Tampereen yliopisto

Miika Heinonen

LEGACY-OHJELMISTON PILVIMIGRAATIO

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Prof. Kari Systä
Lokakuu 2020

TIIVISTELMÄ

Miika Heinonen: Legacy-ohjelmiston pilvimigraatio
Diplomityö
Tampereen yliopisto
Tutkinto-ohjelma
Lokakuu 2020

Pilvipalvelujen käyttö ohjelmistoalustoina yleistyy jatkuvasti. Uusia ohjelmistoja näille alustoille toteutettaessa on pilvinatiivi arkkitehtuuri ja pilven tarjoamat mahdollisuudet otettavissa huomioon alusta alkaen. Uusien ohjelmistojen lisäksi pilvipalveluihin migroidaan myös vanhoja ohjelmistoja, joiden arkkitehtuuri ei suoraan vastaa pilvinatiivia ajatusmaailmaa. Pilvipalvelualustat tarjoavat kuitenkin useita hyötyjä myös näille legacy-ohjelmistoille. Tässä työssä käsitellään eri vaihtoehtoja migraation toteutukseen kyseisille ohjelmistoille.

Tätä työtä varten toteutettiin .NET-ohjelmisto, jolla vertailtiin migraatioprosessia kolmelle eri pilvipalvelulle: Amazon AWS, Microsoft Azure ja Google Cloud Platform. Vertailussa ohjelmisto siirrettiin kullekin alustalle sekä PaaS-, että FaaS-arkkitehtuureille pohjautuviin ratkaisuihin. Vertailun perusteella varsinkin Azuren ja AWS:n olevan kypsiä alustoja monipuoliselle .NET-ohjelmistojen pilvimigraatiolle.

Työssä esitellään päätöksentekomalli, jonka tarkoituksena on tarjota työkalu erilaisten migraatiopolkujen vertailuun. Päätöksentekomalli perustuu Cloudstep-malliin ja pilvimigraation kuuteen strategiatyyppiin. Mallissa kartoitetaan sekä organisaation, ohjelmiston, että pilvipalvelualustan rajoitteet, joiden perusteella voidaan löytää sopiva pilvipalvelu, palvelutyyppi ja migraatiostrategia. Mallin eri vaiheet käydään myös läpi .NET-ohjelmistojen näkökulmasta, yhdistäen tähän käytännön vertailusta saadut kokemukset.

Työn tuloksena tuotettu päätöksentekomalli mahdollistaa eri vaihtoehtojen vertailun loogisella, rajoitteisiin perustuvalla tavalla ja antaa mahdollisuuden arvioida pilvimigraatiota kokonaisuutena organisaatiosta aina pilvipalveluun ja teknisiin ratkaisuihin asti. Sopivia migraatioprosesseja on kuitenkin paljon, ja ne riippuvat lukemattomista eri muuttujista, joten kattavan mallin luominen on haastavaa. Tässä työssä esitelty malli kuitenkin tarjoaa vähintäänkin yhden työkalun lisää sopivien migraatiopolkujen arviointiin.

Avainsanat: pilvipalvelut, migraatio, .NET, Google Cloud Platform, AWS, Azure, Cloudstep

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Miika Heinonen: Cloud migration of legacy application
Master of Science Thesis
Tampere University
Degree Programme
October 2020

The usage of cloud platforms in software development is rising all the time. For new software, the adoption of cloud native architecture and usage of cloud services is easy to implement from the beginning. But for legacy-applications this is not the case. Given the usually monolithic architecture of said applications they cannot take full benefit of the cloud platforms. Still, many benefits can be found even without refactoring. The objective of this master's thesis is to survey different ways to migrate legacy applications to cloud environments.

A simple .NET-application was implemented for this master's thesis, to compare different migration processes on three different cloud platforms: Amazon AWS, Microsoft Azure and Google Cloud Platform. The application was migrated to each provider's app service platform and serverless platform. The comparison showed that both Azure and AWS are mature enough platforms for migrating .NET applications, while GCP is only viable for .NET applications after refactoring.

A model for comparing different cloud migration paths was introduced as part of this thesis. The model is based on the Cloudstep decision model and the six cloud migration strategies. The model takes in account organization's, application's, and cloud platform's constraints for the cloud migration, which makes it possible to find the right match of platform, service type and strategy. The model is then reviewed in .NET-application's context, combining the model with the findings from the practical comparison.

The model introduced in this thesis makes it possible to compare different options for cloud migration in a logical, constraint-based way. This allows to evaluate the process, from organizational aspects to cloud platforms and the application itself. However, there are lots of possible and suitable ways for cloud migration and multiple different parameters at play. This makes it hard to create a comprehensive model, however, this thesis introduces one more tool for evaluating the possible migration strategies.

Keywords: cloud platforms, migration, .NET, Google Cloud Platform, AWS, Azure, Cloudstep

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Diplomityöni aihe valikoitui lopulta hiukan sattumalta. Olin jo hyvän aikaa etsinyt sopivaa aihetta asiakasprojekteista, kun silloinen esimieheni ehdotti pilvimigraatiomahdollisuuksien kartoittamista erääseen projektiin. Kyseinen projekti ei lopulta koskaan toteutunut, mutta idea jäi muhimaan. Lopulta päädyin toteuttamaan tuon samaisen idean, mutta yleistetymmässä muodossa ja itsenäisesti. Pilvipalvelut olivat aiheena kiinnostava, vaikka kokemukseni niistä olikin alkuvaiheessa vähäinen. Opin kuitenkin tämän prosessin aikana paljon ja uskon työskenteleväni pilvipalveluiden parissa vielä jatkossakin.

Jo opintojen alussa ajattelin, ettei minun ole välttämätöntä valmistua viidessä vuodessa. Tämä ajatus todellakin toteutui, sillä aikani yliopistossa antoi mielenkiintoisten opintojen lisäksi todella paljon. Haluaisin kiittää kaikkia niitä tahoja, jotka ovat minua tavalla toisella opintojeni aikana tukeneet ja tehneet hiukan venähtäneestä opiskeluajastani unohtumattoman kokemuksen. Kiitos Urrrheilujoukkue NMKSV vertaistuesta ja kannustuksesta niin koko opiskeluaikana, kuin diplomityön kanssa tuskaillessakin. Kiitokset Tietoteekkarikillalle, Fuksijaostolle/neuvostolle, Teekkarijaostolle ja Rakkauden Wappuradiolle, että tarjositte hienon yhteisön, jossa pääsin tekemään kaikkea mahtavaa ja oppimaan uutta huikeiden ihmisten kanssa. Kiitos professori Kari Syställe työni ohjaamisesta ja hyvistä huomioista aina kirjoitusprosessin alkuvaiheista loppumetreille asti. Perheelle ja suvulle kiitokset tuesta opintojeni aikana, ja ennen kaikkea sopivasta määrästä naljailua, kun diplomityötä ei alkanut kuulua. Erityiskiitos avovaimolleni Iidalle tuesta, jatkuvasta kannustamisesta ja ennen kaikkea sopivasta patistamisesta, joka mahdollisti tämän työn valmistumisen.

Tampereella, 11. lokakuuta 2020

Miika Heinonen

SISÄLLYSLUETTELO

1 Johdanto 1
2 Pilvipalvelut ja -arkkitehtuurit 3
2.1 Pilvipalvelut 3
2.2 Pilvinatiivi arkkitehtuuri 3
2.3 Mikropalveluarkkitehtuuri 4
2.4 Skaalaaminen 5
2.5 Pilvipalvelutyypit 6
2.5.1 IaaS 6
2.5.2 PaaS 6
2.5.3 SaaS 7
2.5.4 FaaS 7
2.6 Säiliönti 9
3 Pilvimigraatio 11
3.1 Legacy-ohjelmistot 11
3.2 Migraatiostrategiat 11
3.2.1 Palvelunvaihto 12
3.2.2 Alustanvaihto 13
3.2.3 Uuden ostaminen 14
3.2.4 Refaktorointi 14
3.2.5 Pitäytyminen 15
3.2.6 Luopuminen 15
3.3 Cloudstep-päätöksentekomalli 16
3.4 Ohjelmiston refaktorointi 18
4 .NET-migraatio eri alustoilla 19
4.1 Testisovellus ja -ympäristö 19
4.2 Testin kuvaus 21
4.3 Pilvisovellusalusta 22
4.3.1 Azure App Service 22
4.3.2 AWS Elastic Beanstalk 23
4.3.3 Google App Engine 24
4.4 Serverless 25
4.4.1 Azure Functions 25
4.4.2 AWS Lambda 26
5 Migraatiovertailun tulokset 28
5.1 Pilvisovellusalusta 28
5.2 Serverless 29

6 Migraation päätöksentekoprosessi 31
6.1 Organisaatioprofiilin määrittäminen 31
6.2 Organisaatiotason rajoitteiden määrittely 32
6.3 Ohjelmistoprofiilin määrittäminen 33
6.3.1 Käyttöprofiili 33
6.3.2 Tekninen profiili 33
6.4 Pilvipalveluprofiilin määrittäminen 34
6.5 Teknisten ja taloudellisten rajoitteiden arviointi 36
6.6 Migraatiostrategian valinta 38
6.7 Strategian valinta .NET-ohjelmistolle 42
6.7.1 Palvelunvaihto 43
6.7.2 Alustanvaihto 43
6.7.3 Refaktorointi 44
6.7.4 Luopuminen ja pitäytyminen 47
7 Yhteenveto 48
Lähteet 49

KUVALUETTELO

2.1 Resurssien jakaminen eri pilvipalvelutyypeissä. 9

3.1 Cloudstep-mallin vuokaavio. 17

4.1 Luokkakaavio sovelluksen rakenteesta. 20

6.1 Taloudellisten rajoitteiden vaikutus mahdollisiin strategiavalintoihin. 39

TAULUKKOLUETTELO

6.1 Pilvipalvelutyyppien sopivuus eri strategiatyypeille. 40
6.2 Palvelutyypin valinta tarpeiden perusteella.
Ominaisuuksien ja tarpeiden sopivuutta arvioitu asteikolla 0-3, jossa 3 tarkoittaa parasta sopivuutta. 42

OHJELMA- JA ALGORITMILUETTELO

4.1 Transaktioiden haku alkuperäisessä sovelluksessa. 21
4.2 Transaktoiden haku .NET Core -totetuksessa. 24
4.3 Transaktioiden haku Azure Functions -sovelluksessa. 26
4.4 LambdaEntrypointin määrittely. 27

LYHENTEET JA MERKINNÄT

.NET	Microsoftin ohjelmistokehys
ASP.NET	.NET-ohjelmistokehyksen verkkopalveluiden kehittämiseen tarkoitettu laajennos
AWS	Amazon AWS -pilvipalvelualusta
CI/CD	jatkuva integraatio ja julkaisu (Continuous integration and delivery)
CRM	Asiakkuudenhallinta (engl. Customer relationship management)
CSP	pilvipalveluntarjoaja (engl. CSP)
FaaS	Palvelimeton arkkitehtuuri (engl. Function as a Service)
FTP	Tiedonsiirtomenetelmä kahden tietokoneen välillä (engl. File Transfer Protocol)
GCP	Google Cloud Platform -pilvipalvelualusta
IaaS	Infrastruktuuri palveluna (engl. Ifrastructure as a Service)
IDE	Ohjelmointiympäristö (engl. Integrated development environment
MSSQL	Microsoft SQL -tietokanta
PaaS	Sovellusalusta palveluna (engl. Platform as a Service)
QOS	Palvelun laatuvaatimukset (engl. Quality of Service)
SaaS	Sovellus palveluna (engl. Software as a Service)
SQL	Tietokantojen käyttämä kyselykieli (engl. Structured Query Language)
VPN	Virtuaalinen erillisverkko (engl. Virtual private network)

1 JOHDANTO

Tämän työn tarkoituksena on kartoittaa ja arvioida vaihtoehtoja migroitaessa .NET-ohjelmistoa pilvialustalle. Arvioinnin lisäksi pyritään muodostamaan ohjeistus päätöksentekoon migraatiostrategian ja teknisen toteutuksen valintaan. Työssä keskitytään .NET ohjelmointikehyksen vanhempiin versioihin, käytännössä versioon 4.5.1, joka valittiin, koska varsinkin oman kokemuksen mukaan kyseistä versiota käytetään vielä laajasti, ja nykyiset kehitysympäristöt tukevat sitä. Version 4.5.1 tuki on kuitenkin kehittäjän puolelta lopetettu vuonna 2016 [43]. Vaikka käytännön testit on tehty vain yhdelle versiolle, käsitellään työssä yleisesti legacy-ohjelmistoja. Vanhentuneilla ohjelmistokehyksen versioilla on toteuttu erittäin paljon ohjelmistoja, joiden pilveen migroimista varmasti harkitaan viimeistään nyt. Vanhempia sovellusversioita migroitaessa ongelmaksi muodostuu joidenkin pilvipalveluntarjoajien kohdalla suoran tuen puute. .NET-maailmassa uudempi .NET Core on paremmin tuettu eri alustoilla, osittain siitä syystä, että sillä toteutetut palvelut ovat suoritettavissa linux-ympäristössä. Legacy-ohjelmiston siirtämisellä pilvipalveluun sellaisenaan ei myöskään saavuteta kuin osa pilven tarjoamista hyödyistä. Tästä syystä refaktorointi on usein kannattava vaihtoehto, jos halutaan hyötyä uudesta alustasta mahdollisimman paljon. Ensimmäinen päätös migraatiota aloittaessa onkin siis tehtävä sen suhteen, kuinka paljon sovellusta ollaan valmiita muuttamaan, ja onko ohjelmointikehyksen päivitys mahdollista tai tarpeellista. Tästä päätöksestä riippuu strategian lisäksi käytettävä pilvipalvelutyyppi ja valittava palveluntarjoaja. Niin migraatiostrategian kuin alustan valintaa ohjaavat teknisten ominaisuuksien lisäksi muutkin seikat, kuten taloudelliset ja organisaatiotason rajoitteet. Tässä työssä pyritään muodostamaan selkeä käsitys mitä päätöksiä on tehtävä, ja mitkä ovat näiden päätösten seuraukset, ja mitä mahdollisuuksia ne avaavat.

Tutkimusta varten toteutettiin yksinkertainen .NET-ohjelmisto. Ohjelmisto koostuu rajapintasovelluksesta, tietokannasta ja toisesta sovelluksesta, joka hakee rajapinnasta transaktiotietoja, joista koostetaan yksinkertaisia raportteja. Tämä erillinen raporttisovellus on toteutettu konsolisovelluksena, kuten on ominaista Windows-palveluille, jollaista sovellus tässä työssä edustaa. Ohjelmisto on toteututtu vanhahkolla .NET-versiolla ja monille legacy-järjestelmille ominaisella arkkitehtuurilla. Tutkimuksessa on tarkoitus toteuttaa sovelluksen migraatio kolmelle eri alustalle (AWS, Azure, GCP) pilvisovellusalustalle, säiliönä ja serverless-ympäristössä, sekä toteuttaa tarvittavat muutokset. Pyrkimyksenä on kartoittaa eri toteutustapojen haasteita ja arvioida mm. työn määrää ja tarvittavan refaktoroinnin laajuutta, laajemman ohjelmiston tapauksessa. Varsinaista suorituskykyvertailua ei tämän työn puitteissa ole mielekästä tehdä, sillä suorituskykyyn vaikuttavat suures-

ti valitut palvelutasot eri alustoilla, eikä yksinkertainen ohjelmisto ole soveltuva suurien datamäärien käsittelyyn mielekkäästi.

2 PILVIPALVELUT JA -ARKKITEHTUURIT

2.1 Pilvipalvelut

Tässä työssä käsiteltävät pilvipalvelut (Microsft Azure, Amazon AWS ja Google Cloud Platform) tarjoavat keskenään hyvin samankaltaisia palveluita sovellusten kehittämiseen, ajamiseen ja ylläpitoon [34]. .NET-taustasovelluksen tarpeita ajatellen tärkeimpiä ovat ajoympäristö, tietokannat ja reititystoiminnot. Ajoympäristö voi tarjota käyttöjärjestelmän ja virtualisoidun ympäristön .NET-sovelluksille, tai sovellus voidaan toteuttaa säiliönä. .NET-ohjelmointikehyksellä toteutettuja sovelluksia ei voida ajaa Linux-ympäristössä, joten säiliöiden on oltava Windows-pohjaisia. Uudempi .NET Core -ohjelmointikehys vastaa tähän ongelmaan, tarjoten mahdollisuuden ohjelmien suorittamiseen niin Windows-, Linux- kuin Mac-ympäristöissä [2].

Pilvipalvelut voidaan jakaa IaaS-, PaaS-, FaaS- ja SaaS-palveluihin [44]. Näistä arkkitehtuureista tässä työssä IaaS:ia edustavat virtuaalipalvelimet, PaaS:ia verkkosovellusalustat ja FaaS:ia serverless-palvelut. Näiden lisäksi tässä työssä käsitellään säiliöitä, jotka voidaan toteuttaa usealla eri pilvipalvelutyypillä. Säiliöitä on mahdollista ajaa niin virtuaalipalvelimilla-, kuin pilvisovellusalustoilla, mutta tämän lisäksi palveluntarjoajat tarjoavat palveluna erilaisia orkesterointiratkaisuja, jotka useiin pohjautuvat Kubernetekseen [27]. Säiliöratkaisuiden tarjoamisesta palveluna käytetään joissakin yhteyksissä myös nimitystä CaaS, eli Container as a Service [51]. Tässä työssä puhuttaessa säiliöistä tarkoitetaan yleensä säiliöitä palveluna, jos tätä ei erikseen täsmennetä. Säiliöt ovat erittäin suosittu tapa toteuttaa mikropalveluarkkitehtuurin mukaisia sovelluksia pilvipalveluissa [42].

2.2 Pilvinatiivi arkkitehtuuri

Cloud Native Foundationin määritelmän mukaan pilvinatiivit teknologiat mahdollistavat sovellusten kehittämisen ja ajamisen moderneissa, dynaamisissa ympäristöissä, kuten julkisissa, yksityisissä ja hybrideissä pilvialustoissa. [25] Tätä ajattelutapaa edustavat esimerkiksi säiliöt, mikropalveluarkkitehtuuri, dynaaminen infrastruktuuri ja deklaratiiviset rajapinnat. Pilvinatiiveilla arkkitehtuureilla tarkoitetaan arkkitehtuureja, jotka käyttävät hyväkseen pilvialustojen tarjoamia palveluja, ja toisaalta ottavat huomioon pilven erityispiirteet. Perinteinen monoliittinen arkkitehtuuri soveltuu tähän huonosti, sillä esimerkiksi skaalautuvuuden toteuttaminen ei onnistu mielekkäästi. Monoliitti-sovellusta on mahdol-

lista ajaa virtuaalikoneella ja kuorman kasvaessa pystytetään uusia vastaavia virtuaalikoneita levykuvasta. Tämä on kuitenkin tapana kankea, eikä varsinaisesti vastaa pilviarkkitehtuurien toimintatapoja [42]. Mikropalvelupohjaiset arkkitehtuurit puolestaan pyrkivät pilkkomaan sovelluksen toiminnot omiksi palveluikseen, jolloin on mahdollista luoda yksittäisestä palvelusta useita kopioita kyseisen palvelun kuorman kasvaessa. Monoliitinkin on kuitenkin teoriassa mahdollista toimia pilvinatiivilla tavalla. Tällöin Monoliitin on käytettävä hyväksi pilvipalvelun rajapintaa ja sen tarjoamia palveluita toteuttaakseen pilvinatiiville sovellukselle ominaisia toimintoja [17]. Perinteiset arkkitehtuurit on yleensä suunniteltu kiinteää hyvin tunnettua ja itse hallittua infrastruktuuria ajatellen. Näiden ympäristöjen hallinta ja muokkaaminen on työlästä ja täten arkkitehtuurit yleensä keskittyvät toimimaan rajatulla määrällä tiettyjä komponentteja. Pilvialustojen yhteydessä voidaan kuitenkin toimia joustavammin, sillä pilvipalvelujen tapauksessa käytössä olevien resurssien muuttaminen on nopeaa. [5]

Pilvinatiivi arkkitehtuuri soveltuu luonnostaan myös jatkuvaan julkaisuun ja integraatioon (CI/CD, Continuous Integration and Delivery), sillä tuoreimmat muutokset voidaan julkaista testiympäristöön ja sieltä edelleen tuotantoympäristöön vaivattomasti. Automaation luominen vie luonnollisesti oman aikansa, mutta saatavat hyödyt ovat yleensä vaivan arvoisia [5]. Toimiva automaatio vaatii suunnittelemista jo arkkitehtuuritasolla. Jos järjestelmän suunnittelussa ei olla otettu alusta asti huomioon pilvinatiiveja ominaisuuksia ja vaatimuksia, ei edes hyvin suunnittelusta testi- ja julkaisuautomaatiosta saada kaikkea irti. Pilvinatiivin järjestelmän tulisi olla suunniteltu skaalautumaan kuorman mukaisesti ja palautumaan itsenäisesti mahdollisista virhetilanteista. Arkkitehtuuritasolla tämä tarkoittaa esimerkiksi ongelmien eriyttämisestä huolehtimista [10]. Jos esimerkiksi mikropalveluarkkitehtuurin mallin mukaisesti jokaiselle erilliselle toiminallisuudelle kehitetään oma palvelu, voi muu järjestelmä jatkaa toimintaansa virhetilanteessa, samoin kuin vioittuneen palvelun muut instanssit. Samaan tapaan järjestelmän kuorman muuttuessa voidaan pystyttää uusia instansseja juuri niistä mikropalveluista, joihin lisääntynyt kuorma kohdistuu. Mikropalveluiden sijaan vastaava järjestelmä voidaan luoda myös serverless-arkkitehtuurilla, erona se, että uusia resursseja ei tarvitse erikseen varata. Vikatilanteetkin kohdistuvat vain yksittäiseen funktion suoritukseen, ellei kyseessä ole vika ohjelmakoodissa itsessään, jolloin ongelmaa ei voida korjata automaation avulla. Vikatilanteiden, samoin kuin kuorman ja resurssien käytön seuraaminen on pilvipalveluissa tärkeää. Tästä syystä ohjelmisto tulee suunnitella helposti ja selkeästä monitoroitavia. Tällä tavoin pysytään ajan tasalla järjestelmän käytöstä, samoin kuin eri palveluiden ja resurssien kunnosta. Monitorointiin voidaan myös liittää automaatiota; sen sijaan, että lokitetaan virhe levyn täyttyessä, allokoidaankin automaattisesti lisää levytilaa, mahdollistaen näin järjestelmän käytön jatkuminen.

2.3 Mikropalveluarkkitehtuuri

Pilvinatiiveille järjestelmille on ominaista noudattaa mikropalveluarkkitehtuuria, sillä pilvelle uniikit ominaisuudet, kuten skaalautuvuus ja virheensietokyky perustuvat mahdolli-

suuteen pystyttää uusia palveluita vaivattomasti ja automatisoidusti esimerkiksi kuorman kasvaessa, tai virhetilanteessa. Pilvinatiivien sovellusten ominaisuuksiin kuuluu myös jatkuvan integraation ja julkaisun käyttäminen. [10] Tähänkin mikropalveluarkkitehtuuri on sopiva, sillä arkkitehtuurin ansiosta on mahdollista julkaista päivityksiä yksittäisiin mikropalveluihin, vaikuttamatta muiden palveluiden toimintaan, pitäen näin saatavuuden korkeana.

