Samuli Juppi

# Testauskäytäntöjen suunnittelu erään dokumenttigeneraattorisovelluksen kehitysputkeen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

1.9.2021

# Tiivistelmä

| | |
|---|---|
| Tekijä: | Samuli Juppi |
| Otsikko: | Testauskäytäntöjen suunnittelu erään dokumenttigeneraattorisovelluksen kehitysputkeen |
| Sivumäärä: | 64 sivua |
| Aika: | 1.9.2021 |
| Tutkinto: | Insinööri (AMK) |
| Tutkinto-ohjelma: | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine: | Ohjelmistotuotanto |
| Ohjaajat: | lehtori Simo Silander<br>Head of Dynamo Pasi Nummisalo |

---

Moderni ohjelmistokehitys on prosessikokonaisuus, jossa ohjelmiston laadunvarmistus on ensiluokkaisen tärkeää. Ohjelmakoodin virheettömyys sekä julkaisunopeus vaikuttavat suoraan yrityksen imagoon, tuottavuuteen, tehokkuuteen sekä kuluihin. Parantamalla laadunvarmennusta pystytään vaikuttamaan suoraan osaamisen tasoon. Tiedon siirtoa ja kommunikaatiota parantavat laadunvarmistustoimenpiteet mahdollistavat muun muassa työntekijöiden osaamisen jakamisen. Erityisillä ohjelmistonkehitystekniikoilla, kuten TDD:llä, voidaan yrityksen kannalta nopeuttaa uusien työntekijöiden ohjelmakoodiin sisälle pääsyä. Laadunvarmennuksella pystytään näin vaikuttamaan yrityksen eri osa-alueisiin tavalla, josta on mahdollista saada suoraa välitöntä taloudellista, sekä teknistä hyötyä nyt ja tulevaisuudessa.

Insinöörityössä paneuduttiin erään dokumenttigeneraatiosovelluksen arkkitehtuuri- ja teknologiakokonaisuuteen. Työn tavoitteena oli luoda suunnitelma, kuinka testaus olisi mahdollista tuoda osaksi ohjelmiston kehitysputkea. Osana testauksen suunnittelua työssä tutkittiin laajemmin laadunvarmistusta jatkuvan integraation, ohjelmistokehitystekniikoiden sekä erinäisten muiden alan käytänteiden osalta.

Työn tuloksena luotiin kaksiosainen suunnitelma, joka jaettiin niin, että se perustui suunnitelmassa mainittujen toimenpide-ehdotusten toimeenpanojärjestykseen. Tämä testauksen ja laadunvarmistuksen kehityssuunnitelma tarjoaa perustellun näkemyksen kustannustehokkaista toimenpiteistä, joilla dokumenttigeneraatiosovelluksen laadunvarmistusta voidaan parantaa, mikä vähentää virheiden määrää sekä lisää julkaisunopeutta. Suunnitelman aihealueet jaettiin testaukseen, jatkuvaan integraatioon sekä muihin käytänteisiin. Aihealueiden toimenpideaskeleet selostettiin systemaattisesti niin, että tuleva konkreettinen toteutus on mahdollisimman suoraviivaista.

| | |
|---|---|
| Avainsanat: | dokumenttigeneraatio, testaus, laadunvarmistus, JavaScript, Java, AWS |

## Abstract

| | |
|---|---|
| Author: | Samuli Juppi |
| Title: | Designing Test Practices for a Document Generator Application’s Development Pipeline |
| Number of Pages: | 64 pages |
| Date: | 1 September 2021 |
| Degree: | Bachelor of Engineering |
| Degree Programme: | Information Technology |
| Professional Major: | Software Engineering |
| Supervisors: | Simo Silander, Senior Lecturer<br>Pasi Nummisalo, Head of Dynamo |

Modern software development is a complex process in which software quality assurance is of paramount importance. The correctness of the software code and the speed of release have a direct impact on the company's image, productivity, efficiency and costs. Improving quality assurance has a direct impact on the level of competence. Quality assurance measures that improve knowledge transfer and communication enable, among other things, the sharing of knowledge among employees. Specific software development techniques, such as TDD, can help the company to speed up the time it takes for new employees to get to grips with the software code. In this way, quality assurance can have an impact on different aspects of the business in a way that has the potential to bring direct immediate economic and technical benefits, both now and in the future.

The engineering work focused on the architectural and technological aspects of a document generation application. The aim of the work was to create a blueprint for how testing could be introduced into the software development pipeline. As part of the testing design, the work looked more broadly at quality assurance in terms of continuous integration, software development techniques and various other industry practices.

The work resulted in a two-part plan, which was divided into two parts based on the order of implementation of the proposed actions identified in the plan. This testing and quality assurance development plan provides a sound vision of cost-effective measures to improve the quality assurance of a document generation application, reducing the number of errors and increasing the release rate. The topics of the plan were divided into testing, continuous integration and other practices. The steps in each topic were systematically described in order to make the future concrete implementation as straightforward as possible.



Keywords: Document generation, Testing, Quality Assurance, JavaScript, Java, AWS

# Sisällys

Lyhenteet

1 Johdanto 1

2 Taustatiedot ja tarkastelun rajaus 2

2.1 Dokumenttigeneraatio 2
2.2 Rajaus 3
2.3 Dynamo 3
2.3.1 Back end 6
2.3.2 Front end 7
2.4 Kehityskaaren lähtökohta 8
2.4.1 Monoliittisuus 10
2.4.2 Kehitys kohti mikropalveluarkkitehtuuria 10
2.5 Kehityskaariehdotuksen sisältö, tavoite ja motivaattori 13

3 Työkalujen ja menetelmien kartoitus 13

3.1 Versionhallinta 14
3.2 Laadunvarmistus 14
3.3 Testaustapojen analysointi 16
3.4 Jatkuva integraatio 21
3.4.1 Jenkins 23
3.4.2 AWS Codepipeline 26
3.4.3 Päätelmät automaatiosta 32
3.5 Muut käytännöt 34

4 Testauksen ja laadunvarmistuksen kehitysehdotus 40

4.1 Suositellut välittömät toimenpiteet 40
4.1.1 Testaus 41
4.1.2 Versionhallinta 46
4.1.3 Muut käytännöt 48
4.2 Tulevaisuuspainotteiset muutosehdotukset 49
4.2.1 Testaus 50
4.2.2 Jatkuva integraatio 52
4.2.3 Muut käytännöt 58

5 Loppusanat 59

Lähteet 61

## Lyhenteet

API: *Application Programming Interface*. Ohjelmointirajapinta, jonka avulla ohjelmat voivat vaihtaa tietoa keskenään.

ATDD: *Acceptance Test Driven Development*. Ohjelmiston kehitystekniikka, jossa ohjelmiston kehitys tapahtuu hyväksymistestausvetoisesti.

AWS: *Amazon Web Services*. Amazonin pilvilaskentapalvelu.

BDD: *Behavior Driven Development*. Ohjelmiston kehitystekniikka, jossa ohjelmiston kehitys tapahtuu testausvetoisesti niin, että toiminnallisuuden käyttäytyminen loppukäyttäjän näkökulmasta on testejä määrittävä seikka.

CI: *Continuous Integration*. Ohjelmistotuotannon menetelmä useiden tekijöiden lähdekoodimuutoksien yhdistämiseen yhteen ohjelmistoprojektiin.

CDK: *Cloud Development Kit*. Ohjelmistokehityskehys pilvisovellusresurssien määrittelyä varten.

CLI: *Command Line Interface*. Komentoliittymä, jossa voi kirjoittaa komentoja.

CRM: *Customer Relationship Management*. Teknologia, jolla hallitaan kaikkia yrityksen suhteita ja vuorovaikutusta asiakkaiden sekä potentiaalisten asiakkaiden kanssa.

DAE: *Dynamo Application Editor*. Dynamon mallipohjaeditori, jolla rakennetaan dokumenttigeneraatiossa tarpeellisia mallipohjia.

DAP: *Dynamo Application Package*. Dynamon mallipohja pakattuna tiedostokokonaisuutena.

IaC: *Infrastructure as Code*. Korkean tason kuvaileva koodauskieli IT-infrastruktuurin käyttöönoton automatisoimiseksi.

IAM: *Identity and Access Management*. Verkkopalvelu, jonka avulla voi hallita pääsyä AWS-resursseihin.

ISV: *Independent Software Vendor*. Yksityishenkilö tai organisaatio, joka kehittää, markkinoi ja myy ohjelmistoratkaisuja, jotka toimivat yhdellä tai useammalla tietokonelaitteiston tarjoajalla.

JSON: *JavaScript Object Notation*. Tiedostomuoto, joka sisältää dataa avain-arvo-pareina.

REST: *Representational State Transfer*. Hajautettujen hypermediajärjestelmien arkkitehtuurityyli.

SAM: *Serverless Application Model*. Avoimen lähdekoodin kehys palvelimettomien sovellusten rakentamiseen.

SaaS: *Software as a Service*. Pilvipohjainen ohjelmistojen toimitusmalli, jossa pilvipalveluntarjoaja kehittää ja ylläpitää pilvisovellusohjelmistoja, tarjoaa automaattisia ohjelmistopäivityksiä ja asettaa ohjelmistot asiakkaidensa saataville internetin kautta pay-as-you-go-periaatteella.

SDK: *Software Development Kit*. Kokoelma ohjelmistokehitystyökaluja yhdessä asennettavassa paketissa.

SPA: *Single Page Application*. Web-sovelluksen toteutus, joka lataa vain yhden web-dokumentin ja päivittää tämän yhden dokumentin rungon sisällön JavaScript-rajapintojen avulla.

TDD: *Test Driven Development*. Ohjelmiston kehitystekniikka, jossa ohjelmiston kehitys tapahtuu yksikkötestausvetoisesti.

VCS: *Version Control System*. Erikoistunut ohjelmisto, jonka ensisijaisena tavoitteena on hallita koodikantojen muutoksia ajan mittaan.

# 1 Johdanto

Insinöörityön tilaajana toimi Documill Oy. Documill on riippumaton ohjelmistotoimittaja (ISV), joka on perustettu vuonna 1997 ja jonka kotipaikka on Espoo. Documillin tuotetarjonta keskittyy yritysasiakirjojen käsittelyyn asiakirjojen löytämisen, automatisoinnin ja uudelleenkäytön yhteydessä. Documillin tarjoamat palvelut sisältävät kaikki asiakirjojen jäsentämiseen, käsittelyyn, asettelulaskentaan ja renderöintiin liittyvät näkökohdat, joita vaaditaan omistettujen asiakirjamuotojen käsittelyssä. Laadullisesti palveluiden tuotoksia voidaan verrata Microsoftin ja Adoben tuotteisiin.

Documillin tuotevalikoima perustuu vahvaan immateriaalioikeudelliseen pohjaan, joka on kehitetty viimeisten 15 vuoden aikana. Sisäisen asiakirjakoneen avulla Documill tekee erittäin skaalautuvia klusterien käyttöönottoja ja rakentaa yritystason palvelinjärjestelmiä. Documillin keskeinen suunnitteluryhmä on työskennellyt yhdessä erilaisissa ohjelmistoyrityksissä 1990-luvun puolivälistä lähtien, mikä tekee siitä yhden kokeneimmista online-ratkaisuinnovaattoreista Suomessa.

Työn aiheena olevat testauskäytännöt suunniteltiin Documillin Dynamo-nimisen dokumenttigeneraattorisovelluksen kehitysputkeen. Työ sisälsi myös kehitysputken kehittämisen manuaalisesta julkaisumallista jatkuvaa integraatiota hyödyntävään malliin. Tavoitteena oli saada aikaiseksi yhdestä kolmeen suunnitelmaan, joiden pohjalta nykyistä kehityskaarta olisi suoraviivaisempi lähteä kehittämään. Suunnitelmat toimivat siis myöhemmin tehtävien ratkaisujen pohjana, mikä antaa arvokasta tietoa tilaajalle, mikä liittyy integraatio- ja testausteknologioihin sekä muihin mahdollisesti hyödyllisiin kehityskaaren kehitysosa-alueisiin. Työssä käydään läpi ohjelmistokehityksen eri työkaluja, tapoja ja trendejä, jotka liittyvät testaukseen ja jatkuvaan integraatioon sekä yleisesti ohjelmistojen luotettavuuden ja julkaisuvarmuuden parantamiseen. Työn konteksti on tärkeässä osassa. Siksi työn alkuosassa keskitytään työn taustoihin ja siihen ympäristöön, johon suunnitelmat laaditaan. Keskiosassa käydään läpi teknologioita, tekniikoita ja tapoja konkreettisesti.

Työn loppuosa vastaa alku- ja keskiosan nivouttamisesta saumattomasti yhteen sisältäen varsinaiset kehityssuunnitelmat.

# 2 Taustatiedot ja tarkastelun rajaus

Työn kohteena ollut sovellus on julkaistu alunperin 2010-luvun alkupuolella. Monet rakenteelliset ominaisuudet ovat ja olivat oman aikansa tuotoksia. Ohjelmakoodia oli vuosien varrella muodostunut kerrostuneesti eri ajanjaksoilta. Samanaikaisesti tulevaisuuden kehityssuunta oli kohti yhä modernimpia työkaluja ja alustoja, kuten taustajärjestelmän siirtäminen singleton-mallisesta Java-pinosta fyysiseltä palvelimelta pilvipalveluntarjoajien tarjoamiin palvelittomiin ratkaisuihin (serverless-arkkitehtuuri) sekä monoliittisen taustajärjestelmä- ja käyttöliittymäratkaisun kehittäminen yhä enemmän kohti mikroarkkitehtuuripohjaista ratkaisua. Oleellista oli siis pystyä suunnittelemaan testaus ja kehityskaari niin, että se kattaisi tarpeelliset osa-alueet ja integroituisi olemassa olevaan ympäristöön sekä kirjavaan ohjelmistorakenteeseen sulavasti, jolloin tarjotaan alan standardiratkaisuja ja parhaita käytänteitä, joista on mahdollista valita lopullinen toteutus myöhempänä ajankohtana.

## 2.1 Dokumenttigeneraatio

Työssä viitataan sovelluksen toimintaan dokumenttigeneraattorina. Dokumenttigeneraatiolla tarkoitetaan tapaa luoda kriittisiä dokumentteja, kuten laskuja, sopimuksia, myyntiehdotuksia, raportteja ja työmääräyksiä automaattisesti. Automaattinen dokumenttigeneraatioprosessi käsittää yleensä tietyn asiakirjan laatimisen, jossa on oma asettelu ja yksittäiset tiedot. Jokainen asiakirja koostuu staattisista ja dynaamisista tiedoista. Staattisilla tiedoilla viitataan tietoihin, jotka eivät muutu dokumenttimallipohjasta asiakkaalta toiselle. Esimerkkinä on yrityksen nimi ja osoite dokumentin ylätunnisteessa. Dynaamisilla tiedoilla puolestaan tarkoitetaan tietoja, jotka muuttuvat jokaisen asiakkaan kohdalla. Esimerkkinä on asiakkaan nimi sekä tilinumero. [1; 2.]

Esimerkki dokumenttigeneraatiosta ja sen hyödyistä:

Pankki muotoilee lainasopimuspohjan. Tähän pohjaan pohjautuva sopimus pitää kuitenkin yksilöidä jokaiselle asiakkaalle erikseen. Prosessi käyttää yhtä sopimuspohjaa, mutta yksityiskohdat ovat erilaisia. Tällaisen tekeminen manuaalisesti vie aikaa ja on vaivaloista sekä toisaalta sisältää riskin inhimillisistä virheistä. Kun tämä prosessi automatisoidaan, nämä ongelmat katoavat.

Dokumenttien sisältävät dynaamiset tiedot ovat siis muuttuvaa dataa. Tämän tietoaineksen tavoittamiseksi dokumenttigeneraattorit keskustelevatkin eri tietolähteiden, kuten Customer Relationship Management (CRM) -järjestelmien kanssa. Dynamon tapauksessa tällainen CRM on Salesforce. CRM on puolestaan tekniikka, jolla hallitaan kaikkia yrityksen suhteita ja vuorovaikutusta asiakkaiden ja potentiaalisten asiakkaiden kanssa. CRM-järjestelmä auttaa yrityksiä pysymään yhteydessä asiakkaisiin, virtaviivaistamaan prosesseja ja parantamaan kannattavuutta.

## 2.2 Rajaus

Dynamo palveluna koostuu useista eri osista varsinaisen dokumenttigeneraation ohella. Kokonaisuutta laajentamassa on käytännössä joukko erilaisia liitännäisohjelmistoja, kuten Dynamo Microsoft Officelle. Näitä liitännäisohjelmistoja ei kuitenkaan käsitellä työssä, vaan keskitytään Dynamon Java-taustajärjestelmäohjelmistoon sekä käyttöliittymäpuolelta mallipohjien tekemiseen keskittyvään alustaan Dynamo Application Editoriin (DAE).

## 2.3 Dynamo

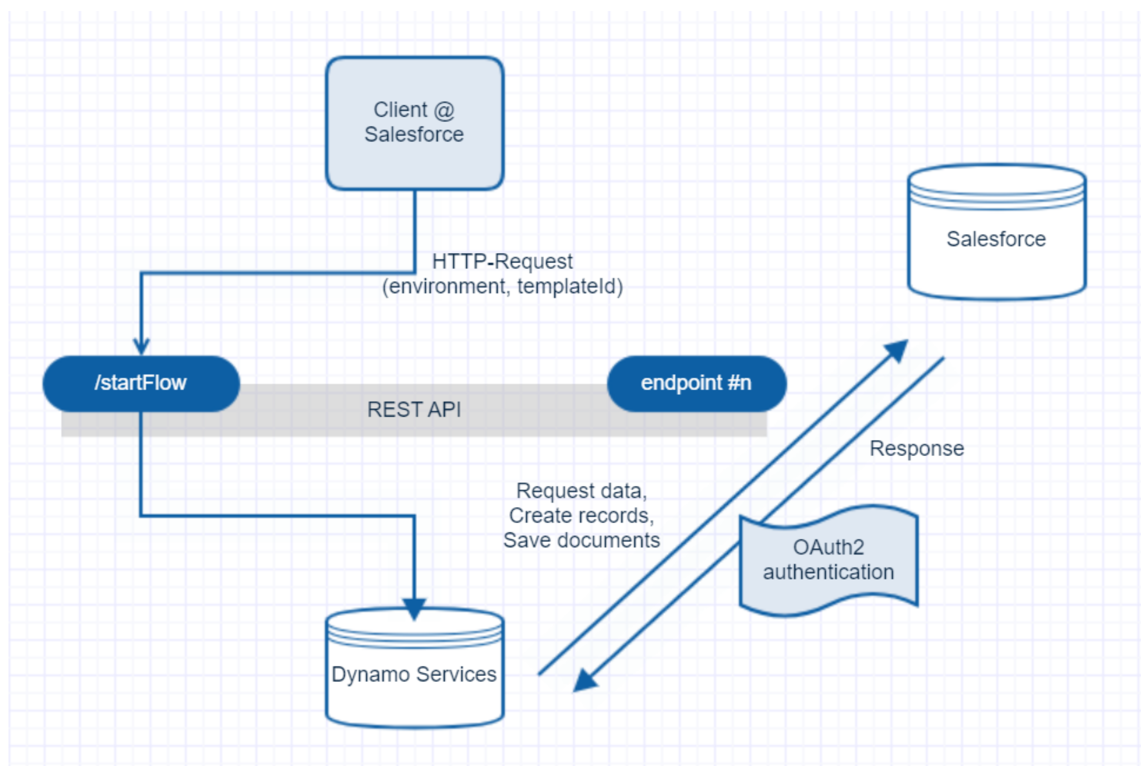
Dynamo automatisoi dokumenttigeneraation alusta loppuun. Se mahdollistaa generoinnin, yhteistyön, seurannan ja sähköisen allekirjoittamisen.

Software as a Service (SaaS) -pohjainen palvelu on suunniteltu toistaiseksi etenkin Salesforce-ympäristöön, josta generaatioon tarvittava tietoaines pääasiallisesti haetaan. Dokumentteja generoidaan pohjautuen Dynamo-

mallipohjien sisältämään logiikkaan. Pakattuina näitä mallipohjia kutsutaan Dynamo Application Packageiksi (DAP). Dynamo käyttää itse kehitettyä ohjelmointikieltä logiikan osalta, mikä mahdollistaa riippumattomuuden back end -arkkitehtuurin suhteen ja luo vahvan pohjan dynaamiselle kehitykselle.

Palvelu rakentuu Javalla kirjoitetusta back endistä sekä JavaScriptillä Single Page Application (SPA) -muotoon rakennetusta front endistä. Sovelluksen konkreettinen toiminta voidaan jakaa dokumenttien generointiin, joka tapahtuu back endissä sekä front endissä tapahtuvaan dokumenttimallipohjien luontiin ja niiden muokkaukseen.

