Jatkuvan kehityksen prosessi GameRefinery SaaS-palvelulle

Ilmari Huovinen
ilmari.huovinen@helsinki.fi

Department of Computer Science, University of Helsinki

Helsinki 2.6.2019

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Ilmari Huovinen ilmari.huovinen@helsinki.fiDepartment of Computer Science, University of Helsinki

Työn nimi — Arbetets titel — Title

Jatkuvan kehityksen prosessi GameRefinery SaaS-palvelulle

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	2.6.2019	

Tiivistelmä — Referat — Abstract

Jatkuva kehitys on joukko ohjelmistokehitysmenetelmiä, jotka mahdollistavat julkaisujen tekemisen luotettavasti ja tiheään tahtiin. Jatkuva kehitys sisältää useita menetelmiä, mutta niistä kolme laajasti tunnettua ovat jatkuva integraatio, jatkuva toimitus ja jatkuva julkaisu. Jatkuvassa integraatiossa muutokset integroidaan jatkuvasti yhteiseen koodikantaan, jatkuvassa toimittamisessa muutokset toimitetaan jatkuvasti tuotantoa muistuttavaan ympäristöön ja jatkuvassa julkaisussa muutokset julkaistaan jatkuvasti.

GameRefinery on pelialan yritys, joka kehittää mobiilipelien analyysi- ja markkinadataa tarjoavaa SaaS-palvelua. Palvelun kasvaessa on huomattu haasteita sen kehittämisessä ja ylläpidossa. Tämän tutkielman tarkoituksena selvittää mitä kehitykseen ja julkaisuun liittyviä ongelmia GameRefinery SaaS:ssa on koettu, ja sunnitella jatkuvan kehityksen menetelmiä käyttävä prosessi korjaamaan koettuja ongelmia.

GameRefinery SaaSin kehitykseen ja julkaisuun liittyviä ongelmat etsittiin tutkimalla GameRefinery SaaSin aiempia versioita ja selvittämällä mitkä niiden ratkaisut haittasivat ohjelmiston kehitystä ja julkaisua ja mitkä tukivat niitä. Tämän jälkeen verrattiin eroja GameRefinery SaaSin versioiden kehitys- ja julkaisuprosesseissa. Lopuksi eriteltiin löydetyt kehitykseen ja julkaisuun liittyvät ongelmat. Ongelmia löydettiin versiohallintakäytännöistä, toimitus- ja julkaisuprosessista ja virheistä palautumisessa. Lisäksi huomattiin, että arkkitehtuurinen ratkaisu siirtää eräajot pois mikropalveluista omiksi projekteikseen aiheutti ongelmia julkaisujen tekemisessä. Löydettyjä ongelmia ratkaistiin suunnittelemalla jatkuvan kehityksen prosessi, joka perustui Jenkins-automaatiopalvelimen käyttöönottamiseen ja jatkuvan kehityksen menetelmiä hyödyntävän automaatioputken toteuttamiseen.

Suunniteltu prosessi selvensi version- ja haaranhallinta käytäntöjä, korjaten ongelmia julkaisuversioiden kasaamisessa eri ominaisuuksista ja estämällä keskeneräisen koodin pääsemisen julkaisuhaaraan. Uudessa prosessissa myös automatisoitiin toimitusprosessi niin, että tietomallin muokkaus otettiin osaksi automaatioputkea poistaen aikaisemmin manuaalisesti suoritetun vaiheen toimituksessa. Näiden lisäksii esitettiin mahdolliset ongelmatilanteet virheestä palautumisessa ja tapoja korjata niitä. Uusi prosessi ei kuitenkaan onnistunut korjaamaan eräajojen siirtämisestä aiheutuneita ongelmia, vaikkakin eräajojen mukaan ottaminen automaatioputkeen lievensi niitä.

Avainsanat — Nyckelord — Keywords
continuous development, saas, micro services

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Jatkuva kehitys (eng. continuous development) on joukko ohjelmistokehitysmenetelmiä, jotka mahdollistavat julkaisujen tekemisen luotettavasti ja tiheään tahtiin[31]. Jatkuva kehitys koostuu useammasta menetelmästä, kuten jatkuvasta integraatiosta, jatkuvasta toimituksesta ja jatkuvasta julkaisusta. Jatkuvassa integraatiossa kehittäjien koodi integroidaan jatkuvasti, jolloin jokaisella kehittäjällä on käytössään viimeinen versio koodista. Jatkuvassa toimituksessa kaikki koodin muutokset viedään tuotantoa muistuttavaan testausympäristöön, jossa se testataan. Jatkuvassa julkaisussa kaikki testausympäristöön onnistuneesti viedyt muutoksen viedään automaattisesti myös tuotantoon. Jatkuvan kehityksen menetelmien käyttöönottaminen vaatii koodin koonti- ja julkaisuprosessin laajamittaista automatisointia, joten se vaatii kehitystiimiltä jonkin verran vaivaa. Tämän helpottamiseksi on luotu useita työkaluja, arkkitehtuureja ja menetelmiä.

GameRefinery on pelialan yritys, joka tuottaa SaaS-palvelua, mikä tarjoaa analyysidataa mobiilipeleistä ja mobiilipelimarkkinoista. SaaS, eli etäsovelluspalvelu, tulee sanoista software as a service. Se on ohjelmiston jakelutapa, jossa sovellusta käytetään kehittäjän omassa ympäristössä ja jossa käyttäjän ei tarvitse itse asentaa mitään. Tämä tuo omat haasteensa siihen miten palvelun saatavuus taataan suurelle määrälle käyttäjiä, jotka voivat olla maantieteellisesti kaukana toisistaan.

Gamerefinery SaaS:in käyttäjä- ja datamäärän sekä itse palvelun kasvaessa on koettu palvelun kehittämiseen ja ylläpitoon liittyviä haasteita. Tämä tutkielman tarkoituksena on selvittää mitkä GameRefinery SaaSin osa-alueet aiheuttavat haasteita ja ratkaista ne suunnittelemalla uusi jatkuvan kehityksen menetelmiä hyödyntävä kehitys- ja julkaisuprosessi. Samalla käydään läpi GameRefinery SaaSin kehitys ja syyt eri versioden välisiin muutoksiin.

GameRefinery SaaS on toteutettu käyttäen mikropalveluarkkitehtuuria. Mikropalveluarkkitehtuurissa palvelu on jaettu useaan erilliseen komponentiin, joita voidaan kehittää, julkaista ja skaalata itsenäisesti. Sen on nähty tukevan jatkuvaa kehitystä ja soveltuvan SaaS-palvelujen toteuttamiseen.

1.1 Tutkimusongelma

Gamerefinery SaaS-palvelun päivittämisestä halutaan tehdä mahdollisimman helppoa ja turvallista. Kun päivitysten tekeminen on helppoa, voidaan palvelua muuttaa ketterästi lyhyellä varoituksella. Myös virheiden korjaamisesta tulee helpompaa. Tähän sisältyy koodin lisäksi tietokannan rakenteen muuttaminen ja mikropalvelujen ja niiden tietokantojen palauttaminen virhetapauksissa. Näiden vaatimusten täyttämiseksi tarvitaan mahdollisimman pitkälle automatisoidut mekanismit palveluiden versioiden julkasemiselle ja niihin liittyviin datamallin muutoksien tekemiselle. Virheiden havaitsemiseksi ja rikkinäisten versoiden julkaisemisen estämiseksi julkaisujen tekemisen yhteyteen tarvitaan automatisoidut testit.

Tämän tutkielman tarkoituksena on suunnitella GameRefinerylle jatkuvan kehityk-

sen prosessi, joka pystyy näihin asioihin. Prosessin tulisi täyttää seuraavat vaatimukset:

1. Jatkuvaa integraatiota tukeva versionhallinta ja julkaisuprosessi.
    - Prosessin on tuettava eri ominaisuuksien samanaikaista kehittämistä, niin että niitä voidaan kehittää ja julkaista toisistaan riippumatta.
    - Prosessin on skaalauduttava suuremmalle kehitystiimille.
    - Mikropalveluulle on voitava julkaista yksitellen niin, ettei muutos yhteen palveluun vaadi muutoksia muihin.
2. Testaaminen ja laadunvarmistus.
    - Laadunvarmistusympäristön luominen ja ylläpitäminen.
    - Automaattinen testaus osana julkaisuprosessia.
    - Tietokantojen testaus.
3. Virheistä palautuminen.
    - Tietokantojen varmuuskopiointi ja niiden palautus varmuuskopioista.
    - Palvelujen palauttaminen virhetilanteessa.
4. Datamallin muutosten vieminen tuotantoon osana toimitusprosessia.

1.2 Tutkimusmenetelmät ja tutkimuksen eteneminen

Tutkielman tarkoituksena on suunnitella GameRefinerylle uusi jatkuvan kehityksen menetelmiä käyttävä prosessi soveltamalla design science-tutkimusmenetelmää. Ensin tutkielmassa käydään läpi GameRefinery SaaS-palvelun eri versiot, miten ne eroavat toisistaan ja syyt versioissa tehtyihin muutoksiin. Versiot ja niissä tehdyt muutokset analysoidaan jatkuvan kehityksen näkökulmasta, eli miten niissä hyödynnettiin jatkuvan kehityksen menetelmiä ja miten versiot tukivat tai eivät tukeneet jatkuvaa kehitystä. Analyysin tulosten perusteella suunnitellaan uusi jatkuvan kehityksen prosessi, jonka tarkoitus on korjata puutteet nykyisen version prosessissa.

1.3 Työn rakenne

Tämä tutkielma koostuu seitsemästä luvusta. Ensimmänen on johdanto. Toinen luku on nimeltään Jatkuva kehitys, siinä selvitetään mitä menetelmiä jatkuvaan kehitykseen kuuluu ja niiden käyttämisen tuomia hyötyjä ja haasteita. Kolmas luku on Jatkuvan kehityksen työvälineet, jossa esitetään menetelmiä jotka edesauttavat jatkuvan kehityksen menetelmien käyttöönottamista. Neljännessä luvussa esitellään

GameRefinery SaaS-palvelu, sen kolme eri versiota ja mitä ongelmia niissä on esiintynyt. Viidennessä luvussa esitetään neljännen luvun löytöjen perusteella suunniteltu uusi jatkuvan kehityksen menetelmiä käyttävä prosessi. Kuudennessa luvussa esitetään menetelmä viidennessä luvussa suunnitellun prosessin arvioimiseen. Seitsemännessä luvussa annetaan yhteenveto tutkielmasta.

2 Jatkuva kehitys

Jatkuva kehitys, tai jatkuva ohjelmistokehitys, on joukko ohjelmointikehitysprosesseja ja -käytäntöjä, joden tarkoituksena on mahdollistaa uusien ominaisuuksien toteuttamisen ja julkaisemisen nopeissa sykleissä[31][18]. Periaatteessa mikä tahansa asia, jota tehdään jatkuvasti ohjelmistokehityksessä, voidaan laskea jatkuvan kehityksen menetelmäksi, kuten jatkuva testaus[15]. Jatkuvassa testauksessa testit on automatisoitu ja ne suoritetaan jokaiselle koodimuutokselle ilman kehittäjältä vaadittavia toimia[30]. Eri menetelmien karttuessa niitä on kuitenkin käytännöllistä käsitellä laajempina prosesseina, jotka sisältävät eri määrän jatkuvan kehityksen menetelmiä. Tässä tutkielmassa keskitytään kolmeen yleiseen jatkuvan kehityksen mentelmään: jatkuvaan integraatioon (eng. continuous integration), jatkuvaan toimittamiseen (eng. continuous delivery) ja jatkuvaan julkaisemiseen (eng. continuous deployment).Tässä luvussa käydään läpi mitä jatkuva integraatio, jatkuva toimittaminen ja jatkuva julkaisu ovat, minkälaisia menetelmiä pitävät sisällään, mitä haasteita niiden käyttöönottamiselle on ja mitä hyötyjä niitä käyttöönottamalla voidaan saavuttaa.

2.1 Jatkuvan kehityksen keskeiset menetelmät

Tässä osiossa esitellään menetelmiä ja käytäntöjä joita tarvitaan jatkuvan integraation, jatkuvan toimittamisen ja/tai jatkuvan julkaisun käyttöönottamiselle onnistuneesti.

2.1.1 Versionhallinta

Versionhallinta tarkoittaa projektin lähdekoodin ja muiden tiedostojen muutosten säilyttämistä. Yleensä tämä hoidetaan jollain versionhallintajärjestelämällä. Versionhallintajärjestelmät mahdollistavat helpon koodin jakamisen kehitsytiimin kesken, projektin muutoshistorian tarkastelemisen ja muutosten kumoamisen[32]. Versionhallinnan käyttäminen on välttämätöntä, jotta kehitystiimi voi ylläpitää yhteistä koodikantaa (eng. code base). Versionhallintaan tulee tallentaa kaikki projektin tarvitsemat asiat, kuten lähdekoodi, testit, tietokantaskeemat ja eri kokoamisja julkaisuskriptit. Kaikki, mitä tarvitaan projektin kokoamiseen, julkaisemiseen ja testaamiseen, tulee löytyä versionhallinnasta[18]. Olisi myös hyvä, että kehitystiimi päättää jonkinlaisen haaranhallintastrategian. Muuten pysyminen selvillä siitä missä on vanhaa tai uutta koodia ja mitkä haarat on jo liitetty päähaaraan.

2.1.2 Kokoamisen automatisointi

Kokoamisprosessin tulee olla automatisoitu. Toisin sanoen ohjelmiston pitää pystyä kokoamaan, testaamaan yhdellä komennolla[18]. Monet IDE:t pystyvät hoitamaan tämän prosessin, mutta tarvitaan myös koontiskripti, joka suorittaa tämän

prosessin niitä ympäristöjä varten, joissa ei voida käyttää IDE:tä, esimerkiksi sovelluspalvelimella[16][18]. Toinen syy, miksi koontiskriptit ova tärkeitä on se, että projektin kasvaessa sen kokoamisesta ja julkaisemisesta tulee moninmutkaisempaa. Kokoamisprosessin muuttuminen vaatii koontiskriptin muuttamista. Tällöin on tärkeää, että skriptin on hyvin ylläpidetty ja testattu, mikä ei ole mahdollista IDE:tä käytettäessä[18].

2.1.3 Automaattinen testaus

Automaattinen testaus liittyy läheisesti kokoamisen automatisointiin, sillä testien ajaminen kuuluu koontiprosessiin. Testien automatisointi eroaa kuitenkin suuresti kokoamisen automatisoinnista. Testien automatisointiin kuuluu kolme eri tasoa: yksikkötestit, integraatiotestit ja hyväksymistestit. Yksikkötesteillä tarkoitetaan testejä jotka kohdistuvat pieneen kokonaisuuteen koodia, kuten metodiin tai pieneen joukkoon metodeja[18]. Integraatiotestit puolestaan testaavat useiden komponenttien toimintaa, joihin voi sisältyä sovelluksen ulkoisia komponentteja, kuten tietokanta tai jokin järjestelmä[18]. Hyväksysmistestit testaavat sitä, täyttääkö sovellus sille asetetut vaatimukset. Vaatimuksia voivat olla sovelluksen tarjoama toiminnallisuus tai jokin muu ominaisuus, kuten saatavuus tai turvallisuus[18]. Jatkuvassa kehityksessä hyväksymistestit nähdään yleensä viimeisenä tarkistuksena version julkaisukelpoisuudesta[26]. Yksikkö- ja integraatiotestit voidaan suorittaa ilman sovelluksen käynnistämistä, mutta hyväksymistestit on hyvä suorittaa koko sovellusta vastaan tuotantoa muistuttavassa ympäristössä[18].

2.1.4 Automaatioputki

Automaatioputki (eng. deployment pipeline) on automatisoitu tapa toimittaa ja testata sovellus eri ympäristöihin[19]. Automaatioputki koostuu useasta tasosta joissa luodaan ja testataan omat koontiversionsa[19]. Automaatioputket ovat keskeinen osa jatkuvaa integraatiota, toimittamista ja julkaisua, sillä niissä on ideana että jokainen muutos koodikantaan laukaisee automaattisen integroimisen, toimittamisen tai julkaisun prosessin, riippuen mitä prosessia käytetään.

Tyypillinen julkaisuun asti menevä putki koostuu seuraavista osista[27]:

1. Kehittäjä liittää muutoksen yhteiseen koodikantaan.
2. Luodaan uusi koontiversio, jolle suoritetaan joukko testejä.
3. Kootut binäärit siirretään testiympäristöön.
4. Suoritetaan hyväksymistestit testiympäristössä.
5. Versio julkaistaan tuotantoon.

Yllä kuvailtu prosessi on automatisoitu, joten kehittäjiltä vaaditaan toimenpitäitä vain virheen sattuessa[27]. Lisäksi on hyvä huomioida, että kuvailtu putki on

yksinkertaistettu. Testaus mainitaan vain kaksi kertaa, mutta testitasoja voi olla enemmän. Testejä ei välttämättä tarvitse suorittaa tietyssä järjestyksessä ja joitain niistä voidaan suorittaa rinnakkain. Tämän takia testien jakaminen useampiin tasoihin on suositeltavaa. Sillä säästää aikaa ja se helpottaa koontiversion etenimesen seuraamista automaatioputkessa[19].

On myös suositeltavaa, että uudelleenkokoamisen sijaan binäärit kootaan automaatioputkessa vain kerran, ja siirtää binääri eri teasteaustasoihin ja ympäristöihin automaatioputkea edetessä[19]. Tavoitteena on, että alussa koottujen binäärien avulla voidaan julkaista sovellus kaikissa eri ympäristöissä. Tällä varmistetaan se, että testataan sitä mikä oikeasti päätyy tuotantoon[19]. Jotta samat binäärit voidaan julkaista eri ympäristöissä on configuraatio ja binäärit pidettävä erillään toisistaan[19]. Binäärit jotka selviytyvät automaatioputken läpi kannattaa tallentaa jonnekin mistä niitä on helppo hakea. Tämän avulla eri ympäristöissä olevia versioita voi vahtaa helposti, esimerkiksi eri versioiden vertailua varten tai palvelun palauttamisen aikaisempaan versioon virheen sattuessa[19].