Mikropalveluarkkitehtuurissa ohjelmisto on jaettu moneen palveluun, yleensä siten, että jokainen palvelu vastaa yksittäisestä ominaisuudesta, ja vain siitä. Ohjelmiston eri osat kommunikoivat keskenään rajapintojen kautta [11]. Näin sovelluksen eri osat eivät ole riippuvaisia toisistaan, ja ongelmien eriyttämisen ansiosta ohjelman rakenne pysyy selkeänä ja helposti korjattavana. Mikropalveluiden suosio on noussut varsinkin pilvipalveluiden keskuudessa, niiden tarjoamien lukuisten pilvinatiivien ominaisuuksien vuoksi. Monoliittiseen arkkitehtuuriin verrattuna mikropalveluiden etuna on niiden ketteryys [26]. Käytönaikaisten hyötyjen lisäksi ketteryys näkyy siinä, että uusia mikropalveluita on vaivatonta kehittää riippumatta jo olemassa olevista palveluista. Toisaalta mikropalveluarkkitehtuuria noudattavan sovelluksen suunnittelussa on enemmän työtä, mutta ylläpito ja jatkokehitys ovat vuorostaan helpompia [26]. Mikropalvelut toteutetaan useimmiten säiliönä, sillä ne soveltuvat luonnostaan palvelun jakamiseen useaan pienempään palveluun. Mikropalvelut ja varsinkin säiliöt mahdollistavat myös monipuolisen teknologioiden käytön, sillä jokainen palvelu voidaan kehittää käyttäen niitä kieliä ja työkaluja, jotka koetaan sopivimmiksi [55]. Mikropalveluarkkitehtuurin mukainen sovellus on mahdollista toteuttaa lisäksi myös pilvisovellusalustoille ja serverless-funktioilla.

2.4 Skaalaaminen

Sovellusten skaalaamisella tarkoitetaan resurssien lisäämistä palvelun toiminnan varmistamiseksi. Skaalaaminen jaetaan yleensä vertikaaliseen ja horisontaaliseen skaalaamiseen [24]. Pilvialustoilla skaalaaminen voi tapahtua automaattisesti ennalta asetettujen rajojen mukaan, tai käyttäjän toimesta manuaalisesti [12]. Vertikaalisesta skaalauksesta puhuttaessa tarkoitetaan resurssien lisäämistä nykyisellään käytettävissä olevaan instanssiin. Vertikaalinen skaalaaminen ei yleensä ole yhtä joustavaa kuin horisontaalinen, sillä resursseja ei voida lisätä ja vähentää samalle instanssille ilman viiveitä. Lisäksi esimerkiksi tallennuskapasiteettia ei voida skaalata edestakaisin jo pelkästään mahdollisen datan menetyksen takia.

Tässä työssä käsitellään yleensä skaalauksesta puhuttaessa horisontaalista skaalaamista, jossa uusia sovellusinstansseja lisätään, jotta voidaan vasta kasvavaan kysyntään tai poistetaan säästöjen saavuttamiseksi. Jotta horisontaalinen skaalaaminen on hyödyllistä, tulee skaalattavan sovelluksen olla rakennettu siten, että uusien sovellusinstanssien lisääminen on mahdollista. Horisontaalinen skaalaaminen on mahdollista jopa virtuaalipalvelimilla, mutta palvelimien käynnistysajoista ja ylläpitokustannuksista johtuen eivät virtuaalipalvelimet ole kuitenkaan optimaalinen ratkaisu. Verkkopalvelusovellukset ja konttien

päälle rakennetut sovellukset taas puolestaan skaalautuvat vaivattomasti. Kaikilla tässä työssä käsiteltävien pilvipalveluiden sovellusalustoilla on tarjolla automaattiset skaalauspalvelut, joissa voidaan määrittää eri metriikoiden, kuten suoritinkäytön perusteella, milloin uusia instansseja luodaan [24].

2.5 Pilvipalvelutyypit

Pilvipalvelut voidaan jakaa neljään päätyyppiin [16]. Yhteisenä ominaisuutena kaikille palvelutyypeille voidaan pitää laskutuksen perustumista pääasiassa toteutuneeseen käyttöön, joka erottaa ne perinteisistä palvelinratkaisuista. Kaikki palvelutyypit perustuvat pohjimmiltaan samaan tekniikkaan, sillä riippumatta valitusta palvelusta käytetään samaa palveluntarjoajan laitteistoa. Asiakkaalle palvelut näyttäytyvät kuitenkin täysin erilaisina, sillä ne mahdollistavat erilaisia tapoja hyödyntää pilvipalvelualustojen tarjoamaa. Eri palvelutyypit ovat siis paljon enemmän kuin erilaisia laskutusmalleja, vaikka mallien ymmärtäminen onkin tärkeä osa pilvipalvelun ja -tyypin valintaa.

2.5.1 IaaS

Infastructure as a service, eli IaaS on pilvipalvelutyyppi, jossa asiakkaalle tarjotaan infrastruktuuri, mutta ei korkeamman tason palveluita [52]. Käytännössä tämä tarkoittaa että palvelusta allokoidaan tietyt määrät erilaisia resursseja ja palveluita aina käyttöjärjestelmään asti, mutta käyttöjärjestelmästä eteenpäin hallinta on asiakkaalla. Tässä työssä keskitytään IaaS-palveluista lähinnä virtuaalikoneisiin, mutta yleisesti IaaS-palvelut kattavat myös verkkopalvelut ja muun infrastruktuurin. IaaS-palvelut eivät lopulta eroa dedikoidusta laitteistosta käyttäjän kannalta muuten, kuin että varsinaiseen laitteistoon ei pääse käsiksi. Tämän sijaan käytössä on erilaisia virtualisoituja vaihtoehtoja, kuten virtuaalipalvelimet ja virtuaaliset lähiverkot. Käyttäjän vastuulla on kuitenkin näiden virtualisoitujen laitteiden turvallisuus; esimerkiksi virtuaalipalvelinten päivittäminen on asiakkaan vastuulla.

2.5.2 PaaS

PaaS, eli Platform as a Service tarkoittaa pilvipalvelua, jossa IaaS-palveluista tutun infrastruktuurin lisäksi palveluntarjoaja vastaa myös käyttöjärjestelmästä ja ajoympäristöstä [44]. Sekä käyttöjärjestelmä, että ajoympäristö on siten piilotettu käyttäjältä ja niihin voidaan tehdä muutoksia lähinnä rajoitetusti ja vain valmiiden työkalujen avulla. Taustalla käytössä olevan laitteiston valinta vuorostaan vaikuttaa luonnollisesti suorituskykyyn ja sen kautta hinnoitteluun. Tässä työssä puhutaan PaaS-arkkitehtuurin yhteydessä yleensä pilvisovellusalustoista tai lyhyemmin sovellusalustoista. Termi ei ole varsinaisesti vakiintunut ja sitä käytetään paremman vastineen puuttuessa. Näille alustoille on mahdollista siirtää ajettavaksi yhteensopivaa ohjelmakoodia, ilman tarvetta muokata palvelun pohjalla toimivaa infrastruktuuria. Ohjelma voidaan julkaista alustalle usealla eri tavalla, ku-

ten suoraan kehitystyökaluista julkaisemalla, käyttäen palvelun tarjoamaa julkaisuprofiilia. Jatkuvan julkaisun ja integraation kannalta suositeltavin tapa on alustan yhdistäminen versionhallintaan. Näin alusta saa itse haettua versionhallinnasta muutokset ja päivittää palvelun.

Pilvisovellusalustat mahdollistat parhaimmillaan hyvinkin pilvinatiivin arkkitehtuurin. Jos ohjelmisto jaetaan useaan sovellukseen, jotka jokainen ajetaan omassa palvelussaan, voidaan sovelluksia muokata ja skaalata yksittäin. Skaalaukseen tarjotaan yleensä kattavat automaattisesti toimivat työkalut. Tarkkailuun ja lokitukseen on myös tarjolla omat työkalunsa. Tarkkailun perusteella voidaan myös suorittaa toimenpiteitä. Koska pilvisovellusalustoilla ajoympäristöön ei voi juurikaan vaikuttaa, eikä ajossa ole kuin sovellus itsessään, eivät ne ole kuitenkaan yhtä joustavia ratkaisuja, kuin kontit. Pilvisovellusalustalle siirryttäessä tietokannat on myös eriytettävä sovelluksesta, sillä palveluun laitetaan suoritettavaksi vain ohjelmakoodi. Yleisesti tietokanta kannattaa siirtää saman pilvipalveluntarjoajan tietokantapalveluun.

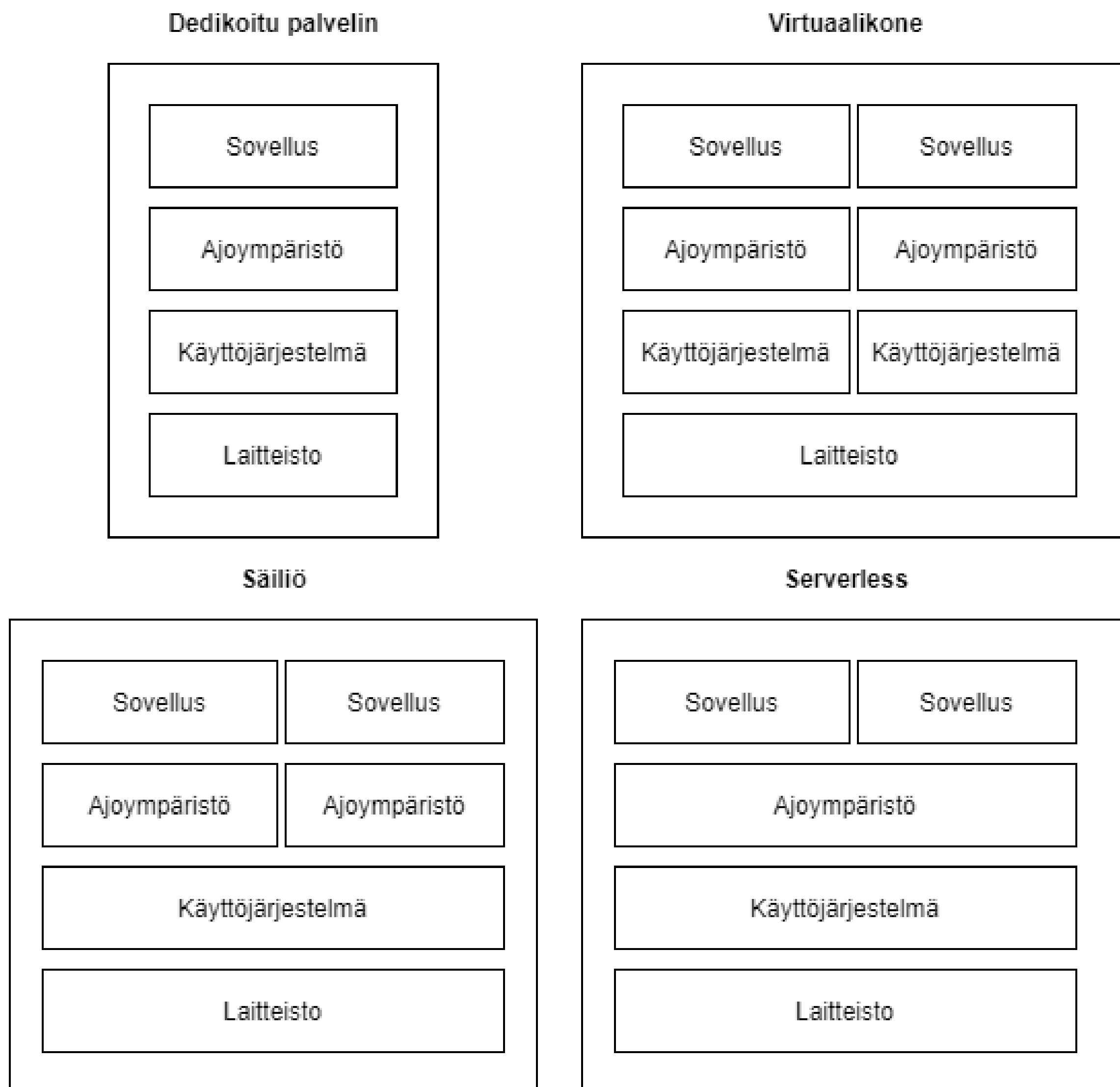
2.5.3 SaaS

Software as Service -pilvipalvelutyypissä asiakkaalle tarjotaan pilvessä ajettava ohjelmisto, johon käyttäjillä on pääsy yleensä selainkäyttöliittymän kautta. SaaS-palveluissa käyttäjällä ei ole lainkaan hallintaa palvelun taustajärjestelmään. Jos SaaS olisi mukana kuvan 2.1 jaottelussa, olisivat kaikki resurssit jaettuja. SaaS:n tapauksessa lisätään usein samankaltaisessa resurssijaossa mukaan vielä data- ja pääsynhallintakerrokset, jotka ovat ainoat asiakkaan hallintaan jäävät resurssit. Tässä mielessä SaaS muistuttaa ostettavaa ohjelmistoa. Esimerkkejä SaaS-palveluista ovat Microsoftin Office 365 -ohjelmistot ja Salesforce CRM-ohjelmisto. Office 365 tarjotaan asiakkaalle melko pitkälti samanlaisena, kuin muillekin asiakkaille, mutta eri asiakasorganisaatiot on eriytetty toisistaan. Salesforce tarjoaa huomattavasti enemmän räätälöinti mahdollisuuksia, mutta koska infrastruktuuri on kaikkien asiakkaiden kesken jaettu, eikä asiakkaille ole siihen minkäänlaista hallintaa, sijoittuu se myös SaaS-palveluiden alle. SaaS-palveluissa on yleistä jatkuvat käyttömaksut palvelun ostamisen sijaan. Tällä tavoin palvelut ovat asiakkaiden käytettävissä joustavasti, eikä kynnys palveluiden käyttöönottoon ole myöskään suuri.

2.5.4 FaaS

Serverless-arkkitehtuuriin perustuva pilvipalvelutyyppi, FaaS (Function as a Service), tarjoaa palvelun, jossa ohjelma koostuu yksittäisistä pilvipalvelussa ajettavista funktioista [44]. Funktiot eivät ole koko ajan ajossa, vaan funktiot suoritetaan vasta jonkin ehdon, kuten ajastimen laukeamisen, täytyttyä. Funktion suorituksen jälkeen funktion käyttämät resurssit vapautetaan. Funktion toimivat siis tilattomasti [54], joka muodostaa omat haasteensa, sillä tarvittavat palvelun tilaan liittyvät tiedot on joko noudettava muualta, tai antaa funktiolle parametreina. Serverless eroaa perinteisistä arkkitehtuureista eniten siinä, että kehittäjä ei ole millään tasolla vastuussa, eikä yleensä edes tietoinen, funktion käyt-

tämistä resursseista tai ajoympäristöstä. Tällä pyritään mahdollistamaan vaivattomampi ja nopeampi ohjelmistokehitys, samalla tehden pilvipalveluntarjoajan resurssienhallinasta joustavampaa [40]. Funktion käyttäessä vain rajattuja resursseja, ja vain sen aikaa, kun on tarvetta, vältytään hukkakapasiteetilta, jota perinteisempien virtualisointiratkaisujen tapauksessa pääsee syntymään, kun ohjelmasta on suorituksessa aina edes jokin osa. Koska serverless-palveluissa ei yleensä varata resursseja funktioita varten ennen suorittamista, syntyy tarvittavien resurssien varaamisesta ja palveluiden käynnistämisestä viivettä. Funktion, jolle ei ole resursseja varattuna käynnistämistä kutsutaankin kylmäkäynnistykseksi [46]. Kylmäkäynnistämistä voidaan estää varaamalla funktiolle resurssit kuten mille tahansa palvelulle, mutta näin menetetään serverless-palvelun hyvistä puolista hinnoittelun joustavuus kuorman mukaan, sillä nyt palvelua laskutetaan varattujen resurssien mukaan. Serverless on vahvemmin abstrahoitu kuin virtuaalikone tai säiliö. Virtuaalikonetta tai säiliötä käytettäessä on koko virtualisoitu tietokone käytettävissä käytännössä kuin mikä tahansa tietokone. Serverless-funktio piilottaa varsinaisen ajoympäristönsä kokonaan. Serverless vaatii myös luonnollisesti oman hinnoittelupolitiikkansa. Perinteisesti pilvipalveluissa hinnoitellaan käytettävien resurssien mukaan, mutta Serverlessin tapauksessa tämä ei ole kovin intuitiivinen tapa, sillä funktioille ei varata resursseja, kuin vasta funktiota suorittaessa. Tästä syystä hinnoittelumallit perustuvat serverless-arkkitehtuurissa yleensä suoritusaikaan ja funktioille varattuun muistin määrään [8].



Kuva 2.1. *Resurssien jakaminen eri pilvipalvelutyypeissä.*

2.6 Säiliönti

Säiliöntiteknologiat ovat tällä hetkellä yksi käytetyimmistä pilviteknologioista, varsinkin kun halutaan hyödyntää pilvinatiivia arkkitehtuuria [42]. Cloud Native Foundationin teettämän kyselyn mukaan 84 prosenttia vastanneista organisaatioista käyttää säiliötä tuotannossa [4]. Konttien avulla on mahdollista paketoida sovellukselle ajoympäristö, joka sisältää kaiken tarvittavan ohjelman ajamiseen, mutta ei mitään ylimääräistä. Kuten kuvassa 2.1 nähdään, jakaa kontti käyttöjärjestelmän muiden samalla tietokoneella ajettavien konttien kanssa. Konttien virtualisointi pitää huolen siitä, että konttien sisältö on eriytetty toisistaan, vaikka pohjalla onkin sama laitteisto ja käyttöjärjestelmä. Tämän ansiosta eri konteissa ajettavilla sovelluksilla on pääsy tarvitsemiinsa resursseihin, kuten kirjastoihin, ja vain niihin. Tämä mahdollistaa muutosten tekemisen yksittäisiin kontteihin, vaikuttamatta muiden toimintaan. Konttien käynnistäminen on myös samasta syystä nopeaa. Sen sijaan, että luotaisiin uusi virtuaalikone käyttöjärjestelmineen, luodaan vain

uusi kontti, jonka yhteydessä tarvitsee ladata käyttöjärjestelmästäkin vain osa. Näin mikropalveluarkkitehtuuria käyttävän sovelluksen skaalaaminen on verrattain yksinkertaista. Yksittäiset säiliöt eivät ole vielä kovinkaan mielekkäitä pilvipalveluiden kannalta. Koska mikropalveluarkkitehtuuri koostuu useista konteista, tarvitaan konttien hallinnointiin omat ratkaisunsa. Kirjoitushetkellä suosituin ratkaisu on Googlen Kubernetes [4], joka mahdollistaa säiliöiden monipuolisen hallinnan [53].

3 PILVIMIGRAATIO

Ennen sopivan strategian valintaa on ohjelmiston ja sen ympäristön nykytilanne ja tavoite kartoitettava mahdollisimman tarkasti; minkälaisista osista sovellus koostuu, mitä riippuvuuksia sovelluksella on, mistä syistä muutosta lähdetään tavoittelemaan ja mitä hyötyjä on saavutettavissa. Kaikista näistä seikoista riippuu valittava strategia ja sen myötä projektin laajuus. On hyvä myös ymmärtää mitkä liiketoiminnalliset tekijät mahdollisesti vaativat tiettyjä toimia, tai mahdollisesti estävät toisia. Tässä luvussa esitellään aluksi kuuden kategorian jaottelu erilaisten migraatiostrategioiden välillä [31]. Jaottelussa ei oteta kantaa tarkempiin teknisiin ratkaisuihin, vaan siihen millaisia toimia migraatio tulee vaatimaan.

3.1 Legacy-ohjelmistot

Legacy-ohjelmistoilla tarkoitetaan ohjelmistoja, joiden toteutustapa on tavalla tai toisella vanhentunut, mutta ohjelma on siitä huolimatta käytössä [48]. Ohjelmiston iästä ei voida suoraan sanoa onko kyseessä legacy-ohjelmisto vai ei, sillä tähän vaikuttavat mm. käytetty ohjelmointikieli, ohjelmointikehys ja arkkitehtuurivalinnat. Termiä käytetään nykyään suhteellisen vapaasti ja varsinkin pilviympäristöistä puhuttaessa legacyksi voidaan kutsua melkein mitä tahansa sovellusta, joka ei vastaa juurikaan pilviympäristöissä käytettäviä ratkaisuja, kuten mikropalveluarkkitehtuuria. Päätöstä pitäytyä vanhoissa legacy-järjestelmissä, sen sijaan, että kehitettäisiin tai ostettaisiin uudet, voitaisiin täydelli-sessä tilanteessa pitää itsessään huonona. Realiteetit kuitenkin tulevat vastaan, yleensä viimeistään rahan muodossa, ja näin päädytään ylläpitämään järjestelmiä, joita voidaan pitää monella mittarilla vanhentuneina. Toimintatavassa ei luonnollisesti ole mitään vikaa, kunhan järjestelmällä on edelleen käyttöä ja ylläpidosta selvitään siedettävällä työmäärällä ja kustannuksilla.

3.2 Migraatiostrategiat

Orban jaottelee erilaiset pilvimigraatiostrategiat kuuteen eri kategoriaan: palvelunvaihto, alustanvaihto, uuden ostaminen, refaktorointi, pitäytyminen ja luopuminen. [31]. Jaottelussa korostuu erilaisten strategioiden kirjo; palvelu voidaan yksinkertaisimmillaan siirtää vain uudelle alustalle, esimerkiksi omalta palvelimelta pilveen. Jo pelkästään tällä tavoin voidaan saavuttaa huomattavia säästöjä, kuten GE Oil & Gas:n tapauksessa saavutetut 30% säästöt palvelinkuluissa [19].

3.2.1 Palvelunvaihto

Palvelu tai sovellus on mahdollista siirtää pilveen sellaisenaan, jolloin on kyseessä alustanvaihto (Rehost, Lift & Shift) [33]. Tämä toimintatapa ei oikeastaan eroa palveluntarjoajan vaihtamisesta perinteisen palveluntarjoajan yhteydessä, varsinkin jos aiemminkin tarvittavat palvelimet on hankittu ulkopuoliselta palvelutarjoajalta, eivätkä ole fyysisesti omia. Pilveen siirtämiseen on oikeastaan vai yksi vaihtoehto; virtuaalikone. Virtuaalikoneelle palvelu, ja sen tarvitsema ympäristö tietokantoineen saadaan siirrettyä kuten mille tahansa palvelimelle, nyt palvelin vain sijaitsee pilvessä.

