SOFTWARE AS A SERVICE - ARKKITEHTUURIN SUUNNITTELU, VALIDOINTI JA VAIKUTUKSET LIIKETOIMINTAMALLIIN

Jussi Haaja

Opinnäytetyö
Huhtikuu 2015
Tietojärjestelmäosaaminen, YAMK
Tampereen ammattikorkea-koulu

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojärjestelmäosaamisen koulutusohjelma, ylempi AMK

JUSSI HAAJA:
Software as a Service -arkkitehtuurin suunnittelu, validointi ja vaikutukset liiketoimintamalliin

Opinnäytetyö 69 sivua
Huhtikuu 2015

Tämän opinnäytetyön tavoitteena oli kehittää Ambientia Oy:n asiakkailleen tarjoamien sovellusten toimitusketjua tilauksesta toimitukseen läpimenoajaltaan mahdollisimman nopeaksi. Ambientia Oy on suomalainen, hieman yli 100 henkilöä työllistävä IT-alan palveluyritys, joka tuottaa pääasiassa verkkosovelluksia asiakkaidensa liiketoiminnan tueksi.

Opinnäytetyössä keskityttiin toimitusketjun optimointiin teknisin ratkaisuin automaatiota kasvattamalla. Työ on osa laajempaa kehittämiskokonaisuutta, jossa kehitetään koko infrastruktuurin automaatiota. Opinnäytetyössä aikaansaatu toteutus on tarkoitus tuotteistaa Software as a Service (SaaS) -palveluksi.

Raportissa esitellään korkeamman automaatioasteen saavuttamiseksi suunniteltu järjestelmäarkkitehtuuri, siinä käytetyt komponentit sekä arkkitehtuurin suunnitteluprosessi. Toteutetun arkkitehtuurin soveltuvuus käyttötarkoitukseensa arvioitiin vertailemalla sitä Software as a Service -kypsyystasomallia vasten ja vertaisarviointina käyttämällä Decision Centric Architecture Review -menetelmää.

Toteutettua järjestelmäarkkitehtuuria on hyödynnetty Ambientia Cloud -nimellä tarjottavissa Software as a Service -palveluissa. Työssä arvioitiin käytännön kokemusten ja kirjatiedon perusteella automaatioasteen kasvatuksen vaikutusta liiketoimintamalliin. Automaatioasteen kasvattaminen voi mahdollistaa uudenlaisia palvelumalleja, joilla voidaan taas tavoittaa uusia asiakaskuntia. Toisaalta automaatioasteen kasvattamisella ei tarvitse olla ulospäin näkyvää vaikutusta, vaan siitä saatavat hyödyt ja säästöt voidaan hyödyntää sisäisesti.

Työssä käsitellään järjestelmäarkkitehtuuria ja sen automaatiota erityisesti palveluntarjoajan näkökulmasta, mutta toteutettua arkkitehtuuria voidaan soveltaa myös muissa ympäristöissä, kuten suurten organisaatioiden sisäisessä palvelutuotannossa.

Asiasanat: järjestelmäarkkitehtuuri, systeemityö, pilvipalvelut, software as a service

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Master's Degree Programme in Information System Competence

JUSSI HAAJA:
Software as a Service Architecture Design, Validation And Impact On The Business Model of an ICT Company

Master's thesis 69 pages
April 2015

The purpose of this thesis is to develop technical solutions to enhance lead time from order to delivery of web applications provided by Ambientia Oy. Ambientia Oy is a Finnish IT services company employing a little over 100 employees. Ambientia mainly develops web based applications to support its customers' business processes. The technical solutions developed in this thesis are intended to form the base for Software as a Service type of services.

The thesis presents a system architecture and its core components, designed to provide a high level of automation and rapid application delivery. The process of system architecture design is also covered in detail. The system architecture is subsequentially evaluated using an architecture evaluation method called Decision Centric Architecture Review.

The system architecture developed here has been utilized in services marketed under the name Ambientia Cloud. The thesis evaluates the potential impact of the system architecture to the business model of the company. Evaluation is based on research articles and practical experience on providing a SaaS-service. It was found that implementing highly automated application delivery mechanisms makes it possible to provide new kinds of services for the business. Still, it is not necessary to alter the business model when automating the infrastructure behind the service that is provided. The benefits and cost savings of higher level automation can be utilized internally.

This thesis deals with system architecture automation mainly from a service provider's point of view. It must be noted that the presented architecture can also be applied to other types of environments, for example large organisations' internal IT service delivery.

Key words: system architecture, systems work, cloud services, software as a service

SISÄLLYS

LYHENTEET JA TERMIT

ASP	Application Service Provider. Sovelluspalveluntarjoaja, Software as a Service -mallin esiaste.
ATAM	Architecture Tradeoff Analysis Method, Carnegie Mellon -yliopiston Software Engineering Institutessa kehitetty arkkitehtuurin arviointimenetelmä.
DCAR	Decision Centric Architecture Review, Tampereen teknillisen yliopiston ja Groningenin yliopiston kehittämä vaihtoehtoinen arkkitehtuurin arviointimenetelmä.
Monitenanttisuus	Ohjelmistoarkkitehtuuri, jossa useat loppukäyttäjäorganisaatiot toimivat saman sovellusympäristön sisällä toisistaan eriytettynä.
IaaS	Infrastructure as a Service, pilvipalveluiden palvelumalli, jossa asiakas hankkii palveluntarjoajalta IT-infrastruktuuria, kuten laskentakapasiteettia ja tallennustilaa.
Käyttöympäristö	Käytetyn laitteiston, käyttöjärjestelmän ja ohjelmistojen muodostama kokonaisuus, jossa muita ohjelmistoja käytetään.
PaaS	Platform as a Service, pilvipalveluiden palvelumalli, jossa asiakas hankkii palveluntarjoajalta IT-infrastruktuurin ohella käyttövalmiin sovellusalustan (platform), johon asiakas voi asentaa hankkimansa tai itse toteuttamansa sovelluksen.
RPM	Red Hatin kehittämä sovellusten paketointiformaatti Linux-käyttöjärjestelmille. Alun perin lyhenne sanoista Red Hat Package Manager, nykyisin rekursiivinen lyhenne sanoille RPM Package Manager.

SaaS	Software as a Service, pilvipalveluiden palvelumalli, jossa asiakas hankkii palveluntarjoajalta käyttövalmiin palvelun tai sovelluksen käytettäväksi verkon yli, tyypillisesti web-selaimen tai vastaavan kevyen asiakasohjelman välityksellä.

1 JOHDANTO

1.1 Taustat

Tämän opinnäytetyön aiheena on kehittää Ambientia Oy:n asiakkailleen tarjoamien sovellusten toimitusketjua tilauksesta toimitukseen läpimenoajaltaan mahdollisimman nopeaksi. Opinnäytetyössä keskitytään toimitusketjun optimointiin teknisin ratkaisuin automaatiota kasvattamalla.

Opinnäytetyö on osa laajempaa kehittämiskokonaisuutta, jossa kehitetään koko infrastruktuurin automaatiota. Opinnäytetyö toimii tämän kokonaisuuden sisällä sovellusten käyttöönottoautomaation ensimmäisenä vaiheena. Ensimmäisessä vaiheessa keskitytään Atlassian Confluence ja Atlassian JIRA -sovellusten käyttöönoton automaatioon siten, että luotuja ratkaisuja pystytään myöhemmin hyödyntämään muussakin Ambientian palvelutarjonnassa. Opinnäytetyössä aikaansaatu toteutus on tarkoitus tuotteistaa Software as a Service (SaaS) -palveluksi.

1.2 Työn toimeksiantaja

Ambientia Oy on suomalainen, hieman yli 100 henkilöä työllistävä IT-alan palveluyritys, joka tuottaa pääasiassa verkkosovelluksia asiakkaidensa liiketoiminnan tueksi (Ambientia Oy). Osana palvelutarjontaansa Ambientia tarjoaa asiakkailleen myös hosting-ratkaisuja, joissa asiakkaalle toteutettava verkkosovellus voidaan sijoittaa Ambientian omaan palvelininfrastruktuuriin. Ambientia on edellä mainittujen JIRA ja Confluence-sovellusten valmistajan, australialaislähtöisen Atlassianin, korkeimman Platinum-tason kumppani. Confluence on erityisesti yrityskäyttöön suunnattu, wiki-tyylinen yhteistyöalusta ja JIRA hyvin monipuolinen tehtävän- ja projektinhallintajärjestelmä. Työn tekijä toimii Ambientialla vanhempana järjestelmäasiantuntijana sekä Ambientian pilvipalveluiden nk. konseptiomistajana, joka vastaa kyseisen liiketoiminnon käytännön kehittämistyöstä. Ambientia lanseerasi omat, tähän työhön vahvasti liittyvät, pilvipalvelunsa vuoden 2013 lopussa.

1.3 Työn tarkoitus ja tavoitteet

Tämän työn tavoitteena on suunnitella sellainen järjestelmäarkkitehtuuri, jonka avulla nykyisin pääsääntöisesti käsityönä tehtävät sovellusasennukset saadaan automatisoitua mahdollisimman pitkälle. Tämä arkkitehtuurin kehittämistyö on se olennainen mekanismi, jolla toimitusketjun läpimenoaikaa pyritään lyhentämään.

Arkkitehtuurin suunnittelun lisäksi arkkitehtuuri validoidaan tarkoitukseen sopivaksi. Arkkitehtuurin validointiin käytetään Tampereen teknillisen yliopiston yhdessä Groningenin yliopiston kanssa kehittämää DCAR-menetelmää (Decision Centric Architecture Review).

Suunniteltu toteutus tuotteistetaan Software as a Service -mallin mukaiseksi ratkaisuksi. Opinnäytetyön tarkoituksena on siis teknisten edellytysten luominen tuotteen mahdollistamiseksi, sekä tuotteistamisen suunnittelu erityisesti teknisestä näkökulmasta. Tuotteistusta tarkastellaan työssä lähinnä liiketoimintamallin muuttumisen näkökulmasta, eli miten Software as a Service -arkkitehtuuri voi muuttaa liiketoimintaa. Työssä kuvataan myös millaisia vaatimuksia järjestelmäarkkitehtuurille Software as a Service -toimintamalli asettaa ja miten niihin voidaan vastata.

Luodun SaaS-ratkaisun tarkoituksena on luoda lisää liiketoimintaa, helpottaa resurssointiongelmia käyttöönottoprojekteissa ja yhdenmukaistaa ympäristöjen arkkitehtuuria. Opinnäytetyön ulkopuolelle rajataan toimitusketjun optimointiin sekä tuotteistukseen liittyvät prosessiasiat.

1.4 Työssä käytetyt menetelmät

Työssä tehtävä tutkimus on konstruktiotutkimusta, jossa toteutetaan Software as a Service -palveluiden tuottamiseen sopivan palvelualustan arkkitehtuuri ja suunnitellaan sen tuotteistusta. Ennen kuin tällainen alusta voidaan suunnitella, täytyy tunnistaa alustaan liittyvät arkkitehtuuriset vaatimukset. Työn toteutus lähtee siis näiden vaatimusten tunnistamisesta ja niihin vastaamisesta organisaation teknologia- ja järjestelmäarkkitehtuuriin sopivalla tavalla.

Suunnitellun arkkitehtuurin pääasiallisena analysointimenetelmänä tullaan käyttämään Decision Centric Architecture Review -menetelmää, jossa arkkitehtuurisista ratkaisuista kiinnostuneet tahot arvioivat tehdyt arkkitehtuuriratkaisut tähän varatussa tilaisuudessa. Arkkitehtuuria verrataan myös soveltuvia arkkitehtuurikypsyysmalleja vasten.

Konstruktiivinen tutkimus on tapaustutkimusta, jossa pyritään ratkaisemaan reaalimaailman ongelma tuottamalla jokin konkreettinen toteutus vastauksena ongelmaan (Lukka 2001). Konkreettisen tuotoksen avulla testataan sen sopivuutta ongelman ratkaisuun käytännössä.

Tässä työssä toteutettava konstruktio on siis järjestelmäarkkitehtuuri, joka mahdollistaa Ambientia Oy:n asiakkailleen tarjoamien sovellusten toimittamisen mahdollisimman nopeasti ja pienellä manuaalisella työllä. Toteutettu järjestelmäarkkitehtuuri on suunniteltu yrityksen toimivan johdon antamista lähtökohdista, testattu käytännössä toimivaksi ja validoitu tarkoituksenmukaiseksi vertaisarvioinnilla sekä soveltuviin kypsyysmalleihin vertaamalla.

2 JÄRJESTELMÄARKKITEHTUURIN MÄÄRITELMIÄ

2.1 Järjestelmä

Jotta järjestelmäarkkitehtuurin käsitteen käsittely on mielekästä, on ensin ymmärettävä arkkitehtuurin kohdetta, järjestelmää. Blanchard ja Fabrycky (2006, 3) määrittelevät järjestelmän yhdistelmäksi osia, jotka yhdessä muodostavat monimutkaisen, yhtenäisen kokonaisuuden. Järjestelmä koostuu komponenteista, ominaisuuksista ja yhteyksistä. Komponentit ovat järjestelmän toimivia osia, ja ominaisuudet näiden komponenttien erityispiirteitä.

Mikä tahansa satunnainen kokoelma komponentteja ei automaattisesti muodosta järjestelmää, sillä komponenttien väliltä puuttuvat järjestelmän muodostavat yhteydet. Komponentit voivat yhdessä muodostaa järjestelmän vain jos niiden välillä on yhteyksiä, joiden kautta komponentit vuorovaikuttavat toisiinsa. Täten vain yhdessä toimivat komponentit muodostavat järjestelmän, jolla on mielekäs tarkoitus. (Blanchard & Fabrycky 2006, 3).

2.2 Järjestelmäarkkitehtuuri

Järjestelmäarkkitehtuurille ei ole mitään yksittäistä, yleisesti hyväksyttyä määritelmää, vaan se on usein määritelty kontekstiriippuvaisesti. Yhteistä määritelmille on kuitenkin se, että järjestelmäarkkitehtuuri kuvaa korkealla tasolla järjestelmän rakenteet, komponenttien väliset yhteydet sekä tiedonvälitysperiaatteet ja ylipäätään sen, miten järjestelmä saavuttaa sille asetetun toiminnallisen tehtävän (Bass, Clements & Kazman 2013, 7; Jaakkola & Thalheim 2011, 97; Blanchard & Fabrycky 2006, 85).

Samoista komponenteista voi usein muodostaa eri järjestelmiä erilaisilla arkkitehtuureilla, mutta yksittäisellä järjestelmällä on aina vain yksi arkkitehtuuri. Täten keskeisimmät osat yksittäistä arkkitehtuuria ovatkin päätökset siitä, miten järjestelmän komponentit liitetään toisiinsa ja mikä rooli milläkin komponentilla järjestelmässä on. Arkkitehtuurissa tehtyjen päätösten tarkastelu onkin tärkeässä osassa arkkitehtuuria validoitaessa. Arkkitehtuurin validointiin palataan tarkemmin luvussa 7.

2.3 Toiminalliset ja ei-toiminnalliset vaatimukset sekä arkkitehtuurin laatuominaisuudet

Toiminnalliset vaatimukset muodostavat arkkitehtuurin lähtökohdat. Ne kuvaavat sen, mitä järjestelmän on tehtävä, eli mikä on järjestelmän tarkoitus. Toiminnalliset vaatimukset eivät kuitenkaan ota kantaa siihen, *miten* järjestelmä suorittaa sille suunnitellun tehtävän. Sama toiminnallisuus voidaan saavuttaa lukuisilla erilaisilla arkkitehtuureilla. (Bass ym. 2013, 64-65).

Toiminnallisten vaatimusten ohella järjestelmään kohdistuu lisäksi ei-toiminnallisia vaatimuksia sekä rajoitteita (Bass ym. 2013, 64). Nämä seikat ovat keskeisiä tekijöitä arkkitehtuuria suunniteltaessa. Järjestelmiä luodaan pääsääntöisesti organisaatioiden toimesta joidenkin päämäärien saavuttamiseksi. Tällöin organisaation tavoitteet järjestelmän luomisessa heijastuvat ei-toiminnallisiksi vaatimuksiksi ja rajoitteiksi. (Bass ym. 2013, 49-50).

Tyypillisiä tietojärjestelmien ei-toiminnallisia vaatimuksia ovat esimerkiksi tietty suorituskykytaso tai korkea tietoturvallisuuden taso. Tyypillisiä rajoitteita ovat esimerkiksi vaatimus käyttää arkkitehtuurissa tiettyä teknologiaa tai toimintamallia, kuten palvelukeskeisyyttä. (Bass ym. 2013, 64). Ei-toiminnallisten vaatimusten ja rajoitteiden erona on, että rajoitteet ovat täysin ehdottomia ja lukitsevat osia arkkitehtuurista. Ei-toiminnalliset vaatimukset voidaan taas täyttää useilla eri tavoilla. Ei-toiminnalliset vaatimukset ohjaavat arkkitehtuurisia päätöksiä tiettyyn suuntaan, mutta rajoitteet asettavat arkkitehtuurisia päätöksiä valmiiksi. Tällöin muu arkkitehtuuri on vain mukautettava rajoitteiden kanssa yhteen.

Tarkasteltaessa arkkitehtuuria ei-toiminnallisia vaatimuksia vasten, tarkastellaan arkkitehtuuria tyypillisesti sen ilmentämien laatuominaisuuksien (engl. quality attributes) kautta. Arkkitehtuurin laatuominaisuudet ovat sen mitattavia ominaisuuksia jotka ilmentävät arkkitehtuurin laatua jossakin ulottuvuudessa. (Bass ym. 2013, 63).

Blanchard ja Fabrycky (2006, 37, 75-77) käyttävät laatuominaisuuksista käsitettä tekninen suorituskykymitta (engl. technical performance measure, TPM) ja ei-toiminnallisista vaatimuksista käsitettä suunnitteluriippuvainen parametri, design-

dependent parameter (DDP). Tässä työssä on päätetty käyttää pääosin Bass ym. (2013, 64) määrittelemiä termejä ei-toiminnallinen vaatimus ja laatuominaisuus, sillä ne ovat paremmin ymmärrettävissä järjestelmäarkkitehtuurisessa kontekstissa kuin Blanchardin ja Fabryckyn yleiseen systeemiteoriaan juurretut käsitteet.

Kansainvälinen standardoimisjärjestö ISO on määrittänyt listan kahdeksasta standardista laatuominaisuuksista ISO-standardissa 25010:2011. Nämä standardin mukaiset (ISO 2011) laatuominaisuudet ovat :

- Toiminnallinen soveltuvuus, järjestelmän kyky täyttää sille asetetut toiminnalliset vaatimukset
- Käyttötehokkuus, järjestelmän kyky suoriutua tehtävistään käytettävissä olevien resurssien puitteissadr
- Yhteensopivuus, järjestelmän kyky toimia yhdessä muiden järjestelmien kanssa ja niitä häiritsemättä
- Käytettävyys, järjestelmän käyttäjää koskevat piirteet, kuten opittavuus, käyttäjävirheiltä suojaaminen ja käyttöliittymän esteettiset ominaisuudet
- Luotettavuus, järjestelmän kyky säilyttää toimintakyky ja saavutettavuus
- Tietoturvallisuus, järjestelmän kyky käsitellä siinä säilytettyä tietoa siten, ettei se altistu oikeudettomalle käytölle tai muutoksille
- Ylläpidettävyys, järjestelmän mahdollistama muunneltavuus ja muuntamisen helppous sen tarkoitettujen ylläpitäjien näkökulmasta
- Siirrettävyys, järjestelmän yhteensopivuus erilaisten käyttöympäristöjen, kuten käyttöjärjestelmien ja prosessoriarkkitehtuurien kanssa, sekä siirtämisen helppous alustalta toiselle

ISO 25010 -standardin mukaiset laatuominaisuudet ovat kuitenkin vain yksi tapa jäsentää ohjelmiston tai järjestelmän laadullisia ominaisuuksia. Bass ym. (2013, 61) kuvaavat seitsemän laatuominaisuutta:

- Saavutettavuus, järjestelmän kyky suorittaa sille annettu tehtävä tarvittaessa, eli olla saavutetavissa tehtävän suorittamiseen
- Yhteensopivuus, järjestelmän kyky vaihtaa tietoja toisten järjestelmien kanssa
- Muunneltavuus, järjestelmän mahdollistama muuntelu ja muuntelusta aiheutuvat riskit ja kustannukset

- Suorituskyky, järjestelmän kyky vastata ajoitukseen liittyviin vaatimuksiin, kuten vasteaika tai prosessointinopeus suoritteina aikayksikköä kohden
- Tietoturvallisuus, järjestelmän kyky suojata siinä olevaa informaatiota oikeudettomalta käytöstä samaan aikaan sallien oikeutettu käyttö
- Testattavuus, järjestelmän kyky havaita ja ilmaista siinä ilmeneviä vikoja
- Käytettävyys, järjestelmän kyky edesauttaa käyttäjän tahtomien toiminteiden tapahtumista

Blanchard ja Fabrycky (2011, 367) puolestaan listaavat kuusi järjestelmien laatuominaisuutta:

- Luotettavuus, järjestelmän kyky säilyttää toimintakyky suunnitellun käyttöajan
- Huollettavuus, järjestelmän huollettavuuden helppous
- Käytettävyys, järjestelmän kyky huomioida sen käyttäjän tarpeet ja soveltua käyttäjänsä käyttöön
- Tuettavuus, järjestelmää tukevan infrastruktuurin valmius tukea järjestelmää sen elinkaaren ajan
- Tuotettavuus ja käytöstä poistettavuus, vaadittu työ järjestelmän käyttöön ottamiseen ja elinkaaren päättyessä käytöstä poistamiseen
- Kohtuuhintaisuus, järjestelmästä aiheutuvat kokonaiskustannukset sen elinkaaren aikana.

Laatuominaisuudet eivät kuitenkaan ole toisistaan irrallisia kokonaisuuksia. Esimerkiksi järjestelmän suorituskyky vaikuttaa sen käytettävyyteen: jos käyttöliittymän vaste on useita sekunteja, se ei kovin hyvin edesauta käyttäjän haluamien toiminteiden aikaansaamista. Järjestelmän kyky kestää palvelunestohyökkäys on toisaalta järjestelmän tietoturvallisuusominaisuus, mutta toisaalta myös saavutettavuusominaisuus. Myös eri määritelmien laatuominaisuuksien keskinäisestä asemasta voidaan tehdä erilaisia tulkintoja. Onko esimerkiksi siirrettävyys vain yhdentyyppistä muunneltavuutta?