2.2 Jatkuva integraatio

Jatkuva integraatio on yleisin jatkuvista menetelmistä[15]. Se sai alkunsa osana Extreme Programming -prosessia yhtenä sen kahdestatoista perusmenetelmästä[16]. Sen suosion takia sille on annettu useita eri määritelmiä[15]. Sen keskeinen idea on muutosten jatkuva integroiminen yhteiseen koodikantaan ja koodin kokoaminen ja testaaminen jokaisen muutoksen yhteydessä, jonka avulla pyritään varmistamaana, että yhteisessä koodikannassa oleva versio on aina toimivassa tilassa[18].

Jatkuvassa integraatiossa muutokset viedään repositorion päähaaraan (eng. main branch) mahdollisimman useasti. Ennen muutosten viemistä päähaaraan tulee kehitääjän ensin testata kokomaisprosessi omassa ympäristössään[18]. Jos kokoaminen ja testit menevät onnistuneesti läpi, voi muutokset liittää yhteiseen koodikantaan. Päähaaran muutoksen tulisi automaattisesti laukaista muutetun version kokoaminen ja testaus integraatiokoneella. Jos tässä vaiheessa ilmenee virheitä, tulee ne korjata heti[16]. Rikkinäinen koontiversio haittaa kaikkien tiiminjäsenten työskentelyä, koska silloin kukaan ei voi integroida päähaaraan versiota omaan koodiinsa tai toisinpäin. Jos virhettä ei saada korjattua nopeasti, voi siihen liittyvien muutosten kumoaminen päähaarassa olla nopein ratkaisu[16]. Sitten kun virheet on korjattu kehittäjän omassa ympäristössä voidaan muutosten integrointia yrittää uudelleen. Tätä jatketaan kunnes kokoaminen ja testaus onnistuvat integraatiokoneella. Integraatioympäristön tulee olla mahdollisimman lähellä tuotantoympäristöä kaikkien mahdollisten ongelmien löytämiseksi[16]. Koonti ja testausprosessi on tärkeää pitää mahdollisimman nopeana. Liian pitkään kestävä kokoaminen ja testaus voivat saada kehittäjät viemään muutoksensa päähaaraan ilman, että he kokoavat ja testaavat muutetun version omassa ympäristössään ensin[18]. Lisäksi liian pitkään kestävä kokoaminen ja testaus aiheuttaa sen, että yhden koontiversion aikana voi päähaaraan olla ehtinyt tulla useata muutoksia. Jos koontiversio, joka sisältää usean kehittäjän muutoksia menee rikki, voi virheen aiheuttaneen muutoksen löytäminen olla vaike-

aa[18].

Yllä olevan perusteella voidaan nähdä, että jatkuvan integraation putki koostuu yksinkertaisimmillaan koontiversion luomisesta ja testien suorittamisesta. Käytännöllisyyden vuoksi nämä on kuitenkin hyvä jakaa tasoihin. Ensimmäisellä tasolla suoritetaan ns. commit-testit, joiden tarkoitus on testata, että päähaarassa olevan version kehittämistä voi jatkaa turvallisesti.[16]. Tämän tason suorittamiseen pitäisi mennä muutamia minuutteja, jotta integraatioprosessin kesto ei kasva liian suureksi[18]. Tämän jälkeiset tasot suoritetaan silloin, kun niihin on mahdollisuus ja niiden epäonnistuminen ei johda yllä kuvattuun version korjausprosessiin[16].

2.3 Jatkuva toimittaminen

Jatkuvassa integraatiossa koodi integroidaan jatkuvasti yhteiseen kodikantaan niin, että kaikilla kehittäjillä on aina käytössä viimeisin yhtenäinen versio projektista. Jatkuva toimittaminen laajentaa tätä edelleen siten, että koodikanta on aina julkaisuvalmis tuotantoon[31][26]. Jatkuva toimittaminen siis vaatii jatkuvan integraation käyttämistä. Lisäksi testaamista on laajennettava niin, että voidaan luottaa siihen, että jokaisen koontiversion voi julkaista tuotantoon. Tämä vaatii hyväksymistestien automatisoimista funktionaalisille ja ei-funktionaalisille vaatimuksille[18]. Lisäksi hyväksymistestit tulee suorittaa mahdollisimman tuotannon kaltaisessa ympäristössä[12] [18][31].

Jatkuvan toimituksen putken ensimmäinen taso on samanlainen kuin jatkuvalla integraatiolla. Sovellus kootaan ja sille ajetaan nopeat testit. Lisäksi ensimmäiseen tasoon kannattaa lisätä staattista analyysiä ei-funktionaalisten laatuvaatimusten testaamiseksi[18]. Mahdollisia testattavia asioita voivat olla esimerkiksi testikattavuus, toistuva- tai kuollut koodi ja koodaustyyli. Staattisen analyysin tuloksia tulee käsitellä kuin testien tuloksia. Jos niille annetut vaatimukset eivät täyty, niin ensimmäinen testaustaso epäonnistuu. Kun jatkuvan integraation sisältämät koonti- ja testaustasot on suoritettu, julkaistaan versio tuotantoa vastaavaan ympäristöön, jossa sille suoritetaan hyväksymistestit[18]. Jotta testausympäristö saadaan mahdollisimman lähelle tuotantoa pitää varmistaa, että: 1. infrastruktuuri on sama, esim. verkkotopologia ja palomuurin asetukset, 2. käyttöjärjestelmä on sama, 3. sovelluspino (eng. application stack) on sama ja sovelluksen data on oikeassa tilassa [18]. Sovelluksen datan oikeassa tilassa pitäminen vaatii sovelluksen käyttämien tietokantojen datan päivittämistä tarvittaessa. Tämän automatisointi vaatii tietokantoja päivittävien skriptien liittämistä jatkuvan toimittamisen putkeen.

Jatkuvan toimittamisen putki eroaa jatkuvan integraation putkesta siten, että siinä kaikki testaustasot suoritetaan putkeen ilman taukoja[18]. Ja toisin kuin jatkuvassa integraatiossa, myöhempien tasojen epäonnistuminen tarkoittaa, että versio on korjattava heti[18]. Jatkuvan toimittamisen tarkoitushan on se, että jokainen versio on julkaisuvalmis, mikä edellyttää kaikkien testien läpäisemistä. Koska integraatioja hyväksymistestit voivat kestää pitkään, tarvitaan tapa testata useita muutoksia samanaikaisesti. Luomalla jokaiselle muutokselle oma jatkuvan toimittamisen putki

-instanssi voidaan testata useita muutoksia samanaikaisesti[18]. Resurssien säästämiseksi tulee putken suorittaminen lopettaa, jos mikä tahansa sen osio epäonnistuu.

2.4 Jatkuva julkaisu

Siitä miten jatkuva julkaisu ja jatkuva toimittaminen eroavat toisistaan on käyty kiivasta väittelyä tiede- ja teollisuuspiireissä[31]. Jatkuva julkaisu eroaa jatkuvasta toimittamisesta siten, että siinä jokainen päähaaran muutos julkaistaan automaattisesti tuotantoon osana jatkuvan julkaisun putkea[15]. Jatkuvan julkaisun putki on siis pitkälti samanlainen kuin jatkuvassa toimittamisessa, paitsi versio julkaistaan tuotantoon kunhan se on läpäissyt viimeiset testit.

Jatkuvassa julkaisussa on voitava olla täysin varma jokainsen muutoksen toimivuudesta, sillä siinä ei ole minkäänlaista manuaalisen testauksen mahdollisuutta. Jatkuvassa toimittamisessa voidaan tehdä esimerkiksi tutkivaa testausta ennen tuotantoon julkaisemista. Tämä ei ole mahdollista jatkuvassa julkaisussa. Jatkuvan julkaisun käyttöönottamiseksi pitää pystyä olemaan varma, että tuotanto ei mene rikki päivitysten seurauksena. Tämän saavuttamiseksi automaattisten testien tulee olla erinomaisia[18]. Yksikkö-, integraatio- sekä funktionaalisten ja ei-funktionaalisten hyväksymistestien pitäisi kattaa koko sovellus[18].

2.5 Jatkuvan kehityksen haasteet

Jatkuvassa integraatiossa halutaan integroida muutokset yhteiseen koodikantaan mahdollisimman usein. Jatkuvassa toimittamisessa halutaan toimittaa toimiva versio sovelluksesta julkaisuvalmiiksi mahdollisimman usein. Jatkuvassa julkaisussa halutaan julkaista muutokset mahdollisimman usein. Ne siis eroavat siinä kuinka pitkälle prosessi muutoksen tekemisestä julkaisuun on automatisoitu. Vaikka ideat prosessien takana ovat yksinkertaisia, on jokaisen käyttöönottamisella omat haasteensa. Seuraavaksi listataan prosessikohtaisia haasteita ja niiden syitä. On hyvä huomioida, että jatkuvan julkaisun käyttöönotto vaatii jatkuvan toimittamisen käyttöönottoa, mikä puolestaan vaatii jatkuvan integraation käyttöönottoa. Tämän seurauksena haasteet jatkuvan integraation käyttöönotolle ovat myös haasteita jatkuvan toimituksen käyttöönotolle ja haasteet jatkuvan toimituksen käyttöönotolle ovat myös haasteita jatkuvan julkaisun käyttöönotolle.

2.5.1 Jatkuvan integraation haasteet

Yleinen ongelma jatkuvan integraation käyttöönotossa on kunnollisen testausstrategian puuttuminen[31]. Automaattinen testaus on tärkeää jatkuvan integraation käyttöönottamiselle, mutta kaikkien testitapausten automatisoiminen voi olla haastavaa. Manuaalisten testien automatisoinnin työläys, huono infrastruktuuri automaattiselle testaukselle ja riippuvuudet ohjelmiston ja laitteiston välillä ovat esteitä testien automatisoinnille[31]. Toinen yleinen testaukseen liittyvä ongelma on heikko

testien laatu[31]. Tämä voi johtua muunmuassa huonosta testikattavuudesta, testien epäluotettavuudesta ja testien pitkästä kestosta[31]. Testien huono laatu haittaa jatkuvan integraation putken toimintaa[31].

Liitoskonfliktit (eng. merge conflict) ovat myös mahdollinen haaste jatkuvan integraation käyttöönotossa. Ne voivat aiheuttaa pullonkaulan integraatiovaiheessa[31]. Kolmannen osapuolen komponentit ja komponenttien väliset riippuvuudet voivat aiheuttaa suuria ongelmia integraatiovaiheessa. Lisäksi ratkaisut joissa on paljon sidonnaisuuksia ovat yleinen liitoskonfliktien lähde[31].

2.5.2 Jatkuvan toimituksen haasteet

Koodin ja eri komponenttien väliset riippuvuudet voivat tulla haasteiksi jatkuvaa toimittamista käyttöönottaessa, jos riippuvuuksia ei hallita oikein[31]. Haasteet tulevat siitä, miten muutokset aiheuttavat sivuvaikutuksia, jotka vaativat lisää muutoksia, jotka puolestaan aiheuttavat lisää sivuvaikutuksia ja niin edelleen[31]. Tämä tulee ongelmaksi varsinkin silloin kun muutokset vaativat työtä useammilta tiimeiltä[31].

Tietokantojen skeemamutokset pitäisi olla tehokkaasti automatisoitu jatkuvassa toimituksessa. Jos skeemamuutoksia ei pystytä tekemään helposti, on huomattu, että tilanteet joissa tarvitaan muuttaa tietokantaskeemaa usein, tai joissa pienet koodinmuutokset vaativat tietokannan skeemamuutoksia tulevat ongelmallisiksi[31]. Ongelmat skeeman muuttamisessa voivat luoda pullonkaulan jatkuvan toimituksen prosessissa.

Kehitystiimien riippuvuus toisistaan voi myös luoda haasteita jatkuvassa toimituksessa. Useamman kehitystiimin työskentely samalla koodikannalla voi johtaa tilanteeseen, jossa tiimit eivät voi kehittää tai julkaista palveluja tai komponentteja tuotantoon itsenäisesti[31]. Tilanteissa, joissa tiimien välisiä riippuvuuksia ei voida estää, on tiimien välinen kommunikkatio ja koordinaatio erityisen tärkeää[31].

2.5.3 Jatkuvan julkaisun haasteet

Enneun kuin harkitaan jatkuvan julkaisun käyttöönottoa on hyvä huomioida, että se soveltuu huonosti tai ei ollenkaan jonkin tyyppisille järjestelmille tai joissain tilanteissa[31][23]. Esimerkiksi tilanteet joissa julkaisu pitää tehdä useamman asiakkaan ympäristöihin ovat ongelmallisia, koska se vaati eri julkaisukonfiguraation jokaiseen ympäristöön[31][15]. Esimerkki tästä on telekummonikaatiojärjestelmien julkaisu[15], jossa eri asiakkailla voi olla eri asetukset verkkolaitteissaan[23]. Myös järjestelmät, jotka eivät siedä keskeytyksiä ovat huono kohde jatkuvalle julkaisulle, kuten tehtaiden ohjaus- ja lennonjohtojärjestelmät[15][23]. Tehdas saatetaan joutua pysäyttämään päivityksen ajaksi, mikä laskee tuottavuutta ja lentoasemat tuskin kestäisivät jatkuvia päivityksistä johtuvia katkoja. Jatkuvan julkaisun käyttäminen voi olla mahdotonta tilanteissa, joissa sovellusta jaetaan jonkun kolmannen osapuolen palvelun, kuten jonkin sovelluskaupan, välityksellä. Sillä palvelulla voi kestää

pitkään ennen kuin se suostuu julkaisemaan päivityksen[23].

Riippuvuudet laitteistoon ja legacy-järjestelmiin voivat olla esteitä julkaisun automatisoinnille[31]. Jotta julkaiseminen voidaan automatisoida turvallisesti pitää varmistaa, että julkaisun tekemisessä ei ole integraatio-ongelmia julkaistavan järjestelmän ja sen riippuvuuksien välillä[31]. Jatkuvan julkaisun käyttöönotto legacyjärjestelmiin voi myös olla ongelmallista[23] [15]. Ongelmat voivat johtua esimerkiksi siitä, että legacy järjestelmiä ei ole alunperin suunniteltu automatisoitua testausta varten, mikä tekee tekee testien automatisoinnista vaikeaa[23].

Testien pitkä kesto on myös todettu haasteeksi jatkuvan julkaisun käyttöönottamiselle[22][23]. Nopeiden julkaisujen tekeminen tulee vaikeaksi, jos testien suorittaminen kestää liian kauan, varsinkin kun testit tehdään jokaisen muutoksen yhteydessä. Tällöin julkaisun kestoksi tulee vähintään testien suorittamiseen menevä aika[23].

2.6 Jatkuvan kehityksen hyödyt

Samoin kuin prosessien käyttöönoton haasteiden kanssa on hyödyissäkin hyvä muistaa, että jatkuva julkaisu vaatii jatkuvan toimittamisen käyttämistä, mikä puolestaan vaatii jatkuvan integraation käyttämistä. Toisin sanoen jatkuvaa julkaisua käyttämällä saavutetaan jatkuvan toimittamisen hyödyt ja jatkuvaa toimittamista käyttämällä jatkuvan integraation hyödyt.

2.6.1 Jatkuvan integraation hyödyt

Jatkuvan integraation on todettu parantavan kehittäjien tuottavuutta helpottamalla vianetsintää, muutosten liittämistä yhteiseen koodikantaan ja usean kehittäjän työskentelyä samassa koodikannassa[33]. Jatkuva integraatio myös parantaa projektin ennalta arvattavuutta auttamalla löytämään ongelmia aikaisessa vaiheessa[33]. Ongelmien aikainen löytäminen ja korjaaminen on myös tärkeää siksi, että jos virheitä ei löydetä, on niillä tapana kasaantua ja kasvaa suuremmiksi, vaikeammin korjattaviksi ongelmiksi. Ongelmien aikaisen löytämisen lisäksi jatkuva integraation avulla ei-funktionaalista järjestelmätestausta voidaan tehdä aikaisessa vaiheessa ja integraatio voidaan tehdä projektin kriittisen polun ulkopuolella, jotka edelleen parantavat projektin ennalta arvattavuutta[33]. Näiden lisäksi jatkuvan integraation on todettu parantavan kehittäjien ja kehitystiimien välistä kommunikaatiota[33].

2.6.2 Jatkuvan toimittamisen hyödyt

Jatkuvan toimittamisen on todettu tehostavan kehittäjien työtä vähentämällä testaukseen tarvittavan työn määrää testiympäristöjen pystyttämisen[12][35] automatisoinnilla. Tämän avulla säästetään aika, joka kuluisi manuaaliseen ympäristöjen pystyttämiseen[12] ja päästään eroon virheistä, jotka voivat johtua testiympäristön manuaalisesta pystyttämisestä[35].

Jatkuva toimittaminen myös vähentää julkaisuihin liittyviä riskejä ja parantaa julkaisujen luotettavuutta[12][35]. Kun julkaisuprosessiin voidaan luottaa, voidaan julkaisuja myös tehdä nopeammin. Jatkuvan toimittamisen onkin havaittu nopeuttavan julkaisujen tahtia merkittävästi[12]. Kun julkaisuja tehdään useammin, niin yksittäisen julkaisun koko pienenee, mikä helpottaa ongelmien löytämistä ja noupeuttaa palautteen saamista[12]. Lisäksi jatkuvan toimittamisen käyttäminen on todettu parantavan tuotteen laatua vähentämällä merkittävästi tuotantoon päässeiden virheiden määrää[12].

2.6.3 Jatkuvan julkaisun hyödyt

Ottaen huomioon, että jatkuva julkaisu muistuttaa hyvinkin pitkälle jatkuvaa toimittamista, ei ole yllättävää, että sen tuomat hyödyt eivät eroa suuresti jatkuvan toimittamisen hyödyistä. Jatkuvan julkaisun on kuitenkin todettu mahdollistavan nopean palautteen saamisen asiakkailta ja käyttäjiltä, mikä antaa kehittäjille paremman kuvan siitä mitä ominaisuuksia käyttäjät tarvitsevat ja mitkä ominaisuudet käyttäjät kokevat hyödyllisiksi[23]. Jatkuva palaute puolestaan mahdollistaa vaihtoehtoisten ratkaisujen ratkaisujen kokeilemisen, mikä helpottaa päätöstä siitä minkä ominaisuuksien kehittämistä jatketaan ja minkä kehitys lopetetaan[23].