Palvelunvaihdon valintaa puoltaa sen helppous ja edullisuus. Strategiaa käytettäessä ei tarvitse muokata sovellusta välttämättä ollenkaan. Tämä luonnollisesti säästää kehityskustannuksia, kun ohjelmistokehittäjiltä ei tarvita lisäpanosta sovelluksen saamiseksi pilveen. Pilven hyötyjä tällä strategialla saavutetaan kuitenkin vain harvoja. Ylläpitäminen voi helpottua, ja tulla halvemmaksi, riippuen aiemmasta ratkaisusta. Järjestelmästä saadaan myös jossain määrin skaalautuva [29]. Skaalautuvuus on kuitenkin toteuttavissa vain virtuaalikonetasolla, eli kuorman kasvaessa pystytetään uusi virtuaalikone, joka luonnollisesti tuplaa kustannukset. Virtuaalikoneiden laskutusmallit eivät yleensä ole käyttömääriin perustuvia, vaan resurssit varataan ainakin jossain määrin juuri näitä koneita varten, oli resursseille käyttöä tai ei. Skaalautuvuus vaatii myös kuormantasaajan pystyttämisen ja konfiguroinnin, joka lisää työmäärää. Vastuun siirtyminen palveluntarjoajalle, samoin kuin parempi laiterikkojen sietokyky ovat kuitenkin itsessään varteenotettavia etuja. Hinnoittelumallista ja käytöstä riippuen säästöjäkään ei välttämättä synnyn kuin aluksi, pidemmän ajan kustannukset voivat hyvinkin olla korkeammat, kuin pilvinatiivissa ratkaisussa.

Palvelunvaihtoa kannattaa harkita, jos varsinaisia pilvinatiiveja ominaisuuksia ei tarvita. Tällöin kannattaa kuitenkin miettiä, onko pilveen siirtymiselle oikeita perusteita. On mahdollista, että myöhemmin tässä luvussa esiteltävä uuden sovelluksen ostaminen olisi mahdollinen strategia. Tämä ei tietenkään ole mahdollista kaikille erilaisille ohjelmille, varsinkin jos ne on räätälöity johonkin omaan tarpeeseen. Mahdollista on myös, että myöhemmin esiteltävät pitäytymis- ja luopumisstrategiat tulisivat kysymykseen. Näiden kahden puolesta puhuu osaltaan se, että tahtoa tai resursseja sovelluksen muokkaamiseen pilveen sopivaksi ei ole. Pilveen siirtyminen ei välttämättä ole kaiken maineensa veroista, jos siitä ei olla saamassa riittäviä hyötyjä. Tarvittavan tahdon ja resurssien puute kertoo myös siitä mahdollisuudesta, että ohjelmiston jatkokehitykseen ei yleisesti ole enää tarvetta, jolloin luopumistakin kannattaa vähintään harkita. Legacy-projektien tapauksessa palvelunvaihto on hyvinkin houkutteleva, sillä työn määrä, jota pilvinatiivimmat toteutustavat vaativat on suurempi, sillä yleensä ohjelmiston nykyinen arkkitehtuuri ei suoraan sovellu mikropalveluarkkitehtuuriin.

3.2.2 Alustanvaihto

Alustanvaihdosta (Replatform) on kyse, kun ohjelmistoa muokataan jonkin verran, mutta sitä ei vielä varsinaisesti refaktoroida [31], ainakaan arkkitehtuuritasolla. Muokkaukset voivat olla pilven vuoksi tehtäviä tai muita parannuksia. Esimerkkinä voidaan mainita tietokannan vaihtaminen paikallisesta tietokantapalvelimesta pilvipalvelun tarjoamaan. Vaikka itse palvelu olisikin ajossa virtuaalipalvelimella, voidaan tietokanta vaihtaa pilvialustan tarjoamaan. Usein kyseessä on nimenomaan jonkin kolmannen osapuolen työkalun, kirjaston tai palvelun vaihtaminen toiseen, joko kustannus- tai tehokkuussyistä. Tämä vaatii luonnollisesti muutoksia myös ohjelmakoodiin, mutta jos arkkitehtuuri on riittävän modulaarinen, keskittyvät muutokset usein vain rajapinta- ja kirjastoluokkiin. Jos alustanvaihdossa tehtävät muutokset koskevat arkkitehtuuria tai varsinaista ohjelmalogiikkaa, aletaan puhua jo refaktrointistrategiasta.

Kuten aikaisemmin mainittiin, ei alustanvaihdon tapauksessa yleensä siirrytä käyttämään säiliöintiä tai pilvisovellusalustaa, vaan todennäköisintä on pysytellä virtuaalipalvelimissa, jotka voidaan pystyttää ja konfiguroida aikaisempien palvelimien pohjalta vastaamaan alkuperäistä tilannetta. On kuitenkin mahdollista, että siirrytään käyttämään sovellusalustaa, jossa on mahdollista ajaa .NET-sovelluksia sellaisenaan. Jos käytössä on paikallinen tietokanta, tai samalla palvelimella on ajettu useita palveluita, jotka keskustelevat keskenään, on näiden väliseen kommunikaatioon tehtävä kuitenkin muutoksia. Säiliöratkaisut, vaativat yleensä jo enemmän muutoksia ja refaktorointia, joten niiden tapauksessa ei enää voida puhua alustanvaihdosta. Alustanvaihto on järkevä valinta, kun ollaan halukkaita ottamaan käyttöön joitakin pilvipohjaisia ominaisuuksia, mutta syystä tai toisesta ei olla kuitenkaan valmiita muokkaamaan sovellusta niin paljoa, kuin mitä varsinaiset pilvinatiivit ratkaisut vaativat [32]. Hyödyt jäävät kuitenkin rajallisiksi, mutta nykytilanteesta riippuen saatetaan säästää suuriakin summia, lisätä tehokkuutta huomattavasti tai esimerkiksi parantaa sovelluksen vakautta.

Tässä työssä käsiteltävän esimerkkiprojektin kannalta alustanvaihto tarkoittaisi tietokannan vaihtamista paikallisesta Microsoft SQL-palvelimesta pilvipalvelun tarjoamaan tietokantapalveluun. Jos päädytään käyttämään Microsoft SQL-yhteensopivaa tietokantaa, on muutos pienimmillään yhteyden uudelleenkonfiguroinnista kiinni. Jos samalla halutaan vaihtaa myös tietokantapalvelinta, on mahdollisesti tehtävä muutoksia myös tietokantaluokkiin, sillä eri tietokantatyyppejä varten luodut luokkamallit eivät välttämättä toimi keskenään. Vaihtamiseen johtavat syyt voivat esimerkiksi olla teknisiä tai taloudellisia. Tietokantaoperaatiot on kuitenkin eriytetty omaan luokkaansa, jolloin muutos ei kuitenkaan näy ohjelmalogiikassa asti. Sovelluksessa, jossa arkkitehtuuri ei ole yhtä hajautettu, refaktorointia joudutaan tekemään syvemmällä ja useammassa paikassa, ja varsinkin isomman sovelluksen tapauksessa tämä on suuritöistä ja aikaa vievää. Toteutuksen testaaminen ja toiminnallisuuden varmistus ovat myös haastavampia lisääntyneen monimutkaisuuden vuoksi.

3.2.3 Uuden ostaminen

On mahdollista, ettei nykyistä ohjelmistoa tai jotain sen osaa kannata enää käyttää sellaisenaan, vaan on syytä harkita korvaavan ohjelmiston hankkimista [31]. Ohjelmiston hankintaan on useita eri mahdollisuuksia: ohjelmiston voi toteuttaa itse, jos organisaatiolta löytyy osaamista tai ohjelmisto voidaan hankkia räätälöitynä ulkopuoliselta toteuttajalta. Korvaava ohjelmisto voidaan joissakin tapauksissa myös ostaa sellaisenaan, jos sopiva tuote on olemassa. Kovassa kasvussa on nykyään myös ohjelmiston hankkiminen palveluna, jolloin lähtökohtaisesti hankitaan pääsyoikeudet käytettävään pilvipalveluun. Tässä työssä keskitytään pääasiassa jälkimmäiseen, sillä se on pilvipohjaisena ratkaisuna mielenkiintoisin.

Erilaiset SaaS-ratkaiset ovat yleistyneet kovaa tahtia [39] ja ymmärrettävistä syistä. Aiemmin on ollut hyvinkin yleistä, että moniin yrityksen tarpeisiin, kuten asiakkuuksien hallintaan tai laskutukseen tehdään oma tarpeeseen räätälöity ohjelmisto. Menettelyssä ilmeisenä hyötynä on luonnollisesti se, että saadaan täsmälleen organisaation tarpeita vastaava sovellus. Projekti on kuitenkin yleensä kallis, eikä lopputuloskaan välttämättä aja asiaansa valmista kaupallista ratkaisua paremmin. SaaS-mallissa ei tarvitse keksiä pyörää uudestaan, mutta usein räätälöintimahdollisuuksia on huomattavasti enemmän, kuin valmisohjelmistojen kanssa. Monesti sovelluksien rajapinnat mahdollistavat sovelluksen muokkaamisen myös sisäisesti, mahdollisesti jopa ilman varsinaista ohjelmointiosaamista. Myös ylläpito siirtyy organisaatiolta itseltään palveluntarjoajalle, joka tarjoaa joustoa ja jatkuvuutta.

On hyvin mahdollista, että esimerkiksi asiakkuuksienhallintaan aikoinaan tilattu ohjelmisto ei enää vastaa tarkoitustaan, ja päivitys tai varsinkin pilveä varten refaktorointi alkaa olemaan kallista ja mahdollisesti jopa mahdotonta. Sovellus on voitu toteuttaa jollain vanhemmalla ohjelmointikielellä tai -ympäristöllä, jonka saaminen pilveen edes virtuaalikoneella voi olla haastavaa vanhanaikaisten riippuvuuksien tai ympäristövaatimuksien takia, sillä virtuaalipalvelimetkaan eivät tue kaikkia vanhempia ympäristöjä.

3.2.4 Refaktorointi

Refaktoroinnissa tavoitteena on sovelluksen muokkaaminen siten, että saadaan hyödynnettyä pilvialustan tarjoamia palveluita [31]. Refaktorointi kattaa kaiken ei-triviaaleista luokkatason muutoksista aina arkkitehtuuritason refaktorointiin. Lähtökohtaisesti tavoitteena on muokata ohjelmistoa pilvinatiiviin suuntaan mahdollistaen näin uuden alustan monipuolisen hyödyntämisen [35]. Tämän lisäksi voidaan luonnollisesti tehdä muitakin parannuksia, mutta jotta voitaisiin puhua refaktoroinnista migraatiostrategiana, tulee pääasiallinen syy muutosten takana olla pilven ominaisuuksien hyödyntäminen. Mitä enemmän ollaan valmiita refaktoroimaan, sitä paremmin saadaan hyötyjä irti, esimerkiksi mikropalveluarkkitehtuuriin siirtymisestä. Vaativuutensa takia on refaktorointihalukkuuden taustalla yleensä merkittäviä taloudellisia tai teknisiä tarpeita. Vaativuudestaan ja

kalleudestaan huolimatta refaktorointi tarjoaa myös suurimmat hyödyt.

3.2.5 Pitäytyminen

On myös mahdollista, että ohjelmistoa ei loppujen lopuksi ole kannattavaa siirtää pilveen [31]. Rahan ja resurssien lisäksi syynä voi olla esimerkiksi ohjelmiston elinkaaren lopun lähestyminen. Ohjelmistoa, jota ei tulla hetken kuluttua enää käyttämään ei luonnollisesti ole kannattavaa migroida pilveen, varsinkin jos siirto vaatii resursseja. Pitäytymisstrategiassa päädytään siis pitämään ohjelmisto sellaisenaan, ainakin toistaiseksi. On myös olemassa skenaarioita, joissa palvelun siirtämistä pilveen on syytä lykätä. Näistä suurin osa on taloudellisia, mutta on myös mahdollista, että jokin ohjelmiston käyttämä resurssi tai palvelu ei ole vielä saatavilla pilvipalveluissa, mutta sellaisen kehityksestä on jo tietoa. Esimerkkinä kieli- ja ajoympäristötuen puuttuminen. Tällaisissa tilanteissa päädytään Pitäytymään nykyisessä ratkaisussa koska pilvimigraatio tulisi tarvittavan refaktoroinnin vuoksi kalliiksi, mutta kustannukset ovat huomattavasti pienemmät, jahka jokin pilvipalveluntarjoajista julkaisee tuen ympäristölle. Tässäkin työssä päädytään Googlen Functions -palvelun kanssa tilanteeseen, jossa puuttuva .NET Core -tuki tarkoittaa, ettei testiohjelmistoa toteuteta lainkaan kyseiselle alustalle.

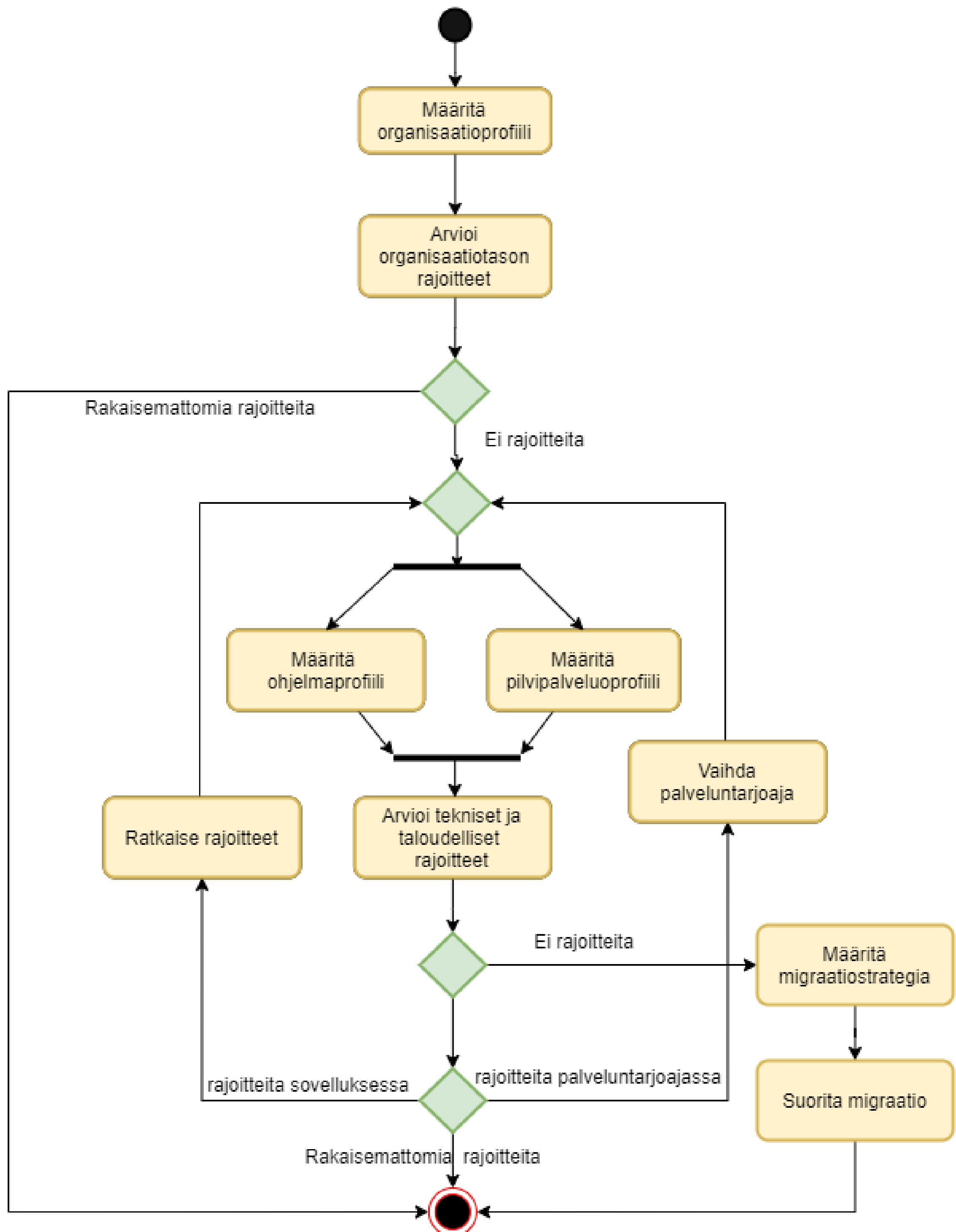
On myös mahdollista, että jotkin ulkoiset vaatimukset, kuten lainsäädäntö tai asiakkaan vaatimukset estävät varsinkin julkiseen pilveen siirtymisen. GDPR-säännökset esimerkiksi rajoittavat eurooppalaisten asiakkaiden tietojen säilömistä Euroopan ulkopuolella [18]. Tässä työssä käsiteltävistä alustoista jokainen tarjoaa kuitenkin mahdollisuuden valita resurssien, kuten levytilan, sijaintipaikan. Muut vastaavat säännökset voidaan kuitenkin nähdä rajoittavana tekijänä, ja näin päädytään pitämään ohjelmisto dedikoiduilla palvelimilla. Myös asiakkaan vaatimukset saattavat liittyä palvelinten fyysiseen sijaintiin, tai asiakas voi jopa suoraan vaatia, että palveluita ajetaan dedikoidulla palvelimella. Tähän yleisin syy on tietoturva. Näissä tapauksissa palvelut ovat dedikoidulla palvelimella, jolla ei ole lainkaan yhteyttää julkiseen verkkoon. Näissä tapauksissa ei luonnollisesti voida käyttää julkisia pilvipalveluita, mutta erilaiset yksityisen pilven -ratkaisut tulisivat kysymykseen.

3.2.6 Luopuminen

Joskus sovelluksen elinkaaressa on tultu siihen vaiheeseen, ettei sitä kannata enää käyttää lainkaan. Viimeinen strategia eroaa uuden ostamisesta siinä, ettei sovelluksen tilalle kannata ostaa uutta. Tästä hyödytään monin tavoin, joista ilmeisimpänä palvelin- ja ylläpitokustannusten laskeminen. Lisäksi, jos sovellusta on edelleen kehitetty, tai se on muuten sitonut henkilöstöresursseja, saadaan ohjattua työntekijöitä tuottavampien projektien pariin. Myös tietoturva paranee, kun suojattava ala supistuu. Orbanin mukaan 10 prosenttia yhtiön IT-ratkaisuista voivat olla nykyään hyödyttömiä, ja ne voidaan täten poistaa käytöstä [31].

3.3 Cloudstep-päätöksentekomalli

Migraation päätöksentekoprosessin pohjana käytetään tässä diplomityössä Cloudstep-mallia [13]. Malli mahdollistaa migraatiostrategioiden vertailun askel askeleelta ja perustuu rajoitteiden määrittämiseen. Eri vaiheissa löydetyt rajoitteet on pystyttävä ratkaisemaan ennen seuraavaan vaiheeseen siirtymistä. Edellisissä vaiheissa löydetyt rajoitteet toimivat syötteenä seuraaville, jotta saataisiin kokonaisvaltainen käsitys eri tasojen vaatimuksista. Arviointi aloitetaan organisaatiotasolta. Organisaatiotasolla pyritään löytämään ne rajoitteet, jotka kohdistuvat migraatiostrategian valintaan organisaation sisältä. Tällaisia voivat olla vakiintuneet toimintatavat, säännöt ja olemassa olevat sopimukset ja lisenssit. Ohjelmistotasolla pyritään löytämään rajoitteita migroitavasta ohjelmistosta itsestään. Ohjelmistossa tai sen käyttötavoissa voi olla erilaisia rajoitteita, jotka estävät siirtymisen pilveen joko kokonaan tai osittain. Palveluntarjoajatasolla kartoitetaan rajoitteita pilvipalvelusta itsestään. Rajoitteita voivat muodostaa esimerkiksi tekniset ratkaisut, huoltovarmuustasot tai palvelutarjonta. Vuokaaviossa 3.1 kuvataan mallin toiminta. Kappaleessa 6 muodostetaan Cloudstep-mallin pohjalta päätöksentekomalli pilvimigraatioon ja kuvaillaan mallin toiminta tässä yhteydessä tarkemmin.



Kuva 3.1. *Cloudstep-mallin vuokaavio.*
[13].

3.4 Ohjelmiston refaktorointi

Refaktoroinnilla tarkoitetaan ohjelmiston tai sen osan uudelleenkirjoittamista, vaikuttamatta sen toimintaan [36]. Refaktoroinnille voi olla monia syitä. Ohjelmistossa käytetyt ratkaisut ovat voineet vanhentua, ohjelmiston luettavuutta halutaan parantaa, tai ohjelmiston käyttämien kirjastojen muutokset vaativat muutoksia myös itse ohjelmistoon. Näiden esimerkkien lisäksi syitä on lukuisia, mutta refaktoroinnista puhuttaessa on hyvä erottaa se bugikorjauksista tai ohjelmiston jatkokehityksestä. Refaktoroinnin selkeänä tunnusmerkkinä voidaan pitää sitä, että toiminnallisuus ei varsinaisesti muutu, vaikka koodiin tehdään laajojakin muutoksia. Tämän työn puitteissa refaktoroinnilla tarkoitetaan lähes poikkeuksetta muutoksia, joita ohjelmistoon on tehtävä pilvimigraation yhteydessä.

Yleisimmin refaktorointi tapahtuu kooditasolla, eikä se tällöin vaikuta ohjelmiston rakenteeseen tai arkkitehtuuriin. Pilveen siirryttäessä kooditason muutoksia ovat esimerkiksi rajapintaluokkien muutokset tai tietokantayhteyttä käsittelevän kirjaston vaihtaminen toiseksi ja sen mukanaan tuomat muutokset. Kooditason muutosten taustalla voi myös olla ohjelmointikehyksen päivitys, jolloin refaktoroimaan joudutaan pääasiassa siksi, että kehyksen omat rajapinnat, samoin kuin mahdollisten kolmannen osapuolen kirjastojen rajapinnat ovat muuttuneet versioiden välissä, eikä käytettyjä ominaisuuksia enää tueta.