Laatuominaisuudet vaikuttavat myös toisiinsa siten, että pääsääntöisesti on mahdotonta suunnitella järjestelmää, jossa kaikki laatuominaisuudet olisivat kaikki yhtä korkealla tasolla. Esimerkiksi korkea tietoturvallisuuden taso voi edellyttää vahvaa salausta, jolla on taas heikentävä vaikutus suorituskykyyn vahvemman salausalgoritmin vaatiessa enemmän laskentatehoa. Laskentatehon kasvattaminen tasolle, jossa vahva salaus ei

haittaa järjestelmän suorituskykyä taas kasvattaa järjestelmän kustannuksia. Iso osa arkkitehtuurisista päätöksistä onkin kompromisseja eri laatuominaisuuksien korostamisen välillä. (Clements, Kazman & Klein 2000, 22-23)

Järjestelmäarkkitehtuuri on siis keskeisesti päätöksiä järjestelmän toteuttamistavasta ja ei-toiminnalliset vaatimukset sekä rajoitteet keskeisesti näihin päätöksiin vaikuttavat tekijät. Tällöin voidaan todeta järjestelmäarkkitehtuurin muotoutuvan pääasiallisesti ei-toiminnallisten vaatimusten ja arkkitehtuuriin kohdistuvien rajoitteiden mukaiseksi. Järjestelmäarkkitehtuuri pyrkii siis täyttämään ei-toiminnalliset vaatimukset arkkitehtuuriin kohdistuvien rajoitusten puitteissa.

Arkkitehtuurin laatuominaisuudet taas mahdollistavat tavan katselmoida miten arkkitehtuuri vastaa ei-toiminnallisiin vaatimuksiin. Koska ei-toiminnalliset vaatimukset eivät ole tyypillisesti yksiselitteisiä kyllä/ei-tyyppisiä -vaatimuksia, voidaan laatuominaisuuksiin sisäänrakennettujen mittausmenetelmien kautta tarkastella miten hyvin arkkitehtuuri pystyy ei-toiminnallisiin vaatimuksiin vastaamaan. Laatuominaisuudet täten kuvaavat arkkitehtuurin laatua mitattuna jotakin ulottuvuutta (suorituskyky, muunneltavuus, tietoturvallisuus) vasten.

2.4 Arkkitehtuurilliset taktiikat

Arkkitehtuurilliset taktiikat (engl. architectural tactics) ovat menetelmiä, joilla arkkitehtuuri suunnitellusti vastaa ei-toiminnallisiin vaatimuksiin (Bass ym. 2013, 70-72). Taktiikka keskeisesti tarkoittaa arkkitehtuurisen päätöksen toteuttavaa osaa, esimerkiksi ”korkean tietoturvallisuuden saavuttamiseksi liikenne salataan 256-bittisellä symmetrisellä salauksella”. Tässä siis taktiikkana on 256-bittisen symmetrisen salauksen käyttö, jolla vastataan tietoturvallisuuteen liittyvään ei-toiminnalliseen vaatimukseen. Esimerkkitapauksessa ei-toiminnallinen vaatimus voisi olla vaikkapa ”palvelimen ja asiakasohjelman välistä liikennettä ei saa olla mahdollista lukea kolmannen osapuolen toimesta”.

Taktiikat pohjaavat teoreettisiin malleihin tai asiantuntijatietoon. Saavutettavuustaktiikoissa voidaan hyödyntää todennäköisyysmatemaattisia malleja ja johtaa siitä taktiikoita. Kaikilla laatuominaisuuksilla ei valitettavasti ole yhtä vankka teoriapohja, vaan esimerkiksi käytettävyydessä nojataan asiantuntijatiedon pohjalta syntyneisiin parhaisiin käytäntöihin. (Bass ym. 2013, 199). Arkkitehtuurisen päätöksen ja arkkitehtuurillisen

taktiikan erona on niiden laajuus. Arkkitehtuurinen päätös koskee yksittäistä arkkitehtuuria, mutta arkkitehtuurillinen taktiikka on käytettävissä kaikissa arkkitehtuureissa.

3 SOFTWARE AS A SERVICE TOIMINTAMALLINA

3.1 Software as a Service -mallin määritelmiä

Yksi eniten viitattuja Software as a Service -mallin määritelmiä löytyy yhdysvaltalaisen NIST:n (National Institute of Standards and Technology) julkaisusta ”The NIST Definition of Cloud Computing”, jossa Software as a Service -malli määritellään suoraan käytettävissä oleviksi sovelluksiksi, joita palveluntarjoaja tarjoaa asiakkailleen hallinnoimastaan infrastruktuurista. Sovelluksia käytetään verkon yli tyypillisesti web-selaimella tai muulla vastaavalla asiakasohjelmalla. Asiakas ei pysty vaikuttamaan sovelluksen taustalla olevaan infrastruktuuriin, mutta pystyy mahdollisesti joltain osin konfiguroimaan käytettävää sovellusta haluamallaan tavalla. (Mell & Grance 2011, 2).

Suomalaisittain vastaavan, joskin merkittävästi suppeamman määritelmän SaaS-mallista on antanut Julkisen hallinnon tietohallinnon neuvottelukunta JUHTA JHS-suositusten yhteydessä ylläpidettävässä JHS-käsitteistössä. JHS-käsitteistö määrittelee Software as a Service -mallin seuraavasti:

”Malli, jossa asiakas käyttää palvelun toteuttavaa ohjelmistoa palveluntarjoajan palvelimelta siten, että asiakkaan omiin laitteisiin ei tarvita erikseen asennettua ohjelmistoa” (TEPA-termipankki).

3.2 Software as a Service -mallin lyhyt historia

Sinänsä ajatus sovellusten etäkäytöstä web-selaimen tai muun asiakasohjelman avulla palveluntarjoajan verkosta ei ole uusi tai mullistava. Jo internetin yleistymisen alkuaikoina, 1990-luvun puolivälissä syntyi lukuisia sovelluspalveluntarjoajia (application service provider, ASP), jotka tarjosivat ohjelmistoja käytettäväksi internetin välityksellä. ASP-toiminnan alkutaipaleella tyypillistä oli, että tarjottavia ohjelmistoja ei sinänsä oltu palveluntarjoajan toimesta mitenkään mukautettu käytettäväksi verkon yli tai useiden eri asiakkaiden toimesta samanaikaisesti. Tarjottavia ohjelmistoja ei oltu välttämättä myöskään toteutettu itse, vaan sovelluspalvelun tarjoajan liiketoiminta pohjautuikin usein muiden toimijoiden toteuttamien ohjelmistojen tarjoamiseen. Kyse olikin käytännössä enemmän sovellusten vuokraamisesta asiakkaille. Taustalla palveluntarjoaja hankki sovellusten tarjoamiseen tarvittavat lisenssit ym. sovellusten valmistajalta ja

sisällytti nämä kulut omaan myyntihintaansa. (Schroff 2010, 27). Tämä toimintamalli ei ollut erityisen kustannustehokas ja suurin osa ensimmäisen aallon sovelluspalveluntarjoajista lopettikin toimintansa ennen pitkää. Eräs Software as a Service -mallin uranuurtajista on Salesforce.com, joka älysi toteuttaa asiakkuudenhallintajärjestelmänsä siten, että se pystyi hyvin tehokkaasti monistamaan tarjomaansa palvelua useille asiakkaille (Schroff 2010, 28).

Kyvyttömyys hyödyntää skaalaetuja (engl. *economies of scale*) aiheutui monen varhaisen ASP-toimijan kohtaloksi. Infrastruktuuriautomaation puute rajoitti pienten toimijoiden kykyä palvella laajempaa asiakaskuntaa. Myös tarjottujen sovellusten arkkitehtuuri haittasi skaalautumista. Käytännössä ensimmäiset sovelluspalveluntarjoajat tarjosivat käytettäväksi samoja sovelluksia omasta verkostaan kuin yritykset olisivat asentaneet omiin konesaleihinsa, jolloin yksittäinen sovellusinstanssi pystyi usein mielekkäästi palvelemaan vain yksittäistä asiakasta. (Schroff, 2010, 28-29). Uusi asiakkuus tarkoitti uuden sovellusinstanssin käyttöönottoa, mikä taas tarkoitti käsintehtävää asennustyötä. Merkittävä käsityön (ja sen aiheuttamien kustannusten) määrä yhdistettynä sovellusten lisensointikustannuksiin vaikeuttivat sovelluspalveluntarjoajien mahdollisuuksia saada aikaan kannattavaa liiketoimintaa. Parhaiten selvisivät ne, jotka pystyivät tarjoamaan sovelluksia siten, ettei uusien asiakkaiden palvelun käyttöönotto aiheuttanut merkittäviä kustannuksia, eli ne palveluntarjoajat joiden palvelut olivat lähimpänä nykyisin ymmärrettyä Software as a Service -palvelumallia.

3.3 Software as a Service -malli verrattuna muihin pilvipalvelumalleihin

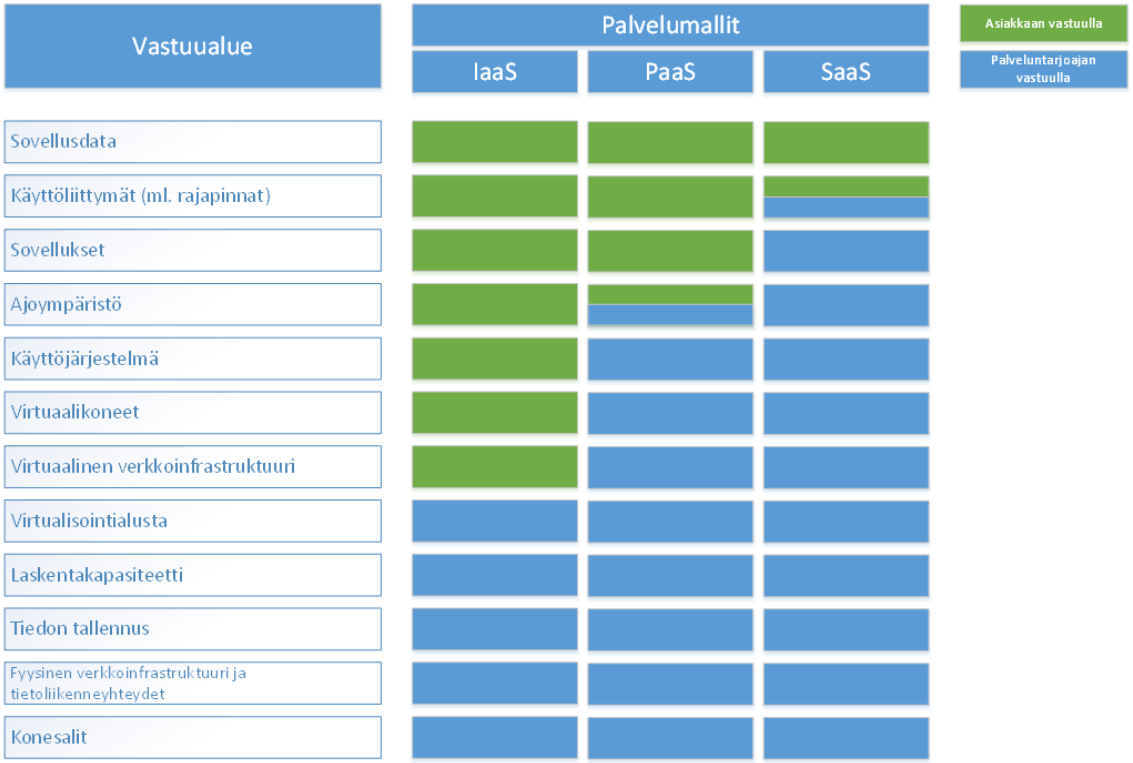
Verrattuna muihin yleisesti käytettyihin, NIST:n määrittelemiin palvelumalleihin (Infrastructure as a Service, *IaaS* ja Platform as a Service, *PaaS*) Software as a Service tarjoaa käyttäjälle käyttövalmiin sovelluksen. Hierarkisesti nämä kolme mallia voidaan ajatella päällekkäin siten, että IaaS-malli tarjoaa loppukäyttäjälle suurimman mahdollisen kontrollin sovelluksen toteuttamiseen, tuoden samalla käytännössä kaiken vastuun sovelluksen toteuttamisesta asiakkaalle. Palveluntarjoaja vastaa vain sovellusten tarjoamiseen tarvittavien perusasioiden, kuten laskentakapasiteetin, tallennustilan ja verkkoyhteyksien tuottamisesta asiakkaan käyttöön (Mell & Grance 2011, 3).

Platform as a Service -malli sijoittuu näiden kahden ääripään väliin. PaaS-mallissa palveluntarjoaja vastaa *joistakin* osista itse sovelluksen teknologiapinoa, tyypillisesti esi-

merkiksi käyttöjärjestelmästä ja sovelluksen ajoympäristöstä ja kirjastoista. PaaS-mallissakin kuitenkin itse sovelluksen käyttöönotto ja konfigurointi jäävät loppukäyttäjän vastuulle.

Software as a Service -malli tarjoaa siis loppukäyttäjän näkökulmasta kaikkein käyttövalmiimman ympäristön, mutta toisaalta myös kaikkein vähiten kontrollia ympäristön ominaisuuksiin. SaaS-mallissa palveluntarjoaja on tehnyt suurimman osan sovellusympäristöön ja sen arkkitehtuurin liittyvistä päätöksistä asiakkaan puolesta, eikä näihin seikkoihin pysty vaikuttamaan.

Kuva 1 havainnollistaa mallien suhdetta sen suhteen, mikä kuuluu asiakkaan ja mikä palveluntarjoajan vastuulle.



KUVA 1. Pilvipalvelumallien vastuurajat (PCI Security Standards Council 2013, muokattu)

3.4 Software as a Service -mallin edut loppukäyttäjäorganisaatiolle

Pilvipalveluiden etuina loppukäyttäjäorganisaatiolle nähdään perinteistä, itse tuotettua palvelujen tarjoamista joustavampi kustannusrakenne sekä tehokkaampi, lähes välitön palveluiden toimitus. Tämä mahdollistaa taas uudenlaisia toimintamalleja, kuten kustannussäästöjä palveluiden kysynnän vaihdellessa voimakkaasti sekä mahdollisuuden hyvin nopeasti hankkia hetkellisesti käyttöön huomattavaa laskentakapasiteettia (Armbrust ym. 2010, 3).

Pilvipalveluiden hyödyntämisen etuna on erityisesti madaltunut kustannus huippukapasiteetin hankkimiseen. Huippukapasiteetilla tarkoitetaan tässä sitä laskentakapasiteettia, joka palveluiden tuottamiseen täytyy olla käytettävissä silloin kun palveluiden käyttöaste on korkeimillaan. (Schroff 2010, 64-65). Omaa konesaliaan operoiva toimija joutuu hankkimaan hyvin merkittävän määrän kapasiteettia ajallisesti hyvin lyhyen aikaa ilmenevää korkeinta käyttöastepiikkiä varten. Tällöin osa hankitusta kapasiteetista on suurimman osan ajasta täysin käyttämättömänä, eikä sen myynti ulospäin ole yleensä käytännössä mahdollista. Pilvipalveluita palvelutuotannossaan hyödyntävä toimija taas voi hyödyntää pilvipalveluita tarjoavan palveluntarjoajan kaikkia asiakkaita varten varaamaansa kapasiteettia oman tarpeensa mukaan. Tällöin kapasiteettia ei tarvitse hankkia yli tarpeen, vaan sitä voidaan joustavasti skaalata käyttöasteen mukaan.

Joissakin tapauksissa huippukapasiteetin ajallista sijoittumista on erittäin vaikea arvioida ennalta, erityisesti jos tarjottava palvelu on yleisölle avoin, julkinen palvelu. Tällöin esimerkiksi erilaiset tapahtumat (esim. toimijaa tai palvelua koskettavat uutiset), kampanjat ja niin edelleen voivat asettaa tarpeita kasvattaa kapasiteettia hyvin äkillisesti ja hyvin paljon kerrallaan (Armbrust ym. 2010, 4). Omassa konesalissaan operoivalla toimijalla ei tyypillisesti ole mahdollista vastata tällaiseen äkilliseen kysynnän kasvuun, ellei toimija ole jostain syystä hankkinut ylimääräistä kapasiteettia ennakkoon. Tämä on kuitenkin harvoin taloudellisesti kannattavaa. Tapauskohtaisesti myös uuden kapasiteetin saaminen käyttöön (palvelinten hankinta, asennukset jne.) voi viedä päiviä tai jopa viikkoja aikaa, jolloin kysyntä on saattanut jo laantua.

Edellä pilvipalveluiden hyödyntämistä on tarkasteltu lähinnä IaaS-malliin vertautuvista, raa'an laskentakapasiteetin hankinnan näkökulmasta. Sinänsä samat periaatteet pätevät myös Software as a Service -palveluita hankittaessa, jopa enenevässä määrin. Koska

kapasiteetin sijaan palveluntarjoajalta ostetaan käyttövalmista palvelua, siirtyy myös vastuu kapasiteetin hallinnasta ostajalta palveluntarjoajalle. Voidaan siis olettaa, että palveluntarjoaja pystyy jatkamaan palvelunsa toimittamista, vaikka kysyntä hetkellisesti kasvaisikin. Samoin palvelun käyttöönoton nopeus korostuu SaaS-palveluissa entisestään, parhaimillaan palvelu voi olla käytettävissä sekuntien kuluessa tilauksesta.

Hankinnan helppouden ohella Software as a Service -palveluiden etuina loppukäyttäjäorganisaatioille voidaan nähdä myös mahdollisuus tietyissä rajoissa itse muokata palvelua ilman palveluntarjoajan apua, sekä palvelun automaattinen päivittyminen ilman erikseen hankittavia versiopäivityksiä (Schroff 2010, 28). Joissakin tapauksissa SaaS-palveluntarjoajat tarjoavat myös palvelua täydentäviä lisäarvopalveluita, kuten koulutusta ja konsultointipalveluja (Luoma & Rönkkö 2011, 6). Tällainen palvelun hankinta kokonaispalveluna vähentää riskiä esimerkiksi siitä, että eri toimijat syyttelevät toisiaan konfliktitilanteissa, eikä kukaan ota vastuuta kokonaisuudesta.

3.5 Software as a Service -mallin haittapuolia loppukäyttäjäorganisaation näkökulmasta

Loppukäyttäjäorganisaation näkökulmasta palveluiden hankinta SaaS-mallilla ei ole yksiselitteisesti pelkästään hyvä asia, kolikolla on myös tummempi kääntöpuoli. Armbrust ym. (2010, 5) listaavat seuraavia asioita, joita pilvipalveluita hankkivan organisaation kannattaa arvioida ennen hankintaa:

- Palveluiden saavutettavuus, suorituskyvyn ennakoitavuus, luvattu palvelutaso
- Riippuvuussuhteen syntyminen yksittäiseen palveluntarjoajaan, datan migraatio- ja konversiovaikeudet vaihdettaessa palveluntarjoajaa
- Tietoturvallisuus ja sen auditoitavuus
- Pilvipalveluiden arkkitehtuuriin liittyvä ongelmat kuten hajautettujen järjestelmien vianmäärityksen vaikeus ja käyttöperusteisen hinnoittelun tehokkaan hyödyntämisen työläys
- Saman palveluntarjoajan muiden asiakkaiden aiheuttamat haitat, kuten IP-osoitealueiden joutuminen estolistoille epäilyttävän toiminnan seurauksena tai oman palvelun suorituskyvyn aleneminen toisen asiakkaan laskentaintensiivisen toiminnan seurauksena

Monet pilvipalvelut ovat niiden toimittajan puolesta hyvin voimakkaasti vakioituja, eikä niiden toiminnallisuutta voida ainakaan rajattomasti muokata yksittäisen loppukäyttäjäorganisaation tarpeiden mukaan. Tällöin käytännössä vaihtoehtoina ovat mukauttaa organisaation toimintaa palvelun rajoitteisiin sopivaksi tai tuottaa palvelu itse.

3.6 Software as a Service -mallin edut ja haitat palveluntarjoajalle

Software as a Service -mallin hyödyt loppukäyttäjäorganisaatiolle ovat siis monessa tapauksessa selviä. Tarvittavat sovellukset saadaan käyttöön heti, kun niitä tarvitaan ja ne säilyvät käytettävissä kasvavasta käyttöasteesta huolimatta. Mutta mitkä ovat mallin edut palveluntarjoajan näkökulmasta, eli minkä takia palveluntarjoajan kannattaa harkita palvelunsa tarjoamista Software as a Service -mallilla?

Luoma ja Rönkkö (2011, 7) mainitsevat Software as a Service -palvelumallin eduksi liiketoiminnan helpomman skaalautumisen suuremmille asiakaskunnille. Keskittymällä tarjoamaan vakiosisältöistä palvelua mahdollisimman tehokkaalla automaatiolla ja toimitusprosessilla voi pienikin yritys palvella paljon itseään suurempaa asiakaskuntaa ilman tarvetta rekrytoida lisää henkilökuntaa. Myös Hohmann (2003, 204) painottaa, että panostamalla käyttöönottoprosessin automaatioon voidaan vapauttaa rajallisia asiantuntijaresursseja rutiiniasennuksia kannattavampiin, oikeisiin asiantuntijatehtäviin sekä parantaa asiakastyytyväisyyttä käyttöönottoprojektien läpimenoajan lyhyentyessä merkittävästi.

Software as a Service -mallin edut palveluntarjoajan näkökulmasta ovat siis liiketoiminnan kannattavuuden potentiaalisessa paranemisessa. Automaation huolehtiessa rutiinitoimenpiteistä voidaan suhteellisen pienellä henkilöstömäärällä palvella suurempaakin asiakaskuntaa. Koska palveluun liittyvät rutiinitoimenpiteet, kuten uusien palvelujen käyttöönotot ja vanhojen ympäristöjen päivitykset, on automatisoitu, voivat yrityksessä työskentelevät asiantuntijat keskittyä haastavampiin ja siten tyypillisesti kannattavampiin työtehtäviin.

Pilvipalveluiden arkipäiväistyminen omalta osaltaan lisää asiakkaiden odotuksia tarjottavien palveluiden suhteen. Erikoistuneet toimijat, kuten Salesforce.com, voivat tarjota monimutkaisiakin ohjelmistoja käytettäväksi liki välittömästi tilauksen jälkeen. Tulevaisuudessa palveluntarjoajien onkin entistä vaikeampi myydä pitkiä ja kalliita ohjel-

mistojen käyttöönottoprojekteja asiakkailleen. Ohjelmiston monimutkaisuuteen tai muuhun tekniseen syyhyn vetoamisella on jatkossa entistä vähemmän pontta.

