PILVIPAVELUJEN HYÖDYNTÄMINEN OHJELMISTOTESTAUKSESSA

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäki, Tietotekniikan Koulutusohjelma

kevät, 2017

Aleksi Muotka

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tietotekniikan koulutusohjelma
Riihimäki

Tekijä Aleksi Muotka **Vuosi** 2017

Työn nimi Pilvipalvelujen hyödyntäminen ohjelmistotestauksessa

Työn ohjaaja Jari Mustajärvi

TIIVISTELMÄ

Opinnäytetyö tehtiin Koneelle, joka on suomalainen hissejä, liukuportaita ja nosto-ovia valmistava ja huoltava yritys. Tämän opinnäytetyön tarkoituksena oli tutkia pilvipalvelujen hyödyntämismahdollisuuksia Koneen ohjelmistokehityksessä. Tutkiminen rajattiin ohjelmistokehityksen automaatiotestauksen hidasteeksi muodostuneeseen yksikkötestaukseen ja hyväksymistestaukseen. Lähtökohtana oli selvittää voidaanko yksikkötestausta ja hyväksymistestausta siirtää osittain pilveen ja näin ollen vähentää Koneen testaukseen käytettävien palvelimien määrää ja niiden huoltoa.

Työn teoriaosassa on käyty läpi ohjelmistokehitystä ja sen eri muotoja. Teoriassa on myös käyty läpi ohjelmistotestausta ja automatisoitua ohjelmistotestausta ja siihen tarkoitukseen olevia työkaluja. Lisäksi on käyty läpi pilvipalvelumalleja ja syvällisemmin vielä Amazonin tarjoamaa Amazon Web Service -palvelua, jota tässä opinnäytetyössä käytettiin pilvialustana. Teorian lopussa käytiin vielä läpi Kubernetes-ohjelmistoa, jota myös käytettiin työssä klusterien hallintaan.

Opinnäytetyön tuloksena selvisi, että pilvipohjainen ratkaisu on toimiva ratkaisu hidasteeksi muodostuneeseen yksikkötestaukseen ja hyväksymistestaukseen. Kubernetes osoittautui potentiaaliseksi vaihtoehdoksi pilvessä käytettävään klusterin hallintaan, mutta Kubernetes tarvitsee vielä lisätutkimusta, koska osa opinnäytetyössä käytettävistä ratkaisuista oli beta-vaiheessa tai kokeiluvaiheessa. Vaikka nykyiset käytännöt tukevat pilveen siirtymistä on vielä tehtävä lisää tutkimuksia, ennen kuin siirtyminen pilvipohjaiseen ratkaisuun voi alkaa.

Avainsanat Kubernetes, Jenkins, Amazon Web Services

Sivut 34 sivua, joista liitteitä 0 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Degree Programme in Information Technology
Riihimäki

Author Aleksi Muotka **Year** 2017

Subject Utilization of cloud computing in software development

Supervisor Jari Mustajärvi

ABSTRACT

This thesis was made for Kone, which is a Finnish company that manufactures elevator and escalators and provides maintenance services for them. The purpose of this thesis was to examine methods to utilize cloud computing in Kone software development. The thesis topic was limited to unit testing and acceptance testing which has become a roadblock for automation testing. The idea was to determine if it were possible to move testing partially into cloud so that there would be lower number in-house servers to use and maintain.

The theory section of the thesis goes through different forms of software development and software testing as well as testing automation and tools for testing automation. The theory part also examines cloud computing service models and Amazon Web Services which are used in this thesis. In the end of the theory part there is a chapter on the Kubernetes software which was used to manage clusters in this thesis.

Result of this project, some findings were made: A cloud based solution is an effective solution for unit testing and acceptance testing. Also Kubernetes is a potential option for cluster management in a cloud, but it needs to be closer examined because some of the solutions used in this thesis were at a beta release or an experimental release state. Although current solutions support moving unit testing and acceptance testing to cloud, still closer examination needs to be made before moving to a cloud based solution.

Keywords Kubernetes, Jenkins, Amazon Web Services

Pages 34 pages including appendices 0 pages

SISÄLLYS

1 JOHDANTO 1

2 TEORIA 1

2.1 Ohjelmistokehitys 2
2.1.1 Vesiputousmalli 3
2.1.2 Scrum 4
2.1.3 Testivetoinen kehitys 5
2.1.4 DevOps 6
2.2 Ohjelmistotestaus 7
2.2.1 Automatisoitu ohjelmistotestaus 8
2.3 Automatisoidun testauksen työkaluja 9
2.3.1 Robot Framework 9
2.3.3 Docker 10
2.4 Pilvilaskenta 11
2.5 Pilvipalvelumallit 11
2.5.1 Software as a Service (SaaS) 12
2.5.2 Platform as a Service, (PaaS) 12
2.5.3 Infrastructure as a Service (IaaS) 12
2.6 Amazon Web Services (AWS) 12
2.6.1 AWS EC2 13
2.6.2 AWS EBS Volumes 13
2.6.3 Amazon Virtual Private Cloud (VPC) 13
2.7 Kubernetes työkalu 14
2.7.1 Kubernetes podit 15
2.7.2 Kubernetes Volumes 16
2.7.3 Kubernetes PersistentVolumes 16
2.7.4 Kubernetes Deployment 18
2.7.5 Kubernetes Node 18
2.7.6 Kubernetes Replication Controller 18
2.8 Terraform 19

3 PILVITESTAUS AWS 20

3.1 Koneen ohjelmistokehityksen testaus 20
3.2 Jenkins-setapin suunnitelma 21
3.3 AWS testausvaiheet 22
3.3.1 Tarvittavat ohjelmistot 22
3.3.2 AWS CLI konfigurointi 23
3.3.3 Verkkotunnuksen rekisteröinti 23
3.3.4 S3 bucket luonti 24
3.3.5 Klusterin luonti 24
3.3.6 Autoskaalaus 25
3.3.7 Jenkins-master volumen konfigurointi 26
3.3.8 Jenkins-master määrittely 28
3.3.9 Jenkins-slave määrittely 29

4 JOHTOPÄÄTÖKSET 31

LÄHTEET 32

1 JOHDANTO

Ohjelmistoviat ovat yleisiä ja niistä johtuen yrityksille koituu rahallisia tappiota aika ajoin. Tänä päivänä ohjelmistokehityksessä investoidaan enemmän aikaa ja resursseja ohjelmiston analysoimiseen ja testaamiseen, jotta varmistetaan ohjelmiston laatu. Ohjelmistokehittäjien ja testaajien on keskityttävä moniin asioihin, jotka nopeuttava ohjelmistokehitys-prosessia. Vikojen löytäminen mahdollisimman varhaisessa vaiheessa ohjelmistokehitystä ja niiden poistaminen heti löytämisen jälkeen on tärkeää. Tämä edesauttaa nopeaa uuden ohjelmiston tuottamista. (Li & Wu 2006.)

Tässä opinnäytetyössä keskityttiin hidasteeksi muodostuneeseen yksikkötestaukseen ja hyväksymistestaukseen. Koneella tehtävässä testauksessa kehittäjä tekee uuden osan ohjelmistoon, minkä jälkeen hän ajaa määrätyt testit ohjelmistolle. Testaukseen varattujen palvelimien määrä on rajallinen ja välillä kehittäjä joutuu odottamaan palvelimen vapautumista ennen kuin pääsee tekemään omat testinsä. Tämän lisäksi laskentakapasiteettia ei voida helposti lisätä haluttaessa ja testit joudutaan tekemään peräkkäin sarjassa, mikä vie taas enemmän aikaa.

Pilvimaailmassa laskentakapasiteettia on mahdollista lisätä nopeasti ja helposti ja myös testien rinnakkain ajo on mahdollista kapasiteetin lisääntyessä. Tämä poistaisi ongelmat palvelimen vapautumisen odottelusta ja laskentakapasiteetin lisäämisestä.

Tässä opinnäytetyössä tehtiin konseptin varmistuskokeilu, jossa pystytettiin testauksen mukainen setappi Amazonin pilveen. Tämän kokeilun tarkoitus oli varmistaa, että Amazonin pilvipalvelu on mahdollinen alusta Koneen ohjelmistotestaukseen.

2 TEORIA

Työn teoriaosassa on käyty läpi ohjelmistokehitystä ja sen eri muotoja. Teoriassa on myös käyty läpi ohjelmistotestausta ja automatisoitua ohjelmistotestausta ja siihen tarkoitukseen olevia työkaluja. Lisäksi on käyty läpi pilvipalvelumalleja ja syvällisemmin vielä Amazonin tarjoamaa Amazon Web Service -palvelua, jota tässä opinnäytetyössä käytettiin pilvialustana. Teorian lopussa käytiin vielä läpi Kubernetes-ohjelmistoa, jota myös käytettiin työssä klusterien hallintaan.

2.1 Ohjelmistokehitys

John Dooleyn mukaan ohjelmistokehitys on prosessi, jossa käyttäjältä otetaan kokoelma vaatimuksia, analysoidaan ne, suunnitellaan ratkaisu niihin ja sitten toteutetaan ratkaisu tietokoneella. Dooleyn mukaan ohjelmistokehityksen tavoite on tuottaa vikavapaata koodia käyttäjille ajallaan ja budjetin rajoissa. Kaikki tämä tapahtuu jatkuvasti vaihtuvien vaatimusten keskellä. (Dooley 2011, 1-5.)
Jokaisella ohjelmalla on elinkaari. Sillä ei ole väliä kuinka pieni tai suuri ohjelma on tai kuinka moni henkilö tekee töitä ohjelman eteen. Kaikki ohjelmat käyvät läpi samat toimenpiteet:

1. Suunnitelma
2. Vaatimusten kerääminen/tutkimus/mallinnus
3. Suunnittelu
4. Ohjelmointi ja virheiden etsiminen ja poistaminen
5. Testaus
6. Julkaisu
7. Huolto/ohjelman evoluutio
8. Eläköittäminen

Ohjelman tekijät voivat tiivistää tai lyhentää joitakin toimenpiteitä, tai yhdistää kaksi tai useampiakin toimenpiteitä yhdeksi kokonaisuudeksi, mutta kaikki ohjelmat käyvät läpi nämä toimenpiteet.
Vaikka jokaisella ohjelmalla on elinkaari, niin on monta erilaista prosessivariaatiota, jotka sisältävät nämä toimenpiteet. Jokainen elinkaarimalli on variaatio kahdesta perustavanlaatuisesta tyypistä. Ensimmäisessä tyypissä projektin tiimi käy läpi vaiheet 2-7 ennen kuin aloittaa ohjelman uuden version tekemisen. Toisessa tyypissä, joka on laajalle levinnyt nykypäivänä, projektitiimi tavallisesti tekee osittaisen elinkaaren, yleensä vaiheet 3-5 ja käy näitä vaiheita läpi useita kertoja ennen kuin etenee julkaisuvaiheeseen.