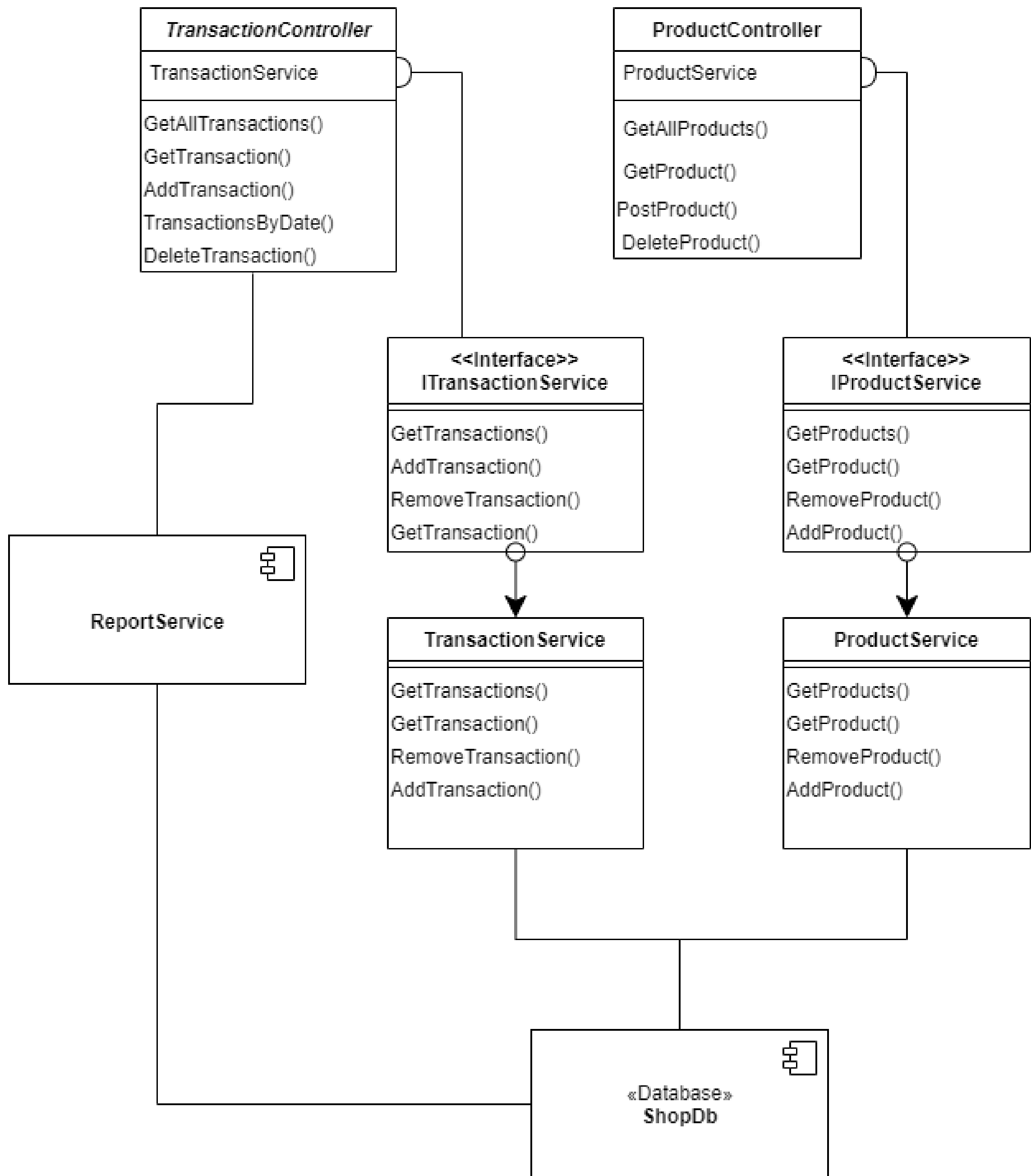
Arkkitehtuuritason refaktoroinnissa muutetaan ohjelmiston rakennetta. Saatetaan vaihtaa jopa kokonaan toiseen arkkitehtuuriin; pilvimigraatiossa yleensä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin. Arkkitehtuuritason refaktorointi ei vielä ole kovin yleistä, mutta legacy-ohjelmistot yleensä vaativat arkkitehtuuritason muutoksia, jotta pilvipalveluita saadaan hyödynnettyä [57]. Syitä vähäiseen refaktorointihalukkuuteen on monia. Arkkitehtuuritason refaktorointi voi olla aikaa vievää ja kallista, varsinkin jos on edes jossain määrin mahdollista käyttää ohjelmistoa sellaisenaan. Arkkitehtuurin refaktoroiminen on myös haastavampaa ja vaikeammin käsitettävää, kuin suhteellisen virtaviivainen kooditason refaktorointi.

4 .NET-MIGRAATIO ERI ALUSTOILLA

4.1 Testisovellus ja -ympäristö

Erilaisten migraatioratkaisujen ja pilvialustojen vertailuun käytetään tätä työtä varten kehitettyä yksinkertaista ASP.NET-ohjelmistoa. ASP.NET on .NET-ohjelmistokehyksen verkkosovellusten kehittämiseen tarkoitettu alusta [49]. Ohjelmisto on kuvitteellisen kaupan kassasovellus, joka muodostuu kahdesta eri sovelluksesta: Rajapintasovelluksesta ja raportointisovelluksesta. Rajapinnan kautta voidaan lisätä tehtyjä ostoksia ja hakea näitä. Raportointisovellus lukee kassasovelluksesta web-rajapinnan kautta tätä ostosdataa, josta muodostetaan raportteja. Ohjelmiston tietokantapalvelimena toimi samalla koneella ajettava Microsoft SQL Server 2018 -palvelin, jossa on kauppaohjelmistolle oma tietokantansa. Tietokannassa on omat taulunsa niin kauppasovelluksessa käytettäville entiteeteille, kuin raporttisovelluksen vastaaville. Käytössä oleva tietokantapalvelin on valittu siksi sen suuren käyttömäärän vuoksi [45]. MSSQL on suosittu ennen kaikkea .NET-sovelluksia kehitettäessä. Suosio selittyy vahvalla integraatiolla .NET-kehitystyökaluihin ja -ympäristöihin. Tietokantaluokkien kuvaaminen ja tietokannan migraatiot toteutettiin Entity Framework -kehyksellä, joka mahdollistaa tietokantaskeeman luomisen ja päivittämiseen tietokantaluokkien pohjalta [47].

Kaupan rajapintasovellus on toteutettu ASP.NET-rajapintasovelluksena, jonka rakenne on esitelty kuvassa 4.1. Alkuperäinen ohjelmisto, samoin kuin kaikki muutkin tämän työn sovellukset, toteuttiin C#-ohjelmointikielellä ja Visual Studio -ohjelmointiympäristöä. TransactionControllerin luomaa rajapintaa käytetään kauppatapahtumien muokkaamiseen. Tämän lisäksi raporttisovellus noutaa transaktiotiedot tästä rajapinnasta. Product-rajapintaa käytetään vain sisäisesti, raporttisovellus hakee tietonsa vain Transaction-rajapinnasta. Molemmilla rajapinnoilla on omat tietokantapalvelunsa, jotka suorittavat tietokantaoperaatiot. Varsinainen toimintalogiikka on näin eriytetty rajapinnasta, joka ainoastaan välittää pyynnöt asianmukaisille funktioille tietokantapalvelussa. Todellisessa sovelluksessa varsinainen ohjelmalogiikka olisi todennäköisesti toteutettu omaan luokkaansa, mutta sovelluksen yksinkertaisuuden takia käytetään tietokantaluokkaa myös jossain määrin ohjelmalogiikan suorittamiseen.



Kuva 4.1. *Luokkakaavio sovelluksen rakenteesta.*

Raporttisovellus on toteutettu Windowsin konsolisovelluksena, ja se on suunniteltu ajettavaksi Windows-palveluna. Sovellus hakee päivittäin toistaiseksi raportoimattomat transaktiot ja tuottaa niistä päiväkohtaiset raportit. Päivittäinen ajastus on toteutettu kolmannen osapuolen kirjastolla. Ajastuksen voisi toteuttaa myös käyttöjärjestelmän omilla työkaluilla, jolloin se olisi todennäköisesti tehokkaampaa ja veisi vähemmän resursseja, sillä nykyisessä ratkaisussa ohjelmaa suoritetaan jatkuvasti, vaikka raportteja luodaan vain kerran päivässä. Ulkoisen kirjaston käyttöön päädyttiin osittain siksi, että voidaan arvioida kolmannen osapuolen kirjaston vaikutusta päivitettäessä uudempaan ohjelmistokehyksen versioon.

Esimerkkisovellus vastaa yksinkertaisuudestaan huolimatta arkkitehtuuriltaan, rakenteel-

```
1  [HttpPost]
2  [Route("bydate")]
3  public IHttpActionResult TransactionsByDate
4  (TransactionsByDateRequest request)
5  {
6    try
7    {
8      var result = _transactionService.
9      GetTransactions(request.StartDate, request.EndDate);
10     return Ok(result);
11   }
12   catch (Exception ex)
13   {
14     return InternalServerError(ex);
15   }
16 }
```

Ohjelma 4.1. *Transaktioiden haku alkuperäisessä sovelluksessa.*

taan ja kirjastovalinnoiltaan melko hyvin vanhempia .NET-sovelluksia. Molempiin sovelluksiin olisi voitu vielä toteuttaa erillinen tietokantaluokka ja muutenkin monimutkaistaa rakennetta, jotta ne vastaisivat paremmin oikeita tuotantosovelluksia, mutta kyseisillä muutoksilla ei olisi vertailun kannalta suurta merkitystä. Ohjelmiston jakaminen kahteen sovellukseen, joista toinen suorittaa ajastettuja tehtäviä Windows-palveluna vastaa myös tuotantosovelluksille yleistä tapaa. Tällä on testin kannalta merkitystä senkin vuoksi, että monet pilvipalvelut tarjoavat juuri yksinkertaisille ajastettaville sovelluksille useita joustavampia ratkaisuja.

4.2 Testin kuvaus

Vertailun tarkoituksena on migroida aiemmin kuvattu ohjelmisto kolmelle eri pilvipalvelualustalle kahdelle eri pilvipalvelutyypille. Alustoiksi valittiin Microsoft Azure, Amazon AWS ja Google Cloud Platform. Palvelutyypeistä vertailuun otettiin pilvisovellusalustat ja serverless-palvelut. Virtuaalipalvelimet ja säiliöt jätettiin käytännön vertailun ulkopuolelle, sillä koettiin, ettei niiden välillä olisi merkittäviä eroja. Virtuaalipalvelimen tapauksessa nykyinen ohjelmisto voidaan ajoympäristöineen siirtää sellaisenaan Windows-pohjaiselle virtuaalipalvelimelle. Säiliöillä tilanne on samankaltainen; jokainen alustoista tarjoaa tuen Windows-pohjaisille säiliöille vähintään Kubernetes-pohjaisen ratkaisun [15][56][38], joka tekee migraatiosta varsin suoraviivaisen. Kontit ovat tällä hetkellä käytetyin teknologia rakennettaessa mikropalveluarkkitehtuurin mukaisia, ja siten pilvinatiiveja järjestelmiä, mutta yksi konttien vahvuuksista on juuri alustariippumattomuus, minkä takia vertailussa ei todennäköisesit saataisi kummoisiakaan eroja. Alustojen ja palvelutyyppien lisäksi vertailussa tulee esiin kaksi migraatiostrategiaa: alustanvaihto ja refaktorointi. Alustanvaihdosta on kyse niiden toteutusten kohdalla, joihin ei tehdä varsinaisia muutoksia koodiin tai arkkitehtuuriin. Refaktoroinnista on puolestaan kyse toteutuksissa, joissa muutetaan

ohjelmiston arkkitehtuuria tai rakennetta.

Lähtökohtana oli siirtää ohjelmisto kullekin palvelutyypille mahdollisimman vähillä muokkauksilla, jotta saataisiin kartoitettua kunkin alustan ja pilvipalvelutyypin vaatimaa työmäärää. Tarvittaessa ohjelmistoa kuitenkin muutettiin niin ohjelmistokehyksen päivityksellä, kuin kevyellä arkkitehtuurimuutoksella. Molemmat sovellukset pyrittiin toteuttamaan kussakin vaiheessa niin ikään samalle palvelutyypille, mutta tarvittaessa käytettiin raporttisovellukselle erillistä palvelutyyppiä, jos .NET-toteutusta ei tuettu. Tietokanta siirrettiin myös pilvipalveluihin. Jokaisella pilvipalvelualustalla käytettiin samaa alustan tarjoaamaa tietokantapalvelua molempien palvelutyyppien vertaamiseen. Migraation vaatimat resurssit luotiin ja konfiguroitiin käyttäen web-käyttöliitymää. Jokainen palvelu tarjoaisi myös kattavat komentorivityökalut, mutta yksittäisiä resursseja luodessa pääsee lähtökohtaisesti helpomalla graafista käyttöliittymää käyttämällä. Isompien kokonanaisuuksien automatisointia varten olisi kuitenkin luontevinta käyttää komentorivityökaluja. Ohjelmistojen julkaisu hoidettiin jokaiselle alustalla Visual Studiosta käsin, käyttäen palveluntarjoajien omia liitännäisiä. Julkaisuun olisi alustasta riippuen lukemattomia muitakin vaihtoehtoja, kuten suoraan versionhallinnasta muutosten mukana julkaiseminen, mutta tässä työssä pitäydyttiin yksinkertaisimmassa vaihtoehdossa.

4.3 Pilvisovellusalusta

Pilvisovellusalustalle tehdyissä toteutuksissa pystyttiin pääasiassa hyödyntämään alkuperäistä .NET-sovellusta. Vain GCP:n tapauksessa jouduttiin päivittämään ohjelmistokehys Core-versioon. Tämän ansiosta migraatioprosessi oli varsin suoraviivainen. Suuria eroja itse migraatiossa ja sen työkaluissa ei myöskään huomattu, vaan jokaisen tarjoajan omat liitännäiset olivat helppokäyttöisiä eikä ongelmia esiintynyt. Tässä vaiheessa vertailua asennettiin myös tietokanta jokaiselle alustalle. Kaikilla alustoilla oli tarjolla Microsoft SQL Serverin jokin versio, joten tietokantamigraatio onnistui helposti. GCP tarjoaa kuitenkin vain SQL Server 2017:n, jonka vuoksi tietokantaskeemaa ei saatu tuotua suoraan, vaan jouduttiin käyttämään hiukan manuaalisempaa lähestymistapaa ja erillisten SQL-skriptien muodossa. Skeeman kääntäminen vanhemmalle versiolle sopivaksi on mahdollista myös erillisillä työkaluilla, joten suurta ongelmaa tästäkään ei pitäisi syntyä. Legacy-ohjelmistojen tapauksessa on muutenkin hyvin mahdollista, että päinvastoin kuin vertailun tapauksessa käytössä oleva versio on vanhempi, kuin mitä pilvipalvelut tarjoavat, joten skeeman muuntaminen uuteen versioon edessä joka tapauksessa. Lähtökohtaisesti tietokannat tarjoavat hyvän taaksepäin yhteensopivuuden, joten on mahdollista, ettei skeemaa tarvitse muokata.

4.3.1 Azure App Service

Azuren tarjonnasta rajapintasovellukselle luontevin valinta oli App Service -palvelu, joka on tarkoitettu verkkosovellusten toteuttamiseen valmiiksi tarjotulla alustalla [7]. App Service tukee ASP.NET-ajoympäristön versiota 4.7. Testiohjelmisto on toteutettu versiolla

4.5, joten ongelmia ei lähtökohtaisesti pitäisi ilmetä, sillä versiosta 4.5. eteenpäin .NET-ajoympäristöt ovat taaksepäin yhteensopivia, ainakin saman pääversion sisällä. Lisäksi tarjolla on tuki myös versiolle 3.5, joten vanhemmatkin ohjelmistot on mahdollista saada toimimaan alustalla suoraan ilman versiopäivitystä.[20] Vanhasta versiosta voi kuitenkin muodostua ongelmia viimeistään riippuvuuksien, kuten kolmannen osapuolen kirjastojen kautta. Web App -resurssin luominen suoritettiin Azure Portalin graafisella käyttöliittymällä, eikä tässä kohtaa kohdattu ongelmia. Sovelluksen tietokannaksi valittiin Azure SQL -palvelin, jonne tietokantaskeema luotiin SQL-skriptien avulla. Vaihtoehtona olisi myös Entity Frameworkin migraatioiden käyttäminen, mutta koska vastaavia migraatioita ei tueta muilla alustoilla yhtä suoraan, käytettiin tälläkin alustalla SQL-kyselyä. Useita muitakin tietokantoja olisi valittavissa, mutta Azure SQL on pohjimmiltaan Microsoft SQL -palvelin, ja siten lähimpänä testisovelluksen alkuperäistä toteutusta.

App Serviceen julkaiseminen onnistui suoraan Visual Studion työkaluilla, joilla saa suoraan yhteyden Azureen ja saa valittua mihin perustettuun resurssiin sovellus asennetaan. Jopa tietokannan tarvitsemat yhteysasetukset saa määritettyä samalla kertaa. Niin tietokannan, kuin muutkin tarvittavat asetukset saa myös määritetty konfiguraatiotiedostossa. Vaihtoehtona IDE:n integroiduille työkaluille olisi komentorivityökalut, versionhallinnasta kloonaaminen tai FTP-siirto suoraan palvelimelle. Raporttisovellus siirrettiin pilveen Azure WebJob-palveluna, joka mahdollistaa skriptien tai ajettavien tiedostojen ajamisen joko ajastetusti tai jatkuvaan ajoon [21]. Ajastaminen olisi ollut hyvä vaihtoehto raporttisovelluksen tapauksessa, mutta koska ajastus on toteutettu ohjelmalogiikassa, ei sitä kannattanut purkaa sovelluksesta ja jättää Azuren vastuulle, mutta alusta asti pilveä varten sovellusta suunnitellessa kannattaa vaihtoehto pitää mielessä.

4.3.2 AWS Elastic Beanstalk

AWS:n tarjonnassa pilvisovellusalusta tunnetaan nimellä Elastic Beanstalk. Elastic Beanstalk mahdollistaa sovellusten julkaisemisen pilveen, ilman, että tarvitsee tuntea alla olevaa infrastruktuuria [50]. Elastic Beanstalk vastaa toiminnaltaan hyvin paljon aiemmin käytettyä Azuren vastaavaa App Service -palvelua. Web-sovelluksen asentaminen Elastic Beanstalkiin on mahdollista tehdä hyvin samaan tapaan, kuin Azuren App Serviceen. AWS:n toteuttamat liitännäiset tarjoavat kaiken tarvittavan, jotta sovellusympäristön luonti ja sovelluksen julkaiseminen onnistuvat Visual Studiosta käsin. Paljon muitakin toimintatapoja on mahdollista valita, kuten suoraan versionhallinnasta tai osana erillistä jatkuvan integraation ja julkaisun ratkaisua. Elastic Beanstalkin tuki .NET-versioille oli odotettua laajempi, sillä Elastic Beanstalkin pohjalla pyörivillä EC2-virtuaalipalvelininstansseilla ajetaan versiota 4.8, jonka luvataan tukevan versioita 4, 2.0 ja 1.[3] Itse migraatioprosessi oli suoraviivainen, vaivaton ja intuitiivinen. Teknisiä ongelmia ei tullut vastaan, vaan ohjelmisto saatiin toimimaan ongelmitta. Tietokannaksi AWS-toteutuksille valittiin RDS for SQL Server -palvelu, jolla on mahdollista luoda SQL Server 2019 instanssi [30], jonka ansiosta tietokantaskeeman migroiminenkin onnistui vaivattomasti.

```
1 [HttpPost]
2 [Route("byDate")]
3 public IActionResult TransactionsByDate
4 (TransactionsByDateRequest request)
5 {
6   var result = _transactionService.
7   GetTransactions(request.StartDate, request.EndDate);
8   return Ok(result);
9 }
```

Ohjelma 4.2. *Transaktoiden haku .NET Core -totetuksessa.*

4.3.3 Google App Engine

Google Cloud Platformin pilvisovellusalustana toimii Google App Engine. App Engine muistuttaa paljon aiemmin esiteltyjä palveluita ja mahdollistaa sovellusten suorittamisen suoraan palvelussa [23]. Google App Engine tukee .NET-ohjelmistokehyksestä ainoastaan Core-versiota. Testiohjelmisto oli siis päivitettävä, jotta vertailua voitiin jatkaa. Tämänkaltainen päivitysprosessi voi todellisen ohjelmiston tapauksessa olla työläs ja aikaa vaativa projekti, mutta pienen testiohjelmiston tapauksessa refaktorointi oli vaivaton. Laajankin ohjelmiston päivittäminen käyttämään Corea voi olla yksinkertaista, ja tähän on tarjolla myös paljon työkaluja. Suurimman ongelma muodostavat kolmannen osapuolen kirjastot, joita ei legacy-sovellusten tapauksessa ole välttämättä päivitetty tukemaan huomattavasti uudempaa .NET Corea. Coressa on kuitenkin myös paljon hyötyjä, kuten mahdollisuus ajaa sovelluksia myös Linux-ympäristössä. Varsinkin jos refaktorointi viedään välttämättömyyksiä pidemmälle, saadaan käyttöön monia Coren ominaisuuksia, joita vastaavia on .NET:n puolella saatavilla vain kolmannen osapuolen kirjastoista. Tästä hyvästä esimerkkinä riippuvuuksien injektointi ilman kolmannen osapuolen kirjastoja [37]. Esimerkki tällaisesta tapauksesta löytyi myös testiohjelmiston raporttisovelluksesta, sillä raporttien ajastamiseen käytetyn kirjaston Core-versiossa ajastusmetodit oli muutettu asynkronisiksi, joten koodia jouduttiin tältä osin refaktoroimaan. Vastaava refaktorointi olisi huomattavan haastavaa vastaavassa, mutta laajemmassa ohjelmistossa.

GCP ei tarjoa lainkaan suoraa mahdollisuutta ajaa .NET-sovelluksia, edes Corella toteutettuja, ajastettuina tehtävinä. Vastaava toiminallisuus olisi kuitenkin toteutettavissa usealla tavalla. Sovellusta varten voitaisiin pystyttää oma virtuaalikone, sovellus voidaan paketoida säiliöön tai se voidaan uudelleenkirjoittaa serverless-funktioksi. Kuten aiemmin on perusteltu, ei tässä työssä ole tarkoitus käsitellä suoraan virtuaalipalvelimille migroimista lainkaan. Serverless-toteutus käsiteltäisiin omassa kappaleessaan, mutta toteutuksen aikana huomattiin, ettei Google tarjoa vielä mahdollisuutta toteuttaa Cloud Functions-alustalla .NET-sovelluksia. GCP:n kohdalla testaaminen jää siis vain osittaiseksi, sillä rajapintasovelluksen testaamiseen käytettiin raporttipalveluna Azuren WebJob-toteutusta.

4.4 Serverless

Serverless-toteutuksia varten käytettiin pohjana GCP:n sovellusalustaa varten päivitettyä versiota ohjelmistosta. Tähänkin versioon täytyi tehdä alustakohtaisia muutoksia, pääasiassa funktiomääritelmien ja konfiguraatioiden muodossa. Muutokset olivat lähtökohtaisesti pieniä ja vaivattomia ja dokumentaatioista löytyi hyvät ohjeistukset, miten toimia. Raporttisovellus saatiin myös toteutettua Serverless-palveluna, jollaiseksi se luonnostaan sopiikin. Tietokantoina käytettiin aiemmassa vaiheessa luotuja ja alustettuja tietokantoja. Yleisesti Serverless-funktioiden julkaisu on yksinkertaista, sillä jahka funktiot on määritetty ohjelmakoodissa luodaan vastaavat funktiot automaattisesti pilvipalvelun puolella. Prosessiin on vain rajallisesti mahdollisuutta vaikuttaa itse, mutta toimiessaan voidaan Serverless:ia pitää tässä vertailussa helpompana julkaisutapana.

Google Cloud Functions ei kirjoittamishetkellä tarjoa lainkaan tukea C#:lle, eikä siten myöskään sovelluksellemme. Tästä syystä GCP:n toteutus jätettiin kokonaan tekemättä, sillä tämän työn puitteissa ei ole mielekästä uudelleenkirjoittaa testisovellusta esimerkiksi Javalla tai NodeJS:llä, joille tuki löytyy [14]. On hyvin mahdollista, että tuki tullaan tulevaisuudessa lisäämään, sillä .NET Core -pohjaisia sovelluksia tuetaan kuitenkin muissa palvelutyypeissä.