Täysin yksiselitteisesti Software as a Service -malli ei kuitenkaan ole edeltäjiään parempi. Luoma ja Rönkkö (2011, 7-8) toteavat erityisesti nk. ”Pure-play SaaS”-palveluita (hyvin vakioituja, pääasiassa verkon kautta myytäviä) tuottavien yritysten tuottavuuden olevan vertailukohtiaan heikompi. Pienillä asiakasmäärillä Software as a Service -mallin edut eivät tule esiin, mutta samaan aikaan voimakkaasti vakioidusta sovelluksesta voidaan veloittaa vain murto-osa räätälöidyn ratkaisun hinnasta.

Pienessä mittakaavassa perinteisempää sovelluspalveluntarjoaja-mallia noudattelevat toimintatavat, joissa asiakkaalle operoidaan itsenäistä, räätälöityä ympäristöä kannattavat paremmin (Luoma & Rönkkö 2011, 2). Toisaalta juuri tämä ympäristöjen itsenäisyys ja räätälöinti ovat suurimmat esteet liiketoiminnan skaalautumiselle kohti suurempaa asiakaskuntaa.

Software as a Service -malli on myös parempi vaihtoehto, jos asiakkaaksi tavoitellaan pieniä tai keskisuuria yrityksiä. Pienemmät yritykset ovat usein isoja kustannustietoisempia, mutta toisaalta helpommin mukauttavat omaa toimintaansa hankkimaansa palveluun, eikä päinvastoin. Koska palvelua ei voida tarjota pienelle yritykselle kovin suurella hinnalla, on palveluntarjoajan myös pyrittävä minimoimaan palvelun tarjoamisesta itselleen aiheutuvat kustannukset pitääkseen toimintansa kannattavana. Tämä taas ohjaa palveluntarjoajaa kohti vakioitua ja automatisoitua ympäristöä, jossa palveluntarjoaja voi mahdollisimman pienillä kustannuksilla tuoda palvelunsa piiriin uusia asiakkaita.

4 SOFTWARE AS A SERVICE -ARKKITEHTUURI

4.1 SaaS-arkkitehtuurin ominaispiirteitä

Kuten luvussa 3 todettiin, nojasivat Software as a Service -palveluntarjoajien edeltäneet sovelluspalveluntarjoajat toiminnassaan usein muiden toimijoiden toteuttamiin valmisohjelmoistoihin, joita vuokrattiin käytettäväksi sovelluspalveluntarjoajan asiakkaille. Tällainen toimintamalli kuitenkin skaalautuu heikosti, mikä puolestaan johti lopulta monen ensimmäisen aallon sovelluspalveluntarjoajan toiminnan päättymiseen.

Jäljelle jääneiden toimijoiden arkkitehtuureissa on useita yhteisiä piirteitä. Palveluita käytetään verkon yli, tyypillisesti Web-selaimella. Palvelut on myös toteutettu siten, että niiden toimintaa voidaan jossain määrin muokata loppukäyttäjän toimesta ilman, että sovelluksen lähdekoodiin tai muuhun toimintalogiikkaan täytyy tehdä muutoksia palveluntarjoajan toimesta. (Schroff 2010, 30-31). Nämä ominaisuudet helpottavat palveluntarjoajan toiminnan skaalautumista. Koska sovellusta käytetään Web-selaimella, eikä perinteisellä, paikallisesti asennettavalla asiakasohjelmalla, ei palveluntarjoajan tarvitse huolehtia sovelluksen versiopäivitysten yhteydessä asiakasorganisaatioiden asiakasohjelmien päivitysten järjestämisestä. Sovelluksen lähdekoodin ulkopuolinen sovelluksen kustomointi taas helpottaa uusien sovellusversioiden käyttöönottoa kahdella tavalla. Ensinnäkin, koska sovellusta ei ole ollut tarvetta mukauttaa lähdekooditasolla eri asiakkaiden tarpeiden mukaisesti, ei näiden mukautusten tuomisesta uusiin sovellusversioihinkaan tarvitse huolehtia. Tämä vähentää myös mukautuksista potentiaalisesti aiheutuvien ohjelmistovirheiden riskiä. Toisekseen, koska sovelluksen lähdekoodi on kaikilla asiakkailla sama, helpottaa tämä päivitysprosessia osaltaan merkittävästi. Riittää, että uusi ohjelmistoversio kopioidaan vanhan tilalle ja otetaan käyttöön.

Ehkä näitä seikkoja merkittävämpi, ja varsinaisia Software as a Service-arkkitehtuureita merkittävästi määrittävä ominaisuus on kuitenkin monitenanttisuus (engl. multitenancy). Tällä tarkoitetaan ohjelmistoarkkitehtuuria jossa useat eri loppukäyttäjäorganisaatiot jakavat saman sovellusympäristön. Sovellukseen sisäänrakennettu pääsynhallintalogiikka eriyttää eri loppukäyttäjäorganisaatiot toisistaan siten, etteivät ne pysty näkemään toistensa tietoja (Schroff 2010, 104). Monitenanttisuus on siis tietyn tyyppistä sovellustason virtualisaatiota. Periaatteessa samaan lopputulokseen, loppukäyttäjäorganisaatioiden tietojen eriyttämiseen toisistaan, päästäisiin myös järjestelmätason virtualisaatiolla,

eli tarjoamalla jokaiselle loppukäyttäjäorganisaatiolle erillinen ajoympäristö. (Schroff 2010, 104).

Palveluntarjoajan toiminnan skaalautumisen kannalta sovellustason virtualisaatio on kuitenkin huomattavasti taloudellisempaa, sillä tällöin esimerkiksi sovelluksen versiopäivitysten toteuttaminen on vaivattomampaa. Järjestelmätasolla virtualisoidussa ympäristössä versiopäivitys on tehtävä yhtä monta kertaa, kuin loppukäyttäjäorganisaatioita on palvelun käyttäjänä, sillä kaikki käyttävät omaa kopiotaan sovelluksesta.

Nämä tekijät yhdessä luovat sovellusarkkitehtuurin pohjan, jolle Software as a Service -mallilla tarjottavat palvelut tyypillisesti pohjautuvat. Tämän lisäksi Mell ja Grance (2011, 2) sekä Schroff (2010, 33) molemmat painottavat, että varsinaisen SaaS-arkkitehtuurin on kuitenkin rakennuttava skaalautuvan pilviarkkitehtuurin päälle, jotta se erottuu perinteisestä hosting-liiketoiminnasta, jossa palveluntarjoaja tarjoaa vuokrattua konesalikapasiteettia. Software as a Service -arkkitehtuuri siis rakentuu Infrastructure as a Service -arkkitehtuurin päälle siten, että sovelluksen käytettävissä olevaa kapasiteettia voidaan helposti ottaa käyttöön ja lisätä tarpeen mukaan.

Luoma ja Rönkkö (2011, 1, 4) korostavat Software as a Service -mallissa erona perinteisempään sovelluspalveluntarjoaja-malliin korkeampaa ohjelmiston vakiointitasoa, korkeaa IT-infrastruktuurin ja prosessien automaatiotasoa sekä kykyä toimittaa tilattuja palveluita mahdollisimman pienellä vaivalla suurelle asiakaskunnalle.

4.2 SaaS-kypsyysmallit

Erityisesti Software as a Service -palveluiden osalta aidon SaaS-palvelun ja ”pilvipestyn” palvelun (l. perinteisen palveluntarjoajan tarjoaman palvelun, jota markkinoidaan pilvipalveluna myynnin lisäämiseksi) raja on varsin epämääräinen. Software as a Service -kypsyysmallit ovat eräs tapa tarkastella asiaa. Niiden avulla voidaan arvioida palvelun toteuttamiseen käytettyä arkkitehtuuria ja arvioida sitä kypsyysmallin määrittelemiä ominaisuuksia vasten.

Kypsyysmallin määrittelemät ominaisuudet on määritelty vastaavina laatuominaisuuksina kuin luvussa 2.3 esitellyt laatuominaisuudetkin. Ne kuvaavat arkkitehtuurin kykyä vastata tiettyihin ei-toiminnallisiin vaatimuksiin.

Software as a Service -kypsyysmalleja ovat julkaisseet esimerkiksi Microsoft ja Forrester Research. Sisällöllisesti mallit ovat keskenään varsin saman kaltaisia, eli kypsyyden määritelmät ovat molemmissa malleissa saman kaltaiset. Tässä työssä on päädytty käyttämään Microsoftin Gianpaolo Carraron ja Frederick Chongin esittelemää mallia, muun muassa sen vapaan saatavuuden ja selkeämmän linkittymisen järjestelmäarkkitehtuurin laatuominaisuuksiin takia.

Carraro ja Chong (2006) määrittelevät hyvin suunnitellulle Software as a Service-arkkitehtuureille kolme ominaisuutta:

- Skaalautuvuus, arkkitehtuuri on skaalutuva jos käytettävissä olevien resurssien kasvattaminen kasvattaa samanaikaisten toimintojen maksimimäärää järkevässä suhteessa
- Mukautettavuus, arkkitehtuurin on mahdollistettava sovelluksen toiminnan mukauttaminen ilman sovelluksen lähdekoodin muuttamista
- Monitenanttisuus, arkkitehtuurin on mahdollistettava resurssien jakaminen tehokkaasti useiden asiakkaiden kesken

Nämä ominaisuudet voidaan nähdä varsinaisen kypsyyden tekijöinä, mutta niitä voidaan tarkastella myös ominaisuuksina sellaisenaan. Carraron ja Chongin (2006) kypsyysmalli määrittää näiden ominaisuuksien perusteella neljä eri kypsyysastetta arkkitehtuureille:

- **Taso 1**: Jokaisella asiakkaalla on oma kopionsa sovelluksesta. Sovellukseen tehtävä muuntelu voi tapahtua lähdekooditasolla. Taso 1 ei oikestaan ole Software as a Service -mallin mukainen toteutus alkuunkaan, vaan lähinnä perinteinen ASP-mallin mukainen toteutus.
- **Taso 2**: Jokaisella asiakkaalla on edelleen oma instanssi (kopio) sovelluksesta. Kaikki asiakkaat käyttävät samaa sovellusversiota, eli instanssien taustalla oleva lähdekoodi on kaikille asiakkaille sama, täten myös versiopäivitykset voivat tapahtua automaattisesti ja keskitetysti. Sovelluksen muuntelu lähdekooditasolla ei ole mahdollista, vaan sovellus mahdollistaa jonkinasteisen muuntelun esimerkiksi metadatan avulla.
- **Taso 3**: Asiakkaat, tai asiakasryhmät, jakavat keskenään sovelluksen instansseja. Olennaista on kuitenkin, että asiakkaiden ja instanssien suhde ei ole 1:1. Sama

asiakas on kuitenkin aina yhteydessä samaan instanssiin. Sovellukseen sisäänrakennettu pääsynhallintalogiikka estää asiakkaita näkemästä toistensa tietoja. Asiakkaille instanssien jakaminen ei näyttäydy mitenkään.

- **Taso 4**: Korkeimmalla kypsyystasolla toteutus on muutoin sama, kuin tasolla 3, mutta asiakkaan ja instanssin välinen 1:1-suhde on purkautunut. Mikä tahansa sovellusinstanssi voi siis palvella minkä tahansa asiakkaan pyyntöjä.

4.3 SaaS-arkkitehtuurin ja liiketoimintamallin täsmäyttäminen

Vertailtaessa Carraron ja Chongin (2006) mallin tasoja kypsyysmallin ominaisuuksiin (skaalautuvuus, muunneltavuus, monitenanttisuus) voidaan todeta erityisesti monitenanttisuuden olevan kypsän SaaS-arkkitehtuurin edellytys. Jonkinasteista kypsyyttä voidaan saavuttaa ilman monitenanttisuuttakin vakioimalla sovelluksen ohjelmakoodi kaikkien asiakkaiden kesken sekä mahdollistamalla sovelluksen muokkaaminen lähdekoodin ulkopuolelta. Näin saavutetaan kypsyysmallin taso 2, mutta ilman tukea monitenanttisuudelle ylempiä tasoja ei voida saavuttaa.

Liiketoimintamallista riippuen voi olla, että tasoa 2 korkeampia kypsyystasoja ei kannata edes tavoitella. Näin on esimerkiksi Luoman ja Rönkön (2011, 5) kuvaamien ”Enterprise SaaS” -palveluiden osalta, jossa tavoiteltu asiakaskoko on isompi ja arvonluonti perustuu pelkkää tarjottavaa sovellusta kattavampaan palvelutarjontaan koulutus-, konsultointi- ja integraatiopalveluineen.

On myös mahdollista, että arkkitehtuurissa käytettyjen ohjelmistojen tekniset rajoitteet tai lisensointimallit eivät mahdollista useamman asiakkaan sijoittamista samaan instanssiin. Tällöinkin arkkitehtuurin kypsyys jää väistämättä mallin tasolle 2. Tämä on tyypillistä erityisesti toimintamalleissa joissa osia tarjottavan palvelun komponenteista lisensoidaan muilta valmistajilta, hieman perinteisempien ASP-mallien mukaisesti.

Kuitenkin tavoiteltaessa erityisesti vaivatonta skaalautuvuutta suurille asiakasmäärille, esimerkiksi luotaessa Luoman ja Rönkön (2011, 5) kuvaamia ”Pure-play SaaS” -palveluita, joissa liiketoiminta kannattaa vasta suurilla asiakasmäärillä, on myös arkkitehtuurin kypsyydellä suurempi merkitys. Mallien määrittelemä korkein mahdollinen

kypsyystaso ei ole kuitenkaan liiketoiminnallisesta näkökulmasta aina yksiselitteisesti paras mahdollinen tai ehdottomasti tavoittelemisen arvoinen vaihtoehto.

5 SAAS-ARKKITEHTUURITYÖ KÄYTÄNNÖSSÄ

5.1 Liiketoiminnallisten tavoitteiden määrittely

Järjestelmäarkkitehtuuri ei synny tyhjiössä tai mielivaltaisesti, eikä erityisesti hyvä järjestelmäarkkitehtuuri. Jotta järjestelmä palvelee tarkoitustaan hyvin, on myös sen arkkitehtuurin oltava linjassa järjestelmän omistavan organisaation tavoitteiden kanssa (Bass & Clements 2010, 1).

Periaatteessa järjestelmän ei-toiminnalliset vaatimukset tulisi olla listattuna järjestelmän suunnitteludokumentaatiossa, osana järjestelmän vaatimusmäärittelyä. Kuitenkin tyypillisesti vaatimusmäärittely keskittyy kuvaamaan järjestelmän toiminnallisia vaatimuksia, eli sitä, mitä järjestelmän on tehtävä. Vaatimusmäärittelyt harvoin ottavat kantaa siihen, *miten* järjestelmän on suoritettava sille suunnitellut tehtävät. Ei-toiminnalliset vaatimukset ja arkkitehtuurin rajoitteet ovat usein haudattuna järjestelmää kehittävän organisaation liiketoiminnallisiin tavoitteisiin ja liiketoimintastrategiaan, josta ne eivät löydä tietään eksplisiittisessä muodossa järjestelmän suunnittelu- ja vaatimusmäärittelydokumentteihin (Bass & Clements 2010, 1).

Parhaassa tapauksessa korkeatasoinen vaatimusmäärittely sisältää yksiselitteisesti myös järjestelmän ei-toiminnalliset vaatimukset sekä arkkitehtuurin kohdistuvat rajoitteet. Jos näin ei kuitenkaan ole, Bass ym. (2013, 292-296) ehdottavat ei-toiminnallisten vaatimusten kartoittamiseen kahta menetelmää:

- Vaatimusmäärittelydokumenttien läpikäynti, ei-toiminnallisia vaatimuksia saattaa olla kuvattuna implisiittisesti määrittelyjen lomassa
- Sidosryhmähaastattelut, järjestelmän tulevat käyttäjät ja muut järjestelmään liittyvät tahot pystyvät kuvaamaan heille tärkeitä asioita järjestelmän suhteen

Jos sidosryhmiä on paljon, tai on syytä olettaa eri sidosryhmillä olevan keskenään ristiriitaisia tavoitteita, voi olla järkevää järjestää sidosryhmien edustajat ja järjestelmän arkkitehtuurista vastaavat tahot laatuominaisuustyöpajaan (engl. Quality Attribute Workshop, QAW). Laatuominaisuustyöpajan tarkoituksena on määritellä ja priorisoida järjestelmään kohdistuvat ei-toiminnalliset vaatimukset ja arkkitehtuurin rajoitteet siten, että työpajassa aikaansaadaan järjestelmän ei-toiminnallisia vaatimuksia koskettava

vaatimusmäärittely, jonka pohjalta varsinainen arkkitehtuurisuunnittelu voidaan aloittaa. (Bass ym. 2013, 294-296). Työpajavetoisen määrittelytyön etuna ovat paremmat edellytykset arkkitehtuuriin liittyvien konfliktitilanteiden ratkaisuun. Jos kaikki sidosryhmät haastatellaan erikseen, voivat he priorisoida järjestelmän eri laatuominaisuuksia hyvin poikkeavalla tavalla. Kuten luvussa 2.3 todettiin, arkkitehtuuri pystyy harvoin täyttämään ilman kompromisseja kaikki siihen kohdistuvat ei-toiminnalliset vaatimukset. Tällöin työpajavetoinen ei-toiminnallisten vaatimusten määrittely mahdollistaa tarvittavien kompromissien huomioimisen jo ennen varsinaisen arkkitehtuurin suunnittelutyön aloittamista.

Tämän opinnäytetyön kohteena olevan järjestelmäarkkitehtuurin osalta liiketoiminnalliset tavoitteet määriteltiin pääosin vuoden keväällä 2013 pidetyssä työpajassa. Työpajaan osallistui tämän työn tekijän ohella yrityksen toimivaa johtoa sekä teknisiä asiantuntijoita. Työpajassa määriteltiin tärkeimmiksi liiketoiminnallisiksi tavoitteiksi:

- Asiantuntijaresurssien vapauttaminen rutiiniasennustöistä vaativampiin tehtäviin
- Muutoksen ennakointi asiakkaiden halukkuudessa maksaa järjestelmien käyttöönotosta
- Liiketoiminnan paremman skaalautuvuuden mahdollistaminen suuremmille asiakasmäärille
- Verkkomyyntikanavan avaaminen Ambientian myyntiorganisaation tueksi
- Kilpailukyvyn säilyttäminen pilvipalveluiden yleistyessä IT-alan palvelutarjonnassa
- Osaaminen kasvattaminen pilvipalveluiden toteuttamisessa ja ymmärryksessä niiden arkkitehtuurista

5.2 Arkkitehtuurin rajoitteiden kartoittaminen

Arkkitehtuurin rajoitteet määrittyvät myös liiketoimintalähtöisesti, eli liiketoiminnallisten tavoitteiden määrittelyn kautta. Liiketoiminnallisesta näkökulmasta ei-toiminnallisilla vaatimuksilla ja rajoitteilla ei ole luokittelullista eroa: osasta liiketoiminnallisia tavoitteita seuraa ei-toiminnallisia vaatimuksia ja osasta rajoitteita.

Arkkitehtuurin rajoitteet kartoitettiin samassa, vuoden 2013 keväällä pidetyssä työpajassa. Käytännössä oleellisimmat rajoitteet liittyivät teknologiavalintoihin, eli toteutettavan

järjestelmäarkkitehtuurin oli toimittava Ambientialla jo käytössä olevien sovellusten, kuten Atlassianin Confluence-wikin ja JIRA-tehtävänhallintajärjestelmän kanssa. Järjestelmäarkkitehtuurin haluttiin olevan myös helposti laajennettavissa muihin Ambientian tarjonnassa oleviin sovelluksiin, kuten Liferay Portaliin.

5.3 Ei-toiminnallisten vaatimusten johtaminen liiketoiminnallisista tavoitteista

Monet luvussa 5.1 kuvatuista liiketoiminnallisista tavoitteista eivät suoraan kosketa järjestelmän arkkitehtuuria. Myös Bass ja Clements (2010, 21) toteavat, että kaikki liiketoiminnalliset tavoitteet eivät määritä ei-toiminnallisia vaatimuksia. Niiltäkin osin, kun liiketoiminnalliset tavoitteet määrittävät ei-toiminnallisia vaatimuksia, on tehtävä jonkin verran tulkintaa liiketoiminnallisen tavoitteen ja siitä seuraavien arkkitehtuuristen vaatimusten välillä. Liiketoiminnallisten tavoitteiden sovittaminen ei-toiminnallisiksi vaatimuksiksi onkin yksi keskeinen osa arkkitehtuurityötä.

Taulukko 1 kuvaa liiketoiminnallisista tavoitteista johdettuja ei-toiminnallisia vaatimuksia ja niihin liittyviä laatuominaisuuksia.

TAULUKKO 1. Liiketoiminnallisista tavoitteista johdetut ei-toiminnalliset vaatimukset

Ei-toiminnallinen vaatimus	Laatuominaisuus
Uuden instanssin käyttöönotto on oltava mahdollisimman nopeaa ja automaattista.	Käyttöönotettavuus
Olemassaolevan instanssin päivittäminen on oltava mahdollisimman nopeaa ja automaattista.	Käyttöönotettavuus
Arkkitehtuurin on oltava yhteensopiva mahdollisimman monien Ambientialla käytettyjen sovellusten kanssa.	Muunneltavuus
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten suorituskykyyn.	Suorituskyky
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten saavutettavuuteen.	Saavutettavuus
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten tietoturvallisuuteen.	Tietoturvallisuus
Arkkitehtuurin on ilmennettävä Software as a Service -arkkitehtuurien parhaita käytäntöjä.	SaaS-kypsyys

Luvussa 2.3 kuvattujen, kanonisten laatuominaisuuksien lisäksi arkkitehtuurille määritettiin kaksi muutakin laatuominaisuutta:

- Käyttöönotettavuus, järjestelmän käyttöönoton helppous
- SaaS-kypsyys, komposiittiominaisuus joka kuvaa Carraron ja Chongin (2006) SaaS-kypsyysmallin mukaisten laatuominaisuuksien (skaalautuvuus, mukautettavuus, monitenanttisuus) kautta arkkitehtuurin kypsyyttä Software as a Service -arkkitehtuurina

Luvussa 2.3 kuvatut laatuominaisuusluettelot eivät siis ole mitenkään täydellisiä luetteloita, vaan arkkitehtuuria suunniteltaessa on toisinaan määriteltävä uusia laatuominaisuuksia heijastamaan liiketoiminnallisia tavoitteita. Liiketoiminnallisia tavoitteita ei siis kannata pakottaa sopimaan johonkin yleisesti hyväksyttyyn laatuominaisuuteen, vaan sen sijaan määritellä tavoitetta vastaava, uusi laatuominaisuus. Uuden laatuominaisuuden määrittely vaatii kuitenkin jonkin verran työtä. Jotta uudella laatuominaisuudella olisi arkkitehtuurisuunnittelun näkökulmasta käyttöarvoa, on ensinnäkin määriteltävä mitä laatuominaisuudella tarkoitetaan ja miten sen läsnä- tai poissaolo arkkitehtuurissa ilmenee. Kun laatuominaisuuden sisältö on täsmennetty, voidaan määritellä arkkitehtuurilliset taktiikat joilla kyseistä laatuominaisuutta pyritään tukemaan. Myös uuden laatuominaisuuden vaikutukset muihin laatuominaisuuksiin on tunnistettava. (Bass ym. 2013, 194-199).