Generaatioprosessi lähtee liikkeelle HTTP-kutsulla back endin REST-rajapintaan.




Kuva 1. Dokumenttigeneraatioprosessin aloitus Salesforce-ympäristöstä.

Kuvan 1 mukaisesti, käyttäjän ollessa Salesforce-ympäristössä, prosessi aloitetaan painamalla erityistä toimintolaukaisinta, joka sittemmin aloittaa yllä

nähtävän havainnollistavan kuvan mukaisen eri palvelujen ja palvelinten välisen kommunikaation. Laukaisin lähettää kutsun Dynamon REST-rajapintaan. Kutsu pitää sisällään dataa, joka yksilöi vähintäänkin prosessin aloittavan käyttäjän sekä mallipohjan, jota hyödynnetään. Kutsun tullessa rajapinnalle alkaa compose-prosessi, jonka aluksi kutsun mukana tulleiden tietojen lisäksi ja pohjalta haetaan Salesforcesta tarkempia tietoja. Kaikki kommunikaatio Salesforcen kanssa tapahtuu OAuth2-protokollan mukaisesti. Mikäli dokumenttigeneraatiota ei aloiteta Salesforce-ympäristöstä ja käyttäjä on uloskirjautunut aiemmin Salesforce-tililtä, vaatii Dynamo käyttäjän sisäänkirjautumaan Salesforceen, jotta palvelulla on vaadittavat oikeudet hakea oleellinen data.

On tärkeää huomata, että Dynamo ei tallenna käyttäjien tai asiakasorganisaatioiden dataa mihinkään, vaan käsittelee niitä vain välttämättömiltä osin prosessin aikana, jotta generaatioprosessi saadaan viedyksi loppuun saakka.

Asiakkaiden on mahdollista saada käyttöönsä integroitu Dynamo-toimintolaukaisin osaksi Salesforce-ympäristöään lataamalla ja asentamallaan tarvittava asennuspaketti Salesforcen Appexchange-sovelluskaupasta. Dynamo tarjoaa erityisenä ratkaisuna myös puhdasta API-lisenssiä generointiprosessin aloitusta varten, asiakastarpeiden mukaisesti.

Salesforce-ympäristössä Dynamon tarjoama lisäarvo ja yleisimmät käyttötapaukset liittyvät tarjousten tekoon, sopimusten luontiin, raportteihin ja muihin tämän tyyppisiin toimenpiteisiin, joiden suoraviivaistaminen, riskidepressio ja tehokkuuden parantaminen tuovat yrityksille merkittäviä mahdollisuuksia ja etuja, niin taloudellisesti tuottavuuden parantuessa kuin sosiaalisesti työympäristössä työn helpottumisena.

Laadullisista ominaisuuksista sekä innovaatiopainotteisesta kehityksestä johtuen Documill Dynamo on Pohjoismaiden markkinajohtaja dokumenttigeneraatiossa.

### 2.3.1 Back end

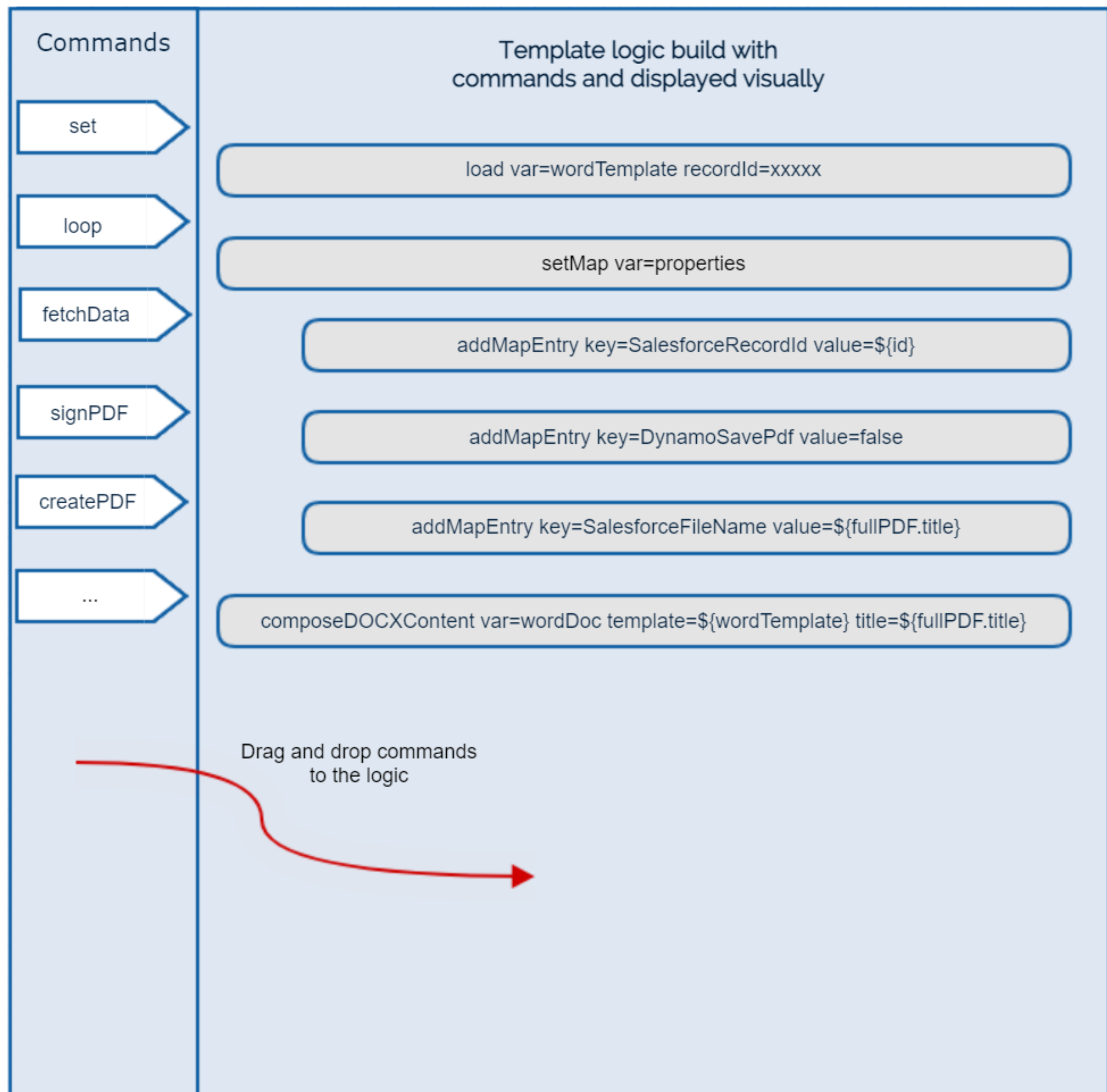
Palvelun taustajärjestelmä on kirjoitettu Java-ohjelmointikielellä, joka vahvasti tyypitettynä, olio-ohjelmointipohjautuvana ja alalla laajalti käytettynä tarjoaa ympäristön kehittää ylläpidettäviä ja tehokkaita ohjelmistoja.

Korkealla tasolla Dynamon taustajärjestelmä ottaa vastaan kutsuja aloittaa dokumenttigeneraatio, jonka jälkeen palvelu lukee asiakkaan käyttämän mallipohjan logiikasta toimintaohjeet. Nämä ohjeet pitävät sisällään tiedon siitä, mistä dataa haetaan, mitä sillä tehdään sekä mitä muita mahdollisia toimenpiteitä suoritetaan logiikan läpikäymisen aikana. Osa toimenpiteistä määräytyy käyttäjän valintojen pohjalta niin sanotusti generoinnin aikana. Näissä tilanteissa Dynamo tarjoaa dokumenttimallin ennalta kirjoitettujen dialogien (screen step) kautta tavan integroida tämä tietoaines osaksi prosessia.

Testauksen kehittämisen kannalta on oleellista ymmärtää palvelun arkkitehtuuri ja rakenne. Tästä syystä dokumenttimallin läpikäynti taustajärjestelmässä on avainasemassa, ja kyse on juuri järjestelmän ydinalueesta, joka on tärkeä havainnollistaa.

Dokumenttimalli käyttää Documillin täysin talon sisällä kehitettyä graafista ohjelmointikieltä, ja se toimii funktionaalisin periaattein.

Ohjelmointikieli sisältää useita ohjelmoinnin peruselementtejä kuten ehtolauseita ja silmukoita. Näiden lisäksi pääosassa ovat komennot, jotka kuvastavat taustajärjestelmässä käytännössä yhtä funktiota. Näin ollen näiden funktioiden testaaminen tuleekin olemaan merkittävässä asemassa.




Kuva 2. Havainnollistava ja yksinkertaistettu kuva Dynamon graafisesta mallipohjalogiikan muokkaukseen tarkoitetusta käyttöliittymästä.

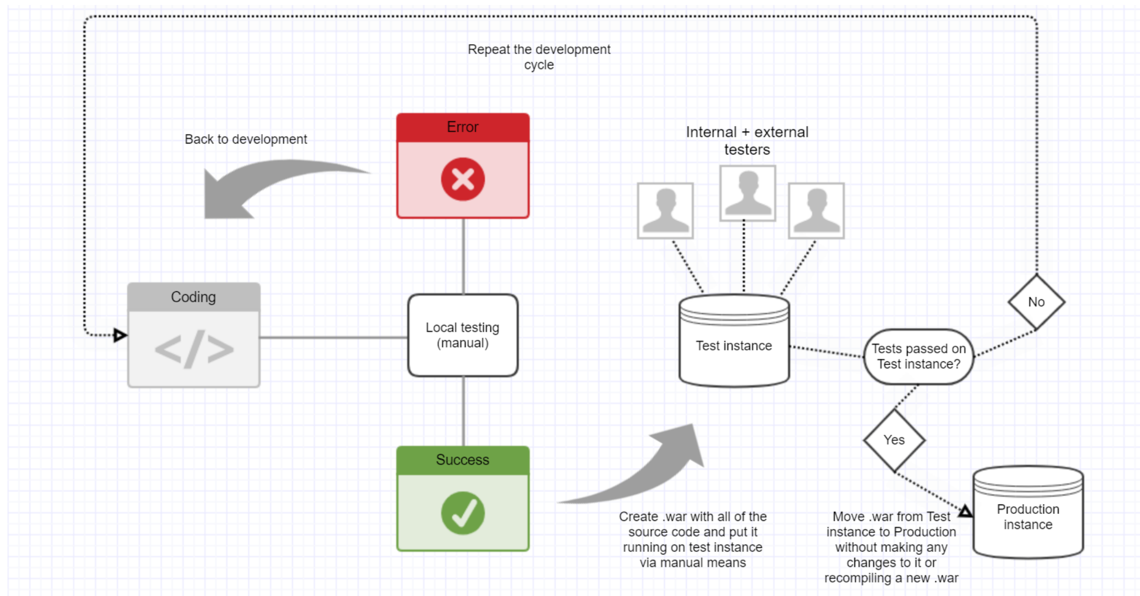
### 2.3.2 Front end

Dynamon front end on JavaScriptilla rakennettu SPA. Tämän lähtökohdan idea on minimoida HTTP-kutsujen määrä muuttamalla näkymää dynaamisesti yhden tai kerralla ladattavan JavaScript-tiedoston tai tiedostojen pohjalta sen sijaan, että käyttäjäinteraktioiden, etenkin navigoinnin, yhteydessä tehtäisiin aina uusi kutsu taustajärjestelmään.

Keskeisenä rakennustyökaluna front endissa käytetään BackboneJS-kehikkoa. Muilta osin käyttöliittymän osa-alueet ovat yhdistelmä eri ajanjaksojen lähestymistapoja. Testauksen näkökulmasta kyseessä on hyvinkin tavanomainen front end -ratkaisu, mutta sen testauksen mallintaminen jälkikäteisesti edellyttää muun muassa resursseista johtuvien rajoitusten sekä mahdollisten kirjastojen yhteensopivuusongelmien huomioonottamista.

## 2.4 Kehityskaaren lähtökohta

Dynamon kehityskaari työn lähtöpisteessä oli vakiintunut rakenteeltaan singleton-tyyppiseksi kehitykseksi. Tällä määrittelyllä viitataan pakatun sovellusversion liikkumiseen muuttumattomana testiympäristöstä myöhemmin hyväksyttynä tuotantoympäristöön. Ajan myötä kehityskaaren osalta myös muun muassa testauksesta oltiin jouduttu tinkimään. Lisäksi todettakoon, että ohjelmakoodin läpikäyntiprosessi ei ollut johdonmukainen ja julkaisuprosessi muutoinkin pääasiallisesti manuaalinen.




Kuva 3. Havainnollistava kuvaus kehityskaaresta lähtötilanteessa.

Dynamon projektihallinnan viitekehyksenä on Scrum, jota yleisesti käytetään ketterässä ohjelmistokehityksessä. Askeleet, joilla uusia toiminnallisuuksia

rakennettiin, lähtivät liikkeelle asetetuista toiminnallisuuskriteereistä, jotka muodostavat Scrumin käyttöympäristössä yhden tarinan, eri tarinoiden kokonaisuutta voidaan kutsua sprintiksi. Sprintit ovat keskimäärin 1-4 viikon ajanjaksoja, joiden sisällä tuotetaan ”valmiin” määritelmän täyttävä, käyttökelpoinen ja potentiaalisesti julkaisukelpoinen tuoteversio [3]. Tämän viitekehyksen sisällä tarina ymmärrettin valmiiksi, kun toiminnallisuus oli kehittäjän oman tulkinnan mukaan valmis, eli paikallisessa ympäristössä ei havaittu virheitä ja toiminnallisuus toimi kuten pitikin. Tämän jälkeen se vietiin osana testijulkaisua pilvipalveluntarjoajan alustalla sijaitsevalle testipalvelimelle. Tämä palvelin toimi alustana niin sisäiselle kuin ulkoisellekin testaukselle. Merkittävä osa testauksesta tapahtui pääsääntöisesti käyttämällä Dynamoa yrityksen sisäisesti niin paljon kuin mahdollista. Menetelmää kutsutaan termeillä Eating your own dogfood sekä dogfooding [4]. Tämän testausvaiheen läpäistyään toiminnallisuus siirtyi osaksi seuraavaa tuotantojulkaisua. Lopullinen tuotantojulkaisu tapahtui siirtämällä testipalvelimella testauksen läpäissyt sovellus war-pakkausmuodossa (Web Application Archive) sellaisenaan pyörimään tuotantopalvelimelle. On tärkeää huomata, että sovellusta ei siis koota uudelleen lähdekoodista, sillä Tomcat pystyy lukemaan ja pyörittämään war-muotoon pakattuja sovelluksia natiivisti sellaisenaan. Tämä tekee mahdolliseksi sovelluspakkauksen siirtämisen palvelimelta toiselle täysin muuttumattomana.

Alkuvaiheessa taustajärjestelmä- sekä käyttöliittymäkoodi sisältyivät samaan Mavenilla tehtyyn war-tiedostoon, jota pilvipalvelimella pyöritti Tomcatin tarjoama HTTP-palvelinympäristö. Myöhemmin arkkitehtuurimuutoksen myötä back end ja front end eriytettiin, ja front endin staattiset resurssit siirrettiin jaeltaviksi Amazonin S3-pilvipalvelusta, mikropalveluarkkitehtuuriambitioiden mukaisesti.

### 2.4.1 Monoliittisuus

Monoliittisella arkkitehtuurilla varustetut sovellukset ovat joukko tiiviisti kytkettyjä moduuleja, joita on vaikea erottaa monoliittisesta koodiosasta [5]. Tällaisen arkkitehtuurin haasteet liittyivät seuraavanlaisiin seikkoihin:

- Erilaisten moduulien päivittäminen ja laajentaminen vaativat palvelun uudelleen asentamista.
- Virhetilanteissa palauttaminen vaatii yhtälailla sovelluksen palauttamisen myös ongelmattomien moduulien osilta.
- Testaus tulisi toteuttaa pääasiassa yhdessä ainoassa integraatioputkessa.
- Ohjelmakoodia on suuri määrä, joka hankaloittaa ja hidastaa sen kanssa toimimista.
- Osa-alueesta vastaavan kehittäjän löytäminen ei ole aina yksinkertaista.
- Sovelluksen skaalaus vaakasuunnassa eli kapasiteetin lisääminen palvelintasolla jonkin osa-alueen aiheuttaman pullonkaulan vuoksi voi olla hankalaa.

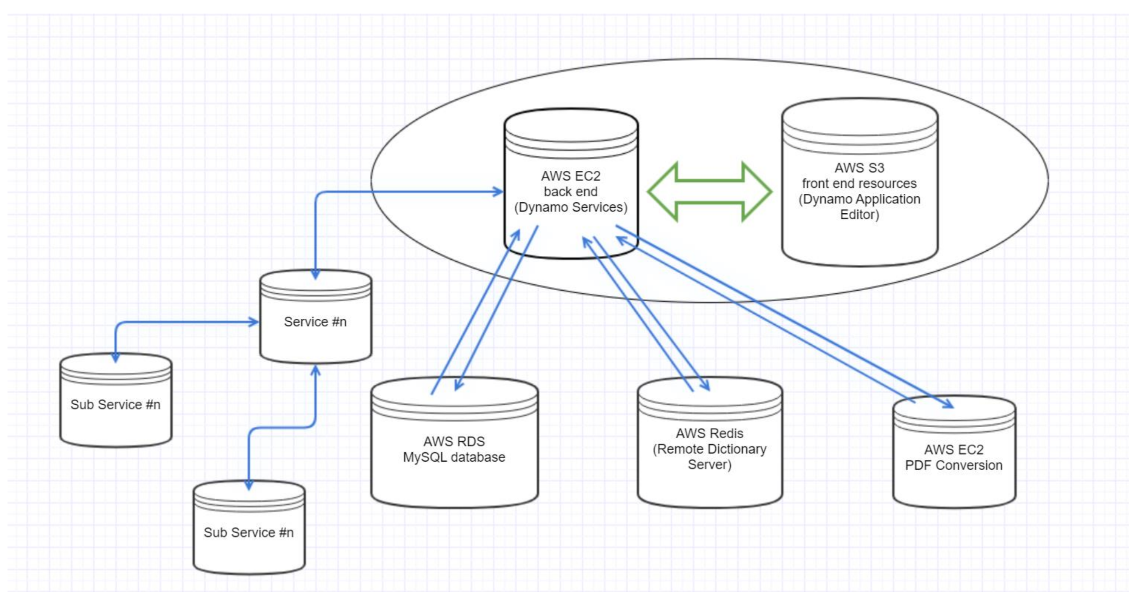
Jatkuvan integraation, testauksen ja kehityskaaren kehityksen kannalta ohjelma-arkkitehtuuri vaikuttaa etenkin kehityskaarten ja jatkuvien integraatioputkien määrään. Monoliittisessa ratkaisussa kehitysputkia on käytännössä yksi. Mikropalveluista koostuvassa kokonaisuudessa jokaisella mikropalvelulla on oma kehitysputkensa.

Yllä esitettyjen haasteiden ratkaisemiseksi Dynamossa lähdettiin kunnianhimoisesti muuttamaan sovelluksen arkkitehtuuria yhä enemmissä määrin kohti mikropalveluiden tarjoamaa arkkitehtuurimallia.

### 2.4.2 Kehitys kohti mikropalveluarkkitehtuuria

Viimeisimpänä muutoksena Dynamon mikropalveluarkkitehtuurissa sovelluksen back end ja front end eriytettiin. Java-pohjainen taustajärjestelmä toimii uuden arkkitehtuurin mukaisesti Amazonin tarjoamilla Amazon Elastic Compute Cloud -pilvipalvelimilla (EC2). Tulevaisuudessa back end on potentiaalisesti tarkoitus

pilkkoa yhä pienempiin osiin ja mahdollisesti rakentaa taustajärjestelmäkokonaisuus täysin palvelittoman laskentapalvelun pohjalle. Tällaisesta arkkitehtuuriratkaisusta esimerkkinä Amazonin Lambda-funktiot, jotka tarjoavat askeleen skaalautuvamman palvelinratkaisun verrattuna fyysisiin palvelimiin, lisäksi ilman palvelinten ylläpito- tai huoltovaatimuksia. Myös muilta palveluntarjoajilta kuten Google (Google Cloud) ja Microsoft (Azure), on saatavilla palvelittomia laskentapalveluita.