Ominaisuuksien julkaiseminen heti niiden valmistuttua on koettu tuovan arvoa asiakkaalle tuomalla konkreettisia tuloksia nopeammin ja mahdollstaen nopean reagoinnin asiakkaiden palautteeseen[23]. Jatkuvat julkaisut myös auttavat asiakkaita pysymään ajan tasalla kehityksen tilasta[23].

3 Jatkuvan kehityksen työkalut

Jatkuvan kehityksen prosessit voivat olla hyvinkin moninmutkaisia ja vaatia laajaa automatisointia. Tässä luvussa käydään läpi teknologioita ja tekniikoita, jotka ovat tarpeellisia tai hyödyllisiä jatkuvassa kehityksessä.

3.1 Staattinen analyysi

Perinteinen testaaminen, missä ohjelma suoritetaan ja tarkastellaan sen toimintaa on dynaamista analyysia[24]. Staattisessa analyysissä tarkastellaan sovelluksen lähdekoodia tai jotain muuta staattista artifactia ilman sovelluksen käynnistämistä[24][29]. Stattista analyysia voidaan käyttää monenlaisten laatuominaisuuksien seuraamiseen, kuten ohjelmointivirheiden löytämiseen, ohjelmointityylin tarkastamiseen ja arkkitehtuurin eheyden testaamiseen[24][21]. Kuten luvussa 2 mainittiin, käytetään jatkuvan kehityksen menetelmissä usein staattista analyysia ei-funktionaalisten laatuvaatimusten tarkastamiseen.

On olemassa lukuisia staattiseen analyysiin käytettäviä työkaluja, mutta suurin osa niistä jakaa tietyt perusominaisuudet[24]. Ne lukevat ohjelman käyttäen lähdekoodia tai jotain muuta staattista artifaktia ja rakentavat siitä mallin jota työkalut voivat käyttää erilaisten patternien tarkistamiseen[24]. Esimerkiksi sääntö: "sisennykset on tehtävä tabilla"on yksi mahdollinen patterni. Sen avulla voidaan löytää missä päin koodia on sisennetty väärin.

Toinen yleinen menetelmä mitä staattisen analyysin työkalut käyttävät on tietovuoanalyysi[24]. Tietovuoanalyysillä voidaan selvittää mitä arvoja muuttujat voivat saada missäkin osassa ohjelmaa[24]. Tämän avulla voidaan löytää muunmuassa kuollutta koodia ja alustamattomia tai käyttämättömiä muuttujia. Tietovuoanalyysi on erityisen hyödyllistä havoittuvuuksia etsittäessä. Aina kun ohjelma vastaanottaa syötteen käyttäjältä on mahdollisuus, että käyttäjä pystyy vahingoittamaan järjestelmää syötteen avulla[24]. Siksi on tärkeää, että käyttäjien syötteiden kulkua järjestelmän sisällä voidaan seurata[24].

3.2 Mikropalveluarkkitehtuuri

Perinteisesti ohjelmistot ovat olleet monoliitteja, eli niillä on ollut yksi koodikanta, joka sisältää kaiken ohjelmistokoodin, ja joka tarjoaa kaikki ohjelmiston tarjoavat palvelut[34]. Monoliittisten ohjelmistojen ongelmana on se, että niiden kehittäminen ja ylläpitäminen vaikeutuu ohjelmiston kompleksisuuden kasvaessa[34]. Monoliittisessa ohjelmistossa uutta toiminnallisuutta lisättäessä, tai vanhaa muutettaessa, on huolehdittava, että muutokset eivät riko ohjelmiston mitään muuta osaa. Tämän seurauksena jokaisen version kanssa joudutaan tekemään enemmän työtä vanhan toiminnallisuuden toimivuuden testaamiseksi kuin aiemmassa versiossa. Monoliittisten ohjelmistojen julkaisu voi myös aiheuttaa ongelmia. Ohjelmisto joudutaan julkaisemaan aina kokonaisuudessaan, joten julkaisun epäonnistuessa koko järjes-

telmä on käyttökelvoton[34]. Lisäksi monoliittisten palvelujen skaalaaminen on ongelmallista, koska ei voida skaalata vain niitä palvelun osia jotka tarvitsevat lisää resursseja, vaan on skaalattava koko palvelua[27].

Monoliittinen arkkitehtuuri tuo täten haasteita jatkuvan kehityksen menetelmiä käytettäessä. Mikropalveluarkkitehtuuri (eng. micro service architecture) on yksi tapa ratkaista näitä haasteita. Mikropalveluarkkitehtuurissa järjestelmä koostuu useasta mikropalvelusta, jotka ovat itsenäisesti kehitettäviä, testattavia ja julkaistavia komponentteja[27]. Mikropalvelujen koon hallitsemiseksi ja niiden sisäisen koheesion säilyttämiseksi mirkopalveluilla tulisi olla tarkasti määritellyt vastuualueet. Ja jos näyttää siltä, että yksi mikropalvelu tekee liikaa asioita kannattaa se hajoittaa useammaksi mikropalveluksi.

Mikropalveluarkkitehtuurin tarkoituksena on helpottaa ohjelmistojen kehittämistä, toimittamista ja julkaisemista ja se nähdään edesauttavan jatkuvan kehityksen menetelmien käyttöönottamista[17]. Jotta mikropalveluita voitaisiin kehittää ja julkaista itsenäisesti, tulee niiden väliset riippuvuudet minimoida. Tämä tarkoittaa, että mikropalvelut eivät jaa lähdekoodia tai tietokantoja ja niiden tulee keskustella keskenään vain teknologia-agnostisten rajapintojen kautta[17].

Itsenäiset palvelut mahdollistavat sen, että jokaiselle palvelulle tehdään oma automaatioputki[17]. Itsesnäisesti julkaistavat palvelut myös nopeuttavat muutosten viemistä tuotantoon, sillä vain palvelu, johon on tehty muutoksia, on testattava ja julkaistava[17][27]. Itsenäiset automaatioputket myös mahdollistavat palveluiden julkaisun rinnakkain, koska jokaisella palvelulla on oma julkaisukanavansa[17].

Mikropalveuiden hajautetun luonteen takia niitä on vaikeampi hallita kuin perinteisia monoliittipalveluita[17].

3.3 Kontit

Kontti (eng. kontainer) on tapa luoda muusta järjestelmästä eristetty ympäristö[11]. Ne eroavat perinteisistä virtuaalikoneista, jotka eristävät virtualisoidun käyttöjärjestelmän eristämällä prosesseja[11]. Kontit siis jakavat käyttöjärjestelmän ja mahdollisesti binäärejä ja kirjastoja[11]. Tämä tekee konteista huomattavasti kevyempiä kuin virtuaalikoneista. Niiden keveyden ansiosta kontteja voi ylläpitää yhdellä koneella moninkertainen määrä virtuaalikoneisiin verrattuna[20][11]. Lisäksi konttien pystyttäminen on huomattavasti nopeampaa kuin virtuaalikoneiden[20].

Virtuaalikoneet kuitenkin tarjoavat paremman eristyksen tason kontteihin verrattuna, joten jos eristystä tarvitaan turvallisuussyistä, ovat virtuaalikoneet parempi ratkaisu[20][11]. Virtuaalikoneet ovat myös parempi ratkaisu silloin, kun yhdellä alustalla halutaan käyttää useita eri käyttöjärjestelmiä tai käyttöjärjestelmien versioita[11].

Konttien ominaisuudet tekevät niistä hyviä web-palvelujen toteuttamiseen. Kontteja voi siirtää paikasta toiseen, joten niiden julkaiseminen uuteen ympäristöön onnistuu helposti, eikä vaadi muutoksia itse konttiin[20]. Konttien keveys ja pystyttä-

misen helppous tekevät palvelujen skaalaamisesta helppoa. Niitä voidaan pystyttää ja ajaa alas tarpeen mukaan[20]. Tämän ansiosta kontit soveltuvat erityisen hyvin pilviympäristöissä ylläpidettävien palveluiden skaalaamiseen[20].

Kontit soveltuvat hyvin myös mikropalvelujen toteuttamiseen. Ne helpottavat mikropalvelujen hallitsemista ja julkaisua ja mahdollistavat mikropalvelujen helpon skaalaamisen. Konttien ja mikropalveluarkkitehtuurin avulla voidaankin rakentaa tehokkaita automaatioputkia ja niiden kombinaatio tehostaa jatkuvan kehityksen menetelmiä[27].

On olemassa monia erilaisia konttiratkaisuja, mutta tämän työn kannalta merkittäviä teknologioita ovat Docker ja Kubernetes, koska niitä käytetään GameRefinery SaaSissa. Docker on konttiratkaisu ja Kubernetes on teknologia Docker -kontti klustereiden hallitsemiseen[11].

Docker hyödyntää LXC:tä, eli Linux -konttia, mikä käyttää Linux -ytimen ominaisuuksia kuten cgroup (control group) ja nimiavaruutta prosessien eristämiseen ja resurssien hallintaan[20]. Docker -kontti sisältää kuvatiedoston (eng. image), joka sisältää kaikki sovelluksen tarvitsemat komponentit sovelluksen suorittamiseen. Dockerin kuvatiedostot koostuvat kerroksista, joista jokainen vastaa käskyä kuvan dockertiedostossa[3]. Kun kuva ladataan konttiin, niin kontti lisää yhden kerroksen kuvatiedostossa olevian staattisten kerrosten päälle. Kaikki muutoksen kuvatiedoston sisältämiin tiedostoihin tehdään tähän viimeiseen kerokseen. Näin kuvatiedosto pysyy muuttumattomana ja sitä voidaan käyttää useassa kontissa.

Kuberneteksen avulla voidaan hallita konteista koostuvia sovelluksia[20]. Se poistaa kytköksen Docker -konttien ja niitä ylläpitävän systeemin välillä[11]. Kubernetes tekee tämän abstraktoimalla yksittäiset Docker kontit ja käsittelee niitä yhtenäisenä ryhmänä, jolle se jakaa resursseja[11]. Ryhmän konttien ei tarvitse sijaita samalla fyysisellä koneella, vaan Kubernetes antaa ryhmälle oman ip-osoitteen, johon mikä tahansa klusterin ryhmä voi ottaa yhteyttä[11].

3.4 SaaS

SaaS (suom. etäsovelluspalvelu, eng. software as a service) on ohjelmiston jakelutapa, jossa ohjelmiston tuottaja ylläpitää palvelua omassa ympäristössään ja laskuttaa sen käytöstä[25], toisin kuin perinteisessä ohjelmistonjakelussa, jossa käyttäjä maksaa lisenssistä ja asentaa ohjelmiston omaan ympäristöönsä. Käyttäjät pääsevät käyttämään SaaS palvelua internetin tai jonkin muun tietokoneverkon välityksellä[25].

Termillä SaaS ei ole vakiintunutta, yleisesti hyväksyttyä määritelmää. Eri määritelmillä on kuitenkin toistuvia piirteitä, joiden kautta SaaSin keskeiset ideat tulevat esille. Nämä yleisesti toistuvat piirteet ovat:[25]

1. Ohjelmiston käyttäminen verkkoselaimen kautta.
2. Ohjelmistoa ei kehitetä erikseen jokaiselle asiakkaalle.

3. Asiakkaan ei tarvitse asentaa ohjelmistoa tai mitään sen osaa.
4. Ohjelmisto ei vaadi erillistä integrointia tai asentamista.
5. Ohjelmiston hinnoittelu riippuu sen käytöstä.
6. Monivuokralaisuus (eng. multitenancy), eli se että käyttäjät käyttävät samaa sovellusympäristöä.

SaaSissa palveluntarjoaja siis tarjoaa asiakkaille internetin kautta käytettävää palvelua, jossa ei ole asiakkaalle asennettavia komponentteja ja jossa asiakkaat jakavat ympäristön. Vain yhden ympäristön ylläpitäminen helpottaa ohjelmistokehitystä siten, ettei yksityisten asiakkaiden vaatimuksia tarvitse ottaa huomioon ja siten, että ylläpidettävänä on vain yksi versio.

SaaS tuo myös omat haasteensa. Palvelua on ylläpidettävä 24 tuntia päivässä 7 päivää viikossa ja sen on kestettävä suuria yhdenaikaisia käyttäjämääriä. Ohjelmointivirheiden korjausten on oltava mahdollisimman nopeaa ja versiopäivitysten ei pitäisi haitata käyttäjien kykyä käyttää palvelua.

4 GameRefinery SaaS

GameRefinery SaaSin tarkoituks on auttaa mobiilpeliyhtiöitä suunnittelemaan pelien ominaisuuksia. Palvelu sisältää julkisesti saatavilla olevaa dataa mobiilipeleistä, yhtiön omaa analyysidataa ja näistä kahdesta jalostettua dataa. Dataa kerätään ja jalostetaan eteenpäin jatkuvasti.

Tässä luvussa käydään läpi GameRefinery SaaSin eri versiot, miten palvelu on kehittynyt versiosta toiseen, miten jatkuva integraatio menetelmiä on käytetty (tai onko niitä käytetty) ja mitä ongelmia versioilla on ollut. Versioden tarkastelu on jaettu viiteen osioon. Osiot ovat: pilvi, versionhallinta, toimittaminen ja julkaisu, tietokanta ja datan päivitys. Pilvi-osio sisältää ympäristön, missä GameRefinery SaaS mikropalveluita ylläpidetään. Versionhallinnassa käydään läpi miten versionhallinta oli toteutettu. Toimittaminen ja julkaisu -osiossa kuvaillaan toimittamisen ja julkaisun prosessit. Tietokanta osiossa kerrotaan, mitä tietokantaa käytettiin. Datan päivittäminen -osiossa käydään läpi, miten versiossa toteutettiin eräajot, joilla dataa kerättiin ja päivitettiin, sekä miten data synkronoitiin palvelujen välillä. Datan päivitys puuttuu ensimmäisestä versiosta, sillä se ei sisältänyt eräajoja tai palvelimia joiden välillä tarvitsisi synkronoida dataa. Lopuksi tehdään lyhyt yhteenveto siitä, miten eri versiot ovat toteuttaneet mitkäkin jatkuvan integraation osa-alueet ja mitä hyviä ja huonoja puolia ratkaisuilla on ollut.

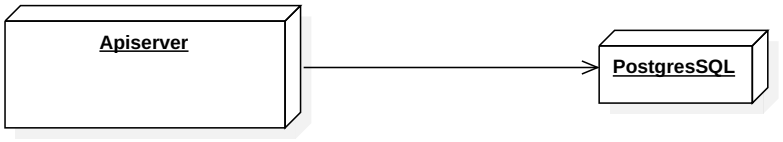
4.1 1. Versio

GameRefinery SaaS-palvelun ensimmäinen versio oli olemassa vain vähän aikaa, eikä sitä ikinä julkaistu asiakkaiden käyttöön. Se oli enemmänkin hahmotelma siitä, minkälainen palvelusta tulisi myöhemmin kuin oikea ohjelmistojärjestelmä. Tämän version back end koostui vain yhdestä PHP:llä toteutetusta palvelusta ja PostgreSQL tietokannasta.

Ensimmäinen versio eroaa myöhemmistä siten, että se on täysin ulkopuolisten kehittäjien tekemä. Tämä aiheutti kommunikaatio-ongelmia GameRefineryn ja kehitystiimin välillä. Palvelua kehitettiin sprinteissä, mikä tarkoitti, että GameRefineryn ja kehitystiimit edustajat kommunikoivat pääasiassa kerran sprintissä. Tämän seurauksena paljon aikaa kului vajavaisesti kuvailtujen tai väärinymmärrettyjen ominaisuuksien korjaamiseen. Kehitys kärsi myös siitä, että kehittäjä halusi vaihtaa käytettyjä teknologioita tiheään tahtiin, mikä vaati merkittävän määrän jo valmiin toiminnallisuuden uudelleentekemistä.

4.1.1 Pilvi

Ensimmäisessä versiossa käytettiin Amazonin AWS OpsWorksiä palvelun ylläpitämiseen. AWS OpsWorks on pilvialusta muunmuassa virtuaalipalvelimien ylläpitämiselle. Sen peruskomponentti on pino (stack) joka on kokoelma muita AWS resursseja, kuten AWS EC2 virtuaalipalvelin tai AWS Relational Database Service tietokanta



Kuva 1: Backend arkkitehtuuri GameRefinery SaaS:in ensimmäisessä versisossa

instansseja[2]. AWS OpsWorks tarjoaa myös muita palvelinten ylläpitämiseen liittyviä ominaisuuksia, kuten kuormantasauksen[2].

Sitä ei pystytty selvittämään, mitä AWS OpsWorksin ominaisuuksia hyödynnettiin GameRefinery SaaS:in ensimmäisessä versiossa. Tietoa käytetyistä AWS OpsWorksin asetuksista ei löytynyt, eikä ympäristöä ole enää olemassa.

4.1.2 Versionhallinta

Ensimmäisessä GameReFinery SaaS-versiossa käytettiin Gittiä versionhallintaan. Repositoriota tutkimalla ei onnistuttu löytämään, käytettiinkö siellä minkäänlaista haaranhallintamallia tai tapaa erottaa versiot toisistaan. Vaikuttaa siltä, että palvelua ei versioitu mitenkään tai edes sitä, missä haarassa pidettiin palvelun kehitettävää versiota.

4.1.3 Toimittaminen ja julkaisu

GameRefiner SaaS:in ensimmäisen version julkaisuprosessi koostui kolmesta osasta: Ensin palvelu koottiin skriptillä, sitten palvelun koontiversio ladattiin Amazon S3:een toisella skriptillä ja viimeiseksi palvelu julkaistiin Amazon OpsWorksiin OpsWorksin konsolia käyttämällä. Viimeisessä vaiheessa konsolissa jouduttiin tekemään vielä jotain valintoja liittyen julkaisuun.

