejä, jolloin voidaan myös karsia mallin kompleksisuutta. Uusiokäyttämällä elementtejä järjestelmien eri osissa taataan, että malli ei pirstaloidu sekä datan eheys voi olla helpompi säilyttää. Tähän liittyen haasteeksi voi nousta se, että mikäli järjestelmä voi muokata samaa dataelementtiä monissa paikoissa, voi olla hankala hahmottaa mitkä kaikki toiminnallisuudet käyttävät kyseistä elementtiä. Hyvin suunniteltu elementti ja hyvä järjestelmätuntemus rajoittavat syntyviä laittomia tilanteita.

Lopuksi on tarpeellista hoitaa datan jälkikäsittely. Se miten data arkistoidaan tai hävitetään voi vaikuttaa hyvinkin paljon esimerkiksi palveluhintoihin ja asetuksien täyteen panoon. Riippuen järjestelmästä, on erittäin todennäköistä, että dataelementtejä halutaan säilyttää pitkiäkin aikoja, jopa vuosikymmeniä. Esimerkiksi Suomessa mm. julkisien palveluiden järjestelmät kuten Omakanta säilyttävät käyttäjien terveystietoja ja niitä on tarvittaessa pystyttävä selaamaan. Nykyään tallennustilan ulkoistaminen pilvipalvelujen muodossa on

hyvinkin ajankohtaista ja palveluntarjoajilla on eri vaihtoehtoja datan arkistointiin. Molemmilla, Amazonin AWS:llä ja Microsoftin Azurella, on arkistointitasoja kylmä-kuuma-akselilla (hot, cold, yms.), jossa kylmälle tasolle tarkoitettua dataa ja elementtejä ei ole tarkoitus muokata usein. Kun mennään akselilla eteenpäin, niin tiheämmin tallennettua dataa halutaan pyytää ja muokata. Arkistointitasoilla on omat hintansa, missä kylmäksi arkistoidut tiedot ovat halvempia säilyttää ja "vaikeammin" eli kalliimmalla hinnalla saatavilla, mutta tallessa (Microsoft. 2023. & Amazon. 2021). Toinen vaihtoehto on poistaa dataelementit käytöstä, kun ne tulevat elinkaarensa päähän, mutta kuten aiemmin mainittu, harvoin nykyjärjestelmissä on sellaista dataa, mistä ei haluttaisi jättää jonkinlaista leimaa tai historiatietoa. Poistettavia tietoja ovat ehkä lähinnä henkilökohtainen data. Euroopan unionin yleinen tietosuoja-asetus (GDPR) pyrkii takaamaan Euroopan unionin jäsenmaiden kansalaisten oikeuden pyytää henkilökohtaisen datan poistoa ja hallitsemaan omia henkilökohtaisia tietojaan. Nämä kaksi jälkikäsittelyn aluetta vaikuttavat merkittävästi dataelementtien suunnitteluun ja niiden huomiotta jättäminen voi kasvattaa kustannuksia palvelumaksujen tai jopa sakkojen muodossa.



Kuva 3. Datan elinkaari (Allen & Terry. 2008. Kuva 1–1)

Toinen esimerkki tyypeistä on datan dimensionaalinen mallinnus, jonka pääkäyttötarkoitus on optimoida tiedon nopea haku. Tekniikka itsessään käyttää hyväksi faktoja (Facts) sekä ulottuvuuksia (Dimensions). Faktat ovat kerättyä dataa, joka on yleensä liiketoimintalähtöistä. Esimerkkinä data voi koostua mitoista tai tapahtumista. Ulottuvuudet ovat kokonaisuuksia,

jotka sisältävät liiketoiminnan mittareita, käytännössä ulottuvuus kertoo mitä ominaisuuksia faktoihin liittyy. (Agrawal. 2023).

3 SUUNNITTELUMALLIT

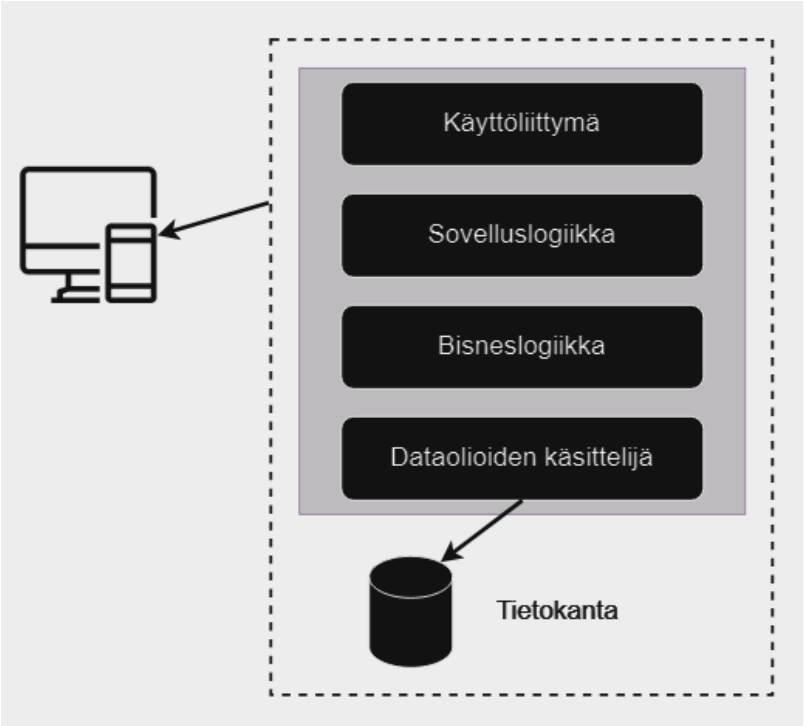
Tässä luvussa käsitellään verrattavien mallien toimintaa, tarkoituksena kuvata mallien yksityiskohtia, minkälaisissa tilanteissa niitä voidaan hyödyntää, sekä tuoda esille mallien hyvät ja huonot puolet. Lukijan kannattaa huomioida, että aiheesta on runsaasti kirjallisuutta esimerkiksi erilaisten artikkeleiden ja julkaisujen muodossa (Villamizar & ym., 2015. Richardson, 2024. Nadareishvili, 2016. Google Cloud, Päiväämätön.), joten luku toimii koosteena useasta eri lähteestä ja pyrkii kokoamaan kirjoitetun materiaalin yhteneväisyydet. Tällä tavalla halutaan vastata TK1:een eli milloin vaihtaminen mikropalveluarkkitehtuuriin on paras vaihtoehto vanhan monoliittisen järjestelmän uudistusprojektissa.

3.1 Monoliittinen malli

Aloitetaan siis kuvaamalla monoliittista mallia. Monoliittinen arkkitehtuuri tarkoittaa järjestelmää, jossa järjestelmä on yleensä yksi suuri kokonaisuus, joka koostuu monesta komponentista. Toisin kuin modulaarisessa järjestelmässä, jossa voidaan ajaa ja rakentaa ohjelmistokomponentit myös tarvittaessa erikseen, monoliittinen järjestelmä tarvitsee toimiakseen kaikkia järjestelmän palasia ja muutos johonkin tiettyyn komponenttiin aiheuttaa suurella todennäköisyydellä muutostarpeen myös muissa järjestelmän osissa. Monoliittisen järjestelmän osat ovat siis erittäin riippuvaisia toisistaan ja järjestelmän päivityksen yhteydessä koko sovellus rakennetaan uudelleen.

Monoliittisen mallin eduiksi voi lyhyesti kuvata seuraavia asioita:

- Nopeampi kehittää järjestelmän elinkaaren alkuvaiheessa
- Yksinkertaisuus
- Pienempi resurssien käyttö
- Testauksen yhtenäistäminen
- Virheiden lokittaminen



Kuva 4. Monoliittisen arkkitehtuurin UML-kaavio

Koska järjestelmä itsessään sisältää käytännössä koko paketin, monoliittinen arkkitehtuuri on yksinkertaisempi ymmärtää. Kaikki lähdekoodit löytyvät samasta paikasta ja tämän takia on helpompi rakentaa sovellusta, joka ei ole pirstaloitunut esimerkiksi eri alustoille. Pienempi resurssien käyttö on suhteellista, koska tarpeeksi iso sovellus tai toiminnallisuus ilman kunnollista optimointia voi kohdata myös resurssiongelmia, mutta käytännössä yhden ohjelman ajaminen voi olla vähemmän intensiivistä kuin monen hajanaisen ohjelman tai kontin ajaminen. Testaus ja virheiden lokitus on myös yksinkertaisempaa, koska testattavat asiat ja mahdolliset ilmentyvät virheet löytyvät samasta lähdekoodipaketista. Jos testaus ja lokitus on järjestetty valmiin kirjaston avulla, niin järjestelmän kaikkiin osiin on helpompi ujuttaa testit ja lokitukset samalla teknologialla. Tämän lisäksi käytetyt teknologiat eivät välttämättä vaihtele monoliittisessa järjestelmässä, jolloin ylläpidon toimenpiteet ovat helpompia ja ylläpitäjien teknologioiden tuntemus ei tarvitse olla niin laaja, kuten arkkitehtuurisessa kokonaisuudessa, jossa käytetään esimerkiksi eri ohjelmointikieliä ja niiden kirjastoja sekaisin. (Awati R. Päiväämätön., Shukla N. 2023)

Monoliittisen arkkitehtuurin haasteet ovat syy miksi monet isot toimijat ovat siirtyneet käyttämään vaihtoehtoisesti mikropalvelumallia. Nykyisin suurilla yrityksillä on käytössä paljon henkilötyövoimaa, jolloin on tärkeää, että projektiryhmät pystyvät tuottamaan ominaisuuksia samaan aikaan. Monoliittisen järjestelmän versionhallinta alkaa hankaloitua

suurilla kehittäjäryhmillä, sekä mitä suuremmaksi sovellus kasvaa, sen vaikeampi on hallita kompleksisuutta, sekä jokaisen julkaisun ohessa järjestelmä täytyy rakentaa uudelleen, mikä vie aikaa, kun halutaan julkaista uusia päivityksiä. Mikäli järjestelmän yksittäinen komponentti kaatuu, se voi lamauttaa koko järjestelmän käytön, koska riippuvuussuhteet estävät muiden komponenttien toimimisen. Myös uusien teknologioiden ja toiminnallisuuksien sisällyttäminen järjestelmään vaikeutuu, koska käytetty teknologia, esim. viitekehys tai ohjelmointikieli rajoittaa vaihtoehtoja toteutukseen. Lopuksi vielä erilaiset skaalautuvuusongelmat ovat merkittävä este monoliittisen mallin kannalta. Mainittakoon, että monoliittista mallia on mahdollista skaalata, se vain ei tapahdu komponenttitasolla, jolloin se vie usein enemmän resursseja ja on kalliimpaa. Skaalautuvuudesta puhuttaessa skaalautuvuus voidaan jakaa kahteen eri määritelmään, vertikaaliseen ja horisontaaliseen skaalautuvuuteen. Vertikaalinen skaalautuvuus tarkoittaa resurssien lisäämistä, esimerkiksi palvelimen laskentatehon kasvatusta ja horisontaalinen skaalautuvuus tarkoittaa työn jakamista monelle eri resurssille. Voi olla, että kahdella eri palvelimella pyörii palvelininstanssi ohjelmasta, jolloin käytetään kuormantasaajaa (engl. load balancer), joka ohjaa pyynnöt tietylle instanssille, missä resursseja on saatavilla (Medhat N. 2021). Tämä tosin ei ole aina kovin kustannustehokasta, koska jos pystytään optimoimaan komponenttien omaa resurssien käyttöä voidaan silloin puuttua tietyn toiminnallisuuden resurssin käyttöön ja skaalata komponentille tarkoitettuja resursseja, jolloin voidaan säästää resursseja ja päästä halvemmalla mahdollisten maksujen suhteen. (Awati R. Päiväämätön., Shukla N. 2023)

3.2 Mikropalvelumalli

3.2.1 Kuvaus

Mikropalvelut (engl. microservices) ovat usein kuvattu pieniksi, itsenäisiksi, sekä löyhästi kytketyiksi ohjelmistoiksi (Microsoft. Päiväämätön). Tosin, vaikka mikropalvelu terminä on vakiintunut viime vuosien aikana yleiseen käyttöön ohjelmistokehityksen saralla, niin mikropalveluiden määritelmästä ei ole täysin tarkkaa yksimielisyyttä. Eri tutkijat ja asiantuntijat ovat julkaisseet teoksia, joissa koitetaan syventää ymmärrystä mikropalveluista ja niiden määritelmästä (Shadija, Rezai & ym., 2017. Microsoft, Päiväämätön. Hilbrich & Lehmann, 2022). Esimerkiksi Shadija ja hänen kollegansa ovat paperissaan huomauttaneet, että “mikro”-sanalla viitataan kirjallisuuslähteissä pieneen kokoon, mutta mikropalveluista puhuttaessa kyse on monitulkintaisesta termistä (Shadija, Rezai & ym. 2017). Tämä luultavasti johtuu siitä, että tämän kaltaisissa järjestelmissä ohjelmistokokonaisuuksien koko

vaihtelee suuresti ja määritelmää pienelle ja suurelle ohjelmistolle on hankala asettaa ilman kontekstia.

Se mistä ohjelmistoalan asiantuntijat ovat usein yhtä mieltä, on se, että mikropalvelut ovat itsenäisiä kokonaisuuksia, jotka on linkitetty löyhästi toisiinsa, ja joiden kommunikaatio tapahtuu viestien välityksellä. Löyhästi kytketyt ohjelmistot ovat lyhyesti järjestelmiä, joiden osat on erotettu toisistaan ja niiden suuri etu on, että riski odottamattomien muutoksien syntymisestä muissa moduuleissa pienentyy, kun tehdään muutoksia yhteen järjestelmän ohjelmistomoduuleista. Esimerkiksi yksittäinen moduuli voidaan kokonaan poistaa, jolloin muiden moduulien toiminta voi silti jatkua normaalisti. Viestintä tapahtuu rajapintojen kautta ja rajapintatoteutus vaihtoehtoja on monia. Nykyisin mm. REST API –toteutukset ovat eniten käytetty teknologiatoteutus rajapintakommunikaatioon, koska ne ovat monipuolisia ja sopivat hyvin verkkosovellusten toteutukseen (Konghq. Päiväämätön). Näiden edellä mainittujen määritelmien ominaisuuksien eri puolista muodostuvat mikropalvelumallien hyvät ja huonot puolet. Seuraavissa luvuissa keskitytään syvemmin mikropalveluarkkitehtuurin ominaisuuksiin.

3.2.2 Ominaisuudet

Mikropalveluarkkitehtuuri on ominaisuuksiltaan varsin joustava. Koska palveluiden on tarkoitus toimia myös erillään toisistaan, tarkoittaa tämä sitä, että on enemmän vaihtoehtoja jokaisen palvelun toteutustavalle. Jotta palvelut ovat itsenäisiä, tarvitsee niiden hoitaa niiden oma tilanhallintansa, joka monoliittisessa palvelussa tapahtuisi yhden yhteisen datan hallinnan kerroksen kautta, jonka tehtävänä on huomioida kaikkien sovelluksen osa-alueiden tila. Microsoft on mikropalveluarkkitehtuuria käsittelevässä dokumentaatiossaan koonnut seuraavan listan mikropalveluarkkitehtuurin hyödyistä:

- Skaalautuvuus
- Datan eristys
- Eri teknologioiden käyttö
- Pienempi määrä lähdekoodia palvelua kohti
- Ketteryys

Skaalautuvuutta edistää palveluiden jakautuminen. Jos yksittäinen palvelu tarvitsee enemmän resursseja toimiakseen, voi mikropalveluarkkitehtuurissa kasvattaa pelkästään sen kokonaisuuden tarvetta (esim. levytilaa tai laskentatehon lisäämistä). Datamalleista

puhuttaessa voi olla tarve muokata mallia tulevaisuudessa uusien ominaisuuksien takia. Mikropalvelumalli mahdollistaa sen, että voi olla helpompaa kehittäjän kannalta lisätä tietomalliin yksittäisiä uusia elementtejä, vaikuttamatta muiden palveluiden toimintaan haitallisesti. Mikäli yksittäiseen mikropalveluun menetettäisiin yhteys, haitallisten tapahtumien syntymistä voi estää myös muiden mikropalveluiden toiminnan jatkuminen kyseisessä tilanteessa, jos on osattu ottaa vastaavat tapaukset huomioon.

Ketteryys ulottuu moniin palvelun kehittämisen vaiheisiin. Koska palvelujen välinen kommunikointi voidaan hoitaa viesteillä, ei ole pakollista turvautua kaikkien palveluiden osalta samoihin teknologioihin vaan jokaisella palvelulla voi olla omanlaisensa toteutus. Esimerkiksi ohjelmien tekemiseen voidaan käyttää eri ohjelmointikieliä ja niiden ohjelmointiparadigmat, sekä -käytännöt voivat poiketa toisistaan. Viestit voidaan tulkita palvelun haluamalla tavalla ja tämä hoituu rajapintatasolla. Jokaiselle palvelulle voidaan allokoida oma kehittäjä- tai ylläpitäjäryhmänsä, jolloin kehittäminen voi tapahtua yhtäaikaisesti ja ryhmiin voi allokoida tietyt henkilöt osaamisen mukaan. Mahdollisesti yksittäinen palvelu sisältää vähemmän lähdekoodia per toteutus, kuin kokonainen monoliittinen sovellus, jolloin ylläpidettävyys ja luettavuus ohjelman osalta on usein helpompaa.

Kompastuskiviä mikropalveluarkkitehtuurissa voi olla muun muassa kokonaisuuden laajuus ja sen hallinta, erilaiset tietoverkko- tai yhteysongelmat liittyen monien viestien lähettämiseen, sekä tallennettavan tiedon laadun varmistaminen. Mikropalvelumallissa toteutus voi paisua ja mitä enemmän mikropalveluja on käytössä, sitä vaikeampi on muodostaa kokonaiskuva järjestelmästä. Mitä enemmän viitekehyksiä tai ohjelmointikieliä on käytössä, sitä enemmän osaamista tarvitaan esimerkiksi ylläpidon osalta ja mikäli näiden määrää ei rajoiteta tai hallita kunnolla, voi lopputulos olla hyvinkin kompleksinen. Koska palvelut käyttävät viestejä kommunikoimiseen, voi viestiliikenne olla runsasta. Palvelut kannattaa mitoittaa sillä perusteella, kuinka usein ne ovat käytössä, koska mitä enemmän viestejä niille generoituu, sen helpommin ne voivat osoittautua pullonkauloiksi, jos ne eivät pysty käsittelemään kaikkea niille ohjattua liikennettä. Tämän lisäksi, jos toteutukset poikkeavat suuresti toisistaan, voi aiheutua ongelmia datan laadun kanssa. Jokaisen palvelun täytyy huomioida, missä muodossa dataa halutaan tallentaa tietokantatasolla, joten mahdollisesti väärän tyyppinen data voi aiheuttaa virheitä järjestelmän toimintaan, jos esimerkiksi rajapinta ei rajoita oikein vietävien datojen tyyppiä. (Microsoft, Päiväämätön. Hillpot J, 2023.)