Tämän työn puitteissa käyttöönotettavuutta tarkastellaan erityisesti uuden palveluinstanssin käyttöönottoon kuluvan ajan suhteen. Käyttöönottoon kuluva aika on siis laatuominaisuuteen käyttöönotettavuus liitetty suure, jolla käyttöönotettavuutta mitataan. Tavoitteena on luonnollisesti mahdollisimman lyhyt käyttöönottoaika per instanssi. Käyttöönotettavuuden katsottiin vaikuttavan kasvattavasti järjestelmän monimutkaisuuteen, joka taas vaikuttaa muunneltavuuteen heikentävästi (Bellomo, Ernst, Nord & Kazman 2014). Tämä aiheutuu korkean käyttöönotettavuuden edellyttämästä, useita eri osa-alueita automatisoivasta, monesta järjestelmästä koostuvasta arkkitehtuurista. Tällöin myös arkkitehtuuria koskettavat muutokset saattavat vaikuttaa useisiin eri järjestelmiin.

Laatuominaisuutena SaaS-kypsyys kuvaa arkkitehtuurin kypsyyttä tarkasteltuna Carraron ja Chongin (2006) SaaS-kypsyysmallin tasoja vasten. Arkkitehtuuri on sitä SaaS-

kypsempi, mitä paremmin se skaalautuu, mitä paremmin siinä ajettavat sovellukset ovat käyttäjän konfiguroitavissa ja mitä paremmin se pystyy majoittamaan useita käyttäjiä saman sovelluksen sisällä. Tämä laatuominaisuus haluttiin ottaa mukaan erityisesti, koska pyrittiin tietoisesti luomaan olemassaolevien sovellusten ympärille Software as a Service -arkkitehtuuri, jolloin SaaS-arkkitehtuureille ominaiset piirteet tulee huomioida mahdollisimman hyvin. SaaS-kypsyyden mukaan ottaminen arkkitehtuurin laatuominaisuutena tukee myös hyvin liiketoiminnallista tavoitetta kasvattaa ymmärrystä pilvipalveluiden arkkitehtuurista. Korkea SaaS-kypsyys laatuominaisuutena vaikuttaa parantavasti moniin muihin laatuominaisuuksiin, kuten suorituskykyyn ja saavutettavuuteen. Hyvin skaalautuvan palvelun suorituskykyä on helppo lisätä kasvattamalla palvelua suorittavien palvelinten määrää. Kypsässä, monitenanttisessa palvelussa yksittäinen asiakas ei ole sidottu yksittäiseen palvelua suorittavaan palvelimeen, joten yksittäisen palvelimen vikaantuminen ei vaikuta palvelun saavutettavuuteen. Muunneltavuuteen SaaS-kypsyys vaikuttaa monimutkaisemmalla tavalla. Toisaalta se edellyttää arkkitehtuurilta korkeatasoista muunneltavuutta (mukautettavuutta), mutta toisaalta rajoittaa tapoja joilla muuntelua voidaan tehdä, sillä korkeilla SaaS-kypsyyden tasoilla instanssien lähdekoodiin ei voi tehdä muutoksia.

5.4 Ei-toiminnallisiin vaatimuksiin vastaaminen prototyyppi- ja pilottivaiheissa

Kuten luvussa 2.4 todettiin, ei-toiminnallisiin vaatimuksiin vastataan käyttöönottamalla arkkitehtuurissa erilaisia arkkitehtuurisia taktiikoita. Käytännön työtä aloitettaessa ei kuitenkaan ollut kovin täsmällisesti selvillä millaisia arkkitehtuurisia taktiikoita voitaisiin soveltaa paremman käyttöönotettavuuden saavuttamiseksi, ja miten paljon nämä taktiikat vaikuttaisivat muihin laatuominaisuuksiin, kuten muunneltavuuteen. Korkea käyttöönotettavuus oli kuitenkin koko arkkitehtuurityön perimmäinen tavoite, joten työ aloitettiin se edellä. Aivan ensimmäinen prototyyppi tehtiin Ambientian Tampereen toimistolla keväällä 2012 pidetyssä niin sanotussa ShipIt Dayssa.

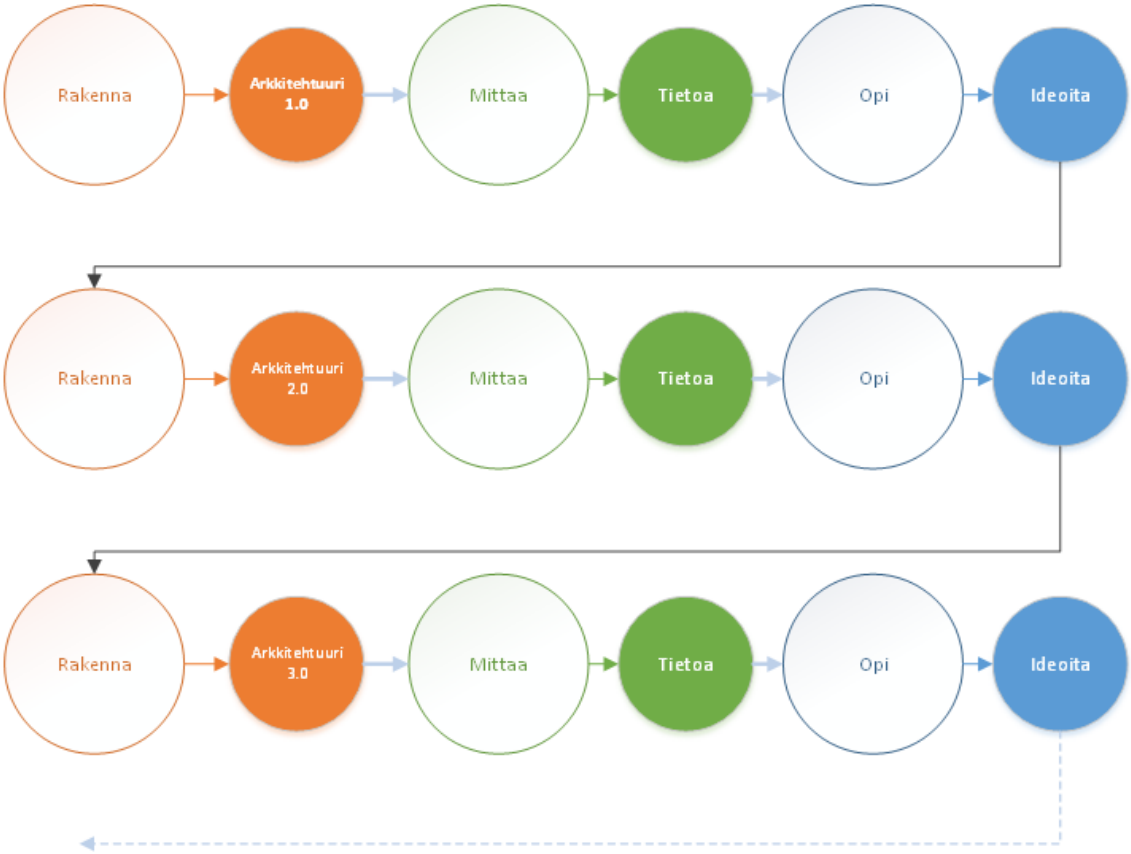
ShipIt Day on noin vuorokauden mittainen tapahtuma, jonka aikana siihen osallistuvat voivat toteuttaa tai kokeilla haluamiaan asioita. ShipIt Day tarjoaa mahdollisuuden kokeilla uusia teknologioita tai radikaaleja ajatuksia, joiden testaamiselle ei muutoin välttämättä löytyisi aikaa. Käytäntö on saanut alkunsa australialaislähtöisessä ohjelmistoyrityksessä Atlassianissa. (Atlassian, 2014). Ambientialle ShipIt Day on löytänyt tiensä

Ambientian ja Atlassianin kumppanuuden kautta. Alun perin kyse oli siis puhtaasti teknisestä kokeilusta, jonka liiketoiminnalliset mahdollisuudet kirkastuivat myöhemmin.

Ensimmäinen prototyyppi osoitti, että tietyin reunaehdoin olisi mahdollista suorittaa Atlassian Confluencen lähes täysin automaattinen asennus hyvin lyhyessä ajassa hyödyntämällä käyttöjärjestelmän paketinhallintatyökaluja. Prototyyppi oli siinä määrin lupaava, että prototyypin kehittelyä jatkettiin kuudella työpajapäivällä vuoden 2012 aikana. Työpajoihin osallistui tämän työn kirjoittajan lisäksi toinen tekninen asiantuntija, jonka kanssa prototyyppiä kehitettiin eteenpäin. Prototyyppivaiheen aikana käyttökelpoiset arkkitehtuurilliset taktiikat täsmentyivät.

Pilottivaihe käynnistettiin keväällä 2013 pidetyn työpajan jälkeen, jolloin virallisesti päätettiin siirtyä pois prototyyppivaiheesta ja tarjota arkkitehtuuriin perustuvia tuotteita asiakkaille. Pilottivaiheen käynnistyessä arkkitehtuuri oli paikoin puutteellinen ja joitakin suunnittelupäätöksiä ei oltu lyöty lukkoon. Pilottivaihe päätettiin toteuttaa hyödyntämällä Minimum Viable Product -menetelmää.

Minimum Viable Productissa pyritään tuottamaan yksinkertaisin järkevä tuote, jota voidaan tarjota ulospäin. Menetelmän tarkoituksena on mahdollisimman nopeasti päästä sisälle rakenna-mittaa-opi -silmukkaan, jossa tuotetta voidaan iteratiivisesti parantaa saaden siitä samalla jatkuvasti palautetta. Menetelmä on omiaan sopimaan juuri pilottivaiheeseen, sillä siinä on mahdollista testata teknisten aspektien ohella myös liiketoiminnallisia olettamuksia (Ries 2011, 93-94). Tässä työssä pyrittiin ensin tuottamaan yksinkertaisin järkevä arkkitehtuuri, jonka avulla oli mahdollista asentaa ja päivittää sovellusinstansseja nopeammin kuin käsityönä. Arkkitehtuuria kehitettiin iteratiivisesti pilottivaiheen aikana saatujen kokemusten perusteella. Kuva 2 havainnollistaa rakenna-mittaa-opi silmukan ja iteratiivisen uusien versioiden julkaisun välistä suhdetta.



KUVA 2. Rakenna-mittaa-opi -silmukka ja iteratiivinen arkkitehtuurityö (Ries 2011, 74, muokattu)

Pilottivaiheen aikana saadun kokemuksen perusteella arkkitehtuurillisia taktiikoita täsmennettiin edelleen siten, että niitä käyttämällä pystyttiin saavuttamaan arkkitehtuurisesti eheä kokonaisuus. Pilottivaiheen voidaan katsoa jatkuneen vuoden 2014 kevääseen asti, jonka jälkeen arkkitehtuuriin ei ole enää tehty merkittäviä muutoksia. Pilottivaiheelle ei sinänsä määritelty selkeää päätöspistettä.

Taulukossa 2 kuvataan luvussa 5.3 esitellyt ei-toiminnalliset vaatimukset ja prototyyppi- ja pilottivaiheissa päätetyt taktiikat, joilla niihin päädyttiin vastaamaan pilottivaiheen jälkeen.

TAULUKKO 2. Taktiikat, joilla ei-toiminnallisiin vaatimuksiin vastattiin

Ei-toiminnallinen vaatimus	Laatuominaisuus	Käytetyt arkkitehtuurilliset taktiikat
Uuden instanssin käyttöönotto on oltava mahdollisimman nopeaa ja automaattista.	Käyttöönotettavuus	- Moduulien yleistäminen - Yleiskäyttöisten palveluiden eristäminen - Moduulien parametrointi - Resurssitiedostojen ja konfiguraatiohallinnan käyttö
Olemassaolevan instanssin päivittäminen on oltava mahdollisimman nopeaa ja automaattista.	Käyttöönotettavuus	- Moduulien yleistäminen - Yleiskäyttöisten palveluiden eristäminen - Moduulien parametrointi
Arkkitehtuurin on oltava yhteensopiva mahdollisimman monien Ambientialla käytettyjen sovellusten kanssa.	Muunneltavuus	- Moduulien yleistäminen - Yleiskäyttöisten palveluiden eristäminen - Moduulien parametrointi - Resurssitiedostojen ja konfiguraatiohallinnan käyttö
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten suorituskykyyn.	Suorituskyky	- Ilmentymien eriyttäminen
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten saavutettavuuteen.	Saavutettavuus	- Ilmentymien eriyttäminen
Arkkitehtuuri ei saa vaikuttaa heikentävästi sovellusten tietoturvallisuuteen.	Tietoturvallisuus	- Ilmentymien eriyttäminen
Arkkitehtuurin on ilmennettävä Software as a Service -arkkitehtuurien parhaita käytäntöjä.	SaaS-kypsyys	- Moduulien yleistäminen - Yleiskäyttöisten palveluiden eristäminen - Moduulien parametrointi - Resurssitiedostojen ja konfiguraatiohallinnan käyttö

Moduulien yleistämisellä tarkoitetaan moduulin sisäisen koheesion kasvattamista ja moduulien välisen kytkennän (engl. coupling) vähentämistä. Moduulin koheesio on sitä korkeampi, mitä tiiviimmin moduulin toteuttamat vastuut liittyvät toisiinsa (Bachmann, Bass & Nord 2007, 10-11). Moduulien välisellä kytkennällä puolestaan tarkoitetaan moduulien välistä yhteyttä, jossa muutos moduulissa A edellyttää muutosta heijastavia muutoksia moduulissa B. (Bachmann ym. 2007, 9-10). Hyvä esimerkki korkeasta koheesiosta ja alhaisesta kytkennästä ovat Unix-filosofian mukaiset komentorivityökalut,

jotka tekevät yhden asian hyvin, mutta toisaalta mahdollistavat liittymisen toisiinsa (Kernighan & Pike 1984, viii). Tässä työssä moduulien yleistämistä hyödynnettiin käyttämällä käyttöjärjestelmän paketinhallintatyökaluja sovellusbinäärien asentamiseen. Jokaisesta sovelluksesta pyrittiin luomaan mahdollisimman yleiskäyttöisiä moduuleita, joiden asennusta on helppo monistaa, näin saavuttaen korkea käyttöönotettavuus. Paketinhallintaa käsitellään tarkemmin luvussa 6.2.

Yleiskäyttöisten palveluiden eristämisellä (engl. abstract common services) tarkoitetaan taktiikkaa, jossa useissa moduuleissa esiintyvä vastuu irroitetaan alkuperäisistä moduuleista ja viedään se erilliseen, yleiskäyttöiseen moduuliin, jonka palveluita alkuperäiset moduulit jatkossa hyödyntävät. Näin vähennetään tarvetta toistaa samaa muutosta useaan moduuliin, tarvittavat muutokset voidaan suorittaa keskitetysti yhteen paikkaan. Yleiskäyttöinen moduuli tuodaan sen palveluja tarvitsevan moduulin käyttöön hyödyntämällä joitakin viivästetyn sidonnan menetelmiä. (Bachmann ym. 2007, 17). Tässä työssä yleiskäyttöisten palveluiden eristämistä hyödynnettiin muun muassa irroittamalla useiden sovellusten (mm. Atlassian JIRA, Atlassian Confluence ja Liferay Portal) toimitettava Apache Tomcat -sovelluspalvelin omaksi paketikseen, joka jaellaan ympäristöihin sovelluksesta erillään käyttöjärjestelmän paketinhallintatyökaluilla.

Viivästetyllä sidonnalla (engl. defer binding) tarkoitetaan ryhmää menetelmiä, joilla voidaan siirtää sidonnan, esimerkiksi jonkin toiminteen arvon tai käyttöönoton ajankohtaa myöhemmäksi sovelluksen elinkaaressa (Bass ym. 2013, 124-125). Bachmann ym. (2007, 13-14) listaavat seuraavia kohtia sovelluksen elinkaaressa jolloin sidontaa voi tapahtua:

- Ohjelmoinnin aikana
- Kääntämisen aikana
- Käyttöönoton aikana
- Alustamisen aikana
- Ajon aikana

Muutos on tyypillisesti helpompi tehdä myöhemmin sovelluksen elinkaaressa, mutta se edellyttää tämän sallivien toiminteiden ja ominaisuuksien olemassaoloa (kuten tukea liitännäisille), joiden toteuttaminen taas vaatii työtä, eli nostaa toteutuksen kokonaiskustannuksia (Bachmann ym. 2007, 11; Bass ym. 2013, 122, 125). Tässä työssä viivästettyä

sidontaa hyödynnettin ensin parametroimalla moduuleita, jolloin sidonta tapahtui kääntämisen aikana. Tämä todettiin kuitenkin pilottivaiheen aikana varsin skaalautumattomaksi menetelmäksi ympäristömäärän kasvaessa. Sidontaa päätettiin viivästää käyttöönottovaiheeseen muuntamalla moduuleita yleiskäyttöisemmiksi ja hyödyntämällä konfiguraationhallintajärjestelmää tarvittavan ympäristökohtaisen muuntelun aikaansaamiseksi. Konfiguraationhallintaa kuvataan tarkemmin luvussa 6.3.

Bellomo ym. (2014, 3-5) esittävät, että käyttöönotettavuuteen liittyvät arkkitehtuurilliset taktiikat ovat lähinnä mukautuksia kanonisempiin laatuominaisuuksiin liittyvistä taktiikoista. Esimerkiksi konfiguraationhallintajärjestelmän käyttö käyttöönotettavuuden nopeuttamiseen voidaan nähdä yhdistelmänä sidonnan viivästämisen muunneltavuustaktiikoita ja testattavuustaktiikkaa *hiekkalaatikko*. Hiekkalaatikko on arkkitehtuurillinen taktiikka, jossa testaamisen helpottamiseksi luodaan järjestelmästä niin sanottu hiekkalaatikkoympäristö, joka sisältää kaikki ympäristössä esiintyvät toiminteet. Joitakin toiminteita (kuten integraatioita) saatetaan jäljitellä tarkoitusta varten luodulla ohjelmakoodilla (Bass ym. 2013, 165-166). Taktiikan perustavoite on kuitenkin sama: toistettava ja helppo tuotantoympäristöä muistuttavan ympäristön käyttöönotto.

Suorituskykyyn, saavutettavuuteen ja tietoturvallisuuteen vastataan hyödyntämällä tietoturvallisuuteen liittyvää taktiikkaa *ilmentymien eriyttäminen* (engl. separate entities). Ilmentymien eriyttämisellä tarkoitetaan ilmentymien loogista tai fyysistä eriyttämistä toisistaan siten, että eri ilmentymät eivät voi kommunikoida keskenään (Bass ym. 2013, 153). Koska prototyyppi- ja pilottivaiheissa käytetyt sovellukset (Atlassian JIRA ja Atlassian Confluence) eivät tue teknisesti monitenanttisuutta, toteutettiin eriyttäminen asentamalla jokainen palvelun asiakaskohtainen instanssi (ilmentymä) omaan virtuaalikoneeseensa. Näin saavutetaan varsin korkea eriyttämisen aste, joskin aavistuksen korkeammalla infrastruktuurin resurssien käytöllä. Sovellusten lisensointimallit eivät myöskään mahdollista monitenanttisuutta (sovellukset lisensoidaan loppukäyttäjäorganisaatiokohtaisesti), joten mahdollisuutta toteuttaa jonkinasteista monitenanttisuutta teknisistä rajoitteista huolimatta ei päätetty tutkia.

6 KUVAUS TOTEUTETUSTA SAAS-ARKKITEHTUURISTA

6.1 Arkkitehtuurin avainkomponentit

Bellomo ym. (2014, 5) esittävät ajatuksen tiiviimmin integroiduista komponenteista ympäristöissä, jotka tavoittelevat erittäin korkeaa käyttöönotettavuutta. Perinteisesti tarkasteltuna IT-ympäristöissä on ollut selvä jakolinja varsinaisen IT-infrastruktuurin ja sen päälle käyttöönotettavan sovelluksen välillä. Jakolinjan molemmilla puolilla on ollut selvästi toisistaan erilliset vastuut. Tämä aiheuttaa epäjatkuvuuskohdan käyttöönottoprosessiin, jossa IT-infrastruktuurista vastaava taho ottaa ensin käyttöön varsinaisen ympäristön (esim. virtuaalipalvelimen) omilla työkaluillaan ja sovelluksesta vastaava taho sovelluksen omilla työkaluillaan. Tavoiteltaessa korkeaa käyttöönotettavuutta tällaisesta jakolinjasta on luovuttava ja pyrittävä luomaan mahdollisimman saumaton ja jatkuva käyttöönottoprosessi.

Tässä luvussa tarkastellaan tällaisen saumattoman käyttöönottoprosessin toteuttamiseen tarvittavia komponentteja, jotka yhdessä muodostavat korkean käyttöönotettavuuden järjestelmäarkkitehtuurin. Käyttöönotettavuuden eri puolien tarkastelun helpottamiseksi on ensin määriteltävä, mitä ollaan ottamassa käyttöön ja millaisia menetelmiä siihen voidaan hyödyntää. Kuten luvussa 5.3 todettiin, tämän arkkitehtuurin puitteissa uusi sovellusinstanssi otetaan aina käyttöön uudessa virtuaalikoneessa, jolloin jokainen käyttöönotto alkaa uuden virtuaalikoneen käyttöönotosta.