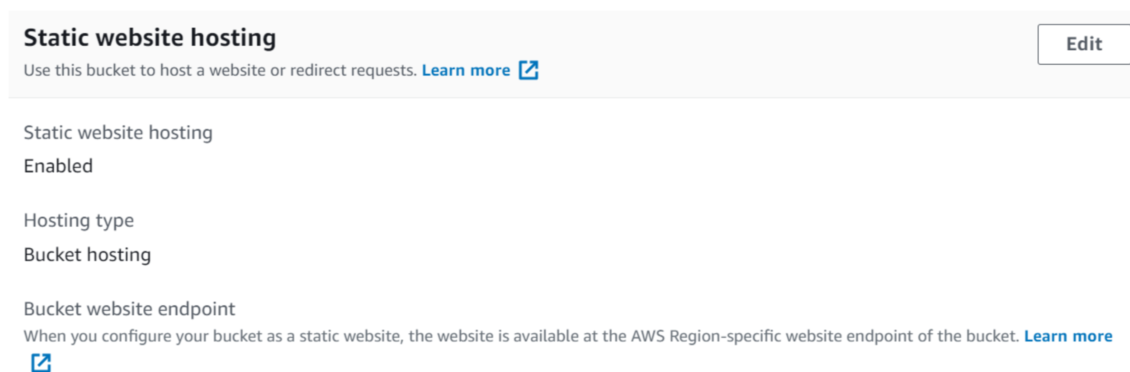



Kuva 4. Havainnollistava kuvaus Dynamon käyttämästä mikropalveluarkkitehtuurista. Kuvassa erityisesti nimettynä merkittävimmät palvelut.

Kuvassa 4 nähdään, kuinka Dynamon toiminta koostuu tällä hetkellä useasta palvelusta. Palvelut ovat pääsääntöisesti AWS:n pilvipalveluympäristössä toimivia yksiköitä. AWS RDS tarjoaa Dynamon käyttämän MySQL-tietokantaratkaisun, jossa säilytetään käyttäjien/asiakkaiden lisenssitietoja sekä suurin osa merkittävistä lokitiedoista. AWS Redis tarjoaa toiminnallisuudeltaan Dynamon kohdalla lyhyempi kestoista tallennusta. Redis pohjautuu avain-arvotyyppisesti tehtyihin tallennusoperaatioihin, jotka tehdään Rediksen omaan sisäiseen muistiin. Tästä syystä Redis tarjoaa perinteisiin tietokantoihin verrattuna erittäin nopeaa tietoaineksen lukua. Dokumenttigeneraation oleelliset PDF-muunnokset on myös eriytetty omiksi palveluiksi, jotka toimivat omilla

EC2-palvelimilla. Edellä mainittujen palveluiden lisäksi myös muun muassa sähköiseen allekirjoitukseen liittyvä toiminnallisuus on omana mikropalvelunaan. Viimeisimpänä muutoksena on mallipohjaeditorikäyttöliittymän eriytys, minkä voidaan havaita johtaneen siihen, että käyttöliittymäresurssit jaetaan AWS Simple Storage Servicestä (S3) irrallaan EC2-palvelimella toimivasta back endistä.

Sovelluksen front end jaellaan siis S3-palvelusta. S3 on Amazonin tarjoama objektien tallennuspalvelu. S3 tarjoaa skaalautuvuutta, tietojen saatavuutta, suojausta ja suorituskykyä. Palvelu tarjoaa myöskin helppokäyttöiset hallintaominaisuudet, jotta tietojen järjestäminen ja käyttöoikeuksien määrittäminen on kätevää. S3-bucketeista voidaan hallintajärjestelmän kautta tehdä lisäksi suoraan staattisia resursseja tarjoileva verkkosivusto muutamalla klikkauksella.




Kuva 5. Hallintajärjestelmän näkymä, josta S3-bucketin voi asettaa tarjoilemaan staattisia resursseja.

Dynamon ajatusmalli on keskittyä ydinalueen ja -osaamisen kehittämiseen sekä pyrkiä hankkimaan tekniset infrastruktuuriratkaisut mahdollisimman standardoituina pakettikokonaisuuksina. Tämä ajatusmalli näkyy mikropalvelukehityksessä ja on oleellista ottaa huomioon kehityskaaren työkaluja ja kehittämistä tarkastellessa.

### 2.5 Kehityskaariehdotuksen sisältö, tavoite ja motivaattori

Työn motivaationa on nopeuttaa sovelluksen julkaisuja ja minimoida virheet sekä manuaalisen työn määrä. Tuotantojulkaisuja on työajankohdan aikaan noin kahden tai kolmen kuukauden välein. Testijulkaisuja on huomattavasti useammin, lähes viikoittain. Mikäli kehitysputki olisi kehittyneempi voisi jatkuvan integraation työkalu esimerkiksi päivittää testipalvelimen automaattisesti, kun testausvaihe on onnistuneesti läpäisty. Tämä prosessi lähtisi liikkeelle aina, kun versionhallinnassa olevassa tiettyssä seuratussa haarassa tapahtuisi muutoksia. Testaus osaltaan loisi varmuutta siitä, että vähintäänkin uudet ominaisuudet ovat onnistuneesti integroitu osaksi sovellusta ilman, että mitään aiempaa, tai ydintoiminnallisuutta ole rikottu.

Työn lopputulemana olevat kehityskaariehdotukset vaativat siis kahden kehitysputken määrittelyä molemmille tarkastelun kohteena oleville palveluosa-alueille. Lisäksi tulee arvioida testaustyökalujen hyödyllisyyttä suhteessa sovellukseen. Kokonaisuus tulee mitoittaa niin, että päätavoite testauksen tuomasta julkaisuvarmuudesta osana manuaalisen työn vähentämiseen tähtäävää automaatiota saavutetaan.

## 3 Työkalujen ja menetelmien kartoitus

Ohjelmistokehitysputket ovat pääasiallisesti hyvin yksilöllisiä jokaiselle organisaatiolle. On kuitenkin mahdollista havaita modernien kehityskaarten koostuvan suurin piirtein seuraavista osista [6]:

- ohjelmakoodin kirjoittamisesta
- koodin tarkistuksesta (review)
- testauksesta
- julkaisusta.

## 3.1 Versionhallinta

Jotta tämä kehitysprosessi toimii mutkattomasti, eräs olennaisista palasista on lähdekoodin versionhallinta. Versionhallinnalla tarkoitetaan tekniikkaa, jolla pidetään kirjaa tiedostoihin (ja joskus myös paperiasiakirjoihin) tehdyistä muutoksista ja säilötään niiden vanhemmat versiot [8]. Ohjelmoinnin osalta tällaista toiminnallisuutta käytetään lähdekoodin hallintaan. Tämän osalta toiminnallisuutta tarjoavia työkaluja on useita, kuten: Git, SVN, CVS, Mercurial. Näistä suosituin ja Dynamossakin käytetty työkalu on Git, joka perustuu vapaaseen lähdekoodiin. Koska Git on alalla lähes standardiratkaisu, todetaan että kehityskaariehdotuksissa ei ole tarvetta tarjota muuta vaihtoehtoa tämän suhteen. Versionhallinnan osalta saattaa olla kuitenkin tarpeellista ottaa kantaa sen käyttötapoihin.

Versionhallinnan nykykäytänteet sisältävät riskejä. Gitin päähaarana käytetään "develop"-haaraa. Tämä versiohaara on myös ainoa varsinainen kehitystä seuraava haara. Pääsääntöisesti alalla hyödynnetään ainakin kahdesta, "main" ja "develop", kehityshaarasta koostuvaa versionhallintamallia. Lisäksi käytänteitä committien osalta ei ole erityisesti vahvistettu, eli paikoin päähaaraan liitetyt commitit ovat varsin monimuotoisia ja niitä on liian useita. Sekavuus liittyen committien määrään tai sisältöön poistuu kuitenkin helposti ottamalla käyttöön useampi kehityshaara, jolloin voidaan erinäisillä teknisillä ratkaisuilla, kuten *git-rebase*, huolehtia siitä, että pääkehityshaaran commit-historia pysyy mahdollisimman selkeänä. [8.]

## 3.2 Laadunvarmistus

Ohjelmistokehityksessä laadunvarmistus käsittää tavat ja keinot tarkkailla ohjelmistokehitysprosesseja ja metodeja, joita käytetään käsillä olevassa projektissa varmistamaan ohjelmiston asianmukainen laatu. Laajassa katsannossa laadunvarmistus sisältää kaikki standardit ja menettelytavat, joita managerit, järjestelmänvalvojat tai jopa kehittäjät voivat käyttää läpikäydäkseen

ja tarkastaakseen ohjelmistotuotteita ja toimintoja varmistaakseen, että ohjelmisto saavuttaa asetetun standardin mukaisen laadun.

Laadunvarmistuksen voidaan katsoa koostuvan kolmesta laajemmasta pääkohdasta [9]:

- organisaation laajuisista käytännöistä, prosesseista ja standardeista
- projektitasoisista käytännöistä, prosesseista ja standardeista
- määriteltyjen menettelytapojen noudattamisesta.

Todettakoon, että laajassa mielessä laadunvarmistuksen elementit kattavat käytännössä koko opinnäytetyön aihealueen, ja on sitä laajempi kokonaisuus. Työn jaksotuksen johdosta tämä luku sisältää kuitenkin yllä esitetyn laadunvarmistuksen määrittelyn lisäksi vain pohjustuksen koodin tarkistukseen sekä testaukseen.

Koodin tarkistus on osa ohjelmiston laadunvarmistusprosessia, jossa yksi tai useampi henkilö tarkistaa kehitetyn toiminnallisuuden lähinnä katsomalla ja lukemalla lähdekoodin osia. Tarkistus tehdään, kun kehittäjä pitää ominaisuutta valmiina. Oleellista prosessissa on, että ainakin yksi tarkistusta suorittava ei ole itse koodin kirjoittaja.

Vaikka laatuongelmien suora löytäminen on usein päätavoite, koodiarvostelut suoritetaan yleensä tavoitteiden yhdistelmän saavuttamiseksi [10]:

- Pyritään parantamaan koodin laatua ja ylläpidettävyyttä. Luettavuus, yhdenmukaisuus ja ymmärrettävyys ovat tärkeitä kriteereitä.
- Etsitään vikoja koodin oikeellisuuden osalta sekä myös tehokkuus- sekä turvallisuusnäkökulmasta.
- Lisätään oppimista ja tiedonjakoa liittyen laatuodotuksiin sekä ratkaisulähtökohtiin, sekä koodin tarkastajalle että tekijälle.
- Kasvatetaan yhteistä vastuuntunnetta koodista ja vahvistetaan omistajuutta sekä yhteistoimintaa.
- Pyritään löytämään uusia ja parempia ratkaisuja, joista on hyötyä niin tarkastettavan koodin osalta kuin myös laajemminkin.

- Toteutetaan laadunvarmistuksen ohjeistuksia ja standardeja niiltä osin, kuin ne ovat pakollisia esimerkiksi lentoliikenteen ja turvallisuuskriittisten ohjelmistojen osalta.

Ohjelmiston testaus on, kuten aiemmin mainittu, osa laadunvarmistusta. Testauksella pyritään minimoimaan virheiden määrä sekä ylläpitämään mahdollisimman korkeaa laatustandardia. Ohjelmiston testaaminen voidaan suorittaa missä tahansa kehitysvaiheessa, riippuen valitusta testaustavasta. Suurin osa testauksesta kuitenkin suoritetaan, kun kaikki vaatimukset on määritelty ja ohjelmointivaihe on suoritettu.

Testaus voidaan karkeasti jakaa muutamaan osa-alueeseen, joita ovat [6]:

- yksikkötestaus
- integraatiotestaus
- systeemitestaus
- hyväksymistestaus.

Myös testaustasoja on useita, jopa yli 50, ja näistä monet ovat systeemitestauksen eri tyyppejä [11]:

- toiminnallinen testaus
- regressiotestaus
- ei-toiminnallinen testaus
- käytettävyystestaus
- tutkiva testaus.

## 3.3 Testaustapojen analysointi

Paremman ja tarkemman kuvan saamiseksi sen suhteen, mitä testaustapoja voidaan ja pystytään hyödyntämään sekä mitkä ovat ne tavat, joita kannattaa hyödyntää. On syytä lähteä liikkeelle tutkimalla eri testaustapojen tavoitteita ja ylipäätänsä minkälaiseen ongelmaan tai ongelmiin kyseinen testaustapa pyrkii antamaan ratkaisun.

**Yksikkötestauksen** ydinajatuksena on testata lähdekoodin osia yhdessä tai erikseen. Periaatteessa voidaan sanoa, että yksikkötestaus antaa pohjan ohjelman pienimpien osien testaukselle, jonka lopputuloksena voidaan varmentua näiden osien oikeellisuudesta. Yksikkötestit on usein automatisoitu ja niiden luonti pääsääntöisesti tapahtuu samaan tahtiin muun kehitystyön edetessä. Ohjelman eri osat on mahdollista korvata yksinkertaistetuilla korvikkeilla testien läpiviemiseksi (mock). Nämä korvikkeet, sikäli kun niitä käytetään, eivät ole osa varsinaista testattavaa yksikköä. [12.]

Yksikkötestauksen toteuttaminen onnistuu parhaiten, kun itse testien kirjoittaminen tapahtuu ajallisesti lähellä varsinaisen ohjelmakoodin kirjoittamista. Näin testit seuraavat tiiviisti uusia toiminnallisuuksia ja tämä toimintatapa onkin oivallista liittää yhteen Test Driven Developmentin (TDD) kanssa, jossa sen sijaan että testit rakennettaisiin jälkikäteisesti, tapahtuukin testien kirjoitus ennakoivasti. Haittapuolena ja rajoituksena yksikkötestauksessa voidaan lisäksi pitää sen potentiaalisesti suhteellisen suurta ajallista panostusvaatimusta. Esimerkiksi yhtä Java-koodiriviä kohden tarvitaan keskimäärin 3-5 JUnit-koodiriviä riittävän kattavuuden saavuttamiseksi. JUnit on eräs keskeisimmistä Java-ohjelmointikielen yksikkötestauskehikoista. [12.]

Johtuen pitkälti edellä mainituista syistä on siis mahdollista nähdä yksikkötestaus kaikkein hyödyllisimmin integroituna silloin, kun toiminnallisuusvaatimukset ovat tarkoin laadittu ja testien kirjoittaminen tapahtuu luontaisesti joko TDD-tyylisesti ennakoivasti tai muutoin ajallisesti lähekkäin ohjelmakoodin luonnin kanssa. Dynamon kannalta yksikkötestauksen implementoinnin haasteet liittyvätkin yleisesti pienten kehitystiimien kohtaamiin haasteisiin toiminnallisuusvaatimusten jokseenkin nopeista muutoksista ja kriteereiden vaihtuvuudesta. Lisäksi yksikkötestauksen rakentaminen jälkikäteisesti sovellukseen, joka koostuu tuhansista yksiköistä, vaatii merkittäviä ajallisia panostuksia sinänsä, ottamatta kantaa mahdollisiin yhteensopivuusseikkoihin.

Dynamon nykyhetkeen sijoittuvassa testausimplementaatiossa on siis suunnattava katse kohti kokonaisvaltaisempaa lähtökohtaa, jollaiseen pyritään muun muassa integraatiotestauksen ja systeemitestauksen osa-alueilla. Mainittakoon, että yksikkötestaus on kohde, joka on syytä huomioida tulevaisuuspainotteisesti uusien yksiköiden rakennuksen yhteydessä sikäli, kun resurssit ja muut olosuhteet sitä tukevat.

**Integraatiotestauksessa** testataan pääsääntöisesti, miten järjestelmän eri osat, yksiköt, toimivat keskenään, eli toimiiko näiden välinen integraatio. Integraatiotestaus on vaihe yksikkötestauksen ja systeemitestauksen välissä. Tarkoituksena on kasata siis joukko yksiköitä, jotka ovat siis läpäisseet jo yksikkötestausvaiheen, ajaa tämä looginen kokonaisuus integraatiotestien läpi ja varmentaa, että testauksen lopputulos on sitä mitä ollaan haluttu. Yksikkötesteistä poiketen integraatiotestien luomisesta vastaavat yleensä erityiset ohjelmistojen testaukseen keskittyneet toimijat. [13.]

Teorian puolesta tällä testausvaiheella on painolastinaan juurikin yksikkötestien puute, ja vaikkakin testaaminen on laajempaa sekä keskittyy laadun varmennukseen eri näkökulmasta kuin yksikkötestit, vaatisi sen implementointi jälkikäteisesti samansuuntaisia ponnistuksia kuin yksikkötestitkin. Olisi lisäksi tutkittava tarkemmin, löytyykö ohjelmistosta osa-alueita, jotka ovat tarpeeksi kriittisiä, että integeraatiotestaus olisi oikea ratkaisu laajuuden ja tarkkuuden puolesta, ja myös niin, että toiminnallisuutta ei pystyisi validoimaan tarpeeksi tehokkaasti ja varmasti systeemitestaustasolla. Näistä seikoista johtuen voidaan integraatiotestaukselle antaa enemmän painoarvoa tulevaisuuden tavoitteena liittyen tuleviin toiminnallisuuksiin. On kuitenkin huomioitava, että kaiken testauksen implementointi ei tulevaisuuspainotteisesti uusien ominaisuuksienkaan osalta ole mahdollista ja tämän lisäksi integraatiotestaus ei myöskään ole osa TDD:tä, vaan oma kokonaisuutensa. Nämä ajatukset puhuvat tämän koko osa-alueen sivuuttamisen puolesta. Tarkastellaan testausta siis vielä laajemmasta näkökulmasta, ja ratkaisu voikin löytyä systeemitestauksesta.

**Systeemitestaus** on testaustaso, jolla validoidaan täydellinen ja täysin integroitu ohjelmistotuote. Testauksen tarkoituksena on arvioida end-to-end-järjestelmän tekninen toimivuus. Yleensä ohjelmisto on vain yksi osa suurempaa tietokonepohjaista järjestelmää, ja viime kädessä ohjelmisto on liitetty muihin ohjelmisto- / laitteistojärjestelmiin. Systeemitestaus on itse asiassa sarja erilaisia testejä, joiden ainoa tarkoitus on käyttää koko ohjelmiston toiminnallisuutta ja tutkia sen oikeellisuus kokonaisuutta tarkastellen. Testaushierarkiassa systeemitestaus tuleekin yksikkötestien ja integraatiotestien jälkeen, mutta ennen hyväksymistestausta. Verrattuna yksikkö- ja integraatiotestaukseen täyttää systeemitasoinen testaus testausjärjestelmässä laaja-alaisempaa aluetta nimensä mukaisesti. Kokonaisvaltaisuuden ja laaja-alaisuuden takia tämä testauksen taso soveltuukin tehokkaasti jälkikäteiseen testauksen implementointiin. [14.]

Systeemitestauksen voi jakaa erinäisiin tyyppeihin, jotka testaavat ohjelmistokokonaisuutta tai ohjelmiston ominaisuuksia eri lähtökohdista. Näitä tyyppejä voidaan ohjelmistojen testauksessa yksilöidä ja havaita olevan useita kymmeniä. Paljon riippuukin organisaatiosta, mitä systeemitestauksen tyyppikokonaisuutta käytetään. Mitä pitää ja mitä tarvitsee testata, ovat ne kysymykset, joiden avulla oikeat systeemitestaustyypit implementoidaan ohjelmiston kehityskaareen. Seuraavassa eri systeemitestaustyypeistä on valikoitu yleisimpiä ja kokonaisvaltaisesti suoraviivaisimmin implementoitavia.

**Toiminnallinen testaus** on eräs näistä systeemitestauksen tyypeistä. Tällaisella testauksella on tarkoitus varmistaa, että ohjelmisto toimii toivotulla tavalla ja täyttää asetetut vaatimukset toiminnallisuutensa puolesta. Toiminnallinen testaus tehdään usein loppukäyttäjän näkökulmasta ja suoritetaan käyttöliittymää käyttäen. Dynamon kannalta voidaan todeta, että sovelluksesta löytyy selkeä käyttöliittymä, jota käyttäen toiminnallinen testaus voidaan suorittaa rakentamalla esimerkiksi laaja dokumenttipohja, joka testaa taustajärjestelmää riittävän laaja-alaisesti. [15.]

**Ei-toiminnallisessa testauksessa** käydään läpi nimenmukaisesti muuta kuin näkyvää toiminnallisuutta, esimerkiksi suorituskykyä, vikasietoisuutta tai resurssien käyttöä. [16.]