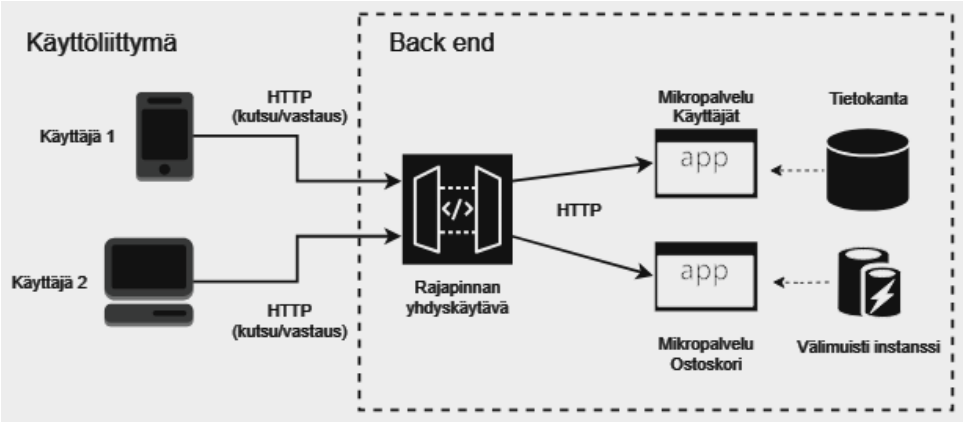
3.2.3 Kommunikaatio

Microsoftin mikropalvelujen kommunikaatiosta kertovan artikkelin mukaan suurin haaste siirryttäessä monoliittisesta mallista mikropalvelumalliin on järjestelmän sisäisen kommunikoinnin järjestäminen (Microsoft, 2022). Koska monoliittisen mallin komponentit ovat tiukasti kytkettyjä toisiinsa, voivat ne käyttää tiedonsiirtoon järjestelmän sisäisiä olioita tai käyttää riippuvuusinjektiota, jossa tuodaan olio tai funktio komponentin käyttöön. Mikropalvelumallissa, jossa mikropalvelut voivat sijaita eri alustoilla kuten palvelimilla tai omilla konteillaan, joten niillä ei ole pääsyä toistensa metodeihin ja tätä lähestymistapaa ei voida käyttää. Kommunikaatio viestintäprotokollien välityksellä ei taivu suoraan monoliittisen palvelun kommunikaatiosta jaetuille järjestelmäkokonaisuuksille, vaan aiheuttaa ongelmia esimerkiksi suorituskyvyn alenemana. Esimerkkinä ongelmasta on ylimääräisten viestien syntyminen, jotka aiheuttavat kuormitusta niitä vastaanottaville palveluille, mikä taas estää tärkeiden viestien käsittelyn ja näin toimintojen suorittamisen. Tämän lisäksi jo vuonna 1994 Peter Deutsch kirjoitti seitsemästä olettamuksesta, johon James Gosling (1997) lisäsi yhden uuden olettamuksen, tietoverkkojen ominaisuuksista, joihin ohjelmistokehittäjät sortuvat. Näille olettamuksille ominaista on, että kehittäjät eivät osaa arvioida kehittämisen aikana tietoverkkojen yli tapahtuvan viestinnän kompastumiskiviä, kuten että viestien kuljettamiseen kuluu aikaa, pystytään kuljettamaan rajaton määrä dataa tai että viestien kuljettaminen verkon yli on salattua automaattisesti (Rotem-Gal-Oz, 2006). Muun muassa näiden haasteiden takia viestinnän toteuttaminen vaatii erityistä huolellisuutta ja arkkitehtuurimallissa ei ole yhtä oikeaa tapaa toteuttaa järjestelmän välistä viestintää, mikä itsessään voi mutkistaa suunnittelua. Näiden ongelmien taklaamiseksi mikropalveluja kehittävät yhteisöt ovat lanseeranneet termin “smart endpoints and dumb pipes”, mikä vapaasti käännettynä tarkoittaa, että mikropalvelut keskittyvät tiedon käsittelyyn ja pitävät huolen omasta viestintälogiikastaan ja viestinnän hoitavat protokollat keskittyvät vain tiedon liikuttamiseen, jolloin niiden toteutus on mahdollisimman pelkistetty.

Erilaisista kommunikaatiovaihtoehdoista on hyvä tietää ovatko käytetyt protokollat synkronisia vai asynkronisia, sekä voiko viestin vastaanottaa vain yksi päätepiste vai monet eri päätepisteet. Näitä vaihtoehtoja voidaan käyttää järjestelmissä sekaisin, missä tavanomaisin lähestymistapa on synkronisen protokollan kuten HTTP:n tai HTTPS:n käyttö yksittäisien viestien välittämiseen, kun kutsutaan rajapintaa. Asynkronisia kutsuja voidaan käyttää muun muassa välittämään monta viestiä samaan aikaan eri mikropalveluille. Käytännössä näiden yksityiskohtainen ymmärtäminen ei ole täysin pakollista, mutta ne

auttavat kehittämään palveluja, jotka pystyvät toimimaan itsenäisesti ja samaan aikaan. (Microsoft, 2022.)

Yhteenvetona kommunikaation järjestämisestä mikropalveluiden välillä voidaan korostaa, että mikropalveluiden välinen kommunikaatio kannattaa koostaa niin, että mikropalvelut kommunikoivat mahdollisimman vähän toistensa kanssa, sekä jos (ja kun) mikropalvelut joutuvat kommunikoimaan toistensa kanssa, niin ne eivät luo pitkiä käsittelyketjuja, sekä toiminnot pystyvät silti toimimaan samanaikaisesti. Oletuksena on, että mikropalvelut toimivat riippumattomina kokonaisuuksina, jolloin pitkät tapahtuma aiheuttavat viivettä järjestelmässä ja luovat näin suunnittelumallien vastaisen toiminnan (anti-pattern). Koska useimmilla järjestelmillä on loppukäyttäjiä varten luotu käyttöliittymä, on standardiksi muodostunut RESTful-pohjaisen ajattelumallin soveltaminen käyttöliittymän ja järjestelmän palveluiden viestinnän välillä, varsinkin verkkosovelluksien kohdalla. REST:in maailmaan kuuluu kutsujen ja vastauksien välittäminen käyttäen HTTP metodeja. Näin käyttöliittymä voi kutsua palveluita ja saada vastauksena tarvitsemansa resurssit, jotka esitetään käyttäjälle. Tämän typpisessä toteutuksessa käyttöliittymä ei tosin kutsu palveluita suoraan, vaan palveluiden ja käyttöliittymän välissä on rajapintayhdyskäytävä (engl. API Gateway), jonka päätepisteeseen käyttöliittymä ottaa yhteyttä ja yhdyskäytävä käsittelee kutsun ja välittää sen oikealle mikropalvelulle, mikä palauttaa vastauksena viestin määritellyssä formaatissa (esim. XML tai JSON) (Microsoft, 2022.). Rajapintayhdyskäytävää ei välttämättä ole pakollista toteuttaa, mutta sen käyttö vaikuttaa olevan vakiintunut ratkaisu mikropalvelutoteutuksilla. Rajapintayhdyskäytävän hyviä puolia ovat mm. yksi tietosuojakerros lisää sovelluksen käyttöön, kutsujen tallentaminen välimuistiin ja kutsujen käytön hallinnointi (Microsoft, 2022. Lenin, 2023. Bugaev, 2023).



Kuva 5. Käyttöliittymä-rajapinta-mikropalvelu-toteutus

3.3 Mallien erot ja käyttötarkoitukset

Kun ajattelee arkkitehtuurisesti monoliittista kokonaisuutta ja sen yksityiskohtia, voi huomata, että myös monoliittisen järjestelmän voi jakaa pienempiin osiin (käyttöliittymä-, palvelin- ja tietokantaratkaisut). Herää kysymys, että jos mikropalvelujärjestelmät arkkitehtuurisesti ovat yksinkertaistettuna järjestelmiä, jotka ovat jaettu pienempiin osiin, niin mikä on monoliittisen arkkitehtuurin ja mikropalveluarkkitehtuurin ero? Esimerkiksi M. Hilbrichin ja F. Lehmannin koottujen mikropalveluiden määritelmien mukaan mikropalvelut arkkitehtuurisena tyylinä ovat järjestelmiä, jotka on jaettu vertikaalisiin osiin syistä, jotka ovat usein liiketoiminnallisia. Lisäksi mikropalvelu on itsenäinen komponentti, joka voidaan ottaa käyttöön automaatiotyökalujen avulla, mutta kuuluu sidottuun kokonaisuuteen, jossa toimivuus on hoidettu viestien välisellä kommunikaatiolla (Hilbrich & Lehmann, 2022). Toisin sanoen mikropalveluarkkitehtuurin voi erottaa monoliittisesta arkkitehtuurista sillä, että järjestelmä koostuu palasista, jotka toimivat itsenäisesti.

Mutta mitä tämänkaltainen rakenne hyödyttää ja eliminoiko se kaikki monoliittisen järjestelmän haasteet? Lisäksi kannattaa huomioida minkälaisia ongelmia mikropalveluarkkitehtuurilla ilmenee itsellään. Vertaamalla näitä ominaisuuksia aiemmin avattuun monoliittisen arkkitehtuurin ominaisuuksiin, jolloin voidaan asettaa tietyt kehykset, milloin kumpikin vaihtoehto on todennäköisesti parempi vaihtoehto.

Voidaan myös tarkastella toimialakohtaisia esimerkkejä ja minkä takia suuret yritykset ovat omaksuneet mikropalveluiden käytön. Tunnetuimpia mikropalveluiden käyttäjiä ovat muun muassa verkkokauppajätit Ebay ja Amazon, suoratoistopalvelu Netflix, sekä kuljetuksen järjestämiseen käytetty alusta Uber. Dreamfactoryn artikkelissa (Hillpot, 2023.) on summattu motivaatiotekijöitä muuttaa sovelluksen arkkitehtuuri monoliittisesta mikropalvelumalliin. Esimerkkejä mikropalvelumallin hyödyistä ovat esimerkiksi asiakaskokemuksen parantuminen mobiili ja työpöytäsovelluksen yhtenäisemmän käyttökokemuksen takia, joka on myös johtanut yleiskustannuksien vähentymiseen, koska samalla kehittäjäpanoksella saadaan enemmän arvoa aikaan molemmille alustoille. Tämän lisäksi ominaisuuksien julkaiseminen käyttäjille on ollut nopeampaa, koska palveluita voidaan kehittää ja julkaista sillä tahdilla, kun ne valmistuvat. Uber ja Netflix siirtyivät käyttämään mikropalvelumallia pääsääntöisesti käyttäjämäärien kasvun takia (Hillpot, 2023.). Koska sovelluksien tarve skaalautua yhä isommalle kuluttajakunnalle aiheutti hankaluuksia monoliittisen mallin

kannalta, pystyttiin mikropalvelumallilla vastaamaan kasvaviin tarpeisiin ja välttämään häiriöajan syntymistä järjestelmään. Amazon on taas hyödyntänyt mikropalvelujaan asiakkaidensa kulutustottumuksien seuraamiseen ja pystynyt lisäämään myyntiä oikeanlaisella kohdentamisella (Nguyen S. Päiväämätön). Muita tunnettuja yrityksiä, jotka ovat uudistaneet toimintaansa mikropalvelujen avulla ovat esimerkiksi Coca-Cola Company (Schwartz S. 2018), Spotify (Kalman R. 2022) ja monet muut.

Monoliittisen mallin etu on selvästi yksinkertaisuus. Koska monoliittisen järjestelmän pystyy melko vaivattomasti aloittamaan tyhjästä ja kehittämään perusominaisuudet saman lähdekoodin alle, toimii se pienemmissä ohjelmistokokonaisuuksissa, jossa palvelua tai järjestelmää kehittävän ryhmän koko on pieni, sekä yksittäisten suurempien ja monien pienempien muutoksien nopealle julkaisutahdille ei ole tarvetta. Myös se, että monoliittisen mallin kohdalla teknologiavaihtoehdot ovat rajatut, yksinkertaistavat vaaditun teknologia- tai alustaosaamisen määrää, jolloin hallinta voi osaltaan olla helpompaa, kuin mikropalvelussa, jossa voidaan käyttää monenlaisia eri vaihtoehtoja. Ongelmat monoliittisten sovellusten kanssa alkavat, mitä suuremmaksi lähdekoodin määrä kasvaa ja mitä enemmän ikää sovellus kartuttaa. Jos järjestelmän elinkaaren aikana on tehty paljon muutoksia, on vaikea välttyä ajamasta itsensä ansaan, jossa pieni muutos aiheuttaa suuria muutosketjuja myös muiden järjestelmän komponenttien sisällä. Tätä ei tosin voi sulkea myöskään täysin pois mikropalvelutoteutuksesta, koska riippuvuussuhteita on aina järjestelmien välillä, mutta mikropalvelutoteutuksen ajatus lähtee pohjalta, jossa riippuvuussuhteet voi minimoida. Myös teknologioiden vanhentuessa on vaikeampi päivittää koko järjestelmää käyttämään viimeisimpiä päivityksiä tai ominaisuuksia. Järjestelmän ylläpitäminen vaikeutuu siis pitkässä juoksussa.

Edellä kuvattu ongelma on yksi suurimmista asioista, mihin vaihtamalla mikropalvelumalliin voidaan vaikuttaa. Koska mikropalvelujärjestelmä koostuu monesta pienemmästä osasta, on helpompi päivittää järjestelmän yksittäisiä osia. Yksittäisten mikropalveluiden osalta voidaan tehdä kattavampia teknologiaan liittyviä päivityksiä ja niitä ei tarvitse tehdä samanaikaisesti, mutta mikäli kehittäjiä on paljon, malli ei estä samanaikaista kehittämistä. Ylläpidosta vastaavat ryhmät voivat käydä komponentit läpi ja ajaa palvelut tuotantoympäristöön päivityksien ja hyväksyttyjen testauksien jälkeen, koska omina kokonaisuuksinaan riippuvuus on minimoitu muihin palveluihin. Myös lähdekoodin optimointi ja palveluiden resurssien lisääminen on mahdollista yksittäiselle palvelulle, jolloin voidaan varmistaa järjestelmän

toimivuus paremmin kuin suuressa monoliittisessa sovelluksessa, jossa yksittäinen sovelluksen osa voi muodostaa pullon kaulan resurssien käytön osalta.

Näiden tarkastelujen perusteella voidaan päätellä vastaus TK1:een, joka kysyi milloin vaihtaminen mikropalvelumalliarkkitehtuuriin on paras vaihtoehto vanhan monoliittisen järjestelmän uudistusprojektissa? Rajataan ensin kriteerit, mitkä puoltavat eri arkkitehtuurimallien käyttöä, jotta saadaan yksinkertaistetumpi vertailupinta mallien välillä. Yksinkertaistettujen kriteerien ja kirjallisuustarkastelun perusteella voidaan tällöin antaa vastaus tutkimuskysymykseen.

Aikaisemman pohdinnan ja kirjallisuuden perusteella voidaan rajata seuraavat kriteerit, joissa mikropalvelumalli on parempi vaihtoehto kuin monoliittinen malli:

1. Järjestelmän lähdekoodin määrä on suuri
2. Järjestelmän on tarkoitus olla käytössä monia vuosia tai vuosikymmeniä
3. Järjestelmän elinkaaren pituus voidaan taata komponenttien jatkuvan ylläpidon avulla
4. Järjestelmään halutaan liittää paljon erilaisia ominaisuuksia ja muutokset järjestelmään halutaan nopeasti käyttöön
5. Kehittäjiä on paljon

Milloin taas monoliittisen mallin vaihtoehtoa puoltavat seuraavat kriteerit:

1. Järjestelmä on yksinkertainen tai lähdekoodimäärältään pieni
2. Järjestelmään ei kohdistu suuria muutoksia elinkaaren aikana
3. Järjestelmän kehitysvaiheessa halutaan edetä nopeasti
4. Kehittäjiä on vähän

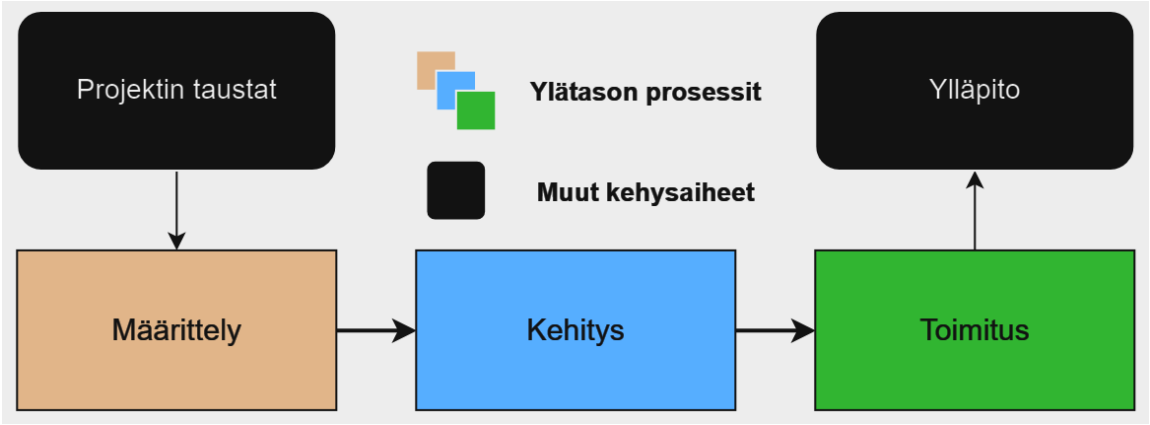
Joten yhteenvetona voidaan tiivistää, että suurempien ja kompleksisien järjestelmien, joiden elinkaaren halutaan olevan mahdollisimman pitkä ja jotka pitävät sisällään monenlaista bisneslogiikkaa, on helpompi ylläpitää mikropalvelumallin avulla, jos pystytään vastaamaan mikropalveluarkkitehtuurin haasteisiin ja riskeihin, esimerkiksi kompleksisuuden hallitsemiseen. Monoliittisen mallin parhaat puolet vastaavat pienempiä sovelluskokonaisuuksia, missä muutoksia halutaan harvemmin järjestelmän toimintaan.

Toisin sanoen TK1:een voidaan vastata: mikropalvelumalli on paras vaihtoehto uudistusprojektissa, kun järjestelmän käyttäjäkunta on suuri, jolloin voidaan hyödyntää palveluiden skaalautuvuusmahdollisuuksia ja optimoida järjestelmän kustannuksia, sekä halutaan varmistaa, että järjestelmän elinkaari on mahdollisimman pitkä.

4 UUDELLEENKEHITYS

Kun aletaan pohtimaan järjestelmän elinkaaren kannalta vaihtoehtoja jatkaa vanhojen ohjelmoitujen kokonaisuuksien kanssa tai kehittää ne kokonaan uudelleen, niin vetovastuussa olevat henkilöt voivat aloittaa päätyessään jälkimmäiseen vaihtoehtoon kysymyksellä: “Mitä voidaan odottaa projektilta, jossa aletaan kehittämään uudelleen järjestelmää, vanhan päivittämisen sijasta?”. Tärkeitä asioita huomioida kyseisessä tilanteessa ovat esimerkiksi, onko olemassa asioita, joita voidaan käyttää uudelleen kehittämisen apuna ja miten niitä hyödynnetään? Millaisiin riskeihin voidaan törmätä ja miten riskejä voidaan hallita? Kuinka rajataan projektin skaala, onko tarkoitus parantaa olemassa olevaa niin, ettei pelkkä uudelleenkirjoitus riitä toteutuksena? Tämän luvun tarkoituksena on esittää ja tarjota vaihtoehtoja uudelleenkehitysprojektien prosesseihin, vastata tutkimuskysymykseen TK2, eli miten suunnitella uudistustyön prosessi vaihdettaessa monoliittisesta mallista mikropalvelumalliin, sekä tuottaa vastauksen yhteydessä prosessikaavio uudistustyöhön liittyen.

Prosessien suunnittelu ei ole kovin yksinkertainen asia varsinkaan nykyaikaisissa IT-projekteissa, joissa on tarjolla useita eri työkaluja, niin hallinnan, dokumentaation, kuin myös kehityksen osalta. Prosessin määritelmä kirjaimellisesti on toimenpiteiden sarja, jonka tarkoituksena on saavuttaa tietty tavoite (Oxford, 2023). Projektityössä tämä tarkoittaa yleensä yksinkertaistetusti määrittely-, kehitys-, toimitus- ja lisäksi ylläpitotyöstä koostuvaa kokonaisuutta, mutta IT-projekteissa voidaan tarkastella pelkästään määrittelystä toimitusvaiheeseen päättyvää ketjua, koska siinä toimitettava asia luovutetaan toimituksen hankkineelle taholle ja ylläpito voi olla erillinen hankita riippuen siitä tekeekö toimituksen tehnyt taho ylläpitoa. Jotta pystytään vastaamaan TK2:een uudistustyön prosessin suunnittelemisesta, on hyvä tiedostaa taustat liittyen projektiin, sekä kuvassa 6 esiintyviin värillisiin laatikoihin ja niiden sisältöihin, jolloin puhutaan prosessin sisällön suunnittelusta. Kuvan 6 värillisten laatikoiden sisältöä avataan tulevissa kappaleissa enemmän ja niistä muodostetaan prosessikaavioita, joiden sisältö yhdistetään yhdeksi kaavioksi. Tämän lisäksi toinen asia mihin kannattaa kiinnittää huomiota, kun suunnitellaan prosessia ovat projektityöhön liittyvät taustat ja rajoitukset, joiden puitteissa prosessia voidaan hyödyntää. Tarkoituksena on muodostaa ylätason prosessisuunnitelmaehdotuksia mikropalvelukehityksessä, jossa tarkastellaan alatason (määrittely-, kehitys-, yms.) prosessien tarpeet ja käytännössä rakentaa kehykset sille, mitä tarvitaan uudistustyön prosessia varten.



