Tampereen yliopisto

Antti Pessa

MIKROPALVELUIDEN RAKENTAMINEN DOCKERILLA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Helmikuu 2021

TIIVISTELMÄ

Antti Pessa: Mikropalveluiden rakentaminen Dockerilla
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Helmikuu 2021

Tutkielma tarkastelee Docker-teknologiaa ja sen käyttöä mikropalveluarkkitehtuurissa kirjallisuuskatsauksen avulla. Virtualisoinnilla voidaan tehdä virtuaalinen versio ohjelmasta tai laitteesta. Tutkielma käy läpi kaksi virtualisointitapaa: virtuaalikone ja konttiteknologia. Docker on konttiteknologiaa hyödyntävä virtualisointiratkaisu, jonka avulla sovellukset saadaan pakattua ja ajettua eristetyissä konteissa. Tutkielma esittelee Dockerin keskeiset ominaisuudet ja miten ne soveltuvat mikropalveluarkkitehtuurin käyttöönottoon.

Mikropalveluarkkitehtuurissa ohjelman toiminnallisuudet jaetaan pieniin itsenäisiin palveluihin. Palveluita voidaan kehittää pienissä tiimeissä ja ne voidaan julkaista nopeasti tuotantoon. Mikropalveluarkkitehtuuri kasvattaa kuitenkin järjestelmän kompleksisuutta ja tekee testaamisesta monimutkaista. Tutkielmassa käydään läpi case-esimerkki mikropalveluarkkitehtuuria ja Dockeria käyttävästä järjestelmästä. Esimerkissä on haastateltu kehitystiimin jäseniä ja kerrottu heidän kokemuksistaan. Dockerin avulla sovelluksen eri komponentit voidaan päivittää ja pystyttää nopeasti. Mikropalveluarkkitehtuuri on mahdollistanut kehittäjille vapauden valita haluamansa teknologiat.

Tutkielma osoittaa, että Docker on avainasemassa mikropalveluiden käyttöönotossa ja DevOps-toimintamallin toteuttamisessa. Se on virtuaalikoneita kevyempi, nopeampi ja tehokkaampi. Yksinkertaisen käyttöliittymän ja siirrettävyyden ansiosta Dockerin suosio ohjelmistokehittäjien keskuudessa kasvaa. Docker helpottaa pilvipalveluihin siirtymistä ja mahdollistaa konttien skaalaamisen työkuormituksen mukaan. Mikropalveluarkkitehtuuri tuo mukanaan parannuksia ohjelmistokehitykseen modulaarisuuden, skaalautuvuuden ja nopean käyttöönoton avulla. Se ei kuitenkaan ole aina oikea ratkaisu ja vaatii ohjelmistokehitystiimiltä tarkkaa suunnittelua ja koordinaatiota sen oikeaoppiseen toteuttamiseen.

Avainsanat: Docker, virtualisointi, mikropalveluarkkitehtuuri, konttiteknologia

Sisällysluettelo

1 Johdanto

Ohjelmistokehitys edellyttää toimintatapojen ja teknologioiden jatkuvaa kehittämistä. Yritykset siirtävät palveluitaan pilveen ja se on johtanut konttiteknologian ja mikropalveluiden käyttöönottoon (deployment). Niiden yhdistelmä tuo parannusta ohjelmistokehityksen skaalautuvuuteen ja joustavuuteen (Wan et al., 2018). Suunnannäyttäjiä tälle kehitykselle ovat olleet suuret teknologiayhtiöt, kuten Netflix (2020), Amazon (2020b) ja Uber (2020). Esimerkiksi Netflix saa noin miljardi suoratoistopyyntöä joka päivä, jotka reititetään ohjelmointirajapinnan kautta usealle eri taustamikropalvelulle (Hassan et al., 2017). Jokaisen mikropalvelun tarkoitus on suorittaa yksi tehtävä. Docker taas mahdollistaa eristetyn ympäristön sovelluksien ajamiselle ja nopean tavan niiden käyttöönottoon.

Tässä työssä pyrin kirjallisuuskatsauksen kautta käymään läpi virtualisointia yleisesti, ja sitä kuinka konttiteknologia eroaa virtuaalikoneesta. Vastaan kysymyksiin mikä on Docker, mitä se sisältää, mitä mikropalvelut ovat ja kuinka Docker on vaikuttanut mikropalveluarkkitehtuuriin. Lisäksi esittelen syitä konttiteknologian suosion kasvuun. Tutustun mikropalveluarkkitehtuurin laatukriteereihin ja taktiikoihin toteuttaa niitä. Esittelen case-esimerkissä matkapuhelinverkon analysointiohjelmiston. Se on rakennettu mikropalveluarkkitehtuurin mukaisesti ja siinä käytetään Dockeria palveluiden ajamiseen ja käyttöönottoon.

Lähteitä on haettu Google Scholarista ja Andor-tietokannoista. Hakusanoina on käytetty eri yhdistelmiä näistä termeistä: ”microservices”, ”Docker”, ”container”, ”DevOps”, ”microservice architecture”, ”virtualization”, ”continuous integration”. Suuri osa lähteistä on tieteellisiä julkaisuja, joskin myös Dockerin virallista dokumentaatiota on käytetty. Koska aihealue on uusi ja jatkuvasti kehittyvä, suurin osa lähteistä on vuosilta 2016–2020. Lähteet ovat englanninkielisiä ja termit ovat kirjoittajan suomentamia.

Luvussa 2 käsitellään yleisesti virtualisointia, virtuaalikoneita ja konttiteknologiaa. Sitten luvussa 3 tutustutaan Dockeriin ja sen pääominaisuuksiin. Se paketoi koodin, riippuvuudet ja kirjastot konttiin, jota voidaan ajaa eri ympäristöissä. Luvussa 4 tutustumme mikropalveluarkkitehtuuriin ja kuinka Docker on vaikuttanut sen kasvuun. Luvussa 5 käymme läpi case-esimerkin sovelluksesta, joka on rakennettu mikropalveluiden ympärille ja käyttää siinä apuna Dockeria. Lopuksi luvussa 6 on yhteenveto.

2 Virtualisointi

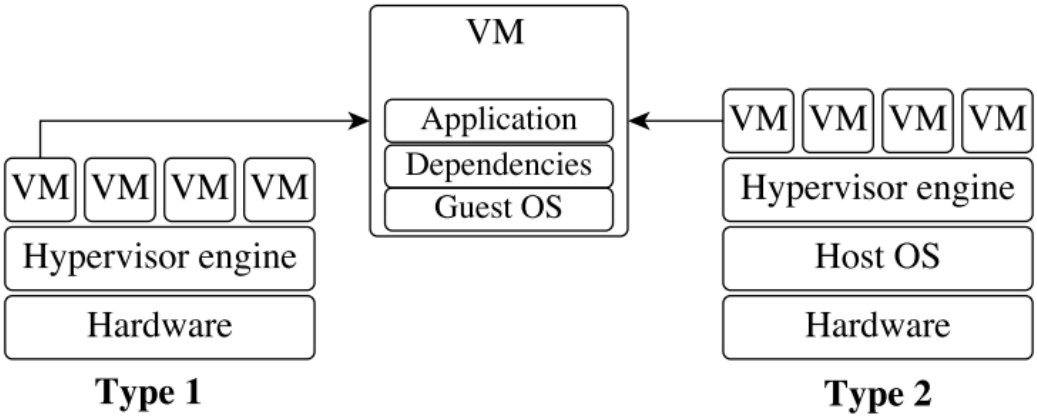
Virtualisoinnilla voidaan tehdä virtuaalinen versio esimerkiksi ohjelmistosta tai laitteesta. Sen avulla tietokoneen resurssit voidaan jakaa eri käyttöjärjestelmien välille. Virtualisointi sai alkunsa 1960-luvulla IBM:än kehittämällä menetelmällä, jolla voitiin jakaa resursseja keskustietokoneesta eri ohjelmien välille. Hypervisor antoi jokaiselle keskustietokoneen käyttäjälle oman osoituskäyttöjärjestelmän (conversational monitor system) (Brodkin, 2009). Nykyään virtualisointia käytetään yritysten IT-infrastruktuurien skaalaamisessa, turvaamisessa ja hallinnoinnissa.

Keskityn tässä työssä koneiden virtualisointiin ja käyn läpi kahta tapaa suorittaa koneiden virtualisointia: hypervisor-pohjainen laitteistotason virtualisointi (hardware virtualization) ja konttiteknologia eli käyttöjärjestelmätason virtualisointi (OS-level virtualization).

2.1 Hypervisor-pohjainen virtualisointi

Hypervisor-pohjaista virtualisointia on käytetty laajasti kuluneen vuosikymmenen ajan sovellusten virtualisointiin ja eristämiseen. Hypervisor on sovellus, laiteohjelmisto tai laitteisto, joka luo ja pyörittää virtuaalikoneita. Virtuaalikone on emulaatio tietokonejärjestelmästä. Se perustuu tietokonearkkitehtuuriin ja tarjoaa fyysisen tietokoneen toimintoja. On olemassa kahdenlaisia virtuaalikoneita: täyden virtualisoinnin virtuaalikoneita ja prosessivirtuaalikoneita. Täyden virtualisoinnin virtuaalikone tarjoaa vastikkeen oikealle koneelle ja käyttöjärjestelmälle, kun prosessivirtuaalikone on tarkoitettu tietokoneohjelmien ajamiseen alustasta riippumattomassa ympäristössä.