Virtuaalikoneen lisäksi on käyttöönotettava itse sovellus, joka koostuu binääreistä (sovelluksen logiikan sisältävä ohjelmakoodi), konfiguraatiosta (ohjelmakoodin suoritusta ohjaava tieto) ja sovelluksen käsittelemästä ja tuottamasta datasta (Humble & Farley 2011, 39, 295-296). Koska tässä työssä käsitellään erityisesti uusien ympäristöjen käyttöönottoa, käyttöönottovaiheessa ei tarvitse huolehtia sovelluksen datan hallinnasta. Uusi sovellus ei ole vielä tuottanut dataa, joten sen kopioinnista uuteen ympäristöön käyttöönoton yhteydessäkään ei tarvitse huolehtia.

Sovelluksen käyttöönotto edellyttää siis kolmivaiheista prosessia: tarvittavan infrastruktuurin (virtuaalikoneen) provisiointia, sovelluksen binäärien asentamista provisioituun infrastruktuuriin ja sovelluksen konfiguraatioparametrien määrittämistä (Humble &

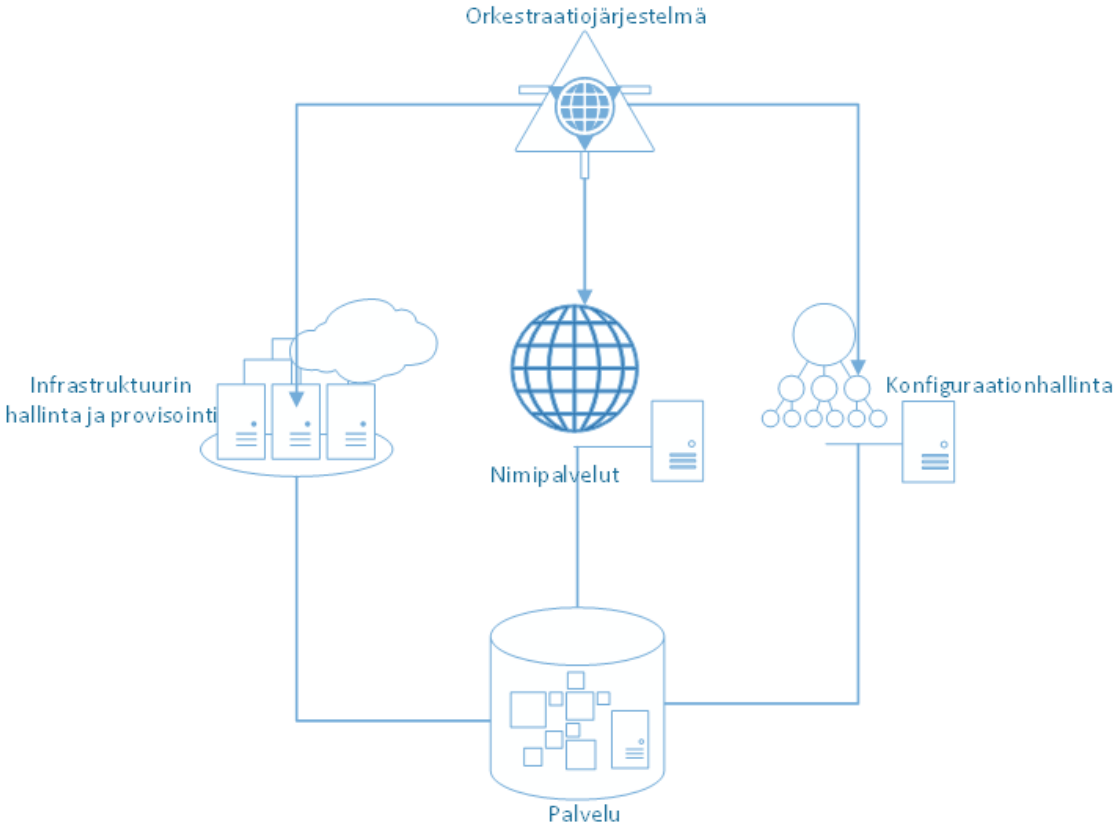
Farley 2011, 25, 277). Nämä toimenpiteet täytyy pystyä toteuttamaan koordinoidusti yhden liitäntäpisteen kautta, jotta prosessiin ei synny epäjatkuvuuskohtia.

6.2 Orkestrointi

Orkestroinnilla tarkoitetaan koko yksittäisen ympäristön elinkaarta hallinnoivaa komponenttia, joka vastaa koordinoinnista ympäristön eri osia konfiguroivien komponenttien kanssa siten, että ne yhdessä tuottavat halutun lopputuloksen, eli käyttövalmiin palvelun (Benatallah & Ranjan 2012, 1; Alcaraz Calero ym. 2010, 2; OpenStack Orchestration).

Orkestrointijärjestelmällä on oltava liitännäisyys kaikkiin niihin järjestelmiin, jotka osallistuvat palvelun käyttöönottoprosessiin. Tyypillisesti tällaisia palveluita ovat esimerkiksi infrastruktruurin hallintapalvelut ja konfiguraationhallinta. Orkestrointi siis ohjaa näiden palveluiden toimintaa. Käyttöönottoon liittyvät työnkulut on siis konfiguroitava orkestrointipalveluun.

Joissakin tapauksissa on mahdollista käyttää käyttöympäristöön integroitua orkestraatiota, kuten Amazon Web Servicesin (AWS) yhteydessä Amazon CloudFormationia tai OpenStack-pohjaisissa ympäristöissä OpenStack Heatia. Ambientia ei kuitenkaan tällä hetkellä hyödynnä omassa palvelutuotannossaan sellaista käyttöympäristöä, joka tarjoaisi tällaisen integroidun orkestraatiokomponentin. Sen sijaan tämä komponentti on toteutettu talon sisällä, kirjoittamalla Python-ohjelmointikielellä liitännät niiden järjestelmien rajapintoihin, joihin orkestraation täytyy ulottua. Tällä hetkellä orkestraatiojärjestelmällä ohjataan infrastruktuurin provisointia, nimipalveluita ja konfiguraationhallintaa. Kuva 3 esittää orkestraatiopalvelun suhdetta muihin palveluihin tämän työn puitteissa tehdyssä toteutuksessa.



KUVA 3. Orkestraatiopalvelu ohjaa muita palvelun käyttöönotossa tarvittavia komponentteja.

6.3 Infrastruktuurin hallintapalvelut

Infrastruktuurin hallintapalveluilla ohjataan erilaisten työnkulkujen kautta infrastruktuuriresurssien (laskentateho, tallennustila, nimipalvelut jne.) käyttöönottoa automatisoidusti (Alcaraz Calero ym. 2010, 2). Infrastruktuurin hallintapalveluita ohjataan orkestraatiopalvelun kautta.

Infrastruktuurin hallintapalvelut voivat sisäisesti koostua useista erillisistä komponenteista, jotka vastaavat eri osista infrastruktuuria (OpenStack Logical architecture). Toteutetussa arkkitehtuurissa infrastruktuurin hallinnasta vastaa VMwaren vRealize Orchestrator ja sen taustalla VMware vSphere -pohjainen virtuaali-infrastruktuuri. Nämä edustavat kuvassa 3 komponenttia ”*Infrastruktuurin hallinta ja provisointi*”.

6.4 Paketinhallinta

Paketinhallinnalla tarkoitetaan järjestelmää, jolla on kyky asentaa ja päivittää ohjelmistoja verkon yli muuhun käyttöjärjestelmään saumattomasti integroidulla tavalla. Käytäntö on erityisen yleinen Linux-pohjaisissa käyttöjärjestelmissä, joissa tyypillisesti kaikki käyttöjärjestelmän komponentit ja niiden päälle asennettavat palvelut, kuten web-palvelimet ja relaatiotietokannat, asennetaan käyttämällä samaa paketinhallintajärjestelmää (Murdock 2007; Alcazar Calero ym. 2010, 2-3).

Paketinhallintajärjestelmä on asiakas-palvelin -arkkitehtuurin mukainen järjestelmä, joka koostuu komponenttirepositoriosta (palvelinkomponentti) sekä palvelimilla suoritettavasti paketinhallintaohjelmistosta (asiakaskomponentti), joka on yhteydessä komponenttirepositorioon. Paketinhallintaohjelmiston saadessa käskyn asentaa tai päivittää paketti, ottaa se yhteyttä komponenttirepositorioon tarkistaakseen onko pakettia saatavilla tai onko siitä saatavilla tuoreempia versioita (Alcazar Calero ym. 2010, 2-3). Tyypillisesti paketinhallintajärjestelmää ohjataan joko suoraan orkestraatiokerroksesta, tai konfiguraationhallintajärjestelmän kautta.

Paketinhallintajärjestelmällä suoritetaan luvussa 6.1 kuvatuista sovelluksen käyttöönottovaiheista keskimmäinen, eli sovelluksen binäärien kopiointi kohdealustalle. Ambientia toimii pääsääntöisesti Linux-pohjaisten ympäristöjen kanssa, joten sovellusten paketinhallintajärjestelmäksi valittiin yleisimmin ympäristöissä käytetyn Red Hat Enterprise Linuxin kanssa yhteensopiva RPM. RPM on myös Linux Foundationin Linux Standard Basessa (Linux-jakeluille tarkoitettu standardi tiettyjen jakelun osien standardinmukaisesta toteuttamisesta) suosittelema paketointiformaatti (Linux Foundation 2010, 652). Myös Humble ja Farley (2011, 154-155) suosittelevat käyttöjärjestelmän paketointityökalujen käyttöä julkaistaessa sovelluksia tuotantoympäristöihin, mainiten hyviksi puoliksi mm. hyvän integraation käyttöjärjestelmän muuhun paketinhallintaan ja konfiguraationhallintaan. Hyödyntämällä paketinhallintaa sovellukselle voidaan määritellä esimerkiksi riippuvuuksia käyttöjärjestelmän komponentteihin, jolloin käsky paketinhallintaohjelmistolle asentaa sovellus tuo myös tarvittavat käyttöjärjestelmätason kirjastot paikalleen automaattisesti. Tämä helpottaa käyttöönotettavuutta ja vähentää riskiä käyttöönoton epäonnistumisesta puuttuvien riippuvuuksien vuoksi.

Jokaista sovellusta varten on luotu RPM-paketti, joka tuo palvelimelle sovelluksen tiedostot ennalta määritettyyn hakemistoon. Hyödyntämällä konfiguraationhallintajärjestelmää määritellään ympäristökohtaisia asetuksia, kuten Java-virtuaalikoneen muistiasetuksia sekä URL-osoite, josta kyseinen palveluinstanssi halutaan vastaamaan selaimelle.

Paketinhallintajärjestelmää ohjataan Ambientian toteutuksessa konfiguraationhallintajärjestelmän kautta. Orkestraatio ei siis suoraan liity paketinhallintajärjestelmään, kuten esim. Alcazar Caleron ym. (2013, 2-3) kuvaamassa toteutuksessa. Tämän päätöksen takana on ollut tavoite pitää orkestraatiojärjestelmä mahdollisimman yksinkertaisena vähentämällä sen liitäntäpisteitä.

6.5 Konfiguraationhallinta

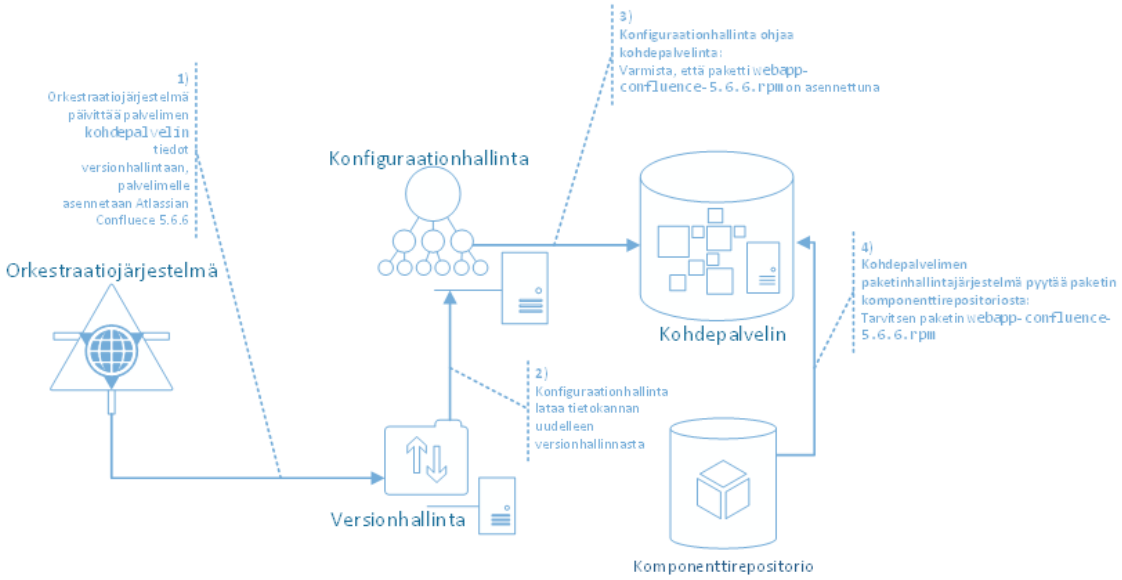
Prototyyppi- ja pilottivaiheissa ei hyödynnetty konfiguraatiojärjestelmää ympäristön parametrointiin, vaan tarvittavat asetukset määritettiin rakennettavaan RPM-pakettiin käyttämällä malleihin (engl. templates) perustuvaa menetelmää, jossa mallin pohjalta luotiin intanssikohtaisia RPM-paketteja. Menetelmän skaalautumattomuus tuli kuitenkin varsin ilmeiseksi pilottivaiheen aikana, ja siitä päätettiin luopua. Sen sijaan päätettiin ottaa käyttöön konfiguraationhallintajärjestelmä, jolla parametrointi voidaan tehdä myöhemmässä vaiheessa käyttöönottoa. Tämä havainnollistaa hyvin Bassin ym. (2013, 124-125) kuvaamaa sidonnan viivästämisen aiheuttamaa lisäkustannusta. Myöhemmällä arvojen sidonnalla saavutetaan tehokkaampi ja skaalautuvampi ympäristö, mutta tämä edellyttää lisää infrastruktuuriresursseja ja työtä konfiguraationhallintajärjestelmän käyttöönoton myötä. Myös Humble ja Farley (2011, 41-42) kannustavat viivästämään konfiguraatioparametrien sidonnan kääntämistä tai paketointia myöhäisempiin vaiheisiin käyttöönottoprosessissa.

Konfiguraationhallintajärjestelmä on se mekanismi, joka mahdollistaa skaalautuville ympäristöille riittävän konfiguraatioparametrien riittävän myöhäisen sidonnan. Sillä suoritetaan luvussa 6.1 kuvatuista käyttöönottovaiheista viimeinen, sovelluksen konfiguraatioparametrien määrittäminen.

Konfiguraationhallintajärjestelmä on tyypillisesti myös asiakas-palvelin -arkkitehtuurin mukainen järjestelmä, joka koostuu kontrollipalvelimesta ja sen asiakkaista. Suositut konfiguraationhallintajärjestelmät, kuten Puppet ja Chef käyttävät deklaratiivista, konfi-

guraatioinformaation määrittelyyn tarkoitettua kieltä. Kontrollipalvelimella määritetään em. kieltä käyttäen haluttu kohdepalvelinten konfiguraatio, kuten mitä paketteja ja niiden versioita palvelimelle halutaan asentaa. Asiakaspalvelimet lataavat konfiguraatiomääritykset kohdepalvelimelta ja suorittavat määrityksen mukaiset konfiguraatiotoimenpiteet, päätyen näin haluttuun lopputilaan (Humble & Farley 2011, 292-293; Alcazar Calero 2013, 2-3, 5).

Ambientian toteutuksessa käytetty konfiguraationhallintajärjestelmä Puppet toimii orkestraation alaisuudessa siten, että uutta järjestelmää käyttöönotettaessa orkestrointi syöttää versionhallinnassa olevaan konfiguraationhallintajärjestelmän avain-arvo -varastoon Hieraan (eräänlainen yksinkertainen tietokanta) tietoja mm. asennettavasta sovellusversiosta. Konfiguraationhallintajärjestelmän kontrollipalvelin käy ajastetusti lataamassa Hiera-tietokantansa sisällön versiohallinnasta. Tämän jälkeen konfiguraationhallintajärjestelmä ohjaa tietokantansa sisällön perusteella kohdepalvelimen paketinhallintajärjestelmää asentamaan halutun sovelluksen palvelimelle. Kuva 4 havainnollistaa järjestelmien välistä suhdetta.



KUVA 4. Orkestraation ohjaama konfiguraationhallintajärjestelmä ohjaa palvelimen paketinhallintaohjelmistoa.

6.6 Avainkomponentteja tukevat komponentit

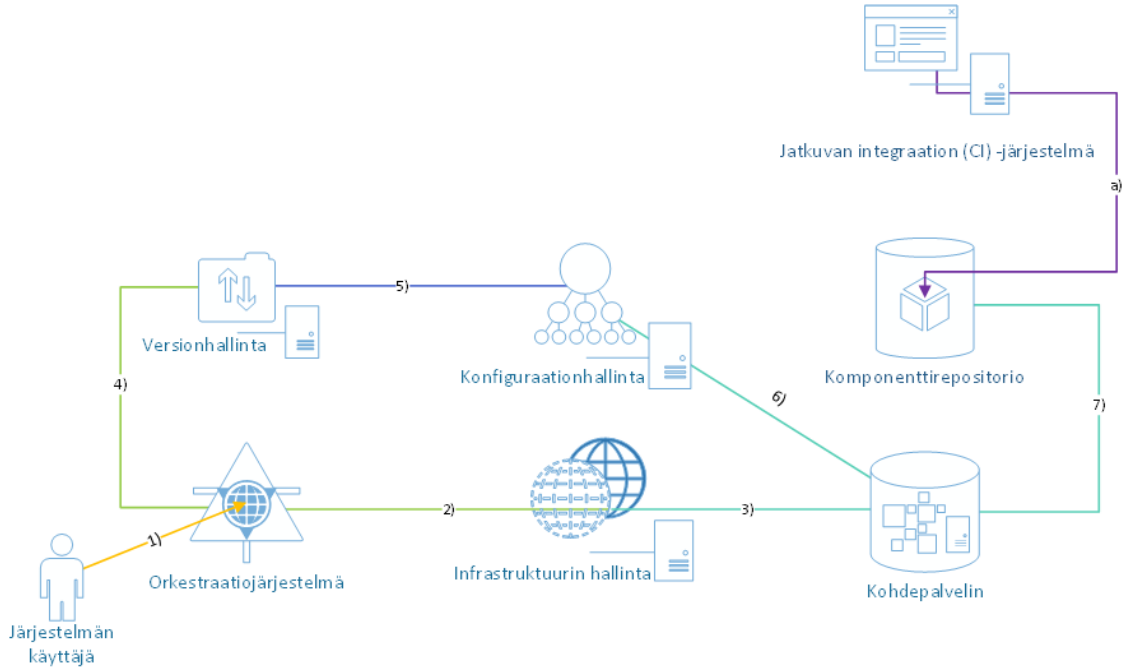
Versionhallintajärjestelmä on järjestelmä, jonka avulla voidaan helposti säilyttää useita versioita samasta tiedostosta, tai ryhmästä tiedostoja. Versionhallintajärjestelmä mahdollistaa helpon palaamisen edellisiin versioihin tiedostosta, sekä pitää myös kirjaa tiedostoihin tehdyistä muutoksista, jolloin sen avulla on helppo vastata kysymyksiin kuka muutti, mitä ja miksi (Humble & Farley 2011, 32-33). Humble ja Farley (2011, 33) suosittavat, että kaikki ympäristöön oleellisesti liittyvät artifaktit (tietokannan luontiskriptit, kääntämiseen tarvittavat skriptit, testit, dokumentaatio jne.) tulisi säilyttää versionhallinnan piirissä. Tätä suositusta on pyritty noudattamaan Ambientian toteutuksessa siinä laajudessa kuin se on käytännöllistä, mm. kaikki konfiguraationhallintajärjestelmän ja pakettirepositorion konfiguraatiot ovat säilöttynä versionhallinnassa. Koska prototyyppi- ja pilottivaiheissa käytetyt sovellukset ovat kolmannen osapuolen toimittamia, eikä niiden käyttöönottoprosessiin kuulu sovellusten kääntämistä lähdekoodista, ei sovelluksiin liittyviä artifakteja ole varastoitu versionhallinnassa.

Jotta paketinhallintajärjestelmä voisi asentaa sovelluksia komponenttirepositoriosta, on komponenttirepositorioon jollakin mekanismilla vietävä paketteja asennettavaksi. Sinänsä tämä olisi mahdollista tehdä myös käsin, mutta noudatellen Humblen ja Farleyn (2011, 25) ohjenuoraa *Automate Almost Everything* myös tästä prosessista on haluttu poistaa manuaalisia vaiheita. RPM-pakettien automaattiseen rakentamiseen ja julkaisemiseen hyödynnetään Ambientian toteutuksessa Atlassianin jatkuvan integraation (Continuous Integration, CI) ja julkaisun (Continuous Delivery, CD) työkalua, Atlassian Bamboota. Käyttämällä Bamboota rakennetaan julkaistaan tarpeen mukaan uusia versioita sovelluksista komponenttirepositorioon. Tämä prosessi on kuitenkin pääosin irrallinen tämän työn kohteena olevasta käyttöönottoprosesissta eikä näiden prosessien välillä ole jatkuvaa riippuvuutta.

6.7 Kokonaiskuvaus toteutuksesta

Edellisissä luvuissa on kuvattu toteutuksen kannalta olennaisia komponentteja osakokonaisuuksina. Tässä luvussa kuvataan järjestelmän toimintaa yksittäisen sovellusintanssin käyttöönotossa alusta loppuun. Kuva 5 havainnollistaa järjestelmän komponentteja ja niiden välisiä suhteita. Toiminnan vaiheet ovat numeroituna kuvaan suoritusjärjestyk-

sessä. Kohta *a* (CI-järjestelmän ja komponenttirepositorion välillä) tapahtuu muusta toiminnasta asynkronisesti.



KUVA 5. Järjestelmäarkkitehtuuri kokonaisuutena.