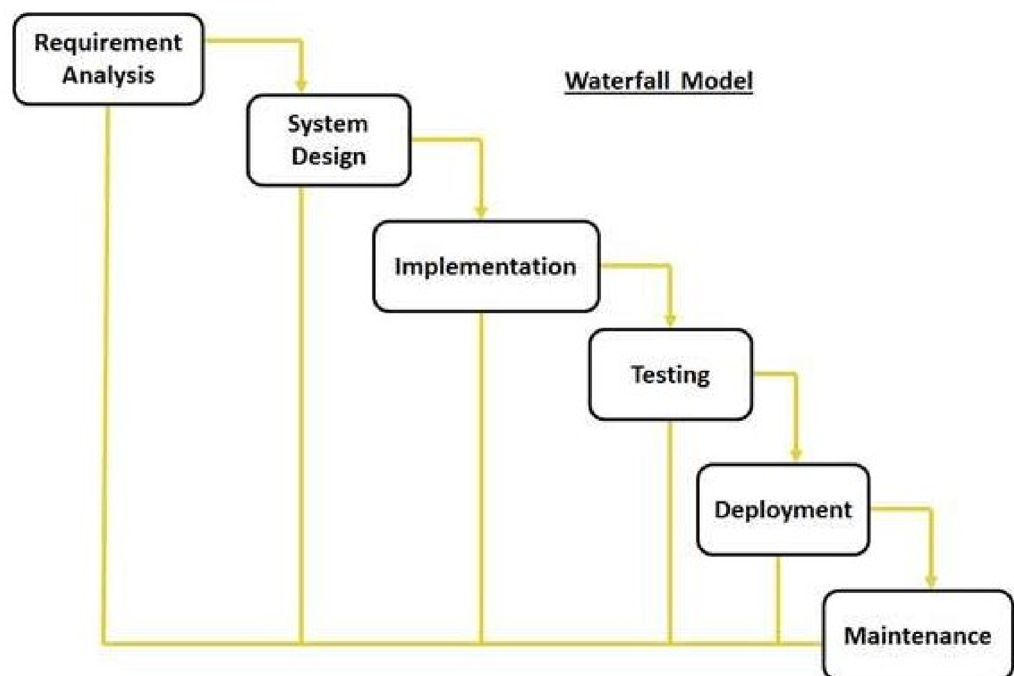
Nykyään ohjelmistokehityksen johtaminen jakaantuu tavallisesti kahteen eri tyyppiin; perinteiseen suunnitelmajohtoisiin malleihin ja uudempiin ketterän kehityksen malleihin. Suunnitelmajohtoisissa malleissa, prosessilla on taipumus seurata tiukemmin prosessin vaiheita. Näillä malleilla on selvästi määritellyt vaiheet, ja enemmän vaatimuksia vaiheen lopettamiselle ennen kuin siirrytään seuraavaan vaiheeseen. Suunnitelmajohtoiset mallit tarvitsevat enemmän dokumentaatiota jokaisessa vaiheessa ja verifikaation jokaisen tuotteen päätökselle. Näillä on tapana toimia hyvin hallituksen sopimuksissa uudelle hyvin määritellyille ohjelmistoille.
Ketterän kehityksen mallit ovat ”luonnostaan lisääntyviä”(uusia toimintoja tehdään kokoajan lisää) ja ne tekevät päätelmän, että pienet usein toistuvat julkaisut tuottavat vankkarakenteisemman tuotteen, kun taas harvemmin toistuvat julkaisut. Ketterien kehitysmallien vaiheilla on tapana sulautua yhteen enemmän kuin suunnitelmajohtoisissa malleissa. Myös

dokumentointia on vähemmän ja perusidea on, että ohjelmistoa tuotetaan ja dokumentoinnin pitäisi keskittyä siihen. (Dooley 2011, 7-8.)
Seuraavissa luvuissa käydään läpi suunnitelmajohtoista vesiputousmallia ja ketterän kehityksen scrum-mallia.

2.1.1 Vesiputousmalli

Vesiputousmallissa jokainen vaihe täytyy suorittaa loppuun ennen kuin seuraava vaihe voi alkaa. Vesiputousmallin mukaan ohjelmistokehitysprosessi on lineaarinen sekvenssivirtaus. Tämä tarkoittaa sitä, että mikä tahansa vaihe ohjelmistokehitysprosessissa alkaa vain jos edellinen vaihe on saatu valmiiksi. Vesiputousmallissa vaiheet eivät mene päällekkäin toisten vaiheiden kanssa.

Vesiputousmallin lähestymistapa kokonaisuudessaan ohjelmistokehityksessä on jaettu eri vaiheisiin. Tyypillisesti jokaisen vaiheen lopputulos on inputtina seuraavalle vaiheelle. Seuraava kuva (Kuva 1.) kertoo vesiputousmallin eri vaiheet.



Kuva 1. Vesiputousmalli (tutorialspoint n.d.)

Vesiputousmallin vaiheet ovat:

- **Vaatimusten määrittely ja analysointi:** Kaikki mahdolliset vaatimukset kehitettävästä järjestelmästä dokumentoidaan vaatimusspesifikaatioon.

- **Järjestelmän suunnittelu:** Vaatimusspesifikaatio ensimmäisestä vaiheesta tutkitaan tässä vaiheessa ja järjestelmän suunnittelua valmistellaan. Järjestelmän suunnittelu auttaa määrittämään laitteiston ja järjestelmän vaatimuksia ja auttaa myös määrittämään järjestelmän rakennetta.

- **Implementointi:** Järjestelmän suunnittelun syötteillä järjestelmää kehitetään ensiksi pienillä ohjelmilla, joita kutsutaan yksiköiksi, jotka yhdistetään seuraavassa vaiheessa. Jokaisen kehitetyn yksikön toiminnollisuus testataan. Tätä kutsutaan yksikkötestaukseksi.

- **Yhdistäminen ja testaus:** Kaikki yksiköt, jotka kehitettiin implementaatiovaiheessa, yhdistetään järjestelmään heti yksikkötestauksen jälkeen. Yhdistymisen jälkeen koko järjestelmä testataan vikojen varalta.

- **Järjestelmän käyttöönotto:** Toiminnallisen ja ei toiminnallisen testauksen jälkeen tuote otetaan käyttöön asiakkaan alustalla tai tuote julkaistaan markkinoille.

- **Ylläpito:** Ylläpito pitää sisällän vikojen korjausta, vanhan version parantamista ja uusien toiminnollisuuksien lisäämistä

Vesiputousmallissa on monia hyötyjä. Aikataulut ja määräajat voidaan asettaa jokaiselle vaiheelle ja tuote voi edetä kehitysprosessin läpi vaihe vaiheelta. Kehitys etenee konseptista suunnitteluun, implementointiin, testaukseen, asennukseen, vian hakuun ja päättyy käyttöön ja ylläpitoon.

Huono puoli vesiputousmallissa on, että se ei salli paljoakaan pohdiskelua tai tarkastusta. Siinä vaiheessa, kun applikaatio on testausvaiheessa, on todella vaikeaa mennä takaisin ja muuttaa jotain mitä ei ollut kunnolla dokumentoitu ensimmäisessä vaiheessa. (tutorialspoint n.d.)

2.1.2 Scrum

Scrum on iteratiivinen runko ohjelmistoprojektien hallintaan, ketterän kehityksen periaatteita mukaillen. Se mahdollistaa tiimejä toimittamaan oikeat toiminnot ajallaan, budjetin rajoissa ja hyvälaatuisina. Scrum auttaa ohjelmistokehitysorganisaatiota mukautumaan vaihtuviin liike-elämän vaatimuksiin, sidosryhmien tarpeisiin ja samalla suojelee tiimiä erilaisilta häiriöiltä työskentelyssä.
Scrumissa 9-5 hengen tiimi työskentelee tuottaakseen toimituskelpoista ohjelmistoa, jota asiakas voi tarkastella lyhyen ennalta määrätyn ajanjakson jälkeen, jota kutsutaan sprintiksi (kutsutaan myös iteraatioksi). Jokaisen sprintin jälkeen tiimi käy läpi suoritustaan ja miettii tapoja parantaa suoristustaan. Scrum auttaa yritystä keskittämään tiimin, korkeimmalla prioriteetilla olevaan asiakkaan vaatimukseen.

Jokapäiväiset scrum-palaverit tapahtuvat samaan aikaan samassa paikassa joka päivä. Vaatimukset ja tehtävät ovat pienillä muistilapuilla fyysisellä taululla joka sijaitsee yhteisellä alueella. Suunnittelusyklin aikana kaikki istuvat pöydän ääressä ja keskustelevat avoimesti vaatimuksista ja näyttävät suunnittelun "pokeri korttinsa" samaan aikaan. Tätä kutsutaan usein scrumissa suunnittelupokeriksi.

Scrum-roolit

Scrum määrittelee kolme roolia, jotka kuuluvat scrum-tiimiin: Scrum-mestari, tuotteen omistaja ja tiimi. Scrum-mestari on tiimin palvelija. Tämä henkilö on vastuussa, että tiimi noudattaa Srumin arvoja, käytäntöjä ja sääntöjä. Scrum-mestari pitää huolen siitä, että palaverit sujuvat hyvin ja että kaikilla tiimin jäsenillä on kaikki vaadittavat asiat onnistuneeseen työntekoon. Jos scrum-tiimille tulee eteen ongelma joka estää työnsä etenemisen, scrum-mestarin työ on keksiä nopea ratkaisu ongelmaan.
Tuotteen omistaja on henkilö, joka omistaa tuotteen kehitysjonon (lista sidosryhmien tarpeista tuotteelle), sijoittaa työn osat prioriteeteittain ja määrittelee hyväksyttävät olosuhteet. Tuotteen omistaja on vastuussa kaikesta mikä liittyy tuotteen julkaisuun. Tuotteen omistaja tapaa sidosryhmistä mieluiten kaikkia seuraavia ryhmiä kerätäkseen heidän tarpeensa tuotteen suhteen:

- **Sisäpiiriläiset—** He, joilla ohjelmistokehityksessä on suora vaikutus projektiin.
- **Yhteistyökumppanit—** Ratkaisun tarjoajat, integraattorit ja muut.
- **Loppukäyttäjät—** Sidosryhmäläiset, jotka käyttävät tuotetta sen kehitysvaiheessa.

Tiimi on itseorganisoituva, itsehallintainen ja eri osaajista koostuva ryhmä. Tiimiin voi kuulua kehittäjiä, testaajia, informaatiokehittäjiä, arkkitehtejä ja muita tarvittavia henkilöitä kehitysjonossa olevien töiden tekemiseen.

Scrum-tiimi koostuu siis tuotteen omistajasta scrum-mestarista ja tiimistä. Sen tehtävänä on ottaa työ tuotteen kehitysjonosta ja selvittää, kuinka tuottaa pieniä havainnollistavia lisäyksiä tuotteeseen joka sprintissä ja samalla hallita omaa työprosessia. (Woodward, Steffan & Ganis 2010.)

2.1.3 Testivetoinen kehitys

Testivetoinen kehitys (TDD Test Driven Development) on menetelmä, joka käyttää testejä auttamaan kehittäjiä tekemään oikeita päätöksiä oikeaan aikaan. TDD käyttää testausta ohjelmiston kehittämiseen helpolla lisäävällä tavalla. Tämä ei pelkästään paranna laatua ja designia, vaan myös yksinkertaistaa kehitysprosessia. TDD:ssa sen sijaan että, tehdään suunnitelma, joka kertoo ohjelmiston rakenteen, tehdäänkin testi, joka määrittelee, kuinka pieni osa ohjelmistostasi pitäisi toimia.

TDD saa sinut keskittymään ohjelmiston tekoon lisäävällä tavalla. Ei ole tärkeää saada ohjelmistoa toimimaan heti ensimmäisellä kerralla, koska TDD:ta käyttämällä voit koko ajan uudelleen vaikuttaa ohjelmistoon ja testeihin, kunnes se täyttää vaatimukset.
Oikeastaan TDD on yksinkertainen menetelmä joka sisältää kaksi pää käsitettä: yksikkötestaus ja uudelleen vaikuttaminen. TDD koostuu seuraavista vaiheista:
1. Kirjoita testi, joka määrittelee kuinka pieni osa ohjelmistosta pitäisi toimia.
2. Aja testi, niin helposti ja nopeasti kuin pystyt. Älä välitä ohjelmiston designista, yritä saada se vain toimimaan.
3. Siivoa ohjelmisto. Nyt, kun ohjelmisto toimii oikein, ota askel taaksepäin ja käy ohjelmisto uudelleen läpi ja poista kaikki päällekkäisyydet ja muut ongelmat, jotka tulivat esille testejä ajattaessa.