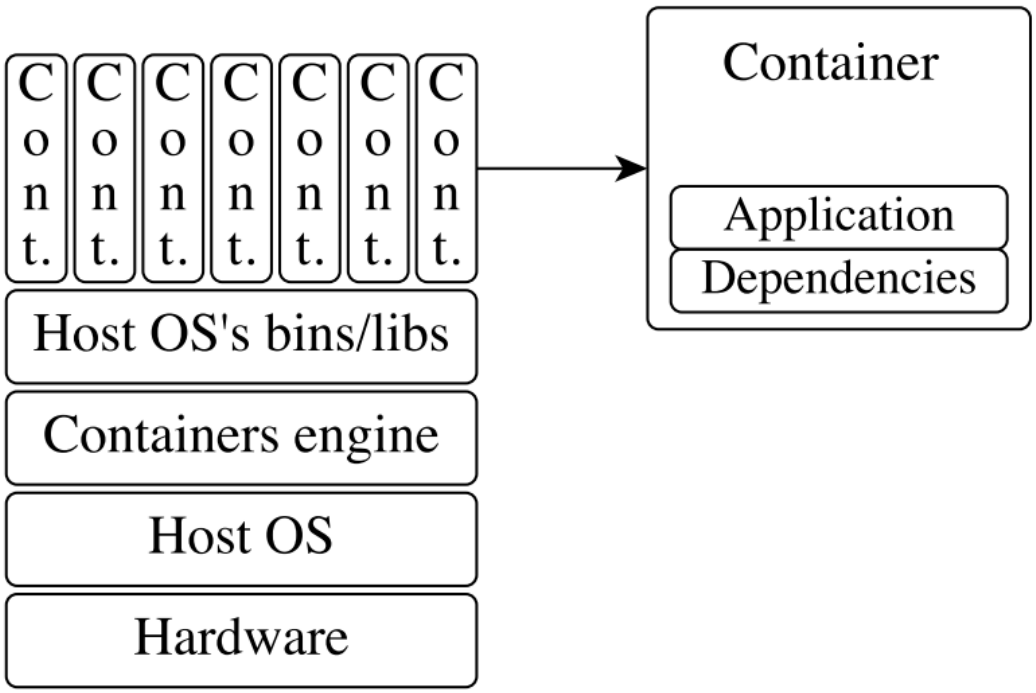
Hypervisorit luokitellaan kahteen tyyppiin: tyyppi 1 on natiivi hypervisor, joka pyörii suoraan laitteiston päällä ja tyyppi 2, joka pyörii koneen käyttöjärjestelmän päällä (Martin et al., 2018; Morabito et al., 2015). Kuvassa 1 kuvataan näitä kahta hypervisortyyppiä. Virtuaalikoneet sisältävät käyttöjärjestelmän, sovelluksen ja riippuvuudet (dependencies) sen ajamiseen. Koneen omaa käyttöjärjestelmää kutsutaan isäntäkäyttöjärjestelmäksi (host OS) ja virtuaalikoneen vieraskäyttöjärjestelmäksi (guest OS). Koska tyyppi 1 hypervisorissa ei ole isäntäkäyttöjärjestelmää, on sen suorituskyky parempi (Bui, 2014). Kahden käyttöjärjestelmän sekä virtuaalisen laitteistokerroksen takia yleiskustannukset suorituskyvyssä ovat korkeat. Microsoftin (2020b) Hyper-V ja avoimen lähdekoodin Xen (2020) ovat esimerkkejä natiiveista hypervisoreista. Oraclen (2020) VirtualBox on sovellus, jolla voi luoda mistä vain käyttöjärjestelmästä virtuaalikoneen.



Kuva 1. Hypervisor-pohjainen virtualisointi (Bui, 2014).

2.2 Konttiteknologia

Konttiteknologia (container virtualization) on kevyt virtualisointiratkaisu, joka käyttää isäntäkäyttöjärjestelmän ydintä (kernel) ja kirjastoja usean virtuaaliympäristön ajamiseen (Bui, 2014; Shu et al., 2017). Käyttöjärjestelmän ydin on ohjelma, joka kontrolloi kaikkea mitä käyttöjärjestelmässä tapahtuu. Kuvassa 2 näkyy konttiteknologian yleinen arkkitehtuuri. Virtualisointi tapahtuu käyttöjärjestelmätasolla ja tämä mahdollistaa useamman sovelluksen ajamisen ilman että tarvitsee turhaan useamman käyttöjärjestelmän ydintä isäntäkäyttöjärjestelmässä (Bui, 2014).



Kuva 2. Konttiteknologia (Bui, 2014).

Nimi kontti tulee rahtilaivojen konteista ja sillä kuvataan sitä, kuinka sovellukset eristetään toisistaan ja ajetaan standardin mukaisesti. Kontit näyttävät normaaleilta prosesseilta, jotka pyörivät isäntäjärjestelmän ytimen päällä. Ne mahdollistavat eristetyn ympäristön ohjelman suorittamiselle tarvittavilla resursseilla. Resurssit voidaan jakaa isäntäjärjestelmän kanssa tai asentaa erikseen kontin sisälle. (Bui, 2014)

Konttiteknologiasta on monta erilaista toteutusta, yksi tärkeimmistä on LXC eli LinuX Containers, joka käyttää Linuxin ytimen kontrolliryhmiä (cgroups) ja nimiavaruuksia (namespace) (Martin et al., 2018). Kontrolliryhmät hallinnoivat sitä kuinka paljon järjestelmän resursseja kukin prosessi käyttää. Nimiavaruuksien avulla tietokoneen resursseja voidaan jakaa prosesseille toisiltaan näkymättömiin alueisiin. Prosessilla tarkoitetaan tietokoneessa ajossa olevaa ohjelmaa.

Oman ytimen puuttuminen ja jaetut kirjastot tekevät konteista hyvin kevyitä ja nopeita käynnistää (Martin et al., 2018). Nopea käynnistys mahdollistaa konttien luomisen tarpeen vaatiessa tai palvelun siirtämisen nopeasti. Konttiteknologian avulla voidaan luoda kymmenen kertaa enemmän virtuaaliympäristöjä hypervisor-pohjaiseen virtualisointiin verrattuna (Bui, 2014). Virtuaalikoneiden oman käyttöjärjestelmän tuoma lisäkerros lisää turvallisuutta, mutta toisaalta se vähentää suorituskykyä. Bui (2014) huomauttaa, että kontit eivät pysty tukemaan yhtä laajaa määrää erilaisia ympäristöjä kuin virtuaalikoneet, koska konttien ympäristöt tulee olla samaa tyyppiä kuin isäntäkäyttöjärjestelmän. Esimerkiksi Windows-konttia ei pysty ajamaan Linux ympäristössä.

3 Docker

Ohjelmistokehityksessä on tärkeää, että kehitetty ohjelma toimii samalla tavalla kaikissa ympäristöissä eikä vain kehittäjän omassa ympäristössä. Eri kehittäjillä voi olla hyvin erilaiset ympäristöt ja asetukset puhumattakaan loppukäyttäjästä. Tähän ongelmaan on ratkaisu ja se on Docker. Docker on avoimen lähdekoodin konttiteknologia, joka mahdollistaa ohjelmien rakentamisen, jakamisen ja ajamisen (Bui, 2014). Se paketoi ohjelman, sen riippuvuudet ja kirjastot konttiin, joka voidaan jakaa yhtenä pakettina. Kehittäjän ei tarvitse enää murehtia toimiiko ohjelma asiakkaalla tai mitä riippuvuuksia hän asensi ympäristöönsä. Docker julkaistiin maaliskuussa 2013, ja sitä kehittää yritys nimeltä Docker Inc. Alun perin Docker käytti LXC-pohjaista konttiteknologiaa, mutta siirtyi vuonna 2014 omaan Go-kielellä kirjoitettuun libcontainer-kirjastoon. Libcontainerin kautta Docker pystyy suoraan hallinnoimaan Linuxin nimiavaruuksia ja kontrolliryhmiä ilman että sen tarvitsee riippua LXC:stä. (Hykes, 2014)

Docker kostuu kahdesta pääkomponentista: Docker-moottorista (Docker Engine) ja Docker Hubista. Moottori on virtualisoinnin mahdollistava ratkaisu ja Hub on alusta

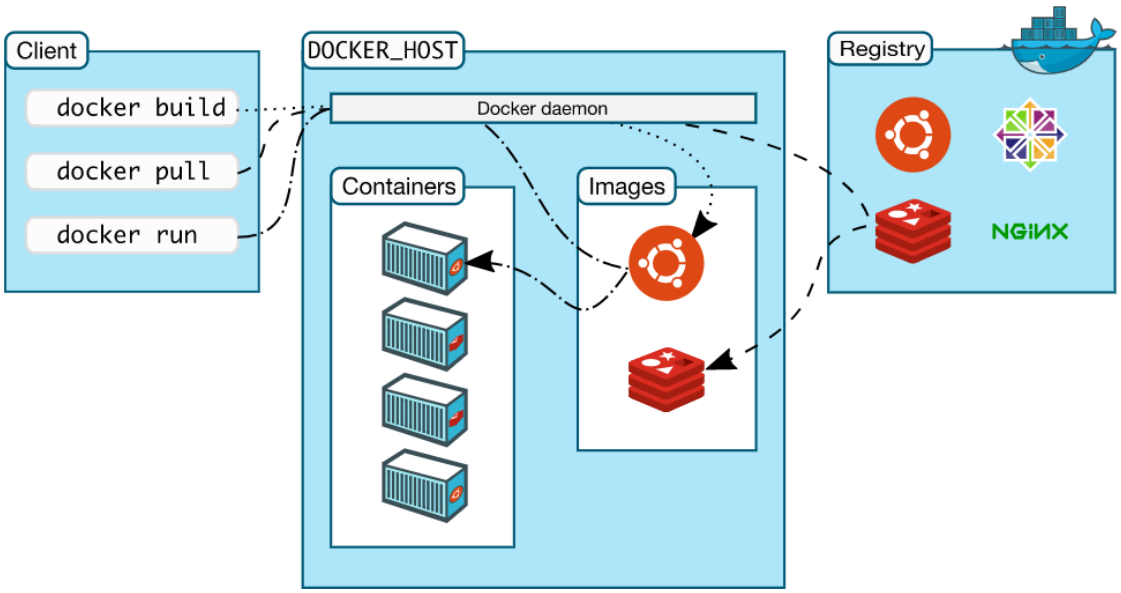
Docker-kuvien (Docker image) jakamiselle. Tällä hetkellä Docker on suosituin konttiteknologia, Docker Hubista on ladattu 29 miljardia konttia ja siellä on yhteensä 900 000 ”dockeroitua” sovellusta (Martin et al., 2018).