**Käytettävyystestauksessa** tutkitaan, kuinka hyvin suunniteltu tai jo toiminnassa oleva ohjelmisto toimii – onko se helppokäyttöinen, niin sanotusti intuitiivinen ja looginen, ovatko värit, kontrastit ja siirtymät hyväksyttäviä ja joutuuko käyttäjä odottamaan toiminnallisuuksia tai latauksia liian kauan, ynnä muuta. Päätelaitteesta ja käyttäjäryhmästä riippuen käytettävyyden vaatimukset voivat olla hyvinkin erilaisia, mutta hyvän käytettävyyden takana on kuitenkin aina vankka teoria ja hyvät käytännöt. Tällaista käytettävyystestausta Dynamon kehitysputkessa tapahtuu sovelluksen testipalvelimella tapahtuvan käytön myötä. [17.]

Tutkiva testaus on vähemmän suunnitelmallista, mikä perustuu usein testaajan ammattitaitoon ja kokemukseen. Yleensä painopistealueet sovitaan etukäteen ja testauksen aikana pidetään kirjaa tehdyistä asioista ja mahdollisista löydöksistä. Tutkiva testaus voi olla hyvin tehokas tapa käydä läpi kompleksisiakin asioita ilman suurta valmistelevan työn ja dokumentoinnin määrää, ja tulokset voivat olla yllättäviäkin – sessioiden aikana ehtii tehdä asioita monelta eri kantilta, ja epäortodoksisetkin lähestymistavat tulevat katettua, toisin kuin täysin suunnitellussa testauksessa. Tältäkin osin tutkivan testauksen elementtejä on havaittavissa Dynamon olemassa olevasta testipalvelintestauksesta.

**Regressiotestaus** pitää sisällään niin toiminnallisen testauksen kuin ei-toiminnallisen testauksenkin, mutta on luonteeltaan erilaista. Siltä osin kun toiminnallisessa (ja ei-toiminnallisessa) testauksessa pyritään kehityksen aikana löytämään mahdollisimman paljon uusia virheitä ja virhetilanteita, regressiotestauksessa tarkistetaan, että ohjelmistoon tehdyt muutokset eivät ole rikkoneet jo toimivaksi todettuja ominaisuuksia. Samalla tarkistetaan, että jo löydetyt virheet ovat pysyneet korjattuina. [18.]

Systeemitestauksen voidaan siis havaita tarjoavan korkeimman tason, jolla ohjelmistoa testataan organisaation sisällä ja sen vaikutusalueen lähipiirissä. Kaikkea ei kuitenkaan tälläkään pystytä varmistamaan, vaan tavallisesti testaushierarkiasta löytyy vielä seuraava aste, joka tunnetaan nimellä hyväksymistestaus.

**Hyväksymistestaus** on testauksen osa-alue, jossa loppuasiakas itse, tai heidän edustajansa, tarkastaa jo valmiin tai lähes valmiin tuotteen, ja tarkistaa vastaako se oikeita käyttötapausten vaatimuksia. Hyväksymistestaus vaatii usein hyvää liiketoiminta-alueen ymmärrystä, ja tietoa asiakkaiden vaatimuksista ja tavoista toimia sekä mahdollisesti alan säädöksistä ja lainsäädännöstä. [19.]

## 3.4 Jatkuva integraatio

Osana työn tavoitetta oli tuoda jatkuva integraatio osaksi kehityskaarien kehittämisehdotuksia.

Jatkuvalla integraatiolla (continuous integration) tarkoitetaan prosessia, jossa koko ohjelmisto koostetaan ja integroidaan jatkuvasti. Perinteisissä kehitysmalleissa integraatio sijoittuu projektin loppupuolella tehtäväksi kertarypistykseksi. Usein rypistyksestä myös tulee hyvin pitkä ja vaikea, varsinkin jos heti projektin alkumetreiltä asti ei ole osattu huomioida integroinnin haasteita. [20.]

Jatkuva integraatio ohjaa kehitystä siihen suuntaan, että ohjelmisto on milloin tahansa ainakin periaatteessa julkaistavissa hyvin nopeasti. Mikäli ohjelmiston arkkitehtuuri on suuri ja eri tiimit toteuttavat eri komponentteja, jatkuvalla integraatiolla varmistutaan siitä, että komponenttien yhteenliittämisessä ei tule suuria yllätyksiä siinä vaiheessa, kun projektia pitäisi alkaa päättämään.

Projektin koosta riippuen integroinnin jatkuvuus voi olla reaaliaikaista (esim. integrointi käynnistyy automaattisesti jokaisen commitin yhteydessä) tai

ajastettua (esim. integrointi tapahtuu joka yö). Mikäli ohjelmiston koostaminen kestää huomattavan pitkän ajan, kannattaa yleensä siirtyä ajastettuihin koosteisiin.

Tyypillisesti jatkuvassa integroinnissa tehdään kaikki toimenpiteet, mitkä kuuluvat tuotteen julkaisuun muutenkin. Yksinkertaisimmillaan tämä voi tarkoittaa vain tuoreimman ohjelmistoversion noutamista versionhallinnasta, kääntämistä ja lopputuotteen siirtämistä asennushakemistoon. Toisessa ääripäässä on järjestelmät, jotka suorittavat samalla kaikki automatisoidut testit (yksikkötestit, hyväksyntätestit ja muut mahdolliset testit), kääntää ohjelmakoodin, alustaa www-palvelimen, asentaa web-palvelut paikoilleen, koostaa käyttöohjeet ja API -kuvaukset ja lopuksi vielä kirjoittaa koko asennuspaketin DVD-levylle. [21.]

Jatkuva integraatio liitetään osaksi ohjelmiston kehitysputkea käyttämällä siihen luotuja erityisiä työkaluja. Näiden työkalujen tavoite on pitkälti edellä kuvatunlaisen automatiikkaan pohjautuvan prosessin mahdollistaminen. Tästä johtuen todettakoon tämä työkalujen joukko tässä mielessä tavoitteiltaan homogeeniseksi massaksi, joista tässä opinnäytetyössä valitaan tarkasteltaviksi ne ratkaisut, jotka parhaiten yhteensopivat yrityskulttuurin sekä organisaatiotasoisten teknologisten ratkaisujen kanssa, tai ovat muuten perusteltavissa erityisillä synergiaeduilla.

Erilaisia jatkuvan integraation työkaluja on tarjoilla laajasti. Suosituimmista voidaan luetella esimerkinomaisesti

- Jenkins
- AWS CodePipeline
- CircleCI
- Azure Pipelines
- Gitlab.

Kun organisaatiossa tehdään päätöksiä integraatiotyökalujen suhteen, otetaan yleensä huomioon muutama tärkeä seikka.

**Versiohallintajärjestelmän tuki.** Jatkuvan integraation välineiden ydinseikka on taustalla oleva versionhallintajärjestelmän (VCS) tuki ja integrointi. Etenkin pilvipohjaiset integraatiotyökalut saattavat olla rajoitetumpia tässä suhteessa. On kriittistä valita CI-työkalu, joka tarjoaa tukea projektin VCS:lle.

**Paikallinen asennus tai pilvipalvelu.** Jotkut edellä mainituista CI-työkaluista, kuten Jenkins, asennetaan niin sanotusti paikan päällä. Tämä tarkoittaa, että organisaatio on vastuussa oman infrastruktuurin CI-järjestelmän määrittämisestä ja hallinnasta. Tämä voi olla hyödyllistä yksityisyyden ja turvallisuuden vuoksi. Esimerkiksi, jos organisaatiolla on asiakkaan tietosuojaongelmia vaatimustenmukaisuusstandardien täyttämiseksi, paikan päällä voi olla vaatimus. Lisäksi paikalliset työkalut voivat tarjota syvempiä mukautus- ja määritysvaihtoehtoja.

Työkalujen osalta opinnäytetyössä tehdään valinta tarkastella lähemmin Jenkinsiä sekä AWS Codepipelineä osana kehityskaariehdotusten jatkuvan integraation ratkaisuja.

### 3.4.1 Jenkins

Jenkins on tunnettu ja pitkään alalla laajalti käytössä ollut jatkuvan integraation työkalu. Se on avoimen lähdekoodin ja yhteisöpäivitysten ohjaama. Jenkins on tarkoitettu lähtökohtaisesti organisaation tai muun käyttäjän niin sanotusti paikan päällä tapahtuvaan asennukseen.

Jenkinsin etu on etenkin sen laajennettavuus. Lisäosia ja laajennuspaketteja on paljon, mikä tekee Jenkinsistä hyvin ketterän ratkaisun. Jenkins graafinen käyttöliittymä auttaa rakentamaan yksilöidyn integraatioprosessin mutkattomasti. Hyvin ylläpidetty dokumentaatio on ollut Jenkinsin kulmakivi koko sen menestyksen ajan.

Kuva 6. Jenkinsin graafinen korkean tason käyttöliittymä helpottaa integraatioputkien luomista.

Kuvassa 6 nähdään Jenkinsin intuitiivinen graafinen käyttöliittymä, josta voidaan havainnoida vasemmalla oleva navigointialue. "Manage Jenkins" -napin kautta käyttäjä voi lisätä lisäosia ja laajennuksia. "New Item" -valinta antaa mahdollisuuden luoda muun muassa uusia integraatioputkia.

Integraatioputkia hallitaan ja kehitetään Jenkinsissä Jenkinsfile-tiedoston avulla.

```
pipeline {
    agent any
    environment {
        EXAMPLE_CREDS = credentials('example-credentials-id')
    }
    stages {
        stage('Example') {
            steps {
                /* CORRECT */
                sh('curl -u $EXAMPLE_CREDS_USR:$EXAMPLE_CREDS_PSW https://example.com/')
            }
        }
    }
}
```

Kuva 7. Havainnollistava esimerkki Jenkinsfilen sisältämästä deklaratiivisesta syntaksista [22].

Kuvassa 7 nähdään esimerkki Jenkinsin kohdalla verrattain uudesta deklaratiivisesta syntaksista, joka verrattuna script-vaihtoehtoon on korkeamman tason sekä huomattavasti dogmaattisempi syntaksi.

Jenkinsfilen kanssa on siis mahdollista käyttää myös script-pohjaista syntaksia.

```
node {
    stage('Example') {
        if (env.BRANCH_NAME == 'master') {
            echo 'I only execute on the master branch'
        } else {
            echo 'I execute elsewhere'
        }
    }
}
```

Kuva 8. Jenkinsin script-syntaksi on rakennettu Groovy-ohjelmointikielen pohjalle [22].

Script-syntaksi tarjoaa suurimman osan samasta toiminnallisuudesta ja joustavuudesta kuin sen pohjana toimiva Groovy-ohjelmointikieli [23].

Jenkinsiä voidaan käyttää myös yhdessä muiden integraatiotyökalujen kanssa, kuten esimerkiksi AWS Codepipelinen. Hiljattain julkaistu uusi laajennus mahdollistaa AWS Codepipelinen liittämisen osaksi Jenkins-integraatioputkea. Tällä laajennuksella integraatioprosessi voidaan jakaa niin, että AWS Codepipeline huolehtii versionhallinnan tarkkailusta uusien töiden aloittamisesta sekä build-välivaiheen jälkeisistä julkaisuvaiheista. Jenkins puolestaan tarkkailee näitä Codepipelinen aktiivisia intergraatiotöitä, ja kun tällainen havaitaan, hakee Jenkins määritellyt artifaktit ja aloittaa build-osion integraatioprosessissa. Tämän build-vaiheen jälkeen, kun se on onnistuneesti suoritettu, pakkaa Jenkins artifaktit ja lähettää ne AWS Codepipelinelle, joka jatkaa prosessia tästä eteenpäin. [23.]

### 3.4.2 AWS Codepipeline

Amazonin tarjoama AWS Codepipeline -integraatiotyökalu valittiin toiseksi tutkinnan kohteeksi, koska se on Dynamon muun teknologisen infrastruktuurin tavoin saman entiteetin kehittämä, ja oletuksena on, että tästä voi olla etua integraatioputkia tai niiden osia luotaessa AWS:n pilvipalveluympäristössä.

Codepipeline erottaa itsensä muun muassa Jenkinsistä sen pilvipohjainen toimintamalli. Codepipelinen kanssa ei ole tarvetta ylläpitää omia palvelimia jatkuvan integraation toteuttamista varten vaan riittää, että käyttäjä ainoastaan luo tarvittavat integraatioputket, jotka ovat siis sarja komento- sekä resurssimäärittelyjä, joiden pohjalta automaatio viedään läpi. Integraatioputkien luomisen helppous ja yksinkertaisuus on toinen merkittävä lisäarvoa tuova seikka. Integraatioputkien luonti onnistuu monella eri tapaa, ja tarjoaa siten käyttäjälle mahdollisuuden sopeuttaa niiden luonti tai muokkaus sulavasti osaksi omaa kehitysympäristöä. Eräitä tapoja integraatioputkien kontrollointiin [24]:

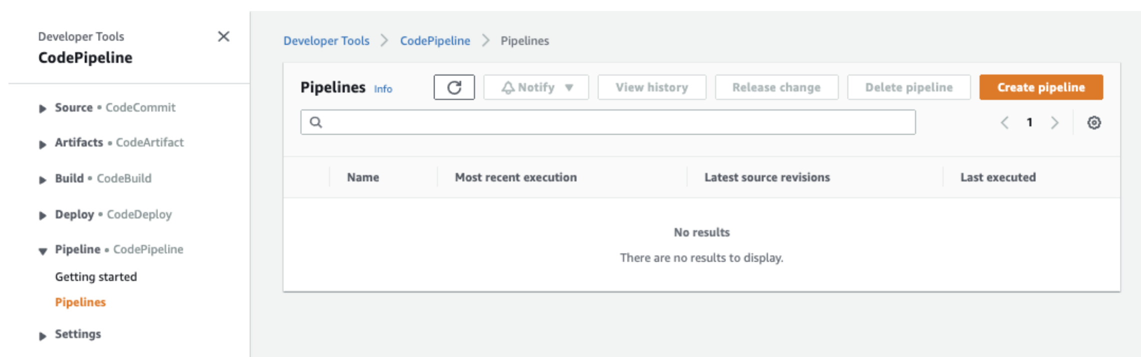
- AWS Web-käyttöliittymä
- AWS Command Line Interface (AWS CLI)
- AWS CloudFormation
- erinäiset AWS SDK:t.

Kuten lista osoittaa, mahdollisuuksia integraatioputkien hallinnointiin AWS Codepipelinessä on useita.

AWS Software Development Kit (SDK) on yksi Codepipelinen hallintatapa. SDK:t jakautuvat ohjelmointikielipohjaisesti ja tarjoavat kattavan joukon funktioita, jotka mahdollistavat kokonaisen AWS Codepipeline -integraatioratkaisun pystyttämisen sekä hallinnoinnin. Tämä onkin oivallinen tapa hallinnointiin silloin, kun organisaatiossa suositaan ohjelmallisempaa

lähestymistapaa, tai hallinnointi halutaan integroida ehkä osaksi jotain toista sovellusta.

Kehitysputken voi määritellä myös AWS:n web-käyttöliittymän kautta (AWS-konsoli). Tämä vaihtoehto voidaan luokitella korkeimmaksi mahdolliseksi, ja se antaakin helpot visuaaliset ja suoraviivaiset työkalut luoda tarvittavat integraatioprosessit ja määritellä siihen liittyviä resursseja ja rutiineja.




Kuva 9. AWS:n web-käyttöliittymä tarjoaa käyttäjälle nopean ja helpon tavan luoda ja hallita integraatioputkia.

Komentotyökalu AWS CLI:n avulla integraatioputkia luodaan suoraan komentorivin kautta. Työkalun käyttäminen vaatii erityisen käyttöjärjestelmäsidonnaisen AWS CLI -ohjelmiston asennuksen, jonka lisäksi työkalu tulee konfiguroida sen saamien oikeuksien sekä tehtävien toimenpiteiden kohteen osalta. Tämä tarkoittaa sitä, että käyttöönotto on verraten raskas. Lisäksi komentorivityökalua potentiaalisesti käytettäessä voidaan pohtia, kuinka helposti ylläpidettävää komentojen hallinnointi on, kun niiden tallentaminen versionhallintaan esimerkiksi mallipohjiin perustuvista rakennusratkaisuista poiketen ei ole aivan triviaalia.

Alhaisimman tason integraatioputkien luonnin ja niiden hallinnan osalta tarjoaa AWS CloudFormation, joka on AWS:n infrastruktuurikoodi (IaC).

```yaml
#####################################################################################
# Prerequisites:
#   - CodeDeploy deploy exists. Update ApplicationName and BetaFleet as needed.
#####################################################################################

Parameters:
  SourceObjectKey:
    Description: 'S3 source artifact'
    Type: String
    Default: SampleApp_Linux.zip
  ApplicationName:
    Description: 'CodeDeploy application name'
    Type: String
    Default: DemoApplication
  BetaFleet:
    Description: 'Fleet configured in CodeDeploy'
    Type: String
    Default: DemoFleet

Resources:
  SourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
  CodePipelineArtifactStoreBucket:
    Type: AWS::S3::Bucket
  CodePipelineArtifactStoreBucketPolicy:
    Type: AWS::S3::BucketPolicy
    Properties:
      Bucket: !Ref CodePipelineArtifactStoreBucket
      PolicyDocument:
        Version: 2012-10-17
        Statement:
          -
            Sid: DenyUnEncryptedObjectUploads
            Effect: Deny
            Principal: '*'
            Action: s3:PutObject
            Resource: !Join [ '', [ !GetAtt CodePipelineArtifactStoreBucket.Arn, '/*' ] ]
            Condition:
              StringNotEquals:
                s3:x-amz-server-side-encryption: aws:kms
          -
            Sid: DenyInsecureConnections
            Effect: Deny
            Principal: '*'
            Action: s3:*
            Resource: !Join [ '', [ !GetAtt CodePipelineArtifactStoreBucket.Arn, '/*' ] ]
            Condition:
              Bool:
                aws:SecureTransport: false
  CodePipelineServiceRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          -
            Effect: Allow
            Principal:
              Service:
                - codepipeline.amazonaws.com
            Action: sts:AssumeRole
      Path: /
      Policies:
        -
          PolicyName: AWS-CodePipeline-Service-3
          PolicyDocument:
            Version: 2012-10-17
```

Kuva 10. Havainnollistava kuva AWS Cloudformation IaC-mallipohjasta YAML-muodossa.

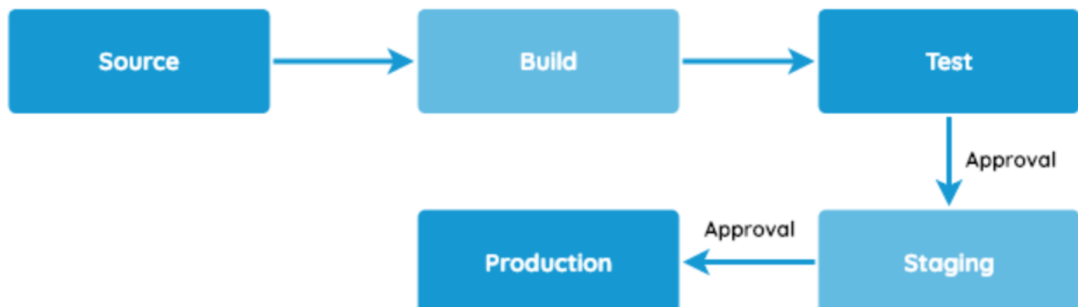
Cloudformation-mallipohjia voidaan luoda niin JSON kuin YAML-muodossa. Kuvassa 10 nähdään Codepipeline-resurssien sekä oikeuksien määrittelyä YAML-tyylisesti. AWS-ympäristössä oikeudet voidaan määritellä erinäisinä

policyinä, usein luodut policyt ovat vielä osa jotain erityistä IAM-roolia. AWS tukee lisäksi "inline"-policyja, jolloin oikeudet määritellään palvelukohtaisesti.

IaC:llä tarkoitetaan ohjelmakoodia, joka hallinnoi fyysisiä laitteita, virtuaalikoneita sekä niihin liittyviä kokoonpanoresursseja. AWS-ympäristössä IaC-infrastruktuurikehityksellä pyritään saavuttamaan sen etuja liittyen [25]:

- Alhaisempiin kustannuksiin, kun palvelinten ylläpitoa ja työntekijöiden fyysisiä ponnisteluja voidaan vähentää.
- Nopeampiin infrastruktuurimuutoksiin, etenkin kun IaC on osana jotain jatkuvan integraation automaatiota.
- Riskien pienenemiseen, kun manuaalisen työn vähentämisen kautta vähennetään myös todennäköisyyttä inhimillisiin virheisiin.

Codepipeline pystyy toteuttamaan kaikki jatkuvaan integraatioon tavallisesti liittyvät vaiheet sekä toimenpiteet.