Kuva 6. Ohjelmistokehityksen prosessit

4.1 Projektin taustat ja rajoitukset

Tarkastellaan aluksi projektiin ja sitä kautta prosesseihin vaikuttavia taustoja, sekä rajoituksia. Tässä vaiheessa voi viritä kysymys, että minkä takia halutaan tarkastella asioita, jotka ovat jokaisessa projektissa erilaiset? On todennäköisesti helpompaa sanoa, ettei mikään projekti ole täysin samanlainen, koska vähintään jokin järjestely eroaa toisistaan, oli se sitten esimerkiksi asiakas, toimitettava järjestelmä tai sitä kehittävä ryhmä. IT-projekteissa on tyypillistä, että yksi asia voidaan tehdä monella eri tavalla, oli kyseessä sitten tekninen toteutus tai tietomallin muutos. Mutta juuri tämän takia on tärkeää hakea yhteisiä piirteitä projekteista, jotta voidaan suunnitella askelia, joilla päästään haluttuun lopputulokseen. Esimerkkejä projektin taustoihin liittyvistä tiedoista, jotka voidaan yhdistää jokaiseen mikropalvelukehitysprojektiin ovat: tavoitteiden rajaaminen, projektin koko ja resurssien määrä. Tavoitteet rajaamalla voidaan projektia varten saada käsitys tehtävän työn määrästä ja projektin koosta, josta voidaan johtaa arvio resurssien tarpeesta. Nämä tiedot vaikuttavat kuitenkin lopulta prosesseihin, muun muassa sen perusteella, että mitä alatason prosesseissa kannattaa hyödyntää parhaan tuloksen saavuttamiseksi.

4.1.1 Tavoitteiden rajaaminen

Ensimmäisenä on hyvä määritellä projektille jonkinlaiset rajat. Projektin skaalan määrittäminenkään ei ole täysin mutkaton prosessi, mutta tarkentamalla edes tiettyyn pisteeseen asti helpotetaan mm. vaatimusmäärittelyä, riskien hallintaa, sekä karsitaan ylilyönnit odotuksien suhteen. Vastaamalla kysymyksiin, kuten mitä halutaan tehdä, minkä takia ja mitä saavutetaan, voidaan löytää projektisuunnitelmaan selvät kriteerit, josta konkreettinen maali muodostuu projektille. Tämä taas vastaa vähintäänkin epäsuorasti siihen,

kuinka iso projektin koko on. Tavoitteiden rajaamiseen voidaan sisällyttää sopimusteknisten asioiden määrittäminen, koska viimeistään sopimuksissa tulisi luetella konkreettiset tavoitteet ja projektin toteutumisen kannalta hyväksytettävyyden taso. Tavoitteiden rajaamisessa voidaan edetä esimerkiksi kahta eri reittiä, kun on kyse uudistusprojektista. Kartoittamalla olemassa olevan järjestelmän toimintoja, jonka perusteella voidaan määrittää pohjataso sille, mitä toimintoja pitää pystyä säilyttämään. Tämän lisäksi voidaan ottaa huomioon kartoituksessa toiveet, esimerkiksi mitä mahdollisia lisäominaisuuksia halutaan toteuttaa. Toisena vaihtoehtona voidaan sivuuttaa olemassa olevan järjestelmän toimintaa ja määritellä kokonaan uusia ratkaisuja, jolloin työmäärä voi olla suurempi ja muutokset radikaalimpia, sekä tilaavan asiakkaan tarvitsee huomioida mahdolliset prosessien muutokset oikeassa maailmassa, mutta joskus tyhjältä pöydältä aloittaminen voi tuoda huomattavia etuja, kun päästään irti sellaisista vanhoista prosesseista, mitkä eivät sovellu enää nykyiseen toimintaan tai ovat vanhentuneita käytäntöjä.

Mikropalveluissa kompleksisuuden takia on suositeltavampaa kartoittaa vanhat ominaisuudet ja miettiä kuinka ne pystytään rajaamaan uusien osien käyttöön. Tavoitteet voi asettaa jokaiselle palvelulle, jolloin saadaan pienempiä kokonaisuuksia, jotka lopulta yhdessä toimiessaan muodostavat hyväksytettävän järjestelmän. Määrittelyt voidaan tehdä sekä projektin todennäköisyydestä onnistua, sekä erikseen vaatimusmäärittely järjestelmän toiminnalle.

Otetaan esimerkiksi tavoitteen rajaamisesta, että monoliittinen järjestelmä ei nykyisyydessään skaalaudu hyvin isommalle käyttäjäkunnalle, vaikka sille on tarvetta. Skaalautumisvaikeudet voivat johtua esimerkiksi siitä, että järjestelmään ei voida liittää uusia osia, koska monet toiminnallisuudet käyttävät järjestelmään liittyviä tietoja ja jokaiseen käsittelijään pitää saada muokattua tuki uudelle järjestelmän osalle rikkomatta olemassa olevia toimintoja. Tässä tapauksessa voidaan tarjota ratkaisua skaalautuvuuteen jakamalla järjestelmän osat omiin mikropalveluihinsa, jolloin tuki voidaan lisätä myös yksittäiselle toiminnallisuudelle hajottamatta muita järjestelmän osia. Saadaan paljon konkreettisempi tavoite, että järjestelmän osien pitää pystyä skaalautumaan ylös tai alaspäin resurssien käytön muuttuessa. Jatkamalla samanlaista ajatusketjua muiden tarpeiden ja jopa toiveiden kanssa syntyy lista, joka määrittää koko projektin skaalan. Mikäli rakennemuutokset ovat suuret, niin silloin voidaan yleensä puoltaa uudelleenkehitystä, jos ei olla alun perin varmoja halutaanko kehittää järjestelmää uudelleen vai muokata vanhaa järjestelmää. Tietenkin pitää muistaa ottaa huomioon

järjestelmän koko, koska suurta järjestelmää voi olla hankala eriyttää ilman erityisosaamista tai tietoa liittyen järjestelmän hallitsemaan aiheeseen.

4.1.2 Projektin koko

Jos pitää määritellä uloin ulottuvuus toteutettaessa uudelleenkehitysprojektia, niin arvioimalla projektin kokoa asetetaan ensimmäiset kehykset, sekä ideat projektin ympärille. Koska työssä käydään läpi vaihtoa monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin ja mikropalvelumalli on jo itsessään kompleksinen rakenne, voidaan ennalta jo ajatella projektin olevan suuri. Projektin kokoa ei välttämättä kannata arvioida vain yhden näkökulman, kuten kompleksisuuden kautta. Brink ja Settlemire ovat projektinhallinnasta kertovassa verkkoartikkelissaan nostaneet alemman kuvan 7 kriteerit projektin koon määrittämistä varten (Brink, Settlemire 2016). Projektin koolla on suuri merkitys pelkästään asennoitumiseen projektin hallinnan kannalta. Jos projekti on organisaatiolle pieni tai keskisuuri, ei ole välttämättä tarpeellista käyttää yhtä paljon hallinnallisia mittareita ja projektinhallinnan keinoja kuin esimerkiksi suuressa projektissa, jonka aikataulu on pitkä ja kustannukset ovat suuret. Voidaankin ajatella, että mitä suurempi projekti, sen suurempi riskien esiintymisen mahdollisuus ja toteutuessaan riskit venyttävät aikataulua ja valmistumista. Bloch, kollegoidensa, sekä Oxfordin yliopiston avustuksella 2012 tehdyssä selvityksessä totesivat muun muassa, että n. 45 % suurista IT-hankinnoista eli projekteista, joiden hinta ylittää 15 miljoonaa dollaria, menee yli budjetin ja 7 %:ia ei valmistu aikataulussa (Bloch, Blumberg & ym. 2012). Jos ajatellaan Suomen mittakaavassa, suureksi luokiteltavat projektit maksavat mahdollisesti vähemmän kuin Yhdysvalloissa, mutta mediassa esiintyy jonkin verran myös esimerkkejä suurista epäonnistuneista projekteista, kuten kuuluisimpana varmaan pääkaupunkiseudun terveydenhuollon IT-ratkaisu Apotti, jonka kustannukset ovat nousseet jopa 40 % alkuperäisarviosta (yli 800 miljoonaa euroa kirjoitushetkellä) (Talouselämä 2022).

Työn määrä	Projektiryhmän kokemus	Kompleksisuus
Rahoitus (määrä)	Suorittavien työntekijöiden määrä	Toteutettavien ominaisuuksien määrä
Projektin hinta	Aikataulu	Muutoksien määrä

Kuva 7. Näkökulmia projektin koon määrittämiseen (Brink, Settlemire 2016)

Vaikka projektin koon määrittäminen ja koon suuruudesta johtuviin toimenpiteisiin ryhtyminen on organisaatiokohtaista, on varsinaista työtä tekevän organisaation hyvä olla varautunut omilla malleillaan vastaamaan projektien koon aiheuttamiin haasteisiin. Jos verrataan mikropalvelumalliin siirtymistä aiemman kuvan 7 kriteereihin aiempien kappaleiden perusteella, kuten luvussa 3.2, niin voidaan todeta, että uudelleenkehitys mikropalveluksi täyttää seuraavat kriteerit suureksi projektiksi: Työntekijöiden määrä on suuri (esim. > 5), projektiryhmällä on hyvä olla vankka kokemus mikropalveluista ja/tai asiantuntemus järjestelmän aihealueelta, muutoksien määrä on suuri jaettaessa ohjelmaa pienempiin osiin, jolloin työn määrä on luultavasti myös suuri. Kuten aiemmin mainittu mikropalvelut lisäävät kompleksisuutta, joten näillä mittareilla projektin suorittamiseen kannattaa varautua suurella kynnyksellä.

4.1.3 Henkilöstöresurssit ja toteutus

Kuten kappaleessa 4.1.2 mainittiin, mikropalveluita rakentava projekti on usein kooltaan suuri, jollei vähintään keskisuuri projekti. Koska suurissa projekteissa on usein mukana useita tekijöitä ja mikropalveluarkkitehtuuri mahdollistaa tarvittaessa monien ryhmien toiminnan samanaikaisesti, joten projektissa on mahdollista käyttää muun muassa useista kehittäjistä koostuvia ryhmiä. Tällaisessa tilanteessa projektinhallintametodit kannattaa keskittää monen ryhmän samanaikaiseen toimintaan, jotta toteutuksessa voidaan edetä nopeammalla aikataululla, mutta välttäen suuret erot järjestelmän toteutuksen välillä. Koska nykyään etä- tai niin sanotun hybridityömallin, jossa työskennellään toimistotiloissa ja etäyhteyksien

välityksellä, vaikutukset ovat suurempia IT-alalla, on hyvä kiinnittää myös huomiota siihen, että tietoa jaetaan tarpeeksi tilanteissa, jossa kehitysryhmä ei työskentele usein samoissa tiloissa. Tämä hoituu muun muassa Scrumin päivittäisissä kokoontumisissa eli dailyissä, sekä muissa kehitykseen liittyvissä tapaamisissa. Riippuen toteutettavan työn määrästä ja järjestelmän tarpeista, kehittäjärooleihin voi kuulua front-end ja back-end -kehittäjiä, tai näiden yhdistelmiä eli niin sanottuja full stack -kehittäjiä. Näitä kattotermejä voidaan jakaa vielä alarooleihin esimerkiksi kokemuksen tai päätösvallan perusteella (yleensä puhutaan juniori- tai seniorirooleista), sekä tietyn erikoisosaamisen mukaan. Erikoisosaamisen mukaan jaetuista rooleista voidaan pitää esimerkiksi back-end kehittäjien jakamista integraatio-, infrastruktuuri- ja palvelukehittäjärooleihin. (Lohani. 2018)

Järjestelmän toiminnan kuvaamisesta vastaavat roolit kuten arkkitehdit ja määrittelijät. Periaatteessa arkkitehdit ovat myös määrittelijöitä, mutta yleensä arkkitehdit huomioivat enemmän järjestelmän teknisiä taipumuksia, missä muita määrittelyjä tekevät roolit voivat ohittaa tarvittaessa teknistä aspektia, joten roolit voidaan erottaa toisistaan. Jos järjestelmällä on käyttäjien käyttämä käyttöliittymä, niin suositellaan myös käyttöliittymä- ja mahdollisesti myös käyttökokemussuunnittelijoita, koska käyttäjäkokemuksella on suuri rooli järjestelmän hyödyntämisessä ja käyttäjien tyytyväisyydestä käyttämiinsä työkaluihin (DesignStudio, 2023). Nämä roolit yhdessä käytännössä koostavat sovelluksen pohjapiirrokset, jolloin muut roolit voivat käyttää suunniteltuja asioita hyödykseen. Joissain tapauksissa arkkitehdin ja kehittäjän roolit voivat yhdistyä, sekä pienissä projekteissa joskus käyttöliittymäkehittäjä voi toteuttaa käyttöliittymäsuunnitelman, mutta isoissa projekteissa suunnittelijarooleista on hyötyä, varsinkin koska työtaakka voi olla suuri.

Laadun varmistajat, eli arkikielellä testaajat ovat myös hyvä voimavara, joiden apua voidaan hyödyntää niin kehityksen aikana, kuten projektin julkaisun jälkeen tarvittaessa. Laadun varmistuksessa on kyse siitä, että järjestelmän toimintaa testataan käyttäjätarinoita ja määrittelyjä vasten, jolloin saadaan tietoa järjestelmän sisältämistä virheistä ja muutostarpeista. Mitä aikaisemmin järjestelmässä piilevät bugit saadaan tietoon, sitä helpompi on jäljittää niiden sijainti ja korjata ne. Testaamalla sovellusta voidaan varmistua siitä, että uudet toiminnallisuudet eivät riko olemassa olevaa toiminnallisuutta, sekä varsinkin mikropalveluissa, joissa monta erillistä palvelua voi käyttää ja muokata järjestelmän tietoja, saadaan selville se, että palvelut puhaltavat yhteen hiileen, eivätkä aiheuta ongelmatilanteita toisilleen.

Muita yleensä hallinnollisia rooleja, voi projektissa olla esimerkiksi ketterässä kehityksessä scrummasteri, jonka tehtäviin kuuluu scrumtiimin ohjaaminen ja tehokkuuden maksimointi fasilitoimalla keskustelua scrumin osalta (Lekman, 2013). Vaikka usein scrummasterina voi toimia jonkin toisen roolin omaava henkilö, on erillinen scrummasteri arvokas lisä varsinkin suuressa projektissa, koska tämä pystyy luovimaan haasteita ja vaikuttamaan projektin suuntaan, jos henkilön työmäärään ei sisälly muita aikaa vieviä tehtäviä (Schwaber, 2004.). Tämän lisäksi projekteissa voidaan pitää projektipäälliköitä, vaikka se on aikaisemmin mainitun scrumin niin sanottujen puhtaiden metodien vastaista. Projektipäälliköiden etu on kokonaisen tilannekuvan hallitseminen ja ylätason asioiden selvittäminen. Tämän lisäksi ketterissä menetelmissä halutaan päättävä elin eli projektin omistaja, joka käytännössä antaa viimeisen leiman päätöksiin. Suositeltavaa on, että projektin omistaja on viime kädessä yksittäinen henkilö, jolla on päätäntävalta sidosryhmien ehdotuksia kuunnellen. Mikäli projektin omistajuus on epäselvää, se voi aiheuttaa viivästystä päätöksissä, sekä hajottaa projektin eheyttä, koska eri tahot voivat olla eri mieltä asioista ja tämän takia päätöksiä ei saada tehtyä.

4.2 Määrittely

Kun projektin tavoitteet ovat tiedossa, voidaan aloittaa varsinainen määrittely. Toki, jotta tavoitteet on saatu asetettua, on toivottavasti tehty alustavaa selvitystä järjestelmän toiminnasta, sekä järjestelmän toimintaympäristöstä. Alkavan määrittelyn tavoitteena on tuottaa suunnitelma toiminnallisuuksien toteuttamiseen kehitysvaiheessa. Määrittelyn pohjana toimii projektin taustatiedot, joiden pohjalta juontuvat esimerkiksi kuvan 8 vaatimukset, sekä niihin liittyvät aikataulut ja vaatimuksiin sekä aikatauluihin liittyvät tunnistettavat riskit. Periaatteessa määrittely on voitu tehdä ennen varsinaista projektin aloitusta, mutta on hyvä tiedostaa, että määrittelyt voivat muuttua paljonkin tapauskohtaisesti kehittämisen aikana. Tämä korostuu varsinkin, jos kehityksessä käytetään ketteriä menetelmiä (engl. agile methods), koska ketterien menetelmien ytimessä toimivat iteraatiot, joissa lyhyissä ajanjaksoissa pyritään tuottamaan tuloksia ja jotka on tarkoitus julkistaa demonstraatiotilaisuuksissa. Mikäli esille tuodut tuotokset eivät saa hyväksynnän leimaa, ne yleensä palautetaan kehityskierrokseen ja tässä vaiheessa määrittelyä voidaan joutua muuttamaan.

Kattavan määrittelyn avulla voidaan vähentää iteraatiokierroksia tietyn kehitettävän osuuden osalta. Jos alkuperäisessä määrittelyssä on osattu huomioida suurin osa perustapauksista ja

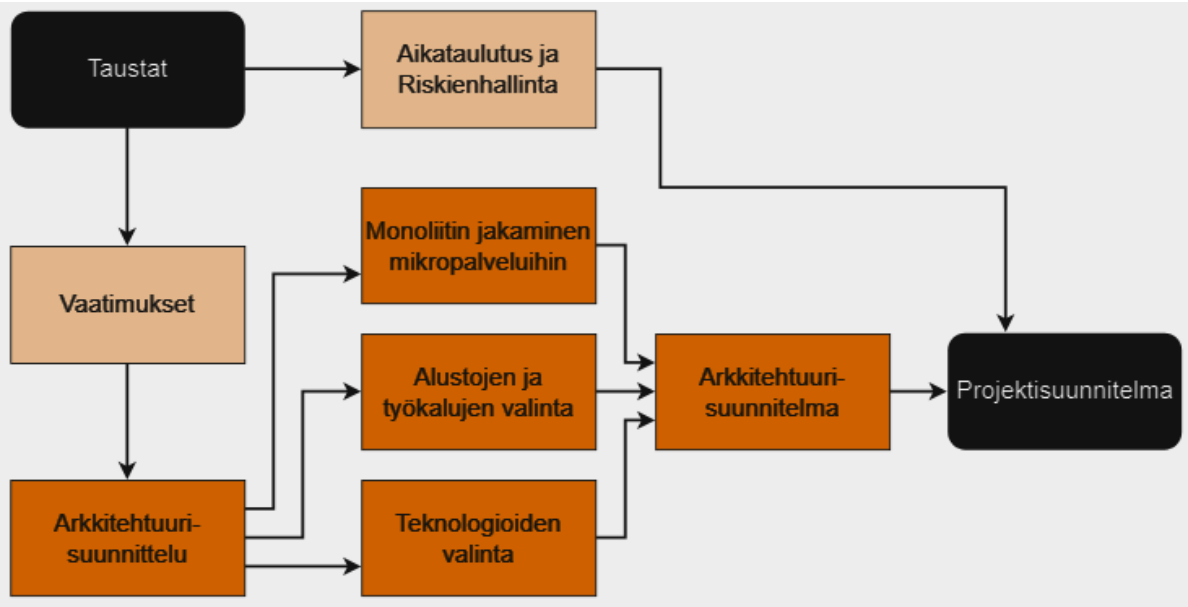
mahdollisia rajatapauksia, on helpompi kehittäjien kannalta toteuttaa arvoa antavaa ohjelmistoa. Realistisesti kaikkia asioita ei voi ennalta määrittää ja määrittelyn tekijän rooli voi isommassa projektissa olla muun kuin kehittäjän tai arkkitehdin harteilla, jolloin teknologiset haasteet voivat alun perin jäädä huomiotta. Tosin edes kehittäjät ja arkkitehditkään eivät voi tietää kaikkea ennalta, joten on hyvä asennoitua tuottamaan määrittelyjä esim. dokumentteja, jotka kertovat kokonaiskuvan toteutettavasta toiminnallisuudesta ja pyrkiä ketterästi silottamaan yksityiskohdat paikalleen. Määrittelyyn on hyvä osallistua myös kehittäjiä ja testaajia. Kun määrittelyjä valmistellaan isommalla ryhmällä tämän tyyppiset määrittelytapaamiset lisäävät henkilöiden ajankäyttöä, mikä voi lyhyellä aikavälillä johtaa varsinaisen kehitystyön hidastumiseen. Tämän takia on tärkeää löytää tasapainoinen tila, jossa määrittelyjä saadaan edistettyä, mutta henkilöt eivät ole kiinni ikuisessa tapaamiskierteessä.