Järjestelmän toiminta käynnistyy järjestelmän käyttäjän käynnistäessä uuden sovelluksen provisiointiprosessin käynnistämällä orkestrointijärjestelmän haluamillaan parametreilla (vaihe 1). Parametreissa välitetään tieto esimerkiksi halutusta sovelluksesta ja sen versiosta. Seuraavassa vaiheessa (vaihe 2) orkestraatiojärjestelmä ottaa yhteyttä infrastruktuurin hallintajärjestelmään ja ohjeistaa sitä luomaan uuden virtuaalipalvelimen (vaihe 3). Kun infrastruktuurin hallintajärjestelmä on luonut uuden virtuaalipalvelimen ja kuitannut luomisen onnistuneeksi orkestraatiojärjestelmälle, lähettää orkestraatiojärjestelmä seuraavaksi sovellusympäristön luomiseksi tarvittavat parametrit versionhallintajärjestelmään (kohta 4). Konfiguraationhallintajärjestelmä käy ajastetusti lataamassa tietonsa versionhallintajärjestelmästä (vaihe 5) ja saatuaan uuden sovelluksen parametrit luo tarvittavat konfiguraatiotiedostot kohdepalvelimelle (vaihe 6) sekä ohjeistaa kohdepalvelimen paketinhallintajärjestelmää lataamaan tarvittavat sovellusbinäärit komponenttirepositoriosta (vaihe 7). Lopuksi konfiguraationhallintajärjestelmä käynnistää sovelluksen, ja se on valmis käytettäväksi.

Uusien sovellusversioiden vieminen komponenttirepositorioon tapahtuu varsinaisesta käyttöönottoprosessista asynkronisesti. Jos sovelluksesta halutaan ottaa käyttöön versio,

jota ei vielä löydy komponenttirepositoriosta, käytetään jatkuvan integraation järjestelmää luomaan tarvittava sovelluspaketti ja viemään se paikalleen komponenttirepositorioon (vaihe a). Uusien sovellusversioiden julkaisutiheys riippuu huomattavasti sovelluksesta. Joidenkin sovellusten osalta uusia versioita tulee päivittäin, toisten osalta kuukausittain tai harvemmin.

Taulukossa 3 esitetään perusteluineen Ambientian toteutuksessa eri rooleihin valitut järjestelmät ja teknologiat.

TAULUKKO 3. Toteutetussa arkkitehtuurissa käytetyt järjestelmät ja teknologiat rooleittain

Rooli	Käytettävä järjestelmä	Perustelut valinnalle
Orkestraatio	Toteutettu itse Python-ohjelmointikielellä	Muuhun arkkitehtuuriin sopivaa, valmista ratkaisua ei ollut saatavilla.
Infrastruktuurin hallinta	VMware vCenter Orchestrator (nyk. vRealize Orchestrator)	Ambientialla on käytössä VMware-pohjainen infrastruktuuri, johon tämän orkestrointityökalun integrointi on helppoa.
Paketinhallinta	RPM Package Manager	Ambientia käyttää ympäristöissään pääasiallisesti Red Hat Enterprise Linux -käyttöjärjestelmää, joka taas puolestaan käyttää käyttöjärjestelmän pakettien hallintaan RPM:ää. Osa sovelluspinosta asennetaan joka tapauksessa Red Hatin paketeista, joten riippuvuuksien hallinta on paljon helpompi toteuttaa jos kaikki binäärit jaellaan samalla paketoinnilla.
Konfiguraation-hallinta	Puppet	Red Hat Enterprise Linux -ympäristöjen keskitettyyn hallintaan käytettävä Red Hat Satellite 6 sisältää Puppetin ydinkomponenttinaan (Red Hat, 2014). Tämä mahdollistaa tiiviimmän integraation käyttöjärjestelmän ja sovelluksen hallinnan välillä, ja poistaa tarpeen käyttää kahta erillistä hallintajärjestelmää (alustalle ja sovellukselle erikseen).
Versionhallinta	Atlassian Stash	Ambientialla kaikki versionhallinta on keskitetty git-pohjaiseen Atlassian Stashiin.
Jatkuva integraatio ja julkaisu	Atlassian Bamboo	Ambientialla kaikki jatkuvan integraation ja julkaisun prosessit on keskitetty Atlassian Bambooseen.

7 ARKKITEHTUURIN VALIDOINTI

7.1 Validointi SaaS-kypsyysmalleja vasten

Toteutettaessa SaaS-arkkitehtuuria voidaan sitä vertailla SaaS-kypsyysmallien esittämiä kypsyytasoja vasten. Luvussa 4.2 on esitelty tässä työssä sovellettu Carraron ja Chongin (2006) kypsyystasomalli. Vertailemalla toteutettua arkkitehtuuria kypsyystasomallin tasoja vasten saadaan siis osviittaa siitä, onko toteutettu arkkitehtuuri SaaS-arkkitehtuuri vai ei. Kypsyystasomallin tasoista voidaan myös nähdä mitkä ovat ne kehittämiskohteet joihin keskittymällä kypsyyttä voidaan parantaa.

Toistaiseksi arkkitehtuurin kanssa käytettävien sovellusten lisenssiehdot ja tekniset rajoitukset rajoittavat näiden sovellusten osalta arkkitehtuurin kypsyystasolle 2. Jokaisella asiakkaalla on siis oma kopionsa samasta sovellusbinääristä, eikä yksittäisessä instanssissa ole koskaan kuin yhden asiakkaan dataa. Arkkitehtuuri ei sinänsä aseta esteitä korkeammille kypsyystasoille, tämä edellyttäisi ainoastaan teknistä ja sopimusteknistä yhteensopivuutta käytetyiltä sovelluksilta. Esimerkiksi kokonaan itse toteutetuilla sovelluksilla voitaisiin haluttaessa saavuttaa kypsyysmallin korkein neljäs taso.

SaaS-kypsyysmallit tarjoavat varsin yksipuolisen tavan arvioida arkkitehtuuria. Arviointikriteerit (kypsyystasot) ovat ennalta määrättyjä, eivätkä varsinaiset liiketoiminnalliset tavoitteet vaikuta arviointiin. Täysin tavoitteet täyttävä arkkitehtuuri voi olla SaaS-kypsyysmallin näkökulmasta epäkypsä, vaikka se olisi liiketoiminnallisesta näkökulmasta hyvä. Luvussa 4.3 todettiin, ettei kaikissa tilanteissa ole edes Software as a Service -liiketoiminnassa järkevää tavoitella korkeimpia mahdollisia kypsyystasoja. Järkevä tavoitetaso riippuu muusta liiketoimintamallista ja tavoitellusta asiakaskunnasta. Tällä hetkellä arkkitehtuurin kanssa käytettävien sovellusten osalta ei voida tavoitella kypsyystasoa 2 korkeampia tasoja sovellusten rajoitteiden takia. Toisaalta on strategisesti päätetty keskittyä käyttämään juuri näitä sovelluksia ratkaisujen teknologisena pohjana, jolloin voidaan todeta kypsyystason 2 olevan pääsääntöisesti riittävä ja liiketoiminnalliset tavoitteet täyttävä.

Varsinaiset arkkitehtuurin arviointimenetelmät tarjoavat välineitä pureutua arkkitehtuurin ja liiketoiminnan tavoitteiden vertailuun. Seuraavassa esitellään kaksi arkkitehtuurin

arviointimenetelmää, vakiintuneempi ATAM ja uudempi, iteratiivisten projektien kanssa käytettäväksi soveltuva DCAR.

7.2 Architecture Tradeoff Analysis Method (ATAM)

Architecture Tradeoff Analysis Method on Carnegie Mellon -yliopiston ohjelmistotekniikkaan keskittyneen tutkimusyksikön Software Engineering Instituten kehittämä arkkitehtuurin arviointimenetelmä. ATAM pohjaa aiempaan Software Engineering Instituten 1990-luvulla tehtyyn arkkitehtuurin arviointimenetelmien kehitystyöhön, kuten Software Architecture Analysis Methodin (Barbacci ym. 1998, 2). Menetelmää on käytetty arkkitehtuurin arviointiin yli vuosikymmenen ajan eri toimialoilla rahoitusalasta puolustusteollisuuteen (Bass ym. 2013, 400).

ATAM on skenaariopohjainen menetelmä, jossa järjestelmän ilmentämiä laatuominaisuuksia tarkastellaan konkreettisten skenaarioiden kautta (Clements ym. 2000, 2; Avgeriou ym. 2014, 159). Tällainen skenaario voi olla esimerkiksi kahdennetussa järjestelmässä pääasiallisen palvelimen vikaantuminen, jolloin arvioidaan toimintaan palautumisaikaa, eli järjestelmän laatuominaisuutena saavutettavuutta.

Menetelmän huonona puolena on sen työläys. Menetelmän kehittäjät arvioivat yksittäisen arvioinnin kestoksi 3-4 kokonaista työpäivää (Clements ym. 2000, 43; Bass ym. 2013, 404). Arviointiin osallistuu kerrallaan 3-6 henkilöä (Clements ym. 2000, 39), jolloin arvioinnin keskimääräiseksi kokonaistyömääräksi muodostuu 9-24 työpäivää. Tämä suhtautuu huonosti iteratiivisten projektinhallinnan viitekehitysten, kuten Scrumin käyttämään tapaan aikatauluttaa ja suunnitella työtä. Scrumissa työ jaotellaan tyypillisesti korkeintaan noin kuukauden mittaisiin sprintteihin (Schwaber & Sutherland 2013, 7). Täten sprint koostuu korkeintaan noin 20 työpäivästä. Suositellun Scrum-kehitystiimin koon ollessa 3-9 kehittäjää (Schwaber & Sutherland 2013, 6), tarkoittaa tämä sitä, että suurin osa yksittäisessä sprintissä tehtävissä olevasta työstä kuluisi arkkitehtuurin arviointiin. Tämä sopii huonosti iteratiiviseen prosessiin, jossa arkkitehtuuria tulisi arvioida toistuvasti, koska arkkitehtuuri potentiaalisesti muuttuu jossain määrin jokaisen sprintin mukana. Tyypillisesti arvioinnit ovatkin kertaluontoisia suoritteita, eivätkä toistuvia prosesseja (Avgeriou ym. 2014, 159).

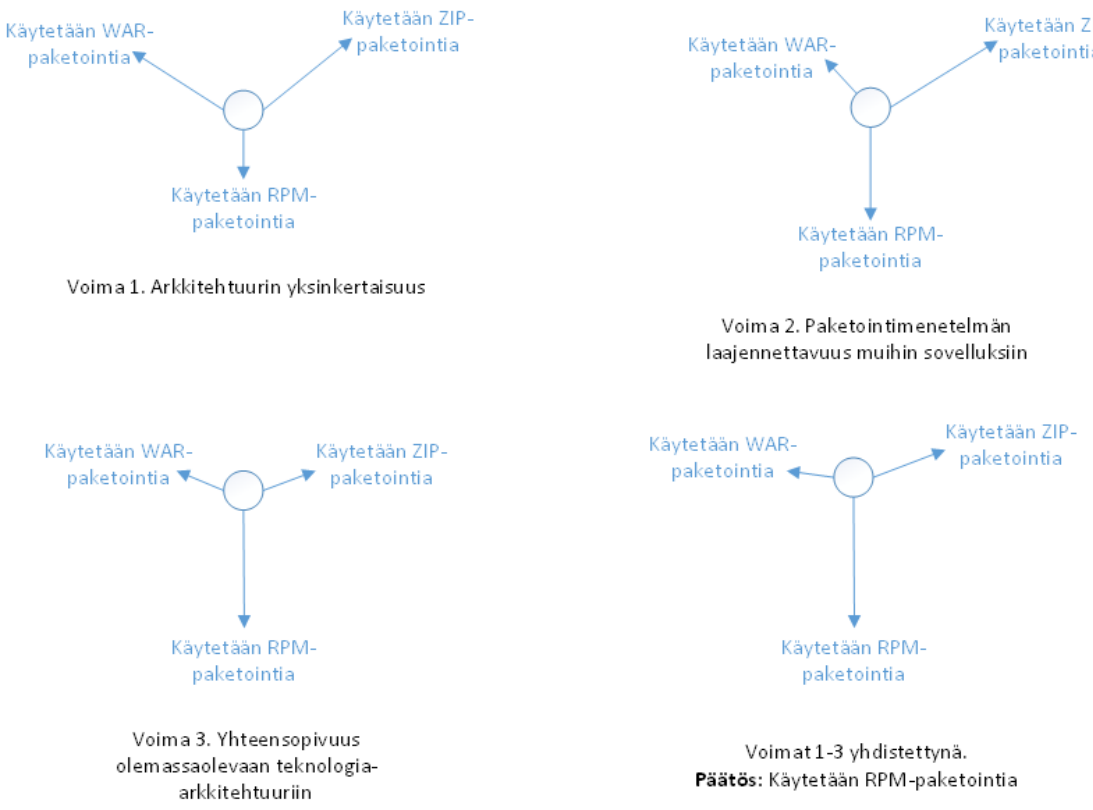
ATAM:n etuna on menetelmän kattavuus. Skenaariopohjainen arkkitehtuurin käsittely antaa hyvin yksiselitteisiä vastauksia siihen, toteuttaako arkkitehtuuri halutut ei-toiminnalliset vaatimukset vai ei (Clements ym. 2000, 13). Menetelmä ei keskity arkkitehtuuristen *päätösten* perustelujen ruotimiseen, vaan arkkitehtuurin kykyyn vastata sille asetettuihin vaatimuksiin (Avgeriou 2014, 160). Menetelmä onkin parhaimillaan projekteissa, joissa jostain syystä arkkitehtuurin iteratiivinen suunnittelu ei ole yksinkertaista, kuten sulautetuissa järjestelmissä. Myös projektit, joissa jostain syystä nähdään suuri arkkitehtuullinen riski (esimerkiksi järjestelmän koon tai tuntemattoman toimialan takia) ovat hyviä kandidaatteja ATAM-arvioinnille.

ATAM-arviointiprosessia ei kuvata yksityiskohtaisesti tässä työssä, sillä kyseistä arviointimenetelmää ei päätetty käyttää järjestelmäarkkitehtuurin arviointiin. Luvussa 7.4 otetaan tarkemmin kantaa arkkitehtuurillisen menetelmän valintaan.

7.3 Decision Centric Architecture Review (DCAR)

Decision Centric Architecture Review on Tampereen teknillisen yliopiston ja Groningenin yliopiston yhteistyössä kehittämä, erityisesti iteratiivisia projektikehyksiä silmälläpitäen suunniteltu arkkitehtuurin arviointimenetelmä (Avgeriou ym. 2014, 158). Skenaariopohjaisen arkkitehtuurin testaamisen sijaan menetelmä keskittyy arkkitehtuurin taustalla olevien arkkitehtuuristen päätösten ja niihin vaikuttavien päätösten arviointiin (Avgeriou ym. 2014, 160).

DCAR:ssä arkkitehtuurisia päätöksiä ja niiden mielekkyyttä arvioidaan niihin kohdistuvien päätösvoimien (decision forces) kautta. Voima voi olla mikä tahansa arkkitehtuuriin vaikuttava tekijä. Olennaista voimassa on, että se ”vetää” arkkitehtuurista päätöstä tiettyyn suuntaan. (Avgeriou ym. 2014, 160.) Käsitteen ymmärtämistä auttaa, jos sen mieltää vektorisuureeksi, jolla on sekä voima että suunta. Tällöin ”summavektori” osoittaa sen arkkitehtuurisen päätöksen suuntaan, joka on siihen vaikuttavien voimien valossa paras valinta. (Avgeriou ym. 2014, 160.) Kuva 6 esittää yksinkertaistettuna erästä tässä työssä esitellyssä järjestelmäarkkitehtuurissa tehtyä päätöstä ja siihen vaikuttaneita voimia. Käytännössä päätökseen vaikuttavia voimia on huomattavasti enemmän.



KUVA 6. Järjestelmäarkkitehtuurinen päätös ja siihen vaikuttaneet voimat (Avgeriou ym. 2014, 164, muokattu)

Itse DCAR-arviointiprosessi on kymmenenvaiheinen prosessi, joista kahdeksan kuuluvat varsinaiseen arviointiprosessiin. Kaksi arviointiprosessin ulkopuolista vaihetta ovat arvioinnin valmistelu ja arvioinnin tuloksista raportointi. Keskimääräiseen DCAR-arviointiin osallistuu noin 8 henkilöä. Minimissään arviointiin tarvitaan noin neljä henkilöä. Menetelmän kehittäjien mukaan DCAR-arviointiprosessiin kuluu tyypillisesti aikaa noin 5 tuntia. (Avgeriou ym. 2014, 165-173.)

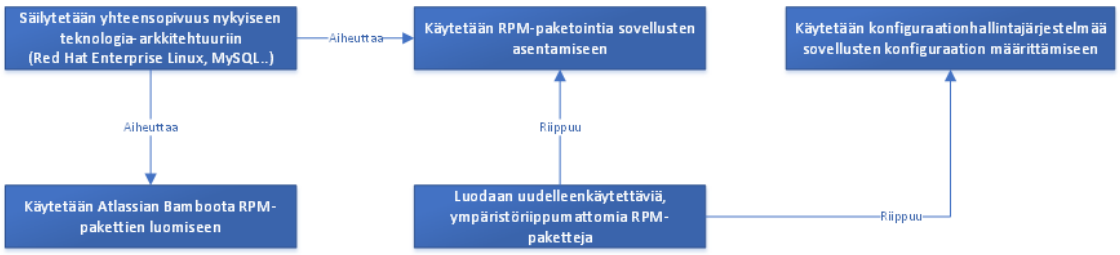
Arviointi aloitetaan valmistelevalla vaiheella, jossa kerätään arviointiin osallistuvat henkilöt. Arviointiin osallistuvat henkilöt jaetaan kahteen ryhmään, arkkitehtuurin sidosryhmään ja arviointitiimiin. Sidosryhmään kuuluvat järjestelmän suunnitellut arkkitehti tai arkkitehdit, sekä järjestelmää liiketoiminnallisesta näkökulmasta edustavia henkilöitä. Arvointiryhmä taas koostetaan henkilöistä, joiden katsotaan omaavan riittävät tiedot ja taidot arkkitehtuurin arvioimiseksi. (Avgeriou ym. 2014, 165-166.) Järjestelmän suunnitellut arkkitehti laatii ennen varsinaista arviointitilaisuutta esityksen, jossa kuvataan järjestelmän arkkitehtuuri. Liiketoiminnan edustaja puolestaan laatii ennen tilaisuutta esityksen, jossa kuvataan liiketoiminnallinen motivaatio järjestelmän toteut-

tamiselle. (Avgeriou ym. 2014, 166-167.) Esitykset toimitetaan arviointitiimille ennen tilaisuutta, jotta niihin on mahdollista tutustua etukäteen.

Varsinaisen arviointitilaisuuden ensimmäisessä vaiheessa esitellään itse arviointimenetelmä lyhyesti. Tämän jälkeen liiketoiminnan edustajat pitävät alle puolen tunnin mittaisen esityksen järjestelmästä liiketoiminnallisesta näkökulmasta. Tarkoituksena on kuvata järjetelmään liittyvä toimintaympäristö, liiketoiminnalliset tavoitteet, tunnetut tekniset rajoitteet ja niin edelleen. Arviointitiimi kerää jo tämän esityksen aikana listaa heidän mielestään arkkitehtuuriin vaikuttavista voimista. (Avgeriou ym. 2014, 167.)

Seuraavassa vaiheessa järjestelmän suunnitellut arkkitehti pitää noin tunnin mittaisen esityksen toteutetusta arkkitehtuurista. Esityksessä käsitellään keskeisiä arkkitehtuurisia ratkaisuja, kuten käytettyjä ohjelmistoja, ohjelmointikehyksiä ja kirjastoja sekä näiden valintaan vaikuttaneita tekijöitä arkkitehdin näkökulmasta. Arviointitiimi kirjaa myös tämän esityksen aikana löytämänsä arkkitehtuuriin vaikuttavat voimat, sekä listan keskeisistä arkkitehtuurisista päätöksistä. (Avgeriou ym. 2014, 167-168.)

Esitysten jälkeen arviointitiimi esittelee kaikille osallistujille tulkintansa arkkitehtuurin keskeisistä päätöksistä ja niihin vaikuttavista voimista. Nämä tarkistetaan sisällöllisesti arkkitehdin ja liiketoiminnan edustajien kanssa, jotta kaikki arvointiin osallistuvat ovat samaa mieltä niiden sisällöstä. Päätöksistä piirretään myös päätössuhdekaavio (decision relationship view), josta selviävät päätösten suhteet toisiinsa. Päätökset ovat harvoin itsenäisiä. Usein ne riippuvat toisista päätöksistä tai aiheuttavat niitä. Tällöin kaaviosta ilmenevät helposti arkkitehtuurin kannalta keskeisimmät päätökset niinä päätöksinä, joista on eniten yhteyksiä toisiin päätöksiin. (Avgeriou ym. 2014, 168). Kuvassa 7 on yksinkertaistettu esimerkki päätössuhdekaaviosta.



KUVA 7. Yksinkertaistettu päätössuhdekaavio

Kun päätökset on saatu dokumentoitua, ne priorisoidaan pisteytettämällä. DCAR ei sinänsä määritä erityistä menetelmää pisteytyksen toteuttamiseen. Avgeriou ym. (2014, 169) kuitenkin ehdottavat käytettäväksi kaksivaiheista menetelmää. Ensin jokainen sidosryhmien (liiketoiminnan edustajat ja arkkitehdit) edustaja merkitsee kolmanneksen kaikista kuvatuista päätöksistä seuraavaa kierrosta varten. Tämän jälkeen ensimmäisellä kierroksella eniten ääniä saaneet päätökset pisteytetään siten, että jokainen sidosryhmien edustaja jakaa 100 pistettä näiden päätösten kesken sen mukaan, mitkä hän korkee tärkeimmiksi. (Avgeriou ym. 2014, 168-169.) Huomioinarvoista tässä vaiheessa on, että priorisoinnin tekevät sidosryhmien edustajat, eivät arvioijat.

Pisteytyksen jälkeen arkkitehti tai arkkitehdit dokumentoivat pisteitä saaneet päätökset DCAR:n määrittelemillä lomakkeilla, tai jollakin muulla sopivalla tavalla. Jokaisesta päätöksestä dokumentoidaan itse päätös, ongelma joka päätöksellä on haluttu ratkaista, vaihtoehtoiset ratkaisut sekä päätöstä tukevat ja vastustavat voimat. (Avgeriou ym. 2014, 169-170).