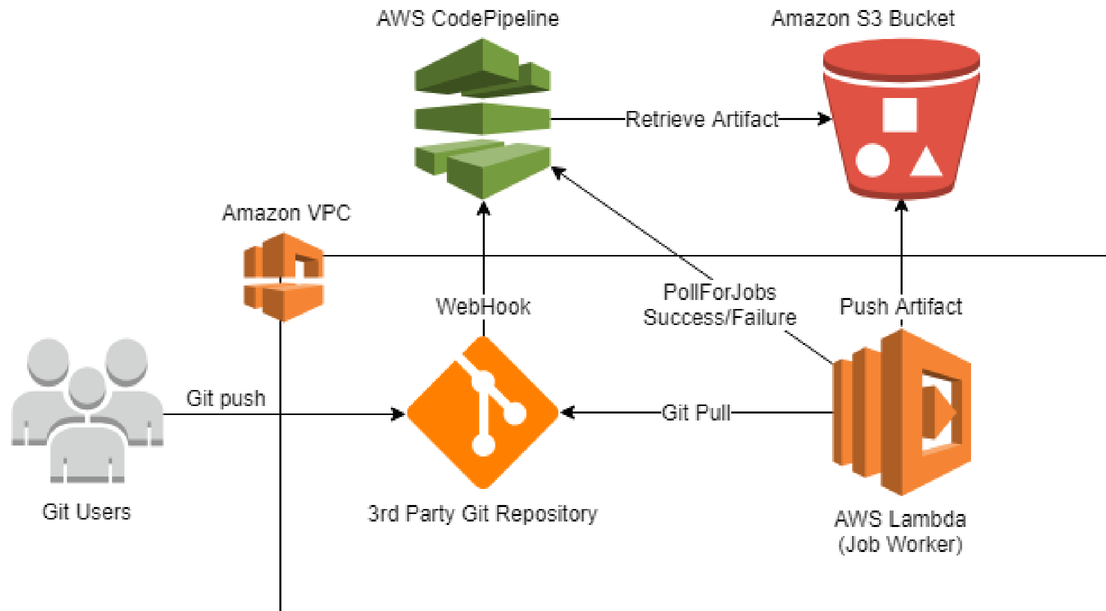

Kuva 11. Automoitujen julkaisuputkien pääsääntöiset vaiheet sekä niiden järjestys.

Kuvassa 11 nähdään, kuinka jatkuvan integraation automaatioputket lähtevät liikkeelle versionhallinnasta ja päättyvät pääsääntöisesti testauksen kautta julkaisuvaiheeseen.

Eräänä automaatiputken peruspalikkana toimii siis versionhallinta. AWS Codepipeline tukee natiivisti Jenkinsiä huomattavasti rajoitetummin eri versionhallintaratkaisuja. Suoraa tukea nauttivat [26]

- AWS CodeCommit
- Github.com
- Github Enterprise Server
- Bitbucket.org.

Natiivisti tuettujen versionhallintaratkaisujen lisäksi AWS Codepipeline mahdollistaa muiden kolmansien osapuolten versionhallintaratkaisujen lisäämisen erityisen webhookkeihin pohjautuvan mukautetun arkkitehtuuriratkaisun avulla. Webhookeilla tarkoitetaan web-kehityksessä mukautettuja takaisinkutsuja, joiden avulla jonkin verkkosivuston tai sovelluksen käyttäytymistä voidaan muuttaa [27]. Kyseessä olevassa Codepipeline arkkitehtuurissa näitä webhookkeja voidaan hyödyntää liittämällä yksi webhook käytettyyn Git-versionhallintaan, jolloin Gittiin tehty muutostoimenpide laukaisee tämän webhookin. Gittiin tehdyn muutoksen jälkeen tämä kyseinen webhook lähettää kutsun Codepipelineen liitettyyn webhookkiin, joka puolestaan aloittaa integraatioputken Codepipelinessa. Integraatioputki laukaisee Lambda-funktion, jonka tehtävä on hakea halutusta kolmannen osapuolen versionhallinasta koodi sekä suorittaa koonti. Lopuksi Lambda-funktio lähettää koonnista syntyneet artifaktit S3-buckettiin, jotka ovat näin myöhemmin Codepipelinen käytettävissä myöhemmissä integraatioputken vaiheissa. Arkkitehtuurin onnistuminen edellyttää, että käytettyyn Git-versionhallintaan pystyy liittämään webhookkeja.




Kuva 12. AWS Codepipeline mahdollistaa myös ei natiivisti tuettujen versionhallintajärjestelmien liittämisen osaksi integraatioputkea erityisen palveluarkkitehtuurin avulla [28].

Kuvassa 12 esitetään, kuinka käyttämällä AWS Lambda -funktioita, WebHookeja ja AWS S3 -buckettia voidaan luoda arkkitehtuuri, joka pystyy integeroida myös kolmannen osapuolen versionhallintajärjestelmiä osaksi AWS Codepipelinea. Tällaisella ratkaisulla pystytään välittämään myös lähdekoodiin liittyvä metatieto versionhallintajärjestelmästä esimerkiksi liittyen integraation aloittaneen commitin tageihin. Samalla tulee todeta kuitekin se tosiasia, että tällaisen hyvin yksilöidyn lisäintegraation tekeminen vaatii aikaa ja perehtyneisyyttä, mikä suoraan kasvattaa vaadittavaa työmäärää.

Kuten jo aiemmin mainittiin, Codepipelinen yksi merkittävä etu on sen integroiminen olemassa oleviin AWS-palveluihin. Dynamo käyttää muun muassa S3-, Cloudfront-, EC2-, LoadBalancer-, Cognito-, Identity and Access Management- (IAM), Lambda- sekä API Gateway -palveluita, jotka kaikki voidaan saada tehokkaasti keskustelemaan keskenään, mikä helpottaa näin esimerkiksi testijulkaisujen tekemistä. IAM tekee lisäksi käyttäjien hallinnan vaivattomaksi ja suoraviivaiseksi. IAM:n kautta organisaatio voi määritellä

tarkasti käyttäjien oikeuksia muun muassa sen osalta kenellä on valtuus toimia AWS Codepipeline -ympäristössä.

AWS Codepipelinen käyttö on potentiaalisesti erittäin kustannustehokasta. Kuten pääsääntöisesti muidenkin AWS-palveluiden kanssa maksaa käyttäjä vain ja ainoastaan oman käyttömääriensä mukaisesti. Kirjoitushetkellä yksi aktiivinen integraatioputki maksaa yhden dollarin kuukaudessa. Ensimmäiset 30 päivää uuden integraatioputken luomisesta ovat ilmaisia, joka on AWS:n tapa edesauttaa Codepipelineen liittyvää testausta. [29.]

Edellä havaittiin, kuinka integraatioputkien hallinnointiin on monia eri lähestymistapoja. Rajoitukset, joita Codepipelineen liittyy suhteessa versionhallintaan, ovat luonteeltaan sellaisia, että ne voidaan kiertää, mikäli näin katsotaan tarpeelliseksi. On syytä kuitenkin käsitellä vielä palvelun ydinajatus, mikä on siis erinäisten palveluiden yhteenliittäminen. Jokainen askel integraatioputkessa määritellään siis korkealla tasolla näiden edellä mainittujen pipeline-tasoisten hallintatyökalujen kautta, mutta askeleen sisällä valittu toiminnallisuus hallitaan kuitenkin työkalukohtaisesti. AWS Codepipeline pystyy integroimaan eri vaihekohtaisia työkaluja laajasti erinäisten liitännäisten avulla.

### 3.4.3 Päätelmät automaatiosta

Integraatiotyökalut pyrkivät antamaan käyttäjille ratkaisuja samanlaisiin ongelmiin. Työkalujen valinnan osalta tuleekin näin ollen kiinnittää huomiota, kuinka eri työkalut sopivat juuri käsillä olevaan projektiin. Avainseikkoja ja suurimpia vedenjakajia ovatkin integraatiotyökalujen asennus- tai toimintaympäristöt, vaaditut taloudelliset ynnä muut resurssit ja panostukset sekä työkalujen käytettävyys ja ketteryys.

Moderni ohjelmistokehitys tavoittelee tehokkuutta ja virheiden minimoimista. Oli kyse sitten testauksesta tai julkaisujen tekemisestä, automaatio vähentää inhimillisten virheiden riskiä kehitysputkessa ja nopeuttaa sovelluksen saattamista kehitysversiosta aina tuotantopalvelimelle saakka. Vaikka testaus

onkin mahdollista ilman tätä integraatioautomaatiota, on huomioitava, että automaation käyttäminen luo varmuutta testien läpiviemisestä jokaisen versionhallintamuutoksen jälkeen, mikä vahvasti edesauttaa kehitysaikaista virheiden havaitsemista ja ylläpitää näin varmuutta ohjelmakoodin laadusta ja virheettömyydestä.

Opinnäytetyön osalta tutkittiin kahta varsin suosittua integraatiotyökalua perustuen niiden olemassa oleviin yhteyksiin työntilaajayrityksen sisäisessä teknologiainfrastruktuurissa.

Jenkinsin osalta yrityksen monet kehitystiimit ovat jo pitkäaikaisia käyttäjiä ja muun muassa palvelimia tälle integraatiotyökalulle on jo delegoitu. Tämän lisäksi tietotaito ylläpitää sekä kehittää kyseiseen integraatiotyökaluun liittyviä integraatioputkia löytyy talon sisältä. Jenkins takaa mukautettavuuden ja yksityisyyden, koska työkalu on täysin organisaation itse hallitsema. Yksityisyysnäkökulma onkin Documillin tapauksessa suuressa painoarvossa yrityksen tietoturvalle antaman korkean tärkeysasteen vuoksi.

AWS Codepipeline oli työn toinen tutkinnan kohde. Selvää oli tämän ratkaisun varsin erilainen lähestymistapa sen pilvipohjaisuuden takia. Myös yhteensopivuus muun AWS-infrastruktuurin osalta on merkittävä. Käyttäjien hallinta IAM-palvelun kautta on sulavaa. Muita AWS-palveluita on helppo sulauttaa osaksi Codepipelineä, esimerkiksi erinäisten AWS Lambda -funktioiden lisääminen, lokitietojen kerääminen AWS Cloudwatch -palvelun avulla tai oikeastaan minkä tahansa muun AWS-palvelun integrointi onnistuu. Usein integrointi tapahtuu varsinaisen integraatioputken kautta, mutta tämän lisäksi palveluiden mukaantuominen onnistuu käyttämällä erityisiä AWS-laukaisimia, mikä on yleinen tapa kaikkialla AWS-ympäristössä, kun palveluita halutaan yhdistellä keskenään. Codepipelinen ylläpidosta vastaa AWS, ja kustannuksiltaan tämän integraatiotyökalun käyttö on varsin edullista kirjoitushetkellä. AWS Codepipelinen rajoitukset liittyivätkin erityisesti, yksityisyysnäkökulman lisäksi, sen versionhallintajärjestelmien tukeen. Tämän osalta natiivisti tuetut järjestelmät Jenkinsistä poiketen ovat jokseenkin kapea-

alainen joukko. Toisaalta kuten analyysissä esitettiin, tarjoaa AWS:n ketterä ympäristö tavan integroida eri palveluja keskenään, mikä mahdollistaa sellaisen arkkitehtuurin luomisen, jonka avulla on siis mahdollista laajentaa tätä versionhallintajärjestelmien joukkoa ilman, että käyttäjäorganisaatio vaarantaisi lähdekoodiin liittyvän metatiedon saatavuutta.

Molemmat työkalut ovat erilaisia, omilla vahvuuksillaan. Codepipelinea tukee yritysympäristössä selkeä pitkäaikainen ja jatkuva muutos kohti palvelitonta arkkitehtuuria. Tämä nykyajan SaaS-megatrendi juontaa juurensa juurikin siitä, että näin yritys pystyy keskittymään olennaiseen, eli varsinaiseen omaan tuotteeseensa, palvelinten hallinnan ja ylläpidon sijaan. Jenkins on puolestaan todistettu ratkaisu, ja kirjoitushetkellä ei ole tiedossa, kuinka paljon painoarvoa tämän ratkaisun datan yksityisyysnäkökulma tuo tälle valinnalle. Kokonaisarviointina analyysin pohjalta arvioin siis, että molemmat työkalut voidaan esitellä osana kehityskaariehdotuksia.

## 3.5 Muut käytännöt

Opinnäytetyössä muilla käytänteillä viitataan ohjelmistonkehitystekniikoihin, jotka ottavat kantaa kehitettävien ominaisuuksien määrittelyjen muotoon ja hyväksyttävyyskriteereihin. Lisäksi myös koodin tarkistukseen liittyvät toimenpiteet luetaan osaksi tätä aihepiiriä.

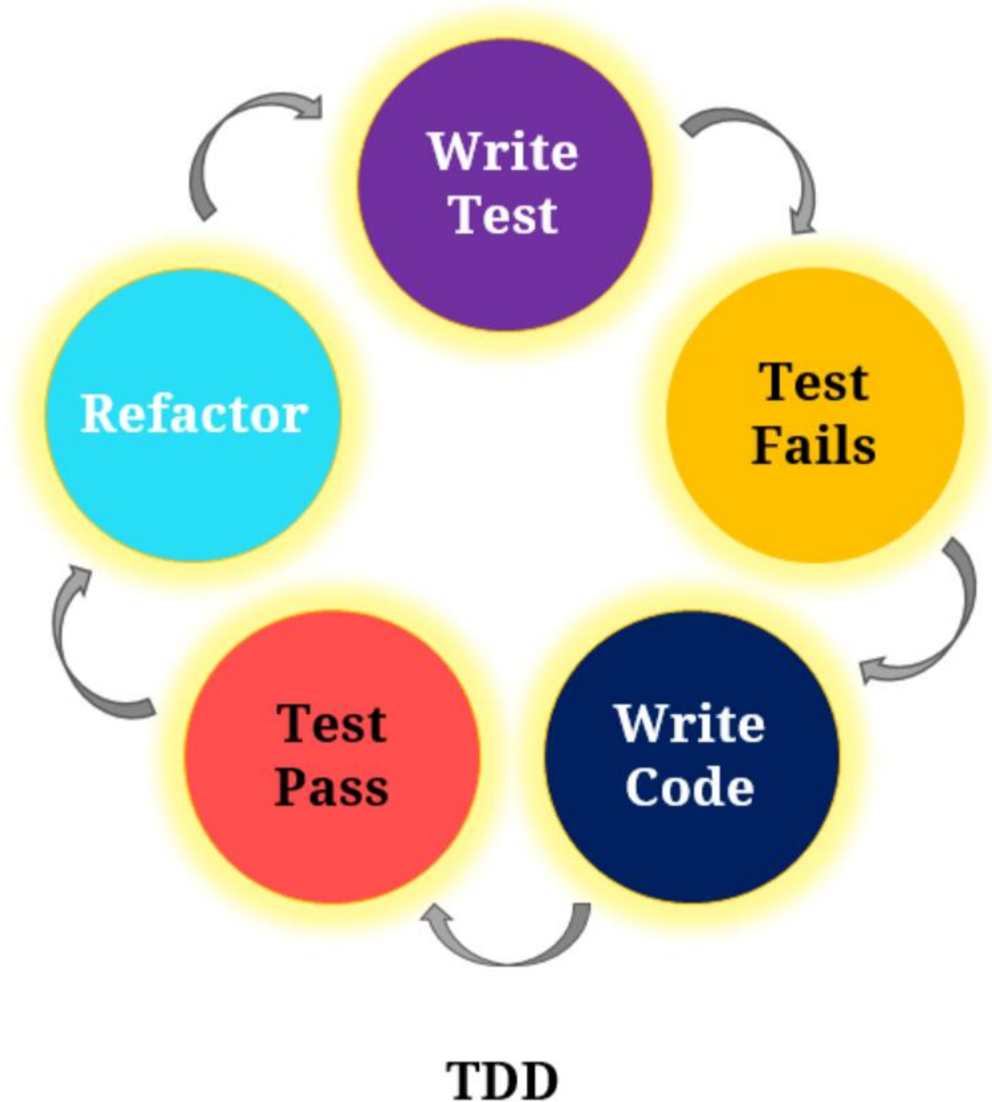
Ohjelmistonkehitystekniikat auttavat valitsemaan oikean testausstrategian kertomalla muun muassa, minkälaiseen testaukseen ominaisuuden hyväksyttävyyden tulee perustua. Kehitystekniikat on mahdollista tulkita olevan osa kehityskaaren laadunvarmennuselementtejä. Nämä tekniikat yksilöivät uusien ominaisuuksien kehitykseen osallistuvat tahot määrittelyjen muodon sekä testauksen painopisteen.

Merkittäviä ohjelmistonkehitystekniikoita ovat muun muassa:

- Test-Driven Development (TDD)
- Behavioral-Driven Development (BDD)

- sekä Acceptance Test-Driven Development (ATDD).

TDD on kehitystekniikka, jossa ohjelmoijan painopiste on voimakkain. Tässä mallissa testejä tehdään ominaisuuden jokaiselle pienelle toiminnallisuuden osalle. Järjestys kehityksessä on nimenomaan kirjoittaa ensin automatisoituja yksikkötestejä pohjautuen toiminnallisuusmäärittelyille, ja vasta tämän jälkeen testit läpäisevä ohjelmakoodi. Toiminnallisuuden laajentaminen tapahtuu kirjoittamalla uusi testi, ja vasta kun tällainen uusi testi epäonnistuu, tullaan laajentamaan varsinaista ohjelmakoodia eteenpäin. Koska testin tulee epäonnistua ennen uuden ohjelmakoodin kirjoittamista, vähentää TDD teoriassa testiscriptien duplikaatioiden määrä,. Tekniikka pyrkii antamaan vastauksen yksinkertaiseen kysymykseen - Onko ohjelmakoodi hyväksyttävä? [30.]




Kuva 13. TDD-kehitystekniikan eri vaiheiden järjestys.

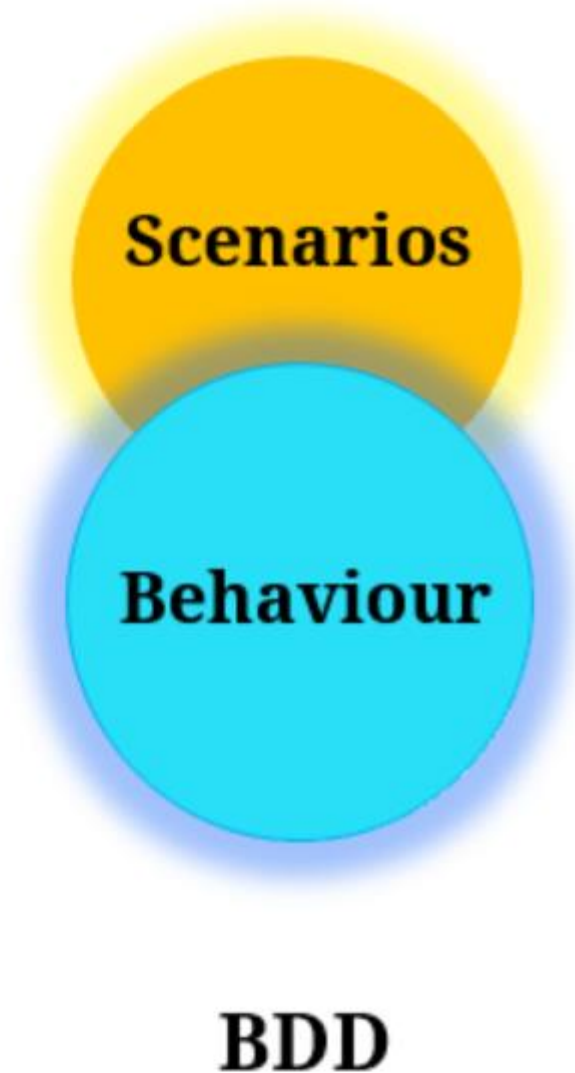
TDD:n potentiaalisia hyviä puolia [30]:

- auttaa vähentämään ohjelmakoodin muokkauksiin kuluvaa aikaa
- auttaa havaitsemaan virheitä varsin nopeasti
- auttaa saamaan nopeampaa palautetta
- rohkaisee kehittämään siistimpiä ja parempia malleja
- parantaa kehittäjän tuottavuutta
- antaa mille tahansa ryhmän kehittäjälle mahdollisuuden jatkokehittää ohjelmakoodia alkuperäisen kehittäjän poissadollessa

- antaa ohjelmoijalle luottamusta muuttaa sovelluksen suurta arkkitehtuuria
- tuottaa kauasulottuvaa ohjelmakoodia, joka on joustavaa ja helppoa ylläpitää.

Mikäli ohjelmistokehitystekniikan halutaan olevan enemmän systeemitason käyttäytymiseen painottuvaa, on syytä tarkastella BDD-tekniikkaa. BDD:ssä määritellään useita eri tapoja kehittää uusi toiminnallisuus pohjautuen sen käyttäytymiseen. Useimmissa käyttötapauksissa "Given-When-Then"-lähestymistapa on pääasiallinen tyyli kirjoittaa testitapauksia. Esimerkkinä tästä lähestymistavasta ovat [31]:

- Given - käyttäjä on täyttänyt oikeat sisäänkirjautumistiedot
- When - käyttäjä painaa sisäänkirjautumispainiketta
- Then - näytä onnistuneen kirjautumisen huomautusviesti.