Ensimmäisen version jatkuvan integraation prosessista ei tiedetä tätä enempää, koska tämän version kehitys oltiin ulkoistettu. Kaikki, mitä asiasta tiedetään tulee ensimmäisen version kehittäjien vähäisestä dokumentaatiosta. Dokumentaation perusteella palvelun julkaisu oli virhealtista.

4.1.4 Tietokanta

PostgreSQL on yksi yleisimmin käytetyistä reaaliotietokannoista. PostgreSQLn laajan suosion takana on se, että se on avointa lähdekoodia ja kaikkien vapaasti käytettävissä, sekä sen pitkä kehitysaika. PostgreSQLää on kehitetty yli 30 vuotta [7].

PostgreSQL on pitkän kehitysaikansa ja laajan käyttöönoton seurauksena hyvin tunnettu ja dokumentoitu ja siitä löytyy paljon materiaalia joka tekee sen käyttämisestä helpompaa kuin vähemmän tunnettujen tietokantojen.

Relaatiotietokantana PostgreSQL takaa vahvan ACID tuen ja monipuolisen kyselyjen hallinnan. Tämän hintana kuitenkin on ylimääräinen työ, mikä tulee tietokantaskeeman suunnittelemisesta ja muuttamisesta. Relaatiokannan toimivuuden kannalta on tärkeää, että sen skeema on suunniteltu oikein. Tämä hidasti GameRefinery SaaSin kehitystä, koska harvoin osattiin sanoa suoraan, minkälaista ja missä muodossa dataa haluttiin tallentaa. Tämä johti useisiin datamallin muutoksiin lyhyen ajan sisällä.

Relaatiotietokannat on suunniteltu strukturoidun datan varastoimiseen. Strukturoitu data on sellaista joka seuraa datamallia, missä on määritelty, miltä datan tulee näyttää, kuten esimerkiksi kenttien nimet ja datan tyyppi. Kun data noudattaa datamallia, niin relaatiotietokanta pystyy tallentamaan ja hakemaan sitä tehokkaasti.

Gamerefinery SaaSissa kuitenkin haluttiin tallentaa merkittäviä määriä strukturoimatonta ja semistrukturoitua dataa, mihin PostgreSQL soveltui huonosti.

4.2 2. Versio

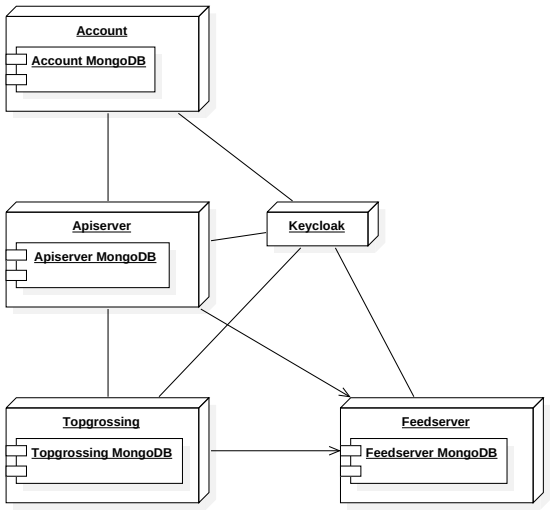
Palvelun toisessa versiossa siirrytiin PosgresSQL relaatiokannasta MongoDB nosql-kantaan. Projektin ohjelmointikieli vaihdettiin Javaan. Lisäksi kehitys siirrettiin GameRefineryn sisäiselle kehitystiimille. Tässä versiossa palvelu koostui neljästä mikropalvelusta, joista kaikki oli ylläpidetty Openshift V2:ssa.

Kehityksen siirtäminen yrityksen sisäiselle tiimille paransi tehokkuutta huomattavasti. Nyt uusia ominaisuuksia kehitettäessä niistä voitiin saada reaaliaikaista palautetta, mikä vähensi turhan työn määrää. Lisäksi ohjelmistokehittäjät ymmärsivät paremmin palvelun tarpeita, koska he olivat jatkuvassa kontaktissa palvelun käyttäjiin.

Suurin motivaatio backend-kielen vaihtamiseksi Javaan oli uuden kehitystiimin aikaisempi kokemus kielestä. Javaan vaihtaminen auttoi myös back- ja frontend toiminnallisuuden parempaan erottelemiseen. Aikaisemmassa versiossa back- ja frontendin raja oli häilyvämpi. Javalla on myös kattava REST-tuki, mikä mahdollisti GameRefinery SaaSin REST-toiminnallisuuden toteuttamisen käyttämällä vain Javan omia kirjastoja. Lisäksi backendiin kuuluvat eräajot ja komentorivityökalut oli helpompi toteuttaa Javalla.

4.2.1 Pilvi

Openshift on Paas (eng. Platform-as-a-Service) palvelu, joka tarjoaa alustan virtuaalipalvelimille. Openshift v2 sisälsi kahdenlaisia kontteja: kasetteja (eng. cartridge) ja rattaita (eng. gear). Kasetit sisälsivät sovelluksen lähdekoodin, sovelluksen käyttämät ulkoiset kirjastot ja sovelluksen kokoamismekanismit [6]. Rattaat puolestaan



Kuva 2: Backend-arkkitehtuuri GameRefinery SaaS:in toisesssa versiossa

olivat kontteja, joille oli allokoitu prosessoreja, muistia ja levytilaa. Rattaat toimivat siis ympäristöinä joissa voitiin ylläpitää kasetteja. Yksi ratas pystyi sisältämään esimerkiksi sovelluspalvelimen, tietokantapalvelimen tai molemmat.

Palvelua ylläpidettäesssä liitettiin sen lähdekoodin sisältävä repositorio Openshiftiin. Palvelun toimittaminen toimi siten, että Openshift latasi sovelluksen lähdekoodin, kokosi sovelluksen ja julkaisi sen konttiin.

GameRefineryn tässä versiossa jokaisella palvelulla oli käytössään kaksi ratasta. Toisella ylläpidettiin sovelluspalvelinta ja toisella palvelimen käyttämää tietokantaa.

4.2.2 Versionhallinta

Ensimmäisen version tapaan toisessa versiossa käytettiin Gittiä versionhallintaan. Pääasiassa ominaisuudet kehitettiin omissa haaroissan ja liitettiin kehityshaaraan siinä vaiheessa, kun ne olivat valmiita testattavaksi staging-ympäristössä. Eri julkaisut versioitiin tekemällä haara, johon liitettiin julkaisuun tulevien ominaisuuksien haarat. Näin tehtiin sen takia, että kehityshaarassa saattoi sisältää koodia, jota ei haluttu julkaistavaan versioon. Lopuksi version haara liitettiin päähaaraan ja päähaara kehityshaaraan, jotta kehityshaara ei varmasti jäisi jälkeen päähaarasta. Staging- ja tuotanto- ympäristöjen koodikannat pidettiin erillään pitämällä molemmille omia repositorioitaan.

4.2.3 Toimittaminen ja julkaisu

Toimittaminen toteutettiin kahdella eri tavalla: viemällä koodia projektin päähaaraan tai Openshiftin rhc- terminaalityökalun avulla. Openshift oli konfiguroitu niin,

että se julkaisi päähaaran aina päähaaran päivittyessä. Rhc-työkalun avulla voitiin tehdä deployment myös muista haaroista. Tätä ominaisuutta käytettiin pääasiassa viemään jokin tietty haara stagingille testaukseen. Julkaisu tuotantoon tehtiin aina viemällä stagingin päähaara tuotannon päähaaraan.

Julkaisua tehdessä stagingin versiohaara mergettiin stagingin päähaaraan. Tämän jälkeen koontiversio teastattiin staging ympäristössä. Jos virheitä ei löydetty, liitettiin stagingin päähaara tuotanto repositorion päähaaraan. Lopuksi julkaisu testattiin tuotannossa.

Virhetilanteessa oli mahdollista palauttaa vanha versio tuotantoon yliajamalla tuotanto-repositorion päähaara vanhalla versiolla tai luomalla tuotannon repositorioon oma haara vanhalle versiolle. Käytännössä näin ei kuitenkaan tehty, vaan kaikki virheet tuotannossa korjattiin korjaamalla viimeisintä staging-repositoriossa olevaa versiota ja toimittamalla korjaukset hotfiksinä normaalin julkaisuprosessin mukaisesti.

Datamallin muutoksille ei ollut omaa mekanismia. Datamallin muuttuessa piti palvelussa pitää huoltokatko ja ajaa muutokset manuaalisesti, joko skripteillä tai suoraan kantaan. Sama datamallin muutoksia kumotessa aiempaan versioon palauttamisen yhteydessä. Tämän seurauksena datamallin muutokset olivat työläitä ja riskialttiita, koska virheiden korjaaminen oli työlästä ja aikaavievää, eikä korjaamisen aikana palvelua ei voinut käyttää.

Kaikkien palvelujen tietokannoista tehtiin varmuuskopiot päivittäin, joten pahimmassa tapauksessa tietokannan pystyi palauttamaan varmuuskopiosta. Varmuuskopiosta palauttaminen tosin vaati usein vanhemman palveluversion palauttamisen ja julkaisuprosessin uudelleenaloittamisen, joten sen tekemistä pyrittiin välttämään.

4.2.4 Tietokanta

Toisessa versiossa sirryttiin PosgresSQL:stä MongoDB:seen. Tietokannat sijaitsivat samassa OpenShift rattaassa mikropalvelun kanssa. MongoDB on dokumenttipohjainen(eng. document-oriented) NoSql tietokanta[5]. Dokumenttipohjaisessa tietokannassa data on tallennettu dokumentteina, jotka sisältävät avain-arvo -pareja. MongoDBn käyttämät dokumentit ovat binäärikoodattuja JSON:ia muistuttavia olioita (BSON = binary JSON).

```
{
    "_id": ObjectId("507f1f77bcf86cd799439011"),
    "value1": "wasd",
    "value2": 0.91,
    "value3":  [1, 2, 3],
    "value4": false
}
```

Kuva 3: Esimerkki MongoDB dokumentista

MongoDB dokumenttien arvoina voi olla yleisesti käytettyjen datatyyppien (integer, long, String, jne) lisäksi taulukoita ja olioita. Tämän ansiosta MongoDBseen

voi tallentaa mielivaltaisen muotoisia olioita. MongoDBssä dokumentit kuuluvat kokoelmiin (eng. collection) vähän samalla tavalla kuin rivi kuuluu tauluun relaatiokannassa. Relaatiokannasta eroten MongoDBn kokoelmat ovat skeemattomia, minkä ansiosta niihin voi tallentaa minkälaisia olioita tahansa. Tallennetun datan oikeellisuuden vahtiminen jää siten tietokantaa käyttävän sovelluksen vastuulle.

GameRefinery SaaSissa siirryttiin PostgerSQL:stä sen takia, että GameRefinery SaaS:ssa käytetyn datan rakenne ei sopinut kunnolla relaatiotietokantaan, mikä aiheutti suorituskyvyn menetystä. Esimerkiksi kahdessa eri kantaoliossa voidaan tallentaa mielivaltaista dataa (datatyyppi Object). Tämä ei onnistu luontevasti relaatiotietokannassa.

Tietokantojen pitäminen mikropalvelujen kanssa samassa rattaassa hidasti mikropalveluja. Joihinkin mikropalveluihin kohdistui paljon datan luku- ja/tai päivitysoperaatioita, jolloin tietokanta vei suurimman osan rattaan resursseista ja muodostui pullonkaulaksi. Tämä oli ongelma erityisesti joidenkien raskaampien eräajojen kanssa. Tietokantojen vaarmuuskopiointi hoidettiin päivittäisellä eräajolla, joka otti tietokannoista dumpit ja tallensi ne pysyvään säilöön.

4.2.5 Datan päivittäminen

GameRefinery SaaS:in toinen versio käytti useita eräajoja datan päivittämiseen. Useimmat eräajot suoritettiin kerran päivässä, joitain tosin suoritettiin pitkin päivää. Eräajojen ongelmana oli se, että ne olivat osa mikropalveluja ja täten kuormittivat palvelimia. Eräajojen käynnissä ollessa palvelu hidastui huomattavasti. Pahimmillaan palvelu hidastui niin paljon, että jotkut sen toiminnoista tulivat käyttökelvottomiksi. Erityisen ongelmallisia olivat eräajot, jotka päivittivät enemmän kuin yhden palvelun dataa. Koska eräajot sijaitsivat aina tietyllä palvelimella, päivittivät useamman palvelun dataa käsittelevät eräajot toisia palveluja REST-rajapinnan avulla.

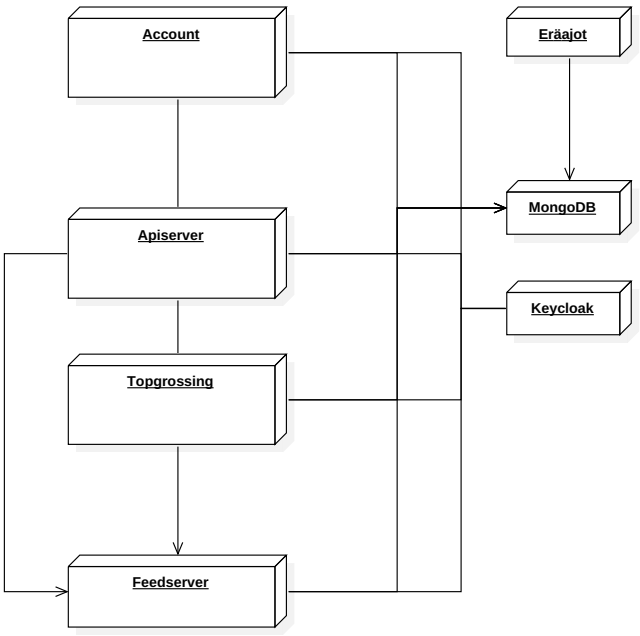
REST:in kautta tapahtuva datan synkronointi lakkasi olemasta käytännöllistä synkronoitavan datan määrän kasvaessa. Suuren datamäärän prosessointi oli raskasta ja aikaa vievää. Tämän lisäksi palvelin joka lähetti dataa toiselle palvelimelle joutui odottamaan toisen palvelimen vastausta voidakseen suorittaa eräajo loppuun.

Tämä kuormitti palvelia varsinkin, jos palvelut vaihtoivat suuren määrän dataa keskenään silloin kun jokin muu eräajo oli käynnissä. Gamerefinery SaaS:in toisen version loppua kohden palvelun käyttäminen oli huomattavasti hitaanpaa eräajojen käynnissä ollessa. Palvelu hidastui välillä niin paljon, että sen käyttämisestä tuli vaikeaa.

Openshift V2:n http kyselyn siirtomekanismilla oli sellainen ominaisuus, joka palautti gateway timeout virheen, jos palvelimella kesti liian kauan vastata kyselyyn. Tämä rikkoi palvelinten välisen synkronoinnin suurilla datamäärillä, jos REST-kutsun lähettänyt palvelin tarvitsi toisen palvelimen vastauksen.

4.3 3. Versio

Palvelun toisen version suurin ongelma oli se, että se ei skaalautunut hyvin suurille data ja käyttäjämäärille. Kolmannessa versiossa pyrittiin parantamaan palvelun suorituskykyä siirtämällä eräajot omalle palvelimelleen ja siirtymään ulkoiseen keskitettyyn tietokantaan. Samalla palvelinten ylläpito siirrettiin Openshiftin kolmanteen versioon ja MongoDB versioon 3.6.4. Tietokanta siirrettiin Openshiftistä MongoDB inc:in tarjoamaan Atlas -palveluun.



Kuva 4: Backend arkkitehtuuri GameRefinery SaaS:in kolmannessa versiossa

4.3.1 Pilvi

Openshiftin kolmas versio siirtyi pois rataspohjaisista konteista Docker -kontteja sisältäviin Kubernetes -kapseleihin (eng. pod). Openshift V3:ssa palvelu koostui kapseleista, jotka olivat pienin hallittava komponentti[4][6]. Kapselit puolestaan sisälsivät yhden tai useamman Docker -kontin[6]. Kapselin sisältämät kontit jakavat nimiavaruuden ja muistialueen (volume). Kapselit ovat karkeasti ottaen Openshift V3:n vastine Openshiftin toisen version rattaisiin[6] ja kapselien sisältämät kontit kasetteihin.

Kapselit mahdollistavat palvelujen helpon skaalaamisen. Palvelun voi säätää lisäämään kapselien määrää tietyn rasitusasteen ylitettäessä. Tällöin Openshift pystyt-

tää uuden Doker kontin ja kopioi sinne Docker kuvan, joka sisältää suoritettavan sovelluksen. Sovelluksen käynnistyttyä Openshift ohjaa sinne osan liikenteestä. Tämä prosessi ei vaadi minkäänlaisia toimenpiteitä ohjelmoijalta.

Samantyylisesti uuden version julkaiseminen tapahtuu käyttäjien huomaamatta. Uutta versiota julkaistaessa Openshift joko hakee ja rakentaa sovelluksen lähdekoodin, tai sille annetaan valmiiksi rakennettu binääri. Tämän jälkeen Openshift pystyttää sovellukselle osoitetun määrän kapseleita ja toimittaa niihin sovelluksen uuden version. Uusien kapseleiden valmistuessa vanhat kapselit tuhotaan ja niihin kohdistuva liikenne ohjataan uusiin. Tämä on suuri parannus verrattuna Openshiftin toisen version hot deploymenttiin, joka aiheutti muistivuotoja Java -sovellusten kanssa käytettäessä.

4.3.2 Versionhallinta

Versionhallinnassa jatkettiin Gitin käyttöä, mutta eri julkaisuille ei enää tehty omia haarojaan, eikä julkaisuja enää merkitty Gittiin. Haarojen käyttöä ei hallittu juuri ollenkaan, vain muutoksia tehtiin suoraan päähaaraan. Versiot arkistoitiin OpinShiftissä, joka säilytti sinne toimitetujen versioiden binäärit. Tämä vaikeutti versioiden löytämistä silloin, kun haluttiin selvittää missä versiossa jokin tietty ominaisuus julkaistiin. Piti selata projektin Git-muutoshistoriaa löytääskeen milloin muutos tehtiin ja etsittävä OpenShiftistä tätä seuraavan version koontiversio. Tätä ei onneksi tarvinnut tehdä kuin hyvin harvoin.