TDD on iteratiivinen prosessi ja nämä vaiheet käydään läpi monia kertoja ennen kuin olet tyytyväinen ohjelmistoon. (Hammell, Gold & Snyder 2007.)

2.1.4 DevOps

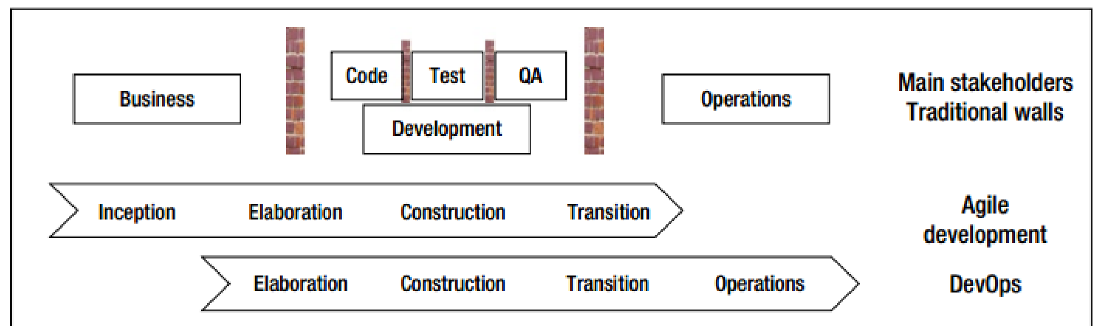
Kehittäjien tuottaessa arvokasta ohjelmistoa asiakkaille, usein kehitys (development) ja toimitus (operations) ovat törmäyskurssilla keskenään. Kehitys haluaa nähdä muutoksensa (esim. ohjelmiston uudet toiminnot) toimitettuina nopeasti asiakkaille, kun taas toimitus on kiinnostunut vakaudesta, joka tarkoittaa, että tuotteen systeemiä ei muuteta liian usein.
Kuilu kehityksen ja toimituksen välillä esiintyy eri tasoilla:

- Eri bonustavoitteista seuraa erilaiset tavoitteet kehitykselle ja toimitukselle.
- Prosessi erot johtuvat eri lähestymistavoista muutosten hallintaan, tuotantoon viemiseen ja ylläpitoon.
- Työkalu eroavaisuudet juontaa juurensa tosiasiaan, että kehitys ja toimitus usein käyttävät omia työkalujaan töidensä tekoon.

Seurauksena kehitys ja toimitus ovat usein kaksi erillistä tiimiä.
DevOps tarkoittaa välien tiivistämistä, laittamalla bonustavoitteet samoiksi ja muuttamalla prosessit ja työkalut samoiksi. DevOps tarkoittaa myös ketterien menetelmien käytön laajentamista toimitukseen kasvattaakseen yhteistyötä ja tehostaa koko ohjelmiston synnytysprosessia kokonaisvaltaisella tavalla.
DevOps käsittää monia toimia ja näkökulmia, mm seuraavia:

- Kulttuuri: Ihmiset ennen prosesseja ja työkaluja. Ohjelmiston tekevät ihmiset ihmisille.
- Automaatio: Automaatio on oleellista DevOpsille, jotta saadaan nopeasti palautetta tehdystä työstä.
- Mittaaminen: DevOps auttaa löytämään oikean tavan mittaamiseen. Laatu ja jaetut tavoitteet ovat erittäin tärkeitä.

- Jakaminen: Luo kulttuurin, jossa ihmiset jakavat ideoita, prosesseja ja työkaluja. (Httermann 2012.)



Kuva 2. Ketterä ohjelmointi kehitys laajentaa prosessia aloituksesta siirtymiseen. DevOps laajentaa kehityksestä toimitukseen ja usein tähän sisältyy osastoja kuten HR ja talousosasto. (Httermann 2012.)

2.2 Ohjelmistotestaus

Ohjelmistotestaus on rinnakkain tapahtuva insinöörityön elinkaariprosessi, jossa käytetään ja ylläpidetään testaukseen liittyviä asioita, testattavan ohjelmiston luotettavuuden mittaamisen ja parantamisen hyväksi. (Craig, Stefan & Jaskiel 2002.)

Ohjelmistotestausta on olemassa monessa muodossa. Seuraavassa listassa on mainittu yleisimpiä testausmuotoja.

- Yksikkötestaus (Unit testing)

Yksittäisen ohjelmistokomponentin tai ohjelmistomoduulin testausta. Yleensä ohjelmoija suorittaa tämän, eikä testaaja, koska tämä vaatii yksityiskohtaista tietoa sisäisestä ohjelmistosuunnittelusta ja koodista. (softwaretestinghelp n.d.)

- Hyväksymistestaus (Acceptance testing)

Yleensä tämän tyyppinen testi tehdään, koska halutaan tietää vastaako systeemi asiakkaan antamia vaatimuksia. Käyttäjät tai asiakkaat tekevät tämän testin päättääkseen hyväksyvätkö he applikaation. (softwaretestinghelp n.d.)

- Systeemitestaus (System testing)

Koko systeemin testaus perustuen asiakkaan vaatimuksiin. Kattaa kaikki yhdistetyt osat systeemistä. (softwaretestinghelp n.d.)

- Turvallisuus testaus (Security testing)

Voiko systeemiin tunkeutua millään hakkerointitavalla. Testataan kuinka hyvin systeemi suojaa luvattomilta sisäisiltä tai ulkoisilta lähestymisiltä. Tarkastetaan, että systeemin tietokanta on turvassa ulkoisilta hyökkäyksiltä.
(softwaretestinghelp n.d.)

2.2.1 Automatisoitu ohjelmistotestaus

Mitkä ovat automatisoidun ohjelmistotestauksen ja manuaalisen ohjelmistotestauksen erot? Automatisoitu ohjelmistotestaus:

- Parantaa manuaalista testaustyöskentelyä keskittymällä automatisoimaan testejä, joita manuaalisella testauksella on vaikea suorittaa.
- On osa ohjelmistokehitystä
- Ei korvaa manuaalisien testaajien analyyttisiä taitoja, testausosaamista ja ymmärrystä testaustekniikoista. Nämä manuaalisten testaajien asiantuntemukset toimivat suunnitelmana automatisoituun testaukseen.
- Ei voi selvästi erottaa manuaalisesta testauksesta. Sekä automatisoitu että manuaalinen ohjelmistotestaus sitoutuvat yhteen ja täydentävät toisiaan.

On mahdollista kehittää ohjelmisto, joka muuttaa kaikenlaiset manuaaliset testit automatisoiduiksi, mutta useita manuaalisia testejä pitäisi modifioida hieman. Vaikka kaikki testit voidaan automatisoida, kaikki testit eivät ole automatisoinnin kannalta järkeviä.

Automatisoitu ohjelmistotestaus viittaa automatisointitarpeisiin koko ohjelmistotestaus elinkaaren ajan keskittymällä integraatio- ja systeemitestauksen automaitisointitarpeisiin. Automatisoidun ohjelmistotestauksen päämäärä on suunnitella, kehittää ja tuottaa automaattista testaus- ja uudelleentestauskapasiteettia, joka lisää testaustehokkuutta. Jos tämä implementoidaan onnistuneesti, se voi tuottaa huomattavan vähennyksen kustannukseen, aikaan ja resursseihin.

Automatisoitu ohjelmistotestaus ei ole rajattu tiettyyn testausvaiheeseen tai kenenkään valmistajan tuotteeseen, eikä sitä voi sovittaa tiettyyn arkkitehtuuriin tai ohjelmointikieleen, jota testattava applikaatio käyttää.

Jos automatisoitu ohjelmistotestaus implementoidaan tehokkaasti, pitäisi sen tukea applikaatioita, jotka

- Ajetaan useilla tietokoneilla

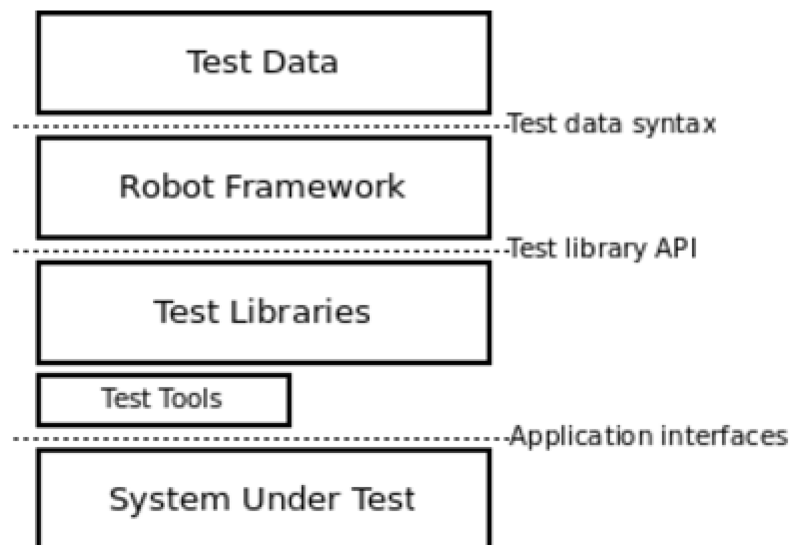
- Kehitetään eri ohjelmointikielillä
- Ajetaan erityyppisilla käyttöjärjestelmillä tai alustoilla
- Sisältää tai ei sisällä graafista käyttöjärjestelmää
- Ajavat minkä tahansa tyyppistä verkkoprotokollaa esim. TCP/IP, DDS jne.

(Dustin, Garrett & Gauf 2009.)

2.3 **Automatisoidun testauksen työkaluja**

2.3.1 Robot Framework

Robot Framework on pythonpohjainen testiautomaatioalusta. Yksi sen käyttökohteista on hyväksymistestaus ja hyväksymistestausvetoinen kehitys. Sitä voi käyttää myös testaukseen yleensä ja erilaisiin applikaatioihin, joissa käytetään eri tekniikoita ja useita rajapintoja. Robot Frameworkin komentorivirajapinnan avulla voidaan se integroida jo olemassa olevaan jatkuvan integraation järjestelmään. Robot Framework ei ole myöskään alustariippuvainen, joten sitä voi käyttää esimerkiksi Windowsilla tai Linux-pohjaisella käyttöjärjestelmällä. Robot Frameworkilla on moduuleista koostuva arkkitehtuuri, joka on esitettynä seuraavassa Kuvassa (Kuva 3.).



Kuva 3. Robot Framework arkkitehtuuri (robot framework user guide n.d.)

Robot Frameworkin testidata on taulukkomuotoista. Robot Frameworkin käynnistyessä se prosessoi testidatan, suorittaa testitapaukset ja luo lokit ja raportit. Ydinrunko ei tiedä mitään testattavasta kohteesta eikä vuorovaikutuksesta jota testikirjastot käsittelevät. Kirjastot voivat joko käyttää applikaation rajapintaa suoraan tai käyttää alemman tason testityökaluja ajureina. (robot framework user guide n.d.)

2.3.2 Jatkuva integraatio ja Jenkins työkalu

Jatkuva integraatio on modernin ohjelmistokehityksen kulmakivi. Silloin, kun jatkuvat integraatio otetaan organisaatiossa käyttöön se muuttaa radikaalisti tiimien ajattelutapaa koko ohjelmistokehitysprosessista. Sillä on potentiaalia sallia ja käynnistää sarja lisääntyviä prosessiparannuksia, helposta ajastetusta automaattisesta koostamisesta jatkuvaan toimitukseen tuotannossa. Hyvä jatkuvan integraation infrastruktuuri voi tehostaa ohjelmistokehitysprosessia käyttöönotossa, vikojen löytämisessä ja korjaamisessa, kojetaulun tuomisesta kehittäjillä ja muille osallisille ja lopulta tuottamaan enemmän liikearvoa loppukäyttäjälle.