Kuva 14. BDD-kehitystekniikan vallitsevat periaatteet ja lähtökohdat: skenaariot sekä käyttäytyminen.

"Given-When-Then"-tekniikka auttaa siis luomaan skenaarioita siitä, miten toiminallisuuden tulisi toimia loppukäyttäjän näkökulmasta. BDD:lle ominaista on sen painotteisuus kehitettävän ominaisuuden käyttäytymisen tarkastaluun sekä kielen osalta skenaarioiden luominen sanallisesti niin, että myös "ei-tekninen" henkilö pystyy ne ymmärtämään. BDD-pohjaiseen kehitykseen osallistuu TDD:tä huomattavasti laajempi toimijakunta: ohjelmoijat, laadunvalvojat ja asiakkaat/käyttäjät. [31.]

BDD:n etuja listattuna [31]

- auttaa tavoittamaan laajemman yleisön käyttämällä ei-teknistä kieltä
- keskittyy siihen, miten järjestelmän tulisi käyttäytyä asiakkaan ja kehittäjän näkökulmasta
- BDD on kustannustehokas tekniikka
- vähentää käyttöönoton jälkeisten vikojen todentamiseen tarvittavia ponnisteluja.

ATDD on kehitystekniikoista se, jota käytetään, kun halutaan noudattaa hyväksymistestauspainotteista kehitystä. Hyväksymistestit kirjoitetaan käyttäjän näkökulmasta. Testeissä keskitytään pääasiassa järjestelmän toiminnallisen käyttäytymisen tyydyttämiseen. Tällä tekniikalla pyritään vastaamaan kysymykseen toimiiko koodi odotetulla tavalla?

BDD:n tavoin ATDD on kehitystekniikka, joka keskittyy saavuttamaan ominaisuudelle asetetut vaatimukset, toisin kuin TDD:ssä, jossa toiminnallisuuden implementaatio on keskeistä. ATDD parantaa kehittäjien, käyttäjien ja laadunvarmistajien yhteistyötä, kun he keskittyvät yhdessä hyväksymiskriteerien määrittelyyn. Seuraavassa on joitakin ATDD:n keskeisiä käytäntöjä [32]:

- reaalimaailman skenaarioiden analysointi ja niistä keskusteleminen
- hyväksymiskriteereistä päättäminen testiskenaarioita varten
- hyväksymistestitapausten automatisointi
- vaatimustapausten kehittämiseen keskittyminen.

Opinnäytetyön lähtötilanteessa Dynamon ohjelmistokehitystekniikan voidaan havaita sisältävän pääasiallisesti BDD-tyylisiä käytänteitä. Kehitys pohjautuu asetettuihin sanallisiin vaatimuksiin. Kehitetty ominaisuus läpäisee testauksen, kun sen voidaan todeta käyttäytyvän halutulla tavalla. Testaukseen osallistuvat kehittäjät, laadunvalvojat sekä tietyissä tilanteissa myös loppukäyttäjät. Pohtia tuleekin, onko tarpeellista muuttaa käytänteitä TDD:n tai ATDD:n suuntaisiksi. Vaihtoehtoisten käytänteiden tarjoamat edut tuleekin asettaa vastakkain BDD:n tarjoaman kustannustehokkuuden kanssa. Historiallisesti Dynamo on kyennyt

selviytymään asiakkaiden vaatimuksista varsin mallikelpoisesti. BDD:tä puoltavana seikkana voidaan myös pitää kehittäjien suhteellisen tarkkarajaisia vastuualueita, joka syö muun muassa TDD:n etua tiedon siirtoon kehittäjältä toiselle, kun kehittäjä teoriassa pysyy aina samana. Kysymys kuuluukin, tuovatko TDD- tai ATDD-kehitystekniikat lisäarvoa tuotteelle. Taustajärjestelmän uusien Lambda-funktioiden yksikkötestit ovat olleet kehitysryhmän tavoitteena. Tulisiko kyseiset yksikkötestit tehdä TDD-mallisesti vai viekö uuden käytänteen opettelu liikaa resursseja?

# 4 Testauksen ja laadunvarmistuksen kehitysehdotus

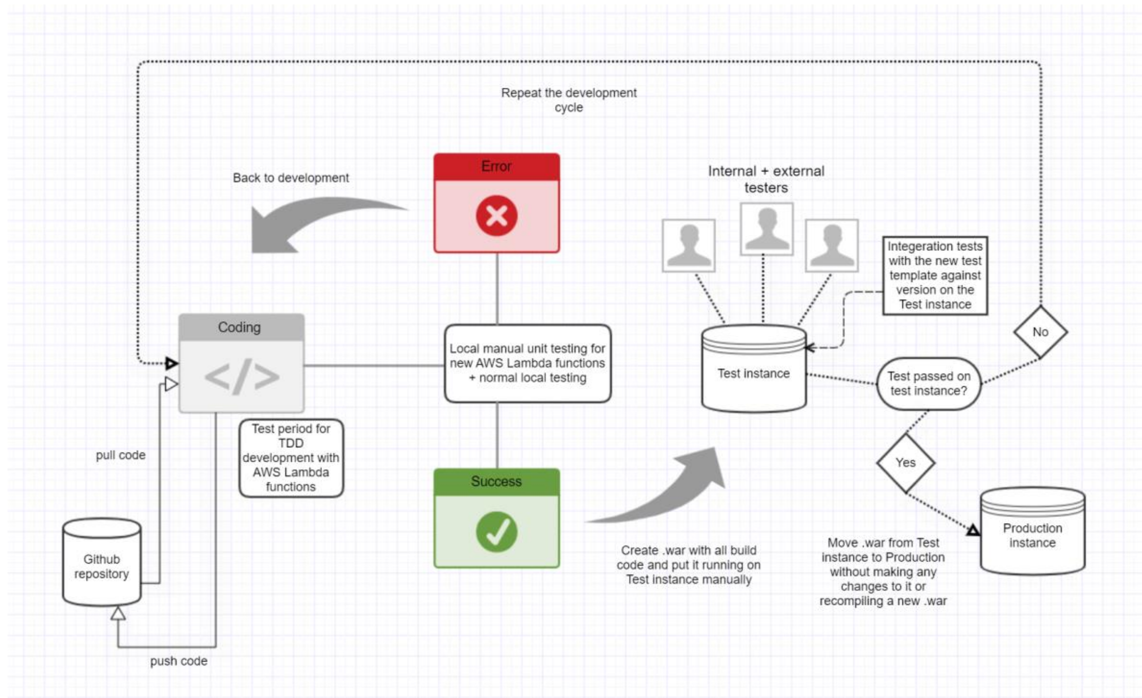
Opinnäytetyön kehitysehdotus Dynamoa varten pohjautuu aiempaan analyysiin eri menetelmien ominaisuuksista. Seuraavaksi pyritään esittämään ehdotus, joka mahdollistaa lähtötilanteeseen verrattuna paremmin testatun ja laadunvarmennukseltaan tehostetun version Dynamon kehitysputkesta.

Kehitysehdotus on jaettu lähtökohtaisesti kahteen osaan. Ensimmäinen osa käsittelee menetelmiä, jotka suositellaan implementoitavan välittömästi. Toinen osa puolestaan käsittelee tulevaisuuteen mahdollisesti kohdistuvia suositeltuja lisäyksiä kehitysputkeen. Jokaisen menetelmän kohdalla pyritään esittämään sellainen vaihtoehto, joka on tosiasiallisesti implementoitavissa resurssit sekä kustannustehokkuus huomioon ottaen. Kustannustehokkuutta lähestytään näkökulmasta, josta implementaatiosta vastaa yksi henkilö. Implementoijan työpanos pitää sisällään toteutukseen liittyvän opiskelun sekä tuottavan työn. Aikarajana kaikille ehdotettujen toimenpiteiden implementoinneille voidaan pitää noin yhtä kvartaalia, muun työn ohessa.

## 4.1 Suositellut välittömät toimenpiteet

Välittömät toimenpiteet ovat lähtökohtaisesti sellaisia, joilla päästään työn tavoitteeseen mahdollisimman tehokkaasti. Testaus, työn keskeisimpänä tavoitteena, on syytä implementoida tavalla, jolla Dynamon laajan ja ajallisesti hajautetun koodipohjan tuomat hankaluudet voidaan ohittaa, kuitenkin niin, että

tehtyjen testien avulla voidaan varmistua enemmissä määrin sovelluksen kriittisten komponenttien ja yksiköiden toiminnoista yksin sekä yhdessä. Testauksen lisäksi myös versionhallinta on osa-alue, joka on välittömästi uusittavissa ja joka tarjoaa selkeitä etuja - nyt sekä tulevaisuudessa.




Kuva 15. Tavoitteellinen kehitysputki välittömien toimenpiteiden jälkeen.

Kuvassa 15 esitetään kehitysputki sellaisena, kuin sen tulisi olla välittömien toimenpiteiden toteutuksen jälkeen. Versionhallinta on kuvattuna siihen esitettyjen muutosten takia.

### 4.1.1 Testaus

Testauksen osalta regressiiivisen yksikkötestauksen rakentaminen ei ole suositeltavaa sen kompleksisuudesta ja aikavaatimuksesta johtuen. On tähdättävä suurempaan kokonaisuuteen. Systeemitason integraatiotestaus tarjoaa pohjan käydä läpi sovelluksen kriittiset toiminnot kohta kohdalta ja varmentua niiden yhteistoiminnasta. Dynamon teknologisesta murrosvaiheesta johtuen hetki on kuitenkin otollinen uusien käytänteiden sisäänajoa varten.

Lambda-funktiot ovat nykynäkemyksen mukaan sovelluksen taustajärjestelmän tulevaisuus, ja mikäli yksikkötestausta halutaan hyödyntää, on ajankohta testien luomisen aloittamiselle tässä murrosvaiheessa.

Ehdotetut testaustoimenpiteet ovat:

- systeemitason integraatiotestaus testimallipohjaa hyödyntäen (REST API -versio)
- yksikkötestaus AWS Lambda -funktioille.

Integraatiotestauksen implementoinnin tavaksi suositellaan erityistä testausmallipohjaa. Ajatus on, että testimallipohja sisältäisi mallipohjalogiikan, joka kutsuisi jokaista testattavaa Dynamon tarjoamaa logiikkakomentoa. Integraatiotestaus ehdotetaan laukaistavaksi erityisen REST-rajapinnan kautta. Rajapinta voi vastaanottaa testimallipohjan sijainnin parametrina, mutta myös ilman parametreja tulisi rajapintaa voida käyttää joillakin perusarvoilla. Kutsu laukaisisi normaalin Dynamon dokumenttigeneraatioprosessin käyden läpi testimallipohjan logiikan.

Dynaamisen tietoaineksen kerääminen tapahtuu normaalissa generaatioprosessissa Dynamon käyttämien ”screen step” -dialogien kautta. Mikäli integraatiotestissä kerätään dynaamista tietoainesta, API-pohjainen manuaalisesti laukaistu integraatiotestaus edellyttää tällöin testaajan välitöntä läsnäoloa ja reagointia testiprosessin aikana esitettyihin dialogeihin. Testauksen toteutusvaiheessa tuleekin pohtia, missä määrin dynaamista tietoainesta tarvitsee kerätä testiprosessin aikana. Myöhemmin laadunvarmennuksen kehittyessä jatkuvaan integraatioon, CI-putken sisältämä automaatio esimerkiksi käyttöliittymä emulaation avulla pystyisi huolehtimaan tästä dynaamisen tietoaineksen antamisesta. Voidaan todeta, että testimallipohjan rakentaminen mahdollisimman vähäiselle riippuvuudelle käyttäjälähtöisestä tietoaineksesta nopeuttaa testin läpivientiä sekä testauksen ylläpitoa.

Dynamon dokumenttigeneraatio on vahvasti integroitu käyttämään Salesforcea tiedon lähteenä. Tästä johtuen generaatioprosessit hyödyntävät kommunikaatiossaan Salesforcen kanssa OAuth2-protokollan mukaista

autentikoitumista. Prosessin aluksi Dynamo hankkii itselleen generaation aloittaneen henkilön tarkasti rajatut oikeudet päästä käsiksi resursseihin, ikään kuin käyttäjä olisi itse tekemässä kyseisiä kutsuja. Dynamo toimii niin sanotusti prosessin aloittaneena henkilönä Salesforcen kanssa kommunikoidessaan. OAuth2:n mukaisesta kirjautumisesta Dynamo saa käyttöönsä rajatun ajan voimassaolevan "access tokenin", jonka liittäminen osaksi Salesforce-kutsuja antaa Dynamolle oikeudet päästä käyttäjään liittyviin resursseihin.

Testimallipohjan osalta tulee varmentua siitä, että testiprosessi käyttää oikeita ja toimivia tokeneita kommunikaatioon. Sinänsä REST API -pohjainen testaus voisi käyttää APIa kutsuvan käyttäjän oikeuksia, eli toimia kuten normaalikin dokumenttigenerointi. On kuitenkin tiettyjä etuja, jos olisi olemassa erityinen Salesforceen luotu testikäyttäjä, jonka oikeuksia sekä resursseja testeissä hyödynnettäisiin. Tällainen erillinen testikäyttäjä toisi etuja myös AWS Lambda -funktioiden yksikkötestauksessa, jolloin on funktiota testattaessa usein tarpeellista päästä keskustelemaan Salesforcen kanssa. Tästä syystä integraatiotestausrajapinnan tulisikin käyttää Salesforceen tehtävän edellä mainitun testikäyttäjän oikeuksia. Näin myös testaukseen liittyvät käyttäjäkohtaiset resurssit on keskitetty selkeästi. Testikäyttäjän oikeuksien hyödyntäminen vaatii taustajärjestelmän muutoksia. Taustajärjestelmää tuleekin muuttaa testirajapinnan osalta niin, että generointiprosessin alussa AWS Key Management Servicestä (AWS KMS) haetaan sinne testauksen toteutusvaiheessa tallennettu testikäyttäjän "refresh token". Refresh token on OAuth2-protokollan yksi työkalu, jolla OAuth2-autentikoitumista tarjoavalta palveluntarjoajalta voidaan pyytää aina uusi access token. Refresh tokenit ovat lähtökohtaisesti Salesforcen ja Dynamon käyttötapauksessa voimassa siihen asti, kunnes ne erityisesti poistetaan käytöstä. Refresh tokeneiden pysyvyyden takia on erityisen tärkeää säilöä niitä huolellisesti. AWS KMS -palvelu on Dynamossa jo aiemmin käytetty teknologia säilömään arkaluontoisia avaimia sekä muita vastaavia resursseja. Palvelu tarjoaa joustavaa ja suojattua avainten säilöntää eri AWS-palveluissa hyödynnettäväksi. AWS KMS käyttää FIPS 140-2 -standardin mukaisesti validoituja tai parhaillaan validoitavia laitteiston turvamoduuleja. [33.]

Testitulosten validointi on eräs testauksen kulmakivistä. Dynamon tapauksessa dokumenttigenerointi tuottaa taustajärjestelmässä lähtökohtaisesti HTML-muotoista dataa, joka sitten muutetaan PDF:äksi erillisen mikropalvelun avulla. Taustajärjestelmässä jatkuvasti saatavilla oleva HTML-data mahdollistaa muun muassa merkkijono-pohjaisen vertailun. Ehdotetaan, että lopputuloksen osalta testien validointi tapahtuisi edellä mainitulla merkkijono-vertailulla. Tekniikka edellyttää, että on olemassa täydellinen HTML-merkkijono, jota vasten testitulosta verrataan. Vertailtavan HTML-merkkijonon rakentamiseen ja ylläpitoon liittyvät haasteet tulee ottaa huomioon toimeenpanovaiheessa.

AWS Lambda -funktioiden yksikkötestaus on toinen testaukseen liittyvä välitön toimenpide. Dynamon aiemman linjauksen mukaisesti uudet taustajärjestelmän toiminnallisuudet luodaan lähtökohtaisesti palvelittomina ratkaisuina AWS-ympäristöön. Lambda-funktiot voidaan nähdä aina omana yksikkönään ja niiden testaus parantaa merkittävästi laadunvarmistuksen kokonaisuutta. Myös TDD-kehitystekniikan sisäänajo on luontevaa aloittaa Lambda-funktioista, joita ei rasita vanha koodipohja.

AWS:n tarjoamat palvelittomat funktiot ovat itsenäisiä ohjelmakoodikokonaisuuksia, joita kutsutaan pääsääntöisesti joko AWS API Gateway REST APIn kautta, tai suoraan eri kehityskirjastojen (SDK) avulla, niiden tarjoamalla invoke API:lla. Lambda-funktiot saavat niitä kutsuttaessa ennalta määritellyt parametrit: event ja context sekä API Gatewayn yhteyteen integroidussa versiossa parametrin callback. Nämä parametrit ovat hieman eri sisältöisiä riippuen siitä, mitä kautta funktioita kutsutaan. Testauksen kannalta AWS tarjoaa keinoja luoda näistä funktioille sisääntulevista parametreista niin sanottuja mock-versioita eli keinotekoisia syötteitä. Lambda-funktioiden natiivitestaus perustuukin pitkälti tähän parametrien keinoteikoiseen rakentamiseen ja näiden Lambda-funktioille syöttämiseen.

Mock-kutsuja ja niihin pohjautuvia testejä on mahdollista tehdä suoraan AWS:n käyttöliittymästä, mutta testien tämän tyyppinen hallinnointi ei ole kovinkaan skaalautuvaa tai helppokäyttöistä. Dynamo hyödyntääkin Lambda-funktioiden

rakennuksessa AWS SAM -työkalua, joka on AWS:n tarjoama erillinen ohjelma, erityisesti suunniteltu palvelittomien arkkitehtuurien hallinnointiin AWS-ympäristössä. AWS SAM tarjoaa ratkaisuja myös testauksen kannalta mahdollistamalla funktioiden paikallisen testiympäristön ylläpidon [34]. AWS SAMin testiympäristöt on jaettu kolmeen osaan, joista puhdas Lambda-rajapinta perustuen SAM-mallipohjassa määriteltyihin resursseihin on testauksen kannalta kauaskantoisin työväline.

SAM-mallipohjat ovat AWS SAM -työkalun käyttämiä tavallista CloudFormation-mallipohjaa korkeamman tason yaml-pohjaisia arkkitehtuurimäärittelyjä. SAM-työkalu osaa lukea näitä SAM-mallipohjia ja generoida niiden pohjalta alemman tason CloudFormation-mallipohjan, jonka pohjalta AWS pystyy sitten luomaan halutun pilvipalveluarkkitehtuurin. AWS SAMin toimintaympäristö on rajattu vain oleellisiin pilvipalveluihin liittyen palvelittomiin sovellusarkkitehtuureihin. SAM keskittyy etenkin Lambda-funktioiden, API-rajapintojen, tietokantojen sekä eri toimintolaukaisimien määrittelyihin.

Puhtaan Lambda-rajapinnan lisäksi AWS SAM kykenee pystyttämään myös kokonaisen API Gateway REST APIa emuloivan paikallisen rajapinnan. Lisäksi SAM-työkalu mahdollistaa myös komentorivikutsut suoraan yksittäiselle Lambda-funktiolle. Komennot joilla Lambda-funktioita voidaan testata AWS SAMin avulla [35]:

- *sam local start-lambda* (Lambda API, jota on mahdollista käyttää SDK:iden *invoke* rajapinnan kautta)
- *sam local start-api* (REST APIa emuloiva ympäristö)
- *sam local invoke* (komentorivi kutsu yksittäiselle funktiolle).

AWS SAMin testiympäristöt hyödyntävät Docker-kontitusteknologiaa. Tämä tarkoittaa sitä, että Docker tulee olla asennettu paikallisesti, jotta SAMin testiympäristö voi toimia. Docker paketoi sovelluksen ja sen tarvitseman sovellusalustan, kuten PHP, Node.js tai muun sellaisen standardilla tavalla, ajaa niitä kevyissä konteissa eli käyttöjärjestelmätason virtuaalipalvelimissa (container virtualization) sekä linkittää kontit keskenään.

AWS SAMin Lambda-rajapinnan kautta SAM-mallipohjassa määriteltyjä Lambda-funktioita voidaan kutsua AWS SDK:iden tai AWS CLI:n kautta niiden tarjoamalla invoke-rajapinnalla.