Bui (2014) tuo esiin syitä miksi juuri Docker on noussut kehittäjien suosioon eri konttiteknologioiden joukosta. Ensinnäkin Docker tarjoaa yksinkertaisia ja turvallisia käyttöliittymiä konttien luomiseen ja hallinnointiin. Toiseksi, ilman mitään muutoksia kehittäjät voivat pakata sovelluksensa kevyihin kontteihin, jotka voivat toimia melkein missä vain. Docker toimii hyvin yhteen muiden kolmannen osapuolen työkalujen kanssa, jotka helpottavat konttien hallintaa ja käyttöönottoa. Myös konttien orkestrointijärjestelmät, kuten Kubernetes (2020), Mesos (2020) ja Shipyard (2020) tukevat Dockeria. Orkestrointijärjestelmät automatisoivat monta kontteihin liittyvää manuaalista tehtävää, kuten niiden käyttöönotto, hallinnointi ja skaalaaminen.

3.1 Docker-moottori

Moottori on asiakas-palvelinsovellus (client-server application) ja se on Dockerin ydin. Siihen sisältyy Docker-taustaprosessi (Docker daemon), REST-ohjelmointirajapinta ja komentoliittymä (Command-line interface) (Docker, 2020). Taustaprosessi on vastuussa konttien suorittamisesta ja hallitsemisesta. Taustaprosessi pyörii root-oikeuksilla isäntäkäyttöjärjestelmässä (Martin et al., 2018). Komentoliittymän kautta käyttäjät voivat hallita kontteja komennoilla. Komennot lähetetään taustaprosessille REST-ohjelmointirajapinnan kautta. Rajapinta mahdollistaa komentojen suorittamisen isäntäkäyttöjärjestelmästä tai sitten ulkopuoliselta palvelimelta (Docker, 2020).

Kuvassa 3 näkyy miten asiakas-palvelinsovellus toimii. Käyttäjä antaa komentoja komentoliittymän avulla, jotka kulkevat REST-rajapinnan kautta taustaprosessille. Taustaprosessi voi ladata kuvan Docker-rekisteristä ja luoda siitä kontin. Komentoja moottorille annetaan docker-sanan avulla. Run-komento käynnistää kontin kun taas build-komento luo Docker-kuvan.



Kuva 3. Dockerin asiakas-palvelinarkkitehtuuri (Docker, 2020).

3.2 Docker-kuva

Kontit rakentuvat Docker-kuvista. Ne sisältävät ohjeet sille mitä kirjastoja, riippuvuuksia ja asetuksia kontti tarvitsee. Kuvat rakennetaan pohjakuvan (base image) päälle, joka on yleensä jokin Linux-pohjainen käyttöjärjestelmä. Pohjakuvaan tehdyt muutokset kasautuvat kerroksittain, jokainen kerros suhteutetaan edelliseen muutokseen. Docker käyttää UnionFS-tiedostojärjestelmää, jonka avulla usean tiedostojärjestelmän tiedot ja hakemistot voidaan yhdistää yhdeksi tiedostojärjestelmäksi (Docker, 2020). Esimerkiksi jos asentaa tietokannan Ubuntu-pohjakuvaan, Docker luo tietokerroksen, joka sisältää tietokannan ja lisää sen kuvaan. Tämä tekee kuvien jakeluprosessista tehokkaampaa, koska vain tehty päivitys pitää jakaa (Bui, 2014). Jos useampi kuva isäntäjärjestelmässä perustuu samaan pohjakuvaan tai samoihin riippuvuuksiin, haetaan päivitykset vain kerran ohjelmavarastoista (code repository) (Martin et al., 2018).

Kuva rakennetaan Dockerfile-nimisen tiedoston mukaan. Tiedosto sisältää kaikki ohjeet kontin koostumuksesta. Docker luo kerrokset tiedostossa esitetyssä järjestyksessä, jokainen uusi komento luo uuden kerroksen. Ylintä kerrosta kutsutaan konttikerrokseksi (container layer). Se on ohut kerros, jolla on luku- ja kirjoitusoikeus. Kuvasta 4 näkyy, kuinka Docker-kuva rakentuu Dockerfilestä. Pohjakuvana on Ubuntu 18.04 ja kuvan app-hakemistoon kopioidaan tiedostot, jotka rakennetaan Linuxin make-komennolla. CMD-komento määrittelee kontille käynnistysparametrit.

Kuva 4. Dockerfile ja sen luomat kerrokset Docker-kuvassa.

Boettiger (2015) nostaa esiin Dockerfilen tuomia hyötyjä. Pienestä tekstitiedostosta on versionhallinnan kautta helppo seurata kuvaan tehtyjä muutoksia. Tekstitiedosto on helppo säilyttää ja jakaa, verrattuna virtuaalikoneiden kuviin, jotka ovat monen gigatavun kokoisia. Tiedosto kuvaa tarvittavat riippuvuudet ja ympäristömuuttajat ihmiselle helposti luettavassa formaatissa. Näin vältytään virheiltä, joita riippuvuuksien manuaalinen asentaminen voisi aiheuttaa. Riippuvuuksia ei tarvitse dokumentoida projektin lopussa, koska riippuvuudet on dokumentoitu suoraan Dockerfileen.

3.3 Docker-rekisteri

Docker-rekisteri on keskitetty säilytyspaikka Docker-kuville. Se on myös keskeisessä asemassa kuvien jakamisessa. Rekisteri toimii samalla tavalla, kuin Git-versionhallintasivustot Github (2020) ja Gitlab (2020). Rekisteriin voi luoda kontille varaston (repository), joka sisältää kaikki kontista tehdyt kuvat. Varastot voivat olla yksityisiä tai julkisia. Kuva merkitään Dockerin tag-ominaisuuden avulla. Merkitseminen on tärkeää, koska ilman sitä kuvia on vaikea erottaa toisistaan. Kun kuva on rakennettu, sen voi Dockerin push-komennolla siirtää varastoon. Sieltä voidaan Dockerin pull-komennolla ladata mikä tahansa kuva, kunhan siihen varastoon on oikeudet. Kun kuva pusketaan varastoon, sen kerrokset pakataan ja arkistoidaan tar-tiedostoksi, jonka jälkeen se siirretään varastoon. Varasto luo kuvasta lastiluettelon, joka listaa kaikki kuvan kerrokset. (Zheng et al., 2018)

Docker Hub on maailman suuri Docker-rekisteri, se on Docker Inc:in tarjoama palvelu Docker-kuvien löytämiseen ja jakamiseen. Docker Hub sisältää kahden tyyppisiä julkisia varastoja: virallisia ja yhteisön tekemiä. Viralliset varastot sisältävät kuvia sertifioiduilta valmistajilta, kuten Docker, Oracle ja Red Hat. Yhteisövaraston voi tehdä kuka tahansa. (Shu et al., 2017)

3.4 Turvallisuus ja haavoittuvuudet

Virtuaalikoneita on pidetty turvallisempina kuin kontteja, koska ne lisäävät erillisen kerroksen eristystä sovelluksen ja isäntäjärjestelmän välille. Kontit taas voivat kommunikoida suoraan isäntäjärjestelmän kanssa ja tämä on herättänyt turvallisuushuolia. (Bui, 2014)

Dockerin turvallisuus pohjautuu kolmeen komponenttiin: prosessien eristäminen käyttäjätilatasolla Docker-taustaprosessin avulla, isäntäkäyttöjärjestelmän ytimen suojaaminen ja verkko-operaatioiden turvaaminen (Martin et al., 2018). Prosessien eristäminen tapahtuu Dockerissa nimiavaruuksien avulla. PID-nimiavaruudet rajoittavat konttien oikeuksia ja näkyvyyttä muihin kontteihin ja isäntäkäyttöjärjestelmään (Bui, 2014). Oletusasetuksena kontin nimiavaruudet ovat eristettynä, mutta kontrolliryhmien rajoitukset pitää erikseen aktivoida. Kontille voi antaa käynnistyksessä lisäparametreja, jotka lisäävät sen pääsyä isäntäkäyttöjärjestelmään. Docker-taustaprosessille voi antaa komennon -icc-false, joka estää konttien välisen kommunikaation, mutta sitä käytetään harvoin (Martin et al., 2018).

Martin ja muut (2018) ovat jakaneet haavoittuvuudet viiteen pääkategoriaan:

- turvattomat järjestelmäasetukset isäntäkäyttöjärjestelmässä,
- haavoittuvuus kuvien jakamisessa, verifikaatiossa ja säilömisessä,
- haavoittuvuus kuvan sisällä,
- Dockeriin liittyvä haavoittuvuus ja
- Linuxin ytimeen liittyvä haavoittuvuus.