Jatkuva integrointi yksinkertaisimmissa muodoissa sisältää työkalun, joka monitoroi versionhallinta systeemiä muutoksilta. Aina, kun muutos tapahtuu, tämä työkalu kääntää ja testaa kehitettävän applikaation. Jos jokin menee vikaan, työkalu ilmoittaa heti kehittäjille, jotta kehittäjät voivat korjata asian pikimmiten.
Jenkins on tähän tarkoitettu avoimen lähdekoodin jatkuvan integraation työkalu, joka on kirjoitettu Javalla. Jenkinsillä on yksinkertainen käyttöliittymä ja sen käyttö on helppo oppia. Jenkins on myös joustava ja helppo ottaa käyttöön. Jenkinsillä on satoja lisäosia mm. versionhallintaan (esim. GIT) ja testiautomaatioalustoihin (esim. Robot Framework). (Ferguson Smart 2011.)

2.3.3 Docker

Docker on kevyt virtualisointi työkalu, joka on suunniteltu helpottamaan applikaatioiden luomisprosessessia käyttäen virtuaaliyksikköjä, joita kutsutaan konteiksi. Docker-kontteja sanotaan usein pieniksi virtuaalikoneiksi, koska konteilla ja virtaalikoneilla on paljon yhtäläisyyksiä. Molemmat on suunniteltu tarjoamaan eristetty ympäristö, jossa voi ajaa applikaatioita. Lisäksi molemmissa tapauksissa ympäristö on esitettynä binäärisenä tuotteena jonka voi siirtää isäntäkoneiden välillä.

Docker-kontit jakavat Docker-isäntäkoneen alustan resurssit. Lisäksi kehittäjät rakentavat Docker-kuvan, johon sisältyy vain se mitä tarvitaan kehittäjän applikaation ajoon.
Virtuaalikoneet rakennetaan vastakkaisesta suunnasta. Ne aloitetaan kokonaisella käyttöjärjestelmällä, ja riippuen applikaatiosta kehittäjät voivat tai eivät voi poistaa ei haluttuja komponentteja.

Docker-kontit pakkaavat sisäänsä palan ohjelmistoa kokonaiseen tiedostojärjestelmään, joka sisältää kaikki mitä tarvitaan ajamaan: koodia, systeemityökaluja, systeemikirjastoja ja mitä tahansa mitä voidaan asentaa palvelimelle. Tämä takaa, että ohjelmisto ajetaan aina samalla tavalla riippumatta sen ympäristöstä.

Kontit, jotka ajavat yhdellä koneella jakavat saman Kernelkäyttöjärjestelmän. Täten kontit käynnistyvät hetkessä ja käyttävät vähemmän RAM-muistia. Docker-kuvat on rakennettu kerroksellisesta tiedostojärjestelmästä ja jakavat samoja tiedostoja tehden kiintolevyn käytöstä ja Docker-kuvien latauksesta paljon tehokkaampaa.

Docker-kontit perustuvat avoimiin standardeihin ja sallivat konttien ajon kaikilla isoilla Linux-jakeluilla ja Windowsilla ja minkä tahansa infrastruktuurin päällä. Kontit eristävät applikaatiot toisistaan ja alustan infrastruktuurista ja tarjoavat samalla lisätyn suojakerroksen applikaatioon. (what is docker n.d.)

2.4 Pilvilaskenta

Informaatioteknologian kehitys verkkomallista kohti pilvimallia johtui osittain siitä, että tarvittiin suuri määrä laskentaresursseja ratkaisemaan yhtä ongelmaa. Potentiaalinen ero verkkomallin ja pilvimallin välillä on se, että verkkomalli tukee useiden tietokoneiden tehon rinnakkaistamista, jotta se pystyisi suorittamaan tietyn applikaation. Pilvimalli taas tukee tehon jakamista useille resursseille, mihin sisältyy laskentaresurssit, jotta se pystyy tuottamaan yhtenäisen palvelun loppukäyttäjälle.
Pilvilaskennassa IT- ja liiketoimintaresursseihin kuuluvat mm. palvelimet, tallennustila, tietoverkko, applikaatiot ja prosessit. Nämä voidaan dynaamisesti varata käyttäjän tarpeiden ja työkuorman mukaan. (Chandrasekaran 2014, 5.)
Yksinkertaisimmillaan pilvilaskenta tarkoittaa datan tallennusta ja dataan ja ohjelmiin käsiksi pääsyä internetin yli sen sijaan, että tämä tapahtuisi oman tietokoneen kovalevyltä. (Chandrasekaran 2014, 12.)

2.5 Pilvipalvelumallit

On olemassa kolmenlaisia pilvipohjaisia palveluita loppukäyttäjille. Software as a Service (SaaS), Platform as a Service, (PaaS), ja Infrastructure as a Service (IaaS). Saas on ohjelmiston levittämismalli, jossa applikaatiot ovat isännöitynä toimittajalla tai palvelun tarjoajalla ja asiakkaat pääsevät niihin käsiksi tietoverkon yli, joka on useimmiten internetti. PaaSin idea on toimittaa käyttöliittymät ja niihin liittyvät palvelut (esim. tietokoneen insinöörityön ohjelmistot, integroidut kehitysympäristöt ohjelmistokehitysratkaisuihin) internetin yli ilman latauksia tai asennuksia. IaaS tarkoittaa

tukitoimilaitteiden ulkoistamista. Näitä ovat tallennustila, laitteistot, palvelimet ja tietoverkkokomponentit. (Chandrasekaran 2014, 16.)

2.5.1 Software as a Service (SaaS)

SaaS-mallissa käyttäjälle tarjotaan pilvialustalla toimivan palveluntarjoajan applikaation käyttömahdollisuus. Applikaatioon pääse käsiksi useilta eri laitteilta, joko verkkoselaimella (esim. verkkopohjainen sähköposti.) tai ohjelman käyttöliittymältä. Käyttäjä ei ohjaa tai kontrolloi pilvi-infrastruktuurin alustaa. Tyypillisesti applikaatiot tarjotaan palveluna, joka sisältää asiakkaiden välisien suhteiden hallinnan, liiketoimintatiedon analysoinnin ja verkkolaskentaohjelmiston. (Chandrasekaran 2014, 16-17.)

2.5.2 Platform as a Service, (PaaS)

PaaS-mallissa käyttäjä pystyy ajamaan ja tuottamaan applikaatioita pilvi-infrastruktuurissa käyttäen ohjelmointikielijä, kirjastoja, palveluita ja työkaluja, joita palveluntarjoaja tukee. Käyttäjä ei ohjaa tai kontrolloi pilvi-infrastruktuurin alustaa, mutta kontrolloi applikaatioita ja mahdollisesti applikaation isännöintiympäristön asetuksia. Toisin sanoen, se on valmiiksi pakattu kehitysrunko. PaaS-toimittajat tarjoavat tietoverkon, palvelimet ja tallennustilan ja hallinnoivat skaalautuvuuden tasoja ja huoltoa. Asiakas tyypillisesti maksaa käytetystä palveluista. Esimerkkejä Paas-tarjoajista ovat IBM Bluemix ja Google App Engine. (Chandrasekaran 2014, 17.)

2.5.3 Infrastructure as a Service (IaaS)

IaaS-mallissa käyttäjälle tarjotaan tallennustila, tietoverkot ja kaikki muu olennainen laskentaresurssi "maksa vain käytöstä –periaatteella”. Käyttäjä voi ajaa satunnaisia ohjelmistoja, joihin voi sisältyä käyttöjärjestelmiä ja applikaatioita. Käyttäjä ei ohjaa tai kontrolloi pilvi-infrastruktuurin alustaa, mutta kontrolloi käyttöjärjestelmiä, tallennustilaa, ja applikaatioiden ajamista ja mahdollisesti kontrolloi rajatusti joitain tietoverkkokomponentteja (esim. palomuuri). Palvelun tarjoaja omistaa laitteet ja on vastuussa jäähdytyksestä ja huollosta. Amazon Web Services (AWS) on hyvä esimerkki suuresta IaaS-tarjoajasta. (Chandrasekaran 2014, 17.)

2.6 **Amazon Web Services (AWS)**

Amazon Web Services (AWS) tarjoaa laskentaresursseja ja palveluja tarvittaessa (on-demand) pilvessä, maksa vaan käytöstä -periaatteella. AWS:ssä

voi ajaa palvelinta, jonka voi konfiguroida ja ajaa sitä, kuin se olisi vieressäsi. AWS:ssä voi käyttää kapasiteettia juuri niin paljon kuin tarvitsee ja maksu vain käytöstä. (AWS getting started n.d.)

2.6.1 AWS EC2

Amazon Elastic Compute Cloud (Amazon EC2) on web-palvelu joka tarjoaa koon muuttavaa laskentakapasiteettia pilvessä. Se on suunniteltu helpottamaan web-skaalaus pilvilaskentaa kehittäjille.

Amazon EC2:sen yksinkertainen web-palvelukäyttöliittymä sallii kapasiteetin saamisen ja konfiguraation minimaalisella työllä. Se tarjoaa täydellisen kontrollin laskentaresursseihin ja sallii Amazonin takaaman laskentaympäristön ajon. Amazon EC2 pienentää tarvittavaa aikaa uuden serverin käynnistämiseen minuutteihin, sallien nopean kapasiteetin skaalauksen ylös ja alas muuttuvissa laskentatarpeissa. EC2 tarjoaa kehittäjille työkalut vikoja kestävävien applikaatioiden rakentamiseen ja niiden eristämiseen vika mahdollisuuksista. (AWS ec2 n.d.)

2.6.2 AWS EBS Volumes

Amazon EBS-volume on kestävä lohkotason tallennustilalaite, jonka voi kiinnittää yhteen EC2-instanssiin. ESB-volumeita voi käyttää ensisijaiseen tallennukseen datalle, mikä tarvitsee toistuvia päivityksiä. Esim. systeemin ajuri instanssille tai tallennustila tietokantasovellukselle tai suoritustehona tehokkaille sovelluksille, jotka suorittavat jatkuvaa levyn skannausta. EBS-volumet jatkavat olemassaoloaan itsenäisesti EC2-instanssin juoksevasta maailmasta. Volumen instanssiin liittämisen jälkeen, sitä voidaan käyttää, kuin fyysistä kovalevyä. Amazon EBS tarjoaa seuraavia volumetyyppejä: General Purpose SSD (gp2), Provisioned IOPS SSD (io1), Throughput Optimized HDD (st1), Cold HDD (sc1), and Magnetic (standard). Ne eroavat suoritusominaisuuksiltaan ja hinnaltaan.
(AWS EBS volumes n.d.)

2.6.3 Amazon Virtual Private Cloud (VPC)

Amazon Virtual Private Cloud (Amazon VPC) sallii provisioinnin loogisesti eristetystä alueesta AWS-pilvessä, jossa voit käynnistää ja hallita AWS-resursseja virtuaaliverkossa, joka voidaan määritellä tarpeiden mukaan. VPC:ssä voidaan kontrolloida virtuaaliverkkoympäristöä, mukaan lukien valittujen IP-osoitteiden joukkoa, aliverkkojen luontia ja reititystaulujen ja verkko yhdyskäytävän konfiguraatiota. VPC:ssä on käytössä sekä IPv4 ja IPv6 joilla pääsee käsiksi resursseihin ja applikaatioihin.

Lisäksi voit luoda Hardware Virtual Private Network (VPN) yhteyden yrityksen datakeskuksen ja AWS:n VPC:n kesken. Näin voit laajentaa yrityksesi datakeskusta.
(AWS VPC n.d.)