4.4.1 Azure Functions

Azure tarjoaa serverless-sovelluksia varten Azure Functions -palvelun, jossa on mahdollista suorittaa niin yksittäisi funktioita, kuin suurempia kokonaisuuksia [9]. Rajapintasovelluksen Serverless-versio toteutettiin Azurelle muuttamalla web-rajapinnan metodit Azure-funktioiksi. Pohjana käytettiin aiemmin GCP:n sovellusalustaversiota varten tehtyä Core-toteutusta. Käytännössä tähän tarvittiin vain pieniä muutoksia. Kuten ohjelmia 4.3 ja 4.2 vertaamalla nähdään, on rajapintametodiin tarvinnut lisätä virheenkäsittely samaan tapaan kuin alkuperäisessä toteutuksessa (Ohjelma 4.1). Metodiin on myös lisätty serverless-funktion määrittely, jonka perusteella Azure Functions luo tarvittavat funktiot. Muu logiikkaa voitiin jättää sellaisekseen ja muutokset kohdistuivat vain rajapintaan. Azure Functions tukisi myös .NET Frameworkin versiota 4.7 [22], joten todellisen ja varsinkin hyvin laajan sovelluksen tapauksessa voisi olla helpointa pitäytyä .NET:n perusversiossa. Pidemmän päälle ei kuitenkaan ole lainkaan mahdotonta, että vahempien ohjelmointikehysversioiden tuki loppuu jossain vaiheessa, ja päivitys Coreen on kuitenkin edessä.

Raportointisovellus toteutettiin Core-päivitetyn version pohjalta yksinkertaisin muutoksin. Raportointitoiminallisuus muutettiin Azure funktioksi, mutta sovelluslogiikkaa ei muuten jouduttu muuttamaan. Myös ajastuslogiikka purettiin funktiosta ja ajastaminen jätettiin Azuren vastuulle.

```
1  //Route-attribuutti korvattu Funktionimen määrittelyllä
2  [FunctionName("ByDate")]
3  public async Task<IActionResult> GetTransactionsByDate(
4      [HttpTrigger(AuthorizationLevel.Anonymous,
5      "post", Route = null)] HttpRequest req)
6  {
7      // Virheenkäsittely tehtävä itse, rajapinta ei hoida automaattisesti
8      try
9      {
10         //Http-pyyntö on käsiteltävä erikseen
11         string requestBody = await new StreamReader(req.Body)
12             .ReadToEndAsync();
13
14         var request = JsonConvert
15             .DeserializeObject<TransactionsByDateRequest>(requestBody);
16
17         var result = _transactionService
18             .GetTransactions(request.StartDate, request.EndDate);
19
20         return new OkObjectResult(result);
21     }
22     catch
23     {
24         return new InternalServerErrorResult();
25     }
26 }
```

Ohjelma 4.3. *Transaktioiden haku Azure Functions -sovelluksessa.*

4.4.2 AWS Lambda

AWS:lle ohjelmiston serverless-versio toteuttiin AWS Lambda-funktioina. Molempien sovellusten toteutukseen käytettiin AWS:n tarjoamia pohjia ja kehitystyökalujen liitännäisiä. Pohjien avulla voidaan luoda tarvittavat tiedostorakenteet ja ladata tarvittavat kirjastot. AWS Lambda tarjoaa mahdollisuuden käyttää .NET Core web-rajapintasovellusta käytännössä sellaisenaan, kunhan projektissa käytetään AWS:n tarjoamat työkaluja. Näin voidaan esitellään Lambda-funktio, jolla kutsutaan Coren Startup-funktiota (Ohjelma 4.4). Startup-funktio vastaa Core-sovelluksissa sovelluksen ja sen ympäristön alustamisesta, joten se tarjoaa pääsyn reflektion avulla tarjolla oleviin rajapintafunktioihin ja luo niille reitityksen [1]. Reititys ja muut vastaavat asetukset ovat myös käyttäjän muokattavissa konfiguraatiotiedostojen avulla.

Rajapintasovellus toteutettiin käyttäen edellä kuvattua kauttakulkufunktiota. Tämä vähensi huomattavasti uudelleenkirjoitustarvetta, sillä koko sovelluslogiikka pysyi sellaisenaan. Pohjana käytettiin Google App Engineä varten toteutettua Core-sovellusta, sillä AWS Lambda ei tue alkuperäistä .NET:ia. Refaktorointia ei tarvinnut tässä tapauksessa tehdä lainkaan, mutta konfiguraatiotiedostoja piti muokata, jotta reititys ja tietokantayhteydet

```
1  // LambdaSerializer tulee ottaa käyttöön
2  [assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization
3  .SystemTextJson.DefaultLambdaJsonSerializer))]
4
5  namespace WebAPI
6  {
7      // Entrypoint periytetään asiaankuuluvasta Gateway-luokasta
8      public class LambdaEntryPoint : Amazon.Lambda
9      .AspNetCoreServer.APIGatewayProxyFunction
10     {
11         protected override void Init(IWebHostBuilder builder)
12         {
13             builder
14                 .UseContentRoot(Directory.GetCurrentDirectory())
15                 //.NET Coren Startup-funktio
16                 .UseStartup<Startup>()
17                 .UseLambdaServer();
18         }
19     }
20 }
```

Ohjelma 4.4. *LambdaEntrypointin määrittely.*

toimisivat. Ohjelmassa 4.4 esitellään LambdaEntrypoint-luokka, jonka avulla AWS Serverless Application saa käyttöönsä Web-rajapinnan. UseStartup-komento kutsuu Coren Startup-luokkaa, joka vuorostaan huolehtii ohjelman alustamisesta.

Raporttisovelluksen toteutuksen pohjaksi otettiin niin ikään GCP:tä varten toteutettu Core-versio. Nyt kuitenkin muokattiin sovellusta siten, että raportointifunktio toteutettiin Lambda-funktiona, joka käyttää palveluluokkaa tietokantojen käsittelyyn, kuten ennenkin. Myös ajastus purettiin itse sovelluslogiikasta, ja siirrettiin AWS:n puolelle erillisen EventBridge-resurssin ajettavaksi. EventBridge mahdollistaa Lambda-funktioiden ajastuksen tapahtumapohjaisesti tai ajastetusti [6]. Tässä tapauksessa luotiin päivittäinen ajastus raportointia varten.

5 MIGRAATIOVERTAILUN TULOKSET

Vertailun tuloksien arvioinnissa käytetään kriteereinä käytön helppoutta, ohjeiden selkeyttä ja työkalujen yleistä toimivuutta. Vertailulle on kuitenkin vaikea muodostaa sen luonteen vuoksi tarkempaa ja teknisempää kriteeristöä. Ensinnäkin suorituskykyvertailu on jo lähtökohtaisesti haastava, sillä palvelutaso ja valitut resurssit vaikuttavat paljon, eikä eri palveluntarjoajilla ole välttämättä keskenään suoraan verrattavissa olevia palveluja. Tietyt migraatiotavat eivät myöskään toimineet kaikilla alustoilla suoraan, tai ollenkaan, joten tätä tulosta voidaan käyttää kertomaan, mitkä alustat eivät ole käytänöllisiä vaihtoehtoja tai vaativat vähintäänkin enemmän työtä.

Strategioiden osalta AWS:n ja Azuren pilvisosovellusalustojen toteutukset edustivat alustanvaihtoa, kun taas GCP:n sovellusalustan toteutus vastaa refaktorointia ohjelmistokehysversion päivityksen vuoksi. Kaikki tehdyt serverless-toteutukset edustavat refaktorointistrategiaa, sillä ohjelmistoon joudutaan tekemään muutoksia, jotta niitä saadan suoritettua valituilla alustoilla. GCP:n serverless-toteutus jouduttiin jättämään tekemättä, joten sen voidaan tulkita sopivan pitäytymis- tai luopumisstrategiaan. Osasta serverless-toteutuksia voitasiin testisovelluksen mittakaavassa puhua myös alustanvaihtona, sillä muutokset ovat lopulta vähäisiä. Vastaavien täysmittaisten ohjelmistojen toteutus vaatisi kuitenkin todennäköisesti huomattavasti enemmän työtä, joten voidaan puhua refaktoroinnista.

5.1 Pilvisovellusalusta

Rajapintasovelluksen tapauksessa Azure ja AWS erottuvat edukseen, varsinkin jos ohjelmointikehyksen versiota ei olla valmiita päivittämään. Molempien tuki vanhemmille versioille pitää ylimääräisen työn varsin vähäisenä. Molemmat tarjoavat myös yhteensopivat tietokantaratkaisut esimerkkisovelluksen kaltaiselle sovellukselle. GCP:n kanssa joudutaan vähintäänkin päivittämään Coreen. Operaation raskaus riippuu paljon käytetystä arkkitehtuurista. Jos varsinainen sovelluslogiikka, rajapinta ja tietokantaa käsittelevä kerros on hyvin eriytetty toisistaan selvitään päivityksessä mahdollisesti hyvinkin helpolla. Suurimman riskin päivityksessä muodostavat oikeastaan kolmannen osapuolen kirjastot, joille ei välttämättä ole toteutettu Core-versioita. Myöskin, vaikka versio olisi toteutettu, on rajapintoja, tai jopa logiikkaa voitu muuttaa siten, että päivityksessä täytyy olla tarkkana. Tämänkaltainen esimerkki kohdattiin esimerkkisovelluksen ajastus-kirjastoa päivitettäessä, kun ainoastaan asynkroninen versio eräästä ohjelmassa käytetystä metodista

oli enää tarjolla. Näistäkin selvitään uudelleenkirjoittamalla kutsuja hiukan, mutta suuren projektin tapauksessa työn määrä kertautuu nopeasti, ellei onnistuta automatisoimaan joitakin operaatoita.

Raportointisovelluksen tapauksessa ainoastaan Azure tarjoaa kaiken valmiina. Mahdollisuus siirtää ohjelma sellaisenaan ajettavaksi WebJobina tarkoittaa monien vastaavien konsolisovellusten kanssa erittäin vähäistä lisätyötä, tämäkin lähinnä yhteyskonfiguraation osalta, jos tietokantakin siirretään samalla pilveen. AWS- ja GCP-alustoilla helpoin tapa saada sovellus sellaisenaan ajoon olisi Windows-virtuaalipalvelimen luominen, jonne voitaisiin asettaa sovellus ajoon palveluna, samaan tapaan kuin sovellusta on alunperin tarkoitettukin. Kokonaisen virtuaalipalvelimen luominen yksittäistä ajettavaa palvelua varten on kuitenkin jokseenkin raskas (ja mahdollisesti kallis) vaihtoehto. Varsinkin jos osaamista löytyy säiliöpohjaisten ratkaisujen parista, kannattaa säiliöitä harkita vähemmän raskaana ja mahdollisesti edullisempana vaihtoehtona. Jos sovelluksen uudelleenkirjoittaminen on vaihtoehto, tarjoavat molemmat alustat kuitenkin kelvollisia tapoja asettaa esimerkiksi JavaScript-pohjaisia sovelluksia ajoon ajastetusti.

Azurea ja AWS:a voi molempia suositella .NET-sovellusten pilvimigraatioon. Suurimmaksi eroksi näiden kahden välille jää juurikin AWS:n puuttuva tuki Windows-palvelun kaltaiselle toiminnolle. Mutta jos tämä on mahdollista toteuttaa edellämainituin keinoin, ei varsinaisa eroja jää. Molempien työkalut tekevät migraatiosta helppoa, toteutetaan se sitten IDE:n työkaluille tai komentoriviltä käsin. Valinnassa näiden kahden välillä korostuu siten aikaisempi kokemus. Jos toisesta on jo kokemusta, on luontevampaa pitäytyä sammassa, kuin hajauttaa järjestelmiä useiden palveluntarjoajien välille. Google Cloud Platformia voi suositella migraatiota varten vain, jos ollaan valmiita refaktroimaan sovellusta sopivaksi. Jos tähän kuitenkin ollaan valmiita, esimerkiksi jos sovelluksen päivitys olisi muutenkin ajankohtaista, tarjoaa Googlen alusta myös kilpailukykyisen vaihtoehdon. Googlen alusta on myös selkein, mahdollisesti osittain siksi, että tarjonta on rajallisempaa kuin kilpailijoilla.

5.2 Serverless

Serverless-sovelluksena toteutetussa rajapintasovelluksessa AWS:n tarjoama mahdollisuus käyttää .NET Core -verkkorajapintasovellusta käytännössä sellaisenaan on erittäin mielenkiintoinen ominaisuus. Koska itse sovelluslogiikka ei muutu, voidaan käyttää vanhaa rajapintaa sellaisenaan, joten uusien yllättävien virheiden mahdollisuus pysyy pienenä. Huolenaiheena nousee lähinnä mahdolliset suorituskykyongelmat, kun yhden Lambda-funktion kautta kutsutaan kaikkia verkkorajapinnan metodeja. Tämän työn puitteissa suorituskykyä ei kuitenkaan kyetty kunnolla testaamaan. Azuren tapauksessa serverless-arkkitehtuuriin siirtymnen on lähes yhtä yksinkertaista. Suurin työ on oikeastaan rajapintametodien muuttamisessa funktio-muotoon. Tällä tarkoitetaan vanhojen reititysten muuttamista Azure-funktioesittelyiksi ja joidenkin verkkorajapinnalle spesifien ominaisuuksien korvaamista funktioiden vastaavilla. Tämän lisäksi oikeastaan vain yhteys-

konfiguraatiot muuttuvat. Raportointisovelluksen toiminallisuuden perustuessa oikestaan vain yhteen funktioon ja sen tarvitsemaan tietokantaan ja tietokantaoliot kuvaaviin luokkiin, oli sen uudelleenkirjoittaminen serverless-funktioksi molemmilla kielillä varsin yksinkertaista. Itse funktio ei muuttunut kummassakaan, paitsi ajastuslogiikan poistamisen verran.

Migraatio serverless-ympäristöön on teknisesti hyvin samankaltaista sekä Azurella, että AWS:lla. Molemmat tarjoavat hyvinkin nopean tavan migroida web-rajapintasovellus sellaisenaan, mutta AWS:n tapauksessa sovelluksen on oltava Core-pohjainen. Myös Azuren tapauksessa on silti suositeltavaa päivittää Coreen, sillä päivitys on hyvin todennäköisesti tehtävä ennen pitkää. Core-pohjaisena molemmat tarjoavat pätevän tavan käyttää rajapintasovellusta lähes sellaisenaan. AWS:n kauttakulkufunktioon perustuva ratkaisu on varsin tehokas, ja vähentää uudelleenkirjoittamistarvetta huomattavasti. Azuren tapa, jossa joudutaan muokkaamaan rajapintaa hiukan enemmän on myös todella intuitiivinen ja tuntuu vähemmän oikaisevalta vaihtoehdolta.

6 MIGRAATION PÄÄTÖKSENTEKOPROSESSI

Tässä luvussa kuvataan päätöksentekoprosessi, jonka tarkoitus on auttaa pilvimigraatiostrategian valinnassa. Pohjana käytetään Cloudstep-mallia [13]. Malli toimii tämän luvun runkona, joten siihen ei lähtökohtaisesti viitata, ellei se ole selvyyden vuoksi tarpeellista. Malli yhdistetään aiemmin esiteltyihin kuuteen strategiatyyppiin ja käytännön vertailusta saatuihin tuloksiin, jotta saataisiin muodostettua malli sopivan migraatiostrategian määrittämiseksi. Cloudstepista mukailtua päätöksentekomallia soveltaen pyritään löytämään tarpeisiin vastaava strategiatyyppi ja näitä molempia tukeva pilvipalveluntarjoaja.

6.1 Organisaatioprofiilin määrittäminen

Ensimmäisenä vaiheena on organisaatioprofiilin määrittäminen. Tällä tarkoitetaan organisaation nykytilan selvittämistä pilvimigraatioon liittyen. Selvitetään mitkä motiivit ovat pilvimigraation takana ja mitä hyötyjä migraatiolla on saavutettavissa. On tärkeää, että migraatiossa otetaan huomioon myös organisaatiotaso, ja nimenomaan alusta asti. Näin varmistetaan, että myöhemmin tehtäville päätöksille on pohja myös organisaatiotasolla ja sen liiketoiminnassa. Varsinkin suurempien migraatiokokonaisuuksien tapauksessa merkitys moninkertaistuu. Organisaatiotason huomioon ottamisella on merkitystä, varsinkin isompien organisaatioiden kohdalla, myös yhtenäisyyden vuoksi. On mahdollista, että useampi eri osasto harkitsee pilvimigraatiota lähes samanaikaisesti. Tällöin on tärkeää, että päätöksentekoon otetaan mukaan kaikki migraatioita harkitsevat. Jo aiemmin tehdyt organisaatioprofiilin määritykset ovat myös tärkeä saada tietoon, sillä on mahdollista, että niissä on tehty eri tulkintoja, kuin mihin nyt päästäisiin. Näiden väliltä on löydettävä konsensus, ennen kuin prosessia voidaan turvallisesti jatkaa.

Rajoitusten lisäksi on hyvä kartoittaa myös organisaatiotason tarpeet. Päätös pilveen siirtymisestä on harvoin puhtaasti tekninen. Taustalla on monesti liiketoimintatason ajureita, kuten kustannusten pieneneminen tai parempi saatavuus. Maantieteellinen skaalautuminen, itsekorjaavuus ja varmuus ovat pohjimmaltaan teknisiä ominaisuuksia, mutta niillä on myös selkeät liiketoiminnalliset puolet. Tämäntapaiset tarpeet ohjaavat omalta osaltaan valintaa, varsinkin alustan ja sovellustyypin osalta. Esimerkiksi halu laajentua nopeasti kansainvälisesti voi olla hyvin painava syy siirtyä pilvipalveluihin. Tällöin hyvä skaalautuvuus, sekä kuorman osalta, että maantieteellisesti on paljon painavampi syy, kuin esimerkiksi mahdolliset kustannussäästöt tai ylläpidon ja sovelluskehityksen suoraviivaistaminen.

Käytännössä organisaatioprofiilin kannalta kaikki tässä työssä esitellyt alustat ovat päteviä. Jokainen tarjoaa samankaltaiset palvelut ja erilaisten organisaation rajoitteiden huomioon ottaminen on mahdollista. Amazon ja Azure ovat mahdollisesti hieman edellä laajemman tarjontansa suhteen, mutta lähtökohtaisesti Google on lähes yhtä hyvä valinta. Jos organisaatiossa on jo siirretty sovelluksia pilveen, tai toteutettu näitä alusta asti, on luultavasti hyödyllistä käyttää samaa alustaa, kuin jo käytetään. Koska pilvipalveluita tarjoavat yritykset toimivat muillakin ohjelmistokehityksen osa-alueilla on mahdollista, että organisaatio on jossain määrin sitoutunut jo johonkin toimittajaan. Tämä sitoutuminen on harvoin estävää, mutta erilaiset kumppanuusohjelmat toimittajien välillä voivat rohkaista valitsemaan tietyn palveluntarjoajan muiden yli, vaikka teknisesti parempia vaihtoehtoja olisi tarjolla. Kumppanuusohjelmat voivat rohkaista pitäytymään samassa toimittajassa muun muassa halvemman hinnoittelun tai laajemman tuen avulla.

6.2 Organisaatiotason rajoitteiden määrittely

Organisaatioprofiilin määrittämisen jälkeen voidaan määrittää rajoitteet, joita organisaatio asettaa pilvimigraatioprosessille. Organisaatioprofiili toimii tässä vaiheessa pohjana rajoitteille. Tärkeimpinä rajoitteina voidaan nostaa esiin lain sanelemat ja organisaation mahdolliset sopimukselliset tekijät, sillä nämä voivat rajoittaa käytettävissä olevia vaihtoehtoja, niin alustavalinnan, kuin valittavan pilvipalvelutyypin osalta. Näiden tekijöiden määrittäminen ajoissa on erittäin tärkeää. Sovelluksen ollessa jo tuotannossa tulee mahdollisesti kalliiksi ja työlääksi suorittaa migraatio uudelleen toiselle alustalle. On myös mahdollista, että joudutaan refaktoroimaan ja näin entisestään kasvattamaan kokonaiskustannuksia.

Mahdollisia organisaation sisäisiä rajoituksia ovat mm. sopivan osaamisen puute tai jopa pelko siitä, että pilveen siirryttäessä jäädään kiinni valittuun alustaan ja sen vaatimiin ympäristöihin niin, että tarjoajan tai palvelutyypin vaihtaminen ei ole mahdollista, tai vain todella kankeaa. Lainsäädäntö tai yhtiön omat vaatimukset voivat myös asettaa vaatimuksia palvelimien maantieteelliselle sijainnille, jolloin täytyy varmistaa palvelua valittaessa, että palvelimet saadaan varmuudella tietyltä alueelta, eikä niiden tietoja esimerkiksi varmuuskopioida muualle. On myös mahdollista, että palvelimien vaaditaan olevan dedikoituja ja niihin on pääsy vain lähiverkosta. Tässä tapauksessa pilviratkaisut eivät tule luonnollisesti kysymykseen, jolloin joko on kyseenalaistettava vaatimus itsessään, tai luopua pilvimigraatiosta vaihtoehtona kokonaan.

Erilaiset pelot ja epävarmuudet ovat myös organisaatiotason rajoitteita. Tietoturva halutaan monesti pitää kokonaan omissa käsissä, mitä se ei luonnollisesti pilvessä ole. Eri pilvipalvelut ovat lähtökohtaisesti hyvinkin tietoturvallisia ja tarjoavat paljonkin eri tapoja vaikuttaa turvallisuuteen, mutta loppujen lopuksi koneet, joilla palveluita ajetaan ja niihin liittyvät fyysiset verkot ovat organisaation näkökulmasta ulkopuolisen vastuulla. Erilaiset epävarmuustekijät, olivat ne todellisia tai eivät, voivat aiheuttaa organisaation sisällä muutosvastarintaa. Vastarintaa kohdattaessa on vastarinnan juurisyyt selvitettävä, ja en-

ne kaikkea otettava selvää kumpuaako vastustus todellisista uhkista. Jos näin on, toimitaan prosessin mukaisesti ja uhkien asettamat rajoitteet otetaan huomioon. Jos kuitenkin vastustus ei pohjaudu todellisiin uhkiin, on vastustajille perusteltava riittävän selvästi tehtävien valintojen järkevyys.

Mikäli organisaatiotason määrittelyssä törmätään rajoitteiseen, joihin ei pystytä vastaamaan, on koko migraatioprosessi syytä keskeyttää tähän vaiheeseen. Migraation päätöksentekoprosessissa on tärkeää olla organisaatiotason arviointi mukana, sillä näitä rajoitteita ei usein voida teknisesti ratkaista, jos ne tulevat tietoon liian myöhään. Varsinkin täysin estävät rajoitteet, kuten vaatimus dedikoiduille palvelimille tekevät koko pilvimigraation mahdottomaksi. Rajoitteiden aikainen havaitseminen on siten tärkeää, jotta selvitään ylimääräiseltä työltä ja kustannuksilta.

6.3 Ohjelmistoprofiilin määrittäminen

Ohjelmistoprofiilin määrityksessä on tavoitteena tunnistaa ohjelmistosta kaikki migraatioon vaikuttavat tekijät. Ennen kaikkea on tärkeää löytää migraatiota rajoittavat tekijät, mutta myös positiivisilla piirteillä on merkitystä. Ohjelmistoprofiilin määrityksellä on merkitystä varsinkin migroitavan sovelluksen tyyppiä arvioitaessa; jotkin ominaisuudet ohjaavat esimerkiksi pilvisovellusalustoja kohtaan ja toiset viestivät uuden ostamisen olevan parempi lähestymistapa. Profiilin määritys suoritetaan kahdessa vaiheessa. Ensimmäisessä keskitytään ohjelmiston käyttöön ja sen pääpiirteisiin. Toisessa ollaan teknisten ominaisuuksien parissa, jotka ohjaavat selvemmin päätöstä pilvialustasta ja lähestymistavasta.