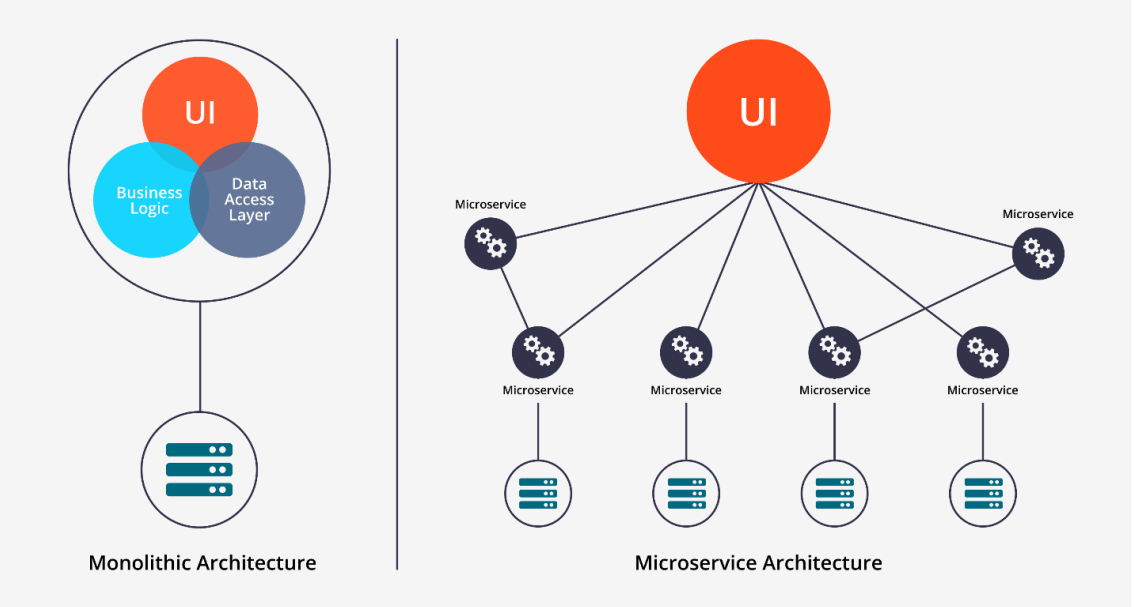
Näistä yleisin haavoittuvuus on Docker-kuvan sisällä. Kuvat voivat sisältää vanhentuneita versioita paketeista, jos kuvat perustuvat vanhaan pohjakuvaan tai kuvan rakennusvaiheessa vanha koodi on haettu ohjelmavarastosta (Martin et al., 2018). Shu ja muiden (2017) tekemän tutkimuksen mukaan Docker Hubissa olleista kuvista löytyi keskimäärin yli 180 haavoittuvuutta ja useimpia kuvia ei ollut päivitetty satoihin päiviin. Kuvien haavoittuvuuksien löytämiseen voidaan käyttää skannereita, kuten Anchore ja Clair. Tämä ei kuitenkaan ole nopea tapa: kuvan koosta riippuen, se voi kestää tunnista päiviin, eikä kaikkia haavoittuvuuksia aina löydy CVE-tietokannasta (Common Vulnerabilities and Exposures) (Martin et al., 2018).

Vaikka Dockerista on löytynyt haavoittuvuuksia, se on oletusasetuksia käyttäen hyvin turvallinen. Turvallisuutta voidaan parantaa Linuxin ytimen koventamisohjelmilla (hardening) kuten AppArmor, SELinux ja GRSEC (Bui, 2014). Ne vähentävät hyökkäyspintaa, tiukentamalla palomuuria ja rajoittamalla ohjelmien käyttöoikeuksia. Martin ja muut (2018) nostavat orkestrointijärjestelmät tavaksi hallinnoida Dockeria. Orkestrointijärjestelmät poistavat riippuvuuden isäntäkäyttöjärjestelmästä ja parantavat eristystä.

4 Mikropalveluarkkitehtuuri

Aikaisemmin useimmat ohjelmistot rakennettiin yhtenäisenä ohjelmana: käyttöliittymä, tietokanta ja palvelin olivat yhtä sovellusta. Tällaista sovellusta kutsutaan monoliittiseksi. Tämä ei enää vastaa nykyajan vaatimuksia, etenkin kun yhä suurempi osa sovelluksista on alettu julkaista pilveen (O'Connor et al., 2017). Viime vuosien aikana on kehittynyt uusi tapa suunnitella ohjelmia, mikropalveluarkkitehtuuri.

Mikropalveluarkkitehtuuri on ohjelmistoarkkitehtuuri, jossa sovellus jaetaan useaan erilliseen palveluun. Se pohjautuu kolmeen Unixin periaatteeseen: ohjelman pitäisi tehdä vain yksi tehtävä ja tehdä se hyvin, ohjelmien pitäisi pystyä toimimaan yhdessä ja ohjelmien pitäisi käyttää yhteistä rajapintaa (Cerny et al., 2018). Mikropalvelut ovat löysästi kytköksissä (loose coupling). Tämä tarkoittaa tietojenkäsittelytieteessä, että komponentit ovat irrallaan toisistaan ja voivat toimia itsenäisesti. Ne kommunikoivat toistensa kanssa REST-ohjelmointirajapinnan tai yrityspalveluväylän kautta kuten Apache Kafka tai RabbitMQ (O'Connor et al., 2017). Kuvassa 5 kuvataan kuinka eri tavalla monoliittisessa arkkitehtuurissa ja mikropalveluarkkitehtuurissa ohjelma rakentuu. Monoliitissa kaikki on rakennettu yhteen ohjelmaan, kun taas mikropalveluarkkitehtuurissa toiminnallisuudet on jaettu pieniin osiin.



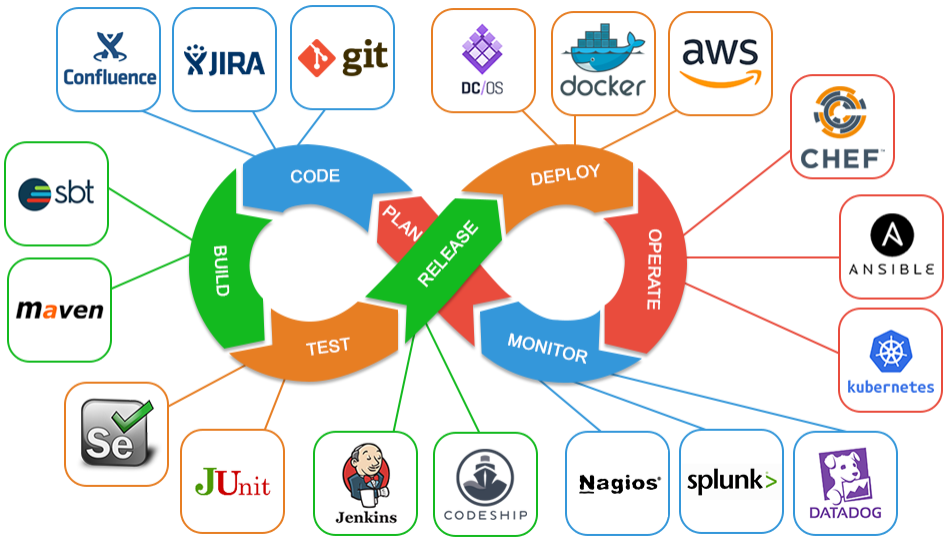
Kuva 5. Monoliittinen arkkitehtuuri ja mikropalveluarkkitehtuuri (Malav, 2017).

Mikropalveluita rakennetaan pienissä kehitystiimeissä. Tämä mahdollistaa samaan järjestelmään eri mikropalveluita rakentaville tiimeille vapauden toimia itsenäisesti. Jokainen kehitystiimi voi julkaista oman sovelluksensa ilman erillistä koordinaatioita. Pienet kehitystiimit johtavat parempaan tuottavuuteen, kun vältetään isojen tiimien aiheuttamat kommunikaatiokatkokset. (Richardson, 2020)

4.1 DevOps-toimintamalli

DevOps on uusi ohjelmistokehitystapa, jolla halutaan yhdistää ohjelmistojen kehittäminen (Dev) sen käyttöönoton ja operoinnin (Ops) kanssa (Kang et al., 2016). Se on joukko käytäntöjä, jotka nopeuttavat ohjelmien kehittämistä, testaamista ja käyttöönottoa ja kannustavat eri tiimien välistä yhteistyötä. Tämä toimintamalli on evoluutio ketterästä kehityksestä. DevOps automatisoi ohjelmistokehityksen prosessissa koodimuutosten siirtämistä kehitysympäristöstä tuotantoon (Waseem et al., 2020). Mikropalveluarkkitehtuurin katsotaan syntyneen yhteisten DevOps-tapojen käyttöönoton kautta.

Docker on DevOps-työkalu, jota käytetään sovellusten käyttöönotossa. Se on avainasemassa DevOps-toimintamallissa ja se mahdollistaa prosessin automatisoinnin. Kuvassa 6 on kokoelma eri ohjelmistoja, joita voidaan käyttää DevOps-toimintamallin toteuttamiseen. Kaikkia kuvassa mainittuja ohjelmistoja ei esitellä tutkielmassa sen tarkemmin. Se kuvaa myös toimintamallin kulkua: suunnittelu (plan), koodaus (code), rakennus (build), testaus (test), julkaisu (release), käyttöönotto (deploy), operointi (operate) ja monitorointi (monitor). Kun kehittäjä puskee muutokset Git-versionhallintaan, rakennetaan uusi Docker-kuva, jolle ajetaan yksikkö- ja integraatiotestit. Jos testit menevät läpi voidaan kuva ottaa käyttöön tuotantoympäristössä.



Kuva 6. DevOps-toimintamalli (Shown, 2020).

Devops-toimintamallin avulla voidaan kehittää, ottaa käyttöön ja hallita mikropalveluita. Mikropalveluarkkitehtuurilla ja DevOpsilla on monia yhteisiä ominaisuuksia, jonka takia ne sopivat täydellisesti yhteen. Molemmat korostavat isojen ongelmien jakamista pienemmiksi ja niiden käsittelemistä pienissä monialaisissa tiimeissä. Kontissa pyörivät

mikropalvelut voidaan toteuttaa itsenäisesti jatkuvan toimituksen (continuous integration) ja käyttöönoton avulla. DevOpsin ja mikropalveluiden yhdistelmän odotetaan nostava tiimien tuottavuutta, suorituskykyä ja järjestelmän kokonaisuuden laatua. Muita hyötyjä ovat säännöllinen ohjelmistojen julkaisu, järjestelmien luotettavuus ja skaalautuvuus ja nopea palautuminen vikatiloista. (Waseem et al., 2020)

4.2 Dockerin rooli mikropalveluissa

Nadareishvili ja muut (2016) tuovat esiin, että konttiteknologiaa ei luotu mikropalveluita varten. Ne olivat vastaus tarpeeseen saada universaali ja ennustettava työkalu monimutkaisten ohjelmistojen käyttöönottoon. Palvelimelle tarvitsee asentaa pelkkä Docker. Tämä yksinkertaistaa käyttöönottoprosessia, kun vertaa siihen, että ohjelma asennettaisiin suoritettavana tiedostona (executable), jonka riippuvuuksista pitää itse huolehtia.