2.7 Kubernetes työkalu

Konttien käyttämisen yleistyttyä, saattaa yrityksellä olla satoja tai jopa tuhansia kontteja ajamassa pilvessä samaan aikaan. Tällöin tarvitaan strategia konttien ylläpitoon. Kuinka automaattinen palautuminen tapahtuu, kun kontti vikaantuu? Mihin palveluihin tämä katkos vaikuttaa? Kuinka päivittää ohjelmistot minimaallisella hukka-ajalla. Kuinka skaalata lisää tilaa konteille ja palveluille, kun liikennöinti kasvaa? Kuinka sijoittaa kontit lähekkäin klusterissa? Onko yhteistä dataa joihin kontit tarvitsevat pääsyä? Kuinka uudet palvelut huomataan ja kuinka ne tehdään saatavilla oleviksi muille systeemeille?

Avain on resurssien hyötykäyttäminen. Konttien pieni "jalanjälki" tarkoittaa sitä, että infrastruktuuri voidaan optimoida paremmin hyödynnettäväksi. Tämä tapahtuu laajentamalla säästöjä joita saadaan elastisessa pilvimaailmassa kohti tuhlatun raudan minimointia. Kuinka aikataulutamme työkuormamme kaikkein tehokkaimmin? Kuinka pidämme huolen siitä, että kaikkein tärkeimmillä applikaatiollamme on resursseja? Kuinka voimme ajaa vähemmän tärkeitä työkuormia varakapasiteetilla?

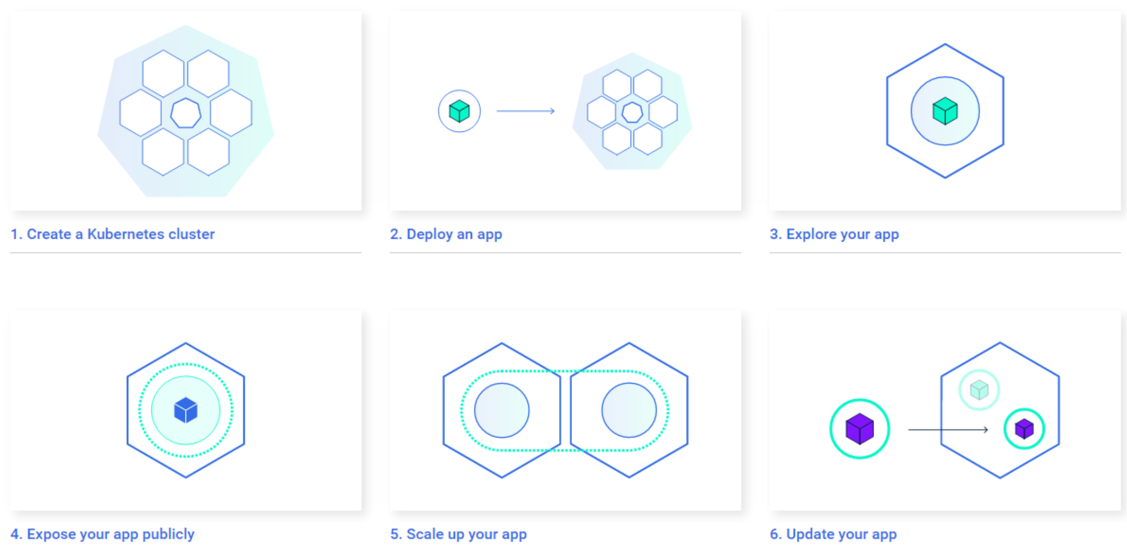
Lopullinen avain monien organisaatioiden siirtämiseen konttiaikaan on siirrettävyys. Docker tekee helpoksi standardin kontin elinkaariprosessin monilla eri käyttöjärjestelmillä, pilvitarjoajilla ja paikallisesti raudalla tai jopa kehittäjien kannettavilla tietokoneilla. Kuitenkin tarvitsemme työkalun konttien liikuttamiseen. Kuinka siirtää kontteja eri noodien välillä klusterissa? Kuinka julkistaa päivitykset minimaalisella häiriöllä?

Tässä vaiheessa orkestrointi työkalut, kuten Kubernetes tuovat suuren lisäarvon. Kubertnetes on avoimen lähdekoodin projekti, jonka Google julkaisi vuonna 2014. Google julkaisi projektin osana Googlen oman infrastruktuurin ja teknologia edun jakamista yleisesti.

Google laukaisee 2 miljardia konttia viikossa omassa infrastruktuurissaan ja on käyttänyt kontti teknologiaa jo yli 10 vuotta. Alunperin google rakensi systeemiä nimeltä Borg, ja sittemmin Omega jatkuvasti laajentuvalle palvelinkeskukselle. Google otti vuosien varrella opitut asiat huomioon ja uudelleen kirjoittivat data centerin hallinto työkalun kaikille käytettäväksi. Tuloksena oli Kubernetes avoimen lähdekoodin projekti. (Baier 2015.)

Kubernetes on siis avoimen lähdekoodin systeemi, konttivetoisten applikaatioiden ohjelmiston elinkaariprosessin automatisointiin, skaalaukseen

ja hallintaan. Se ryhmittää kontteja, jotka tekevät applikaatiosta loogisia yksikköjä, jotka ovat helposti hallittavissa ja löydettävissä.
(Kubernetes Container Orchestration n.d.)



Kuva 4. Kubernetes periaate (Kubernetes Overview n.d.)

2.7.1 Kubernetes podit

Pod on ryhmä kontteja (esim. Docker-kontteja), jaettu tallennustila näille konteille ja vaihtoehdot konttien ajamiselle. Podit ovat aina samapaikkaistettuja ja sama-aikakataulutettuja ja ne ajetaan jaetulla sisällöllä. Pod mallintaa applikaatiospesifistä ”loogista isäntää”, joka sisältää yhden tai useamman applikaatiokontin, jotka ovat suhteellisen tiivisti yhdistetty. Maailmassa ennen kontteja nämä olisi suoritettu samalla fyysisellä tai virtuaalisella koneella.

Samalla, kun Kubernetes tukee useampia konttien ajoja, kuin Docker, niin Docker on tunnetuin konttien ajoalusta ja se auttaa kuvailemaan podeja Dockerin termein.

Podin jaettu sisältö on sarja Linuxin nimiavaruuksia, kontrolliryhmiä (cgroups) ja potentiaalisesti muita eristyksen puolia. Nämä ovat samoja asioita, jotka eristävät docker-kontteja. Yksittäisellä applikaatiolla voi olla alieristyksiä podin sisällön sisällä.

Podin sisällä olevat kontit jakavat IP-osoitteen ja porttitilan, ja löytävät toisensa localhostin kautta. Kontit voivat myös kommunikoida toistensa kanssa käyttäen standardoituja prosessien välisiä kommunikaatioita (IPC) kuten SystemV semaphores tai POSIX shared memory. Eri podien konteilla on eri IP-soitteet ja ne eivät voi kommunikoida käyttäen IPC:tä.

Podin sisäisillä applikaatioilla on pääsy jaettuihin volumeihin, jotka ovat määritelty osaksi podia ja ovat tehty saatavilla oleviksi ja ne liitetään jokaisen applikaation tiedostojärjestelmään.

Docker termein, pod on mallinnettu ryhmäksi docker-kontteja jotka jakavat nimiavaruudet ja volumet. PID(process identification number) nimiavaruuksien jakaminen ei ole vielä implementoitu dockeriin.

Niin kuin yksittäisiä applikaatiokontteja, podeja pidetään suhteellisen hetkellisinä (ennemminkin kuin pitkäikäisinä) kokonaisuuksina. Podit luodaan, niille määrätään uniikki ID (UID) ja ajoitetaan noodehin mihin ne jäävät kunnes ne terminoidaan (uudelleenkäynnistyskäytännön mukaan) tai poistetaan. Jos noodi kuolee, siihen ajoitetut podit ajoitetaan poistoon aikakatkaisu jakson jälkeen. Annettu pod (jonka UID on määritellyt) ei ole uudelleen ajoitettu tähän noodiin. Tämän sijaan se voidaan korvata identtisellä podilla, jolla on jopa sama nimi jos tarpeen mutta uusi UID.

Kun jollakin on sama elinikä, kun podilla esim. volumella se tarkoittaa, että se on olemassa yhtä kauan kun pod (sillä UID:llä) on olemassa. Jos se pod poistetaan jostain syystä, vaikka identtinen pod luodaan tilalle, niin volume poistetaan myös ja tilalle luodaan uusi. (Kubernetes pods n.d.)

2.7.2 Kubernetes Volumes

Levyllä olevat tiedostot konteissa ovat lyhytaikaisia, mistä aiheutuu joitain ongelmia epätriviaalien applikaatioiden kanssa, joita ajetaan konteissa. Ensimmäiseksi kun kontti kaatuu kubelet uudelleenkäynnistää sen mutta tiedostot katoavat. Tämän jälkeen kontti käynnistyy puhtaassa tilassa eli kaikki tallennetut tiedostot kontista katoavat. Toiseksi, kun ajetaan kontteja yhdessä podin kanssa on usein tarpeellista jakaa tiedostoja konttien välillä. Kubernetes volumet ratkaisevat nämä molemmat ongelmat. (Kubernetes Volumes n.d.)

2.7.3 Kubernetes PersistentVolumes

Tallennustilan hallinnointi on erityinen ongelma, kun hallinnoidaan laskentaa. PersistentVolume alisysteemi tarjoaa rajapinnan käyttäjille ja ylläpitäjille vetää yhteen yksityiskohdat siitä, kuinka tallennustilaa tarjotaan ja kuinka sitä käytetään. Tähän on tarjolla kaksi rajapintaresurssia: PersistentVolume ja PersistentVolumeClaim.

PersistentVolume on pala verkotettua tallennustilaa klusterissa, jonka ylläpitäjä on provisioinut. Se on resurssi klusterissa niin kuin noodi on klusterin resurssi. PersistentVolumet ovat volumeliitäännäisiä, kuten volumet,

mutta niillä on elinkaari, joka on riippumaton kaikista yksittäisistä podeista, jotka käyttävät PersistentVolumea. Tämä rajapinta objekti kaappaa yksityiskohdat tallennustilan implementoinnista esim. pilvitarjoajaspesifisessä tallennustilasysteemissä.

PersistentVolumeClaim on pyyntö tallennustilasta käyttäjältä. Se on samanlainen kuin pod. Podit käyttävät noodiresursseja ja PersistentVolumeClaimit käyttävät PersistentVolumen resursseja. Podit voivat pyytää spesifisen tason resursseja (CPU ja muisti). Claimit voivat pyytää spesifisen koon ja pääsyn ilmenemismuotoja (esim. voidaan liittää luku/kirjoitus tai monta kertaa pelkkä luku)

Siinä missä PVC sallii käyttäjän käyttää abstrakteja tallennusresursseja, on tavallista , että käyttäjät tarvitsevat PV:tä vaihtelevilla ominaisuuksilla, kuten suorituskykyä erilaisiin ongelmiin. Klusterin ylläpitäjien tarvitsee voida tarjota erilaisia PV:tä, jotka eroavat muillakin tavoilla kuin koko ja pääsyoikeus. Tämä kaikki täyttyy tapahtua ilman, että paljastaa käyttäjälle yksityiskohtia volumen implementoinnista. Näihin tarpeisiin on StorageClass-resurssi.

StorogeClass tarjoaa yllöpitäjille tavan kuvailla ”luokkia” joita he tarjoavat. Eri luokat voivat kartoittaa palvelun laadun tasoja tai varmuuskopiokäytäntöjä, joita klusterin ylläpitäjät ovat määrittäneet. Kubernetesilla ei ole mielipidettä siitä kuinka luokkia esitetään. Tämä konsepti on joskus nimeltään ”profiilit” toisissa tallennustilasysteemeissä.