4.3.3 Toimittaminen ja julkaiseminen

Sovellusten julkaisuprosessia muutettiin siten, että jokainen julkaisu on aina ensin julkaistava stagella ennen kuin sen pystyi julkaisemaan tuotantoon. Tämä toteutettiin siten, että Openshift laitettiin rakentamaan ja julkaisemaan stagella uusi versio aina projectin päähaaran päivittyessä. Version pystyi viemään tuotantoon vain merkitsemällä Openshiftissä oleva koontiversio versionumerolla ja toprod -merkinnällä. Tämä siirsi staging-ympäristössä olevan Docker -kuvan tuotannon vastaavaan Openshift -palveluun.

Koska julkaisujen koontiversiot arkistoitiin OpenShiftiin, voitiin virhetilanteista palautua julkaisemalla jokin vanhempi versio, kunnes uusi saatiin korjattua ja julkaistua staging-ympäristöön. Tämän jälkeen korjattu versio julkaistiin normaalin julkaisuprosessin mukaisesti. Tämä prosessi oli ongelmallinen silloin, kun tuotannosta löytyi virhe jonkin aikaa julkaisun jälkeen ja stagin-ympäristössä oli uusi versio testissä. Koska staging-ympäristöön pystyi julkaisemaan vain viemällä koodia päähaaraan ja tuotantoon voitiin julkaista vain staging-ympäristössä olevan version.

Usein tällaisessa tilanteessa yksinkertaisin ratkaisu oli tehdä stagella testissä oleva toiminnallisuus loppuun ja julkaista korjaukset osana seuraavaa julkaisua. Varsinkin sillon, kun virheelliseen versioon liittyi datamallin muutoksia. GameRefinery SaaS:in kolmannessa versioon ei kuulunut tapaa datamallin muuttamiseen aikaisem-

paan versioon. Minkä takia datamallin palauttaminen virhetilanteessa oli hankalaa.

Tämä ongelma ratkaistiin useimmiten niin, että datamallia muutettiin vain lisäämällä luokkia tai kenttiä olemassa oleviin luokkiin. Silloin palvelun pystyi palauttamaan aikaisempaan versioon ja se vain jätti huomioimatta kentät, mitä ei löytynyt kyseisen version tietokantaolioluokista.

Tämä puolestaan aiheutti vanhojen kenttien kasautumisen luokkiin, mikä teki luokista vaikeaselkoisia ja tarpeettoman suuria. Luokkia ei myöskään refaktoroitu kuin harvoin, koska muutosten palauttaminen oli työlästä ja riskialtista.

Kolmannessa versiossa jatkettiin tietokantojen päivittäistä varmuuskopiointia, joten tietokanta pystyttiin aina palauttamaan varmuuskopiosta. Varmuuskopiosta palautettaessa kuitenkin menetetään varmuuskopion jälkeen luodun datan, jonka takia se ei ole hyväksyttävä ratkaisu tilanteessa, jossa virhe huomataan vasta useamman päivän jälkeen. Varmuuskopiot on myös otettu eri hetkinä, joten on todennäköistä, että ne eivät ole konsistentteja toistensa kanssa. Tämä ei kuitenkaan ole suuri ongelma, sillä Gamerefinery SaaSin mikropalvelut on suunniteltu synkronoituvan toistensa kanssa dynaamisesti, joten suurimmassa osassa tapauksissa palvelu pääsee konsistenttiin tilaan 24 tunnin sisällä.

Suurempi ongelma on se, että palvelussa lasketaan jatkuvasti uutta dataa eräajojen avulla. On siis lähes varmaa, että varmuuskopio tehdään samanaikaisesti jonkin eräajon kanssa, jolloin osa varmuuskopioidusta datasta on palautushetkellä ajankohtaista ja osa vanhentunutta. Vanhentuneen datan voi saada ajantasalle vain ajamalla keskeytyneen eräajon uudelleen, mutta tämä johtaa duplikaattidatan kertymiseen siinä datassa, minkä eräajo oli jo ehtinyt prosessoida.

4.3.4 Tietokanta

Kolmannessa versiossa jatkettiin MongoDBn käyttöä, mutta tietokannat siirrettiin pois mikropalvelujen palvelimilta. Ne siirrettin MongoDB inc:in tarjoamaan Atlas palveluun. Tämä nopeutti mikropalveluja, koska niiden ei enää tarvinnut jakaa resursseja tietokantojen kanssa. Atlaksen käyttöönottaminen myös paransi tietokantojen suorituskykyä tarjoamalla suuremmat laskentaresurssit ja helpon tavan skaalata tietokantoja sirpaleiden (eng. shard avulla). Lisäksi Atlas automatisoi tietokantojen varmuuskopioinnin, joten aikaisemmin käytettyjä eräajoja ei enää tarvittu.

4.3.5 Datan päivittäminen

Edellisen version suurin ongelma oli eräajojen palvelimille aiheuttama rasitus. Palvelujen nopeuttamiseksi tietokannat ja eräajot siirrettiin omille palvelimilleen. Lisäksi eräajoja muokattiin siten, että yksi ajo pystyi tekemään operaatioita useampaan tietokantaan ajojen nopeuttamiseksi välttämällä eri tietokantojen synkronoinnin RESTin yli.

Kaikkea palvelinten välistä synkronointia ei kuitenkeen siirretty pois palvelimilta.

Jotkin yksittäiseen olioon tehtävät muutokset levitettiin palvelimelta toiselle RESTin läpi, mutta nämä olivat hyvin kevyitä operaatioita ja eivät vaikuttaneet huomattavasti palvelujen toimintaan.

Eräajojen siirtämisessä omalle palvelimelle rikottiin mikropalvelujen periaatetta: Yksi eräajo pystyi operoimaan useamman palvelun tietokantaa ajon tehostamiseksi. Tämä luo kytköksiä eri palvelujen välille. Seurauksena on se, että päivitettäessä palvelimen tietokantaolioita on päivitettävä myös eräajojen vastaavia luokkia. Tällöin muutosten julkaiseminen vaati mikropalvelujen ja eräajojen samanaikaisen julkaisemisen. Pahimmassa tapauksessa päivitykset pitää tehdä neljään eri paikkaan. Jonkin paikan päivittäminen aiheuttaa sen, että kyseinen eräajo poistaa kannasta päivityksessä tuodut muutokset. Pahimmassa tapauksessa tämä rikkoo merkittävän osan SaaS-palvelun toiminnallisuudesta.

4.4 Versioiden yhteenveto

Seuraavaksi lyhyt yhteenveto siitä miten eri GameRefinery SaaSin versiot eroavat toisistaan.

4.4.1 Versionhallinta

Ensimmäisessä versiossa käytettin GitHubia versionhallintaan, mutta siinä ei tuntunut olevaan mitään muita käytäntöjä. Kaikki koodi kuului yhteen repositorioon, jossa oli sekavasti nimettyjä haaroja. Haarojen nimistä ei pystytty päättelemään oliko minkäänlaista versiointia käytössä.

Toisessa versiossa käytettiin myös GitHubia, mutta siinä ohjelmiston versiot erotettiin toisestaan säilyttämällä ne omissa haaroissaan. Näin pystyttiin palattamaan vanhempi versio palvelimelle, jos sille oli tarvetta. Binäärejä ei tallennettu, joten versiot oli koottava uudelleen palautuksen yhteydessä. Tämän seurauksena ei pystytty palaamaan täysin samaan tilaan.

Kolmannessa versiossa Gitin lisäksi versionhallintaan käytettiin OpenShiftiä, jossa versioitiin eri koontiversioiden binäärit. Gitissä ei enää luotu haaroja julkaisuille, tai merkitty versioita millään mullakaan tavalla. Aikaisempaan versioon pystyttiin palaamaan julkaisemalla version binääri arkistosta. Koska versioiden committeja ei merkitty ja niillä ei ollut omia haarojaan, tuli vanhojen versioiden koodin löytämisestä ja selvittämisestä missä versiossa mikäkin muutos tehtiin vaikeaa.

4.4.2 Toimittaminen ja julkaisu

Ensimmäisessä versiossa julkaisun tekeminen vaati ensin kahden skriptin suorittamisen ja lopuksi julkaisun viimeistelemisen Amazon OpsWorksin konsolissa. En itse ikinä tehnyt julkaisua tässä versiossa, mutta julkaisuprosessi, johon kuuluu kolme askelta ei vaikuta nopealta eikä helpolta. Varsinkin kuin julkaisun viimeisteleminen

OpsWorkissä vaati jodenkin asetusten tekemisen manuaalisesti.

Toisessa versiossa toimittaminen toteutettiin viemällä koodia projektin päähaaraan tai Openshiftin rhc-työkalulla, jos haluttiin toimittaa jokin toinen haara. Tämä prosessi mahdollisti stagingin version ketterän vaihtamisen ja eri ominaisuuksien jakamisen eri haaroihin. Julkaistessa ominaisuuksien haarat yhdistettiin päähaaraan, joka laukaisi automaattisen toimittamisen staging ympäristöön. Tämän läpimennessä päähaaran koodi vietiin tuotannon haaraan, mikä laukaisi automaattisen toimittamisen tuotannon ympäristöön. Ainoa haittapuoli tässä menettelyssä on se, että koodi kasataan kahteen kertaan. Tämä tarkoittaa sitä, että stagingilla ja tuotannossa on eri binäärit, mikä voi johtaa eroavaisuuksiin niiden toiminnassa.

Kolmannessa versiossa päähaaran päivittäminen laukaisi automaattisen toimituksen staging ympäristöön. Tämä oli myös ainoa tapa toimittaa versoita, mikä johti siihen, että aina kun jokin ominaisuus haluttiin stagin-ympäristöön piti se liittää päähaaraan. Tämän takia ominaisuuksia ei pystytty kunnolla eristämään omiin haaroihinsa. Julkaisuja tehdessä piti välillä päättää, haluttiinko julkaista keskeneräisiä ominaisuuksia, vai muuttaa koodia julkaisun ajaksi ja palauttaa muutokset julkaisun jälkeen, jotta ne saadaan takaisin stagingille. Versio julkaistiin tuotantoon merkkaamalla staging-ympäristön binääri Openshiftin oc-työkalulla. Tämä latasi kyseisen binäärin tuotanto-ympäristöön ja julkaisi sen. Tässä prosessissa julkaiseminen tuotantoon oli yksinkertaista ja turvallisempaa kuin ennen. Se, että kaikki koodi oli käytännössä pakko viedä päähaaraan, johti julkaisujen koon kasvamiseen, koska usein piti odottaa sellaisten ominaisuuksien valmistumista ja testausta, joita ei välttämättä oltaisi haluttu julkaista vielä. Tämä vaikeutti myös julkaisujen testausta, koska kerralla oli testattavana useampi määrä ominaisuuksia.

4.4.3 Virheistä palautuminen

Ensimmäisessä versiossa ei ollut virheistä palautumista, koska palvelusta ei tehty julkaisuja. Koodia toimitettiin vain esittelytarkoituksessa sprinttien yhteydessä.

Toisessa versiossa rikkinäisestä julkaisusta voitiin palautua julkaisemalla varhaisempi versio aikaisemmassa osiossa kuvatulla tavalla. Datamallin muutoksia ei kuitenkaan voitu palauttaa helposti. Ne piti tehdä käsin tai erillisellä skriptillä ja palvelussa piti pitää huoltokatko tämän ajaksi.

Kolmannessa versiossa virheistä palautuminen toimi pitkälti samalla tavalla kuin toisessa versiossa. Merkittävin ero oli se, että toimituksen epäonnistuessa Openshift palautti automaattisesti palvelimelle mikropalvelun aikaisemman version. Datamallin muutokset olivat vieläkin vaikeita, mutta tässä versiossa ei pidetty huoltokatkoja, joten käyttäjät saattoivat kohdata virheitä datan ollessa vääränlaista.

4.4.4 Datan päivittäminen

Ensimmäisessä versiossa ei ollut merkittaviä eräajoja eikä muita palvelimia joiden välillä olisi tarvinnut päivittää dataa. Lähin asia datan päivittämiselle olivat tieto-

kannan alustus skriptit.

Toisessa versiossa eräajot toimivat ajastetusti mikropalvelun sisällä vieden palvelimen resursseja. Eräajot olivat mikropalvelukohtaisia, joten päivittäessä dataa useammalle palvelimelle ne joutuivat tekemään raskaita http-kutsuja. Tämä oli hidasta ja ei toiminut kunnolla, koska Openshift katkaisi kutsut niiden kestäessä liian kauan. Eräajojen ulkopuolella palvelimet kommunikoivat http-kutsujen avulla. Nämä olivat huomattavasti kevyempiä ja niitä tehtiin harvemmin kuin eräajojen tapauksessa, joten niiden kanssa ei ollut ongelmia.

Eräajojen nopeuttamiseksi kolmannessa versiossa eräajot siirrettiin pois mikropalveluista omalle palvelimelleen. Eräajot myös pystyivät operoimaan useamman mikropalvelun kantaan suoraan. Tämä teki eräajoista huomattavasti nopeampia ja vähensi palvelinten kuormaa. Haittapuolena tämä loi epäsuoran kytköksen eri palvelinten datamallien välille, mikä rikkoo mikropelveluarkkitehtuurin periaatteita.

4.5 Havaitut ongelmat

Tällä hetkellä julkaisujen tekeminen on vaikeaa. Tämä on seurausta useamman asian yhteisvaikutuksesta. Ensinnäkin, staging-ympäristöön toimittamiseen vaaditaan muutos repositorion päähaaraan, joka laukaisee uuden version toimittamisen. Tämä vaatii melkein jokaisen työn alla olevan ominaisuuden liittämisen päähaaraan tehokkaan kehittämisen mahdollastamiseksi. Varsinkin kun kehityksen prioriteettien muuttaminen kesken kehityksen johti usean keskeneräisen ominaisuuden kasautumiseen päähaarassa.

Tämä puolestaan vaikeutti julkaisujen tekemistä. Julkaisun tekeminen tehtiin merkkaamalla staging-ympäristössä oleva koontiversio versionumerolla ja toprod-merkinnällä. Mutta yllämainitusta syysta staging ympäristössä tuli usein paljon keskeneräisiä ominaisuuksia, joita ei haluttu ottaa mukaan julkaisuun. Pahimmassa tapauksessa tämä vaati joidenkin ominaisuuksien poistamisen päähaarasta julkaisun ajaksi ja niiden palauttamisen julkaisun jälkeen. Koska ominaisuuksia ei jaettu kunnolla omiin haaroihinsa, niin keskeneräisten ominaisuuksien löytäminen saattoi olla vaikeaa tässä tapauksessa.

Se, että julkaisujen versioita ei eritelty Gitissä teki hotfikseistä vaikeita. Usein päähaara oli sellaisessa tilassa, että siitä ei voinut tehdä julkaisua turvallisesti löydetyn virheen korjaamiseksi. Tästä nähdään, että versionhallinnan käytäntöjen ja toimittamis- ja julkaisuprosessin yhteisvaikutus aiheuttaa laajoja ongelmia.

Datamallin muutokset toteutettiin manuaalisesti siihen tarkoitetuilla skripteillä. Muutoksesta riippuen skriptejä pystyi olla useita, jotka piti suorittaa tietyssä järjestyksessä. Tämä moninmutkaisti julkaisuprosessia lisäämällä siihen manuaalisia välivaiheita. Datamallin muutokset olivatkin yleinen ongelmien syy julkaisuissa.

Eräajojen siirtämisellä mikropalveluista omiksi projekteikseen myös hankaloitti julkaisua. Tämä johtuu siitä, että siirto loi vahvoja kytköksiä mikropalvelujen ja eräajojen välillä. Joten mikropalvelun datamallin muuttaminen vaati muutoksia myös

siihen liittyviin eräajoihin. Tämän seurauksena pitää aina muistaa sisältääkö julkaistava versio datamallin muutoksia, ja julkaista myös eräajot jos sisältää. Tämä johti julkaisussa lisätyn datan menettämiseen, jos jokin eräajoista unohdettiin julkaista.

Mikropalvelujen palauttaminen aikaisempaan versioon on helppoa virhetilanteen sattuessa. Mutta datamallien palautusskriptien puuttuminen tekee siitä käytännöllistä vain silloin kun virhe huomataan nopeasti ja tietokanta voidaan palauttaa varmuuskopiosta ilman merkittävää datan menetystä.

5 Uusi jatkuvan kehittämisen prosessi

Tässä luvussa esitetään uusi jatkuvan integraation prosessi. Se käydään läpi osa-alueittain aloittaen jatkuvan integraation palvelimesta, joka ei itsessään ole osa mitään tiettyä prosessin osa-aluetta, vaan jolla mahdollistetaan jatkuvan integraation prosessin toiminta. Tämän jälkeen siirrytään versionhallintaan ja julkaisuun, jota seuraa datamallin hallinta.

5.1 Jenkins: Palvelin jatkuvalle integraatiolla

Jatkuvaa integraatiota on mahdollista käyttää ilman sille tarkoitettua palvelinta, mutta sellaisen käyttäminen tehostaa prosessia huomattavasti. Integraatiopalvelin toimii paikkana, johon keskitetään jatkuvan integraatio -prosessin automatisoidut vaiheet. Esimerkiksi koodin kokoaminen, testaaminen ja staattinen analyysi voidaan suorittaa automatisoidusti integraatiopalvelimella.

Jenkins on avoimen lähdekoodin automaatiopalvelin. Se tukee kokoamisen ja testaamisen automatisointia, sekä automaattisten raporttien luomista ja ilmoitusten lähettämistä testien tuloksista. Jenkinsissä automaatioputki toteutetaan koontiajojen (eng. build job) avulla. Niissä määritellään mitä tarkalleen ottaen automaatioputkessa tapahtuu. Yksinkertainen koontiajo voi koostua koontiversion luomisesta, yksikkötestien suorittamisesta ja koontiversion palvelimelle viemisestä, jos kaikki testit suoritettiin onnistuneesti.