Voidaan sanoa, että Docker ajaa enemmän yhtiöitä mikropalveluiden käyttöönottoon, kuin toisinpäin (Nadareishvili et al., 2016). Se painottaa Unixin periaatetta ”tee yksi asia ja tee se hyvin”. Dockerin dokumentaatio ohjeistaa: ”Aja vain yksi prosessi konttia kohti. Lähes kaikissa tilanteissa sinun pitäisi ajaa vain yksi prosessi yhdessä kontissa. Sovellusten irrotus useampaan konttiin tekee vaakasuuntaisesta skaalautumisesta (horizontal scaling) ja konttien uudelleenkäytöstä helpompaa.”. Dockerin ydinfilosofia on hyvin lähellä mikropalveluita. Sen avulla voidaan ratkaista mikropalveluiden tuomia haasteita ohjelmistokehityksessä. Docker mahdollistaa itsenäisten tiimien työskentelyn eri teknologioiden kanssa. Sen siirrettävyys auttaa yksittäisten mikropalveluiden testaamista eristyksessä muista. Docker-konttien keveyden ja skaalautuvuuden takia mikropalveluiden käyttämät palvelin- ja pilvipalveluresurssit käytetään paremmin hyväksi. (Jaramillo et al., 2016)

Bogner ja muiden (2019) tekemässä tutkimuksessa haastateltiin ohjelmistokehittäjiä, jotka kehittivät neljäätoista eri mikropalvelua. Yksitoista mikropalvelua käytti Dockeria ohjelmistojen käyttöönotossa. Ne kolme, jotka eivät vielä käyttäneet, suunnittelivat siihen siirtymistä. Kehittäjät kehuivat Dockerin käytettävyyttä ja siirrettävyyttä. Dockerin avulla järjestelmiä voidaan ajaa niin pilvipalvelussa kuin asiakkaan omalla palvelimella.

Nadareishvili ja muut (2016) näkevät Dockerin ja mikropalveluarkkitehtuurin saman tien kahtena päänä, molemmat vievät samaan määränpäähän, jatkuva toimitus ja toiminnallinen tehokkuus.

4.3 Hyödyt ja heikkoudet

Mikropalveluarkkitehtuuri tuo mukanaan monia hyötyjä. O'Connor ja muut (2017) nostavat kolme tärkeintä. Ensinnäkin modulaarisen ja irrallisen järjestelmän ylläpitäminen on helpompaa kuin klassinen luokkahierarkian. Toiseksi mahdollisuus julkaista palveluita nopeasti tuotantoympäristöön, koska palvelut ovat toisistaan riippumattomia ja ainoastaan muuttunutta palvelua tarvitsee testata. Kolmanneksi mikropalvelut ovat hyvin yhtenäisiä koodikantoja, joita on helpompi ylläpitää. Tämä puolestaan vähentää kehittäjien taakkaa ja mahdollistaa selkeämmän koodin kirjoittamisen ja vähentää vikojen syntymistä. Bogner ja muiden (2019) haastattelussa mikropalveluiden ylläpidettävyyttä pidettiin pääsyynä siirtyä pois monoliittisesta arkkitehtuurista. Koska mikropalvelut ovat luonnostaan modulaarisia, se johtaa ketterään ja itsenäiseen julkaisuprosessiin.

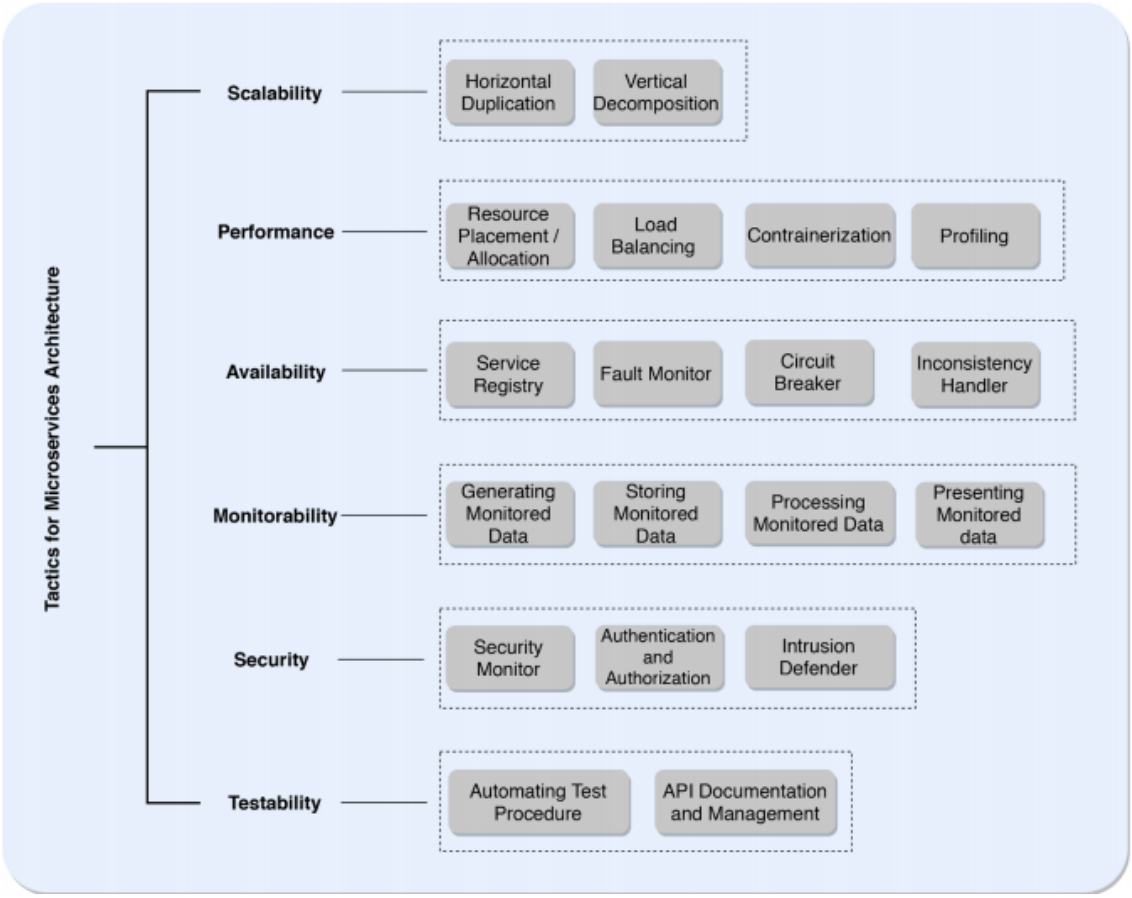
Toisin kun monoliittisessa sovelluksessa mikropalvelut voidaan rakentaa eri ohjelmointikielillä. Mikropalveluarkkitehtuuri antaa kehittäjille vapauden valita oikean teknologian tehtävää ja kehittäjiä varten (Jaramillo et al., 2016). Mikropalveluarkkitehtuuri mahdollistaa joustavan skaalautuvuuden. Pullonkaulana toimivia palveluita voidaan luoda enemmän työkuormituksen kasvaessa. Tämä tarjoaa myös joustavuutta virhetilanteiden varalta, koska pyyntöjä voidaan tasapainottaa usean mikropalveluinstanssin välille (Cerny et al., 2018).

Mikropalveluarkkitehtuurissa on omat heikkoutensa ja se ei ole aina oikea ratkaisu. Suurin heikkous on hajautetun järjestelmän lisäämä kompleksisuus. Yksittäisen palvelun kompleksisuus on alhainen mutta kokonaisuus on vaikea hallita. Pyyntöjen tekeminen usean palvelun kautta on vaikeaa ja vaatii tarkkaa koordinaatiota tiimien välillä (Richardson, 2020). Mikropalveluiden kompleksisuuden hallitseminen vaatii selkeää suunnittelua ja koordinoitua ohjelmistokehitysprosessia (Apel et al., 2018).

Vaikka yhden palvelun testaaminen helpottuu, voi lisääntynyt kompleksisuus vaikeuttaa koko järjestelmän testaamista. Myös ohjelmien laadun analysointi vaikeutuu, kun koodi on hajautettuna useampaan ohjelmavarastoon. Tietoturva on monelle huolenaihe lisääntyneen hyökkäyspinta-alan takia. ”Ennen piti varmistaa yksi ovi, nyt kaksikymmentä”, mainitsi yksi haastatelluista. (Bogner et al., 2019)

4.4 Mikropalveluarkkitehtuurin laatukriteerit

Mikropalveluarkkitehtuurin käytön yleistyessä on tärkeää tarkastella yleisiä ohjelmistotuotannon laatukriteerejä ja miettiä miten ne kannattaa ottaa huomioon mikropalveluiden suunnittelussa ja laadun varmistamisessa. Li ja muiden (2021) tekemässä systemaattisessa kirjallisuuskatsauksessa tunnistettiin mikropalveluarkkitehtuurille kuusi tärkeintä laatukriteeriä. Ne ovat skaalautuvuus (scalability), suorituskyky (performance), saatavuus (availability), seurattavuus (monitorability), turvallisuus (security) ja testattavuus (testability). Kirjallisuuskatsauksessa käytiin läpi 72 tutkimusta vuosilta 2015-2018. Laatukriteerien tunnistamisessa käytettiin yleistä standardia ISO/IEC 25010. ISO/IEC-laatumalli määrittelee mitkä laatukriteerit otetaan huomioon, kun arvioidaan ohjelmistotuotteen ominaisuuksia (ISO, 2021). Li ja muut (2021) havaitsivat yhdeksäntoista taktiikkaa, jotka arkkitehtuurillisesti vastaavat näihin kriittisiin laatukriteereihin. Kuvassa 7 on ryhmitelty laatukriteereihin liittyviä taktiikoita.



Kuva 7. Mikropalveluarkkitehtuurin laatukriteerit (Li et al., 2021).