AWS SAMin hyödyntäminen Lambda-funktioiden testauksessa sen natiiveilla tukityökaluilla mahdollistaakin koko koodipohjan säilyttämisen versionhallinnassa keskitetysti - toisin kuin, mitä esimerkiksi testien rakennus puhtaasti korkean tason käyttöliittymän kautta mahdollistaisi. Lisäksi paikallinen testaus on ensisijaisen tärkeää, kun ajatellaan testien myöhemmin liittämistä osaksi jatkuvaa integraatiota ja testien automatisointia.

Oleelliset seikat tiivistettynä liittyen Lambda-funktioiden onnistuneeseen paikalliseen testaukseen:

- varmistettu tarvittavat paikalliset asennukset: SAM CLI ja Docker
- yksikkötestit jokaiselle määritellylle Lambda-funktiolle hyödyntäen sopivaa testauskirjastoa (suosittuja Node.js -ympäristössä käytettyjä kirjastoja: Mocha, Jest ja Jasmine)
- Lambda-rajapinnan pystyttäminen AWS SAM CLI -komennolla: *sam local start-lambda*
- testien ajo ja tulosten validointi.

### 4.1.2 Versionhallinta

Dynamon nykyinen versionhallintajärjestelmä pohjautuu yrityksen omalla palvelimella, yrityksen omissa tiloissa toimivaan Git-versionhallintaratkaisuun. Git on hyvin laajalti käytössä oleva yleisesti hyväksytty tapa ylläpitää lähdekoodin eri versioita.

Laadunvarmennusta yleisesti parannettaessa on kuitenkin tutkittava, voiko nykyistä ratkaisua parantaa, mahdollisesti tulevaisuuden muutoksia silmällä pitäen. Eräitä nykyisen ratkaisun heikkouksia on sen monoliittisuus. Nykyisessä mallissa sama lähdekoodisäilö sisältää useita eri projekteja Dynamon lisäksi. Myös käytäntö haarojen nimien suhteen ei ole alan yleisten toimintatapojen

mukainen. Tulevaisuudessa mahdollisesti voimallisemman toteutettava koodin tarkistus ei saa nykyisestä versionhallintajärjestelmästä minkäänlaisia synergiaetuja.

Olemassa olevien ongelmakohtien ratkaisemiseksi ehdotetaan nykyisestä versionhallintajärjestelmästä siirtymistä Github-versionhallintapalveluun. Tämän ratkaisun etuja ovat etenkin sen pitkän tähtäimen synergiaedut muiden palveluiden kanssa. Github on yksi luontaista tukea saavista ratkaisuista muun muassa AWS Codepipelinen jatkuvan integraation palvelun kanssa. Koodin tarkistus on helppoa implementoida osaksi sovelluksen kehityskaarta hyödyntämällä Githubin "pull request" -toimintoja. Pull requestien etu on etenkin niiden visuaalisuus. Koodi löytyy käyttöliittymän kautta sekä koodin kommentointi, että palaute tapahtuu saman näkymän välityksellä. Myös pull requestin toiminnallisuushaaran liittäminen osaksi kehitys- tai päähaaroja on kätevää ja tapahtuu yhdellä painalluksella palautteen jälkeen, mikäli näin halutaan. Dynamon nykyinen järjestelmä koodin tarkistuksen suhteen sen sijaan vaatii tarkastajan hakevan ensiksi oikean toiminnallisuushaaran Gittiä käyttämällä manuaalisesti. Koodin tarkistuksen jälkeen kommentointi tapahtuu muita väyliä pitkin, kuten esimerkiksi Jira tai Slack. Lopuksi kehittäjä jonka koodi on hyväksytysti tarkistettu voi yhdistää toiminnallisuuden johonkin päähaaraan, manuaalisesti. Github voidaan siis nähdä kehitystä helpottavana tekijänä. Sen avulla puhdas koodiin liittyvä kommentointi on myös mahdollista keskittää yhteen paikkaan. [36.]

Versionhallintaympäristön muutos antaa oivan mahdollisuuden päivittää muita versionhallinnan käyttöön liittyviä käytänteitä. Kehityshaarojen suhteen ehdotetaan siirtymistä selkeään kehitys ja päähaara -jakoon. Näiden keskeisten koodihaarojen lisäksi kehityksessä tulee käyttää erillisiä toiminnallisuushaaroja, jotka siis sisältävät kyseisten uusien toiminnallisuuksien koodin. Kun toiminnallisuus on läpäissyt tarvittavat laadunvarmennusvaiheet, se voidaan liittää osaksi sen hetkistä kehityshaaraa. Kehityshaara voi olla yleinen "develop"-haara, tai jokaiselle isommalle sovellusversiolle luotu erillinen haara, esimerkiksi "branch-v1", "branch-v2" ja niin edelleen. Versionhallinnan

päähaara, joka kulkee uusien yleisten nimeämiskonventioiden mukaan "main"-nimellä, olisi tarkoitus pitää mahdollisimman selkeänä commit-historiansa puolesta. Tämän saavuttamiseksi kehittäjien on mahdollista käyttää muun muassa Gitin "git rebase" -komentoa, jonka avulla ylimääräisiä kehityksen aikana muodostuneita välicommitteja voidaan tiivistää yhdeksi commitiksi [37].

Käyttöliittymä- ja taustajärjestelmäkoodin eriytys omiin säilytyspaikkoihinsa on järkevää tehdä versionhallintauudistuksen yhteydessä. Eriyttäminen helpottaa jatkuvan integraation implementointeja. Taustajärjestelmällä ja käyttöliittymällä on myös nykyisessä mallissa oma julkaisusyklinsä, joten myös tältä kannalta tarkasteltuna oma säilytyspaikka Gitissä on perusteltua.

Versionhallintaan liittyvät muutosehdotukset:

- Githubin käyttöönotto
- uusien versionhallinnan nimeämis- ja toimintakäytänteiden hyödyntäminen
- koodin tarkistuksen sisäänajo Githubin pull request -toiminnallisuuden avulla
- käyttöliittymä- ja taustajärjestelmäkoodin säilytyspaikan eriyttäminen.

### 4.1.3 Muut käytännöt

Kuten työn aikana on aiemmin selostettu, lasketaan erilaiset kehitystekniikat sekä koodin tarkistus "muut käytänteet" -otsikon alle.

Dynamon nykyinen kehitystekniikka on luonteeltaan BDD-tyylistä kehitystä, jossa toiminnallisuuden hyväksyttämiseen osallistuu toimijoita laajalta skaalalta aina yrityksen sisäisistä operaattoreista loppukäyttäjään saakka. Eräitä nykyisen kehitystekniikan ratkaisemattomia ongelmia ovat kuitenkin kysymykset liittyen henkilöstövaihdoksiin, kehittäjien osaamisen kehittämiseen sekä nykyisen tekniikan suhteellisen hitaaseen virheraportointiin. On siis mahdollista

pohtia, toisiko toisenlainen kehitystekniikka ratkaisun ongelmakohtiin tai voisiko nykyistä mallia laajentaa kattamaan tavallaan useamman eri kehitystekniikan. Ero ATDD:n ja TDD:n välillä kiteytyy lähinnä testien ajoitukseen ja laajuuteen. Koska Dynamolla ei ole varsinaista testausryhmää tai henkilöä, joka keskittyisi pitkälti testien ylläpitämiseen, ei ATDD:n mukaisten hyväksymistestien tekemistä voida pitää kovinkaan realistisena. TDD toisaalta on hyvin kehittäjälähtöinen tekniikka, joka teoriassa lupaa pitkälle kantavia hyötyjä kehittäjäpainotteisille ohjelmointiyrityksille. Näillä perusteilla ehdotetaan, että Dynamon sisällä käyttöön kehittäjälähtöinen TDD.

Uusien käytäntöjen ja teknologioiden sisäänajo on yrityksissä prosessi, joka vaatii panostuksia ja aikaa. Tästä syystä Dynamossa tapahtuvaa taustajärjestelmän muutosprosessia voidaan pitää tässäkin suhteessa oivana ajankohtana. Koska kehitystekniikka on kuitenkin uusi, ehdotetaan sisäänajon suhteen pitää rajatun pituinen koejakso. Koejakson aikana kaikki uudet AWS Lambda -funktiot tulisi rakentaa TDD-tekniikkaa käyttäen. Jälkikäteen kokemukset käydään kehitysryhmän sisällä läpi ja päätetään pitkän tähtäimen toimenpiteistä.

Ehdotus:

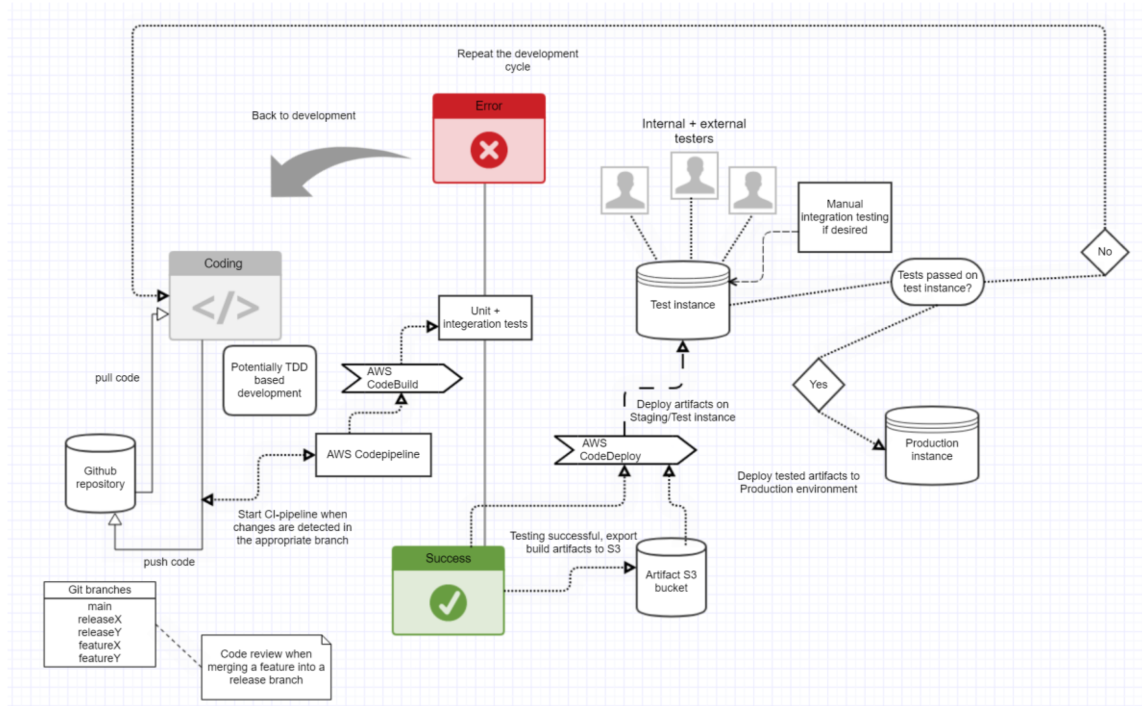
- TDD-kehitystekniikan koejakso uusien AWS Lambda -funktioiden osalta, jonka jälkeen muutoksen vaikutuksia arvioidaan.

## 4.2 Tulevaisuuspainotteiset muutosehdotukset

Työn jo aiemmissa osissa läpikäydyistä syistä johtuen kaikkia ehdotettavia laadunvarmennuksen sekä testauksen uudistuksia ei suositella implementoitavaksi saman tien. Tarkoituksena on jättää toimenpiteet, jotka ovat hankalampia ottaa käyttöön tai muutoin sopivat paremmin tulevaisuuden teknologisten tai arkkitehtuuristen muutosten yhteydessä implementoitaviksi odottamaan sopivaa ajankohtaa.

Kehityskohteet:

- Front-end ohjelmakoodin yksikkötestaus
- jatkuva integraatio AWS Codepipelinella
- muina käytänteinä koodin tarkistuksen koejakso sekä TDD-kehitystekniikan jatkotarkastelu.




Kuva 16. Kehityskaari tulevaisuudessa kaikkine muutoksineen.

Kuvassa 16 nähdään lopullisen kehityskaaren hahmotelma, kun tässä luvussa alla käsiteltävät ehdotukset implementoidaan.

### 4.2.1 Testaus

Testauksen saralla tulevaisuudessa kiinnitetään huomio käyttöliittymään. Dynamon suunnitelmissa on ollut pitkään uudistaa ja päivittää front endissa käytetyt kehikot ja kirjastot. Nykyinen mallipohjien rakentamiseen keskittyvä käyttöliittymä on luotu pitkälti hyödyntäen aiemmin erittäin suosittuja kirjastoja kuten jQuery, Backbone.js sekä lodash. Näistä Backbone.js-kehikko on erityisen tarkastelun alla johtuen nykyaikaisten ohjelmointikehikkojen

merkittävästä kehityksestä. Kehitys on johtanut pohdintaan siitä, tulisiko Dynamossa tehdä vaihdos johonkin modernimpaan vaihtoehtoon. Tällaisia potentiaalisia vaihtoehtoja ovat muun muassa React tai Vue.js.

Käyttöliittymän testauksen kannalta, etenkin yksikkötestien näkökulmasta, Backbonesta siirtyminen johonkin toiseen ohjelmointikehikkoon tarjoaa samantyylisen tilaisuuden päivittää testauskäytäntöjä käyttöliittymän osalta kuin mitä AWS Lambda -funktiot tarjosivat Dynamon taustajärjestelmän puolella. Nykyistä käyttöliittymää sen tämänhetkisillä kirjastoilla rasittaa suuri testaamattoman koodin määrä, eikä näin ollen yksikkötestien tekemistä vanhalle pohjalle voida pitää ajankäytön näkökulmasta järkevänä. Nykytilanteessa tulevaisuuden uusille komponenteille ja moduuleille voisi kyllä luoda yksikkötestejä, mutta tällöin yhdenmukaisuus kärsisi. Selkeämmän rajauksen vuoksi yksikkötestausta ehdotetaan siis suuremman muutoksen yhteydessä implementoitatavaksi. Muuta käyttöliittymän testausta, kuten end-to-end- tai integraatiotestausta, ei suositella niiden yksikkötestausta korkeampien ajallisten sekä rahallisten vaatimusten takia.

Yksikkötestaus tulee implementoida käyttäen toteutusvaiheessa valittavaa testauskirjastoa. Tämän hetkisiä suosittuja front end -testikirjastoja ovat esimerkiksi Jest, Mocha, Enzyme ja Jasmine. Kuten työssä on aiemmin käyty läpi, yksikkötestien idea on testata aina jotakin tiettyä yksikköä - funktiota tai komponenttia. Testikirjastoja käyttäen yksikköä testataan tarkastamalla ja vertailemalla sen ulospäin antamia arvoja. Yksikkötestit on suositeltavaa sisällyttää osaksi "testi suiteja" eli testikokonaisuuksia, jotka testaavat kerralla jonkin tietyn samaan kokonaisuuteen liittyvän yksikköryhmän. Yksikkötestikoodi on suositeltavaa sijoittaa samaan hakemistoon testattavan yksikön kanssa yhtenäisyyden säilyttämiseksi.

Mikäli välittömästi toteutetun suunnitelman mukaan tehty TDD-koejakso osoittaa teorian mainitsemat edut myös reaalimaailmassa toteutuviksi, on käyttöliittymän yksikkötestit suositeltavaa rakentaa yhteistoiminnassa TDD-kehitystekniikan kanssa.

Tulevaisuudessa implementoitava testausehdotus:

- Käyttöliittymäkomponenttien yksikkötestauksen implementointi pääohjelmointikehikkojen uudistuksen yhteydessä.

### 4.2.2 Jatkuva integraatio

Nykyaikaisessa ohjelmistokehityksessä jatkuva integraatio ja jatkuva toimitus voidaan nähdä ikään kuin laadunvarmistuksen lukkona, joka yhdistää ja integroi suuren osan laadunvarmistuksen kokonaisuuteen liittyviä tekijöitä yhteen, mikä vähentää manuaalista työtä entisestään sekä tarjoten koottuja tuloksia muiden laadunvarmennustoimenpiteiden lopputulemasta. Dynamon kohdalla jatkuva integraatio kohdistuu tässä kontekstissa pitkälti taustajärjestelmään, koska käyttöliittymän testaukseen paneudutaan mahdollisesti vasta suuremman teknologisen muutoksen myötä. Tästä johtuen taustajärjestelmä ja käyttöliittymä olisi järkevää käsitellä erillisinä julkaisuputkinaan, joilla on omat automaatioon liittyvät yksityiskohtansa sekä omat versionhallinnan säilytyspaikansa. Käyttöliittymän osalta on mahdollista hyödyntää jatkuvaa toimitusta niin, että versionhallinnan kehityshaaran päivitys johtaisi esimerkiksi S3-testiympäristöön tehtävään julkaisuprosessiin - tässä vaiheessa ilman testausvaihetta. Lisäksi on muistettava, että AWS Codepipeline -putket voivat kutsua ja hyödyntää toisia Codepipeline-putkia - sekä niissä syntyneitä artifakteja eli build-tiedostoja. Tämä mahdollistaa eri putkien yhteistoiminnan, mikäli siihen suuntaan kehitystä halutaan viedä.

Jatkuvan integraation lisäämistä osaksi Dynamon kehityskaarta ehdotetaan toteutettavaksi tulevaisuudessa AWS Codepipelinea käyttäen. Ehdotuksen osalta valinta Jenkinsin ja AWS Codepipelinen välillä tehtiin perustuen työkalujen samankaltaisuuteen, niiden mukauttamismahdollisuuksiin sekä potentiaalisiin synergioihin muiden toteutettavien ehdotusten kanssa. Kumpikin vaihtoehto pystyy toteuttamaan tarvittavan automaation, mutta AWS

Codepipelinen etuna katsotaan olevan etenkin sen lähempi yhteys Dynamon muuhun, tulevaan ja olemassa olevaan, AWS-arkkitehtuuriin.

Kehitysehdotuksen mukainen AWS Codepipelinen implementointi nivoutuu seuraaviin yksityiskohtiin:

- AWS Codepipelinen määrittely AWS-ympäristössä
- AWS CodeBuild -määrittely buildspec-tiedoston kautta
- AWS CodeDeploy -määrittely appspec-tiedoston avulla
- CI-putkessa käytettävien parametrien sekä salasanojen tallennus.

AWS Codepipelinen kanssa liikeelle lähdetään määrittelemällä AWS-ympäristöön, joko työn aiemman analyysin mukaisesti AWS CloudFormation -mallipohjan avulla tai suoraan AWS:n käyttöliittymästä. Näistä voidaan toteutusvaiheessa valita se, kumpi sopii paremmin vallitsevaan tilanteeseen. Mallipohjan avulla tehty määrittely mahdollistaa tämän AWS-arkkitehtuuriosan siirtämisen ja uudelleen luomisen AWS-ympäristöön vaivattomasti, mutta toteutusvaiheessa se on merkittävästi työläämpi kuin käyttöliittymätoteutus.

Codepipeline on tehokkaasti moduloitu kokonaisuus, joka koostuu useasta eri yksiköstä. Yksiköt on jaeteltu niin, että jokainen merkittävä CI-prosessin osa, kuten versionhallinta, build, test ja deploy ovat erikseen määriteltävissä osana AWS Codepipelinea. Jokaisen yksikön toteuttamiseen on lähtökohtaisesti mahdollista käyttää monipuolisesti myös eri palveluntarjoajien ratkaisuja integroimalla ne osaksi AWS Codepipelinea erityisillä liitännäisillä tai joissain tapauksissa suoraan AWS:n tarjoamilla natiiveilla tukimuodoilla.

Versionhallinta toteutetaan käyttämällä Githubia. Näin saadaan natiivisti tuettu versionhallintapalvelu hyödynnettyä täysimääräisesti. Natiivin implementaation etu versionhallinnan kohdalla on Git committien metatietoihin käsiksi pääsy. Metatiedoilla tarkoitetaan committeihin liittyviä tietoja commitista, kuten kuka commitin on tehnyt, milloin sekä millaisella viestillä varustettuna.

Build-vaiheen toteukseen on tarjoilla AWS:n oma AWS CodeBuild -palvelu, joka on pääsääntöisesti tämän vaiheen keskeinen rakennuspalikka. Myös muita työkaluja, kuten Jenkinsiä, voi käyttää build-vaiheen implementointiin. On kuitenkin perusteltua pysyä käsillä olevassa implementaatiossa AWS-ekosysteemin sisällä. AWS CodeBuild tarjoaa CI-putken build-ympäristön. Käyttäjä voi määritellä itse missä käyttöjärjestelmäympäristössä build-vaihe tapahtuu sekä mitkä ovat tarpeelliset ajoympäristöt ohjelmakoodin kannalta. Kaikki AWS CodeBuildin määrittelyt tapahtuvat buildspec.yml-tiedoston avulla.