Yleisesti ottaen voi olla haastavaa luoda määrittelyjä iteraatiokierroksia tai yleistäen ketteriä kehitystapoja varten. Uudelleenkehitysprojektissa, jossa pyritään pois monoliittisesta arkkitehtuurista kohti mikropalveluita, kannattaa erityishuomiona nostaa määrittelyihin liittyvissä asioissa edellä mainitun teknisen osaamisen tärkeys. Vaikka määrittelyille on mahdollinen vakaa pohja vanhan järjestelmän kautta, mikropalveluihin jakaminen voi tuottaa vaikeuksia, koska kokonaisuuden hallinta on kompleksisempaa, sekä tietyt aikaisemmat ohjelmistoparadigmat eivät välttämättä ole yhteensopivia mikropalveluiden kanssa. Koska mikropalvelut käyttävät kommunikaatiossaan hyödyksi rajapintakutsuja, on määrittelyssä tärkeää ennakoida järjestelmän resurssien käyttöä ja mikä on ohjelmateknisesti toteutettavissa.

Määrittelyn ohella voidaan ottaa huomioon riskit ja niiden hallinta. Tämä tarkoittaa riskien kartoittamista, sekä suunnitelmallista vaikutusta ilmentyvien riskien eliminoimiseen. Esimerkkejä ilmentyvistä riskeistä vaihdettaessa monoliittisesta mallista mikropalveluihin voivat olla muun muassa teknologia erot, ns. toimittajaloukku (engl. vendor lock-in) ja järjestelmää käyttävän organisaation, organisaatioiden tai käyttäjien haasteet omaksua mikropalveluiden toiminta. Jokaiselle nostetulle riskille pyritään määrittämään riskin taso, kuinka helposti riski voi ilmentyä ja kuinka vakavaa riskin toteutuminen projektille on. Esimerkiksi teknologiaongelmat voivat johtaa arvioitua suurempaan ajankäyttöön, jolloin riskin toteutuminen aiheuttaa aikataulumuutoksia ja pahimmassa tapauksessa projektin hyllyttämisen. Palaamalla takaisin määrittelyjen vaikutukseen voidaan huomata, että hyvin tehdyt määrittelyt tuottavat paremman aikatauluarvion, sekä vähentävät riskien toteutumistasoa. Luvun lopussa esitetyssä prosessikaaviossa eli kuvassa 8 on niputettu

aikataulutus, sekä riskienhallinta yhteen laatikkoon, koska aikataulutus on hyvin riippuvainen riskientoteutumisprosentista.

Jos mietitään taustatietojen jakamista dokumentaatioksi, jonka tarkoitus on koota yhteen muun muassa vaatimukset tai projektin laajuus, riskit ja projektin aikataulutus, voidaan puhua projektisuunnitelman määrittämisestä. Vaatimukset luovat pohjaa myös arkkitehtuurisuunnitelmille, josta olemme kiinnostuneita tämän työn ja tulevien kappaleiden osalta. Ylemmän pohjalta voimme luoda määrittelystä kaavion, joka on esitetty kuvassa 8. Olemassa olevasta tiedosta jalostetaan pohja tässä tapauksessa uudelleenkehitysprojektille. Kuvassa 8 on jaettu arkkitehtuurisuunnittelu tärkeisiin pienempiin palasiin, kun on tarkoitus vaihtaa monoliitista mikropalveluksi. Varmaan tärkein osio on pystyä jakamaan monoliitti loogisiin kokonaisuuksiin, jotta kehitettävät mikropalvelut toimivat yhdistettynä. Teknologioiden alustojen ja työkalujen valinta voi olla vapaampaa, koska palvelut toimivat omina kokonaisuuksinaan ja välittävät keskenään viestejä toimiakseen. Kuitenkin ne pohjaavat itse siihen, miten palvelut on jaettu. Kompleksisuuden vähentämiseksi on suositeltavaa asettaa rajoja teknologioiden ja työkalujen osalta. Ympäristöt tai alustat, joissa palveluita ajetaan, on päätettävä myös hyvin aikaisessa vaiheessa, jotta pystytään luomaan esimerkiksi kehitysympäristöt. Pilvipalveluiden osalta on yleensä kysymys pilvipalveluiden tarjoajan valinnasta, eikä niitä käyttäessä tarvitse keskittyä toimitettavaan laitteistoon, vaan pilvialustoille pystytään konfiguroimaan tarvittavat virtuaaliset alustat.



Kuva 8. Määritelmävaiheen prosessikuva

4.3 Kehitys

Kehitykseen kuuluu järjestelmän luominen. Kehitys ja sen alaprosessit voivat olla hyvinkin kirjava joukko toimenpiteitä, koska mikään ei tavallaan sido kehittäjiä käyttäytymään joka kerta samalla tavalla, kun työskennellään ohjelmistoprojektien kanssa. Yleensä kehittäjille on muodostunut jonkinlainen kuva työskentelytavoista omien kokemuksiensa perusteella, joita voi myös kutsua prosesseiksi. Kehityksen alussa on hyvä käydä läpi sidosryhmien kanssa, mitä kehitys pitää sisällään ja millaisia toimenpiteitä odotetaan kehittäjiltä, sekä muilta kehitystä auttavilta tahoilta, esimerkiksi kun reagoidaan muutoksiin tai esiintyviin ongelmatilanteisiin. Tässä luvussa keskitytään yksityiskohtaisemmin toimintatapoihin, sekä viitekehyksiin, joita kehityksen aikana voidaan hyödyntää. Suosituksien kautta voidaan tarkastella lopuksi kehityksen näkökulmasta, miten uudistustyön prosessia voi suunnitella, kun halutaan vaihtaa mikropalvelumalliin monoliittisesta järjestelmästä TK2 mukaisesti.

4.3.1 Refaktorointi

Myös uudelleenkehityksen, kuin tavallisen ohjelmiston elinkaaren ylläpidon aikana voidaan tehdä refaktorointia. Refaktorointi terminä tarkoittaa sitä, että lähdekoodia muokataan sillä tavalla, että ohjelma säilyttää käyttäjille näkyvän toimintansa samanlaisena, mutta esimerkiksi tehostaa ohjelman suoritusaikaa eliminoimalla kompleksisuutta tai parantaa ylläpidettävyyttä yksinkertaistamalla ohjelman suorittavaa koodia.

Mikäli projektissa on käytössä ketterät menetelmät projektinhallinnassa, niin iteraatiokierroksien aikana syntyy usein refaktorointitarpeita ohjelmistokehityksessä. Refaktorointi voi tuottaa mm. seuraavia parannuksia kuten helpottaa koodin luettavuutta, parantaa koodin uudelleen käytettävyyttä, syventää ohjelmoijan ymmärrystä, sekä välttää sudenkuoppia, jotka tulevaisuudessa kasaantuisivat tekniseksi velaksi. TechTargetin artikkelin (Gillis. Päiväämätön) mukaan, refaktorointi kannattaisi suorittaa ennen suurempien toiminnallisuuksien tai päivityksien lisäämistä, koska yksinkertaistamalla koodia ennen isoa päivitystä helpotetaan päivityksen luomista, koska ymmärrys toiminnasta, sekä lähdekoodin luettavuus paranee. Kolikolla on tässäkin suhteessa kääntöpuolensa, eli koska refaktorointi on yleensä aikaa vievää, niin se voi johtaa aikatauluviivästyksiin. Refaktorointi ei myöskään suoraan poista suunnittelutyöstä johtuvia ongelmia, vaan tehostaa koodin, eikä ennalta suunnitellun prosessin toiminnallisuutta.

Käytännössä jokainen uudelleenkehitysprojekti on suuri refaktorointiprojekti, koska halutaan usein säilyttää sama toiminnallisuus mihin on aikaisemmin totuttu. Refaktoroinnin olemassaolo on hyvä tiedostaa, koska usein ohjelmistokokonaisuuksien toiminnallisuuksia pystytään parantamaan ensimmäisestä kerrasta, kun toiminnallisuus on saatu valmiiksi. Aikataulurajoitteet ja ohjelmoijan kokemus, tavat, sekä käytetyn teknologian paradigmojen tuntemus voivat jättää epäoptimaalisia toteutuksia ohjelmakoodin sekaan, joten on suositeltavaa, että refaktorointi on suunniteltua. Hyvien tapojen mukaan uudelleenkirjoitettu ohjelmisto tai sen osa on vertaisarvioitu ennen julkaisua.

4.3.2 Document driven development / Test driven development

Kehityksen apuna voidaan käyttää myös sellaisia tekniikoita kuten dokumentti- tai testivetoista kehitystä (engl. document driven development eli DDD tai test driven development eli TDD). Näiden tekniikoiden tarkoituksena on tarkentaa kehityksessä keskityttävään suuntaan, joko dokumentaation hallinnan tai yksikkötestien avulla.

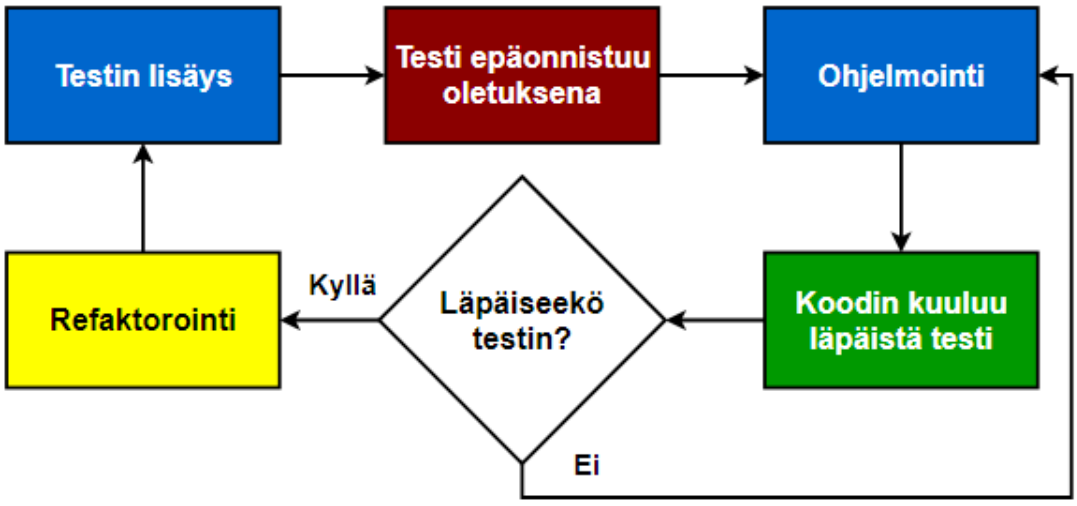
DDD:n tarkoitusperänä on kattava dokumenttien käsittely, jolla pystytään vaikuttamaan vaihtuviin vaatimuksiin, mikä on kovin yleistä ohjelmistokehitysprojekteissa. Tutkivaa kirjallisuutta aiheesta on jonkin verran, mutta muun muassa Bano kollegoineen 2012 julkaistussa paperissaan nostivat huomioksi, että lisää empiiristä tutkimusta kaivataan muuttuvien vaatimuksien syiden selvittämiseksi, koska aiheesta on paljon keskustelua, mutta vähemmän tutkimuksen kestävää tietoa (Bano, Imtiaz & ym. 2013). DDD voi sisältää epäformaaleja, sekä formaaleja dokumenttimuotoja (Luqi, Zhang & ym. 2004), esimerkkinä epäformaaleista dokumenteista videot, kuvat ja luonnollinen kieli, jota ihmiset ymmärtävät, sekä formaaleista dokumenteista, jota koneet ymmärtävät paremmin ovat testitapaukset, matemaattiset notaatiot, suunnittelu- ja ohjelmointikielet, erilaiset lähdekoodit ja niin edelleen. Kaikkea tätä informaatiota pitää pystyä hallitsemaan, jolloin tarvitaan dokumenttien säilytyspaikka, sekä säännöt dokumenttien luomiseen ja päivittämiseen. Tämän lisäksi Luqi, Zhang, sekä muut ovat nostaneet tarpeelliseksi prosessin arviointijärjestelmän (engl. process measurement system), jolla tarkkaillaan muutoksia vaatimuksiin. Prosessin arviointijärjestelmä toimii yksinkertaistettuna siten, että dokumenttien säilytyspaikasta saatavilla olevaa tietoa voidaan tuoda arviointijärjestelmän käyttöön, jonka perusteella pystytään määrittämään mittarit riskeille ja arvioimaan luottamusta järjestelmän valmistumiseen, kun vaatimukset muuttuvat. Riskit voidaan jakaa neljään eri kategoriaan, jotka ovat: prosessi-, tuote-, resurssi- ja teknologiariskit, sekä niitä varten on annettu

ehdotukset, kuinka muodostaa niistä mittareita, jotka oikeasti kuvaavat riskien ilmentymistä. Työssä on esitetty myös DDD:n hankaluuksia, jotka ovat tulleet ilmi tutkimuksen aikana. Näitä olivat teoksen mukaan monien työkalujen tukeminen, koska dokumenttiformaatteja voi olla erittäinkin paljon, sekä ontologisien ongelmien selvittäminen, mikä tässä yhteydessä tarkoittaa tiedon muuntamista olevaan muotoon eli dokumenteiksi, joita pystytään tulkitsemaan ja mahdollisesti johtamaan niistä automaattisesti mm. lähdekoodia.

Toinen mielenkiintoinen huomio liittyen DDD:hen on sen yhteensopivuus ketterien menetelmien kanssa. Esimerkiksi paperissaan “Agile Software Development and its Compability with a Document-Driven Approach? A Case Study”, tutkivat Heeager sekä Nielsen yhteensopivuutta projektissa, jossa pyrittiin käyttämään DDD:tä suuren ketterän sulautettujen ohjelmistojen projektin kanssa (Heaager & Nielsen, 2009). Tutkimuksessa keskityttiin 17 ohjelmoijan kokemuksiin tekniikoiden käytöstä ja tutkimuksen lopputuloksena tavallaan voidaan puoltaa DDD:n toimivuutta ketterän kehityksen apuna, mutta esimerkkinä toimivassa projektissa haasteeksi nousi täysin näiden kahden kehityksen tekniikan yhdistäminen, koska osa DDD:n periaatteista haastaa ketterän manifestin teesejä, kuten sen, että ketterässä kehityksessä ohjelmoijan kyky saada aikaan ohjelmistoa ja kommunikoida kasvotusten on suuremmalla painoarvolla, kuin hallita suurta määrää dokumentteja (Beck, Grenning & ym. 2001). Mutta toiselta kannalta katsoen, tämän tyyppinen sekoitettu prosessi, missä kahden tai useamman tekniikan hyviä puolia pyritään yhdistämään, on realistisempi vaihtoehto käyttää oikean maailman projekteissa koska on erittäin vaikeaa idealistisesti seurata esimerkiksi yhden tekniikan puristista tapaa tehdä asioita. Mitä tulee DDD:n käyttämiseen ohjelmistokehitysprojektissa, on selvästi hyötyä siitä, mitä tarkemmalla tasolla voidaan dokumentoida määritelmiä ja toteutuksen vaiheita, koska tämä lisää tiedon jakoa projektissa eri sidosryhmien kesken, mutta dokumenttien luomiseen, hallitsemiseen ja säilyttämiseen tarvitaan yhteiset käytännöt, jotta dokumenteista on loppukädessä hyötyä.

Toinen esitetty vaihtoehto, eli testivetoinen kehitys (TDD) lähtee ajatuksesta, että ohjelmointia varten rakennetaan ensin testitapaukset ja yksikkötestit, joita vastaan lähdekoodia testataan ja ohjelman osa on valmis, kun se läpäisee omat testinsä. TDD:n hyötyjä ovat muun muassa se, että se on hyvin yhteensopiva iteroivan kehityksen kanssa, se parantaa rakennettavien osien laatua refaktoroinnin ollessa täsmällisempää nopeamman palautteen perusteella ja vähentää riskiä bugien ja ei haluttujen muutoksien syntymiseen (TestDriven. Päiväämätön; Acosta, Gajda. Päiväämätön).

Kuvassa 9 on havainnollistettu TDD:n mukainen prosessi, jossa kehitetään komponentti tai järjestelmän osa testien avulla. TDD:ssä lähdetään siitä, että kirjoitetaan testi sille komponentille määrityksien perusteella mitä halutaan saada aikaiseksi. Tämä voi käytännössä olla esimerkiksi, että komponentin laskiessa 2 lukua yhteen summan täytyy olla aina suurempi kuin kumpikaan yhteenlasketuista luvuista, jolloin hylätään muut tapaukset (ei syötetä lukuja tai syötetään negatiivisia lukuja). Testitapauksen kuulu epäonnistua ajettaessa se ensimmäisen kerran, koska sille ei tässä vaiheessa ole luotu suorittavaa koodia. Näin varmistetaan se, että testi ei jokaisessa tapauksessa palauta totuusarvoa totta eli true, jolloin testi ei varmista mitään lopputulosta. Kun testi tai testit on luotu, voidaan toteuttaa ohjelma, jonka tehtävänä on päästä läpi testeistä. Tällä tavalla eliminoidaan ohjelmoidessa huonoa käytäntöä, jossa mennään siitä missä aita on matalin, toteuttamalla jotain mikä näyttää vastaavan alkuperäistä määrittelyä, mutta voi sisältää tapauksia, joissa ohjelma käyttäytyy määrittelyjen vastaisella tavalla. Joskus tietenkin määrittelyt eivät ole tarpeeksi tarkkoja huomioimaan rajatapauksia, mutta testien tarkoitus on estää poikkeavat tilanteet ja nostaa nopeasti esille tällaisten ilmentyminen. Jos ohjelma suoriutuu kaikista testeistä, voidaan todeta, että ohjelma toimii halutulla tavalla ja voidaan refaktoroida mahdolliset ylimääräiset osat pois lähdekoodin joukosta, sekä lisätä uusia testejä ja iteroida näin varmistamalla samalla, etteivät järjestelmän muut osiot muutu uusien ominaisuuksien luonnin yhteydessä, koska testit ajetaan uudestaan ohjelmaa rakennettaessa.



Kuva 9. Test Driven Development -viitekehyksen mukainen prosessi.