6.3.1 Käyttöprofiili

Käyttöä tutkittaessa pyritään löytämään ne toiminnalliset ja ei-toiminalliset ominaisuudet, jotka vaikuttavat pilvimigraatioon tavalla tai toisella. Ohjelmistosta kannattaa määrittää sen tärkeimmät ominaisuudet, ja niiden erilaiset käyttötapaukset. Näin saadaan muodostettua kuva siitä, miten ohjelmaa todellisuudessa käytetään. Ohjelmiston käyttömäärät ja niiden mahdolliset jaksottaiset muutokset on myös hyvä määrittää. Eritoten siksi, että yksi pilvimigraation hyvistä puolista ja mahdollisesti yksi syistä miksi pilvimigraatiota harkitaan, on juuri mahdollisuus skaalata vaivattomasti käyttömäärien mukaan. Talouspuolen kannalta on tärkeää, että nykyiset ylläpito-, käyttö- ja kehityskustannukset saadaan määritettyä mahdollisimman tarkkaan, jotta kustannusvertailu pilvipalveluiden kanssa on mahdollisimman paikkansapitävä.

6.3.2 Tekninen profiili

Teknisessä määrityksessä pyritään selvittämään ohjelmiston käyttämät tekniset ratkaisut, joilla on merkitystä pilvimigraation kannalta. Tämä kattaa sekä ohjelmiston sisäiset tekniset ratkaisut, että mahdollisesti käytettävät kolmannen osapuolen kirjastot ja ohjelmat.

Ohjelmiston ajoympäristö on varsinkin merkityksellinen, sillä se määrittää pitkälti suoraan käytettävissä olevat pilvialusta- ja tekniikkavaihtoehdot. Toinen merkittävä tekijä käytetyt ohjelmointikielet, ohjelmointikehykset ja niiden vaatimat ajoympäristöt. Kuten luvussa 4 huomattiin .NET Coren ja Google Cloudin tapauksessa. Joiltakin pilvialustoilta puuttuu tuki tietyille teknologioille kokonaan, tai se on mahdollisesti puutteellinen.

Datan käsittely, kuten tietokannat ja tiedostojärjestelmät ja niiden käsittelyyn liittyvät kirjastot ovat myös merkityksellisiä. Vaikka pilvipalveluissa on nykyisellään laaja tuki erilaisille tietokantaratkaisuille, on vaihtoehtoja kuitenkin edelleen rajallisesti. Pilveen siirtyminen on myös hyvä tilaisuus arvioida tehdyt tietokantavalinnat uudelleen. Esimerkiksi perinteisemmät SQL-tietokannat, kuten Microsoftin ja Oraclen tuotteet ovat aikaisemmin olleet suosittu valinta, mutta varsinkin lisenssimaksut ja järjestelmien raskaus ovat hyvä syy harkita kevyempää ratkaisua. On myös mahdollista, että vastaavaa tietokantaa ei sellaisenaan ole tarjolla joissain, tai yhdessäkään palvelussa, jolloin vähintään tietokantaskeemaa joudutaan muokkaamaan alkuperäisestä.

Samaan tapaan kuin käyttöä arvioitaessa määritetään käyttömäärät ja ajat, on teknisestä näkökulmasta tärkeää selvittää ohjelmiston lähettämä ja vastaanottama datamäärä. Tällä on samaan tapaan merkitystä, kun arvioidaan eri palveluiden kokonaiskustannuksia. Pilvipalvelujen hinnoittelupolitiikan nojatessa vahvasti käyttömääriin, on tärkeää saada mahdollisimman tarkat tiedot käsiteltävistä datamääristä. Toinen hinnoittelua vahvasti ohjaava tekijä on pilvipalvelusta allokoitava laitteisto ja sen suorituskyky. Tämän takia määritetään myös tarvittava laitteistokapasiteetti ohjelmiston suorittamiselle. Tähän liittyen on mahdollisesti myös QoS-vaatimuksia, jotka on otettava niin ikään huomioon.

Ohjelmiston mahdolliset riippuvuudet ulkoisiin palveluihin on myös saatava selville. Näiden kohdalla merkittävintä on tietää, onko näihin palveluihin mahdollista yhdistää ulkoverkosta käsin, tai ovatko nämäkin ohjelmistot siirrettävissä pilveen. Mikäli yhdistäminen ei nykyisellään ole mahdollista, on tarvittavat muutokset tehtävä palomuureihin ja mahdollisesti määrittää VPN:n tai muun tekniikan avulla virtuaalinen verkko, jonka kautta yhteys on mahdollinen.

6.4 Pilvipalveluprofiilin määrittäminen

Pilvipalveluprofiilin määrityksessä pyritään selvittämään pilvipalvelun sopivuus suunniteltuun migraatioon. Eri pilvipalvelut tarjoavat eri palveluita, ja samoja palveluita on toteutettu palveluiden välillä eri tavalla. Myös pilvipalveluprofiilin määrityksessä pyritään löytämään rajoitteita, jotka ovat esteitä pilvipalvelun valinnalle. Nämä esteet tulee pystyä ratkaisemaan tavalla tai toisella, muuten pilvipalvelu tulee hylätä ja jatkaa kartoitusta toisella palvelulla.

Pilvipalveluprofiilit määritetään yhdelle tai useammalle palveluntarjoajalle. Ennen tätä vaihetta on siis täytynyt tehdä alustava kartoitus tarjolla olevista palveluista, joihin tässä vaiheessa tutustutaan tarkemmin, ja valitaan tarjoajista ne, jotka täyttävät aiemmissa vaiheissa asetetut ehdot ja rajoitteet. Palveluprofiilin määrittäminen kannattaa aloittaa

mahdollisten palvelutyyppien määrityksestä. Käytännössä kaikki palvelut tarjoavat virtuaalipalvelimia, mutta sovellusalustat, säiliöt ja serverless-palvelut rajoittuvat luonteestaan johtuen tiettyihin teknologioihin ja ohjelmointikieliin. Tässä kohtaa ei vielä kuitenkaan hylätä palvelutarjoajaa vain sen takia, että jokin ratkaisu vaatisi refaktorointia tai tekniikan vaihdosta, ellei näitä ole erikseen määritetty aiemmin rajoitteiksi, eli toisin sanoen halutaan pitäytyä tietyissä teknologioissa. Palvelutyyppien lisäksi kannattaa myös kartoittaa tarkemmin mitä palveluita tarjotaan mihinkin palvelutyyppiin. Skaalautuvuus, virheensieto ja virheistä palautuminen ovat esimerkkejä ominaisuuksista, joita voidaan liittää eri palvelutyyppeihin.

Pilvipalveluiden hinnoittelumallit ovat lähtökohtaisesti monimutkaisia ja eroavat toisistaan paikoitellen paljonkin. Tämän takia hinnoittelumallien keskinäinen vertailu on tehtävä tarkasti, pohjaten aiemmista kohdista saatuun tietoon. Lopullisen kustannuksen arviointi on sitä tarkempaa, mitä paremmin aiemmissa vaiheissa on osattu määrittää palvelun käyttäjätason ja tekniset vaatimukset. Vertailua vaikeuttaa entisestään palveluntarjoajien erilainen tarjoama, mitä tulee laitteistoon. Lähtökohtaisesti laitteistot määritetään tasoittain, ja jokaisen tason tarjoama suorituskyvyn yläraja määritetään. Rajat eivät kuitenkaan ole tarjoajien kesken linjassa, vaan menevät paikoitellen paljonkin limittäin, toisten tarjotessa paljon tarkempia tasoja. Palvelutaso vaikuttaa osaltaan hintaan paljonkin. Suurimalla osalla palveluntarjoajista on useita eri palvelutasoja, jotka määrittävät kuinka nopeasti ja minkälaista tukea on saatavilla. Palvelutasomallitkaan eivät ole keskenään suoraan verrattavissa, mutta jos haluttu minimitaso on määritetty, on palvelutasojen vertaaminen jo huomattavasti yksinkertaisempaa. Palvelutasojen lisäksi on myös hyvä kartoittaa tukea yleisesti, kuten tarjolla olevat yhteydenottotavat ja kielet.

Sovelluksen ja palvelutyypin ulkoiset palvelut on myös hyvä kartoittaa. Näihin kuuluvat erilaiset monitorointi-, skaalaus- ja varmuuskopiointipalvelut. Vaikka sovellus ei nykyisellään käyttäisi vastaavia palveluita, on hyvä ainakin kartoittaa tarjonta, sillä kyseisistä palveluista on varsinkin pilvinatiivien sovellusten kohdalla hyötyä. Varmuuskopiointi on itsessään hyödyllinen sovellukselle kuin sovellukselle. Ulkoisiin, mutta mahdollisesti hyvinkin merkityksellisiin palveluihin voidaan myös laskea erilaiset CI/CD-ratkaisut. Kuten kappaleessa 2 todetaan, varsinkin pilvinatiivien ratkaisujen tulisi pyrkiä hyväksikäyttämään jatkuvan integraation ja julkaisun tarjoamia mahdollisuuksia. Jos sovellusta ei enää juurikaan kehitetä, laskee kyseisten palveluiden tarve hieman, mutta jos sovellukseen joudutaan edes silloin tällöin julkaisemaan päivityksiä, helpottavat CI/CD-työkalut tätä prosessia.

Varsinkin jos aiemmissa vaiheissa on noussut esiin tarve saada pilvipalvelut tietyltä maantieteelliseltä alueelta, selvitetään missä palveluntarjoajan palvelinkeskukset sijaitsevat ja mitä palveluita mistäkin on saatavilla. Tämän selvittäminen on kannattavaa vaikkei maantieteellisellä sijainnilla olisikaan väliä esimerkiksi laillisista syistä. Tekniseltä kannalta on monestakin syystä järkevää käyttää palvelinkeskuksia, jotka ovat lähellä sovelluksen käyttäjiä. Palveluntarjoajien tarjoama tietoturva on myös tarkistettava. On selvitettävä, millaista tietoturvaa palveluntarjoajat tarjoavat ja miten siihen pääsee itse vaikut-

tamaan. Eri palvelutyypeillä voi olla omanlaisiaan tietoturvapalveluita, jota asiakas voi hallinnoida.

Palveluntarjoajan tukemat teknologiat, kuten ohjelmointikielet, ohjelmointikehykset ja ajoympäristöt kartoitetaan luonnollisesti myös, sillä näiden perusteella tullaan tekemään lopullinen migraatiopolun valinta. Ohjelmointikielien ja -kehysten lisäksi tulee olla tarkkana myös versioiden kanssa, sillä useat pilvipalvelut tukevat tietyistä teknologioista vain uudempia versiota. Itse sovellukseen liittyvien teknologioiden lisäksi myös tarjolla olevat tietokannat ja muut datan säilytyspalvelut tulee myös kartoittaa.

6.5 Teknisten ja taloudellisten rajoitteiden arviointi

Teknisten ja taloudellisten rajoitteiden arvioinnissa pyritään löytämään yhteensopivuus organisaatio-, ohjelmisto- ja pilvipalveluprofiilien välillä. Tämän selvittämiseen suositellaan tiettyjen rajoitteiden käyttämistä. Rajoitteet voidaan jakaa seitsemään kategoriaan, joiden perusteella voidaan arvioida profiilien yhteensopivuutta: taloudelliset rajoitteet, organisaatiotason rajoitteet, tietoturvarajoitteet, kommunikaatiorajoitteet, suorituskykyrajoitteet, saatavuusrajoitteet ja sopivuusrajoitteet.

Arviointi tulisi aloittaa taloudellisista rajoitteesta, sillä ne asettavat raamit harkittaville vaihtoehdoille. Taloudelliset rajoitteet ovat myös tyypillisesti hyvin kriittisiä, varsinkin pienille ja keskisuurille yrityksille. Tärkeimmät arvioitavat taloudelliset seikat ovat sovelluksen käyttökustannukset, joissa on otettava huomioon kaikki erilaiset palvelut ja instanssit, joita sovellus vaatii pilvessä toimiakseen. Kuten pilvipalveluprofiilin yhteydessä puhuttiin, on kokonaiskustannuksen arviointi haastavaa sen riippuessa useista erilaisista seikoista. Tarvittavien resurssien lisäksi käyttömäärät vaikuttavat suuresti kustannuksiin. Käyttökustannusten lisäksi itse migraatioprosessin aiheuttamat kulut on arvioitava. Tähän vaikuttaa suuresti valittava tekninen toteutus ja palvelutyyppi, varsinkin mahdollisten refaktorointikustannusten muodossa. Lisäksi on arvioitava mahdollisen käyttökatkon vaikutukset, samoin kuin tiedon siirtämiseen liittyvät kustannukset. Varsinainen pilviasennuskin tulee muodostamaan kuluja vähintään työn muodossa, sillä vaikka käytettäisiin virtuaalipalvelinta, joka on lähtökohtaisesti yksinkertaisin, pitää sovellus ja sen tarvitsemat ajoympäristöt ja muut riippuvuudet asentaa.

Organisaatiotason arvioinnissa keskitytään aiemmissa vaiheissa esiin tulleisiin rajoitteisiin. Näiden rajoitteiden tarkemman luonteen tuntemisiksi on todennäköisesti osallistuttava ohjelmistokehittäjien lisäksi myös muita asiantuntijoita. Tietoturvan arvioinnissa tärkeintä on varmistaa, että organisaation nykyinen tietoturvapolitiikka, sekä palveluntarjoajan tarjoama tietoturva ovat keskenään yhteensopivia, ja tarjotut ratkaisut riittäviä. Käsiteltäviä asioita ovat esimerkiksi tuetut salausmenetelmät, toimenpiteet eri asiakkaiden sovellusten virtualisoitujen ympäristöjen eriyttämiselle ja pääsynhallinta ulkoverkosta tulevalle liikenteelle.

Kommunikaatiorajoitteilla tarkoitetaan sovelluksen tarpeita verkkoyhteyteen liittyen. Arvioinnin alla ovat siis sovelluksen vaatimukset verkkoyhteyden laadulle, nopeudelle ja

luotettavuudelle. Palveluntarjoajan puolelta on merkitystä palvelinten maantieteellisellä sijainnilla. Verkkoyhteydet näihin palvelinkeskuksiin ovat pääasiassa varsin riittäviä, joten sijainti ja valitut palvelut merkitsevät eniten. Sovelluksen tarpeista riippuen pilveen migroiminen voi olla huonokin vaihtoehto, mikäli esimerkiksi viiveettömyydelle on suuret vaatimukset. Keskenään paljon kommunikoivien sovellusten on luonnollisesti hyvä sijaita samassa paikassa, joten tällaisten resurssien tapauksessa on hyvä siirtää joko kaikki, tai ei yhtäkään. Jos kyse on sisäisestä sovelluksesta, on organisaation omalla verkkoyhteydellä myös suuri merkitys. Vaikka palveluntarjoaja mahdollistaisi kuinka nopean ja vakaan yhteyden, ei siitä ole juurikaan hyötyä, jos organisaation oma internet-yhteys tai sisäverkko muodostaa pullonkaulan. Kommunikaatiorajoitteiden kohdalla on siis tärkeää tiedostaa jo olemassa olevan verkkoinfrastruktuurin rajoitteet.

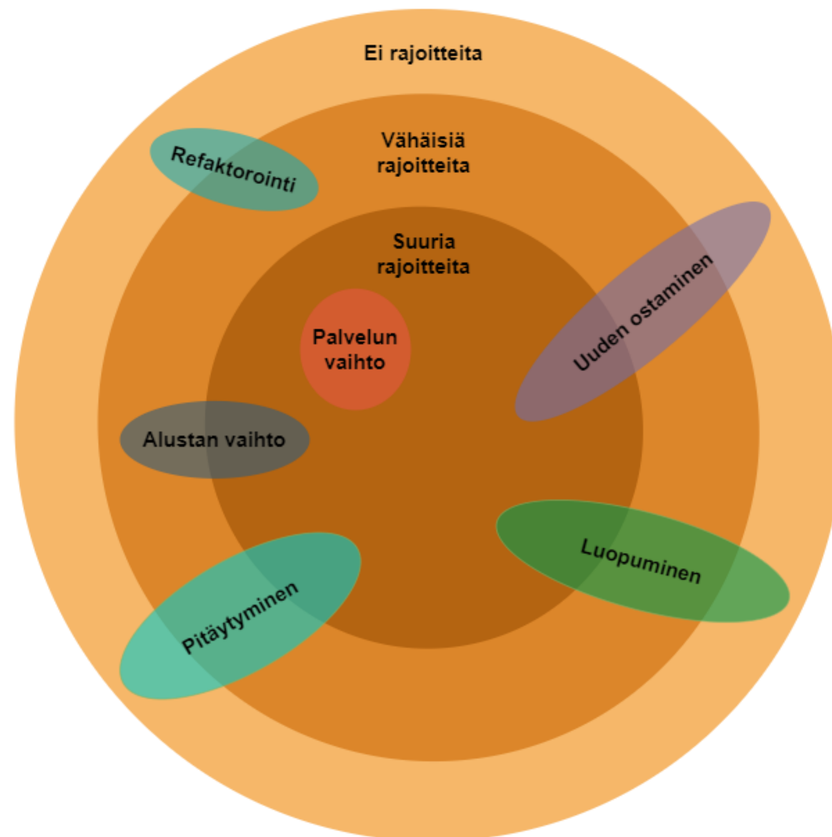
Suorituskykyrajoitteet ottavat suorimmin kantaa siihen, minkälaista laitteistoa tai tarkemmin suorituskykylupauksia palveluntarjoaja pystyy tarjoamaan. Osalle sovelluksista suorituskykyrajoitteita ei välttämättä juurikaan ole, kun taas toisille on hyvinkin tärkeää esimerkiksi vastata kyselyihin riittävän nopeasti. Suorituskykyrajoitteisiin liittyvät tietyssä määrin myös saatavuusrajoitteet. Skaalautuvuus voidaan katsoa kuuluvan jossain määrin molempiin, sillä kuorman alla skaalaaminen ei pelkästään mahdollista sovelluksen palvelevan siltä vaaditulla nopeudella, vaan pitää myös huolen, ettei kasvava kuorma kaada sovellusta, tai muuten estä saavutettavuutta. Molempiin vaikuttavien teknisten ratkaisujen lisäksi saatavuuteen vaikuttaa myös palvelutaso.

Sopivuusrajoitteissa on kyse niistä ohjelmistoon liittyvistä ominaisuuksista, jotka eivät sellaisinaan toimi, tai ole saatavilla pilvessä. Tekniset rajoitteet, kuten tuki sovelluskehitysympäristölle tai ohjelmointikielelle ovat ilmeisiä rajoitteita, mutta näiden lisäksi esimerkiksi kolmannen osapuolen ohjelmistojen tai kirjastojen lisenssien puutteet voivat aiheuttaa rajoitteen. Esimerkiksi tietokantavaihtoehtoja on eri pilvipalveluntarjoajilla tarjolla monenlaisia, mutta jokainen ei tue kaikkia. Ennen kaikkea vanhemmat versiot ovat harvinaisia. Tässä esitellyt rajoitteet eivät suinkaan ole toisistaan riippumattomia, vaan keskinäisiä vaikutuksia on nähtävissä [13]. Varsinkin taloudelliset rajoitteet vaikuttavat sopivuus- ja suorituskykyrajoitteisiin. Organinisaatiotaso taas sanelee kommunikaatioja tietoturvarajoituksia ja vaikuttaa niihin siten erittäin vahvasti. Saatavuusrajoite riippuu taas kommunikaatio- ja suorituskykyrajoitteista, sillä saatavuus saadaan aikaiseksi juuri näiden kahden rajoitteen käsittelemillä resursseilla.

Kun kaikki rajoitetyypit on kartoitettu ja esteitä ei ole tai löydetyt rajoitteet ovat kaikki ratkaistavissa tyydyttävällä tavalla, voidaan siirtyä seuraavaan vaiheeseen ja valita migraatiostrategia. Jos näin ei kuitenkaan ole, voidaan joko pyrkiä ratkaisemaan rajoitteiden esiin nostamat ongelmat joko sovellustasolla tai kartoittaa muita pilvipalveluntarjoajia. Jos kumpikaan vaihtoehto ei vaikuta mahdolliselta, on prosessi syytä keskeyttää kokonaan. Sellaiset palveluntarjoajat, jotka tukevat nykyisellään käytettyä ohjelmointikehystä tai ohjelmointikieltä ovat suositeltavia vaihtoetoja. Vähintäänkin siksi, että näin jätetään mahdollisuus käyttää nykyistä ohjelmakoodia lähes sellaisenaan auki. Tämän ollessa kustannustehokkain ratkaisu, ei sitä kannata ainakaan kovin kevein perustein hylätä.

6.6 Migraatiostrategian valinta

Mikäli aiemmissa vaiheissa ei olla löydetty rajoitteita, joita ei saada ratkaistua, voidaan siirtyä migraatiostrategian valintaan. Oikean strategian valintaan on vaikea muodostaa yksiselitteisiä ohjeita, sillä migroitavat ohjelmistot ovat hyvin erilaisia ja pilvipalvelut jatkuvassa muutoksessa. Kaikkein rajoitteiden selvittämisen jälkeen on kuitenkin paljon helpompi lähteä selvittämään mitkä strategiat ja pilvipalvelutyypit ovat mahdollisia. Tässä vaiheessa on kuitenkin jo selvillä mihin pilvipalveluun sovellus on tarkoitus siirtää. Vielä on kuitenkin auki mitä pilvipalvelutyyppiä tullaan käyttämään. Palvelutyypin valinnassa taloudelliset tekijät määrittävät ensin mahdollisten strategioiden avaruuden, jonka jälkeen voidaan vertailla eri strategioiden hyviä ja huonoja puolia. Kuvassa 6.1 esitellään taloudellisten rajoitteiden vaikutusta strategiavalintaan. Pitäytyminen, luopuminen ja uuden ostaminen ovat mahdollisia kaikilla taloudellisten rajoitteiden tasolla, mutta niiden valintaan voi ohjata vahvat taloudelliset rajoitteet. Palvelunvaihdon rajalliset hyödyt tarkoittavat, ettei sitä kannata valita kuin vahvojen taloudellisten rajoitteiden takia. Alustanvaihto tarjoaa jo huomattavasti enemmän etuja, joten sitä kannattaa harkita, vaikka taloudelliset rajoitteet eivät olisikaan sitä vaatimassa. Refaktorointi on lähtökohtaisesti vaihtoehdoista kallein, joten sitä voidaan harkita vain, jos taloudelliset rajoitteet ovat vähäiset.