Buildspec-tiedoston tulee sijaita koonnin kohteena olevan projektin juuritasolla kuvan 17 mukaisesti.

```
(root directory name)
    |-- pom.xml
    |-- buildspec.yml
    `-- src
         |-- main
         |     `-- java
         |           `-- MessageUtil.java
         `-- test
               `-- java
                     `-- TestMessageUtil.java
```

Kuva 17. Havainnollistava kuva buildspec-tiedoston sijainnista projektin juuritasolla [38].

Build-vaihe tuottaa pääsääntöisesti niin sanottuja artifakteja, jotka tulee säilöä esimerkiksi S3-bucketiin. Artifaktit ovat tiedostoja, joita lähdekoodista on rakennusvaiheessa saatu aikaan. Ne ovat yleisiä ohjelmointiympäristöissä, eivätkä sidoksissa AWS:n palveluihin. Jatkuvan integraation kannalta artifakteja käytetään pääsääntöisesti myöhemmin julkaisuvaiheessa, jossa sovellus laitetaan toimintaan testi- tai tuotantoympäristöön. Artifaktit tarjoavat myös versionhallintaa tarkempaa sovellusversiohallintaa sekä mahdollistavat useissa ohjelmointiympäristöissä kätevän tavan välittää toiminnallisuuksia eri projektien välillä. AWS CodeBuildissa artifakteihin liittyvät säädöt tehdään *buildspec*-tiedostossa.

```yaml
version: 0.2

phases:
  install:
    runtime-versions:
      java: corretto8
  build:
    commands:
      - echo Build started on `date`
      - mvn test
  post_build:
    commands:
      - echo Build completed on `date`
      - mvn package
artifacts:
  files:
    - target/my-app-1.0-SNAPSHOT.jar
    - appspec.yml
  discard-paths: yes
```

Kuva 18. Havainnollistava kuva buildspec-tiedoston sisällöstä [38].

Dynamon tapauksessa taustajärjestelmän integraatiotestauksen automatisointiin on pääsääntöisesti kaksi eri lähestymistapaa. Ensimmäinen vaihtoehto on build-vaiheessa rakentaa ohjelmakoodista paikallinen versio taustajärjestelmästä ja ajaa testausta varten Salesforcessa olevan testikäyttäjän testimallipohja esimerkiksi curl-komennolla tehtävällä HTTP-kutsulla. Testin jälkeen validoidaan integraatiotestin palauttama HTML-merkkijono. Toinen vaihtoehto integraatiotestaukseen on sen sijoittaminen joko täysin manuaaliseksi testaukseksi tai sitten omaksi pipelineksi deploy-vaiheen jälkeen. Tällainen deploy-vaiheen jälkeinen testaus olisi siis oma vaiheensa, kun ohjelmakoodi on osana ensimmäistä CI-putkea asennettu toimimaan testipalvelinympäristöön. Deployn jälkeen ajetaan integraatiotestit testipalvelimella pyörivää sovellusta ja sen rajapintoja vasten. Johtuen testimallipohjan komentojen laajuudesta ei ole täyttä varmuutta, miten kaikki taustajärjestelmän testattavat kommennot toimivat paikallisessa asennuksessa. Tuotanto- ja testiympäristöt toimivat tavallisesti omilla ympäristömuuttujillaan Dynamon nykyisessä mekanismissa. Näiden Tomcatin muuttujien vaikutukset selviävät toteutusvaiheessa, ja silloin myös selviää, mikä CI-putki integraatiotestaukseen soveltuu. Lisäksi tulee huomioida se todennäköinen

skenaario, että taustajärjestelmän komentoja, jotka käyttävät AWS Lambda -toteutusta, vaativat testausmielessä sitä, että uudet AWS Lambda -funktiot on asennettu AWS-ympäristöön ennen integraatiotestin ajoa. Mikäli näin ei kuitenkaan toimita, niin vaatimuksena on välittää tieto testattaville komennoille siitä, että tällaisen CI-putken osana olevan integraatiotestin Lambda-funktioita tulisi kutsua vasten paikallista Lambda-rajapintaa, eikä kuten normaalia, AWS-ympäristössä toimivaa Lambda-funktiota tai niiden rajapintoja.

Uusien AWS Lambda -funktioiden avulla luotujen taustajärjestelmäkomentojen yksikkötestaus osana CI-putkea toimii hieman integraatiotestausta poikkeavalla tavalla. Yksikkötestaus tapahtuu pääsääntöisesti osana CI-putkea, jossa osana build-vaihetta paikallinen Lambda-rajapinta pystytetään. Tätä Lambda-rajapintaa vasten ajetaan valitulla testikirjastolla luodut yksikkötestit. Dynamon uudet Lambda-funktiot kirjoitetaan lähtökohtaisesti NodeJS:llä. Tästä johtuen buildspec.yml-tiedostossa tuleekin oletusarvoisesti määritellä sekä Java- että NodeJS-ajoympäristöt.

Jotta CI-putki toimisi moitteettomasti, on Codepipelinen päästävä käsiksi määriteltyihin parametreihin, joilla kontrolloidaan esimerkiksi versionhallinnan sijaintia tai seurattavan haaran nimeä. Myös erilaiset tokenit, kuten testeissä käytettävän testikäyttäjän refresh token, on syytä säilyttää keskitetysti hyvin suojattuna. Ratkaisuna näihin ongelmiin voidaan käyttää AWS:n tarjoamia palveluita AWS Parameter Store sekä AWS Secret Manager. Nimiensä mukaisesti Parameter Store keskittyy erilaisten parametrien suojattuun tallennukseen, Secret Manager puolestaan eri tietokantatunnuksien, API-avainten ynnä muiden hallintaan ja hakemiseen.

Julkaisuvaiheen toiminnallisuus onnistuu myös AWS-työkaluilla. AWS CodeDeploy vastaa testausvaiheen jälkeisestä vaiheesta, jossa onnistuneesti testattu koodi julkaistaan esimerkiksi Staging (testi) -ympäristöön. Julkaisuvaihe on mahdollista asettaa viemään julkaisu myös tuotantoon asti, mikäli näin halutaan tavoitella täydellistä automaation tasoa prosessin osalta. AWS CodeDeploy -julkaisuputket rakennetaan pääsääntöisesti hyödyntäen build-

vaiheessa luotuja artifakteja, jotka löytyvät ehdotuksen mukaisesti aiemmin mainitusta S3-bucketista. AWS CodeDeploy -määritykset tehdään appspec-tiedostoon [39].

```
version: 0.0
os: linux
files:
  - source: /
    destination: /var/www/html/WordPress
hooks:
  BeforeInstall:
    - location: scripts/install_dependencies.sh
      timeout: 300
      runas: root
  AfterInstall:
    - location: scripts/change_permissions.sh
      timeout: 300
      runas: root
  ApplicationStart:
    - location: scripts/start_server.sh
    - location: scripts/create_test_db.sh
      timeout: 300
      runas: root
  ApplicationStop:
    - location: scripts/stop_server.sh
      timeout: 300
      runas: root
```

Kuva 19. Havainnollistava kuva AWS CodeDeploy -määrittelyistä appspec-tiedostossa [39].

Appspec-tiedoston tallennus sijainti on buildspec-tiedoston tapaan projektin juurihakemistossa.

Jatkuvan integraation toimeenpanovaiheessa on tärkeää kiinnittää erityistä huomiota siihen, että AWS Codepipeline sekä sen yhteydessä hyödynnetyt

palvelut saavat riittävät oikeudet toimiakseen tehokkaasti. Oikeuksia kontrolloidaan AWS:n Identity & Access Management (IAM) -palvelun kautta.

### 4.2.3 Muut käytännöt

Koodin tarkistus oli työssä määritelty osana muita käytänteitä. Aiemman analyysin pohjalta koodin tarkistaminen yhteistoiminnassa jonkun toisen, yleensä vanhemman kehittäjän, kanssa saa aikaan konkreettisen laatua varmistavan vaikutuksen. Käytänne parantaa tiedonsiirtoa sekä tarjoaa tilaisuuden vaihtaa ajatuksia konkreettisesti liittyen kehitettävien toiminnallisuuksien yksityiskohtiin ohjelmakooditasolla.

Ohjelmakoodin tarkistuksesta on selkeitä potentiaalisia hyötyjä, mutta aivan ilmaiseksi käytänteen implementointi olemassa olevaan kehityskaareen ei tule. Haasteita saattavat tuovat etenkin löytää tarvittavat resurssit toteuttamaan tarkistuksia sekä käytänteen yhteensovittaminen olemassa olevaan yrityskulttuuriin. Dynamon kehitysryhmän pienestä koosta ja enemmän tai vähemmän tarkoin rajatuista vastuualueista johtuen koodin tarkistusta ei voida ehdottaa otettavaksi käyttöön välittömästi. Tosiasiallinen käyttöönotto on järkevää tehdä joko tilanteessa, jolloin kehitysryhmässä on käytössä enemmän resursseja, tai tuoda käytänne hitaasti mukaan osaksi kehityskaarta, niin sanotusti rauhallisella uuttamismenetelmällä. Ehdotus onkin aloittaa koodin tarkistuksen sisäänajo välittömien toimenpiteiden jälkeen lähtien liikkeelle rajatusta koejaksosta, jota edellyttää perinpohjin mietitty suunnitelma siitä, mitkä ovat ne prosessuaaliset elementit, joita tällainen kehityskaaren muutos vaatii kehitysryhmätasolla. On oltava selkeä näkemys siitä, kuka on vastuussa tarkistuksista ja missä kohtaa tarkistus tehdään - koejakson aikana tarkistuksen voisi suorittaa kuka tahansa toinen Dynamon kehittäjä, toisaalta tarkistuksen ajoitus puolestaan on yleensä sijoitettu kohtaan, jossa versionhallinnan toiminnallisuushaara liitetään osaksi jotain päähaaraa. Koejakson aikana ajetaan koodin tarkistusta sisään myös teknologiselta kannalta, eli toimitaan uusia tekniikoita käyttäen esimerkiksi Githubin pull requestien käyttö, lisäksi

myös analysoidaan käytänteen yleistä vaikutusta kehitysprosessiin peilaten sitä aikaisempaan lähtötilanteeseen.

TDD-kehitystekniikan koejakson pohjalta tulee puolestaan tehdä päätös, otetaanko tekniikka käyttöön vai ei.

Ehdotus:

- koodin tarkistuksen koejakso
- TDD-kehitystekniikan arviointi ja potentiaalinen implementointi koko projektin laajuisesti.

# 5 Loppusanat

Opinnäytetyö saavutti asetetut tavoitteet varsin hyvin. Työn esittämillä toimenpiteillä testaus pystytään implementoimaan tehokkaasti niin ajankäytön kuin kattavuudenkin kannalta. Työssä tulkitaan testauskäytäntöjä laveasti sisällyttäen muitakin laadunvarmennustoimenpiteitä osaksi esityksiä. On tärkeää huomata, että alkuperäisestä kehityskaaresta ei työn saatossa pääsääntöisesti poisteta mitään laadunvarmennukseen liittyvää vaihetta, vaan ainoastaan tuodaan uusia elementtejä. Ehdotetut toimenpiteet on dokumentoitu kattavasti AWS-ympäristön osalta. Näin ollen kaikkien esitettyjen toimenpiteiden myöhempi toimeenpano ja toteutus tulee pääsääntöisesti olemaan mahdollista sekä myös varsin suoraviivaista.

Työssä käsiteltiin aiheena ollutta aihepiiriä varsin laajasta näkökulmasta, joka asetti tiettyjä rajoitteita sen suhteen, kuinka syvällisesti eri asioita oli mahdollista tutkia ja analysoida. Dynamon resurssit sekä ohjelmakoodin ominaispiirteet toimivat merkittävimpinä eri työkalu- sekä palveluvalintojen taustavaikuttimina.

Avonaisia kysymyksiä jäi suunnitelman pohjalta auki liittyen toimeenpanovaiheen yksityiskohtiin. Näihin kysymyksiin saadaan vastaus toimeenpanon yhteydessä tehtävien päätösten puitteissa. Eräitä mielenkiintoisia toimeenpanovaiheen ratkaisukohtia ovat etenkin kirjastot, joilla testausta lähdetään rakentamaan sekä muiden käytänteiden implementointien laajuus.

Mikäli suunnitelman yksityiskohtia myöhempien päätösten johdosta lähdetään merkittävästi muuttamaan, saattaa se vaikuttaa merkittävästi siihen, kuinka tarkasti työssä selostettuja askelmerkkejä voidaan todellisuudessa seurata. Opinnäytetyön suunnitelma on siten osiensa puolesta merkittävästi yhteenkietoutunut, joka on syytä todeta ja ymmärtää riskinä toimeenpanoon liittyviä päätöksiä myöhemmin tehtäessä.

Opinnäytetyön tarjoamat hyödyt eivät rajoittuneet ainoastaan kirjoitettuun tilaajalle tarkoitettuun testaus- ja laadunvarmennussuunnitelmaan. Myös työn tehneen omakohtainen ymmärrys aihepiiristä kasvoi huomattavasti. Pilvipalveluiden, etenkin AWS:n, tarjoamien palveluiden määrä jatkaa kasvuaan ja onnistuu usein yllättämään. Näiden palvelujen hyötynä on usein niiden suhteellinen helppokäyttöisyys. Toisaalta palvelukokonaisuuksien hinnoittelu ei ole aina erityisen selkeää niiden koostuessa pienistä palasista. Katsotaan, mitä tulevaisuus tuo tullessaan.

# Lähteet

1 Franks, Kenneth. 2021. What Is Document Automation? Verkkoaineisto. https://www.jotform.com/what-is-document-automation/. Luettu 03.08.2021.

2 Salmela, Kimmo. 2019. The Essential Guide to Document Generation in Salesforce. Verkkoaineisto. https://www.documill.com/blog/the-essential-guide-to-document-generation-in-salesforce. Luettu 10.05.2021.

3 The Tech Platform. 2020. What is Sprint (software development). Verkkoaineisto. https://www.thetechplatform.com/post/what-is-sprint-software-development/. Luettu 10.05.2021.

4 Chen, Vivienne. 2020. What is Dogfooding? And Why It's Still Important to Tech. Verkkoaineisto. https://bubble.io/blog/dogfooding-startup-tech/. Luettu 11.05.2021.

5 Monolithic vs. Microservices: a guide to application architecture. 2020. Verkkoaineisto. https://www.talend.com/resources/monolithic-architecture/. Luettu 12.05.2021.

6 Ohjelmistotestaus ja laadunvarmistus. Verkkoaineisto. https://www.valagroup.com/fi/palvelut/ohjelmistotestaus-ja-laadunvarmistus/. Luettu 13.05.2021.

7 What is Version Control? Verkkoaineisto. https://www.atlassian.com/git/tutorials/what-is-version-control/. Luettu 14.05.2021.

8 Why Git for your Organization. Verkkoaineisto. https://www.atlassian.com/git/tutorials/why-git/. Luettu 15.05.2021.

9 Maxim, B.R. 2017. Software Quality Management. Michiganin yliopisto – Dearborn. http://groups.umd.umich.edu/cis/course.des/cis376/ppt/lec14.ppt. Haettu 22.05.2021.

10 Radigan, Dan. Why code reviews matter (and actually save time). Verkkoaineisto. https://www.atlassian.com/agile/software-development/code-reviews. Luettu 22.05.2021.

11 Hamilton, Thomas. 2021. What is Software Testing? Definition, Basics & Types in Software Engineering. Verkkoaineisto. https://www.guru99.com/software-testing-introduction-importance.html . Luettu 25.05.2021.

12 Hamilton, Thomas. 2021. Unit Testing Tutorial: What is, Types, Tools & Test Example. Verkkoaineisto. https://www.guru99.com/unit-testing-guide.html. Luettu 25.05.2021.

13 Hamilton, Thomas. 2021. Integration Testing: What is, Types, Top Down & Bottom Up Example. Verkkoaineisto. https://www.guru99.com/integration-testing.html. Luettu 26.05.2021.

14 Hamilton, Thomas. 2021. What is System Testing? Types & Definition with Example. Verkkoaineisto. https://www.guru99.com/system-testing.html. Luettu 27.05.2021.

15 Hamilton, Thomas. 2021. What is Functional Testing? Types & Examples. Verkkoaineisto. https://www.guru99.com/functional-testing.html. Luettu 27.05.2021.

16 Hamilton, Thomas. What is Non Functional Testing? Types with Example. 2021. Verkkoaineisto. https://www.guru99.com/non-functional-testing.html. Luettu 28.05.2021.

17 Hamilton, Thomas. 2021. What is Usability Testing? UX Testing Example. Verkkoaineisto. https://www.guru99.com/usability-testing-tutorial.html. Luettu 28.05.2021.

18 Hamilton, Thomas. What is Regression Testing? 2021. Verkkoaineisto. https://www.guru99.com/regression-testing.html. Luettu 28.05.2021

19 What Is Acceptance Testing? 2021. Verkkoaineisto. https://www.software-testinghelp.com/what-is-acceptance-testing/. Luettu 29.05.2021.

20 What is CI/CD? 2018. Verkkoaineisto. https://www.redhat.com/en/topics/devops/what-is-ci-cd. Luettu 29.05.2021.

21 Ketteryys haltuun: Yleisimmät ketterät käytännöt. 2013. Verkkoaineisto. https://meteoriitti.com/2013/06/06/ketteryys-haltuun-yleisimmat-ketterat-kaytannot/. Luettu 29.05.2021.

22 Pipeline Syntax. Verkkoaineisto. https://www.jenkins.io/doc/book/pipeline/syntax/. Luettu 29.05.2021.

23 Jenkins - AWS Codepipeline plugin. 2021. Verkkoaineisto. https://plugins.jenkins.io/aws-codepipeline/. Luettu 29.05.2021.

24 Continuous delivery and continuous integration. Verkkoaineisto. https://docs.aws.amazon.com/codepipeline/latest/userguide/concepts-continuous-delivery-integration.html. Luettu 29.05.2021.

25 Vasiljevic, Rastko. 2020. Infrastructure As Code Demystified: IaC Benefits, Challenges & Best Practices. Verkkoaineisto. https://superadmins.com/infrastructure-as-code-demystified-iac-benefits-challenges-best-practices/. Luettu 29.05.2021.

26 Integrations with CodePipeline action types. Verkkoaineisto. https://docs.aws.amazon.com/codepipeline/latest/userguide/integrations-action-type.html. Luettu 29.05.2021.

27 Guay, Matthew. 2020. What are webhooks? Verkkoaineisto. https://zapier.com/blog/what-are-webhooks/. Luettu 29.05.2021.

28 Schultz, Tom. 2017. Using Custom Source Actions in AWS CodePipeline for Increased Visibility for Third-Party Source Control. Verkkoaineisto. https://aws.amazon.com/blogs/devops/using-custom-source-actions-in-aws-codepipeline-for-increased-visibility-for-third-party-source-control/. Luettu 29.05.2021.

29 AWS CodePipelin pricing. Verkkoaineisto. https://aws.amazon.com/codepipeline/pricing/?nc=sn&loc=3. Luettu 29.05.2021.

30 Hamilton, Thomas. 2021. What is Test Driven Development (TDD)? Verkkoaineisto. https://www.guru99.com/test-driven-development.html . Luettu 29.05.2021.

31 Elliot, Eric. 2019. Behavior Driven Development and Functional Testing. Verkkoaineisto. https://medium.com/javascript-scene/behavior-driven-development-bdd-and-functional-testing-62084ad7f1f2 . Luettu 29.05.2021.

32 How ATDD Guides You in Collaboratively Creating User-Centric Applications. Verkkoaineisto. https://www.digite.com/agile/acceptance-test-driven-development-atdd/ . Luettu 29.05.2021.

33 AWS Key Management Service. Verkkoaineisto. https://aws.amazon.com/kms/. Luettu 01.06.2021.

34 What is the AWS Serverless Application Model (AWS SAM)? Verkkoaineisto. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html. Luettu 01.06.2021.

35 AWS Sam reference. Verkkoaineisto. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-reference.html. Luettu 01.06.2021.

36 About pull requests. Verkkoaineisto. https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests. Luettu 01.06.2021.

37 Git Branching – Rebasing. Verkkoaineisto. https://git-scm.com/book/en/v2/Git-Branching-Rebasing. Luettu 02.06.2021.

38 Step 2: Create the buildspec file. Verkkoaineisto. https://docs.aws.amazon.com/codebuild/latest/userguide/getting-started-create-build-spec-console.html. Luettu 03.06.2021.

39 CodeDeploy AppSpec File reference. Verkkoaineisto. https://docs.aws.amazon.com/codedeploy/latest/userguide/reference-appspec-file.html. Luettu 03.06.2021.