Lisähuomiona TDD:hen liittyen puhutaan yleensä kuvassa 9:kin esiintyvään värikoodaus-sykliin, jossa toistetaan punainen-vihreä-refaktorointi-iteraatiota. Tämän merkitys on siinä,

että viitekehyksessä merkataan epäonnistuneet testit punaisella, läpäistyt testit vihreällä, sekä refaktorointiin liittyvässä askeleessa voidaan korjata koodissa esiintyviä virheitä, koska aina ensimmäinen toteutus ei ole se paras ja selkein, vaikka se lopulta toteuttaisikin testit minimivaatimuksena. Toinen mainittavan arvoinen asia TDD:stä on se, että siihen liittyy myös omat haasteensa ottaa käyttöön kehitysprojekteissa. Testausvetoinen kehitys vie aikaa, koska uusien, hyvien testien kirjoittaminen ottaa oman aikansa, mutta toisaalta voi maksaa itsensä monin kerroin takaisin, koska testit itsessään varmistavat sen, että kehittäjien täytyy palata vähemmän takaisin korjaamaan yhteensopimattomia vanhoja toteutuksia, joita ei välttämättä huomata, kunnes pitkä aika on kulunut kehittämisestä. Osittain liittyen kulutettuun aikaan, testivetoinen kehitys voi vaatia kehittäjiä oppimaan uusia tekniikoita, jolloin testien tekeminen ja niiden tosiasiallinen tehokkuus voi kärsiä. Kaikille mahdollisille tapauksille voi olla vaikea tuottaa yksikkötestejä, jolloin tehokkuus kärsii, koska tietty määrä ohjelmasta ei ole testattavissa testien kautta. Lisäksi käytetyt teknologiat voivat vaikuttaa testien kirjoittamisen helppouteen ja kaikilla teknologioilla ei välttämättä ole integroitua testausalustaa, jolla testit voidaan ajaa, sekä käyttöliittymille on vaikea toteuttaa yksikkötestejä, koska käyttäjät eivät yleensä toimi kuin koneet ja noudata tiettyä järjestelmällisyyttä käyttäessään ohjelmaa. (Prasdika. 2023)

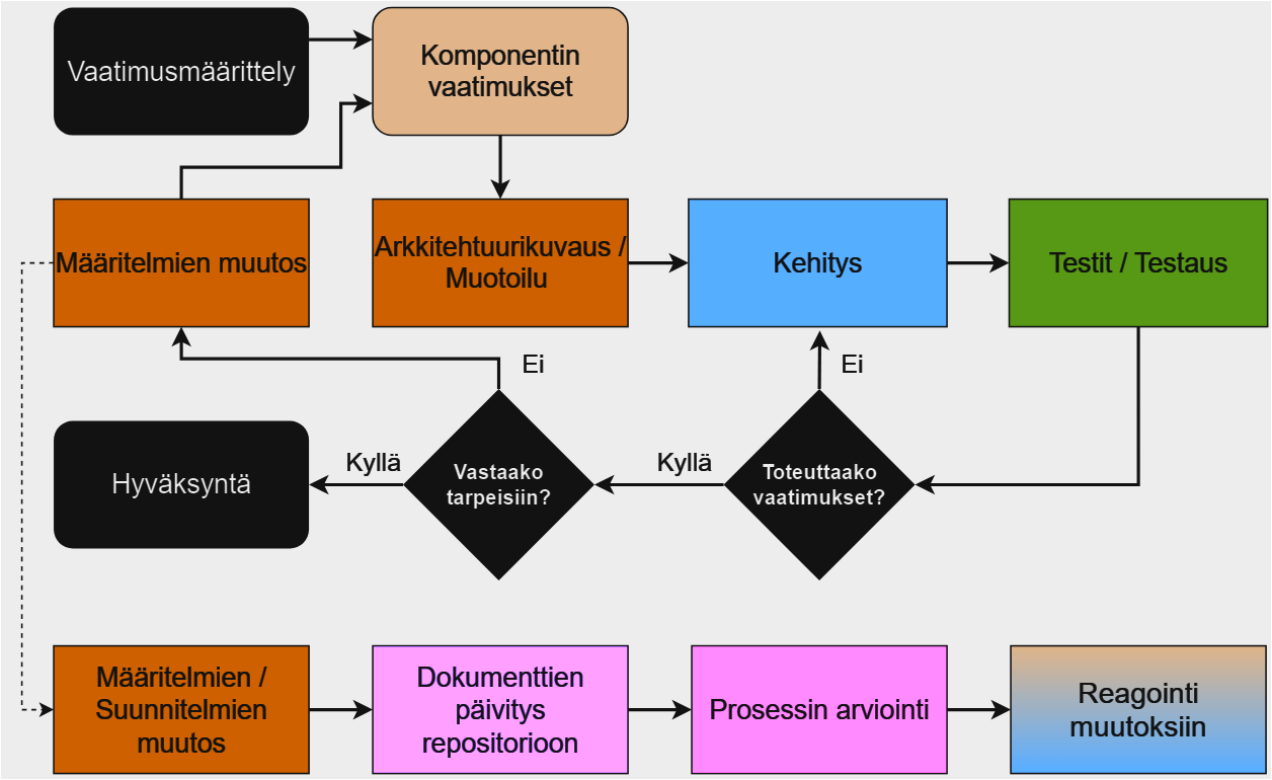
4.3.3 Kehityksen suosituksia

Mitä tulee kehityksen kannalta prosessien suunnitteluun, niin jos palataan luvun 3 selvityksiin, missä totesimme testattavuudesta, että monoliittisen järjestelmän testattavuus on helpommin yhtenäistettävissä, kun taas mikropalveluissa jokaisen palvelun on tarkoitus olla oma kokonaisuutensa. Tämän takia mikropalveluissa testattavuus ei ole yhtä yhtenäistä, mutta siltikään kumpikaan toteutus ei varsinaisesti sodi esimerkiksi testivetoisen kehityksen kanssa. Mikropalveluiden ongelmaksi voi tosin koitua se, että jos kaikille palveluille halutaan kattavat testit ja pyritään toteuttamaan täysin testivetoinen viitekehys, niin testien kirjoittamiseen kulutettu aika voi lisätä resurssien käyttöä ja koska mikropalveluiden kehitys on yleisesti hitaampaa kompleksisuuden takia, kannattaa harkita onko mahdollista järjestää kattava testaus, varsinkin kun määrittelyt voivat muuttua matkan varrella. Paremmalta vaikuttavampi keino mikropalveluiden kehityksessä on kehittää järjestelmän osat ja varmistaa kriittisille toiminnallisuuksille testitapaukset, jotta ohjelma toimii jatkossa samalla tavalla, kun kehitettävän palvelun maturiteetti on tarpeeksi korkealla ja hyväksytetty sidosryhmien kautta.

Määritelmät ja niistä syntyvät dokumentit ovat erittäin tärkeitä varsinkin mikropalveluiden osalta. Koska komponentteja voi olla monia, niiden toiminnasta ja toteutuksesta, sekä niitä koskevista muutoksista on hyvä olla selvä formaali tai vähintään epäformaali selonteko. Myös riskien arviointia helpottava prosessin arviointijärjestelmä vaikuttaa hyvältä konkreettiselta mittarilta kehitysvaiheessa. Kehitystyö voi edistyä hyvinkin paljon, mutta se ei välttämättä näy konkreettisesti sidosryhmille, koska määritykset voivat päivittyä tiuhaan tahtiin kehityksen edetessä. Tällainen arviointityökalu voi mahdollistaa suuremman näkyvyyden kohdatessa ongelmia, jotka jarruttavat tiettyyn pisteeseen pääsemistä. Muun muassa näiden nostettujen huomioiden ja kirjallisten lähteiden tarkastelulla kehitysvaiheessa kannattaa panostaa dokumenttivetoiseen kehitykseen, kun kehitetään mikropalveluita. Se vaikuttaa realistisemmalta lähestymistavalta, koska mikropalveluiden kehityksessä aloittamalla testien kirjoittamisella kulutetaan suuri määrä aikaa, joka voidaan käyttää hyödyksi tiettyjen kriittisten palveluiden, sekä niiden ympäristöjen nostamiseen. Tietenkin pitkällä aikavälillä testien kirjoittaminen maksaa itsensä takaisin, joten emme suosittele kokonaan testien kirjoittamatta jättämistä.

Esimerkiksi dokumenttivetoisen kehityksen prosessista mallinnetaan kuvassa 10. Prosessi käytännössä alkaa aina jonkinlaisista taustatiedoista, joka kuvassa on esitetty laatikolla vaatimusmäärittelystä. Tämän perusteella jokaisella komponentilla on tarkoituksensa, mutta komponentin tarkoitus, eli mitä vaatimuksia sen pitää toteuttaa, voi muuttua myös kehityksen aikana. Tämän takia tarvitaan keino reagoida muutoksiin. Jos vertaamme esimerkiksi testivetoisen kehityksen prosessia kuvasta 9, niin testivetoisessa prosessissa otetaan kyllä huomioon, että komponentin yksityiskohdat voivat muuttua ja muutoksien takia voidaan lisätä testejä, mutta testi ei välttämättä avaa ihmiselle miksi muutos on toteutettu tai avaa mikropalveluiden välisiä yhteyksiä, joista muutos on voinut juontua, koska mikropalvelutoteutuksissa jokaiselle mikropalvelulle on tarkoitus luoda omat testinsä. Dokumenttivetoinen kehitys vaikuttaa tässä suhteessa tarjoavan paremman tavan ottaa huomioon muutokset, sekä arvioida muutoksien kautta prosessien toimivuutta. Testivetoinen kehitys ei esimerkiksi piittaa kovinkaan paljon siitä, kuinka nopeasti asioita saadaan edistettyä, niin kehityksen kuin alkuperäisten määritelmien osalta. Prosessi on samankaltainen kuin testivetoisessa kehityksessä, mutta siinä on huomioitu erikseen vaatimuksien ja tarpeiden toteutuminen, koska vaikka ohjelma toteuttaakin halutut asiat, se voi silti poiketa käyttäjien odotuksista. Lisäyksenä prosessikaaviossa on myös määritelmien muutosvaihe, joka kohottaa prosessin sisällön kypsyyttä, sekä tuo inhimillisen näkökulman mukaan prosessiin, mikä

ilmenee siten, että muutokset dokumentteihin tuovat arvoa myös ei teknisille ihmisille ja avaavat muun muassa sen miksi vaatimukset vaihtuvat kehityksen aikana.



Kuva 10. Kehityksen prosessikaavio

Kyseisessä kuvassa 10 alkuperäisten vaatimusmäärittelyjen pohjalta voidaan jalostaa yksittäisen komponentin vaatimukset. Komponentti voi tässä yhteydessä olla myös kokonainen mikropalvelu ja sen toiminnallisuudet, sekä sen käyttämät tietomallit, mutta kokonaisen mikropalvelun toteuttaminen voi olla sen koosta riippuen erittäin aikaa vievää. Tämän takia mikropalvelun toteutus on myös hyvä jakaa pienempiin kokonaisuuksiin tarvittaessa. Vaatimuksia käytetään arkkitehtuurisuunnitelmien ja mahdollisesti muotoilusuunnitelmien pohjana, jos kyse on käyttöliittymän kehityksestä.

Kun suunnitelmat ovat olemassa, voidaan näiden dokumenttien pohjalta rakentaa komponentti, jonka jälkeen komponentin toimintaa testataan peilaten vaatimuksiin. Mikäli vaatimukset eivät toteudu, niin komponentti voidaan palauttaa kehitykseen, jolloin refaktoroinnin avulla korjataan komponentin toiminta vastaamaan määrityksiä. Jos komponentin toiminta toteuttaa määritetyt vaatimukset, mutta ei vastaakaan alkuperäiseen tarpeeseen johon komponentti on tarkoitettu, tarvitaan muutos alkuperäisiin määrityksiin.

Määritelmien muutos käynnistää dokumenttien läpikäynnin ja päivityksen, pyrkien vastaamaan miksi alkuperäisiä suunnitelmia joudutaan muuttamaan, sekä päivittämään komponentin kehitykseen liittyvät dokumentit. Pahimmassa tapauksessa muutoksella on suuri vaikutus kehitettävään järjestelmään, jolloin projektin riskit, kuten aikataulun venyminen saattavat toteutua. Tämän takia dokumenttien päivityksen yhteydessä prosessin arviointia suorittamalla voidaan reagoida ilmeneviin haasteisiin ja ennalta ehkäistä riskien ilmentymistä, kun tiedostetaan muutokset määrittelyihin ja niiden vaikutus kehitysprosessin aikana. Mikropalvelumallissa muutos yksittäiseen palveluun ei välttämättä aiheuta suurta muutosketjua kaikkiin muihin järjestelmän osiin, koska palveluiden on tarkoitus toimia omina kokonaisuuksinaan, pitämällä huoli omasta tilastaan ja välittämällä tietonsa rajapinnan kautta. Kuitenkin yksittäisen palvelun myöhästyminen voi aiheuttaa ketjureaktion muiden palveluiden kehitykseen, jos esimerkiksi tietty palvelu tarvitaan käsittelijäksi ennen muita toiminnallisuuksia. Tämän takia prosessin arviointi muutoksien yhteydessä on suositeltavaa. Lopulta järjestelmän osa voidaan hyväksyä, kun se täyttää sille asetetut määritelmät ja vastaa järjestelmän tarpeisiin.

Tietomalli johdetaan alustavasti määrittelyn perusteella. Tietomallin tarkoitus on kuitenkin toteuttaa reaalimaailman tiedon tallentaminen ja tallennettavan datan perusteella sen käsittely ohjelmallisesti. Tällöin kehityksessä ilmentyvät haasteet vaikuttavat myös tietomallien rakenteeseen. Aiempaan prosessikaavioon liittyen, arkkitehtuurikuvaus ja sen muutokset vastaavat myös usein tietomallien ja niiden havaintokaavioiden muokkaamista. Näissä tilanteissa mikropalvelumalli toimii myös edukseen, koska erilleen jaetut palvelut ja niihin liittyvä dokumentaatio voidaan erottaa toisistaan, jolloin tietomallimuutos koskee tietyn komponentin dokumentaation muokkaamista. Tämä voi vähentää muokattavien dokumenttien kompleksisuutta, sekä helpottaa muutoksien seuraamista ja auttaa tulevaisuudessa dokumentaation käyttäjää selvittämään syy- ja seuraussuhteita. Kun mietitään mikropalvelun vaikutuksia dokumentaatioon yleisesti, niin kannattaa huomioida, että monien mikropalveluiden dokumentaation hallinta voi paisua verrattuna monoliittiseen toteutukseen palveluiden määrästä riippuen. Laajoissa tapauksissa dokumenttien säilytyspaikan ja sen muokkauksen säännöistä kiinni pitämällä taataan säilytyspaikan eheys ja että se tuottaa arvoa jatkossakin eri sidosryhmille.

4.4 Toimitus

Yksinkertaisuudessaan ohjelmiston tai järjestelmän toimittamisen voi mieltää prosessiksi, jossa jokin tuote julkaistaan ja luovutetaan sen tilanneen asiakkaan käyttöön. Toimitukseen kuuluu tiedon kerääminen kaikista järjestelmän osista, ja siitä mitä tarvitaan esimerkiksi ympäristöjen pystyttämiseen, nimipalveluiden hankkimiseen ja muihin järjestelmiin yhdistämiseen. Toimitus on hyvin riippuvainen ohjelmiston tai järjestelmän toteutuksesta ja käytännössä jokaiselle voidaan määrittää omanlaisensa prosessi. Tässä työssä käsittelyn alaisena ovat mikropalvelut, joten voidaan olettaa ympäristöjen pystyttämisen olevan yksi aihealue, joka kannattaa huomioida toimituksen yhteydessä.

Tämä tarkoittaa esimerkiksi päätöksiä siitä, sijaitseeko järjestelmä esimerkiksi jollain pilvipalvelualustalla, kuten Microsoftin Azure, Amazonin verkkopalvelut (Amazon Web Services) tai Googlen pilvialusta (Google Cloud Platform), vaiko paikallisella konesalitoteutuksella. Mikropalveluiden kanssa vallitseva trendi kirjoitushetkellä on käyttää pilvipalveluita, koska mikropalveluiden hyötyjä kuten skaalautuvuutta pystytään hyödyntämään helpommin pilvessä, muun muassa lisäämällä tai laskemalla laskentatehoa yksittäisen palvelun tarpeisiin kulutuksen mukaan. Rakennettaessa pilvialustoille toteutuksia on järjestelmän toimittaminen helpompaa, kun luodaan tuotantoympäristöä järjestelmälle, koska pilvialustat tarjoavat vaihtoehtoja nopeasti pystytettäville resursseille, kuten konteille tai virtuaalikoneille.

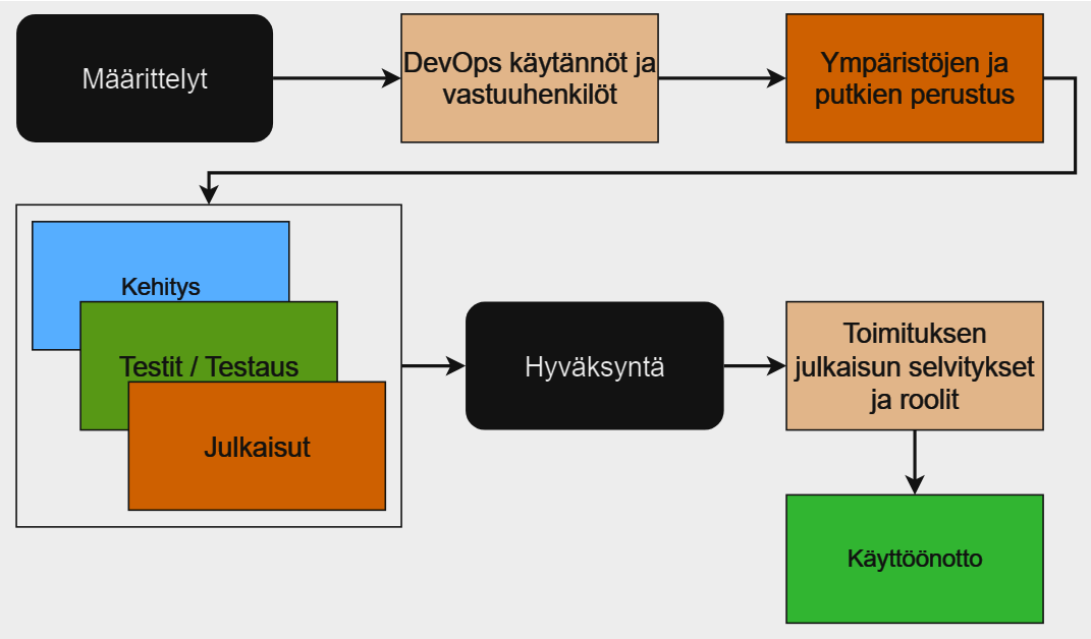
Nykyään käsitteellä DevOps, joka juontuu sanoista kehitys (engl. Development) ja operaatiot (engl. Operations), on suuri rooli ohjelmistojen kehityksessä ja niiden julkaisuissa. DevOpsin idea on yksinkertaistettuna automatisoida mahdollisimman paljon järjestelmän kehityksen osioita ja sujuvoittaa sitä määrää työtä, mikä normaalisti manuaalisesti suoritettuna vie paljon aikaa, kun muun muassa julkaistaan järjestelmien osia käyttöön kehityksen jälkeen (GitLab, Päiväämätön). DevOpsiin liittyy myös järjestelmän kunnon tarkastelu, esimerkiksi järjestelmälokien avulla, jotta voidaan olla varmoja sen toimivuudesta ja näin pystyä reagoimaan ylläpidettävään järjestelmään (GitLab, Päiväämätön). Joissakin projekteissa voi olla erikseen DevOps-insinöörin rooli, joka on vastuussa järjestelmän julkaisujen hallinnasta (RedHat, 2019. Hall, Päiväämätön.). Roolin vastuisiin kuuluu esimerkiksi DevOps-työkalujen valinta, julkaisukonfiguraatioiden hallinta, ympäristöjen hallinta, julkaisut ja tietenkin kommunikointi kehittäjien kesken. Usein DevOps-insinöörin tehtäviin kuuluu muitakin tehtäviä kuten ohjelmointikehitystä, koska DevOps-käytännöt kannattaa omata jo järjestelmää

luodessa. Takautuvasti on ainakin ennen ollut vaikeampi järjestää julkaisuun liittyviä asioita, koska yleensä aikaisemmin järjestelmiä kehitettäessä ympäristöt ovat asettuneet kehityksen alussa. Nykyään kuitenkin pilvipalveluiden mahdollistamana on helpompi pystyttää virtuaaliympäristöjä ja vaikuttaa julkaisualustoihin, mutta kun DevOps-käytännöt on leivottu järjestelmän putkeen jo alusta asti, helpottaa se esimerkiksi toimituksen taakkaa tulevaisuudessa. DevOpsin kanssa puhutaan ns. CI/CD-putkista, mikä juontuu sanoista jatkuva integraatio (engl. Continous Integration) ja jatkuva toimitus (engl. Continuous Delivery) (Redhat, 2023). Jatkuvan integraation ja toimituksen tehtävä on pitää julkaistavat päivitykset yhteensopivana ja kuten aiemmin mainittiin automatisoida julkaisuihin liittyvät tehtävät, jotta päivityksistä syntyvät julkaisut ajautuvat repositorioihin ja niiden kautta ympäristöihin, joissa niitä voidaan testata tai käyttää tuotannossa.

Miten ylläkuvatut asiat sitten vaikuttavat toimitukseen? Ensinnäkin vaikka toimituksen voi mieltää järjestelmän luovutukseksi käyttöön tuotantoympäristönä, niin tähän julkaisuun tehtävä työ alkaa käytännössä jo projektin alussa. Ottaen huomioon DevOps-käytännöt, on hyvä automatisoida mahdollisimman paljon ympäristöjen pystytykseen ja päivitykseen liittyen. Toinen asia, joka voi helpottaa käyttöönottoa mikropalveluiden kanssa on mahdollisuus julkaista tietyt palvelut porrastetusti käyttöön, jolloin pystytään tarkastamaan järjestelmän kuormitusta ja kuinka paljon käyttöönotossa nousee huomioita, yksittäisten palveluiden osalta. Huomioon täytyy tietenkin ottaa myös se, että jotkin palvelut eivät välttämättä tuota hyötyä yksin järjestelmän käytössä, vaan tarvitsevat muita palveluja tehdäkseen järjestelmästä käytettävän.