Yksi Jenkinsin vahvuuksista on sen lisäosien laajuus. Lisäosien avulla Jenkins pystytään liittämään muihin ohjelmistokehitystyökaluihin, kuten Gittiin tai Maveniin. Lisäosilla voidaan myös laajentaa Jenkinsin toiminnallisuutta, esim. testikattavuuden selvittäminen lisäämällä tämän suorittava lisäosa. Jenkinsin Maven- ja Git -tuki ovat hyvin hyödyllisiä GameRefinery SaaSin tapauksessa, koska siinä käytetään valmiiksi näitä työkaluja.

Mavenia käytettäessä Jenkins ymmärtää Mavenin projektirakenteen ja osaa lukea sen pom-tiedoston, mikä vähentää manuaalisen konfiguroinnin määrää[14]. Jenkins osaa lukea kokoamisen konfiguraation pom-tiedostosta, mikä mahdollistaa Mavenin lisäosien käyttämisen, esimerkiksi koodin staattiseen analyysiin. Tämä myös poistaa erillisen koontikonfiguraation tarpeen, koska kaikki tarvittava löytyy pom-tiedostosta.

Openshift 3 tukee Jenkins palvelimia, joten Jenkinsin liittäminen jatkuvan integraation prosessiin on helppoa. Se on mahdollista pienillä muutoksilla olemassa olevaan deployment prosessiin. Nykyisin päähaaraa muutettaessa muutettu versio kootaan ja toimitetaan automaattisesti staging-ympäristöön. Jenkinsiä käytettäessä versio kootaan Jenkins -palvelimella. Kokoamisen jälkeen versiolle suoritettaisiin testit ja staattinen koodin analysointi, jos sellaisia on määritelty koontiajossa. Jos testit saadaan suoritettua ilman virheitä, vie Jenkins kokoamansa binäärin staging ympäristön palvelimelle. Julkaisu tehdään edelleen merkkaamalla staging-ympäristössä oleva koontiversio.

Jenkinsin lisääminen ei muuta itse prosessia. Muutos päähaarassa aiheuttaa uuden binäärin kokoamisen, joka viedään automaattisesti staging ympäristöön, joka voidaan sitten julkaista tuotantoon merkitsemällä binääri Opensiftin oc työkalulla. Kehittäjien tarvitsee vain luoda koontiajot Jenkinsiin ja lisätä tarvittavat määrittelyt projektin pom-tiedostoon.

5.2 Versionhallinta ja julkaiseminen

Nykyisen versionhallintakäytäntöjen ja julkaisuprosessien huomattiin haittaavaan toisiaan. Versionhallinnasta puuttuu tapa eritellä palvelujen versioita, mikä hankaloittaa julkaisemista. Nykyinen julkaisuprosessi puolestaan pakottaa keskeneräistenkin ominaisuuksien koodin päähaaraan, josta julkaisut tehdään. Näiden ongelmien korjaamiseksi tarvitaan muutoksia versionhallinnan käyttöön ja julkaisuprosessiin.

5.2.1 Mikropalvelujen versionhallinta ja julkaiseminen

GameRefinery SaaSin kolmannen version julkaisuprosessissa on se ongelma, että julkaisu staging- ja tuotantoympäristöihin toimii samalla tavalla. Openshift huomaa projektien päähaaroihin tehtävät muutokset ja luo jokaisesta automaattisesti uuden julkaisun staging ympäristöön. Tuotantoon julkaistessa staging ympäristön binääri merkataan, jolloin Openshift vie merkatun binäärin tuotantoon.

Koska jokainen muutos projektin päähaaraan laukaisi projektin kasaamisen ja julkaisemisen stagingiin, ei tuotantoon ikinä päässyt täysin rikkinäistä koodia. Tämä julkaisutapa kuitenkin aiheutui ongelmaksi silloin kun kehityksessä oli samaan aikaan useampia ominaisuuksia. Jotta niitä päästiin testaamaan kunnolla, niin ne piti viedä päähaaraan. Useampaa ominaisuutta toteutettaessa päähaarassa saattoi olla useampia keskeneräisiä ominaisuuksia. Tässä tapauksessa vain yhden ominaisuuden julkkaiseminen tuotantoon hankaloituu, koska silloin tuotantoon menevät myös muut stagingilla työstettävät ominaisuudet. Varsinkin hotfiksien tekemisestä voi tulla vaikeaa, esimerkiksi silloin kuin hotfiksi pitäisi tehdä sellaiseen osaan kodia mitä ollaan muuttamassa stagella.

Versionhallinnan käytännöt ovat muutenkin hyvin epämääräiset. Tämänhetkinen versionhallintaprosessi on huonommin määritelty kuin aikaisemmassa GameRefinery SaaS versiossa. Aiemmin versiot tehtiin omiin haaroihinsa, jotka liitettiin päähaaraan julkaisuvaiheessa. Tällä hetkellä päähaaraan joudutaan viemään uutta koodia siihen tahtiin kun sitä tarvitaan staging ympäristössä. Tämä hankaloittaa eri haarojen erillään pitämistä.

Aiemmissa GameRefinery SaaS versoissa on ollut ongelmana se, että testaaminen on antanut eri tuloksia staging- ja tuotantoympäristöissä. Tämä on johtunut pääasiassa datan eroista staging- ja tuotantoympäristöjen välillä. Staging ympäristössä peräkkäiset testit voivat muuttaa dataa niin, että seuraava testi korjaa edellisessä löydetyn virheen. Erityisen suuri riski tälle on datan muutoskriptejä testattaessa. Peräkkäiset testit eivät enää testaa lähtötilannetta. Kaikkien julkaisun vaativien muutosten

testaaminen ennen julkaisua tulee vaikeaksi. Tätä on koitettu korjata ottamalla tuotannosta tietokantadumppi ja ajamalla se staging-ympäristöön. Tämä on kuitenkin hidasta tietokantojen koon takia.

Tällä hetkellä koodin kokoamisen päähaaraan ennen julkaisua ei aiheuta suuria ongelmia. Se vaatii sen, että uudet ominaisuudet tehdään niin, että ne eivät vaikuta olemassa oleviin ominasuuksiin ja niin, että ne on helppo laittaa ja ottaa pois päältä. Näin pitää tehdä siksi, että ne voidaan pitää pois käytöstä siltä varalta, että jokin tärkeä palvelun osa on korjattava ennen kyseisen ominaisuuden valmistumista. Tässä tapauksessa on muistettava, mitkä asiat tuotannossa ovat keskeneräisiä, ettei niitä oteta käyttöön ennen kuin ne on saatu valmiiksi tuotantoon. Tämä tapa vaatii asioiden ulkoa muistamista. Siksi se ei tule skaalautumaan suuremmalle kehitystiimille.

Näiden ongelmien korjaamiseksi julkaisuprosessia on muutettava niin, että staging-ympäristöön voi tehdä julkaisun mistä tahansa haarasta. On myös luotava erillinen dev-haara, johon voi koota eri haarat joita halutaan testata stagingilla. Lisäksi staging- ja tuotantoympäristöjen väliin tulee luoda uusi ympäristö laadunvarmistusta varten. Laadunvarmistusympäristön tietokannat päivitetään päivittäin tuotannon tietokannoista otetuilla tietokantadumpeilla, jotta se muistuttaisi mahdollisimman paljon tuotantoympäristöä. Laadunvarmistusympäristössä testataan koko julkaisu alusta loppuun tuotannon datalla ennen julkaisua. Jos testien tuloksiin ollaan tyytyväisiä, niin binääri julkaistaan tuotantoon. Päähaara toimii kuten tähänkin asti, eli se julkaistaan staging-ympäristöön automaattisesti jokaisen muutoksen jälkeen. Mutta sinne viedään koodia vasta julkaisuvaiheessa. Eli julkaistavat ominaisuudet koottaisiin päähaaraan, joka julkaistaan staging-ympäristöön. Staging-ympäristön binääri sirretään laadunvarmistusympäristöön viimeisiä testejä varten. Jos virheitä ei löydy, niin binääri siirretään tuotantoon.

5.2.2 Eräajojen versionhallinta ja julkaiseminen

Nykyisessä versiossa eräajot kootaan ja julkaistaan käsin OpenShiftin ulkopuolella olevalle palvelimelle, eikä niillä ole kunnollista versionhallintaa. Tämä on aiheuttanut väärien versioiden tuotantoon päätymistä ja hankaloittanut julkaisujen tekemistä siten, että aina ei ole onnistuttu pitämään kirjaa siitä, mitä kaikkea kuuluu julkaista ja mitä kaikkea milloinkin on tuotannossa. Se, että eräajot julkaistaan poikkeavalla tavalla moninmutkaistaa myös julkaisuprosessia. Jenkinsin käyttöönoton yhteydessä pitää luoda oma automaatioputki eräajoille.

Eräajojen kokoamisen ja julkaisemisen siirtäminen Jenkinssiin yhtenäistää prosessin eräajojen ja mikropalveluiden osalta. Lisäksi tämä mahdollistaa Jenkinsin työkalujen käytön eräajojen testaamiseen. Putkea tehdessä on huomioitava, että palvelimet joilla eräajot ajetaan on OpenShiftin ulkopuolella, joten niiden julkaiseminen vaati erillisen skriptin binäärien viemiseen eräajo-palvelimelle.

5.2.3 Haaranhallinta

Käyttöön tulee myös ottaa Gitflow [13], se on haaranhallinta- ja julkaisumalli. Gitflowssa käytetään kahdenlaisia haaroja: pysyviä ja väliaikaisia[13]. Pysyviä haaroja ovat päähaara ja kehityshaara ja väliaikaisia eri ominaisuuksien, julkaisujen ja hotfiksien haarat. Pysyvät haarat ovat nimensä mukaisesti pysyviä, eli niitä ei poisteta missään vaiheessa. Väliaikaiset haarat puolestaan poistetaan silloin kun ne liitetään, joko pää- tai kehityshaaraan.

Päähaaran muutokset ovat aina julkaisuja. Kehityshaara käytetään ominaisuuksien haarojen luomiseen. Ominaisuuksien haarat luodaan kehityshaarasta ja niitä käytetään yhden ominaisuuden toteuttamiseen. Kun ominaisuus on valmis ja se halutaan liittää seuraavaan julkaisuun, liitetään ominaisuuden haara takaisin kehityshaaraan ja ominaisuuden haara poistetaan. Julkaisuhaarat luodaan myös kehityshaarasta ja ne liitetään päähaaraan, kun halutaan tehdä julkaisu. Julkaisuhaaran päähaaraan liittämisen yhteydessä päähaara merkitään julkaisun versionumerolla. Päähaaraan liittämisen jälkeen julkaisuhaara liitetään takaisin kehityshaaraan. Tämän jälkeen julkaisuhaara poistetaan. Hotfiksien haarat toimivat samoin, kuin julkaisuhaarat, mutta ne luodaan päähaarasta kehityshaaran sijaan. Haarojen yhdistämisessä on hyvä käyttää –no-ff komentoa, joka luo uuden commit olion[13]. Näin säilytetään eri ominaisuuksien kehityshistoria

Gitflow:hn kuuluu myös se, että jokaisella kehittäjällä on oma repositorionsa, jossa kehittäjä pitää omat työn alla olevat versionsa eri haaroissa. Kehittäjät vetävät (eng. pull) ja työntävät (eng. push) koodia päärepositoriosta, joka on yleisesti nimetty originiksi. Kehittäjät vetävät ja työntävät koodia päärepositorioon, mutta voivat vetää koodia myös toistensa repositorioista luoden alitiimejä. Tämän avulla laajaa ominaisuutta voi työstää useampi kehittäjä samanaikaisesti ilman, että keskeneräistä ominaisuutta tarvitsee työntää pääreposition [13].

5.3 Datamallin hallinta

Nykyisessä prosessissa datamallin muutokset tehdään niihin tarkoitetuilla skripteillä. Kyseiset skriptit toimivat vain yhteen suuntaan. Jos dataa halutaan palauttaa, se tehdään yliajamalla muutettu data tietokantadumpista. Kyseiset skriptit on tehty ad-hoc tyylisesti ja tarkoitettu vain kertakäyttöisiksi. Tämän seurauksena ne ovat vaikeaselkoisia ja niiden uudelleenkäyttäminen voi olla haasteellista.

Kuten kaikkien muidenkin järjestelmän julkaisemiseen ja päivittämiseen liittyvien asioiden, tulisi tietokantamigraationkin olla automatisoitua ja siinä käytettävien skriptien osana versionhallintaa[18]. Nykyinen malli, jossa tietokantamigraatiot hoidetaan manuaalisesti ajettavilla skripteillä, on epäluotettava, koska siinä luotetaan kehittäjän muistavan, mitä skriptejä on ajettava ja missä järjestyksessä. Tämä on ongelmallista varsinkin suuremmissa julkaisuissa, joissa tulee paljon muutoksia olemassa oleviin tietokantakokoelmiin tai uusia tietokantakokoelmia.

Jokaisen palvelun tulisi sisältää skriptit tietokantakokoelmien luomiseen ja/tai päi-

vittämiseen julkaisun yhteydessä. Julkaisun yhteydessä pitää myös luoda skriptit datamallin palauttamiselle siltä varalta, että palvelu pitää palauttaa aikaisempaan versioon. Ideaalisesti palautusskriptin tulee muuttaa dataa siten, että se on yhteensopivaa aikaisemman version kanssa siten, että dataa ei hävitetä. Tällä tavalla data, jota ei löydy aikaisemmassa versiossa, voidaan ottaa takaisin käyttöön, kun seuraava versio on korjattu ja valmis uudelleen julkaistavaksi.

Tällä hetkellä datanmuokkaisskriptit on toteutettu Java-luokkina, jotka suoritetaan manuaalisesti komentoriviltä. Skriptit voidaan tulevaisuudessa toteuttaa samalla tavalla ja ne voidaan ajaa automaattisesti Jenkins -putkessa. Tämän toteuttamiseksi tarvitaan jokin mekanismi, jonka avulla datanmuokkausskriptit suoritetaan vain silloin, kun niille on tarvetta. Esimerkiksi poistamalla ne käytöstä lähdekooditasolla tai luomalla omat Jenkins putket niille deploymenteille, jotka tarvitset datanmuokkausskriptien suorittamista.

5.4 Varmuuskopiointi ja virheistä palautuminen

Virhetilanteessa yksittäinen mikropalvelu voidaan palauttaa helposti aikaisemmasta binääristä, ja tarvittaessa tietokantadumpista. Vaikkakin suurin osa ajasta mikropalvelut on aikaisempaan versioon palauttamisen sijasta korjattu hotfiksillä ja mahdollisella korjausskriptillä, jos myös dataa on korjattava. Datan korjaaminen siihen tarkoitetulla skriptillä voi olla tapauksesta riippuen moninmutkaista, mutta silloin ei menetetä viimeisen kannan varmuuskopioinnin jälkeen tullutta dataa. Mikropalvelun korjaaminen aikaisempaan version palauttamisella on epäkäytännöllistä myös silloin, kun virhe huomataan tai se ilmenee useampaa päivää päivityksen jälkeen.

Mikropalveluista koostuvan sovelluksen varmuuskopioinnissa ei ole mahdollista saavuttaa konsistenssia ja palvelujen saatavuutta samanaikaisesti [28]. Jos mikropalvelujen annetaan tehdä kirjoitus- ja päivitysoperaatioita vapaasti varmuuskopioita tehdessä, niiden avulla ei voida palauttaa koko palvelua takaisin konsistenttiin tilaan[28].

Yhtenäisten varmuuskopioiden tekemiseksi palvelun tilan muuttaminen, eli kirjoitus- ja päivitysoperaatioiden tekeminen pitäisi estää varmuuskopioinnin ajaksi [28]. Tämä ei ole sopiva ratkaisu, koska se rajoittaisi palvelun käyttämistä. Varsinkin, kun varmuuskopiot otetaan päivittäin. Se myös hankaloittaisi datan keräämistä ja jalostamista, koska myös eräajot jouduttaisiin keskeyttämään varmuuskopioinnin ajaksi.

5.4.1 Virheestä palautumisen mahdollisesti aiheuttamat datavirheet

Mikropalveluiden datan epäjohdonmukaisuus ilmenee pääasiassa kahdella tavalla[28]: rikkinäisenä linkkinä(eng. broken link) ja puuttuvana tilana(eng. orphan state). Rikkinäinen linkki on tilanne, missä viitettä ei voida seurata. Esimerkiksi mikropalvelun 1 olio A viittaa mikropalvelussa 2 olevaan olioon B, mutta palautuksen jälkeen B:tä ei enää löydy mikropalvelun 2 kannasta. Puuttuvassa tilassa menetetään viite olemassa olevaan olioon, jolloin viitattava olio ei tule koskaan näkyviin asiakkaalle. Esi-

merkiksi mikropalvelu 1 menettää palautuksessa olion A, joka viittaa mirkopalvelun 2 olioon B.

Yllämainitut epäjohdonmukaisuudet voivat ilmentyä GameRefinery SaaS:ssa silloin kun varmuuskopioita otettaessa on menossa datanmuokkaus operaatio useampaan tietokantaan ja operaation lopputulos ei päädy kaikkien kyseessä olevien tietokantojen varmuuskopioon. Tätä voi tapahtua kahdella eri tavalla: silloin kun varmuuskopio otetaan kesken useampaan palveluun kohdistuvaa eräajoa tai silloin kun varmuuskopiota otetaan kesken käyttäjän tekemää http-kutsua jonka seurauksena tehdään muutoksia useampaan kuin yhteen palveluun.

Kesken eräajoja otettavat varmuuskopiot aiheuttavat myös virheen, missä osa varmuuskopioidusta datasta on eräajon luomaa tai päivittämää ja osa vanhaa. Tämä voi luoda tilanteita jossa tietokannan oliot rikkovat niille asetettuja implisiittisiä sääntöjä. Esim. varmuuskopio, joka otetaan kesken pelilistojen päivittämistä, voi sisältää kaksi peliä samalla sijalla.

5.4.2 Datavirheiden seuraukset

Rikkinäiset linkit voivat aiheuttaa näkymien ja eräajojen rikkimenemistä nullpointer -virheen seurauksena. Tämän tapahtuessa asia yleensä huomataan nopeasti, koska kaikista epäonnistuneista eräajoista tulee virheilmoitus GameRefineryn Slack kanavalle. Suurimman osan ajasta GameRefineryn työntekijät huomaavat rikkinäiset näkymät suorittaessaan työtehtäviään. On tosin mahdollista, että asiakas löytää rikkinäisen näkymän enesin. Tätä halutaan välttää, joten rikkinäisten linkkien tunnistaminen ja löytäminen on tärkeää.