Volumen elinkaari ja vaatimus
PersistentVolumet ovat klusterin resursseja. PersistentVolumeClaimit ovat pyyntöjä noille resursseille ja toimivat myös vaatimustarkastuksina resurssille. Vuorovaikutus PersistentVolumen ja PersistentVolumeClaimin välillä seuraa seuraavaa elinkaarta:

Provisiointi
On olemassa kaksi tapaa PersistentVolumen provisiointiin: staattinen ja dynaaminen.

Staattinen
Klusterin ylläpitäjä luo jonkin määrän PersistentVolumeja. Ne sisältävät yksityiskohdat oikeasta tallennustilasta joka on saatavilla klusterin käyttöön. Ne ovat Kubenetes-rajapinnassa ja ne ovat valmiita käyttöön.

Dynamic
Silloin kun yksikään staattisista PersistentVolumesta, joita ylläpitäjä on luonut ei vastaa käyttäjän PersistentVolumeClaimia, klusteri voi yrittää dynaamisesti provisioida volumen nimenomaan PersistentVolumeClaimille. Tämä provisiointi perustuu StorageClasseihin. PersistentVolumeClaimin täytyy pyytää luokkaa ja ylläpitäjän on täytynyt luoda ja konfiguroida tämä luokka, jotta dynaaminen provisiointi tapahtuu. Claimit jotka pyytävät

luokkaa ”X” tehokkaasti kytkevät pois päältä provisioinnin. (Kubernetes persistent volumes n.d.)

2.7.4 Kubernetes Deployment

Deployment tarjoaa selittäviä päivityksiä podeille ja ReplicaSeteille (seuraavan sukupolven ReplicationController). Tarvitsee vain kuvailla haluttu tila Deployment objektissa ja Deployment ohjain muuttaa tilan haluttuun tilaan kontrolloidulla tasolla. Voidaan myös määritellä Deployment luomaan uusia resursseja tai korvaamaan olemassa olevia resursseja uusilla.
Tyypillisiä käyttötapauksia ovat:

- Luodaan Deployment nostamaan ylös ReplicaSet ja podeja
- Tarkistetaan Deploymentin status nähdäkseen onnistuiko se vai ei
- Myöhemmin päivittämään Deployment ja uudelleen luomaan podit (esim. jos halutaan käyttää uutta kuvaa)
- Palauttaa aiempi Deployment revisio jos nykyinen Deployment ei ole vakaa
- Pysäyttää ja jatkaa Deploymenttia

(Kubernetes Deployments n.d.)

2.7.5 Kubernetes Node

Kubernetes Node eli noodi on työtä tekevä kone Kubernetes ohjelmassa, joka tunnettiin ennen nimellä minion. Noodi voi olla virtuaalikone tai fyysinen kone riippuen klusterista. Jokaisella noodilla on tarvittavat palvelut podien ajamiseen ja master-komponentit manageroivat noodeja. Noodin palveluihin sisältyy Docker, kubelet ja kube-proxy. (Kubernetes node n.d.)

2.7.6 Kubernetes Replication Controller

Replication Controller varmistaa, että spesifinen määrä pod ”kopioita” ajaa samaa aikaan. Toisin sanoen Replication Controller varmistaa, että pod tai yhtenäinen sarja podeja on aina saatavilla. Jos podeja on liikaa Replication Controller tappaa ne. Jos podeja on liian vähän, niin Replication Controller käynnistää niitä lisää. Toisin kuin manuaalisesti luodut podit, Replication Controllerin ylläpitämät podit korvataan automaattisesti jos ne vikaantuvat, ne poistetaan tai ne terminoidaan. Esimerkiksi pod uudelleen luodaan noodilla häiritsevän huollon jälkeen, kuten kernel päivitys. Tästä syystä Replication Controllerin käyttö on kannattavaa vaikka applikaatio tarvitsee vain yhden podin. Replication Controller on vähän kuin prosessin valvoja, mutta sen sijaan, että se valvoo yksittäistä prosessia yhdellä noodille, se valvookin useita podeja useilla noodeilla.

Yksinkertainen tapaus on luoda yksi Replication Controller objekti, tarkoituksena luotettavasti ajaa pod instanssia toistaiseksi. Monimutkaisempi käyttötapaus on ajaa useita samanlaisia kopioita kopioiduista palveluista, kuten webpalvelimet. (Kubernetes replication controller n.d.)

2.8 Terraform

Terraform on työkalu infrastruktuurin rakentamiseen, muuttamiseen ja versiointiin. Terraformilla voi hallita olemassa olevia ja suosittuja palvelun tuottajia kuin myös kustomoituja in-house ratkaisuja.

Konfiguraatio tiedostot kertovat Terraformille, mitä komponentteja tarvitaan yksittäisen applikaation tai koko palvelinkeskuksen ajamiseen. Terroform luo suoritussuunnitelman, missä kerrotaan mitä Terraform tekee, jotta päästään haluttuun tilaan. Tämän jälkeen Terraform suorittaa suunnitelman ja rakentaa suunnitellun infrastruktuurin. Konfiguraation muuttuessa, Terraform pystyy määrittelemään mikä muuttui ja luomaan lisääntyvän suoritussuunnitelman, jota voidaan käyttää.

Infrastruktuuri, jota Terraform voi hallita sisältää alatason komponentteja kuten laskenta instansseja, säilytystä ja tietoverkkoja kuin myös ylätason komponentteja kuten DNS-kirjauksia ja SaaS-ominaisuuksia.

Terraformin avainominaisuuksia ovat:

- **Infrastruktuuri koodina:** Infrastruktuuri on kuvailtu käyttäen ylätason konfiguraatio syntaksia. Tämä sallii palvelinkeskuksen piirustuksien versioinnin ja sen, että sitä voi kohdella kuin mitä tahansa ohjelmistoa. Lisäksi infrastruktuuria voidaan uudelleenkäyttää ja jakaa.

- **Suoritussuunnitelmat:** Terraformilla on "suunnitteluvaihe" missä se luo suoritussuunnitelman. Suoritussuunnitelma näyttää mitä Terraform aikoo tehdä kun ajetaan apply-komento. Tämän avulla vältytään yllätyksiltä, kun Terrafor manipuloi infrastruktuuria.

- **Resurssigraafi:** Terraform rakentaa graafin kaikista resursseista ja rinnakkaistaa luonnin ja muutoksen teot kaikille toisista riippumatta oleville resursseille. Tämän takia Terraform rakentaa infrastruktuurin mahdollisimman tehokkaasti ja operaattorit saavat käsityksen riippuvuuksista infrastruktuurissa.

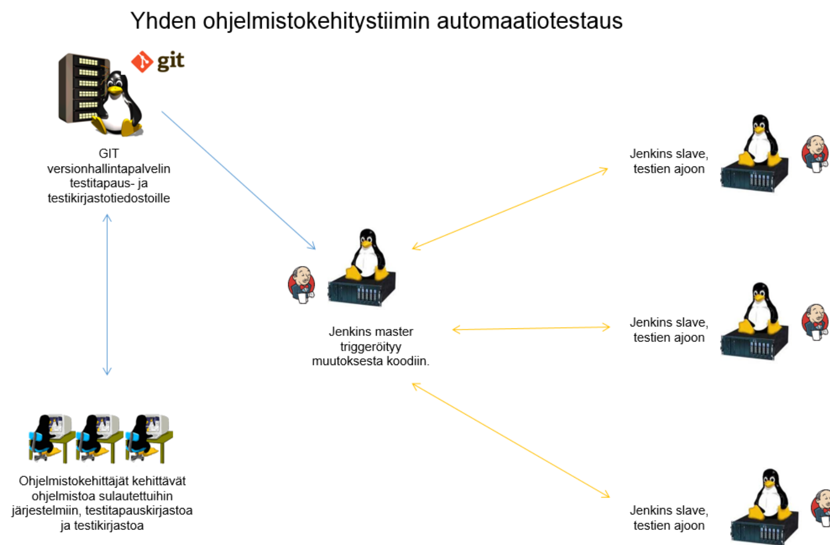
- **Muutosautomaatio:** Monimutkaiset muutossarjat voidaan panna toimeen infrastruktuuriin minimaalisella ihmisvaikutuksella. Aiemmin mainitulla suoritussuunnitelmalla ja resurssigraafilla tiedetään tarkalleen mitä Terraform aikoo muuttaa ja missä järjestyksessä ja tällöin vältytään mahdollisilta ihmisen virheiltä. (Terraform intro n.d.)

3 PILVITESTAUS AWS

3.1 Koneen ohjelmistokehityksen testaus

Koneen tuotekehityksen ohjelmistokehityksen testaus tapahtuu tyypillisesti seuraavanlaisesti: Ohjelmistokehittäjät kehittävät uutta ohjelmistoa yleensä paikallisesti omilla työasemillaan Scrum-mallin mukaisesti. Kehittäjät ”kloonavat” koko ohjelmiston Git-palvelimelta omalle koneelleen ja alkavat kehittämään ohjelmistoa. Kun koodiin saadaan tehtyä esimerkiksi uusi funktio, ”työnnetään” koko ohjelmisto takaisin versionhallinta ohjelmistoon, jolloin Jenkins-palvelin ”haistaa” muutoksen ohjelmistossa ja käynnistää automaattisten testauksien putken. Jos testit menevät läpi, jatkaa kehittäjä uuden funktion tai toiminnollisuuden tekemistä. Jos testeissä löytyy virhe koodista, korjaa kehittäjä koodin ja ajaa testit uudestaan. Näin syntyy uutta ohjelmistoa Koneen tuotteisiin, joita kutsutaan sulautetuiksi järjestelmiksi.

Jokaisella ohjelmistokehitystiimillä on yksi oma Jenkins-master-kone joka ”haistelee” Git-ohjelmaan ”työnnettyjä” ohjelmistoja ja käynnistää testiputken kun muutos ohjelmistoon tapahtuu. Kun testiputki käynnistyy, varaa Git yhden slave-koneen testaamista varten. Slave-koneiden määrä on rajattu ja niitä on yhdelle master-koneelle varattuna rajattu määrä. Tämä tuo eteen ongelman tilanteessa jossa useampi kehittäjä on käynnistänyt testiputken samaan aikaan. Tällöin Jenkins laittaa jonoon viimeisenä testiputken käynnistyksen aloittaneen kehittäjän. Kehittäjä joutuu odottelemaan, kunnes slave-kone vapautuu ja testiputki voi alkaa. Tästä syystä kehittäjä voi joutua odottelemaan pitkänkin aikaa testiputken alkamista. Kun testit lopulta suoritetaan, niin kehittäjällä ei välttämättä ole enää ihan tuoreessa muistissa tekemänsä ohjelmisto, mikä vaikeuttaa mm. vianhakua. Kuvassa 5. on havainnollistettuna yhden ohjelmistokehitystiimin automaatiotestaus.



Kuva 5. Yhden ohjelmistokehitystiimin automaatiotestaus

Toinen ongelma on laskentakapasiteetin määrä. Koska yhdellä tiimillä on rajattu määrä Jenkins-slave-koneita, suoritetaan kaikki testit sarjassa peräkkäin. Jos laskentakapasiteettia olisi enemmän, pystyisi testejä rinnakkaistamaan ja testiaikaa pienennettyä.
Koska slave-koneet ovat vain testien ajon aikana käytössä ja koska laskentakapasiteetin maksimoiminen tarkoittaisi suuria investointeja laitteisiin joiden käyttäminen tapahtuisi suurimmaksi osaksi vain arkipäivisin klo 8-16 tarvitaan vaihtoehto, jossa voidaan ”vuokrata” slave-koneita silloin kun testeille on tarvetta. Pilvipalveluissa tarjotaan laskentakapasiteettia tarvittaessa (on-demand). Tälläin voidaan pystyttää slave-koneita testien ajaksi ja ajaa slave-koneet alas testien jälkeen. Tällöin pilvipalvelu jossa maksetaan vain käytöstä olisi sopiva vaihtoehto.