Kun kaikki pisteitä saaneet päätökset on dokumentoitu, käydään ne lävitse seuraavassa vaiheessa, aloittaen eniten pisteitä saaneesta päätöksestä. Päätöksen dokumentoinut henkilö esittelee päätöksen muille. Esittelyn on tarkoitus olla interaktiivinen ja päätöksiä on tarkoitus arvioida kriittisesti. Tässä vaiheessa arvioidaan tukeeko päätöstä riittävä määrä voimia vai ei. Lopuksi sidosryhmien edustajat äänestävät jokaisen päätöksen kohdalla sen tilasta: onko päätös hyvä, tyydyttävä vai tarviiko sitä harkita uudelleen. Kun kaikki päätökset on käyty lävitse, arvioidaan vielä lyhyesti itse arvioinnin toteutus ja mahdolliset poikkeamat menetelmästä. Tämän jälkeen aarviointitilaisuus päättyy. (Avgeriou ym. 2014, 170.) Taulukko 4 esittää DCAR:n vaiheet ja niiden päätteeksi syntyvät tuotokset.

TAULUKKO 4. DCAR:n vaiheet ja niissä aikaansaatavat tuotokset (Avgeriou ym. 2014, 166, muokattu)

Vaihe	Tuotokset
1. Valmistelu	Materiaali (esitykset) arviointitilaisuutta varten
2. Arviointimenetelmän esittely	-
3. Liiketoiminnallisten tavoitteiden esittely	Arkkitehtuuriset päätösvoimat
4. Arkkitehtuurin esittely	Arkkitehtuuriset päätökset, lisää päätösvoimia
5. Päätösten viimeistely	Viimeistelty lista arkkitehtuurisista päätöksistä ja päätössuhdekaavio
6. Päätösten priorisointi	Priorisoitu lista tärkeimmistä päätöksistä
7. Päätösten dokumentointi	Määrämuotoinen dokumentointi tärkeimmistä päätöksistä ja niihin vaikuttavista voimista
8. Päätösten analysointi	Päätösten hyväksyminen tai hylkääminen, päätöskohtainen lista siihen vaikuttavista voimista, viimeistelty päätösdokumentaatio
9. Arviointitilaisuuden palaute	Kehityskohteet seuraavia arviointitilaisuuksia silmälläpitäen
10. Raportointi	Lopullinen arviointiraportti

Arviointitilaisuuden päätyttyä arviointitiimi koostaa tilaisuudessa kerätystä materiaalista raportin, jossa kuvataan arviointitilaisuuden löydökset. Olennaisin osa raportista ovat dokumentoidut arkkitehtuuriset päätökset, erityisesti ne joita tarvitsee harkita uudelleen. (Avgeriou ym. 2014, 171.).

Arkkitehtuurin arvioinnilla on muitakin hyötyjä pelkän arkkitehtuurin validoinnin ohella. Se takaa, että arkkitehtuuri on tärkeimmiltä osiltaan dokumentoitu tietyn mallin mukaisesti (Avgeriou ym. 2014, 159). Arviointitilaisuus tarjoaa myös mahdollisuuden jakaa tehokkaasti ja interaktiivisesti tietoisuutta ja ymmärrystä järjestelmän arkkitehtuurista (Avgeriou ym. 2014, 167-168).

7.4 Arkkitehtuurin validoinnin käytännön toteutus

Menetelmien vertailun jälkeen päädyttiin toteuttamaan järjestelmäarkkitehtuurin arviointi käyttämällä DCAR-menetelmää. ATAM koettiin menetelmänä liian työlääksi, sillä useita päiviä ja useita henkilöitä sitovan arvioinnin toteuttamismahdollisuudet eivät olleet realistisia. Myös DCAR:n perustoteutuksesta jouduttiin poikkeamaan, sillä arviointiin halutut osallistujat työskentelevät eri paikkakunnilla ja ovat sidottuja eri projekteihin. Tästä aiheutui vaikeuksia löytää sopivaa ajankohtaa saada kaikki tarvittavat henkilöt samaan paikkaan samaan aikaan. Arviointiin osallistuivat sidosryhmien edustajina minä järjestelmän arkkitehdin roolissa, sekä liiketoiminnan edustajana Ambientian teknologia- ja palvelujohtaja. Arviointitiimin muodostivat arkkitehtuurin toteuttamiseen osallistunut järjestelmäasiantuntija ja eräs käyttöönottoautomaatiosta kiinnostunut sovelluskehittäjä.

Validoinnin toteuttamisesta ei kuitenkaan haluttu luopua. Varsinaisen arviointitilaisuuden sijaan arviointi päätettiin toteuttaa Ambientian intranetinä ja yhteistyöalustana käytettävässä Confluencessa. Vaiheita mukautettiin siten, että DCAR:n vaiheet 1-4 tehtiin ennakkoon dokumentoimalla liiketoiminnalliset tavoitteet järjestelmän arkkitehtuuri Confluenceen.

Johtuen arvioinnin asynkronisesta luonteesta, vaiheita 5-8 jouduttiin mukauttamaan enemmän. DCAR:n mukaisesti arkkitehtuuriin vaikuttavat voimat ja arkkitehtuuriset päätökset tulisi dokumentoida arvioijien toimesta. Jotta arviointi ei veisi kohtuuttomasti aikaa arvioijilta, arkkitehtuuriin vaikuttavat voimat ja päätökset dokumentoitiin etukäteen toimestani. Arvioijia kuitenkin ohjeistettiin täydentämään dokumentaatiota, jos he toteavat siitä puuttuvan voimia tai päätöksiä.

Arvioijien keskeiseksi tehtäväksi jäi priorisoida dokumentoidut päätökset. Tämän jälkeen tärkeimmistä kolmesta päätöksestä käytiin vielä tarkempi keskustelu Confluencen välityksellä siten, että arvioijat kirjoittivat päätöksistä lyhyen, muutaman lauseen mittaisen arvion, johon arkkitehdin roolissa vastasin. Tämän jälkeen arvioijat kommentoivat vielä vastinettani. Myös tässä poikettiin DCAR:n perustoteutuksesta, jossa päätösten pisteyttäminen kuuluu sidosryhmien edustajien vastuulle. Käytännössä arviointi tehtiin nyt päinvastoin kuin menetelmä ehdottaa, eli arkkitehti dokumentoi päätökset ja arvioijat priorisoivat ne.

Korkeimmat pisteet saaneet päätökset koskivat konfiguraationhallintajärjestelmän käyttöä sekä arkkitehtuurin taaksepäin yhteensopivuutta nykyisten ratkaisujen kanssa. Nämä ovat toisaalta ne päätökset, joiden saattoi odottaakin saavan paljon pisteitä. Konfiguraationhallintajärjestelmää ei ole aiemmin ollut Ambientialla käytössä. Päätös järjestelmän käyttöönotosta on iso muutos, jolla on potentiaalisesti paljon vaikutusta. Kommenteissaan arvioijat näkivät toisaalta konfiguraationhallintajärjestelmän käyttöönoton hyvänä päätöksenä. Arkkitehtuurin taaksepäin yhteensopivuuden säilyttämisen taas nähtiin vaikeuttavan kehittämistyötä. Tietysti puhtaalta pöydältä aloittamalla voitaisiin saavuttaa optimaalisin lopputulos teknisessä mielessä. Tämä taas edellyttäisi organisaatiolta kykyä ja halua omaksua potentiaalisesti paljonkin uutta teknologiaa ja totutusta poikkeavia toimintatapoja. Päätös puoltaa paikkaansa sikäli, että työn tarkoituksena oli nimenomaan kehittää nykyisiä toimintamalleja automaattisemmiksi. Näin ratkaisu on merkittävästi helpompi viedä käyttöön. Mahdolliset teknologiamuutokset voidaan tehdä iteratiivisesti.

Arviointi itsessään onnistui varsin hyvin. Ainoastaan arvioijien pieni lukumäärä (kaksi) aiheutti hieman ongelmia pisteytettäessä päätöksiä. Esimerkiksi erään päätöksen kohdalla toinen arvioijista antoi kolmanneksen pisteistään päätökselle, ja toinen ei antanut lainkaan pisteitä päätökselle. Tällöin päätöksen tärkeyden tulkitseminen vaikeutuu. Jos arvioijia olisi ollut vaikkapa neljä, kuvatun kaltaisia pattitilanteita olisi luultavasti syntynyt vähemmän. Suurimmasta osasta päätöksiä arvioijat olivat kuitenkin samaa mieltä. Arvioinnin perusteella ei päätetty tehdä muutoksia nykyiseen arkkitehtuuriin, vaan tärkeimmiksi koetut päätökset koettiin myös arvioijien toimesta oikean suuntaisiksi.

8 SOFTWARE AS A SERVICE -ARKKITEHTUURIN LIIKETOIMINNALLISET VAIKUTUKSET

8.1 Perinteisen toimintamallin kuvaus

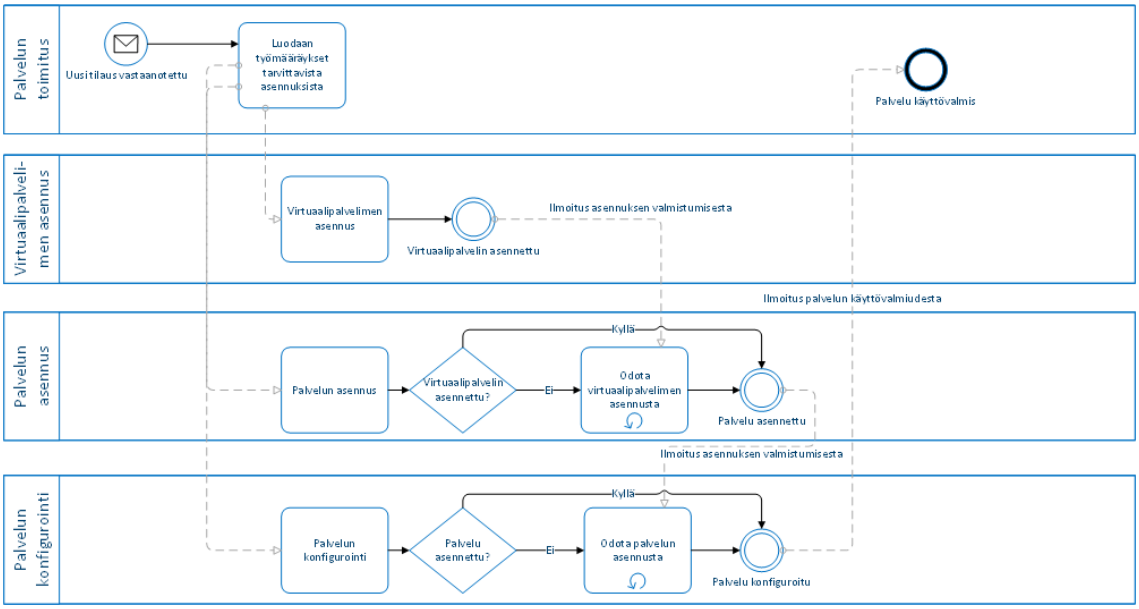
Perinteisessä Ambientialla käytetyssä toimintamallissa palvelun toimitus on kaksivaiheinen prosessi. Ensin asennetaan toimeksiannon mukaisesti virtuaalipalvelin, ja tästä täysin erillisenä prosessina myöhemmin itse sovellus palvelimelle. Molemmat asennukset tehdään saman tiimin asiantuntijoiden toimesta, mutta näitä koordinoidaan erillisinä tehtäväjonoina.

Sovellusten asennuksissa ei ole perinteisesti käytetty luvuissa 6.4 ja 6.5 kuvattuja paketoinnin ja konfiguraatiohallinnan menetelmiä, joissa sovelluksen käyttöönotto olisi tiiviimmin integroitu alustaan. Asennuksissa käytetään sovellusten valmistajan, Atlassianin, valmiiksi koostamia niin kutsuttuja ”bundleja”, jotka sisältävät samassa paketissa valmiiksi konfiguroidun Apache Tomcat -sovelluspalvelimen ja itse sovelluksen. Toimintatapa on varsin yleinen, myös Ambientian toinen iso kumppani, amerikkalainen Liferay, toimittaa portaalinsa vastaavalla tavalla paketoituna kokonaisuutena.

Virtuaalipalvelinten käyttöönotossa käytetään luvuissa 6.2 ja 6.3 kuvattuja järjestelmiä, mutta toistaiseksi orkestraatiota ei ole oletuksena ulotettu koskemaan koko sovelluspinoa, vaan sen avulla konfiguroidaan pelkästään infrastruktuuriin liittyvät osat alustasta. Virtuaalipalvelimen asentamisen jälkeen sovelluksen asentaminen tehdään kokonaan käsityönä. Prosessi on erittäin monivaiheinen ja siten virheille altis. Yksityiskohtaisesta ohjeistuksesta huolimatta asennuksissa ilmenee asentajasta ja tämän henkilökohtaisista preferensseistä johtuvia eroavaisuuksia asennuksesta toiseen.

Palvelimen ja sovelluksen asentamiseen kuluu aikaa nykyprosessilla parhaassa tapauksessa hieman yli puoli päivää. Prosessi on erityisen hidas tehtäessä sovellusten versiopäivityksiä. Sovellukset eivät tue paikallaan tehtäviä versiopäivityksiä, vaan versiopäivitys tarkoittaa käytännössä uuden sovellusversion asentamista vanhan rinnalle, ja datan migraatiota vanhasta palvelusta uuteen. Tällöin jokainen versiopäivitys, oli kyseessä kuinka pieni versiomuutos tahansa, tarkoittaa aina kokonaan uuden sovellusasennuksen tekemistä. Päivitysprosessiinkin kuluu tyypillisesti noin puoli päivää. Ympäristöt ovat myös toisistaan täysin irrallisia, joten yhdessä ympäristössä tiettyyn kohdeversioon teh-

ty versiopäivitys ei vähennä laisinkaan tarvittavaa työtä toisessa, täysin vastaavassa päivitystyössä. Versiopäivitysten läpimenoajan lyhentäminen on yksi keskeinen ongelma, johon tällä työllä haluttiin parannusta. Sovelluksen asentamisen jälkeen sovellusta tyypillisesti vielä konfiguroidaan asiakkaan tarpeiden mukaan, toteuttamalla esimerkiksi erilaisia integraatioita tai asentamalla sovellukseen liitännäisiä (plug-in). Tämä jatkokonfiguraatiotyö tapahtuu varsinaisesta käyttöönottoprosessista täysin erillisesti. Käyttöönoton ja konfiguroinnin välissä voi kulua paljonkin aikaa. Kuva 8 havainnollistaa perinteistä asennusprosessia.



KUVA 8. Perinteisen asennusprosessin vaiheet.

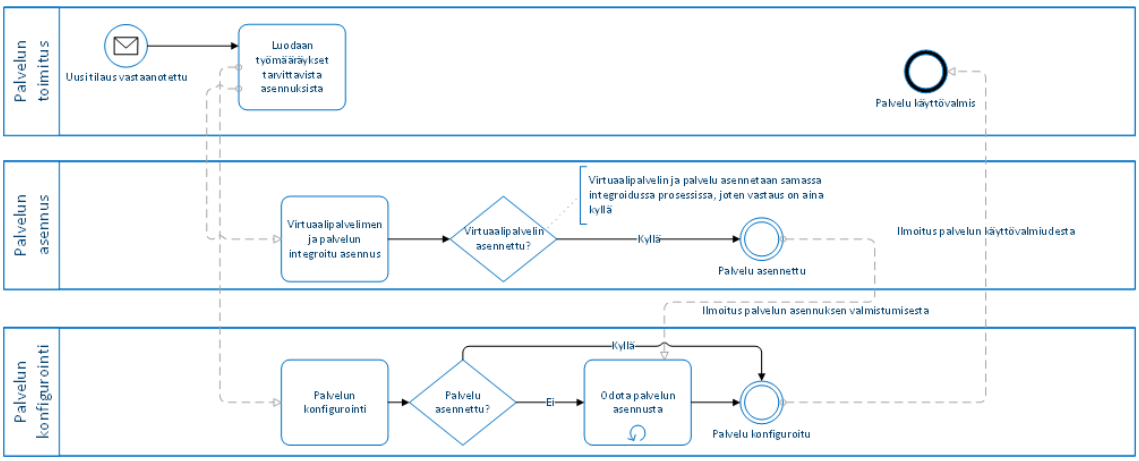
8.2 SaaS-toimintamallin kuvaus

Kuten luvussa 6.7 esitettiin, SaaS-toimintamallissa palvelimen ja sovelluksen käyttöönotto ovat osa samaa, integroitua prosessia. Tässä toimintamallissa ollaan myös vakioitu käytäntö, jossa yksittäiselle virtuaalikoneelle asennetaan aina vain yksi sovellus. Tämä tekee asennusprosessista hyvin yksiselitteisen. Uusi sovellusasennus tarkoittaa aina samanlaista prosessia, jossa ensin automaation avulla provisioidaan uusi virtuaalipalvelin ja sen jälkeen asennetaan siihen konfiguraationhallinta- ja paketinhallintajärjestelmien yhteistyöllä haluttu sovelluspino.

Sovelluksen ja siihen liittyvän virtuaalipalvelimen käyttöönottoon kuluu merkittävästi vähemmän aikaa, kuin perinteisessä käsin tehdyssä asennuksessa. Erityisesti SaaS-toimintamallin edut tulevat esiin tehtäessä sovellusten versiopäivityksiä. Toimintamal-

lissa ollaan hyödynnetty lukujen 6.4 ja 6.5 mukaisia paketin- ja konfiguraationhallintajärjestelmiä, joiden avulla versiopäivitys voidaan tehdä hyvin nopeasti, noin viidessätoista minuutissa. Tämä on saavutettu luopumalla valmiiksi koostettujen ”bundlejen” käytöstä sovellusten paketoinnissa ja paketoimalla sovellukset itse uudelleen paketinhallintajärjestelmän käyttämässä RPM-muodossa. Uudelleenpaketointiin kuluu uuden sovellusversion osalta aikaa noin 30 minuuttia per uusi versio, mutta tähän käytetty aika saadaan moninkertaisesti takaisin nopeutuneiden päivitysten johdosta. Kertaalleen RPM-paketoitua sovellusta voidaan siis hyödyntää kaikissa sovellusta suorittavissa ympäristöissä. Tämä tietysti edellyttää, että kaikki ympäristöt voivat käyttää samaa, mukauttamatonta sovellusversiota, eli samaa sovellusbinääriä.

SaaS-toimintamallissakin joitakin sovelluksen käyttöönoton loppuvaiheita joudutaan tekemään käsin, sillä näiden automaatio on toistaiseksi nähty liian työlääksi. Nämä vaiheet koskevat normaalisti sovelluksen käyttöliittymästä tehtäviä asetuksia, jotka sovellukset tallentavat tyypillisesti tietokantaansa. Näidenkin toimenpiteiden automaatiota ollaan suunniteltu, jolloin voitaisiin automatisoida joitakin peruskonfiguraatiotoimenpiteitä, kuten tiettyjen liitännäisten käyttöönottoa. Kuva 9 kuvaa SaaS-toimintamallilla tehtävää asennusprosessia.



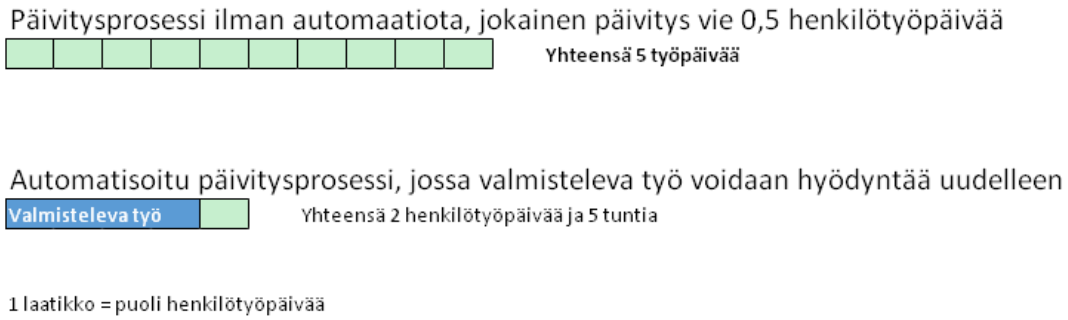
KUVA 9. SaaS-toimintamallin mukainen asennusprosessi.