Puuttuva tila ei aiheuta virheitä ja sitä on vaikea tunnistaa. Ne huomataan yleensä vain silloin, kun tietokannan sisältöä tarkastellaan manuaalisesti komentoriviltä. Puuttuva tila ei haittaa SaaS-palvelun käyttämistä eikä eräajoja, joilla dataa päivitetään, minkä takia sen etsiminen ja korjaaminen ei ole korkealla prioriteeteissa.

Tapaukset, joissa tietokannan palauttamisen jälkeen tietokannassa on vanhentunutta dataa, näkyvät käyttäjille loogisesti mahdottomina tuloksina. Esimerkiksi pelilistauksia katsottaessa voi samalla sijalla näkyä useampi peli. Käyttäjä voi kokea tämän oudoksi ja epäillä näkemäänsä dataa, mutta se ei estä häntä käyttämästä palvelua. Suurempi ongelma on se, että tällä tavalla virheellinen data tuottaa vääriä tuloksia, kun dataa jalostetaan edelleen myöhemmissä eräajoissa. Esimerkiksi virheellinen arvo pelin sijoituksessa tuottaa väärän tuloksen laskettaessa dataa, jossa käytetään pelien sijalukuja, joka puolestaan tuottaa virheellisen tuloksen, kun sen avulla lasketaan uutta dataa jne. Tämän takia tällaiset virheet on tärkeä löytää ja korjata mahdollisimman nopeasti, koska virhe yhdessä paikassa leviää nopeasti muualle.

5.4.3 Datavirheiden löytäminen ja korjaaminen

Eri datavirheiden etsiminen manuaalisesti ei ole käytännöllistä ja odottaminen, että joku törmää niihin, on riskialtista. Datavirheiden löytämiseksi tulee tehdä testit, jotka tarkistavat tietokantojen datan eheyden. Testit ovat raskaita ja joutuvat tekemään hakuja useamman palvelun tietokantoihin. Tämän takia testit on ajettava manuaalisesti tarvittaessa. Eräajot tulee pysäyttää testauksen ajaksi, koska kesken tiettyjen eräajojen testit voivat antaa vääriä virheilmoituksia.

Riippuen löydetyistä virheitä ne voidaan korjata joko manuaalisesti, tähän tarkoitukseen tehdyllä skriptillä tai eräajojen uudelleen ajamisella. Eräajojen uudelleen ajamisessa voi generoitua duplikaattidataa, joka pitää poistaa erillisellä skriptillä.

6 Arviointi

Tässä luvussa arvoidaan aikaisemmassa luvussa esitetty jatkuvan kehittämisen prosessi. Tekninen arviointi toteutetaan esittämällä testausprosessi, jonka avulla voidaan varmistaa, että aiemmassa luvussa suunniteltu jatkuvan kehityksen prosessia käyttämällä tuotettu järjestelmä toimii koko kehityskaaren ajan. Teknisen arvioinnin lisäksi arvioidaan mahdollisuuksia jatkotutkimukselle.

6.1 Tekninen arviointi

GameRefinery SaaSissa ei olla tähän mennessä otettu käyttöön automatisoitua testaamista. Sen sijaan on turvauduttu manuaaliseen testaukseen. Tämä on toiminut kohtuullisen hyvin pienten muutosten yhteydessä, mutta suurissa julkaisuissa testaaminen on voinut kestää päiviä. Manuaalisessa testauksessa jää myös usein löytämättä harvinaisempia virheitä. Virheiden paremman löytämisen ja julkaisujen nopeuttamisen vuoksi tarvitaan testauksen automatisoimista.

Automatisoidut testit ajetaan jatkuvan integraation palvelimella jokaiselle koontiversiolle. Palvelin tunnistaa kehittäjien tekemät koodimuutokset, tekee niistä automaattisesti koontiversion ja ajaa sille testit. Jos koontivetsio ei läpäise testejä, sitä ei viedä sovelluspalvelimelle. Koontiversion virheet on myös korjattava heti, jotta kukaan ei hae viallista koodia. Tarvittaessa koodi on palautettava aikaisempaan versioon, jos virheet vaikuttavat vaikeasti korjattavilta tai vakavilta. Tärkeää on, että yhteisessä repositoriossa ei ole rikkinäistä koodia, koska se vaikeuttaa muidenkin kehittäjien työtä. Jos koontiversio puolestaan läpäisee testauspalvelimen testit, se jatkaa matkaansa julkaisuputkessa sovelluspalvelimelle.

GameRefineryllä ei ole suurta kehitystiimiä, eikä erillisiä testaajia. Hyväksymistestauksessa on käytetty sellaisia GameRefineryn työntekijöitä, jotka käyttävät palvelua osana työtehtäviään. Näin on saatu testattua tärkeimmät käyttötapaukset. Tämä on kuitenkin aikaavievää ja sitoo suuren määrän työntekijöitä. Koska GameRefinery SaaS:lla ei ole olemassaolevia testejä, pitää ne tehdä siten, että niihin ei käytetä liikaa resursseja. Tämän vuoksi testaus kannattaa kohdentaa palvelun kannalta elintärkeisiin osiin, kuten datan keräämisestä ja sen jalostamisesta vastaaviin osiin, sekä Niihin osiin, jotka on todettu yleisesti virhealttiiksi.

6.1.1 Yksikkö- ja integraatiotestaus

Mikropalveluarkkitehtuurin ideana on koodin jakaminen loogisiin, vastuualueiltaan erilaisiin osiin. Koska palvelun sisäiset komponentit ovat erillään toisistaan, tuisi niiden yksikkötestaaminen olla melko yksinkertaista. Lisäksi laajat yksikkötestit ovat hyödyllisiä vanhaa koodia muutettaessa. Aikaisemmin vanhan koodin muuttaminen on ollut työlästä, koska ei ole ollut helppoa tapaa tarkistaa, mitä vaikutuksia muutoksilla on ollut. Jos muutetulla komponentilla on kattavat yksikkötestit, sitä voidaan muuttaa luottamalla siihen, että testit kertovat, mikäli jotain on mennyt

vikaan. GameRefinery SaaS perustuu dataan, joten kaikki datan keräämiseen ja manipuloimiseen liittyvillä komponenteilla tulisi olla kattavat yksikkötestit.

Integraatiotestien tekeminen on puolestaan hankalampaa eri integraatiorajapintojen määrän takia. Useat mikropalvelut saattavat kommunikoida keskenään yhden operaation suorittamiseksi. Lisäksi mikropalvelut kommunikoivat monien ulkoisten palveluiden kanssa. Jokaisen eri tilanteen testaaminen on liian työlästä, joten integraatiotestaus kannattaa rajoittaa pääasiassa testeihin, joissa testataan saadaanko ulkoiseen palveluun yhteys ja vastaako se ei-virheellä. Poikkeuksena ovat erityisen tärkeät osat, kuten päivittäisen pelidatan haku.

Yksikkö- ja integraatiotestit voidaan toteuttaa mock-datalla, esimerkiksi staattisilla json-tiedostoilla. Erillisiä testitietokantoja ei kannata luoda. Tämä sen takia, että hyväksymistestauksessa käytetään päivittäin virkistettävää laadunvarmistusympäristön tietokantaa. Tästä lisää Hyväksymistestaus-osiossa.

6.1.2 Eräajojen testaus

Eräajot ovat erittäin tärkeä osa GameRefinery SaaS palvelua. Ne hakevat ja laskevat suuren osan datasta johon koko palvelu perustuu. Eräajot ovat myös virhealttiita. Ne käyttävät yhden tai useamman mikropalvelun tietokantaa, joten mikropalvelun datamallin muuttaminen voi rikkoa kyseisen mikropalvelun tietokantaa käyttävät eräajot. Esimerkiksi mikropalvelun luokkaan lisätään kenttä, mutta kenttä unohdetaan lisätä eräajoon. Päivitystä julkaistaessa ajetaan skripti, joka luo kyseiset kentät mikropalvelun olioihin. Kentän puuttuessa eräajosta eräajo voi poistaa kentän olioilta päivittäessään niitä. Toinen mahdollinen virhe on se, että olion jonkin olemassa olevan kentän sisältöä muutetaan, mutta muutos unohdetaan tehdä eräajoon. Tällöin eräajo voi päivittää vanhan tyyppistä dataa kenttään ja rikkoa mikropalvelun, koska mikropalvelu ei enää osaa kääntää vääränmuotoista kantaliota javaolioksi. Eräajojen algoritmien muutosten julkaiseminen on myös hidasta tällä hetkellä. Koska niillä ei ole yksikkötestejä, on niiden testaaminen tehtävä ajamalla niitä staging-ympäristössä useampia päiviä ja tarkastamalla niiden tulokset.

Eräajojen yksikkötestaus ei eroa merkittävästi mikropalveluiden yksikkötesteistä. Eräajojen integraatiotestaaminen on puolestaan merkittävästi hankalampaa. Niillä halutaan testata sitä miten data on muuttunut ajossa. Testaamista vaikeuttaa se, että eräajon testaus on tehtävä oikealla datalla. Testidataa ei ole järkevä käyttää, koska se, miten data kehittyy päivittäin, voi vaikuttaa siihen toimivatko eräajot vai eivät. Jotta testidatalla saataisiin tarkkoja tuloksia, sitä pitäisi päivittää jatkuvasti. Tämä olisi niin aikaa vievää, ettei sitä kannata tehdä. On järkevämpää Käyttää QA-ympäristön dataa ja verrata sitä miten se on muuttunut eräajossa. Näin voidaan tehdä, koska tiedetään, minkä tyyppisiä muutoksia ajoissa tapahtuu. Esimerkiksi pelien sijoituksia päivitettäessä tiedetään, että kaikkien pelien sijoitukset eivät voi olla samoja kahtena päivänä peräkkäin. Tiedetään myös, että päivityksessä pelin sijoitushistoriassa pitäisi olla uusi arvo, jonka aikaleima on noin 24 tuntia edellisen historia-arvon jälkeen.

Eräajoja ei muuteta kovinkaan usein, joten yleisin tarve niiden testaamiselle tulee siitä, että halutaan tarkastaan mikropalvelujen ja eräajojen datamallien olevan samoja. Kuten yllä mainittiin, yleinen ongelma julkaisuissa on se, että jostain eräajosta puuttuu jokin uusi tietokantaluokan kenttä, jonka seurauksena kenttä katoaa tietokannasta eräajon jälkeen. Tämä voidaan tehdä samalla kun testataan eräajojen tuloksia. Silloin pitää testata olioiden kenttien nimiä niiden arvojen sijaan. Jos Eräajon jälkeen olioissa on vähemmän kenttiä, jotain on menetetty.

Testaamalla eräajoja suorittamalla ne ja tarkistelemalla niiden tuloksia tulee ongelmaksi eräajojen kesto. Kokonaisuudessaan niiden suorittamiseen menee melkein vuorokausi. Testejä varten voidaan suorittaa vain osa normaalisti ajettavista eräajoista, esimerkiksi vain yhden markkinan eräajot. Toinen mahdollisuus on suorittaa eri markkinoiden eräajot rinnan. Tämä on hyvin tietokanta-intensiivistä, mutta kaikkien eräajojen testaaminen voi olla tarpeellista joissain tilanteissa.

Koska eräajoja testatteassa voidaan haluta suorittaa eri laajuisia testejä, tulee niiden testit toteuttaa niin, että on mahdollista suorittaa eri testijoukot tilanteesta riippuen. Esimerkiksi mikropalvelun päivitystä testattaessa voidaan haluta testata, että siinä lisätty data ei katoa eräajoissa. Tällöin todennäköisesti selvitään minimaalisella eräajojen testauksella. Toisaalta tilanteessa, jossa eräajoja on refaktoroitu merkittävästi, halutaan suorittaa mahdollisimman laajat testit.

6.1.3 Hyväksymistestaus

Se, kuinka paljon hyväksysmistestaamiseen tarvitsee käyttää aikaa, riippuu pitkälti julkaisun koosta. Esimerkiksi versio, jossa on vain lisätty endpointteja ilman, että tietokantaoliluokkiin on tehty muutoksia, voidaan testata automaattisesti. Mutta jo tietokantaoliluokkien muuttaminen vaatii myös eräajojen testaamisen.

Tämä vaikeuttaa testaamista, koska eräajot, niihin liittyvät skriptit ja niiden testit ovat eri projektissa kuin testattava mikropalvelu. Siksi niitä on hankala liittää mikropalvelun integraatioputkeen. Varsinkin laajojen päivitysten yhteydessä kannattaa hyväksymistestaus hoitaa pääasiassa tai jopa kokonaan manuaalisesti. Testaamiseen kannattaa käyttää kehittäjätiimin ulkopuolista henkilöstöä, niin kuin tähänkin asti. Näin voidaan simuloida paremmin käyttäjien toimintaa, varsinkin kun monien GameRefineryn työntekijöiden työhön kuuluu palvelun käyttäminen. Aliluvussa "Versionhallinta ja julkaisu"kuvailtu QA-ympäristö soveltuu tähän mainiosti.

Suurten julkaisujen testausprosessi menisi seuraavanlaisesti:

1. Mikropalveluiden vieminen laadunvarmistusympäristöön ja niihin liittyvien automatisoitujen skriptien ja testien suorittaminen.
2. Eräajojen vieminen laadunvarmistusympäristöön ja niiden automaattisten testien suorittaminen.
3. Manuaalinen hyväksymistestaus.

Tämän prosessin ensimmäinen ja toinen askel päivittävät laadunvarmistusympäristön datan oikeanlaiseksi, joten ne on suoritettava ennen kuin voidaan tehdä manuaalista testausta. Tapauksesta riippuen tässä voi kestää kauan, esimerkiksi silloin, jos pitää luoda suuri määrä uutta dataa. Tämä voi olla ongelmallista, jos hyväksymistestejä ei saada suoritettua päivän aikana, sillä laadunvarmistusympäristön data yliajetaan kerran vuorokaudessa, jotta se olisi mahdollisimman lähellä tuotantoympäristöä. Tässä tapauksessa menetetty data on palautettava jollain tavalla, Esimerkiksi ajamalla datan päivitys/luonti-skriptit uudelleen.

6.1.4 Staattinen analyysi

Perinteisten testien lisäksi on hyvä analysoida koodin laatua, esimerkiksi epäoptimaalisten luokka- tai metodi-toteutuksien, kuolleen tai toistuvan koodin tai ihan vain huonon formatoinnin varalta. Vaikka nämä asiat eivät aiheuta virheitä ohjelman suorituksessa, voivat ne hidastaa ohjelman toimintaa. Kasaantuessaan ne tekevät koodista vaikeammin ylläpidettävää ja hidastavat sen kehittämistä. Jatkuva integraation tarkoituksena on tehdä muutosten tekemisestä mahdollisimman nopeaa. Tämän mahdollistamiseksi on tärkeää, että koodin laatu pidetään tarpeeksi korkeana.

FindBugs[1], CheckStyle[8] ja PMD[9] ovat suosittuja staattisen analyysin työkaluja Javalle. Ne tekevät osittain päällekkäisiä asioita, mutta niitä voi ja kannattaa käyttää yhdessä, sillä jokainen tarjoaa jotain mitä kaksi muuta eivät. FindBugs nimensä mukaisesti yrittää löytää ohjelmointivirheitä Javan binääreistä erilaisten virhepatternien avulla[1]. Toisin sanoen FindBugs koittaa tunnistaa javan tavukoodista tilanteita, jotka muistuttavat jotain tunnettua virhettä tai muunlaista mahdollisesti ongelmallista tilannetta. FindBugsin etsimät epäkohdat eivät rajoitu pelkkiin ohjelmointivirheisiin, vaan se etsii myös mm. huonoja ohjelmointikäytäntöjä ja tietoturvariskejä ja haavoittuvuuksia. FindBugs luokittelee löytämänsä ongelmat kahdeksaan kategoriaan[1]:

1. Huono käytäntö.
2. Oikeellisuus (eng. Correctness).
3. Kansainvälistäminen (eng. Internationalization).
4. Vihamielinen koodin haavoittuvuus (eng. Malicious code vulnerability).
5. Monisäikeisyyden oikeellisuus (eng. Multithreaded Correctness).
6. Suorituskyky.
7. Tietoturva.
8. Epäilyttävä koodi.

Tämän lisäksi FindBugs luokittelee ongelmat vakavuuden mukaan[10]. FindBugsiin voi myös tehdä ja liittää omia patterneja, jos sen mukana tulevia ei koeta riittäviksi.

CheckStyle puolestaan tarkistaa noudattaako koodi sille määriteltyä tyylistandardia. Sen avulla voidaan löytää ongelmia luokkien ja metodien sunnittelussa, sekä ongelmia koodin asettelussa ja muotoilussa[8]. CheckStyle toimii niin, että sille määritellään ehtoja, joita koodin on noudatettava. Esimerkki mahdollisesta ehdosta on se, että ei saa olla tyhjiä catch-lauseita. CheckStyle sisältää joukon valmiita ehtoja, mutta sitä voi laajentaa valmiilla liitännäisillä tai omia ehtoja tekemällä. CheckStylellä on kuitenkin muutamia rajoitteita. Niistä ehkä merkittävimmät ovat ne, että CheckStyle pystyy analysoimaan vain yhtä tiedostoa kerrallaan ja että koodi on pystyttävä kääntämään, jotta se voidaan analysoida CheckStylellä.