Skaalautuvuudella mitataan järjestelmän kykyä säädellä resursseja palvelupyyntöjen käsittelemiseksi. Horisontaalinen ja vertikaalinen skaalaaminen ovat kaksi erilaista tapaa lisätä resursseja. Horisontaalisessa skaalaamisessa resursseja lisätään nostamalla käytet-

tyjen palvelimien määrää, kun taas vertikaalisessa skaalaamisessa lisätään tehoa käytettyihin palvelimin. Skaalautuvuuden toteuttamiseen löytyi kaksi taktiikkaa: horisontaalinen monistaminen ja vertikaalinen hajottaminen. Horisontaalisen monistamisen idea on päättää kuinka monta mitäkin mikropalvelua tarvitaan käsittelemään saatu liikenne. Tähän taktiikkaan voidaan käyttää reaktiivista skaalautumista, joka käyttää valmiiksi määriteltyjä kynnyksiä, http-viivettä tai prosessorikuormaa mikropalvelu instanssien kohdentamisessa. Toinen tapa on proaktiivinen skaalaaminen, joka ennustaa tarvittavan työmäärän mikropalveluille käyttäen apuna historiatietoa saaduista pyynnöistä. Vertikaalinen hajottaminen parantaa skaalautuvuutta erottamalla vastuita, toimintoja ja tietoa. Järjestelmän oikeaoppinen hajottaminen eri mikropalveluihin on edellytys skaalautuvuudelle. (Li et al., 2021)

Suorituskyky mittaa järjestelmän kykyä vastata tapahtumiin nopeasti ja mahdollisimman vähän resursseja käyttäen (ISO, 2021). Suorituskyky ja skaalautuvuus ovat kääntäen verrannollisia. Mikropalveluiden pieni koko parantaa skaalautuvuutta, mutta heikentää suorituskykyä vuorovaikutusten määrän kasvaessa mikropalveluiden välillä. Suorituskyvyn parantamiseen vaikuttavat taktiikat jakautuvat neljään kategoriaan: resurssien hallinta ja jakaminen, kuormantasaus (load balancing), konttiteknologia ja profilointi. Kuormantasaus on taktiikka, jossa yksittäiselle mikropalvelulle tuleva liikenne jaetaan sen eri instanssien välille. Yleisin on keskitetty kuormantasaaja ja se toimii käyttöliittymän ja mikropalveluiden välissä. Konttiteknologiaa edustava Docker vaikuttaa mikropalveluiden suorituskykyyn kahdella tavalla. Ensinnäkin kontit jakavat isäntäkäyttöjärjestelmän ja sen takia konttien välinen kommunikointi on nopeampaa, se kuluttaa vähemmän muistia ja vähentää infrastruktuurikuluja. Toiseksi Docker-konttien keveyden ansiosta mikropalveluita voidaan luoda ja ajaa enemmän. Se puolestaan johtaa korkeampaan resurssien käyttöasteeseen. (Li et al., 2021)

ISO/IEC 25010 mittaa saatavuuden missä määrin järjestelmä on toiminnassa ja käytettävissä, kun sitä tarvitaan. Saatavuus määritellään mikropalvelun kykynä toipua virhetilanteista mahdollisimman nopeasti. Saatavuus on hyvin tärkeä ominaisuus mikropalveluille, koska ne ovat herkkiä häiriöille ja ongelman juurisyyn löytäminen mikropalveluista on vaikeaa monimutkaisten vuorovaikutusten takia. Saatavuuden parantamiseksi voidaan käyttää: vianvalvontaa, palvelurekisteriä tai katkaisijaa. Palvelurekisteri on keskitetty palvelu, joka tallentaa tietoa kaikista järjestelmän aktiivisista mikropalveluista. Aina kun mikropalvelu käynnistyy, sen tiedot tallentuvat rekisteriin ja se pitää tiedon tallessa, kunnes mikropalvelu ei enää vastaa periodisiin tilatieto pyyntöihin. (Li et al., 2021)

Mikropalveluarkkitehtuurissa monitorointi seuraa infrastruktuuri-, ohjelma- ja ympäristötietoa. Taktiikat sen toteuttamiseen jakautuvat neljään osaan: seurattavan tiedon keräämiseen, tallentamiseen, käsittelyyn ja esittämiseen. Monitorointi auttaa selvittä-

mään juurisyyt virhetilojen syntymiseen ja analysoimaan mikropalveluiden välisiä kutsuja ja vuorovaikutuksia (Li et al., 2021). Yleinen käytäntö on rakentaa mikropalveluun kirjaustoiminto (logging), joka tallentaa levylle tiedon jokaisesta kutsusta ja yhteydestä mitä mikropalvelu suorittaa. Kerätty tieto pitää käsitellä, jotta sitä voidaan tulkita. Siihen on kaksi tapaa: aggregoitu käsittely, jossa kerätty tieto yhdistetään ja ei-aggregoitu käsittely, jossa tieto pidetään alkuperäisessä muodossa. Molemmissa on hyvät ja huonot puolensa. Aggregoitu käsittely toimii pitkänaikavälin monitorointiin, kun taas ei-aggregoitu lyhytaikaiseen analysointiin. Monitoroidun tiedon esittäminen jakautuu eri näkymiin riippuen kunkin sidosryhmän tarpeista. Mikropalvelukohtainen näkymä näyttää mittareita vasteajasta, epäonnistumisprosentista, suorituskyvystä sekä prosessorin ja muistin käytöstä. (Li et al., 2021)

Turvallisuus mittaa järjestelmän kykyä suojella tietoa ja luvatonta pääsyä järjestelmään ja samalla hallinnoi pääsyä tietoon, johon henkilöillä ja palveluilla on lupa (ISO, 2021). Turvallisuusmonitorointi on taktiikka, jossa valvotaan mikropalveluja epänormaalin toiminnan ja hyökkäysten varalta. Monitorointia voidaan tehdä paikallisesti mikropalvelun sisällä tai ulkoisesti. Toinen taktiikka on käyttää todennusta ja valtuutusta. Avainperustainen todennus on yksi tapa varmentaa yhteys kahden palvelun välillä. Toinen tapa on asiakastodistus, kuten transport layer security (TLS). Kun mikropalvelut käyttävät REST-rajapintaa kommunikointiin voi TLS todentaa molemmat osapuolet sertifikaattien avulla. (Li et al., 2021)

Testattavuus on järjestelmän kyky esittää viat normaalin testauksen kautta. Mikropalveluiden vuorovaikutusten kompleksisuuden ja niiden ominaisuuksien kasvaessa on otettava huomioon järjestelmän testattavuus. Li ja muut (2021) esittävät eri testausmetodeiksi: järjestelmä-, palvelu-, joustavuus- ja regressiotestauksen. Automaattinen testaus on riippuvainen toisesta testattavuustaktiikasta, rajapintadokumentoinnista ja -hallinnasta. Jotta testejä voidaan toteuttaa, on tiedettävä rajapintojen kuvaukset, sisään- ja ulostulevat muuttujat sekä rajapinnan syntaksi. Hyvin ylläpidetty dokumentaatio auttaa kehittäjiä jatkuvasti muuttuvien ja päivittyvien mikropalveluiden kanssa.

5 Case-esimerkki: matkapuhelinverkon analysointiohjelmisto

Tässä case-esimerkissä käyn läpi Nokialla kehitettyä matkapuhelinverkon analysointiohjelmistoa. Järjestelmä kerää tukiasemilta reaaliajassa dataa ja analysoi sen avulla verkon tilaa. Sen avulla voidaan diagnosoida viallisia tukiasemia ja puhelimia. Etenkin virhetapahtumia koskeva tieto kiinnostaa teleoperaattoreita ja tukiaseman laitevalmistajia. Järjestelmä rakentuu keskuspalvelimesta (central) ja keräilypalvelimesta (collector). Keräilypalvelimia voi olla käytössä useampia. Keräilypalvelimilla kerätään tukiasemien lähettämää tietoa, joka sitten lähetetään keskuspalvelimen ohjelmille. Esittelen osan järjestelmästä ja kuinka sen web-käyttöliittymä on rakennettu useasta eri käyttöliittymästä ja Docker-kontista. Käyn läpi mitä hyötyjä kehitystiimi on saanut Dockerin käytöstä ja mikropalveluarkkitehtuurista.

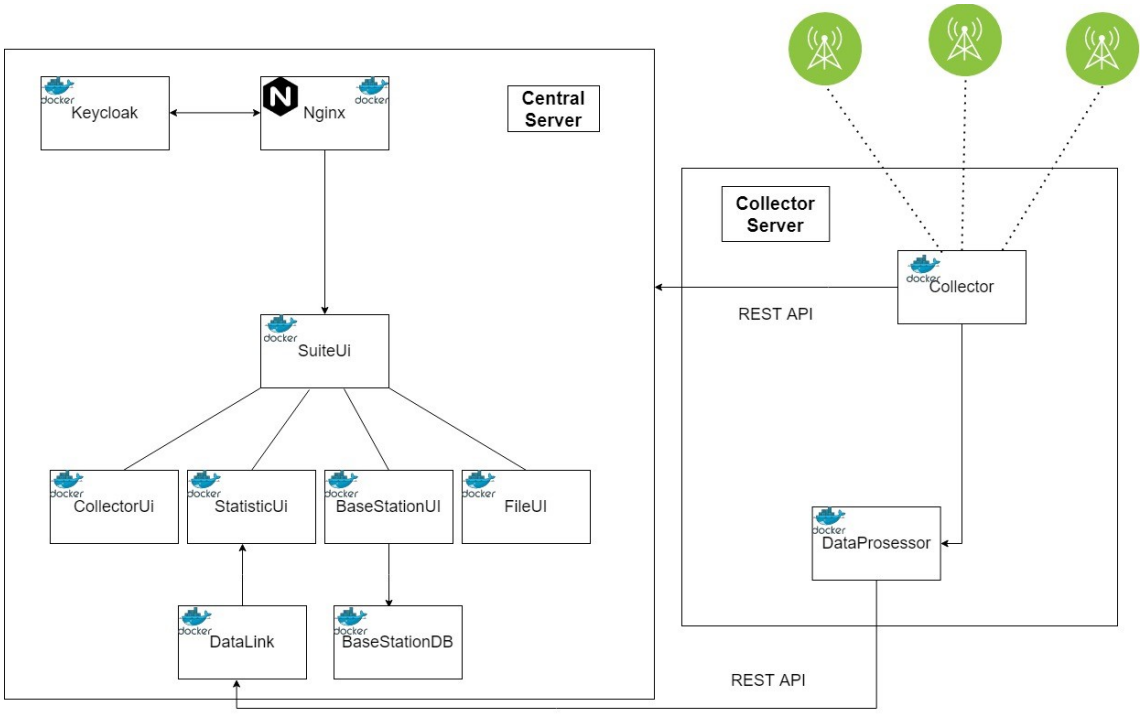
Haastattelin tutkielmaa varten eri henkilöitä kehitystiimistä. Henkilöiden roolit vaihtelivat projektipäälliköstä, testaajaan ja järjestelmän kehittäjiin. Suoritin haastattelut avoimina haasteluina VoIP-palvelun kautta. Haastattelut olivat vapaamuotoisia ja puheenaiheet vaihtelivat henkilöstä ja roolista riippuen. Keskusteluiden aiheet pyörivät Dockerin, mikropalveluiden ja ohjelmistokehityksen toimintatapojen ympärillä. Kysyin heiltä mitä he ovat tykänneet Dockerista, mikropalveluarkkitehtuurista, DevOps-toimintavoista ja onko trendi jatkumassa yhä hajautetumpiin järjestelmiin. He myös vastasivat järjestelmän arkkitehtuuriin liittyviin kysymyksiin ja siihen, miten näihin ratkaisuihin oli päädytty. Oma roolini kehitystiimissä on ollut testiautomaation kehittäminen ja tuotantoputkien rakentaminen. Testauksen kautta olen tutustunut tuotteen eri komponentteihin kattavasti ja saanut hyvän kuvan järjestelmän kokonaisuudesta.