3.2 Jenkins-setapin suunnitelma

Tarkoituksena on pystyttää Jenkins-setappi PoC-menetelmää (Proof of Concept) käyttäen AWS pilveen. Tarkoituksena ei ole pystyttää tuotantoympäristöä vaan todentaa, että tuotantoympäristön pystyttäminen on mahdollista.

Seuraavat vaiheet sisältyvät setapin pystytykseen:

1. AWS konfigurointi
2. Tarvittavien ohjelmien asennus
   - AWS CLI
   - Kubernetes
   - Kops
   - Terraform
3. AWS CLI konfigurointi

4. Verkkotunnuksen rekisteröinti
5. S3 bucket luonti
6. Klusterin luonti
7. Klusterin autoskaalaus
8. Jenkins-volumen konfigurointi
9. Jenkins-master konfigurointi

Ensimmäisenä on tehtävä tarvittavat konfiguroinnit Amazon Web Servicessä, jotta AWS CLI komentorivi käyttöliittymältä päästään käsiksi AWS pilveen. Tämän jälkeen voidaan asentaa AWS CLI ja konfiguroida se ja asentaa Kops, Kubernetes ja Terraform.

Toisena täytyy rekisteröidä verkkotunnus, jotta käyttäjät pääsevät luotuun klusteriin käsiksi ja S3 bucket luodaan, jotta Kops pystyy tallentamaan konfiguraation AWS:ään. Klusteri luodaan Kopsilla ja Terraformia käytetään klusterin versionhallintaan.

Klusterin autoskaalaukseen käytetään cluster-autoscaler konttia, joka automaattisesti säätää klusterin kokoa tarpeen vaatiessa.
Jenkins-master-kontille täytyy myös määritellä volume, jotta master-kontin tiedot eivät katoa jos kontti uudelleenkäynnistetään. Lopuksi vielä määritellään Jenkins-master ja Jenkins-slave tarpeet.

3.3 AWS testausvaiheet

Seuraavassa osiossa on kerrottu vaiheittain, kuinka PoC-testaus etenee. Tässä PoC-testauksessa on käytetty Ubuntu 14.04 LTS käyttöjärjestelmää.

3.3.1 Tarvittavat ohjelmistot

- **Kubernetes:** Kubernets on ohjelma konttivetoisten applikaatioiden ohjelmiston elinkaariprosessin automatisointiin, skaalaukseen ja hallintaa. Se ryhmittää kontteja, jotka tekevät applikaatiosta loogisia yksikköjä, jotka ovat helposti hallittavissa ja löydettävissä.

- **Kops (Kubernetes Operations):** Kops:lla voi ajaa tuotantolaatuisia Kubernetes klustereita komentoriviltä. Kops tukee Kubernetes ajamista AWS:ssä. (What is Kops n.d.)

- **Terraform:** Terraform on työkalu infrastruktuurin rakentamiseen, muuttamiseen ja versiointiin. Terraformilla voi hallita olemassa olevia ja suosittuja palvelun tuottajia kuin myös kustomoituja in-house ratkaisuja.

- **AWS CLI:** AWS CLI on komentorivi käyttöliittymä, jolla voi hallita AWS palveluita

3.3.2 AWS CLI konfigurointi

Ensimmäinen vaihe on konfiguroida AWS CLI. AWS käyttää Access Key avainta CLI konfiguraatiossa, joten ensimmäiseksi täytyy luoda Access Key avain. Access Key luodaan AWS Identity and Access Management selainkäyttöliittymässä ja vain luonti hetkellä on turvallisuus syistä mahdollisuus tallettaa AWS Secret Access Key.
Lisäksi valitaan alue missä halutaan toimia ja tulostemuoto. AWS tukee kolmea eri tulostemuotoa:

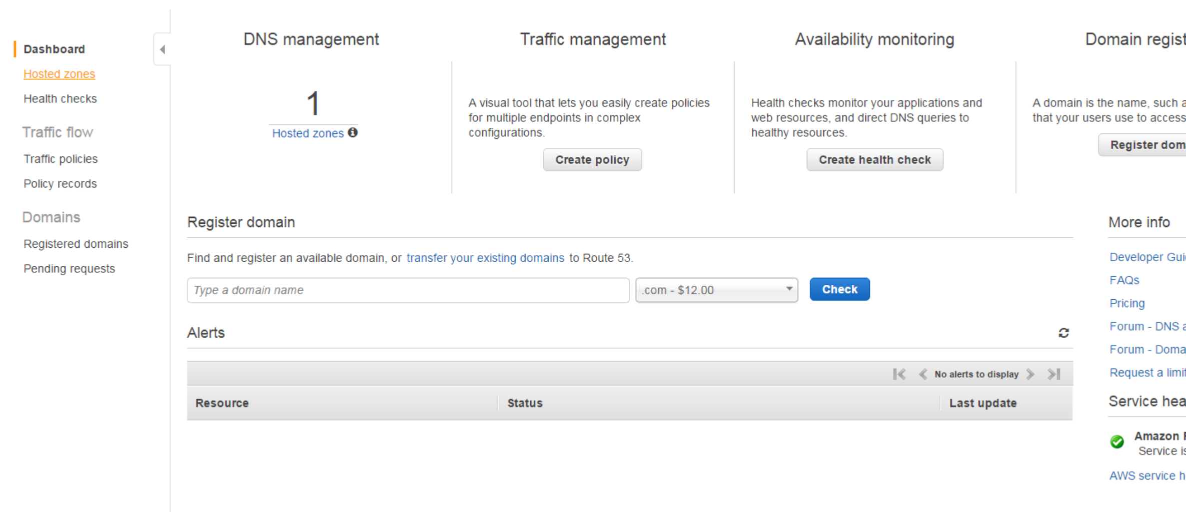
- JSON (json)
- Tab-delimited text (text)
- ASCII-formatted table (table)

```
$ aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: ENTER
```

Kuva 6. AWS CLI konfigurointi (AWS CLI getting started n.d.)

3.3.3 Verkkotunnuksen rekisteröinti

AWS käyttää Route 53 nimistä Domain Name System (DNS) palvelua. Tätä reittiä käyttää myös Kops/Kubernetes. Kops käyttää DNS-palvelua löytämiseen klusterin sisällä ja myös siihen, että asiakasohjelmalta pääse kubernetes API rajapintaan kiinni. (Kubernetes getting started n.d.)
Verkkotunnuksen voi rekisteröidä joko komentoriviltä tai selainkäyttöliittymästä, kuten kuvassa 7. Tässä esimerkissä käytetään kantaosoitetta, mutta myös alisoitteiden käyttö ja luonti on mahdollista.



Kuva 7. Verkkotunnuksen rekisteröinti (Route 53 home n.d.)

3.3.4 S3 bucket luonti

S3 on Amazonin yksinkertainen tallennuspalvelu (Simple Storage Service) internetissä. Bucket taas on kansio joka luodaan S3 tallennuspalveluun. Kops tallentaa klusterin konfiguraation ja muut tiedot klusterista S3 buckettiin. S3 bucketin voi luoda komentoriviltä komennolla: aws s3 mb s3:// example tai selainkäyttöliittymästä.
Lisäksi täytyy määritellä ympäristömuuttuja KOPS_STATE_STORE, jotta Kops tietää mitä buckettia käyttää. Tämä onnistuu komennolla: **export KOPS_STATE_STORE=s3://example.**
(Kubernetes getting started n.d.)

3.3.5 Klusterin luonti

Bucketin luomisen jälkeen voidaan aloittaa klusterin valmistelu. Ensimmäiseksi tehdään luonnos klusterista seuraavalla komennolla:

```
kops create cluster \
  --zones us-west-2a \
  --cloud=aws \
  Example.com
```

Tämän jälkeen ”syötetään” klusterin luonnos Terraformille seuraavalla komennolla:

```
kops update cluster \
  --target=terraform
  Example.com
```

Tämän jälkeen tekstieditorilla voi muokata klusterin haluamansa mukaiseksi. Muokkauksen jälkeen ajetaan komento joka luo suunnitelman klusterille ja tallentaa sen tässä esimerkissä nimellä example.tf:

terraform plan -out=example.tf

Tämän jälkeen muokatusta suunnitelmasta tehdään klusteri AWS-palveluun seuraavalla komennolla:

terraform apply example.tf
(Kops docs n.d.)

Nyt kun klusteri on saatu pystyyn AWS-palveluun voidaan sen toiminta vielä testata Kubernetesilla komennolla:
kubectl get nodes
Tämän komennon pitäisi löytää noodit ja ulostulon pitäisi näyttää seuraavalta:

```
NAME           LABELS                                    STATUS
10.240.99.26   kubernetes.io/hostname=10.240.99.26       Ready
127.0.0.1      kubernetes.io/hostname=127.0.0.1          Ready
```

Kuva 8. Noodit haettuna (Testing kubernetes cluster n.d)

3.3.6 Autoskaalaus

Klusteri autoskaalaus tapahtuu cluster-autoscalerkonttia käyttäen. Kontin tehtävä on säätä klusterin kokoa kun:

- jollain podilla ei ole tarpeeksi tilaa klusterissa
- jotkin noodit klusterissa ovat alikäytettyjä niin pitkän aikaa, että ne voidaan poistaa ja niiden podit voidaan laittaa joillekin muille noodeille

Kuvassa 9. on Kubernetes Deployment, jolla otetaan cluster-autoscaler kontti käyttöön. Tämä tapahtuu komennolla:
kubectl create –f cluster-autoscaler.yaml
(Cluster Autoscaling n.d.)

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    app: cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    spec:
      containers:
        - image: gcr.io/google_containers/cluster-autoscaler:v0.4.0
          name: cluster-autoscaler
          resources:
            limits:
              cpu: 100m
              memory: 300Mi
            requests:
              cpu: 100m
              memory: 300Mi
          command:
            - ./cluster-autoscaler
            - --v=4
            - --cloud-provider=aws
            - --skip-nodes-with-local-storage=false
            - --nodes=1:10:k8s-worker-asg-1
          env:
            - name: AWS_REGION
              value: us-east-1
          volumeMounts:
            - name: ssl-certs
              mountPath: /etc/ssl/certs/ca-certificates.crt
              readOnly: true
          imagePullPolicy: "Always"
      volumes:
        - name: ssl-certs
          hostPath:
            path: "/etc/ssl/certs/ca-certificates.crt"
```

Kuva 9. cluster-autoscaler deployment (Autoscaling readme n.d.)

3.3.7 Jenkins-master volumen konfigurointi

Seuraavaksi olisi tarkoitus määrittää Jenkins-master volume. Koska kontti-maailmassa data, joka tallennetaan konttiin katoaa jos kontti uudelleen-käynnistetään täytyy Jenkins-master-kontti ”sitoa” AWS PersistentVolumeen. Koska Jenkinsin virallisessa kontissa kaikki tärkeä data tallennetaan

hakemistoon /var/jenkins_home, niin juuri tämä kansio täytyy sitoa PersistentVolumeen. Tällöin Jenkins-master-kontin uudelleenkäynnistyessä kaikki tallennettu data haetaan PersistentVolumesta. (Jenkins docker n.d.)

Luodaan uusi volume Jenkins-masterin /var/jenkins_home kansiolle komennolla:
aws ec2 create-volume --region eu-west-1a --availability-zone eu-west-1a --size 30 --volume-type gp2
Tämä komento luo uuden volumen AWS eu west 1a alueelle kooltaan 30 gigatavua ja tyypiltään gp2(General Purpose SSD).

Seuravaksi kerrotaan Kubernetesille missä kyseinen volume sijaitsee. Se tapahtuu seuraavalla komennolla jenkins-pv.yaml tiedostoa käyttäen:
kubectl create –f jenkins-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: jenkins-data
spec:
  capacity:
  storage: 30Gi
  accessModes:
  - ReadWriteOnce
  awsElasticBlockStore:
  volumeID: aws://eu-west-1a/vol-XXXXXX
  fsType: ext4
```

Kuva 10. jenkins-pv.yaml tiedosto (Running jenkins on kubernetes 2016.)