Kuvassa 11 on havainnollistettu toimitukseen vaikuttavia askeleita kaavion muodossa. Määrittelyjen ja projektin taustojen perusteella valitaan tietenkin teknologiat ja tehdään alustava arkkitehtuurisuunnitelma, jonka perusteella voidaan myös valita käytettävät ympäristöt. Näiden ympäristöjen hallinnan ympärille perustetaan DevOps-käytännöt ja vastuuhenkilöt, joilla on muun muassa oikeus tehdä julkaisuja ja päätösvalta ympäristöistä. Kun infrastruktuuri on paikallaan, voidaan edetä kehittämisen, testauksen ja julkaisujen kanssa, joka on kaaviossa yhdistetty tässä tapauksessa kokonaisuudeksi. Kun putkesta saadaan toteutuksia ulos, voidaan toteutukset hyväksyttää ja kun kaikki tarvittava käyttöönottojulkaisua varten on hyväksytetty tai jopa vähän ennen kaikkien toiminnallisuuksien valmistumista, selvitetään käyttöönoton yksityiskohdat eli onko muita julkaisualustoja tai ympäristöjä, mitä resursseja vielä tarvitaan ja miten reagoidaan

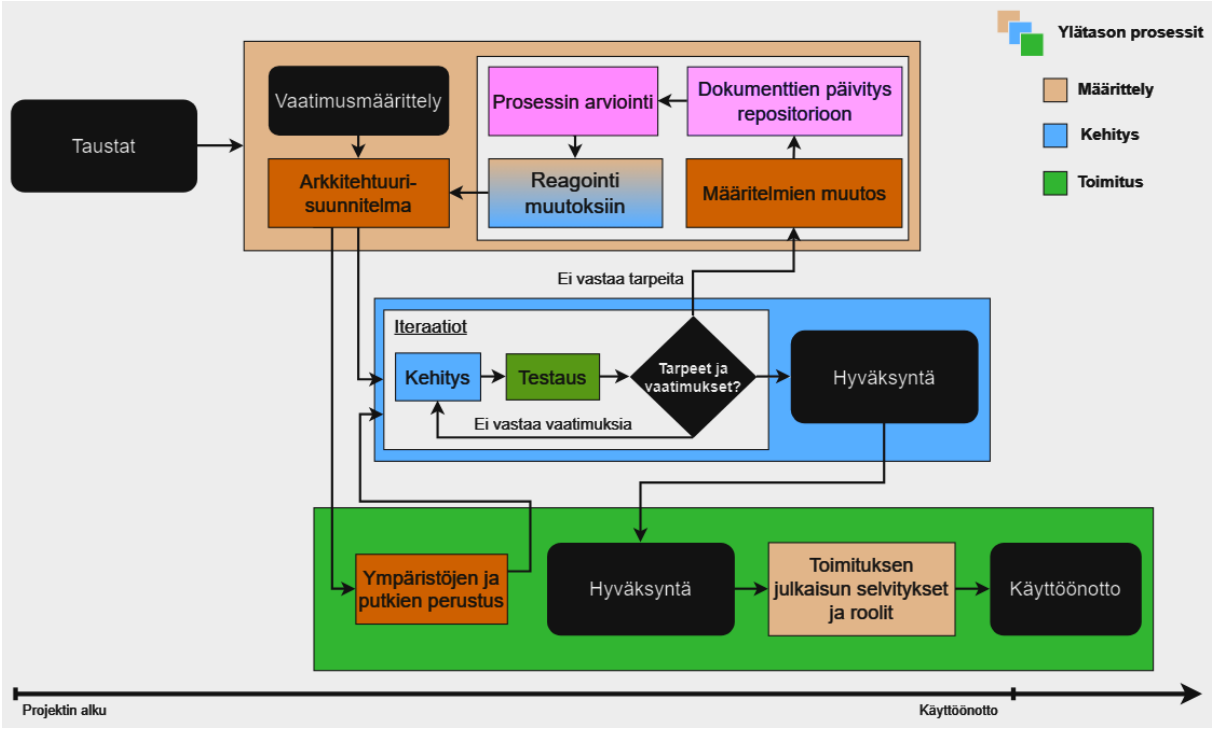
käyttöönoton yhteydessä mahdollisesti ilmeneviin haasteisiin. Tällä tavalla muodostuu toimitukseen käyttöönottosuunnitelma, jonka mukaan käyttöönotto etenee.



Kuva 11. Toimituksen prosessikuva

4.5 Prosessien yhteenveto

Jos kokoamme 4.2–4.4 aliluvuissa esitetyt kaaviot yhdeksi prosessikaavioksi, niin voimme johtaa yleisen suunnitelman mikropalveluiden kehittämisen prosessista. Tämä prosessi on kuvattu kaavioksi kuvassa 12. Kuvassa on esitetty prosessit määrittely, kehitys, sekä toimitus. Näiden aliprosessien lisäksi kuvaan on merkitty projektin taustat, joiden perusteella projekti käynnistetään. Kaavion alaosaan on lisätty jana, joka merkitsee aikaa ja ylätason prosesseina kuvatut laatikot on asetettu vastaamaan suurin piirtein niiden sijoittumista aikajanalla. Laatikoiden sisällä olevat kuvaukset kertovat yksityiskohtaisemmin, mitä jokaisessa prosessissa tapahtuu.



Kuva 12. Kuva uudelleenkehitysprojektin prosesseista

Mihin sitten kannattaa kiinnittää huomiota, jos haluamme vastata TK2:n kysymykseen: “Miten suunnitella uudistustyön prosessi vaihdettaessa monoliittisesta mallista mikropalvelumalliin?”. Kirjallisuuslähteiden ja niiden perusteella luomamme kuvan 12 avulla, voimme nostaa huomioita prosessien suunnittelusta. Ensinnäkin mikropalveluihin siirtyminen monoliittisesta arkkitehtuurimallista ei ole yksinkertainen asia, mikä todetaan muun muassa hyvin Kalskeen, Mäkitalon ja Mikkosen paperissa “Challenges When Moving from Monolith to Microservice Architecture”, joka on julkaistu vuonna 2018 (Kalske & ym., 2018.). Paperissa, kuten muissakin lähteissä, todetaan että vaihtaminen voi viedä paljon aikaa, sekä organisaation on tärkeä pystyä muuttamaan myös toimintatapojaan mikropalvelumallin ajattelun mukaiseksi.

On tärkeää ymmärtää projektiin liittyvät taustat, mitkä voivat toimia kannustimena vaihtaa arkkitehtuurimallia ja pohtia niiden kautta hyötyjä ja haittoja mikropalvelumallin puolesta. Mikäli taustojen kautta päätetään ja voidaan edetä uudistustyön osalta, niin järjestelmän määrittelyt ja niistä johdettavat arkkitehtuurisuunnitelmat, jotka käsittävät muun muassa alkuperäisen monoliitin osien jakamisen omiksi toimiviksi kokonaisuuksiin, ovat tärkeä kulmakivi prosessien, sekä tehtävän työn osalta. Niiden muuttumiseen voidaan peilata sitä, edistytäänkö kehitystyössä ja kuinka paljon resursseja, kuten aikaa ja osaamista tarvitaan uudelleenkehitysprojektin valmistumiseen.

Mikropalveluiden kehityksessä ketteristä kehityssykleistä vaikuttaa olevan enemmän hyötyä toisin kuin esimerkiksi vesiputousmallista, koska se mahdollistaa reagoinnin muuttuviin määritelmiin ja täytettäviin vaatimuksiin, joita kompleksisessa toteutuksessa voi ilmentyä paljon. Tosin, jos mikropalveluiksi jaon suunnitelma on tarpeeksi yksityiskohtainen ja järjestelmän toimialan tuntemus, sekä kokemus mikropalveluiden tekemisestä on korkealla tasolla, on mahdollista käyttää lähestymistapaa, jossa rakennetaan mikropalvelu kerrallaan toteutusta julkaisten yksittäisiä osia järjestelmästä.

Prosessien suunnittelussa on hyvin paljon kiinni siitä, millaiset lähtökohdat projektilla on, sekä mihin käytäntöihin osallistuvat sidosryhmät ovat tottuneet. Kuitenkin esimerkiksi tämän työn kirjoituksen aikana kuvan 12 mukainen prosessi tai ainakin variaatiot siitä vaikuttavat olevan käytettyjä malleja vastaavissa uudelleenkehitysprojekteissa. Kuva 12 ja aiemmissa neljännen luvun aliluvuissa käsitellyt kuvat (8, 10, 11) ovat ehdotuksia prosessien suunnittelusta. On hyvä asia tiedostaa, että erilaiset prosessit voivat muuttua hyvinkin paljon projektin aikana erilaisien syiden takia. Tämän takia tässä työssä on nostettu dokumenttivetoinen kehitys erityisesti esille, koska sen tavoitteena on mitata konkreettisesti muutoksia ja pystyä reagoimaan niihin. Huomioimalla mikropalveluiden kompleksisuuden, sen tuottamien teknisten ja organisaationaalisten haasteiden, sekä luomalla reagointikanavia ilmeneviä haasteita varten vaikuttaa olevan avainasemassa prosessin suunnittelussa vaihdettaessa monoliittisestä palvelusta mikropalvelupohjaiseen järjestelmään.

5 CASE: Toiminnanohjausjärjestelmä

Tämän työn kirjoittamisen hetkellä on kirjoittaja ollut mukana projektissa, jossa vanha toiminnanohjausjärjestelmä on käynyt läpi uudelleenkehitysprosessin. Seuraavien kappaleiden on tarkoitus avata lukijalle uudelleenkehitysprosessia, missä vanha monoliittinen järjestelmä on korvattu mikropalveluarkkitehtuurilla. Lisäksi tarkoituksena on verrata työssä esiintyvän kirjallisuuden, sekä niiden perusteella tehdyn pohdinnan kautta kokemuksia oikeaan projektiin nähden. Tällä tavalla pyrimme vahvistamaan tai jopa kiistämään kirjallisuudessa esiintyviä nostoja, sekä voimme peilata tässä työssä jo johdettuja vastauksia tutkimuskysymyksiin TK1 ja TK2. Lopuksi tämän läpikäynnin pohjalta vastaamme viimeiseen tutkimuskysymykseen TK3: “Millä prosesseilla sekä toimenpiteillä saadaan olemassa olevat datamallit muutettua mikropalveluarkkitehtuuria tukeviksi?”.

Koska työssä esimerkkinä esiintyvä projekti sisältää toiminnanohjausjärjestelmää käyttävän organisaation tai organisaatioiden toimintatapoja ja immateriaalioikeuksia, sekä sitä kehittävän tahon immateriaalioikeuksia, on siihen liittyvät yksityiskohdat jätetty tuntemattomaksi, jotta vältetään näiden, sekä mahdollisten organisaatiokohtaisten salaisuuksien loukkaaminen. Kuvaus pidetään mahdollisimman yleisellä tasolla, sekä esiintyvät esimerkit muun muassa tietomalleista voivat olla havainnollistavia.

5.1 Kuvaus

Kuten aiemmin mainittiin, esimerkissä on kyseessä toiminnanohjausjärjestelmä eli ERP. ERP, joka tulee englannin kielen sanoista Enterprise Resource Planning, on ohjelmistojärjestelmä organisaation toimintojen hallitsemiseen (Microsoft Dynamics, Päiväämätön). ERP on usein suuri kokonaisuus, joka käyttää organisaation hallinnoimaa dataa esimerkiksi inventaarion, kaupankäynnin, sekä henkilöstöratkaisujen hallintaan. Kyseinen ohjelmisto, joka on tarkoituksena päivittää vastaamaan nykyajan tarpeita, on monen vuosikymmenen ajalla kehittynyt monoliittinen järjestelmä, jonka kehityksestä ja ylläpidosta on vastannut erillinen organisaatio, kuin sitä käyttävä tai käyttävät tahot. Vaikka alkuperäinen järjestelmä on pysynyt toimintakuntoisena jo vuosikymmeniä, tekninen velka on kasvanut suureksi rasitteeksi ja vanhan teknologian haasteet alkavat vaikeuttamaan ylläpidosta ja kehityksestä vastaavien henkilöiden tehtäviä, kun järjestelmään haluttaan muuttaa. Tämän lisäksi voidaan huomioida, että vuosikymmeniä jatkunut kehitys on saanut aikaan tilanteen, jossa järjestelmän yksityiskohtia tai niiden toimintaa ei tunneta täydellisesti, koska tiedettävien asioiden määrä

on paisunut erittäin suureksi ja vaikeaksi hallita, sekä dokumentoida riittävällä tasolla. Myös henkilöstön vaihtuvuus on tietenkin vaikuttanut asiaan, koska vuosien varrella järjestelmän kanssa on ollut tekemisissä moni asiantuntija.

Ylläpidosta ja kehityksestä vastaava organisaatio eli toimittaja on ohjelmistoa käyttävien tahojen eli tilaajien kanssa käytyjen keskustelujen pohjalta tarjonnut suunnitelman uudelleenkehitystä varten, jossa järjestelmä on jaettu mikropalveluihin. Ylätason tavoitteena toimittajalle päivityksessä on kyse pystyä tarjoamaan tulevaisuudessakin järjestelmän vakaa toiminta, päivittämällä järjestelmän arkkitehtuuri ja teknologiat vastaamaan nykymaailman standardeja. Tämän lisäksi toimittajalla on tietenkin taustalla myös taloudellinen houkutus, koska uudelleenkehitys mahdollistaa myös tilanteen, jossa järjestelmästä voidaan muokata kokonaisvaltaisempi myös muiden mahdollisten tilaajien käyttöön. Nykyisen ohjelmiston ikä on aiheuttanut muun muassa sen, että massiiviset datan käsittelyt ovat koko ajan hankalampia järjestää ja näin datan eheys kärsii, taustalla olevan infrastruktuurin takia on vaikeampi skaalata ylöspäin esimerkiksi palvelintehoa ja uhka teknologioiden tuen loppumisesta häämöttää horisontissa.

Tilaajan osalta tavoite on sama kuin toimittajalla, mutta tulevaisuuden toimivuuden lisäksi tilaajan toiveena on saada selvästi järjestelmäuudistuksen avulla vikasietoisempi järjestelmä ja hakea myös taloudellisia edellytyksiä, kuten säästöjä, jotka liittyvät ylläpitokustannuksiin, mitkä syntyvät vanhan järjestelmän monimutkaisuudesta ja koosta, sekä ympäristöstä, jossa ohjelmisto toimii. Tilaajalla on varsin hyvä edellytys tehdä nykyisen toimittajan kanssa yhteistyötä jo olemassa olevan toimittajan kanssa, koska kontekstiosaaminen on hyvin paljon nykyisen toimittajan käsissä ERP:iin ja sen prosesseihin liittyen.

5.2 Kokemuksia

5.2.1 Projektin kulku

Alun perin uudelleenkehitysprojekti on käytännössä alkanut ylätason arkkitehtuurisuunnittelusta, arkkitehtuurikaavioista, sekä käyttöliittymäuudistuksen suunnittelusta. Tämä tarkoittaa sitä, että aikaisessa vaiheessa on tehty hahmotelmat, joiden perusteella teknologiavalinnat on pystytty perustelemaan ja minkä takia esimerkiksi puolletaan mikropalveluarkkitehtuurin käyttämistä uudelleenkehityksessä. Arkkitehdit ja käyttöliittymäsuunnittelijat ovat tutustuneet vanhemman toiminnanohjausjärjestelmän toimintaan, sekä heillä on ollut konsulttiapuna aiemman järjestelmän ylläpidosta vastaavia

henkilöitä. Tällä tavalla on pyritty vaikuttamaan toimialan tuntemukseen, koska kaikki suunnittelevat henkilöt eivät ole välttämättä aikaisemmin olleet tekemisissä kyseisen toimialan kanssa, johon järjestelmä kytkeytyy. Toimialan tunteminen on erittäin arvokasta, koska tämä helpottaa esimerkiksi palveluiden jakamista mikropalveluihin ja selventää mitä alustaratkaisuja on tarvittu tilaajan käyttöön.

Kun tehdyt ylätason suunnitelmat on hyväksytty, on sen jälkeen suunnittelun ohella tuotettu määrittelydokumentteja. Nämä dokumentit kokosivat vaatimuksia vanhan järjestelmän kautta, luoden pohjaa sille, mitä uuden järjestelmän pitää tukea. Määrittelyjä tehtiin yhdessä tilaajan kanssa, koska heillä on käsitys siitä, miten oikean maailman tilanteet hoituvat yhdessä järjestelmän käytön kanssa. On hyvä huomioida, että määrittelyt eivät olleet täysin aukottomia kertomuksia, sekä määrittelyä on jatkettu hyvin pitkälle projektin edetessä, osin muutostarpeista edellisiin vaatimuksiin ja myös uusien vaatimusten ilmentyessä. Määrittelyt ovat ensisijaisesti vaikuttaneet esimerkiksi käyttöliittymäuudistukseen näkyvästi, mutta samalla on projektissa koitettu parantaa olemassa olevan järjestelmän heikkoja kohtia ja näin suuria muutoksia on ajettu taustalla toimiviin järjestelmän osiin, vaikka se ei loppukäyttäjille suoraan näkyisikään. Käyttöliittymän suunnittelijat ovat kertomuksien ja vaatimusten perusteella työstäneet työkalujen näkymiä, sekä pyrkineet tekemään mahdollisimman helposti käytettäviä rakenteita tuleville käyttäjille. Nämä suunnittelu- ja määrittelydokumentit ovat käytännössä muodostaneet projektissa vaatimusmäärittelyn ytimen.

Merkityksellistä on myös huomioida projektin sidosryhmät. Toteuttajan puolelta on ollut mukana vaihteleva määrä henkilöstöä projektin alusta alkaen. Suurimmillaan projektiryhmä on koostunut monesta määrittelijästä, käyttöliittymäsuunnittelijasta, kehittäjästä, testaajasta, arkkitehdistä, sekä projektipäälliköistä samaan aikaan. Suuresta henkilömäärästä on ollut hyötyä, mutta myös haasteita, näistä enemmän tulevissa kappaleissa. Kehittäjät on pääasiallisesti voinut jakaa front-end ja back-end tekijöihin, sekä arkkitehtuurista vastaavia henkilöitä on ollut mukana back-end toteutuksessa. Ympäristöjen pystyttämistä on ollut molemmilla osapuolilla, mutta suurempi työ infrastruktuurin luomisessa on selvästi ollut back-end puolen osaajilla. Kehittäjiä on ollut melkein projektin alusta asti mukana, mutta on myös tarvittaessa liitetty uusia kehittäjiä projektiryhmään, sekä osa tekijöistä on siirtynyt muihin tehtäviin tarpeen vaatiessa. Vaihtuvuutta on siis esiintynyt toimittajan puolella projektin aikana.

Tilaajalla on oman alan asiantuntijoita eli organisaatioihin kuuluvia henkilöitä sidottuna projektiin arviolta noin 10 kappaletta vaihtelevasti, tietenkin taustalla vaikuttaa tilaajan puolella enemmän henkilöitä epäsuorasti. Nämä asiantuntijat omaavat kokemusta järjestelmän toiminnasta omilta osa-alueiltaan, esimerkiksi henkilö voi olla vastuussa laskutuksen tai inventaarion hallinnasta vanhan ERP:in kautta. Tilaajan puolelta vaihtuvuus on vaikuttanut selvästi vähäisemmältä, mutta koska asiantuntijoilla on muitakin työtehtäviä, he eivät ole täyspäiväisesti olleet kiinni uudelleenkehityksessä, joten tämä on tuottanut myös haasteita ainakin alkuperäisiä määrityksiä tehdessä, koska kaikki keskittyminen ei ole voinut olla suunnattuna uudelleenkehitykseen.

Alkuperäiset prosessit ovat tuottaneet näiden taustatietojen kautta tietyn määrän dokumentteja, joiden pohjalta uudelleenkehitys on aloitettu. Kuten aiemmin tässä luvussakin mainittiin, dokumenttien kattavuus ei ole ollut täysin absoluuttisen tarkka, mikä heijasteli mahdollisia kipupisteitä ensimmäisissä prosesseissa, joista ketteriin sprintteihin johdettiin tehtävää työtä. Projektin alkupäässä infrastruktuurin kehikon muodostaminen vei suuren panoksen varsinkin back-end kehittäjien ajasta ja tämä vaikutti suoraan päätoimintojen kehityksen viivästymiseen. Kuten luvussa 3.2 mainitsimme mikropalveluiden kompleksisuudesta, on pelkästään infrastruktuurin pystyttäminen hyvä esimerkki siitä, että tarvitaan suuri työpanos mikropalveluarkkitehtuurissa kompleksisuuden selättämiseen tai hallitsemiseen.

Mitä kauemmin projekti on ollut käynnissä, sen paremmiksi prosessit ovat yleisesti ottaen muodostuneet. Tämä on tietenkin tarkoitus jo esimerkiksi scrumin seremonioissa, joissa tiimin on tarkoitus olla itseohjautuva, sekä pystyä sprinttien jälkeiseen reflektointiin. Prosessit ja niistä keskustelu ovat selvästi parantuneet, varsinkin kun kohtalaisen pitkän ajan jälkeen scrummasterin rooli eristettiin paremmin muista kehittävistä rooleista. Tavallaan tiimillä oli käytössään scrummasteri koko projektin ajanjakson aikana, mutta koska rooli oli alun perin käytännössä toisena roolina jonkun muun harteilla, vaikutti tämä selvästi siihen, kuinka paljon ohjausta tiimi sai ja pystyi parantamaan omia prosessejaan, pelkästään niistä keskustelun sijaan.