PMD on CheckStylen tapaan lähdekoodin analysointityökalu, joka tarkistaa lähdekoodin ehtojen perusteella. Kuten CheckStyle, PMD sisältää joukon valmiita sääntöjä ja tukee uusien tekemistä. PMD eroaa CheckStylestä siinä, että se keskittyy tyylistandardien valvomisen sijaan huonolaatuisen koodin löytämiseen. PMD:n valmiit säännöt jakautuvat kahdeksaan osa-alueeseen:

1. Parhaaat käytännöt (eng. best practices)
2. Ohjelmointityyli
3. Suunnittelu
4. Dokumentaatio
5. Virhealttius
6. Monisäikeisyys
7. Suorituskyky
8. Tietoturva

Nopea tarkastelu paljastaa, että PMD:llä ja FindBugsilla on päällekkäisyyksiä tarkastamissaan ongelmissa, mutta molemmat tarkastavat myös asioita joita toinen ei. Lisäksi niiden tavat analysoida koodia eroavat toisistaan: PMD analysoi lähdekoodia ja FindBugs käännettyjä binäärejä.

Ottaen huomioon, että PMD ja FindBugs etsivät pitkälle samankaltaisia ongelmia koodissa, voisi luulla, että ne antavat suurimmaksi osaksi varoituksia samoista asioista. Näin ei kuitenkaan ole, vaan ne huomaavat pääasiassa eri ongelmia[29]. Tämän takia niitä kannattaa käyttää yhdessä mahdollisimman kattavan virheiden havaitsemisen saavuttamiseksi.

Staattisen analyysin työkalujen käytössä on ongelmana varoitusten suuri määrä, joka tekee varoitusten filtteröinnistä pakollista, jotta voidaan erottaa tärkeät tulokset vääristä positiivisista ja turhista varoituksista[29][24]. Kaikki tässä esitellyistä työkaluista tarjoavat tapoja valita, mitä sääntöjä käytetään koodin tarkastamiseen ja

tulosten filtteröintiin. Näiden avulla projektissa voidaan ottaa käyttöön vain sille relevantit säännöt ja saada esille vain ne tulokset mitä halutaan. Sopivien sääntöjen ja filtterien selvittäminen on työläs ja aikaavievä prosessi, joka pitää tehdä työkaluja testaamalla projekti kerrallaan.

FindBugs, PMD ja CheckStyle tarjoavat Maven liitännäiset, joten niiden lisääminen GameRefinery SaaS mikropalveluihin on helppoa. Työkalujen liittäminen Jenkinsiin on myös helppoa. Jenkins tarjoaa Warnings Next Generation -nimistä liitännäistä joka kerää ja visualisoi eri staattisen analyysin työkalujen tulostamat varoitukset. FindBugs, CheckStyle ja PMD ovat sen tukemia työkaluja. Palvelujen eri versiolle voidaan ajaa FindBugsin, PMD:n ja CheckStylen testit automaattisesti Jenkins-putkessa, joten niiden käyttäminen ei lisää prosessin moninmutkaisuutta.

6.2 Jatkotutkimusta

Tässä tutkielmassa esitetty prosessin tarkoitus on helpottaa ohjelmistokehitystä ja julkaisua automatisoimalla koodin integrointia ja toimittamista. lisäksi tutkielmassa on selvitetty mahdollisia ongelmia virheistä palautumisessa ja tapoja korjata niitä. Luonnollinen seuraava askel tutkimuksessa on implementoida tässä tutkielmassa esitetty prosessi ja selvittää sen vaikutuksia ohjelmiston kehitykseen ja julkaisemiseen.

Lisäksi esitetty prosessi ei tee mitään korjatakseen eräajojen omiksi projekteikseen siirtämisestä aiheutuneita kytköksiä mikropalvelujen ja eräajojen välillä. Kytkökset vaikeuttavat ohjelmistokehitystä ja aiheuttavat ongelmia julkaisuja tehdessä. Tämän vuoksi niiden poistaminen tai niiden haittojen minimoiminen ovat toinen mahdollinen jatkotutkimuksen kohde.

Prosessin automaation laajentaminen on myös alue joka vaatii jatkotutkimusta. Tämänhetkinen prosessi sisältää jatkuvan integraation menetelmien käyttöönoton, mutta kehitysprosessin laajentaminen jatkuvaan toimittamiseen asti olisi hyödyllistä. Tämänhetkinen prosessi vaatii mikropalvelujen ja eräajojen toimittamisen erikseen. Olisi hyödyllistä, jos molemmat toimitettaisiin automaattisesti samaan aikaan, tai että eräajojen toimittaminen ei olisi pakollista mikropalveluja toimitettaessa.

7 Yhteenveto

GameRefinery SaaS on mobiilipeli- ja mobiilipelimarkkinadataa tuottava ja myyvä palvelu. SaaS tarkoitta etäsovelluspalvelua ja tulee sanoista software as service. SaaS:ssa ideana on kehittää ohjelmisto ja tarjota sitä palveluna internetin välityksellä. Sen yleisiä piirteitä ovat monivuokralaisuus, eli se, että käyttäjät jakavat sovellusympäristön ja se, että palvelua ei kehitetä jotain tiettyä asiakasta varten. Gamerefinery SaaS:ssa on käyttäjä- ja datamäärän kasvaessa koettu kehitykseen ja julkaisemiseen liittyviä haasteita. Tässä tutkielmassa selvitettiin GameRefinery SaaS:in ongelmia näihin osa-alueisiin liittyen ja suunniteltiin uusi jatkuvan kehityksen menetelmiä käyttävä kehitys- ja julkaisuprosessi korjaamaan näitä ongelmia.

Aluksi tässä tutkielmassa esitettiin tutkielman tavoitteet ja miten niihin pyrittiin pääsemään. Tavoitteena oli suunnitella uusi jatkuvan kehityksen menetelmiä käyttävä prosessi GameRefinery SaaS -palvelulle. Uusi prosessi tuli suunnitella GameRefinery SaaS:in aiempien versioden pohjalta. Versioista tuli selvittää, miten ne erosivat toteutukseltaan ja miten niissä käytettiin jatkuvan kehityksen menetelmiä.

Johdannon jälkeen selvitettiin mitä jatkuva kehittäminen tarkoittaa, mitä menetelmiä se sisältää ja mitä asioita menetelmien käyttöönottaminen vaatii. Tässä tutkielmassa keskityttiin kolmeen keskeiseen jatkuvan kehityksen menetelmään: jatkuvaan integraatioon, jatkuvaan toimittamiseen ja jatkuvaan julkaisuun. Jatkuvassa integraatiossa tarkoituksena on integroida kehittäjien koodi mahdollisimman usein yhteiseen koodikantaan, ja yhteinen koodikanta kehittäjien koodiin. Tällä pyritään varmistamaan, että jokaisella kehittäjällä on käytössään viimeinen versio sovelluksen lähdekoodista. Jatkuvassa toimittamisessa jokainen muutos koodikantaan toimitetaan tuotantoa muistuttavaan ympäristöön, josta se on valmis julkaistavaksi. Jatkuvassa julkaisussa jokainen muutos julkaistaan automaattisesti tuotantoon asti. Menetelmien kuvailemisen jälkeen käytiin läpi niiden tuomia hyötyjä ja haasteita niiden käyttöönottamiselle.

Näiden menetelmien kuvailemisen jälkeen käytiin läpi työkaluja ja menetelmiä, jotka ovat hyödyllisiä jatkuvan kehityksen menetelmiä käyttöönotettaessa ja relevantteja tälle tutkielmalle. Läpikäydyt menetelmät olivat: staattinen analyysi, mikropalveluarkkitehtuuri, kontit ja SaaS. Staattinen analyysi on hyödyllistä projektin eifunktionaalisten laatuvaatimusten valvomiseen. Mirkopalveluarkkitehtuurissa palvelu jaetaan osiin, eli mikropalveluihin, joita kehitetään ja ylläpidetään erikseen toisistaan. Tämä helpottaa jatkuvien menetelmien käyttämistä jakamalla palvelun pieniin, helpommin käsiteltäviin osiin. Kontit avulla mikropalvelut voidaan virtualisoida ja ylläpitää pilviympäristössä, mikä tekee mikropalvelujen hallinnasta helpompaa. SaaS ei suoranaisesti auta jatkuvan kehityksen käyttämistä, mutta se liittyy vahvasti siihen, mitä suunniteltavalta prosessilta tarvitaan.

Jatkuvan kehityksen menetelmien ja niiden työkalujen jälkeen käytiin läpi GameRefinery SaaS:in kolme versiota ja selvitettiin mitä jatkuvaan kehitykseen liittyviä ongelmia niissä oli. Ensimmäinen versio erosi huomattavasti kahdesta seuraavasta. Se koostui vain yhdestä PHP:llä toteutetusta palvelusta ja PostgreSQL tietokannas-

ta, ja siitä puuttui lähes kaikki toiminnallisuus, mikä löytyi myöhemmistä versioista. Ensimmäinen versio oli ulkoisen kehittäjän tekemä, joten suuri osa sen ongelmista oli seurausta huonosta kommunikaatiosta GameRefineryn ja kehittäjän välillä.

GameRefinery SaaS:n toisessa versiossa kehitys siirrettiin sisäiselle kehitystiimille ja palvelun kehittäminen aloitettiin kokonaan alusta. Ohjelmointikieleksi tuli Java ja tietokannaksi otettiin MongoDB. Suurin syy ohjelmointikielen vaihtoon oli uusien kehittäjien aikaisempi kokemus Javasta, mutta sen koettin myös auttavan front end- ja back end-koodin erottamisessa toisistaan. Tietokannan vaihtaminen taas johtui pääasiassa PostgreSQL:n ongelmista strukturoimattoman datan tallentamiseen. Toisessa versiossa myös siirrettiin palvelujen ylläpito Amazon AWS OpsWorksistä Openshiftiin. Toisen version kokemat ongelmat liittyivät siihen, että mikropalvelut sijaitsivat niihin kuuluvien tietokantojen kanssa samassa Openshift -kontissa ja dataa päivittävien eräajojen toteuttamiseen osana mikropalveluita. Kaiken tämän sijaitseminen samassa kontissa oli liian raskasta OpenShiftin toisen version konteille, mikä aiheutti hitautta palveluissa.

Kolmannessa versiossa mikropalvelujen ylläpito siirrettiin OpenShiftin kolmanteen versioon. Lisäksi eräajot siirrettiin pois mikropalveluista omiksi projekteikseen. Tämä vähensi mikropalveluiden kokemaa rasitusta, mutta loi vahvan kytköksen mikropalvelun ja sen tietokantaa käyttävien eräajojen välille. Nämä kytkökset vaikeuttivat kehittämistä ja julkaisua vaatien tietokantaolioiden muutosten tekemisen moneen paikkaan ja vaativan eräajojen julkaisemisen mikropalvelun julkaisemisen yhteydessä, kun julkaistiin päivitystä, jossa oli kyseisiä muutoksia. Lisäksi versionhallintakäytöntöjen ja toimitusprosessin muuttuminen kolmannessa versiossa aiheuttivat ongelmia. Niiden seurauksena keskeneräiset ominaisuudet oli käytännössä pakko liittää päähaaraan, ennen kun ne olivat julkaisuvalmiita samalla, kun julkaisut voitiin tehdä vain päähaarasta. Tämä vaikeutti julkaisujen tekemistä, koska päähaarassa oli melkein aina keskeneräisiä ominaisuuksia. Tämä myös teki hotfiksien tekemisestä haastavaa, koska usein päähaaraan oli ehditty liittää jokin keskeneräinen ominaisuus ennen kuin korjausta vaativa virhe huomattiin. Datamallin muutokset vaikeuttivat julkaisuja myös sen takia, että niiden tekemiseen ei ollut kunnollista mekanismia. Datan muutokset tehtiin manuaalisesti tarkoitukseen tehdyillä skripteillä. Skriptejä ei ylläpidetty palvelun kehittyessä, joten niiden uudelleenkäytettävyys oli huono. Datamuutosten palauttamiseen ei ollut skriptejä. Tämän seurauksena tietokanta piti palauttaa varmuuskopiosta, jos mikropalvelu piti palauttaa aikaisempaan versioon. Varmuuskopiosta kannan palauttaminen ei ollut mahdollista, jos virhe huomattiin yli vuorokauden jälkeen julkaisusta, koska silloin menetettäisiin eräajoissa kerättyä dataa.

Uusi prosessi suunniteltiin ratkaisemaan näitä ongelmia. Prosessissa versionhallintakäytäntöj' parannettiin ottamalla käyttöön haaranhallintamalli Gitflow auttamaan julkaisuversioiden luomista ja keskeneräisten ominaisuuksien koodin erillläänpitämistä. Lisäksi otettiin käyttöön dev-haara kehitystä varten, jotta päähaaraan ei tarvitsisi viedä keskeneräistä koodia. Toimitusprosessia muutettiin niin, että staging-ympäristöön voitiin toimittaa mistä haarasta tahansa, mutta dev- ja päähaaran muutokset toimitettiin automaattisesti. Lisäksi Staging- ja tuotantoympäristöjen

väliin lisättiin tuotantoa muistuttava ympäristö laadunvarmistusta ja hyväksymistestausta varten. Ongelmat datamallin muuttamisen kanssa korjattaisiin tekemällä datan muutoksille päivityskohtaiset skriptit, jotka ajettaisiin automaattisesti osana toimitusprosessia. Skriptit luotaisiin myös muutosten kumoamiseksi, niitä tapauksia varten, joissa halutaan palauttaa mikropalvelu aikaisempaan versioon. Uusi prosessi toteutettaisiin Jenkins-automaatipalvelimen avulla, jossa versiot koottaisiin ja ajettaisiin niiden toimittamiseen liittyvät skriptit. Myös eräajojen toimittamiseen käytettäisiin jenkinsiä. Tämä yksinkertaistaa toimitusprosessia, yhtenäistämällä mikropalvelujen ja eräajojen toimittamisen. Lisäksi se mahdollistaa datan muokkaus skriptien automaattisen suorittamisen eräajoja toimitettaessa. Lisäksi selvitettiin ongelmia tilanteessa, jossa koko palvelu pitäisi palauttaa virheestä ja miten tällaisen palautuksen voisi suorittaa.

Lopuksi esitettiin testausprosessi jonka avulla voidaan varmistaa prosessin toimivuuden ja kohteita jatkotutkimukselle. Testausprosessi koostuu automaattisesta ja manuaalisesta osuudesta. Automaattiset testit sisältävät yksikkö- ja integraatiotestit sekä staattisen analysoinnin PMD-, Checkstyle ja FindBugs työkaluilla. Yksikkö- ja integraatiotestien avuilla varmennetaan sovelluksen komponenttien oikea toiminta ja stattisen analyysin avulla varmennetaan, että ei-funktionaaliset vaatimukset täyttyvät. Tämän jälkeen tulee manuaalinen hyväksymistestausvaihe laadunvarmennusympäristössä, jossa varmistetaan funktionaalisten laatuvaatimusten täyttyminen. Testausprosessi toteutetaan osana Jenkins -putkea niin, että kaikki vaiheet manuaaliseen automaatiotestaukseen asti voidaan suorittaa automatisoidusti.

Seuraavaksi tutkimuksen kohteeksi esitettiin tässä tutkielmassa esitellyn prosessin toteuttamista ja sen vaikutuksien selvittämistä ohjelmiston kehittämiseen ja julkaisemiseen. Toinen jatkotutkimusta vaativa asia on eräajojen aiheuttamien kytkösten poistaminen tai niiden aiheuttamien ongelmien minimoiminen, jos kytköksien poistaminen ei ole mahdollista. Automaatioprosessin laajentamista jatkuvaan toimittamiseen asti esitettin kolmanneksi jatkotutkimuksen kohteeksi.

Lähteet

1 Findbugs home page. http://findbugs.sourceforge.net/, 2015.

2 Amazon aws opsworks documentation. https://docs.aws.amazon.com/opsworks/latest/userguide/welcome_classic.html, 2018.

3 Docker documentation. https://docs.docker.com/, 2018.

4 Kubernetes pod description. https://kubernetes.io/docs/concepts/workloads/pods/pod/, 2018.

5 Mongodb documentation. https://docs.mongodb.com/manual/, 2018.

6 Openshift core concepts. https://docs.openshift.com/enterprise/3.0/, 2018.

7 Postgresql. https://www.postgresql.org/about//, 2018.

8 Checkstyle home page. http://checkstyle.sourceforge.net/, 2019.

9 Pmd sourceforge. https://pmd.sourceforge.io/pmd-5.0.0/, 2019.

10 N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 9 2008.

11 D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

12 L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 3 2015.

13 V. Driessen. Gitflow. https://nvie.com/posts/a-successful-git-branching-model/, 2010.

14 J. Ferguson. *Jenkins: The Definitive guide*. Wakaleo Consulting, Wellington, New Zealand, 2011.

15 B. Fitzgerald and K. J. Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 1 2017.

16 M. Fowler. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, 2006.

17 R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226. ACM, 2017.

18 J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test And Deployment Automation.* Addison-Wesley, Boston, MA, 2010.

19 J Humble, C Read, and D North. The deployment production line. In *AGILE 2006.* IEEE, 2006.

20 A. Joy. A comparison of static architecture compliance checking approaches. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346. IEEE, 2015.

21 J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 12–12. IEEE, 2007.

22 E. Laukkanen, J. Itkonen, and C. Lassenius. Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55–79, 2017.

23 M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 4 2015.

24 P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 7 2006.

25 T Mäkilä, T Järvi, M Rönkkö, and J Nissilä. How to define software-as-a-service ? an empirical study of finnish saas providers. In *International Conference of Software Business*, pages 115–124. Springer, 2010.

26 S. Mäkinen, M. Leppänen, T. Kilamo, A. L. Mattila, E. Laukkanen, M. Pagels, and T. Männistö. Improving the delivery cycle: A multiple-case study of the toolchains in finnish software intensive enterprises. *Information and Software Technology*, 80:175–194, 12 2016.

27 R. V. O'Connor, P. Elger, and P. M. Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11):e1866, 2017.

28 G. Pardo, C. Pautasso, and O. Zimmermann. Consistent disaster recovery for microservices: the bac theorem. *IEEE Cloud Computing*, 5(1):49–59, 1 2017.

29 N. Rutar, C. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *15th International Symposium on Software Reliability Engineering.* IEEE, 2004.

30 D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 281–292. IEEE, 2003.

31 M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 3 2017.

32 D. Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, 2005.

33 D. Ståhl and J. Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, pages 736–743, 2013.

34 M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference*, pages 583–590. IEEE, 2015.

35 M. Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82. IEEE, 2015.