Kuva 6.1. *Taloudellisten rajoitteiden vaikutus mahdollisiin strategiavalintoihin.*

Migraatiostrategiat jaettiin luvussa 3 kuuteen päätyyppiin. Strategiatyypit ohjaavat korkealla tasolla sitä, miten sovellus tullaan pilveen migroimaan. Varsinainen strategia on aina yksilöllinen ja riippuu monesta eri seikasta. Lopulliseen strategiaan voidaan nähdä kuuluvan myös käytännön toimien määrittely. Näihin toimiin ei tässä työssä oteta kantaa, vaan keskitytään strategiatyypin ja sopivan alustan määrittämiseen. Strategiatyypeistä ei tässä yhteydessä käsitellä luopumista tai pitäytymistä, sillä ne vastaavat Cloudstep-mallissa prosessin keskeytystä. Uuden ostaminen jätetään myös vähälle huomiolle, sillä se ei vastaa sovelluksen pilveen migroimista edes silloin, kun valitaan SaaS-palvelu. Sovelluksen ostaminen ei suinkaan ole yksinkertaista, sillä jo pelkästään sopivan tarjoajan valinta on monimutkainen prosessi. Alkuperäinen ohjelmisto ei kuitenkaan tässä tapauksessa liity migraatioon kuin pääasiassa datan ja skeemojen muodossa. Sopivan tarjoajan valintaa ei myöskään koettu järkeväksi käsitellä, sillä sopiva korvaaja on vahvasti riippuvainen nykyisen sovelluksen toiminnasta ja ohjelmalogiikasta, ei niinkään teknisestä toteutuksesta.

Taulukko 6.1. *Pilvipalvelutyyppien sopivuus eri strategiatyypeille.*

Strategiatyyppi	Virtuaalipalvelin	Sovellusalusta	Säiliöt	Serverless
Palvelunvaihto	X			
Alustanvaihto	X	X	(X)	
Refaktorointi	(X)	X	X	X

Eri palvelutyypit sopivat luonnostaan toisille migraatiostrategioille, kun taas toiset soveltuvat useampaan. Taulukossa 6.1 esitellään eri strategiatyypeille hyvin soveltuvat palvelutyypit. Virtuaalipalvelimet ovat luonnollisesti sopiva palvelun- ja alustanvaihdolle. Näissä strategioissa sovellus pysyy käytännössä muuttumattomana, mutta sen siirretään pilveen, jossa sen käyttöä jatketaan pääasiassa kuin ennenkin. Virtuaalipalvelimia kannattaa harkita, jos pääasiallisesti tavoitellaan pilven mahdollistamia säästöjä, tai joitain ominaisuuksia, kuten kuormantasausta. Yleisesti pilvestä virtuaalipalvelimilla saavutettavat hyödyt eivät juurikaan liity itse sovellukseen, vaan palveluihin, joita siihen on liitettävissä. Virtuaalipalvelin ei vastaa pilvinatiivia ajattelua, mutta tarjoaa silti monia muitakin kuin taloudellisia hyötyjä, joita ei dedikoidulla palvelimella saavuteta. Refaktorointiin virtuaalipalvelimet soveltuvat huonosti. nykyinen sovellus voidaan refaktoroida virtuaalipalvelimessa ajettavaksi, mutta kuten mainittua, tällä ei saavuteta pilvinatiiviuden hyötyjä, joten se ei ole varsinaisesti suositeltavaa. Voidaan myös ajatella, ettei ole kannattavaa tyytyä virtuaalipalvelimiin pohjautuvaan ratkaisuun, jos ohjelmisto on jo päätetty refaktoroida.

Refaktorointi on pilvipalvelutyypin puolesta vapain strategiatyyppi. Sinänsä palvelutyypiksi voidaan valita mikä tahansa tässä työssä esitellyistä, mutta pilvinatiivein lopputulos saadaan säiliö- tai serverless-pohjaisilla ratkaisuilla. Molemmat mahdollistavat luonnostaan useita pilvinatiiveja ominaisuuksia, kuten skaalautuvuuden ja virheensiedon. Lopulliseen strategiatyypin ja palvelutyypin valintaan ei voi antaa suoria ohjerajoja, mutta jonkinlaisia rajatapauksia voidaan muodostaa. Näitäkin päätöksiä ohjaavat määritetyt rajoitteet ja tavoitteet. Mikäli taloudelliset rajoitteet eivät ole ongelma ja organisaatiotasolla ollaan halukkaita modernisoimaan sovellusta voimakkaasti, on syytä harkita refaktorointistrategiaa ja valita alustaksi joko säiliöt tai serverless, mikropalveluarkkitehtuuria toteuttaen. Varsinkin siinä tapauksessa, että sovellus on jaettavissa loogisiin kokonaisuuksiin, joista voidaan muodostaa erilliset mikropalvelut. Se, valitaanko lopulta säiliöt vai serverless on paljolti kiinni myös organisaation nykyisestä osaamisesta ja mieltymyksistä. Serverless vaatii todennäköisesti suhteessa enemmän refaktorointityötä, sillä jokainen toiminallisuus tulee toteuttaa uudelleen. Alkuperäisestä alustasta riippuen sama pätee myös säiliöhin, mutta huomattavasti laajempien ajoympäristövaihtoehtojen ansiosta todennäköisyydet päästä vähemmällä ovat suuret. Valitsemalla näistä pilvipalvelutyypeistä toisen ja toteuttamalla mahdollisuuksien ja tarpeiden mukaan luvun 2 mainitsemat ominaisuudet, saadaan pilvinatiivi kokonaisuus, jolla on mahdollista ottaa pilvestä sen tarjoamat hyödyt parhaiten käyttöön. On myös mahdollista käyttää useita eri pilvipalvelutyyppejä saman kokonaisuuden toteutukseen. Tietyt osat sovelluksesta voi olla helpoin toteuttaa sovellusalustalle, kun taas toiset säiliöinä ja osa mahdollisesti serverless-funktioina.

Aina ei ole tarpeellista tai tavoitteiden mukaista lähteä täysin pilvinatiiviin toteutukseen, ja sen vaatimaan refaktorointiin. Maltillisemmalla strategialla saadaan pilven tarjoamat hyödyt käyttöön huomattavasti pienemmillä kustannuksilla. Palvelunvaihto-strategiatyypissä sovellus siirretään pilveen sitä muokkaamatta. Käytännössä tämä tapahtuu yleensä, varsinkin legacy-sovellusten tapauksessa, virtuaalikoneena. Virtuaalikone on mahdollista pystyttää vastaamaan nykyistä ajoympäristöä ja lisenssien niin salliessa saadaan virtuaalipalvelimelle asennettua kaikki dedikoidun ajoympäristön palvelut. Prosessi ei välttämättä ole kuitenkaan näin suoraviivainen, vaan palveluntarjoajat tarjoavat erilaisen määrän käyttöjärjestelmävaihtoehtoja ja muita palveluita. Lähtökohtaisesti kyseessä on kuitenkin vähätöisin ja -kuluisin tapa siirtää sovellus pilveen. Näin saadaan kuitenkin hyödynnettyä pilviympäristön hyötyjä vain rajallisesti, joten tavoitteet pilveen siirtämiselle eivät saa nojata pilvinatiiveihin ominaisuuksiin. Pilveen siirtämisellä saatetaan kuitenkin saavuttaa säästöjä mm. ylläpitokustannusten muodossa ja hyvin yksinkertainen skaalaaminen sekä kuormantasaus on kuitenkin mahdollista.

Mikäli pilven ominaisuuksia halutaan hyödyntää laajemmin, mutta talousrajoitteet, tai tavoitteet yleisesti eivät puolla laajaa refaktorointia, voidaan strategiatyypiksi valita alustanvaihto. Tässä strategiatyypissä sovellusta muokataan vain jossain määrin, jotta saadaan enemmän irti pilven tarjoamista eduista. Erona refaktorointiin voidaan pitää juuri kooditason muutosten vähyyttä. Sovelluksen ohjelmointikehys voidaan joutua päivittämään uuteen versioon ja tämän vaatimat muutokset tehdä, mutta esimerkiksi .NET-sovelluksen muuttaminen .NET Core -sovellukseksi menee yleensä jo refaktoroinnin puolelle. Raja on luonnollisesti jossain määrin häilyvä. Alustanvaihdolle sopivat pilvipalvelutyypit ovat virtuaalipalvelin ja pilvisovellusalusta. Sovellusalustalla saadaan todennäköisesti hyödynnettyä pilvipalvelua ja sen tarjoamaa virtualisoitua palvelinta parhaiten, sillä sovellusalustat tarjoavat ajoympäristön lisäksi useita erilaisia hyödyllisiä ominaisuuksia. Säiliöpohjainen ratkaisu menee yleensä helposti refaktorointi-strategian puolelle, ellei sovellus entuudestaan noudata mikropalveluarkkitehtuuria. Sovellusalustalle toteutetulle sovellukselle voidaan kuitenkin saada virtuaalipalvelinratkaisuja paremmin pilvinatiiveja ominaisuuksia, kuten virheensieto, skaalautuminen ja jatkuva julkaisu. Alustanvaihto virtuaalipalvelimelle tarkoittaisi, että sovellukseen tehdään muutoksia, joiden avulla saadaan joitain pilvinatiiveja ominaisuuksia paremmin käyttöön. Teoriassa tällainen voisi olla tietokannan vaihtaminen virtuaalikoneen sisällä ajettavasta tietokantainstanssista palveluntarjoajan tietokantapalveluun.

__Taulukko 6.2.__ Palvelutyypin valinta tarpeiden perusteella.
Ominaisuuksien ja tarpeiden sopivuutta arvioitu asteikolla 0-3, jossa 3 tarkoittaa parasta sopivuutta.

	Virtuaalipalvelin	Sovellusalusta	Serverless	Säiliöt
Ohjelmiston ominaisuudet				
Mikropalveluarkkitehtuuri	0	1	3	3
Monoliittiarkkitehtuuri	3	2	0	1
Web-rajapintasovellus	1	3	2	2
Useita erilaisia instansseja samasta palvelusta	0	2	1	3
Raskasta ohjelmalogiikkaa	2	2	1	2
Ohjelmiston tarpeet				
Horisontaalinen skaalaus	3	2	1	1
Vertikaalinen skaalaus	1	2	3	2
Monitorointi	1	3	3	2
Vikasietoisuus ja palautuminen	1	2	3	3
Jatkuva integrointi ja julkaisu	1	3	3	3
Ajoympäristön muokattavuus	3	1	0	3

6.7 Strategian valinta .NET-ohjelmistolle

Edellä kuvailtu päätöksentekomalli toimii sellaisenaan ympäristöstä riippumatta. Mallin avulla on mahdollista arvioida strategiavaihtoehtoja, oli migroitava sovellus kehitetty millä alustalla tahansa. Pilvipalveluita arvioitaessa on mahdollista ja suositeltavaakin ottaa vertailuun mukaan muitakin pilvipalvelualustoja, kuin tässä työssä käsiteltävät. Organisaatioprofiiliin ei tässä aliluvussa oteta juurikaan kantaa, sillä käytettävän ohjelmointikielen ja -ympäristön kannalta organisaatiolla ei ole suoraan merkitystä. Luonnollisesti näiden valintaa on voinut liittyä organisaatiotason rajoituksia, mutta näiden yleistäminen on tässä tilanteessa tarpeetonta. Microsoftin Azure-pilvipalvelu on todennäköisin valinta siirrettäessä .NET-sovelluksia pilveen, sillä Microsoft tuo luonnollisesti ensimmäisenä uudet ominaisuudet omaan palveluunsa ja toisaalta tukee omaa kehitysympäristöään. Kuten tässä työssä on kuitenkin huomattu, ovat muut pilvipalvelut myös varteenotettavia vaihtoehtoja, kunhan ollaan tietoisia mahdollisista rajoituksista.

.NET:llä toteuttujen legacy-sovellusten migraatiota suunnitellessa suuri merkitys on ohjelmistoprofiilin määrityksellä. On tärkeää kartoittaa ohjelmiston nykyinen arkkitehtuuri, riippuvuudet ja ajoympäristö. Järjestelmät, jotka ovat olleet pitkään käytössä ovat saattaneet paisua huomaamattomasti ja kokonaisuuden jäljittäminen ei välttämättä ole yksinkertaista. Varsinaiseen pääsovellukseen lisättyjen ominaisuuksien lisäksi on voitu toteuttaa ulkoisia sovelluksia, joita saatetaan ajaa samalla tai erillisellä palvelinkoneella. Myös kolmansien osapuolien palvelut, joihin ollaan yhteydessä, on selvitettävä, sillä on mah-

dollista, että näihin kyseisiin palveluihin saadaan yhteys virtuaaliverkon tai muun ratkaisun kautta, jolloin julkisesta pilvestä suora yhteys on mahdoton. Tähänkin on kuitenkin ratkaisuja, joista lisää pilvipalveluprofiilin määrityksen yhteydessä.

Ohjelmistoprofiilin mukaiset rajoitteet, vaatimukset ja mahdollisuudet ovat suurimmassa roolissa tässä tarkkailussa. Käsiteltävien kolmen palveluntarjoajan pilvipalveluprofiilit rajaavat joitakin mahdollisia strategioita pois toisilta alustoilta, mutta kaikki päätyypit ovat mahdollisia vähintään yhdellä näistä alustoista. Taloudelliset rajoitteet otetaan myös huomioon, sillä yleensä taloudelliset tekijät sanelevat lopulta valittavan ratkaisun. Täydellinen refaktorointi on aina mahdollinen, mutta toteutukseen tarvittavat resurssit vaihtelevat. Taloudelliset rajoitteet muodostavat rajan, jonka alapuolelle jäävät käytettävissä olevat vaihtoehdot.

Migroitavaa ohjelmistoa tulee myös tutkia kokonaisuuden lisäksi osiensa summana, sillä yksittäiset sovellukset, jotka kuuluvat samaan ohjelmistoon, voi olla mahdollista migroida jopa eri strategioita käyttäen, tai vähintään eri arkkitehtuureilla. Arkkitehtuurityypit eivät sulje toisiaan pois, vaan niitä voidaan käyttää myös yhdessä. Jos ohjelmasta on tunnistettavissa modulaarisia osia, jotka voidaan toteuttaa omina palveluinaan, toteutetaan nämä osat omina mikropalveluinaan ja jätetään suurin osa ohjelmasta koskematta. Pääohjelma voidaan toteuttaa esimerkiksi verkkopalvelusovelluksena ja erillset palvelut serverless- tai säiliöratkaisuina.

6.7.1 Palvelunvaihto

Halvin ja yksinkertaisin tapa suorittaa migraatio on, kuten usein on jo mainittu, palvelunvaihto. .NET-sovellukselle palvelunvaihto tarkoittaa käytännössä nykyisen dedikoidun Windows-palvelimen vaihtamista virtuaalipalvelimeen. Ratkaisu tulee kyseeseen, kun migraatiolla tavoitellaan lähinnä infrastruktuurisäästöjä, eikä muille pilven tarjoamille eduille ole tarvetta. Ratkaisu tyydyttää hyvin todennäköisesti taloudelliset rajoitteet yksinkertaisuutensa takia. Organisaatiotason rajoitukset todennäköisesti vaikuttavat lähinnä siihen, mikä pilvipalveluntarjoaja valitaan. Lähtökohtaisesti jokaisen palveluntarjoajan alustoilla on tarjolla samat työkalut ja palvelut, vain hinnoittelussa ja tarjolla olevassa laitteistossa on eroja. Määritettävää ja valittavaa jää siis melko vähän, eikä valinnoilla ole kovin suurta merkitystä, sillä jokainen valinta on vähintäänkin riittävän hyvä.

6.7.2 Alustanvaihto

Haluttaessa ottaa joitain pilven ominaisuuksia käyttöön, mutta on päädytty kartoitettujen taloudellisten, organisaatiotason tai ohjelmistorajoitusten puolesta pitäytymään pienissä muutoksissa, voidaan valita strategiaksi alustanvaihto. Tällöin toimitaan hyvin samaan tapaan, kuin palvelunvaihdossa, mutta tehdään joitakin muutoksia ohjelmistoon tai sen käyttämiin palveluihin. Aiemmin mainittu dedikoidun tietokannan vaihtaminen palveluntarjoajan tietokantaan on hyvä esimerkki. Tässäkin käytännössä siirretään ohjelmisto

ympäristöineen virtuaalipalvelimelle, joten valinnan kannalta samat lainalaisuudet pätevät, kuin edellä palvelunvaihdossa. Tehtävä muutos saattaa kuitenkin olla sellainen, että jokin palveluntarjoaja ei tue sitä suoraan, tai se voi olla esimerkiksi yllättävän kallis. Tietokantojen tapauksessa on mahdollista, että sopivaa tietokantapalvelimen versiota ei ole saatavilla, jolloin joudutaan vähintään päivittämään nykyinen tietokantaskeema uudempaa versiota vastaavaksi.

Teoriassa alustanvaihdossa voidaan virtuaalipalvelimen lisäksi harkita myös Pilvisovellusalustaa tai säiliöitä, mutta tämä vaatii sovelluksen nykyiseltä rakenteelta tiettyjä piirteitä. Lievimmillään sovellusalustalle siirtyminen vaatii tietyn ohjelmointikehysversion. Näihin päivittäminen ei vielä välttämättä tarkoita, että puhutaan refaktorointistrategiasta, mutta ollaan jo hyvin lähellä. Varsinkin jos ohjelmiston rakenne on yksinkertainen, on sovellusalustalle siirtyminen mahdollisesti hyvin suoraviivaista ja siten kannattavaa, sillä näin saavutetaan jo useita pilvinatiivin ohjelmiston etuja, kuten skaalautuminen, vikasietoisuus ja monipuoliset monitorointi- ja ylläpitotyökalut. Työn määrä on sitä suurempi, mitä enemmän erillisiä sovelluksia täytyy julkaista, mutta työ on suoraviivaista. Käytännön testeissä havaitut rajoitteet .NET:llä kirjoitettujen Windows-palveluiden toteuttamisesta sovellusalustalle rajaavat pilvipalvelut käytännössä Azureen, jos palveluita on tarve käyttää, sillä muille alustoille jouduttaisiin kirjoittamaan nämä palvelut uudestaan. Säiliöiden käyttäminen alustanvaihtostrategiassa vaatii ohjelmistolta mikropalvelumaista luonnetta, jotta saavutetaan mainittavia hyötyjä. Joissain tapauksissa sovellus voi olla hyödyllistä toteuttaa säiliöissä ajettavana, vaikka ei olisikaan kyse mikropalveluarkkitehtuurista. Säiliöiden avulla on mahdollista toteuttaa esimerkiksi skaalaus huomattavasti paremmin, kuin virtuaalipalvelimella. Säiliöitä varten jokainen palveluntarjoaja on varteenotettava vaihtoehto, sillä jokaisella on omat palvelunsa Windows-pohjaisten säiliöiden ajamiseen, monitorointiin ja hallintaan.

6.7.3 Refaktorointi

Kaikkein eniten valittavaa ja liikkumavaraa on, kun strategiaksi harkitaan refaktorointia. Lähtökohtaisesti refaktorointi on strategiatyypeistä kallein ja työläin, mutta tarjoaa myös parhaat mahdollisuudet hyödyntää uutta alustaa tehokkaasti. Refaktorointiakin voidaan suorittaa monella tasolla; muutokset joihinkin ohjelmiston luokkiin tai ohjelmointikehyksen päivittäminen ja sen mukanaan tuomion kirjastomuutosten toteuttaminen on vielä huomattavasti vaivattomampaa, kuin ohjelmiston arkkitehtuurin muuttaminen, esimerkiksi monoliitista mikropalveluihin. Tätäkin päätöstä tehdessä on hyvä peilata ohjelmiston nykyistä rakennetta ja migraatiolle asetettuja tavoitteita. Palvelutyypeissä on paljon valittavaa ja samoihin tavoitteisiin voi päästä usealla eri tavalla. Toiminnaltaan identtiset ohjelmistot voidaan toteuttaa käytännössä jokaisella palvelutyypillä.

Vanhalla .NET-ohjelmointikehyksellä toteutetun ohjelmiston tapauksessa refaktorointi voi olla suurimman osan alustoja tapauksessa kannattavaa aloittaa päivittämällä ohjelmointikehys Coreen. AWS:n ja Azuren tuki eri ohjelmointikehyksen versioille on suhteellisen

kattava, mutta esimerkiksi Googlen App Engine tukee vain Corea. Luultavasti suurin osa muista markkinoiden pilvipalveluista tukee GCP:n tapaan vain Core-versioita, jos niitäkään. Serverless-ratkaisut vaativat Core-päivityksen lähes kaikilla alustoilla, eli näiden välille ei synny merkittävää eroa, joten organisaation käytännöt ja hinnoittelu nousevat ratkaisevaan asemaan. Säiliötoteutuksessa ja IaaS-ratkaisuissa ei ole tarvetta ohjelmistokehyksen päivitykselle, sillä tässä käsiteltävillä alustoilla tuetaan sekä Windows-säiliöitä, että -virtuaalipalvelimia. Tässä vaiheessa on kuitenkin syytä harkita Coreen päivittämistä joka tapauksessa. Riippuu täysin ohjelmiston luonteesta ja käyttötarkoituksesta, onko päivitys vaivan arvoista, mutta ainakin se lisää joustavuutta tuleviin alustavalintoihin. Jos refaktorointi on tarkoitus tehdä todella kattavasti ei ole välttämätöntä pitäytyä .NET-kehyksessä, vaan teoriassa voidaan siirtyä toiseenkin teknologiaan, jos esimerkiksi nykyisen henkilöstön osaaminen tukee valintaa paremmin.

Refaktoroinnissa merkittävimmät pilvipalvelutyypit ovat sovellusalustat, serverless ja säiliöt. virtuaalipalvelin on luonnollisesti myös vaihtoehto, mutta tarpeet refaktoroida sovellus, mutta toteuttaa se silti virtuaalipalvelimelle vaatii jo tämän mallin ulkoisia syitä päätökselle. Kolme jäljelle jäävää vaihtoehtoa vastaavat monelta osin samaan tarpeeseen, mutta toteutustavat, samoin kuin refaktoroinnin vaatima työmäärä kuitenkin eroavat. Työmäärä ja refaktoroinnin haastavuus riippuu myös nykyisestä arkkitehtuurista ja ympäristöstä. Nyrkkisääntönä voidaan kuitenkin sanoa, että serverless-toteutus on todennäköisesti suuritöisin. Säiliöiden ja sovellusalustojen välinen ero ei ole yhtä selkeä ja vaatii tarkempaa tutustumista. Molempien tapauksessa sovellus on kannatavaa pilkkoa mikropalveluarkkitehtuurin mukaisesti useaan palveluun, ja tämä työmäärä on suurin piirtein sama. Säiliöt on mahdollista toteuttaa versiopäivityksellä tai ilman kaikissa vertailussa mukana olleilla alustoilla, mutta sovellusalustalle migroiminen onnistuu vain AWS:lla ja Azurella. Mahdolliset palveluina ajetut sovellukset eivät, kuten mainittua, ole suoraan .NET:llä toteutettavissa kuin Azuressa, joten näitä joudutaan uudelleenkirjoittamaan, jos päädytään valitsemaan GCP tai AWS. Seuraavaksi voidaan arvioida ohjelmiston monimutkaisuutta ja tarvetta päästä tarkemmin käsiksi ajoympäristöön. Lähtökohtaisesti sovellusalustat ovat parhaimmillaan yksinkertaisten sovellusten kanssa, joiden toimintaan ei ole tarvetta koskea ajonaikaisesti tai päästä tutkimaan tarkemmin, kuin mitä pilvipalvelu tarjoaa. Sovellusalustan yksinkertaisuus tuo mukanaan myös rajoitteita. Kaikki ajettavat instanssit ovat täsmällisiä kopioita toisistaan, eikä instanssissa ole kuin ajettava ohjelma ja sen tarvitsema ajoympäristö [28]. Säiliöt tarjoavat tässä mielessä paljon paremmin vaihtoehtoja. On kehittäjän käsissä mitä säiliöissä ajetaan ja yksittäiseen säiliöön on mahdollista ottaa etäyhteys ja tutkia suoraan lähteellä mikä aiheuttaa ongelmia. Konfigurointi on myös monella tavalla tarkempaa. Kaikki nämä hyödyt saavutetaan kuitenkin helppokäyttöisyyden kustannuksella. Pääasiallisesti valinta säiliöiden ja sovellusalustan välillä jää siis kahden ääripään välille; monimutkainen, mutta joustava ja muokattava toteutustapa vai helppokäyttöinen ympäristö, jonka rajoitteet on hyväksyttävä. Harva ohjelmisto on suoraan kummassakaan ääripäässä, mutta säiliöt ovat edelleen syystä vallitseva pilvipalveluratkaisu, joten kaikissa paitsi selvästi riittävän yksinkertaisissa ohjelmistoissa kannattanee valita säiliöt. Tämä myös siksi, että säiliöiden toteutustapa on alustavapaampi, joten palvelun-

tarjoajan vaihtaminen on tarvittaessa vaivattomampaa.