Muita haasteita liittyen prosesseihin, on ollut esimerkiksi kaikkien pitäminen muutoksien mukana. Tämä on näkynyt kehityksessä, mutta myös kommunikaatiossa tilaajan ja toimittajan välillä. Esimerkiksi suuret määrät iteraatiokierroksia varsinkin projektin alussa oli mahdollista välttää jälkikäteen ajatellen tarkentamalla määrittelyjä ja vaatimuksia, koska usein kehityksen

alle otettiin varsinkin projektin alussa asioita, joista voitiin tehdä karkeasti ottaen puoliksi tai melkein kokonaan ominaisuudet loppuun. Kestävämpään asetelmaan voi päästä myös avaamalla työmäärää mikropalvelujen osalta, jolloin keskittyminen tiettyihin ominaisuuksiin loisi vakaamman pohjan tulevaisuuden kehitykselle. Vaikka projektissa on esiintynyt haasteita, on yleishenki pysynyt hyvänä, mikä on tarkoittanut pidemmässä juoksussa, että jokaisessa sprintissä on saatu asioita aikaiseksi, eikä työ varsinaisesti ole pysähtynyt missään kohtaa ainakaan kehittävän tiimin osalta.

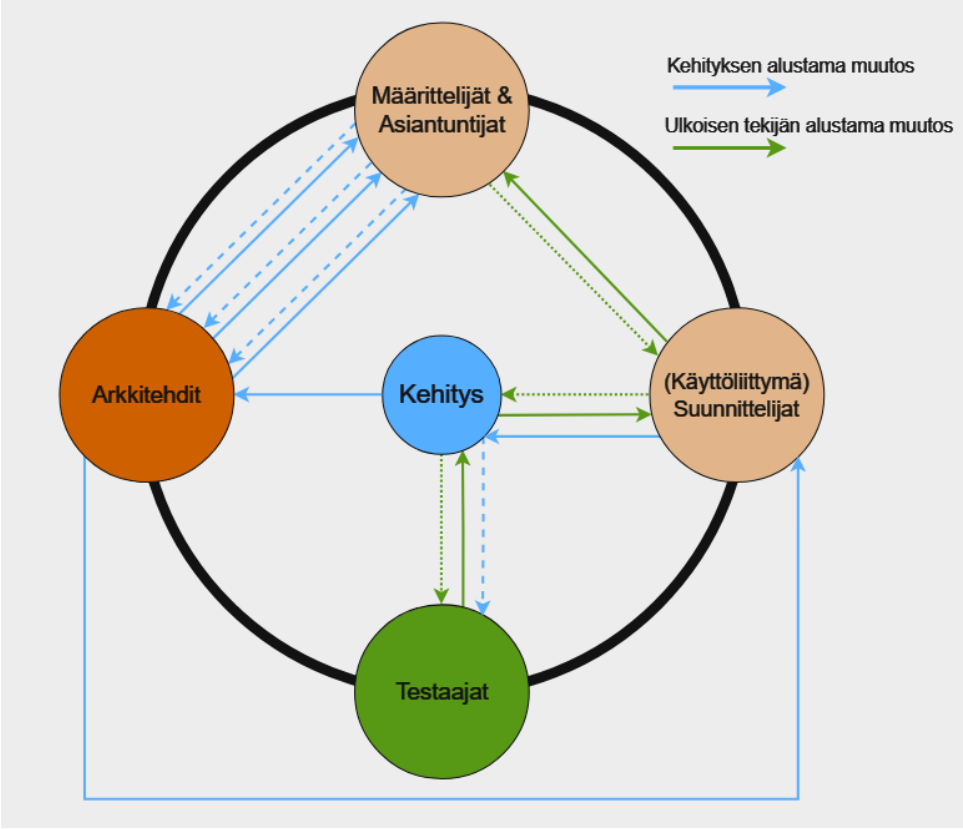
5.2.2 Kehitys ja tietomallin muodostaminen

Jos avataan tarkemmin mikropalveluiden kehitystä ja miten kehityksen prosessit ovat edenneet ja kehittyneet ajan kuluessa, niin tärkeäksi asiaksi on noussut resurssien keskittäminen ja yhteistyön kohentaminen. Tähän on vahvasti sidoksissa myös tietomallien kypsyminen toimialatuntemuksen ja muiden sidoksien ymmärryksen kasvaessa.

Uudelleenkehitysprojektissa aloitettiin periaatteessa kehityksen osalta täysin puhtaalta pöydältä. Yleinen ohjelmistokehityksen kokemus kehitystiimissä vaihteli muutamista vuosista vuosikymmeniin asiantuntijoiden välillä kehitykseen osallistuvilla henkilöillä. Kokemusta käytetyistä teknologioista oli taustalla muista projekteista, mutta projektien skaala on ollut vaihteleva verrattuna esimerkin toiminnanohjausjärjestelmän uusimiseen. Kehitysvaiheen voi katsoa alkaneen ensimmäisten ympäristöjen pystytyksessä, koska DevOps-putket vaativat omat ohjaustietonsa, sekä niin sanotusti putkien päät toimiakseen, on ensimmäiseksi luotu GIT-repositoriot. Repositorioilla hoidettaan ohjelmistojen versionhallinta ja niihin luotiin ohjelmistojen alkukantaiset versiot. Näin ensimmäisten kehityssprinttien aikana pystytettiin puitteet sille, että kaikki mikropalvelut omaavat ensimmäiset versionsa, jotka on kytketty toisiinsa rajapinnan kautta ja niiden julkaisu toimii DevOpsin mukaisia putkia pitkin.

Kun alla oleva infrastruktuuri oli tarpeeksi valmis kehitystyön aloittamiselle, jatkettiin sekä käyttöliittymän, että loppukäyttäjille näkymättömien taustaohjelmien parissa. Olemassa olevaa materiaalia kehityksen aloittamiseksi oli jo luotu, mutta varsinkin projektin alussa huomattiin, että monet määrittelyistä eivät olleet tarpeeksi tarkkoja mikropalvelukonseptiin nähden. Toisin sanoen, haasteeksi muodostui kehityksessä usein saada järjestelmän kokonaisuuksia valmiiksi tai kehittäminen oli arvioitua hitaampaa, koska reagointi muuttuviin vaatimuksiin ja määrittelyihin ei ollut tarpeeksi tehokasta. Mikä siis heikensi reagoinnin tehokkuutta yleisesti? Alla olevassa kuvassa 13 havainnollistetaan määrittelyn muutoksen aiheuttamaa kuormaa ja askelia muutoksien hyväksymiseen kehityksen näkökulmasta, jossa

kehitys on tarkastelun keskiössä. Jotta saadaan käsitys, miksi määrittelyn muutoksilla on suuri vaikutus tehokkuuteen, tarkastellaan kuvassa kahta esimerkkiä eli reaktioketjua, jotka on merkitty vihreällä ja sinisellä värillä. Kuvassa yhtenäiset nuolet kuvastavat kommunikointia ja muutoksiin reagointia, kun taas katkoviivan omaavat nuolet ilmentävät vastauksia johonkin reaktioon. Vihreä ketju alkaa testauksen huomiosta ja sininen ketju taas kehityksen nostamasta tarpeesta.



Kuva 13. Hahmotelma määritelmien muutoksista

Kuvan 13 kautta pyritään visualisoimaan reaktioiden määrää ja siitä johtamaan tehokkuuden laskemisen syitä. Tarkastelemalla esimerkiksi vihreällä värillä merkattua reaktioketjua, voidaan huomata, että testaajien nostama huomio kehittäjälle voi aloittaa ketjun muutostarpeen ilmentyessä, joka taas työllistää monta sidosryhmää. Koska kehittäjät pyrkivät täyttämään heille annetut määrittelyjen kautta esiintyvät vaatimukset rakentaessaan järjestelmän osia, voi jokin määrittelyjen mukainen toiminto olla ristiriidassa halutun lopputuloksen kanssa. Tämä korostuu tilanteissa, jossa poiketaan vanhan järjestelmän toimintatavoista, kun on haluttu parantaa uuden järjestelmän lähtökohtia. Koska kehittäjillä ei ole valtuuksia poiketa sovituista suunnitelmista, ajaa tämä kehittäjän siirtämään reagoinnin eteenpäin esimerkiksi suunnittelijalle, jonka täytyy esittää muutokset asiantuntijoille, jotka tässä tapauksessa toimivat myös suunnitelmien hyväksyjinä. Tällainen ketju yksittäisenä

esiintyessään ei pienissä määrin ole haitallista, mutta kun samanlaisia ketjuja syntyy monia ja esimerkiksi kuvan 13 sinisellä piirretyssä ketjussa asiat voivat jäädä pallottelemaan sidosryhmien välillä, syntyy melkein kestämätön tilanne, jossa kasaantuvat muutostarpeet tukkivat kehityksen edistymisen ja syövät kaikkien sidosryhmien aikaa. Sitten kun huomioidaan vielä ajan vaikutus, jossa mahdolliset päätökset jäävät pitkäksi aikaa pyörimään paikoilleen yksittäisen sidosryhmän keskuudessa, on todella haastavaa saada kiinni halutuista aikatauluista, puhumattakaan toimintakohtaisiin tavoitteisiin pääsemisestä.

Tietomalli, josta oli alustava käsitys taustaselvityksen ja ensimmäisten määrittelyjen pohjalta, kehittyi näiden muutosreaktioiden kautta tarkemmaksi. Ensimmäisenä virallisena mallinnuksena voidaan pitää kaaviota, joka käsitti järjestelmän kokonaisarkkitehtuurin, kyseisen kuvan tarkoituksena hahmottaa varsinaisen järjestelmän jako mikropalveluihin, sekä järjestelmään liitetyt integraatiot muihin tarpeellisiin järjestelmiin. Tämän ajatuksen ja vanhan käytetyn ohjelman tietomallien perusteella johdettiin arkkitehtuurisia hahmotelmia. Käytännössä tietomallien muodostuminen näkyi siten, että kartoittamalla vanhan järjestelmän toimintaa saatiin toimialatietämyksen prosesseja hahmoteltua karkeiksi UML-kaavioiksi, joista taas johdettiin mitä tietokenttiä tarvitaan prosesseja varten. Tämän ohella koitettiin eliminoida tai siirrellä kenttiä tietorakenteesta toiseen, optimaalisemman käytön varmistamiseksi.

5.3 Kirjallisuus vs kokemus

Tämän alaluvun tarkoituksena on verrata tutkimuskysymysten TK1 ja TK2 vastauksia projektissa koettuihin asioihin. Tältä pohjalta johdetaan vastaus myös kysymykseen TK3. TK1 ja TK2 vastaukset perustuivat kirjallisuuslähteiden tarkasteluun ja tulkintaa, jolloin projektista saatu data antaa pohjan kirjallisuuden, sekä kerätyn kokemuksen vertailuun. Vastaus tutkimuskysymykseen TK3, eli “Millä prosesseilla sekä toimenpiteillä saadaan olemassa olevat datamallit muutettua mikropalveluarkkitehtuuria tukeviksi?”, pyritään antamaan listana suosituksia, mitkä syntyvät kirjallisuuden ja kokemuksen yhteisvaikutuksesta. Jokaisen käsittely on jaettu omiin alalukuihinsa.

5.3.1 TK1: Monoliitista mikropalveluarkkitehtuuriin

Tutkimuskysymyksessä TK1 vastattiin kysymykseen: “Milloin vaihtaminen mikropalvelumalliarkkitehtuuriin on paras vaihtoehto vanhan monoliittisen järjestelmän uudistusprojektissa”. Vastauksena loimme listan kriteereistä, joiden perusteella voimme

verrata esimerkkiin toiminnanohjausjärjestelmän uudistusprojektista ja tarkastella siten projektin näkökulmasta toteuttaako se luodut kriteerit.

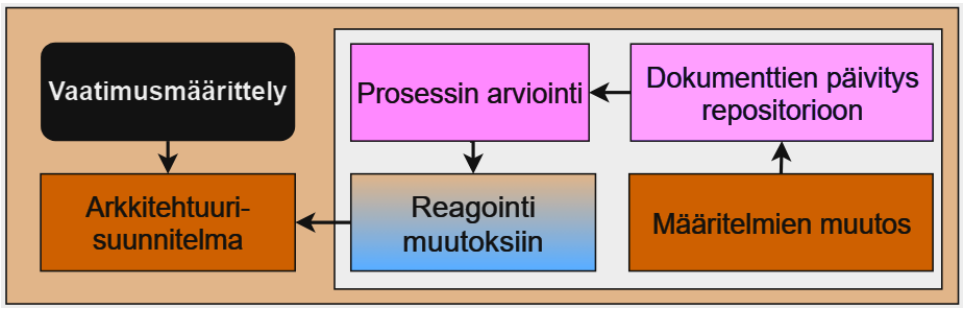
Kuten luvussa 5.1 alustimme, vanha toiminnanohjausjärjestelmä oli ollut käytössä jo monia vuosia, sekä lähdekoodin määrä oli valtava. Vaikka järjestelmän päivittäminen oli mahdollista, pienemmätkin päivitykset tai korjaukset saattoivat vaatia suuren työmäärän pelkästään uusien virhetilanteiden välttämiseksi ja käytännössä julkaisujen tekeminen, sekä päivitysten julkaiseminen oli ajoittain erittäin työlästä. Tässä törmäämme ensimmäisiin kriteereihimme, eli järjestelmän lähdekoodin määrä on suuri, järjestelmän elinkaari on pitkä, sekä sivutaan elinkaaren pituuden takaamista sujuvamman ylläpidon kautta. Mikropalvelumallissa tämä suuri määrä lähdekoodia voidaan pilkkoa pienemmiksi palasiksi ja näin lisätä koodin ymmärrettävyyttä, sekä tarvittaessa tulevaisuudessa julkaisut voidaan hoitaa mikropalvelukohtaisesti. Julkaisemalla komponenttikohtaisia päivityksiä luodaan pohja sille, että muutokset saadaan nopeammin käyttöön ja mikropalveluihin jaettuun järjestelmään voidaan liittää uusia ominaisuuksia helpommin. Kehityksestä vastaavien henkilöiden määrän osalta vanha järjestelmä tarvitsi jo ylläpidettäessä kohtuullisen määrän henkilöitä varmistamaan järjestelmän toimivuus ja kehitys. Vaikka uudistusprojektin kohdalla kehitystiimin henkilöiden määrä on vaihdellut, niin voidaan puhua siitä, että kehittäjiä on ollut mukana paljon kehitysvaiheessa, kun taas tulevaisuuden ylläpitovaiheessa luultavasti oletetaan pienemmän määrän riittävän automaatioiden avulla.

Vertaamalla luomaamme listaukseen, voimme todeta, että mikropalvelumalli on vähintäänkin hyvä, jos ei paras vaihtoehto, esimerkin projektin uudelleenkehityksessä. Mikropalvelumalli on toki aiheuttanut haasteita, pääasiassa kompleksisuuden ja ympäristöjen osalta. Esimerkiksi projektiin kuuluvilta henkilöiltä on tullut kipuilua ajettavien ympäristöjen määrästä, sekä rakenteiden monimutkaisuudesta, mutta toimiessaan voidaan mikropalveluarkkitehtuurilla varmistaa komponenttien ylläpito ja kattavammat muokkausmahdollisuudet tulevaisuudessa.

5.3.2 TK2: Kuinka suunnitella prosessit?

Tutkimuskysymyksessä TK2 pyrimme vastaamaan kysymykseen: “Miten suunnitella uudistustyön prosessi vaihdettaessa monoliittisesta mallista mikropalvelumalliin?”. Tuloksena loimme kaavion (kuva 12), jonka on tarkoitus esittää prosessia ylätasolta katsoen kirjallisuuden pohjalta. Luvussa 5.2.2 nostimme esiin varsinkin kuvassa 13 ja siihen liittyvissä kappaleissa ilmentynyttä haastetta, jossa prosessien suoritus varsinkin määritelmien osalta kärsi liian monista askelmista, kun haluttiin saada päätöksiä aikaiseksi. Jos tarkastelemme

tarkemmin kuvasta 12 erotettua määrittelyn kaaviota kuvassa 14, niin tähän osuuteen on hyvä puuttua projekteissa, joissa vaihdetaan monoliittisesta mallista mikropalvelumalliin.



Kuva 14. Määrittelyn prosessi

Ensinnäkin, vaikka järjestelmän tarkoitus uudelleenkehitysprojektin jälkeen on toimia täysin samalla tavalla kuin vanha järjestelmä, teknologiset erot ja palveluiksi jakautuneet ohjelmiston osat voivat silti muuttaa alla olevaa teknistä toteutusta hyvinkin paljon. Tämä pätee vielä enemmän projektissa, jossa koitetaan esimerkiksi parantaa järjestelmän toteutusta jo uudelleenkehitysvaiheen aikana. Jos muutoksiin reagoiminen on hidasta ja/tai niille ei saada nopeaa hyväksyntää, viivästyttää tämä kaikkien osapuolien työtä, koska kyseinen toimintamalli aiheuttaa kasautuvaa kuormaa, jota ei pystytä purkamaan yhtä nopeasti kuin sitä syntyy. Toiseksi, jos hyväksytyistä muutoksista ei jää jälkeä mihinkään, voi projektin sidosryhmillä olla eriäviä käsitys mitä halutaan saada aikaiseksi. Jos nämä ajatusmallit ovat ristiriidassa esimerkiksi toimittajan ja tilaajan kesken, aiheutuu tästä lisää ratkottavaa, yleensä ylimääräisten tapaamisten muodossa.

Siksi jos halutaan pystyä seuraamaan muutoksien määrää, sekä poistamaan iteraatiokierroksia kuvassa 14 esitetyssä kaaviossa on tärkeää saada muutokset dokumentoitua, arvioida kokonaisuudessaan muutosten aiheuttamaa kuormaa arvioimalla prosesseja ja pystyä oikeasti reagoimaan tapahtuviin muutoksiin. Reagoinnin osalta voi antaa esimerkiksi enemmän valtaa kehittäjälle poiketa sovituista asioista, tietenkin hyvin perusteltuna, tai saada tarvittavat sidosryhmät käymään läpi muutostarpeet mahdollisimman nopeasti, että asiat eivät jää roikkumaan. Prosessien vaiheet kannattaa avata kaikille sidosryhmille, sekä sidosryhmien kuuluu pystyä täyttämään omien rooliensa tehtävät parantaakseen prosessin kulkua.

Jos vielä vertaamme hiukan kirjallisuuden pohjalta luodun kaavion, sekä esimerkkiprojektin kokemuksia suoraan, niin kun tätä työtä on kirjoitettu projektin aikana, prosessit ovat muokkaantuneet melko paljon kohti kaavion formaattia verrattuna projektin alkuperäisistä prosesseista. Sekä tilaaja, että toimittaja ovat nähneet ongelmia saada asioita valmiiksi

sprinttien aikana ja huomiota on kiinnitetty prosesseihin ja otettu askelia prosessien sulavoittamiseksi. Erillisenä nostona haluamme kiinnittää huomion myös siihen, että projektin rooleilla ja niiden määrällä per henkilö on suuri vaikutus asioihin reagoimiseen. Projektissa on selvästi ollut muutamia hetkiä, jolloin tuottavuus on parantunut ainakin toimittajan näkökulmasta rooleihin liittyvissä kysymyksissä. Ne ovat olleet roolien tehtävien tarkentaminen tai roolijakojen parantaminen. Esimerkiksi mainitsimme kappaleessa 5.2.1, jossa scrummasterin rooli erotettiin kehittävistä rooleista ja tämä mahdollisti paremman ohjauksen, sekä tietyllä tavalla kehityksen ulkopuolisen suunnan säilyttäjän. Jos henkilöillä on monia rooleja hoidettavanaan, niin usein lopputulos on se, että roolien tehtävät hoidetaan kaikki huonommin kuin keskittyessä yhteen rooliin tai yhden pääroolin tehtävät hoituvat hyvin, mutta muut tarvittavat tehtävät jäävät huomiotta. Roolien määrän kasaantuminen kasvattaa myös riskiä olla riippuvaisia vain yhdestä henkilöstä, jota kannattaa välttää varsinkin, jos projektin dokumentaation taso on huono.