8.3 Liiketoiminnallisten vaikutusten arviointi

SaaS-toimintamallilla on siis perinteiseen toimintamalliin nähden selvä etumatka, kun tarkastellaan pelkästään läpimenoaikaa. Toisaalta työhön käytetty aika on tyypillisesti ollut asiakkaalta laskutettavaa projektityötä, jolloin herää kysymys, onko sitä järkevää

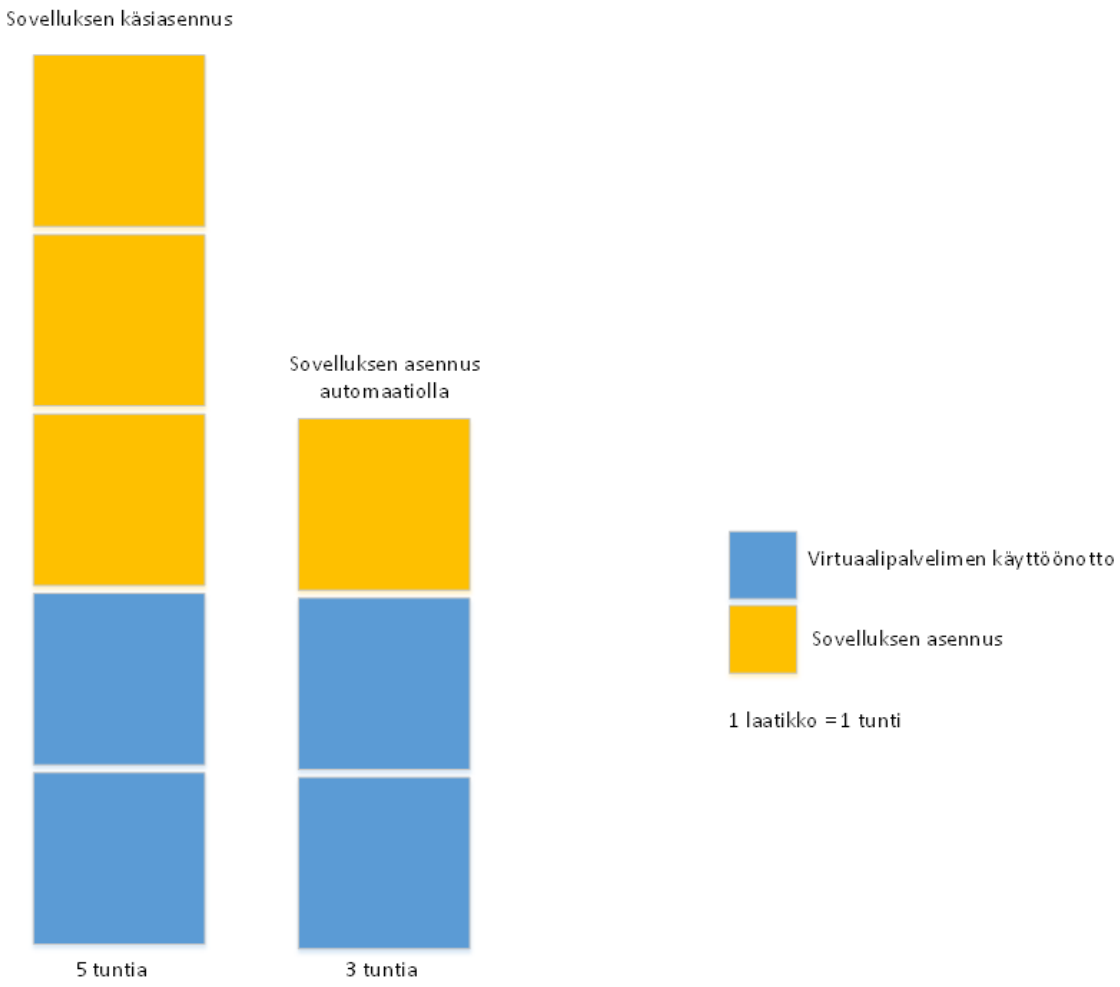
automatisoida. Tällöinhän menetetään käyttöönotosta saatava tuotto. Asia olisi toki näin, jos palveluiden käyttöönottoja ja päivityksiä olisi jatkuvasti tehtävänä rajattomasti. Tällöin toimintaa voisi teoriassa skaalata lisäämällä työn tekijöitä kysynnän mukaisesti. Käytännössä kuitenkin tämän esteenä ovat käyttöönotto- ja päivitystöiden epätasainen jakautuminen suhteessa kalenteriaikaan sekä käytettävissä olevien asiantuntijoiden saatavuus. Modernit sovellukset ovat usein varsin monimutkaisia asentaa, ja niiden asentaminen vaatii kohtalaisen syvällistä ymmärrystä asennettavasta sovelluksesta (Humble & Farley 2011, 6). Tällöin asiantuntijoiden määrän kasvattaminen ei ole optimaalinen ratkaisu satunnaisesti suuren työkuorman keventämiseksi. Kuten luvussa 3.5 todettiin, SaaS-toimintamallin avulla voidaan rutiiniasennukset jättää automaation huolehdittavaksi ja vapauttaa asiantuntijat varsinaisiin asiantuntijatehtäviin, jotka vaativat tilanneriippuvaista päätöksentekoa, eivätkä ole siten helposti automatisoitavissa.

Käytännön kokemus on osoittanut SaaS-toimintamallin läpimenoaikojen olevan korkeamman automaatioasteen johdosta noin 3-4 kertaa lyhyempiä kuin perinteisellä mallilla tehtävät toimenpiteet (käyttöönotot ja päivitykset). Perinteisellä toimintamallilla sovelluksen päivitykseen kuluu noin puoli työpäivää. SaaS-toimintamallissa varsinaiseen päivitykseen kuluu noin 15-30 minuuttia, mutta laskettaessa kokonaistyömäärää täytyy huomioida, että SaaS-toimintamallissa päivitykset tehdään keskitetysti kaikkiin samaa sovellusta käyttäviin ympäristöihin. Tällöin päivitysten valmisteleviin toimenpiteisiin, kuten uuden sovellusversion RPM-paketointiin kuluva työaika täytyy jyvittää instanssikohtaiseen aikaan. Koska valmisteleva työ voidaan hyödyntää jokaisessa käytössä olevassa ympäristössä (toisin kuin perinteisessä toimintamallissa), kasvaa toiminnan tehokkuus instanssimäärän kasvaessa, koska pienempi määrä valmistelevasta työstä jyvittyy instanssikohtaisesti. Kuva 10 havainnollistaa päivitysprosessin valmistelevan osan uudelleenhyödyntämisen merkitystä. Kuvassa suoritetaan 10 perättäistä sovelluspäivitystä samaan kohdeversioon.



KUVA 10. Automatisoidun päivitysprosessin tehokkuus suhteessa vanhaan toimintamalliin

Uusien ympäristöjen käyttöönotossa tehokkuuden kasvu ei ole aivan yhtä huomattava kuin päivityksissä. Virtuaalipalvelimen käyttöönotto tehdään samalla tavalla, mutta sovelluksen asentamiseen käsityönä kuluu enemmän aikaa 2-3 kertaa enemmän kuin automaation avulla tehtävään asennukseen. Merkittävämpi suorituskyvyllinen tekijä on prosessien integraation puute. Palvelimen ja palvelun asennukset tehdään toisistaan erillisinä toimenpiteitä, ja näiden välissä voi kulua paljonkin kalenteriaikaa. Kuva 11 havainnollistaa käyttöönottoautomaation vaikutusta uusien palveluiden käyttöönotossa. Virtuaalipalvelimen käyttöönotto on tälläkin hetkellä varsin pitkälle automatisoitu toimenpide, mutta tiettyjä, lähinnä hallinnollisia asioita on edelleen tehtävä käsin.



KUVA 11. Automatisoidun käyttöönottoprosessin vaikutus uusien palveluiden käyttöönottoon.

Nopeat, ennakoitavat ja helpot sovelluspäivitykset mahdollistavat myös kokonaan uudenlaisia tuotteita, joissa sovelluspäivitykset kuuluvat palvelun kuukausihintaan. Tässä kuitenkin piilee jonkin verran liiketoiminnallista riskiä, sillä myös perinteisessä mallissa väärin arvioidut päivitysprojektien työmäärät ovat toisinaan syöneet projektien kannattavuutta. Kun versiopäivitykset sisältyvät kuukausihintaan, jäävät versiopäivityksen epäonnistuessa ongelmien korjaamisen kustannukset palveluntarjoajan hoidettavaksi. Atlassianin sovellusten kanssa toimittaessa versiopäivityksiin liittyvät ongelmat ovat tyypillisesti liittyneet liitännäisten (plug-in) yhtensopimattomuuteen uudempien sovellusversioiden kanssa. Näiden ongelmien ratkaisuja kuvataan seuraavassa luvussa.

8.4 Arkkitehtuuriin perustuvien ratkaisujen tuotteistaminen

Jotta SaaS-toimintamalliin perustuvien ratkaisujen tunnistetuilta riskeiltä voidaan suojautua, on palvelu käytännössä pakko jossain määrin tuotteistaa. Tuotteistamisella tarkoitetaan palvelun sisällön vakioimista (Jaakkola, Orava & Varjonen 2009, 1), voidaan puhua myös palvelujen konseptoinnista tai kaupallistamisesta (Parantainen 2008, 10). Olennaista on kuitenkin, että palvelua tuotteistettaessa täytyy selvästi määritellä, mitä palveluun sisältyy ja mitä ei (Parantainen 2008, 30).

SaaS-toimintamallia tuotteistettaessa haluttiin asiakkaalle tarjota samat perusasiat, kuin perinteisestikin on tarjottu: tarvittava palvelinkapasiteetti, palvelun asennus mahdollisine lisensseineen sekä palvelun toiminnan varmistavat ylläpitopalvelut. Näiden lisäksi osaksi palvelua haluttiin paketoida sovellusten versiopäivitykset, jotka on aiemmin toteutettu erillisinä päivitysprojekteina. Atlassianin sovellusten tarjoamisena palveluna osalta pääasiallinen motivaatio tälle oli kohdentaa palveluita samaan linjaan Atlassianin itse ylläpitämien Atlassian OnDemand (nykyisin Atlassian Cloud) -palveluiden kanssa. Atlassian tarjoaa omasta konesalistaan sovelluksistaan karsitumpia, mutta myös automaattisesti päivittyviä, versioita Atlassian Cloud -tuoteperheen alla.

Kuten luvussa 8.3 todettiin, aiempi kokemus Atlassianin sovelluksista oli osoittanut suurimpien versiopäivityksiin liittyvien hankaluuksien liittyvän kolmansien osapuolien toteuttamien sovelluksen liitännäisten (plug-in) yhteensopimattomuuteen uusien sovellusversioiden kanssa. Jotta hintaan sisältyvistä sovelluspäivityksistä aiheutuvaa riskiä voitaisiin alentaa Ambientian tuotteistamassa ratkaisussa, päätettiin rajoittaa palveluun tarjottavia liitännäisiä joukkoon ennalta seulottuja ja hyväksi tiedettyjä liitännäisiä. Perinteisellä mallilla toteutetuissa ympäristöissä vastaavia rajoituksia ei ole tehty, mutta toisaalta myöskään versiopäivityksien kustannuksia ei ole sidottu kiinteisiin kuukausihintoihin.

Toinen keskeinen ero perinteiseen palvelumalliin on ollut tarjota palvelua kiinteällä, yksittäisellä kuukausihinnalla. Perinteisessä palvelumallissa asiakkaalle on myyty erikseen tarvittava palvelinkapasiteetti, sovelluksen asennus ja ylläpitopalvelut, sovelluksen lisenssi sekä versiopäivitykset erillisinä projekteina. SaaS-toimintamallissa kaikki nämä asiat on liitetty yhteen kuukausihintaan.

9 POHDINTA

Tässä työssä on käytetty pääasiallisesti sovellusarkkitehtuurillista käsitteistöä ja menetelmiä, vaikka tehty työ ei kuitenkaan ole sisältänyt varsinaista sovellusarkkitehtuurityötä alkuunkaan. IT-alan eri arkkitehtuurilliset suunnat (sovellus-, järjestelmä- ja kokonais-) ovat kuitenkin toiminnoiltaan varsin samankaltaisia. Kaikissa on kysymys tavoitteiden ja vaatimusten ymmärtämisestä ja niiden täyttämisestä suunnittelemalla arkkitehtuuri. Käytännössä suurimmat erot ovat vakiintuneessa käsitteistössä arkkitehtuurisessa suunnasta toiseen (Clements 2013, xiv). Kirjallisuuden näkökulmasta järjestelmäarkkitehtuuri vaikuttaa olevan muihin arkkitehtuurisuuntauksiin nähden jokseenkin laiminlyöty alue, jossa vakiintunutta teoriaa on vähemmän kuin esimerkiksi sovellusarkkitehtuuriin liittyen. Onnekseni sovellusarkkitehtuurilliset menetelmät ovat pääosin käyttökelpoisia myös järjestelmäarkkitehtuurityössä. Bellomo ym. (2014, 5) esittävät sovellus- ja järjestelmäarkkitehtuurin lähenevän tulevaisuudessa toisiaan, ja muodostavan keskenään toisiinsa tiiviimmin integroidun ekosysteemin, jossa perinteisesti kahtena erillisenä osa-alueena toimineet arkkitehtuurit entistä laajemmin vaikuttavat toisiinsa.

Esitelty järjestelmäarkkitehtuuri ja sitä tukeva toimintamalli on tuotteistettu Ambientia Cloud -nimiseksi palveluksi. Palvelun ensimmäinen versio julkaistiin vuoden 2013 lopussa. Tämän jälkeen palvelu on kokenut kaksi merkittävää arkkitehtuurillista iteraatiota, joista jälkimmäinen, niin sanottu kolmannen sukupolven Ambientia Cloud -arkkitehtuuri on tätä kirjoitettaessa maaliskuussa 2015 vielä viemättä tuotantokäyttöön. Erona tällä hetkellä käytettyyn, toisen sukupolven Ambientia Cloud -arkkitehtuuriin on erityisesti tiiviimpi integraatio konfiguraationhallintajärjestelmään. Tässä työssä ollaan kuvattu arkkitehtuuria pääsääntöisesti kolmannen sukupolven arkkitehtuurin näkövinkkelistä. Huolimatta kolmannen sukupolven arkkitehtuurin teknisestä ylivertaisuudesta suhteessa edeltäjäänsä, ollaan jo nyt käytössä olevalla toisen sukupolven arkkitehtuurilla voitu lyhentää sovelluspäivitysten toimittamisen läpimenoaikoja kolmannekseen lähtötilanteesta.

Toistaiseksi Ambientia Cloud -palveluina tarjotaan Atlassianin Confluence -yrityswikiä sekä JIRA-tehtävänhallintajärjestelmää, mutta pitemmän tähtäimen tavoite on laajentaa tarjontaa myös muihin Ambientian tarjonnassa oleviin ratkaisuihin. Monien Ambientian ratkaisujen pohjalla olevien sovellusten tekninen toteutus on sinänsä lähes suoraan yhteensopiva toteutetun arkkitehtuurin kanssa. Tarjonnan laajentamisen haasteet ovatkin

enemmänkin prosessien määrittelyssä ja ennen kaikkea palvelumuotoilussa, eli tuotteistamisessa. Confluence ja JIRA tarjoavat hinnoittelun pohjaksi loogisen lähtökohdan, käyttäjämäärään perustuvan lisenssihinnan. Näin ei kuitenkaan ole kaikkien sovellusten osalta. Esimerkiksi Ambientian ratkaisuissa laajasti käytetyn Liferay Portalin lisenssihinnoitteluun ei ole liitetty minkäänlaisia käyttäjätasoja. Tällöin olisi harkittava aivan uudenlaisia tapoja tehdä hinnoittelua, esimerkiksi käyttöperusteisesti siirretyn datan tai sivulatausten määrän mukaan.

Kaikkia tarjottavia palveluita tuskin voidaan koskaan tuotteistaa niin tehokkaasti, kuin Ambientia Cloud -palvelut on tuotteistettu. Räätälöidyille ratkaisuille on toki paikkansa. Sovellukset toisaalta mahdollistavat jatkuvasti enemmän mukauttamista ilman muutoksia varsinaiseen sovelluksen lähdekoodiin, jolloin taas tässä työssä esitellyn kaltaisille SaaS-arkkitehtuureille löytyy käyttöä. Jos käyttöönottoprosessi on automatisoitavissa ja vakioitavissa, se kannattaa ehdottomasti tehdä. Asiantuntijoiden arvokasta työaikaa säästyy läpi sovelluksen elinkaaren. Automatiikan luomia ympäristöjä on myös helpompi ylläpitää, sillä ei ole syytä epäillä käyttöönottovaiheen inhimillisiä virheitä ongelmien aiheuttajaksi. Tämän työn puitteissa ei kerätty tietoa tästä syntyneestä todellisesta ajan säästöstä, mutta ajatustasolla asia vaikuttaa yksiselitteiseltä.

Työssä esiteltyyn järjestelmäarkkitehtuuriin tiiviisti liittyvää infrastruktuurin automaatiota ollaan tässä työssä käsitelty tietoisesti varsin pintapuolisesti. Kyseessä on varsin laaja ja monitahoinen aihealue. Asiasta kiinnostuneet voivat tutustua tässä työssä hyödynnettyyn infrastruktuuriautomaatioon DI Ville Törhösen diplomityön Designing a Software-Defined Datacenter (Törhönen 2014) kautta. Törhönen on myös osallistunut tässä työssä esitetyn järjestelmäarkkitehtuurin suunnitteluun ja toteutukseen. Siitä välitän tätä kautta hänelle kiitokseni.

Software as a Service -arkkitehtuuri ei ole pelkästään palveluntarjoajien toimintamalli. Sen parhaita käytäntöjä, kuten integroitua konfiguraation- ja paketinhallintaa voi ottaa käyttöön muissakin IT-organisaatioissa, joskin uskon sen tarjoaman automaation ja ympäristöjen vakioinnin hyötyjen tulevan aidosti esiin vasta riittävän suuressa kokoluokassa. Palveluntarjoajienkaan osalta SaaS-toimintamallin käyttöönotolla ei tarvitse välttämättä olla tässä työssä kuvatun kaltaisia liiketoiminnallisia vaikutuksia. Asiakkaille tarjottavia palveluita ei tarvitse välttämättä muuttaa mitenkään, arkkitehtuurin tuomat hyö-

dyt parempana liiketoiminnan skaalautumisena voidaan hyödyntää sisäisesti, esimerkiksi parempana käyttöönottoprojektien kannattavuutena.

LÄHTEET

Alcaraz Calero, J., Edwards, N., Kirschnick, J., Wilcock, L. 2010. Towards an Architecture for the Automated Provisioning of Cloud Services. IEEE Communications Magazine. Volume 48, Issue 12. IEEE Communications Society.

Ambientia Oy. Yritysesittely. Luettu 25.1.2015. http://www.ambientia.fi/fi/ambientia

Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G. Patterson, D., Rabkin, A., Stoica, I., Zaharia, M. 2010. A View Of Cloud Computing. Communications of the ACM Vol 53. New York: Association for Computing Machinery.

Atlassian. 2014. Atlassian ShipIt Days. Luettu 25.2.2015. https://confluence.atlassian.com/display/DEV/Atlassian+ShipIt+Days

Avgeriou, P., Eloranta, V-P., Harrison, N., van Heesch, U., Koskimies, K. 2014. Lightweight Evaluation of Software Architecture Decisions. Teoksessa Bahsoon, R., Eeles, P., Mistrik, I., Roshanak, R., Stal, M. (ed) Relating System Quality and Software Architecture. Waltham, Massachusetts: Elsevier Inc. 157-179

Bachmann, F., Bass, L., Nord, R., 2007. Modifiability Tactics. Pittsburgh: Software Engineering Institute, Carnegie Mellon University

Barbacci, M., Carriere, J., Kazman, R., Klein, M., Lipson, H., Longstaff, T. 1998. The Architecture Tradeoff Analysis Method. Pittsburgh: Software Engineering Institute, Carnegie Mellon University

Bass, L., Clements, P. 2010. Relating Business Goals to Architecturally Significant Requirements for Software Systems. Pittsburgh: Software Engineering Institute, Carnegie Mellon University

Bass, L, Clements, P., Kazman, R. 2013. Software Architecture in Practice. 2013. Kolmas painos. Upper Saddle River, New Jersey: Pearson Education Inc.

Bellomo, S., Ernst, N., Nord, R., Kazman, R. 2014. Toward Design Decisions to Enable Deployability. Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. Pittsburgh: Software Engineering Institute, Carnegie Mellon University

Benatallah, B., Ranjan, R. 2012. Programming Cloud Resource Orchestration Framework: Operations and Research Challenges. Luettu 26.2.2015. http://arxiv.org/abs/1204.2204

Blanchard, B., Fabrycky, W. 2006. Systems Engineering and Analysis. Neljäs painos. Upper Saddle River, New Jersey: Pearson Education Inc.

Carraro, G., Chong, F. 2006. Architecture Strategies for Catching the Long Tail. Luettu 22.2.2015. https://msdn.microsoft.com/en-us/library/aa479069.aspx

Clements P. 2013. Esisanat teoksessa Bahsoon, R., Mistrik, I., Stafford, J., Tang, A. (ed.) Aligning Enterprise, System, and Software Architectures. Hershey, Pennsylvania: IGI Global, xiv-xviii.

Clements, P., Kazman, R., Klein, M. 2000. ATAM: Method for Architecture Evaluation. Pittsburgh: Software Engineering Institute, Carnegie Mellon University

Hohmann, L., 2003. Beyond Software Architecture. Creating And Sustaining Winning Solutions. Upper Saddle River, New Jersey: Pearson Education Inc.

Humble, J., Farley, D. 2011. Continuous Delivery. Reliable Software Releases Through Build, Test and Deployment Automation. Upper Saddle River, New Jersey: Pearson Education Inc.

ISO/IEC. 2011. ISO 25010:2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

Jaakkola, E., Orava, M. Varjonen, V. 2009. Palvelujen tuotteistamisesta kilpailuetua. Opas yrityksille. Helsinki: Tekes.

Jaakkola, H., Thalheim, B. 2011. Architecture-Driven Modelling Methodologies. Teoksessa Heimbürger, A., Kiyoki, Y., Tokuda, T., Jaakkola, H., Yoshida, N. (ed.) Information Modelling and Knowledge Bases XXII. Amsterdam: IOS Press BV, 97-116.

Kernighan, B., Pike, R. 1984. The UNIX Programming Environment. New Jersey: Prentice Hall Inc.

Linux Foundation. 2010. Linux Standard Base Core Specification 4.1. Linux Foundation.

Lukka, K. 2010. Konstruktiivinen tutkimusote. Luettu 27.2.2015. http://www.metodix.com/fi/sisallys/01_menetelmat/02_metodiartikkelit/lukka_const_research_app/

Luoma, E., Rönkkö, M. 2011. Software-as-a-Service Business Models. Communications of the Cloud Software, Volume 1, Issue 1. Espoo: Cloud Software Finland

Mell, P., Grance, T. 2011. The NIST Definition of Cloud Computing. Gaithersburg, Maryland: National Institute of Standards and Technology

Murdock, I. 2007. How Package Management Changed Everything. Luettu 27.2.2015. http://ianmurdock.com/solaris/how-package-management-changed-everything/

OpenStack Foundation. Heat - OpenStack Orchestration. Luettu 26.2.2015. https://wiki.openstack.org/wiki/Heat

OpenStack Foundation. OpenStack Logical Architecture. Luettu 27.2.2015. http://docs.openstack.org/admin-guide-cloud/content/logical-architecture.html

Parantainen, J. 2008. Tuotteistajan pikaopas 3.0. Espoo: Noste Oy.

PCI Security Standards Council. 2013. PCI DSS Cloud Computing Guidelines. https://www.pcisecuritystandards.org/pdfs/PCI_DSS_v2_Cloud_Guidelines.pdf

Red Hat. 2014. Introducing Red Hat Satellite 6. Luettu 17.3.2015. https://access.redhat.com/sites/default/files/attachments/rh_satellite_6_datasheet_us_12513777_0814_sw_web.pdf

Ries, E. 2011. The Lean Startup. How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. New York: Random House LLC.

Schroff, G. 2010. Enterprise Cloud Computing. Cambridge: Cambridge University Press.

Schwaber, K., Sutherland, J. 2013. The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game. Boston, Massachusetts: Scrum.org

TEPA – Sanastokeskus TSK:n termipankki. JHS-suositusten käsitteistö. Luettu 1.2.2015. http://www.tsk.fi/tepa/netmot.exe?page=results&UI=figr&Opt=8&dic=4&SearchWord=saas

Törhönen, V. 2014. Designing a Software-Defined Datacenter. Tampereen teknillinen yliopisto. Tieto- ja sähkötekniikan tiedekunta. Tietotekniikan laitos. Diplomityö.