Serverless eroaa ratkaisuna aiemmin vertailluista säiliöistä ja pilvisovellusalustoista erito-ten siinä, että ajettavaan instanssiin ei ole lainkaan hallintaa. Ohjelma jaetaan funktioksi, joita suoritetaan jonkin liipaisevan toiminnon tapahtuessa. Serverless-instanssi on lähtö-kohtaisesti tilaton, vaikka tilallisiakin ratkaisuja on olemassa [54]. Instanssit eivät myös-kään ole ajossa kuin ainoastaan silloin, kun funktioita kutsutaan. Tämän takia serverless kärsii kylmäkäynnistymisestä. Funktioiden ajaminen tauon jälkeen kestää huomattavasti kauemmin, sillä resurssit kyseiselle funktiolle allokoidaan vasta kutsun saapuessa [46]. Hallinnan puute, tilattomuus ja kylmäkäynnistys ovat ominaisuuksia, jotka on serverless-arkkitehtuuria harkittaessa otettava huomioon. Mutta jos ohjelma on loogisesti jaettavissa funktioihin, eikä tilattomuus tai suuret viiveet kylmäkäynnistyksen takia tuota ongelmia, on serverless varteenotettava vaihtoehto. Funktioita laskutetaan vain ajetun ajan perusteel-la [41], joten taloudellisessakin mielessä serverless voi olla järkevä valinta. Ennen kaik-kea funktioita kannattaa harkita toteuttamaan erilaisia ajastettuja tai jonkin muutoksen tapahtuessa suoritettavia ohjelmiston osia. Edellä mainitut serverless:n rajoitteet eivät näiden tapauksessa yleensä tuota ongelmia ja serverless tarjoaa helpon ja kustannus-tehokkaan tavan toteuttaa näitä .NET Core-sovellus on mahdollista toteuttaa serverless-arkkitehtuurilla niin Azuressa, kuin AWS:ssa. Eroja kahden välillä ei juurikaan ole, kuten käytännön vertailussa huomattiin. AWS Lambdan tarjoama tapa kauttakulkufunktion avat kokonainen web-rajapinta voi olla mielenkiintoinen vaihtoehto, jos migroitava ohjelmisto, tai sen osa toteuttaa web-rajapinnan. Web-rajapinnan toteutus on muutenkin molemmil-la suhteellisen virtaviivaista; rajapintametodien kuvaukset muutetaan funktioesittelyiksi ja julkaisukonfiguraatiot muokataan serverlessin mukaisiksi.

Oikean pilvipalvelutyypin valinta on refaktoroitaessa ilmeisen haastavaa, sillä vaihtoehto-ja on runsaasti. Taulukossa 6.2 on pisteytetty eri palvelutyypit pilvimigraation tarpeiden ja migroitavan sovelluksen ominaisuuksien mukaan. Pisteitä on annettu nollasta kolmeen, sen perusteella kuinka hyvin kukin alustoista vastaa kyseiseen tarpeeseen tai ominai-suuteen. Kolmen pisteen alusta vastaa tarpeeseen täysin, kahdella pisteellä alusta vas-taa tarpeeseen, mutta ei yhtä hyvin kuin jokin toinen alusta. Yhden pisteen alustalla on joitakin tarvetta tukevia ominaisuuksia, mutta se ei varsinaisesti vastaa tarpeeseen. Omi-naisuuksilla tarkoitetaan sovelluksen ominaisuuksia siinä vaiheessa, kun migraatiota pil-veen suoritetaan, eli mahdollisten refaktorointien ja muiden muutosten jälkeen. Tarpeet puolestaan kuvastavat lopullisen ohjelmiston tarpeita. Pisteytyksen pohjana on käytetty tässä työssä esiteltyjä ominaisuuksia, käytännön vertailua ja omaa kokemusta. Taulukkoa voi käyttää apuna palvelutyypin valitsemisessa ottamalla huomioon taulukossa luetelluis-ta ominaisuuksista migroitavaa sovellusta koskevat rivit ja laskemalla kunkin palvelutyypin tuottamat pisteet. Mallin epätarkkuuden takia varsinkin lähes samoihin pisteisiin pääsevät palvelutyypit kannattaa ottaa tarkempaan selvitykseen. Taulukkoa voi myös käyttää poh-jana omiin vertailuihin lisäämällä ominaisuuksia ja tutkimalla palvelutyyppien tarjoamia hyötyjä kyseisille ominaisuuksille.

Jokaisella pilvipalvelutyypillä on omat hyvät ja huonot puolensa, ja täyden hyödyn näis-

tä saa vain tuntemalla migroitavan ohjelmiston ja sen vaatimukset. Muutama sovelluksen päätyyppi voidaan avata seuraavaan tapaan. Yksinkertaiset sovellukset sopivat pilvisovellusalustalle tai serverless-ympäristöön. Valinta näiden kahden välillä riippuu rakenteesta ja tarpeesta. Mikäli havitellaan vahvaa mikropalveluarkkitehtuuria ovat säiliöt ilmeinen valinta. Mikropalveluarkkitehtuuri on toteuttavissa myös sovellusalustalle. Toteutus voi olla säiliöitä helpompi, mutta sovellusalustan instanssien heikko yksilöitävyys on tiedostettava. Jos ohjelmiston arkkitehtuuri pidetään edelleen monoliittisena, on sovellusalusta paras vaihtoehto. Monoliittisen rakenteen vuoksi ei kaikkia pilven hyötyjä saada hyödynnettyä, mutta sovellusalustat tarjoavat kuitenkin oman osansa, kuten vikasietoisuuden, kuormantasauksen ja ainakin horisontaalisen skaalautuvuuden.

6.7.4 Luopuminen ja pitäytyminen

Jos päätöksentekomallissa on päädytty keskeyttämään migraatioprosessi, on vielä jäljellä valinta pitäytymisen ja luopumisen välillä. Keskeytetty prosessi kertoo jo itsessään, että ohjelmiston tarvetta ja nykytilaa on syytä tutkia. Onko mahdollista, ettei ohjelmistolle ole enää riittävän merkittävää käyttöä puolustamaan sen ylläpitokustannuksia? Jossain tilanteessa voi olla jopa järkevää toteuttaa ohjelmisto kokonaan uudestaan, refaktoroinnin sijaan. Tämä on kuitenkin tämän migraatiota käsittelevän työn ulkopuolella. Jos ohjelmiston käyttökartoituksen perusteella voidaan todeta, että käyttöä ja tarvetta on luonnollista jatkaa ohjelmiston käyttöä entiseen tapaan. Kartoitetut rajoitteet on kuitenkin hyvä pitää mielessä siltä varalta, että esimerkiksi tekninen kehitys, lakimuutos tai organisaation toimintamallien muutos mahdollistaa pilvimigraatioprosessin uudelleen käynnistämisen.

7 YHTEENVETO

Tässä työssä käsiteltiin legacy-ohjelmistojen pilvimigraatiota sekä yleisesti, että .NET-sovelluksille. Käytännön vertailulla kartoitettiin eri pilvipalveluiden eroja migroimalla yksinkertainen .NET-ohjelma. Tämän lisäksi käsiteltiin erilaisia migraatiostrategioita ja pyrittiin määrittämään kullekin sopiva käyttötarkoitus. Käytännön vertailun tulokset ja esitellyt migraatiostrategiat yhdistettiin vielä Cloudstep-malliin, pyrkien näin muodostamaan mallin, jonka perusteella voidaan arvioida kokonaisvaltaisesti sopivaa migraatiotapaa legacy-sovelluksille, varsinkin .NET-ohjelmiston tapauksessa.

Käytännön vertailun avulla selvitettiin kolmen eri pilvipalvelun: Amazon AWS:n, Google Cloud Platformin ja Microsft Azuren eroja migroitaessa .NET-sovellusta pilvisovellus- ja serverless-alustoille. Vertailussa havaittiin, että Azure ja AWS tarjoavat keskenään kilpailukykyisiä ratkaisuja eri tavalla toteutetuille .NET-ohjelmistoille. Google Cloud Platformin tarjonta .NET-sovelluksille oli rajallisempi, mutta sopivien palveluiden löytyessä vastaa myös GCP samoihin tarpeisiin kuin AWS ja Azure.

Työssä muodostettiin Cloudstep-malliin pohjautuva päätöksentekomalli, jolla pyritään valitsemaan sopiva migraatiostrategia, pilvipalvelutyyppi ja pilvipalveluntarjoaja. Mallissa muodostetaan profiilit organisaatiotasosta, ohjelmistotasosta ja mahdollisista pilvipalveluista. Näiden profiilien avulla arvioidaan organisaatiotason, tekniset ja taloudelliset rajoitteet. Rajoitteiden kautta saadaan rajattua se joukko ratkaisuja, joita voidaan harkita pilvimigraation toteuttamiseen. Sopivan strategian löydyttyä arvioidaan eri palvelutyyppejä ja valitaan lopulta ratkaisulle sopiva pilvipalvelualusta. .NET-ohjelmistojen näkökulmasta kartoitettiin eri pilvipalveluntarjoajien sopivuus eri toteutustavoille.

Legacy-ohjelmistojen tapauksessa on yleensä kannattavaa panostaa migraatiovaiheessa refaktorointiin, jos ohjelmistoa ollaan todennäköisesti käyttämässä vielä pitkään. Ensinnäkin on hyvin mahdollista, että täysin ilman refaktorointia ei pilvimigraatio edes ole mahdollinen. Toisekseen, koska ohjelman pilveen siirtämiseksi joudutaan joka tapauksessa tekemään muutoksia, on kannattavaa vähintäänkin selvittää mahdollisuudet samalla nykyaikaistaa ohjelmistoa ja taata sille pidempi käyttöikä jatkossa. Varsinkin taloudelliset rajoitukset voivat estää mahdollisuudet refaktoroinnille, jolloin voidaan tukeutua edullisempiin ja suoraviivaisempiin tapoihin migroida sovellus pilvipalveluun, mutta tällöin ei saavuteta pilvinatiiviuden mukanaan tuomia etuja. Saavutettavat hyödyt voivat silti olla riittävät perustelemaan pilvimigraation.

LÄHTEET

[1] *.NET Core 3.0 on Lambda with AWS Lambda's Custom Runtime*. Amazon Web Services. Section: .NET. 24. lokakuuta 2019. URL: https://aws.amazon.com/blogs/developer/net-core-3-0-on-lambda-with-aws-lambdas-custom-runtime/ (viitattu 11. 10. 2020).

[2] *.NET introduction and overview*. URL: https://docs.microsoft.com/en-us/dotnet/core/introduction (viitattu 11. 10. 2020).

[3] *.NET on Windows Server platform history - AWS Elastic Beanstalk*. URL: https://docs.aws.amazon.com/elasticbeanstalk/latest/platforms/platform-history-dotnet.html (viitattu 11. 10. 2020).

[4] *2019 CNCF Survey results are here: Deployments are growing in size and speed as cloud native adoption becomes mainstream*. Cloud Native Computing Foundation. URL: https://www.cncf.io/blog/2020/03/04/2019-cncf-survey-results-are-here-deployments-are-growing-in-size-and-speed-as-cloud-native-adoption-becomes-mainstream/ (viitattu 11. 10. 2020).

[5] *5 principles for cloud-native architecture—what it is and how to master it*. Google Cloud Blog. Library Catalog: cloud.google.com. URL: https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it/ (viitattu 27. 07. 2020).

[6] *Amazon EventBridge | Event Bus | Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/eventbridge/ (viitattu 11. 10. 2020).

[7] *App Service | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/services/app-service/ (viitattu 11. 10. 2020).

[8] *AWS Lambda – Pricing*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/pricing/ (viitattu 27. 09. 2020).

[9] *Azure Functions Serverless Compute | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/services/functions/ (viitattu 11. 10. 2020).

[10] Balalaie, A., Heydarnoori, A. ja Jamshidi, P. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software* 33.3 (2016). Publisher: IEEE, 42–52. ISSN: 0740-7459. DOI: 10.1109/MS.2016.64.

[11] Balalaie, A., Heydarnoori, A. ja Jamshidi, P. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. *arXiv.org* (2015). Yhteistyössä P. Jamshidi. URL: http://search.proquest.com/docview/2083225800/?pq-origsite=primo (viitattu 08. 03. 2020).

[12] Bellenger, D., Bertram, J., Budina, A., Koschel, A. ja Serowy, C. Scaling in Cloud Environments (), 6.

[13] Beserra, P. V., Camara, A., Ximenes, R., Albuquerque, A. B. ja Mendonça, N. C. Cloudstep: A step-by-step decision process to support legacy application migration

to the cloud. *2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*. 2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA). ISSN: 2326-6937. Syyskuu 2012, 7–16. DOI: 10.1109/MESOCA.2012.6392602.

[14] *Cloud Functions Overview | Cloud Functions Documentation*. Google Cloud. Library Catalog: cloud.google.com. URL: https://cloud.google.com/functions/docs/concepts/overview (viitattu 18. 07. 2020).

[15] *Create a Windows Server container on an AKS cluster by using Azure CLI - Azure Kubernetes Service*. URL: https://docs.microsoft.com/en-us/azure/aks/windows-container-cli (viitattu 27. 09. 2020).

[16] ESDS. *Cloud Computing – Types of Cloud*. URL: https://www.esds.co.in/blog/cloud-computing-types-cloud/ (viitattu 11. 10. 2020).

[17] Gannon, D., Barga, R., Sundaresan, N., Goasguen, S., Gustaffson, N., Davis, C., Subramanian, B. ja Kohn, D. An Asynchronous Panel Discussion: What Are Cloud-Native Applications? *IEEE Cloud Computing* 4.5 (2017). Publisher: IEEE, 50–54. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250941.

[18] *GDPR and the impact on cloud computing | Cyber Security | Privacy*. Deloitte Netherlands. URL: https://www2.deloitte.com/nl/nl/pages/risk/articles/cyber-security-privacy-gdpr-update-the-impact-on-cloud-computing.html (viitattu 11. 10. 2020).

[19] *GE Oil & Gas Case Study*. Amazon Web Services, Inc. Library Catalog: aws.amazon.com. URL: https://aws.amazon.com/solutions/case-studies/ge-oil-gas/ (viitattu 16. 04. 2020).

[20] gewarren. *Version compatibility in .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/version-compatibility (viitattu 11. 10. 2020).

[21] ggailey777. *Run background tasks with WebJobs - Azure App Service*. URL: https://docs.microsoft.com/en-us/azure/app-service/webjobs-create (viitattu 11. 10. 2020).

[22] ggailey777. *Supported languages in Azure Functions*. URL: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages (viitattu 11. 10. 2020).

[23] *Google App Engine Documentation | Google Cloud*. URL: https://cloud.google.com/appengine/docs (viitattu 11. 10. 2020).

[24] *How to explain vertical and horizontal scaling in the cloud - Cloud computing news*. URL: https://www.ibm.com/blogs/cloud-computing/2014/04/09/explain-vertical-horizontal-scaling-cloud/ (viitattu 27. 09. 2020).

[25] Jose, S. *What it means to be Cloud-Native approach — the CNCF way*. Medium. 5. huhtikuuta 2020. URL: https://medium.com/developingnodes/what-it-means-to-be-cloud-native-approach-the-cncf-way-9e8ab99d4923 (viitattu 06. 09. 2020).

[26] Kharenko, A. *Monolithic vs. Microservices Architecture*. Medium. 10. huhtikuuta 2018. URL: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59 (viitattu 11. 10. 2020).

[27] *Kubernetes Services: AWS vs. Azure vs. Google Cloud*. Cloud Academy. Section: Kubernetes. 12. marraskuuta 2019. URL: https://cloudacademy.com/blog/kubernetes-services-aws-vs-azure-vs-google-cloud/ (viitattu 11. 10. 2020).

[28] *Kubernetes vs. PaaS*. Datica. URL: https://datica.com/blog/kubernetes-vs-paas (viitattu 22. 09. 2020).

[29] *Lift and Shift: An Essential Guide*. 27. tammikuuta 2020. URL: https://www.ibm.com/cloud/learn/lift-and-shift (viitattu 11. 10. 2020).

[30] *Microsoft SQL Server on Amazon RDS - Amazon Relational Database Service*. URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_SQLServer.html (viitattu 11. 10. 2020).

[31] Orban, S. *6 Strategies for Migrating Applications to the Cloud*. Medium. Library Catalog: medium.com. 6. huhtikuuta 2018. URL: https://medium.com/aws-enterprise-collection/6-strategies-for-migrating-applications-to-the-cloud-eb4e85c412b4 (viitattu 12. 03. 2020).

[32] Perry, Y. *Cloud Migration Approach: Rehost, Refactor or Replatform?* URL: https://cloud.netapp.com/blog/cvo-blg-cloud-migration-approach-rehost-refactor-or-replatform (viitattu 11. 10. 2020).

[33] Perry, Y. *Lift and Shift: Business Benefits, Planning and Execution*. URL: https://cloud.netapp.com/blog/what-is-a-lift-and-shift-cloud-migration (viitattu 11. 10. 2020).

[34] Purohit, R. Comparative Analysis of few Cloud Service Providers Considering Their Distinctive Properties. *International Journal of Advanced Research in Computer Science; Udaipur* 8.5 (toukokuu 2017). Num Pages: 1908-1916 Place: Udaipur, India, Udaipur Publisher: International Journal of Advanced Research in Computer Science, 1908–1916. DOI: http://dx.doi.org.libproxy.tuni.fi/10.26483/ijarcs.v8i5.4018. URL: http://search.proquest.com/docview/1912632236/abstract/2EEF4381FC354F58PQ/1 (viitattu 11. 10. 2020).

[35] *Refactoring Applications for Cloud Migration: What, When, And How - DZone Cloud*. dzone.com. URL: https://dzone.com/articles/refactoring-applications-for-cloud-migration-what (viitattu 11. 10. 2020).

[36] *Refactoring Home Page*. URL: http://refactoring.com (viitattu 11. 10. 2020).

[37] Rick-Anderson. *Dependency injection in ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection (viitattu 11. 10. 2020).

[38] *Run Windows Server containers on GKE*. Google Cloud Blog. URL: https://cloud.google.com/blog/products/containers-kubernetes/run-windows-server-containers-on-gke/ (viitattu 27. 09. 2020).

[39] *SaaS in 2020: Growth Trends & Statistics – BMC Blogs*. URL: https://www.bmc.com/blogs/saas-growth-trends/ (viitattu 11. 10. 2020).

[40] Savage, N. Going serverless. *Communications of the ACM* 61.2 (2018). Publisher: ACM, 15–16. ISSN: 0001-0782. DOI: 10.1145/3171583.

[41] *Serverless (FaaS) vs. Containers - when to pick which?* serverless. URL: https://serverless.com/blog/serverless-faas-vs-containers (viitattu 11. 10. 2020).

[42] Singh, V. ja Peddoju, S. K. Container-based microservice architecture for cloud applications. *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017 International Conference on Computing, Communication and Automation (ICCCA). Toukokuu 2017, 847–852. DOI: 10.1109/CCAA.2017.8229914.

[43] *Support Ending for the .NET Framework 4, 4.5 and 4.5.1*. .NET Blog. 9. joulukuuta 2015. URL: https://devblogs.microsoft.com/dotnet/support-ending-for-the-net-framework-4-4-5-and-4-5-1/ (viitattu 11. 10. 2020).

[44] Suryavanshi, N. *What are Cloud Computing Services [IaaS, CaaS, PaaS, FaaS, SaaS]*. Medium. 9. marraskuuta 2017. URL: https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e (viitattu 27. 09. 2020).

[45] *The Most Popular Databases 2019 | Blog*. Explore Group. 21. huhtikuuta 2019. URL: https://www.explore-group.com/blog/the-most-popular-databases-2019/bp46/ (viitattu 11. 10. 2020).

[46] *Understanding serverless cold start*. URL: https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/ (viitattu 24. 09. 2020).

[47] wadepickett. *Entity Framework documentation*. URL: https://docs.microsoft.com/en-us/ef/ (viitattu 11. 10. 2020).

[48] *What is a Legacy System? - Talend*. Talend Real-Time Open Source Data Integration Software. URL: https://www.talend.com/resources/what-is-legacy-system/ (viitattu 27. 09. 2020).

[49] *What is ASP.NET Core? A cross-platform web-development framework*. Microsoft. URL: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core (viitattu 11. 10. 2020).

[50] *What is AWS Elastic Beanstalk? - AWS Elastic Beanstalk*. URL: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html (viitattu 11. 10. 2020).

[51] *What is Containers as a service (CaaS)?* 28. heinäkuuta 2020. URL: https://www.ibm.com/services/cloud/containers-as-a-service (viitattu 06. 10. 2020).

[52] *What is IaaS? Infrastructure as a Service | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/overview/what-is-iaas/ (viitattu 27. 09. 2020).

[53] *What is Kubernetes?* Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (viitattu 11. 10. 2020).

[54] *What is Serverless?* Serverless Stack. 23. joulukuuta 2016. URL: https://serverless-stack.com/chapters/what-is-serverless.html (viitattu 11. 10. 2020).

[55] *Why should you use microservices and containers?* IBM Developer. 14. syyskuuta 2020. URL: https://developer.ibm.com/depmodels/microservices/articles/why-should-we-use-microservices-and-containers/ (viitattu 11. 10. 2020).

[56] *Windows containers - Amazon Elastic Container Service*. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_Windows.html (viitattu 27. 09. 2020).

[57] Zimmermann, O. Architectural refactoring for the cloud: a decision-centric view on cloud migration. *Computing. Archives for Informatics and Numerical Computation; Wien* 99.2 (helmikuu 2017), 129–145. ISSN: 0010485X. DOI: http://dx.doi.org.libproxy.tuni.fi/10.1007/s00607-016-0520-y. URL: http://search.proquest.com/docview/1865256164/abstract/EED4129421F4DB3PQ/1 (viitattu 01. 03. 2020).