5.1 Järjestelmän esittely

Järjestelmä on suunniteltu mikropalveluarkkitehtuurin mukaisesti. Toiminnallisuudet on jaettu pieniin osiin ja ne kommunikoivat REST-rajapinnan ja TCP-yhteyksien avulla. Komponentit ovat eri ihmisen tekemiä ja niitä on tehty eri ohjelmointikielillä. Jotkin käyttöliittymät on ohjelmoitu Angularilla ja toiset Reactilla. Mikropalveluarkkitehtuuri on mahdollistanut kehittäjille vapauden valita oikeat työkalut tehtävään ja mitä kukin osaa parhaiten käyttää. Osa komponenteista on tehty ennen Dockeriin siirtymistä. Se on vaikuttanut siihen, kuinka paljon ne hyödyntävät Dockeria ja kuinka kontit on rakennettu.

Web-käyttöliittymän avulla voidaan kerätä verkon diagnostiikkaa ja seurata tilastojen avulla verkon tilannetta. Käyttöliittymä koostuu useasta erillisestä komponentista, jotka on yhdistetty yhteen. Käyn läpi lyhyesti, miten kokonaisuus rakentuu ja mitä eri komponentit tekevät. Kuvassa 8 on kuvattu pelkistetty versio järjestelmän arkkitehtuurista, ja siitä miten eri Docker-kontit toimivat yhteen. Keräilypalvelimella pyörivä Collector-ohjelma kerää reaaliajassa tietoa tukiasemilta ja lähettää sitä TCP-yhteyksien kautta muille ohjelmille. Collectorilla on myös REST-rajapinta, jonka avulla muut ohjelmat voivat pyy-

tää siltä tietoa ja antaa sille käskyjä. Dataprossesor-ohjelma yhdistyy Collectoriin ja analysoi ja laskee sen keräämää tietoa. Kerätty tieto lähetetään REST-rajapinnan kautta keskuspalvelimen DataLink-ohjelmalle, joka ohjaa sen käyttöliittymälle.



Kuva 8. Järjestelmän arkkitehtuuri.

Keskuspalvelimella käytetään Nginxiä (2020) käänteisenä välityspalvelimena (reverse proxy), joka ohjaa yhteyksiä käyttöliittymään ja muihin järjestelmän sovelluksiin. Nginx on avoimen lähdekoodin ohjelmisto, jota voi myös käyttää www-palvelimena, välityspalvelimena tai kuormantasaajana. Kun käyttöliittymään ottaa yhteyden, ohjaa Nginx käyttäjän kirjautumaan ja siihen käytetään Keycloakia (2020). Se on myös avoimen lähdekoodin ohjelmisto ja se on ollut suosittu etenkin mikropalveluiden kanssa. Se mahdollistaa kertakirjautumisen (single sign-on) useaan palveluun ja hallinnoi käyttäjätilien oikeudet eri resursseihin. Käyttäjänhallinnan hoitaminen Keycloakin avulla vähentää kehittäjien työmäärää, kun se hoituu valmiilla ratkaisulla. Keskuspalvelin toimii myös Docker-rekisterinä, jonka kautta kontit ladataan keräilypalvelimille.

Suite-käyttöliittymän kautta käyttäjä voi navigoida eri käyttöliittymiin. Muut käyttöliittymät on upotettu sen sisälle niin että kokonaisuus vaikuttaa yhtenäiseltä web-sovellukselta. Basestation-käyttöliittymästä voidaan lisätä ja tarkastella yhdistettyjä tukiasemia. Sillä on oma sql-tietokantakontti, jossa se säilyttää tukiasemien tietoja. File-käyttöliittymän avulla voidaan ladata ja etsiä kerättyjä tiedostoja keräilypalvelimilta. Se kutsuu Filemanager-ohjelmaa keräilypalvelimelta, joka etsii haettuja tiedostoja levyltä. Collector-käyttöliittymän kautta voidaan hallinnoida käytössä olevia Collectoreita. Sen kautta

voidaan nähdä kuinka paljon liikennettä tukiasemat lähettävät niille. Käyttöliittymä käyttää Collectorin REST-rajapintaa tiedon hakemiseen ja komentojen antamiseen. Statistic-käyttöliittymä näyttää tukiasemien tilastoja, hälytyksiä ja visualisoi niiden terveydentilaa. Monella käyttöliittymällä on backend omassa kontissa.

5.2 Dockerin käyttökokemukset

Kehitystiimi käyttää Docker-rekisterinä Jfrog (2020) Artifactorya. Se on kaupallinen ohjelmistovarasto, johon voi luoda yksityisiä Docker-rekistereitä. Komponenteilla on Gitlab-ohjelmistovarastossa omat tuotantoputket (CI pipeline). Aina kun tehdään muutos komponenttiin, tuotantoputki ajaa automaattiset yksikkötestit komponentille. Jos ne menevät läpi, luodaan komponentista uusi Docker-kuva, joka julkaistaan Artifactoryyn. Sieltä testaajat voivat helposti ladata kuvan testiympäristöön ja kokeilla muutoksia. Konttien hallinnointiin käytetään Dockerin työkalua docker-compose. Sen avulla voi käynnistää monta konttia samanaikaisesti. Se käyttää yaml-tiedostoa, johon kirjoitetaan käynnistettävät palvelut. Tämä jälkeen voidaan kaikki kontit käynnistää yhdellä komennolla. Lisäksi voidaan määritellä, hakeeko Docker-moottori kontin kuvan isäntäkäyttöjärjestelmästä vai Docker-rekisteristä. Tulevaisuudessa kehitystiimin on tarkoitus siirtyä docker-composesta konttien orkestrointijärjestelmään. Sen tuomat edut ovat palveluiden skaalaaminen liikenteen mukaan ja konttien hallitseminen automaattisesti. Ilman orkestrointijärjestelmää hyödynnetään vain osa Dockerin tuomista ominaisuuksista.

Haastatteluissa kysyin järjestelmän kehittäjiltä, miten he suhtautuvat Dockerin käyttöön. Eräs kehittäjä kommentoi sitä, kuinka helppoa Dockerin avulla on päivittää versiota sovelluksessa käytetyistä kirjastoista ja riippuvuuksista. Ei tarvitse kuin muuttaa Dockerfile-tiedostoa ja uudet versiot ovat käytössä seuraavassa kuvassa. Hän kehui myös sitä, kuinka helposti ympäristön pystyttäminen ja päivitys Dockerin avulla onnistuu. Kehittäjän mukaan toisinaan pienen muutoksen tekeminen käyttöliittymään ja sen testaaminen vaati kuitenkin paljon työtä, koska kontit ovat riippuvaisia toisistaan ja ympäristössä on oltava kaikki yhdistettynä, jotta muutos voidaan testata. Tietoturvaongelmat ovat pysyneet ennallaan, ongelmat ovat vain siirtyneet Docker-kuvien sisälle. Ne vaativat yleensä kehittäjiltä päivityksiä käytettyihin pohjakuviin tai kirjastoihin. Tämän takia yritetään mahdollisimman paljon käyttää samaa Docker-pohjakuvaa eri komponenttien välillä. Docker on helpottanut korjausten julkaisua asiakkaille. Kun tehdään korjaus, tarvitsee vain lähettää päivitetty Docker-kuva komponentista ja ottaa se käyttöön vanhan tilalle.

Testit voidaan ajaa Docker-kontin sisällä ja näin pitää testien ajoympäristöt aina samana. Kontit on helppo nollata Docker-komentojen avulla, jos haluaa aloittaa testin uudestaan. Ei tarvitse tietää komponenttikohtaisia komentoja, vaan samat komennot toimivat kaikkiin kontteihin. Pienet tiimit olivat kehittäjien mielestä hyvä asia, testauksen kannalta pienet kehitystiimit vaikuttavat tiedon saatavuuteen. Dokumentaatio voi olla vähäistä, koska komponentista huolehtii vain pari henkilöä. Toisaalta vastauksen saaminen

on helpompaa, mutta jos tietty kehittäjä ei ole tavoitettavissa, on ongelmatilanteiden ratkaiseminen vaikeampaa. Dockerin tuoma lisäkerros voi myös aiheuttaa ongelmatilanteita, joissa on vaikea selvittää, johtuuko vika komponentista vai Dockerin asetuksista.