Seuraavaksi tehdään claim jossa vaaditaan käyttöön koko 30 gigatavua muistia. Se tapahtuu seuraavalla komennolla jenkins-pvc.yaml tiedostoa käyttäen:
kubectl create –f jenkins-pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: jenkins-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
  requests:
  storage: 30Gi
```

Kuva 11. jenkins-pvc.yaml tiedosto (Running jenkins on kubernetes 2016)

3.3.8 Jenkins-master määrittely

Jenkins konfigurointi tapahtuu niin, että kopioidaan /var/jenkins_home kansion sisältö valmiiksi konfiguroidusta Jenkins docker-kontista AWS S3 buckettiin. Tämän jälkeen Kubernetes deploymenttia käyttäen voidaan aws-cli konttilla hakea konfiguraatio ja sitoa se PersistentVolumeen.
Konfiguraation haku tapahtuu seuraavalla komennolla jenkins-population.yaml tiedostoa käyttäen:
kubectl create –f jenkins-population.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: population
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: population
    spec:
      containers:
      - name: population
        image: fstab/aws-cli
        command: ["/bin/bash"]
        args: ["-c", "echo 'Checking jenkins_home'; if [ ! -f
        /var/jenkins_home/initdone.txt ]; then echo 'Initializing
        jenkins_home'; /home/aws/aws/env/bin/aws s3 cp
        s3://example/jenkins_home.tgz /var/jenkins_home/ --region
        eu-west-1a; cat /var/jenkins_home/jenkins_home.tgz; cd
        /var/jenkins_home/; tar zxvf jenkins_home.tgz; fi"]
        volumeMounts:
        - mountPath: /var/jenkins_home
          name: jenkinshome
      securityContext:
        fsGroup: 0
      volumes:
      - name: jenkinshome
        persistentVolumeClaim:
          claimName:jenkins-data
```

Kuva 12. jenkins-population.yaml

Nyt kun /var/jenkins_home-kansio on sidottu volumeen voidaan käynnistää Jenkins-master-kontti. Se tapahtuu seuraavalla komennolla jenkins-.yaml tiedostoa käyttäen:
kubectl create -f jenkins-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 annotations:
 labels:
 app: jenkins
 name: jenkins
spec:
 replicas: 1
 selector:
 matchLabels:
 app: jenkins
 template:
 metadata:
 labels:
 app: jenkins
 spec:
 containers:
 - image: jenkins:2.19.2
 imagePullPolicy: IfNotPresent
 name: jenkins
 ports:
 - containerPort: 8080
 protocol: TCP
 name: web
 - containerPort: 50000
 protocol: TCP
 name: slaves
 resources:
 limits:
 cpu: 500m
 memory: 1000Mi
 requests:
 cpu: 500m
 memory: 1000Mi
 volumeMounts:
 - mountPath: /var/jenkins_home
 name: jenkinshome
 securityContext:
 fsGroup: 1000
 volumes:
 - name: jenkinshome
 persistentVolumeClaim:
 claimName: jenkins-data
```

Kuva 13. jenkins-deployment.yaml (Running jenkins on kubernetes 2016.)

Tällä yaml-tiedostolla kiinnitämme Jenkins-kontin aiemmin tehtyä PersistentVolumeClaimia käyttäen volumeen. Claimin nimi tässä tapauksessa on jenkins-data.

3.3.9 Jenkins-slave määrittely

Nyt kun Jenkins-master on toiminnassa on aika määritellä Jenkins-slavejen volume tarpeet. Määrittely tapahtuu seuraavalla komennolla sc_jenkins_slave.yaml tiedostoa käyttäen:

kubectl create -f sc_jenkins_slave.yaml

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: j_slave
  labels:
    application: "jenkins_slave"
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  zone: eu-west-1a
  encrypted: "false"
```

Kuva 14. sc_jenkins_slave.yaml

Tällä Storage Class määrittelyllä saamme volumen luotua silloin kun slave-kontissa ajetaan testejä. Nyt kun käyttäjä haluaa ajaa testit Jenkins-slave-kontissa, saadaan käyttäjälle tarvittava volume pystyyn ajon ajaksi.

Seuraavaksi täytyy tehdä PersistentVolumeClaim, joka sidotaan edellä tehtyyn Storage Class määrittelyyn. PersistentVolumeClaim tehdään seuraavalla komennolla pvc_jenkins_slave.yaml tiedostoa käyttäen:

kubectl create -f pvc_jenkins_slave.yaml

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: jenkins
  labels:
    application: "jenkins"
  annotations:
    volume.beta.kubernetes.io/storage-class: "j_slave"
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Kuva 15. pvc_jenkins_slave.yaml

Tällä määrittelyllä sidotaan PersistentVolumeClaim Storage Classiin ja kerrotaan, että haluttu tallennustila on 5 gigatavua.

Klusteri on nyt valmis Jenkinsille. Nyt voidaan siirtyä käyttämään Jenkinssiä seuraavalla komennolla:
kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT
(Port forward)

Nyt pääsemme käsiksi Jenkinsin käyttöliittymään ja voimme käynnistää slave kontteja Jenkinssistä.

4 JOHTOPÄÄTÖKSET

Ensimmäisenä huomiona on AWS helppokäyttöisyys. AWS selainkäyttöliittymä on selkeä ja johdonmukainen. Dokumentointi ja ohjeistus on hyvällä mallilla Amazonissa, joka helpotti työskentelyä pilvessä. Tutoriaalit ovat hyvä lisä alkuun pääsemiseksi ja opetusvideot helpottavat ymmärtämistä AWS:stä. Ainoana huonona puolena on oikeuksien jaon vaikeus. Tarvittavien oikeuksien antaminen osoittautui hankalaksi, koska vaikeutena oli selvittää mitä oikeuksia Kubernetes klusterin eri osat tarvitsevat toimiakseen. AWS CLI toimii hyvin yhteen Kubernetes-ohjelman kanssa jolloin pilvessä käytettävän klusterin konttien hallinta onnistuu helposti. Kopsilla pystyy taas pystyttämään klustereita ja hallinnoinmaan niitä, kunhan oikeudet sai kuntoon. Kubernetes-ohjelmassa on myös hyvät työkalut vikatilojen löytämiseen ja korjaamiseen ja Kubernetes-dokumentointi oli myös kattavaa. Tästä huolimatta Kubernetes tarvitsee vielä lisätutkimusta, sillä osa kokeilussa käytetyistä ratkaisuista oli vielä beta-vaiheessa tai kokeilu-vaiheessa. Lisäksi muitakin ratkaisuja on hyvä kokeilla, jotta saadaan vertailukohta Kubernetes-ohjelmaan.

Kaiken kaikkiaan kokeilusta saatiin tarvittavat tulokset, jotta päästään suunnittelemaan ja toteuttamaan seuraavaa vaihetta. Pilvipohjainen ratkaisu tuo lisäarvoa testaukseen ja Amazonin laskentakapasiteetti nopeuttaisi testausta. Konttimaailma tuo nopeutta ja skaalautuvuutta testaukseen, koska konttien pystyttäminen ja alasajo tapahtuu nopeasti. Lisäksi Kubernetes-ohjelmaa voi käyttää sekä pilvessä, että tarvittaessa yrityksen palvelimilla.

LÄHTEET

Autoscaling readme Haettu 10.12.2016 osoitteesta https://github.com/kubernetes/contrib/blob/master/cluster-autoscaler/cloudprovider/aws/README.md

AWS CLI getting started Haettu 10.12.2016 osoitteesta http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html

AWS EBS volumes Haettu 31.1.2017 osoitteesta http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumes.html

AWS ec2 Haettu 31.1.2017 osoitteesta https://aws.amazon.com/ec2

AWS getting started Haettu 31.1.2017 osoitteesta http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-intro.html

AWS VPC Haettu 31.1.2017 osoitteesta https://aws.amazon.com/vpc/

Baier, J. (2015). Getting Started with Kubernetes.

Chandrasekaran, K. (2014). Essentials of Cloud Computing.

Cluster Autoscaling Haettu 10.12.2016 osoitteesta https://github.com/kubernetes/contrib/tree/master/cluster-autoscaler

Craig, R. Stefan, P. & Jaskiel (2002). Systematic Software Testing.

Dooley, J. (2011). Software Development and Professional Practice.

Dustin, E., Garrett, T. & Gauf, B. (2009). Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality.

Ferguson Smart, J. (2011). Jenkins: The Definitive Guide Continuous integration for the masses John Ferguson Smart.

Httermann, M. (2012). DevOps for Developers.

Jenkins docker Haettu 10.12.2016 osoitteesta https://hub.docker.com/_/jenkins/

Kops docs Haettu 10.12.2016 osoitteesta https://github.com/kubernetes/kops/blob/master/docs/aws.md

Kubernetes Container Orchestration Haettu 10.12.2016 osoitteesta http://kubernetes.io/

Kubernetes Deployments Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/user-guide/deployments/

Kubernetes getting started Haettu 10.12.2016 osoitteesta http://kubernetes.io/docs/getting-started-guides/kops/

Kubernetes node Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/admin/node/

Kubernetes Overview Haettu 10.12.2016 osoitteesta http://kubernetes.io/docs/tutorials/kubernetes-basics/

Kubernetes persistent volumes Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/user-guide/persistent-volumes

Kubernetes pods Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/user-guide/pods/

Kubernetes replication controller Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/user-guide/replication-controller/

Kubernetes Volumes Haettu 10.12.2016 osoitteesta https://kubernetes.io/docs/user-guide/volumes/

Li, K. & Wu, M. (2006). Effective Software Test Automation: Developing an Automated Software Testing Tool.

Port forward Haettu 31.1.2017 https://kubernetes.io/docs/user-guide/kubectl/kubectl_port-forward/

Robot framework user guide Haettu 31.1.2017 osoitteesta http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Route 53 home Haettu 10.12.2016 osoitteesta https://console.aws.amazon.com/route53/home

Running jenkins on kubernetes (2016) Haettu 31.1.2017 osoitteesta https://wealthwizards.io/2016/11/25/running-jenkins-on-kubernetes/

Softwaretestinghelp types of software testing Haettu 31.1.2017 osoitteesta http://www.softwaretestinghelp.com/types-of-software-testing

Terraform intro Haettu 10.12.2016 osoitteesta https://www.terraform.io/intro/index.html

Testing kubernetes cluster Haettu 10.12.2016 osoitteesta http://janet-kuo.github.io/kubernetes/v1.0/docs/getting-started-guides/docker-multinode/testing.html)

Hammell, T., Gold, D. & Snyder T. (2007). Test-Driven Development: A J2EE Example.

Tutorialspoint Waterfall Modell Haettu 31.1.2017 osoitteesta http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

What is docker Haettu 31.1.2017 osoitteesta https://www.docker.com/what-docker

What is Kops Haettu 10.12.2016 osoitteesta https://github.com/kubernetes/kops

Woodward, E., Steffan, S. & Ganis, M. (2010). A Practical Guide to Distributed Scrum.