5.3.3 TK3: Toimenpiteet tietomallien avustajaksi

Viimeisenä haluamme vastata tutkimuskysymykseen TK3: “Millä prosesseilla sekä toimenpiteillä saadaan olemassa olevat datamallit muutettua mikropalveluarkkitehtuuria tukeviksi?”. Tähän kysymykseen ei ole yksiselitteistä ratkaisua, mutta kirjallisuuden ja kokemuksen perusteella voimme alustaa toimintamalleja, joita seuraamalla tietomallien käsittely mikropalveluja varten helpottuu.

Toimialatuntemuksen, eli kokemuksen ja tiedon aihealueelta, merkitys on valtava. Mitä parempi käsitys mallintajilla on esimerkiksi oikean maailman prosesseista, sitä kattavammin voidaan ottaa huomioon tallennettava data ja käytettävät dataelementit. Uudistusprojektissa kannattaa käyttää mahdollisimman paljon hyödyksi saatavilla olevaa informaatiota vanhan järjestelmän toiminnoista ja keskustella henkilöiden kanssa, jotka ovat olleet tekemisissä alkuperäisten prosessien kanssa. Tästä kokemuksesta voidaan kartoittaa muun muassa minkä takia jotain tiettyjä ratkaisuja on ennen tehty ohjelmiston suhteen. Esimerkin uudelleenkehitysprojektissa on usein luotu tietorakenteille kehys, mutta kehityksen edistyessä tarkennukset ovat olleet tarpeen ja usein on ollut kätevää peilata myös vanhaan kokemukseen.

Toinen tietomallien muodostumista auttava prosessi ovat itseasiassa kehityksen prosessit, kun tietomallia ei ole suunniteltu tarkasti valmiiksi etukäteen. Jos kehityksen prosessi on tarpeeksi tarkka, sekä siihen on sisällytetty ongelmatilanteisiin reagoiminen ja muutoksien vaikutusten arvioiminen, niin se luo pohjan tietomallin kehittämiselle, missä muutokset eivät kasaannu

padoksi, joka täytyy purkaa. Esimerkin projektissa on ollut ajoittain ongelma, että tiettyihin muutoksiin ei olla reagoitu tarpeeksi nopeasti, joka pitkällä aikavälillä vaikuttaa muihin ohjelmiston osiin. Tässä kannattaa kiinnittää huomiota toteuttajan, sekä tilaajan puolelta, niin että asiat tuodaan mahdollisimman nopeasti esille ja että tilaajan puolelta halutaan saada ratkaisu aikaiseksi, sekä pystytään tekemään päätös esitetyn materiaalin pohjalta. Tähän liittyy myös kokonaisuuksiin keskittyminen ja toimivuuden varmistaminen. Vaikka sidosryhmien jäsenille voi olla tärkeää, miltä lopputulos näyttää visuaalisesti, pitää silti tarkastella objektiivisesti ohjelman toimintojen toimivuutta ja priorisoida ohjelmistoprosessien valmistumiseen. Esimerkiksi liiallinen visuaalisten yksityiskohtien nostaminen luo ylimääräistä työtä, joka usein voidaan käyttää muun toiminnallisuuden parantamiseen. Iteratiivisessa prosessissa scrumtiimin pitää pystyä tekemään tehtäviä loppuun ja poimia vähemmän tärkeitä tehtäviä, kun sille on aikaa, koska muuten iteratiivisuudesta ei ole hyötyä.

Kokonaisuuksiin keskittyminen onkin varsin toimiva vaihtoehto, kun halutaan saada datamallit rakennettua toimiviksi. Projekti sai selvästi paremman suunnan, kun jokainen kehitystiimin jäsen sai keskittyä jonkin tietyn rakenteen valmistamiseen, jotka liittyivät lopulta toisiinsa. Koska alkuperäinen toteutus on nippu kompleksisia toiminnallisuuksia nidottuna yhteen saman järjestelmän alle, niin kaikkia haluttuja ominaisuuksia, joita kehitettiin projektin alussa ei saatu viimeisteltyä pienien riippuvuussuhteiden takia. Kun yhteistyötä ja skaalaa tarkennettiin, niin selvästi saatiin ulos useampia toiminnallisuuksia, joita kehitetyt tietomallit tukivat.

6 TULOKSET JA YHTEENVETO

Tämän diplomityön tuloksena voidaan pitää kirjallisuuden pohjalta tehdyn selvityksen ja asiakasprojektista saadun kokemuksen vertailua. Työssä haluttiin vastata yleisemmän tason kysymyksiin, kuten milloin mikropalveluarkkitehtuuri on parempi vaihtoehto monoliittisen arkkitehtuurin sijasta ja millaisia vaihtoehtoja on mallintaa dataa, sekä mitä tiedonmallintamiseen kuuluu. Lisäksi työssä haluttiin peilata kokemuksia uudelleenkehitysprojektista, kuten miten mikropalvelukehityksen prosesseja kannattaa suunnitella kirjallisuuden pohjalta ja kuinka suunnitellut prosessit toteutuvat käytännössä. Lopuksi pyrimme luomaan toimintamalleja, joita voi käyttää jatkossa mikropalvelujärjestelmien tietomallin kehityksessä, jotta tulevaisuudessa kehitys voi olla johdonmukaisempaa. Lisäksi muita vähemmän konkreettisia tavoitteita työn osalta oli toimia osittaisena dokumentaationa ja läpileikkauksena mikropalvelukehityksen projektiin, antaa suuntaa miksi valita mikropalvelukehitys ja kasvattaa tekijän tietoutta, sekä osaamista mikropalveluista.

Tehdyn tutkimuksen perusteella kokemuksen ja kirjallisuuden yhteys on selvästi näkyvillä, kun tarkastellaan tutkimuskysymysten TK1, TK2 ja TK3 vastauksia. Johtopäätöksenä voimme siis todeta, että työssä käytetyt lähteet ja niiden tulokset vaikuttavat olevan hyvin verrattavissa oikeisiin ohjelmistokehitysprojekteihin, huomioiden, että jokainen ohjelmistokehitysprojekti eroaa toisistaan taustojen puolesta eli tuskin on olemassa täysin identtistä projektia, johon kaikki toimintatavat toimivat suoraan kuten muissa projekteissa.

TK1 vastasimme kysymykseen listalla kriteereistä, joiden täyttyessä mikropalveluarkkitehtuuri on parempi vaihtoehto kuin monoliittinen arkkitehtuuri. Esimerkkiprojekti täyttää kyseiset kriteerit ja hylkii monoliittista järjestelmää puoltavia kriteerejä, joten mikropalveluarkkitehtuuri on ollut niiden perusteella hyvä valinta. Kuten kaikilla malleilla, myös mikropalveluarkkitehtuurilla on omat kompastuskivensä, joten kaikkia ongelmia se ei poista, mutta tulevaisuudessa siitä voi olla hyötyä järjestelmän elinkaaren ylläpitämisessä.

TK2 tuotimme prosessikaavion, jossa yhdistelimme määritys-, kehitys- ja toimitusvaiheen prosessit yhden kaavion alle kirjallisuuden ja lähteiden pohjalta. Tavoitteena luoda paras mahdollinen lähestymistapa kehitystä varten, joka on johdonmukainen ja helposti ymmärrettävä. Ehkä suurimpana huomiona TK2 vastauksessa nostimme määrittelyn ja

muutoksiin reagoimisen haasteista. Kuten muissakin kirjallisuuslähteissä ja esimerkkiprojektissa muuttuviin tarpeisiin reagointi aiheuttaa yleensä liiallista työtaakkaa, joko prosessien tai yhteisten sääntöjen puuttuessa. Prosessien olemassaoloon kannattaa kiinnittää huomiota ja sidosryhmien kouluttaminen asian yhteydessä saattaa tuottaa huomattavia vaikutuksia ymmärtää ja ratkaista kasaantuvia muutostehtäviä.

TK3 varten loimme toimintamalleja, joiden pohjalta tulevaisuudessa on helpompaa asennoitua mitä vaaditaan tietomallien rakentamiseksi tai tarkentamiseksi. Periaatteessa TK2 vastaus auttaa myös TK3 kysymykseen vastaamisessa, koska keskittyminen tiettyyn rakenteeseen hyvien prosessien avulla tuottaa yleensä tulosta, sekä muutokset huomioon ottamalla prosesseissa päästään tarkentamaan muuttuvia vaatimuksia ja sitä kautta tietomallia. Tiivistettynä helpottaakseen tietomallien rakentamista kannattaa ajatella tietorakennetta käyttävän ohjelmiston luontia MVP-muodossa (pienin julkaisukelpoinen tuote eli engl. Minimum Viable Product). Tämä tarkoittaa sitä, että rakennetaan ohjelmiston tarvitsemat ydinasiat ja tällä tavalla saadaan myös tietomallin ydin selville, jotta saadaan tarvittava ominaisuudet kuntoon. Tällä tavalla saadaan lopulta luotua palaset, joista mikropalvelu lopulta koostuu.

Yllä olevan pohdinnan perusteella voimme sanoa, että työ täyttää sille asetetut tavoitteet, sekä sen tekeminen on vahvistanut työn tekijän osaamista mikropalveluista ja yleisesti informaatioteknologian arkkitehtuureihin liittyvien aiheiden käsittelyä. Työtä tehdessä haasteita työn kirjoittamisessa aiheutti terminologian käyttö ja vaihteleva käsitys siitä, millainen toteutus mikropalvelumalli oikeasti on eri lähteiden perusteella. Aiheesta on kuitenkin runsaasti tietoa tarjolla, sekä erilaiset yritykset ovat avanneet ainakin suppeasti mikropalveluarkkitehtuuriaan ja syitä siihen vaihtamiseen. Työn osalta aiheita voi käsitellä tulevaisuudessakin tutkivassa kirjallisuudessa. Esimerkiksi tässä työssä tarkastelun kohteena oli dataa vain yhdestä mikropalvelukehitysprojektista, mutta suuremman otannan kautta tuloksia voidaan mielestämme tarkentaa, sekä mikropalveluista aiheena saa monia muita kysymyksiä aikaan. Tulevaisuudessa tutkimusta voi jatkaa esimerkiksi seuraavista aiheista, kuten kuinka paljon esimerkiksi kyseinen esimerkkinä käytetty projekti tarjoaa elinkaaren osalta parannuksia kehitystä varten, millaisia taloudellisia vaikutuksia mikropalveluarkkitehtuurilla on sitä käyttävän yrityksen toimintaan ja nouseeko tulevaisuudessa muita arkkitehtuurimalleja haastamaan mikropalvelumallin asemaa.

Lähteet

Acosta J; Gajda K. Päiväämätön. “Test-driven development”. Viitattu 29.12.2023. https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/

Agrawal V. 2023. “Dimensional Data Modelling: 6 Critical Aspects”. Viitattu 26.8.2023 https://hevodata.com/learn/dimensional-data-modelling/

Allen, S., Terry, E. 2005. “Beginning Relational Data Modeling: A practical, step-by-step guide to data modeling for all IT professionals”. 2nd ed. Apress Berkeley, CA. https://doi.org/10.1007/978-1-4302-0015-4

Amazon: Gong, C; Giuli, R. 2021. “Choose the right storage tiers for your needs in Amazon OpenSearch Service”. Viitattu 26.8.2023. https://aws.amazon.com/blogs/big-data/choose-the-right-storage-tier-for-your-needs-in-amazon-opensearch-service/

Amit. 2018. “All You Need to Know About UML Diagrams: Types and 5+ Examples”. Tallyfy. Viitattu 6.10.2023 https://tallyfy.com/uml-diagram/

Awati R. Päiväämätön. TechTarget. “monolithic architecture”. Viitattu 6.10.2023 https://medium.com/swlh/scaling-monolithic-applications-3c69193f942a

M. Bano, S. Imtiaz, N. Ikram, M. Niazi and M. Usman, "Causes of requirement change - A systematic literature review," 16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012), Ciudad Real, 2012, pp. 22-31, doi: 10.1049/ic.2012.0003. http://doi.org/10.1049/ic.2012.0003

Beck, K; Grenning, J & CO. 2001 “Manifesto for Agile Software Development”. Viitattu 13.12.2023. https://agilemanifesto.org/

Brink, H; Settlemire M.W. 2016. “Project Scaling Methodology-Effective Use of Project Complexity Attributes for Determination of Project Size & Scale”. Viitattu 6.12.2023. https://www.projectmanagement.com/articles/319592/Project-Scaling-Methodology-Effective-Use-of-Project-Complexity-Attributes-for-Determination-of-Project-Size---Scale

Bugaev, L. 2023. ”Why do microservices need an API gateway?”. Tyk. Viitattu 9.3.2024. https://tyk.io/blog/microservices-api-gateway/

DesignStudio. 2023. “Importance of UI / UX Design in Today’s Digital World”. Viitattu 29.12.2023. https://www.designstudiouiux.com/blog/importance-of-ui-ux-design-in-todays-digital-world/

Brian Dobing and Jeffrey Parsons. 2006. How UML is used. Commun. ACM 49, 5 (May 2006), 109–113. https://doi.org/10.1145/1125944.1125949

Heeager, Lise Tordrup and Nielsen, Peter Axel, "Agile Software Development and its Compatibility with a Document-Driven Approach? A Case Study" (2009). ACIS 2009 Proceedings. 84. https://aisel.aisnet.org/acis2009/84/

Gillis, A. Päiväämätön. TechTarget. “Refactoring”. Viitattu 26.8.2023. https://www.techtarget.com/searchapparchitecture/definition/refactoring

GitLab. Päiväämätön. “What is DevOps?”. Viitattu 20.1.2024. https://about.gitlab.com/topics/devops/

Google Cloud. Päiväämätön. “Refactoring a monolith into microservices”. Viitattu 20.12.2023. https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths

Hall T. Päiväämätön. Atlassian. “What is a DevOps engineer?”. Viitattu 20.1.2024. https://www.atlassian.com/devops/what-is-devops/devops-engineer

Hillpot J. 2023. Dreamfactory. “7 Key benefits of Microservices”. https://blog.dreamfactory.com/7-key-benefits-of-microservices/

Hillpot J. 2023. Dreamfactory. “4 Microservices Examples: Amazon, Netflix, Uber, and Etsy”. Viitattu 8.3.2024. https://blog.dreamfactory.com/microservices-examples/

M. Hilbrich and F. Lehmann, "Discussing Microservices: Definitions, Pitfalls, and their Relations," 2022 IEEE International Conference on Services Computing (SCC), Barcelona, Spain, 2022, pp. 39-44, doi: 10.1109/SCC55611.2022.00019. http://doi.org/10.1109/SCC55611.2022.00019

Kalman, R; Wallin, J. 2022. “Software Visualization – Challenge, Accepted”. https://engineering.atspotify.com/2022/07/software-visualization-challenge-accepted/

Kalske, M., Mäkitalo, N., Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture. In: Garrigós, I., Wimmer, M. (eds) Current Trends in Web Engineering. ICWE 2017. Lecture Notes in Computer Science(), vol 10544. Springer, Cham. https://doi.org/10.1007/978-3-319-74433-9_3

Kirvan P. Päiväämätön. TechTarget. “Loose coupling” https://www.techtarget.com/searchnetworking/definition/loose-coupling

Kong. Päiväämätön. “Types of APIs and Use Cases”. https://konghq.com/learning-center/api-management/different-api-types-and-use-cases

Lekman L.; Eskelinen A.; Heiramo P.; ym. 2013. “The Scrum Guide™ Scrumin —

määritelmä ja pelisäännöt”. Käännös alkuperäisteoksesta Schwaber & Sutherland: The Scrum Guide. Viitattu 2.2.2024. https://scrumguides.org/docs/scrumguide/v1/Scrum-Guide-FI.pdf

Lenin, S. 2023. ”When do you need an API gateway”. Software AG. Viitattu 9.3.2024. https://blog.softwareag.com/when-do-you-need-an-api-gateway/

Lennes, M. 2004. “Mikä on skripti”. Viitattu 26.8.2023. https://lennes.github.io/praat-opas/node43.html

Lohani R. 12.3.2018. “Roles and Responsibilities in a microsevices world”. Viitattu 25.1.2024. https://rlohani.medium.com/roles-and-responsibilities-in-a-microsevices-world-bda7e43bccee

Luqi, L. Zhang, V. Berzins and Y. Qiao, "Documentation driven development for complex real-time systems," in IEEE Transactions on Software Engineering, vol. 30, no. 12, pp. 936-952, Dec. 2004, doi: 10.1109/TSE.2004.100. http://doi.org/10.1109/TSE.2004.100

Medhat N. 2001. “Scaling Monolithic Applications”. Medium. Viitattu 6.10.2023. https://medium.com/swlh/scaling-monolithic-applications-3c69193f942a

Microsoft; Github: jamesmontemagno; erjain; et. al. 2022. Microsoft Learn. “Communication in a microservice architecture”. Viitattu 9.9.2023 https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

Microsoft; Github: normesta; fhryo-msft; et. al. 2023. Microsoft Learn. “Access tiers for blob data”. Viitattu 26.8.2023 https://learn.microsoft.com/en-us/azure/storage/blobs/access-tiers-overview

Microsoft. Päiväämätön. “Microservice architechture style”. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Microsoft Dynamics. Päiväämätön. “Mikä on ERP?”. Viitattu 10.2.2024. https://dynamics.microsoft.com/fi-fi/erp/what-is-erp/

Nadareishvili I.; Mitra R.; McLarty M; Admundsen M. 2016. “Microservice Architecture: Aligning Principles, Practices, And Culture”. Viitattu 24.1.2024. broadcom.com/doc/microservice-architecture-aligning-principles-practices-and-culture

Oracle. Päiväämätön. “What is a Relational Database (RDBMS)?”. https://www.oracle.com/in/database/what-is-a-relational-database/

Oxford Learner's Dictionaries. 2023. "Process". Viitattu 6.12.2023. https://www.oxfordlearnersdictionaries.com/definition/english/process1_1

Prasdika. 28.4.2023. Medium: "The Benefits and Challenges of Test Driven Development". Viitattu 29.12.2023. https://medium.com/@bintangseptiandaru/the-benefits-and-challenges-of-test-driven-development-e3d29a73bc91

Redhat. 8.1.2019. "Who is a DevOps engineer?". Viitattu 20.1.2024. https://www.redhat.com/en/topics/devops/devops-engineer

Redhat. 12.12.2023. "What is CI/CD?". Viitattu 20.1.2024. https://www.redhat.com/en/topics/devops/what-is-ci-cd

Richardson C. 2024. "What are microservices". Viitattu 24.1.2024. https://microservices.io/index.html

Rotem-Gal-Oz, A. 2006. "Fallacies of Distributed Computing Explained". Viitattu 9.9.2023. https://www.se.rit.edu/~se442/doc/fallacies.pdf

SAP. Päiväämätön. "What is data modeling". Viitattu 26.8.2023 https://www.sap.com/products/technology-platform/datasphere/what-is-data-modeling.html

Satklichov, A. 2008. "What is Modeling and Modeling languages" Viitattu 6.10.2023. http://sahet.net/htm/swdev9.html

D. Shadija, M. Rezai and R. Hill, "Towards an understanding of microservices," 2017 23rd International Conference on Automation and Computing (ICAC), Huddersfield, UK, 2017, pp. 1-6. http://doi.org/10.1109/10.23919/IConAC.2017.8082018

Shukla, N. 2023. "Monolithic Architecture". Geeks For Geeks. Viitattu 6.10.2023 https://www.geeksforgeeks.org/monolithic-architecture/

Schwaber K. 2004. "Agile project management with Scrum". 1st ed. Microsoft Press. ISBN: 9780735619937

Schwartz, S. 2018. "How Coca-Cola migrated from a single data warehouse to global application deployment". CIODIVE. Viitattu 6.10.2023 https://www.ciodive.com/news/how-coca-cola-migrated-from-a-single-data-warehouse-to-global-application-d/519070/

Talouselämä. 2018. "Apotti-hankkeen hinta noussut jo yli 800 miljoonaan euroon – summam on 40 prosenttia alkuperäisarviota suurempi". Viitattu 6.12.2023 https://www.talouselama.fi/uutiset/apotti-hankkeen-hinta-noussut-jo-yli-800-miljoonaan-euroon-summa-on-40-prosenttia-alkuperaisarviota-suurempi/6474e6c5-0626-48dd-8744-de583b65e47f

TestDriven Labs. 2023. “What is Test-Driven Development”. Viitattu 29.12.2023. https://testdriven.io/test-driven-development/

M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," 2015 10th Computing Colombian Conference (10CCC), Bogota, Colombia, 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476 http://doi.org/10.1109/ColumbianCC.2015.7333476