6 Yhteenveto

Tässä tutkielmassa käsiteltiin Dockeria ja sen käyttöä mikropalveluarkkitehtuurissa. Tutkin virtualisointia ja kahta eri tapaa sen toteuttamiseen, virtuaalikoneet ja konttiteknologia. Kontit ovat virtuaalikoneisiin verrattuna kevyempiä, nopeampia ja paremmin skaalautuvia. Merkittävimpänä haasteena on heikompi turvallisuus ja eristys.

Docker on suosituin konttiteknologia ja yleistyy ohjelmistokehittämisen parissa jatkuvasti. Sen avulla kehittäjä voi rakentaa, jakaa ja ajaa sovelluksiaan. Kontti paketoidaan kuvaksi, joka sisältää kaiken tarvittavan toimiakseen. Kuvia on helppo jakaa rekisterien avulla, josta muut kehittäjät voivat ladata uuden version kontista. Mikropalveluarkkitehtuuri on uusi tapa suunnitella ohjelmistoja. Sovelluksen toiminnallisuudet jaetaan pieniin osiin. Mikropalvelut mahdollistavat pienien tiimien toimimista itsenäisesti toisistaan. Docker ja mikropalvelut on luotu toisiaan varten, ne jakavat Unixin periaatteen: ”tee yksi asia ja tee se hyvin”. Docker on helpottanut mikropalveluiden käyttöönottoprosessia ja ylläpitoa.

Case-esimerkissä esittelin matkapuhelinverkon analysointiohjelmistoa, joka on suunniteltu mikropalveluarkkitehtuuria ja Dockeria hyödyntäen. Kehitystiimin kokemukset Dockerin käytöstä ovat olleet positiivisia. Sen avulla mikropalveluiden käyttämät kirjastot ja riippuvuudet on helppo pitää ajan tasalla. Mikropalveluarkkitehtuuri on mahdollistanut kehittäjille vapauden valita haluamansa teknologiat. Uusien versioiden testaaminen on nopeaa ja helppoa, mutta kokonaisuuden testaaminen mikropalveluarkkitehtuurissa on monimutkaista. Mikropalveluarkkitehtuuri kasvattaa järjestelmän kompleksisuutta ja siksi DevOps-toimintamallin hyödyntäminen on tärkeää. Docker on tärkeä osa DevOps-toimintamallia, mutta sen toteuttamiseen tarvitaan paljon muutakin. Ohjelmistotuotannon organisoinnin ja ohjaamisen merkitys korostuu, sillä erikoistuneisiin osiin ja riippuvuuksiin perustuva arkkitehtuuri vaatii selkeää suunnittelua ja koordinoitua ohjelmistokehitysprosessia. Mikropalveluarkkitehtuurin laatukriteerit pitää ottaa huomioon mikropalveluiden suunnittelussa. Etenkin skaalautuvuuden ja suorituskyvyn maksimoinnissa on mietittävä tarkkaan, miten toiminnallisuudet jaetaan palveluihin ja kuinka ne kommunikoivat keskenään.

On odotettavissa, että Docker ja mikropalveluarkkitehtuuri tulevat yleistymään tulevaisuudessa. Ne tuovat ohjelmistokehitykseen vapauksia ja parannuksia, joita monoliittisessa arkkitehtuurissa ei olisi mahdollista toteuttaa. Pilvipalvelut kuten Amazonin (2020a) AWS ja Microsoftin (2020a) Azure rakentavat palveluitaan Dockerin ympärille.

IT-alalla käytetyt teknologiat vaihtuvat jatkuvasti, mutta uskon että Docker on tullut jäädäkseen.

Lähdeluettelo

Amazon. (2020a). Amazon Web Services (AWS). https://aws.amazon.com/ (Haettu 5.12.2020)

Amazon. (2020b). Verkkokauppa. https://www.amazon.com/ (Haettu 5.12.2020)

Apel, S., Hertrampf, F., & Späthe, S. (2018). Microservice architecture within in-house infrastructures for enterprise integration and measurement: An experience report. *I4CS 2018 - Proceedings of the Innovations for Community Services*. 18-20.5.2018, Žilina, Slovakia. 3-17. https://doi.org/10.1007/978-3-319-93408-2_1

Bogner, J., Fritzsch, J., Wagner, S., & Zimmermann, A. (2019). Microservices in industry: Insights into technologies, characteristics, and software quality. *Proceedings of the 2019 IEEE International Conference on Software Architecture Companion.* 25-26.3.2019, Hamburg, Germany. 187-195. https://10.1109/ICSA-C.2019.00041

Brodkin, J. (2009). *With long history of virtualization behind it, IBM looks to the future; Decades before VMware, mainframe virtualization changed computing forever.* https://www.networkworld.com/article/2254433/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html (Haettu 29.11.2020)

Bui, T. (2014). Analysis of Docker security. *Proceedings of the Aalto University T-110.5291 Seminar on Network Security*. https://arxiv.org/abs/1501.02967

Cerny, T., Donahoo, M., & Trnka, M. (2018). Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review, 17*(4), 29-45. https://10.1145/3183628.3183631

Docker. (2020). *Docker overview*. https://docs.docker.com/get-started/overview/ (Haettu 29.11.2020)

Github. (2020). Git-versionhallintasivusto. https://github.com/ (Haettu 5.12.2020)

Gitlab. (2020). Git-versionhallintasivusto. https://gitlab.com/ (Haettu 5.12.2020)

Hassan, S., Ali, N., & Bahsoon, R.. (2017). Microservice ambients: An architectural meta-modelling approach for microservice granularity. *ICSA 2017 - Proceedings of the 2017 IEEE International Conference on Software Architecture*. 3-7.4.2017, Gothenburg, Sweden. 1-10. https://10.1109/ICSA.2017.32

Hykes, S. (2014). *Docker 0.9: introducing execution drivers and libcontainer* https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/ (Haettu 25.11.2020)

ISO. (2021). ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (haettu 10.2.2021)

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using Docker technology. *Proceedings of the SoutheastCon 2016*. 30.3-3.4.2016, Norfolk, USA. 1-5. https://10.1109/SECON.2016.7506647

Jfrog. (2020). Artifactory. https://jfrog.com/artifactory/ (Haettu 5.12.2020)

Kang, H., Le, M., & Tao, S. (2016). Container and microservice driven design for cloud

infrastructure DevOps. *Proceedings of the 2016 IEEE International Conference on Cloud Engineering*. 4-8.4.2016, Berlin, Germany. 202-211. https://10.1109/IC2E.2016.26

Keycloak. (2020). Käyttäjänhallintaohjelma. https://www.keycloak.org/ (Haettu 5.12.2020)

Kubernetes. (2020). Orkestrointijärjestelmä. https://kubernetes.io/ (Haettu 5.12.2020)

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. (2021). Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology, 131,* 23. https://doi.org/10.1016/j.infsof.2020.106449

Malav, B. (2017). *Microservices vs Monolithic architecture*. https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4 (Haettu 20.11.2020)

Martin, A., Raponi, S., Combe, T., & Di Pietro, R. (2018). Docker ecosystem – Vulnerability analysis. *Computer Communications, 122*, 30-43. https://https://doi.org/10.1016/j.comcom.2018.03.011

Mesos. (2020). Orkestrointijärjestelmä. https://mesos.apache.org/ (Haettu 5.12.2020)

Microsoft. (2020a). Azure. https://azure.microsoft.com/en-us/ (Haettu 5.12.2020)

Microsoft. (2020b). Hyper-V. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/ (Haettu 5.12.2020)

Morabito, R., Kjallman, J., & Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*. 9-13.3.2015, Tempe, USA. 386-393. https://10.1109/IC2E.2015.74

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture*. O'Reilly Media.

Netflix. (2020). Suoratoistopalvelu. https://www.netflix.com/ (Haettu 5.12.2020)

Nginx. (2020). Proxy-palvelin. https://www.nginx.com/ (Haettu 5.12.2020)

O'Connor, R.,V., Elger, P., & Clarke, P. M. (2017). Continuous software engineering – A microservices architecture perspective. *Journal of Software: Evolution and Process, 29*(11), 12, e1866-n/a. https://10.1002/smr.1866

Oracle. (2020). VirtualBox. https://www.virtualbox.org/ (Haettu 5.12.2020)

Richardson, C. (2020). *Pattern: Microservice Architecture*. https://microservices.io/patterns/microservices.html (Haettu 25.11.2020)

Shipyard. (2020). Orkestrointijärjestelmä. https://shipyard-project.com/ (Haettu 5.12.2020)

Shown, S. (2020). *How to Become an DevOps Engineer in 2020*. https://medium.com/swlh/how-to-become-an-devops-engineer-in-2020-80b8740d5a52 (Haettu 25.11.2020)

Shu, R., Gu, X., & Enck, W. (2017). A study of security vulnerabilities on Docker Hub. *Proceedings of the Seventh ACM on Conference on Data and Application Se-curity and Privacy*. 22-24.11.2017, Arizona, USA. 269-280.

https://10.1145/3029806.3029832

Uber. (2020). Uber Company - Henkilökuljetuspalvelu. https://www.uber.com/ (Haettu 5.12.2020)

Wan, X., Guan, X., Wang, T., Bai, G., & Choi, B. (2018). Application deployment using microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications, 119*, 97-109. https://10.1016/j.jnca.2018.07.003

Waseem, M., Liang, P., & Shahin, M. (2020). A systematic mapping study on microservices architecture in DevOps. *Journal of Systems and Software, 170*, 30. https://https://doi-org.libproxy.tuni.fi/10.1016/j.jss.2020.110798

Xen. (2020). Hypervisor-virtualisointiohjelmisto. https://xenproject.org/ (Haettu 5.12.2020)

Zheng, C., Rupprecht, L., Tarasov, V., Thain, D., Mohamed, M., Skourtis, D., Warke, A., & Hildebrand, D. (2018). Wharf: Sharing Docker images in a distributed file system. *SoCC 2018 - Proceedings of the ACM Symposium on Cloud Computing*. 11-13.10.2018, California, USA. 174-185. https://10.1145/3267809.3267836