Tampereen yliopisto

Tomi Rantanen

DOCKERIN HYÖDYNTÄMINEN OHJELMISTOKEHITYKSESSÄ

Tietotekniikka
Kandidaatintyö
Heinäkuu 2019

TIIVISTELMÄ

Tomi Rantanen: Dockerin hyödyntäminen ohjelmistokehityksessä
Kandidaatintyö
Tampereen yliopisto
Tietotekniikka
Heinäkuu 2019

Docker on avoimen lähdekoodin virtualisointialusta, jonka avulla voidaan paketoida ohjelmakoodi ja sen tarvitsemat kirjastot virtualisoituun konttiin. Tässä työssä esitellään Docker-virtualisointialustan toimintaperiaate ja verrataan sitä virtuaalikoneella toteutettuun virtualisointiin. Työssä tutkitaan myös Dockerin hyötyjä ja haittoja ohjelmistokehityksessä kirjallisuuskatsauksen avulla.

Työn tuloksena havaittiin, että Docker helpottaa erityisesti mikropalveluarkkitehtuuria noudattavien sovellusten hallinnointia ja automatisoitua julkaisua palvelimille. Haittapuolena on Docker-konteista koostuvan järjestelmän monimutkaisuus, joka lisää kehittäjien työmäärää. Dockerin käyttöönottoa tulisi arvioida projektikohtaisesti vertailemalla siitä saatavia hyötyjä ja haittoja.

Avainsanat: virtualisointi, ohjelmistokehitys, Docker

SISÄLLYSLUETTELO

1 JOHDANTO

Viime vuosien aikana ohjelmistojen toimitustahti on muuttunut yhä nopeammaksi (Novak 2016). Yrityksillä on tarve saada ohjelmistoon tehdyt muutokset mahdollisimman nopeasti ja automatisoidusti ohjelmiston käyttäjien saataville (Leppänen et al. 2015). Ohjelmistojen automatisoitua julkaisua varten on kehitetty uusia työkaluja kuten Docker, joka mahdollistaa sovelluksen ja sen suoritusympäristön paketoimisen helposti siirrettävään virtualisoituun konttiin. Näitä sovelluksen sisältäviä kontteja voidaan kopioida automatisoidusti useille palvelimille käyttöjärjestelmästä ja palvelimen sijainnista riippumatta.

Tässä työssä tutkitaan, mitä ohjelmistokehityksen ongelmia Docker-virtualisointialusta ratkaisee. Työssä tarkastellaan myös, onko Dockerin käyttöönotossa jotain haittapuolia. Lisäksi tarkastellaan Dockerin toimintaa teknisellä tasolla ja verrataan sitä virtuaalikoneilla toteutettuun ohjelmistojen virtualisointiin.

Luvussa 2 esitellään DevOps-toimintamalli ja siihen läheisesti liittyvä jatkuvan toimituksen malli. Luvussa käsitellään myös Docker-teknologialle hyvin soveltuvaa mikropalveluarkkitehtuuria. Luvussa 3 verrataan Docker-konttien virtualisointia virtuaalikoneisiin. Lisäksi luvussa esitellään Docker-ekosysteemiin kuuluvat sovellukset ja työkalut. Luvussa 4 tutkitaan Dockerin hyötyjä ohjelmistokehitysprosessin eri vaiheissa kehityksestä ohjelmiston julkaisuun asti. Luvussa 5 käsitellään työn yhteenveto.

2 KETTERÄ OHJELMISTOKEHITYS

2.1 DevOps-toimintamalli

Perinteisesti ohjelmistoja on kehitetty vesiputousmallilla, jossa ohjelmistoprojekti etenee vaihe kerrallaan alusta loppuun. Aluksi määritellään ohjelmiston vaatimukset, minkä jälkeen ohjelmisto suunnitellaan ja toteutetaan. Lopuksi ohjelmisto testataan ja asennetaan tuotantokäyttöön. Vesiputousmallin käyttäminen vaatii ohjelmiston täydellisen määrittelyn ennen toteuttamisen aloitusta, mikä on erittäin vaikeaa. Usein vaatimukset muuttuvat projektin aikana, jolloin joudutaan palaamaan takaisin määrittely- ja suunnitteluvaiheeseen, jotta muuttuneet vaatimukset voidaan toteuttaa ohjelmistoon. Tämä aiempiin vaiheisiin palaaminen hidastaa projektin valmistumista. Ratkaisuna vesiputousmallin haasteisiin on otettu käyttöön ketterän ohjelmistokehityksen menetelmiä, joissa ohjelmisto toteutetaan pienissä osissa koko ajan iteroiden. Ohjelmiston vaatimukset muuttuvat ja täydentyvät usein kehityksen edetessä, jolloin ketterän ohjelmistokehitysmallin käyttö mahdollistaa nopeamman reagoinnin näihin muutoksiin. (Stober & Hansmann 2010, s. 16–28)

Ketterästä ohjelmistokehityksestä huolimatta ohjelmistoon tehtyjen uusien ominaisuuksien käyttöönottoa on hidastanut erillisten kehitys- ja ylläpitotiimien käyttö ja ohjelmiston käyttöönottovaiheen vaatima manuaalinen asennus- ja konfigurointityö (Humble & Farley 2010, s. 4–5). Ratkaisuna näihin ongelmiin on kehitetty DevOps-toimintamalli, jonka tavoitteena on yhdistää kehitys (development) ja ylläpito (operations), jotta ohjelmistoon kehitetyt uudet ominaisuudet saadaan otettua mahdollisimman nopeasti käyttöön. Ohjelmistoon tehtävät muutokset, kuten uudet ominaisuudet tai virheiden korjaukset, tuottavat ohjelmiston omistajalle arvoa vasta sitten kun muutokset ovat ohjelmiston käyttäjien saatavilla. DevOps-toimintamallin tavoitteena on lyhentää ohjelmistokehityksen sykliä automatisoitujen työkalujen ja hyvien käytäntöjen avulla. (Humble & Farley 2010, s. 28)

Yksi esimerkki hyvästä käytännöstä on palvelinympäristöjen mallintaminen ohjelmistokoodin tapaan (Infrastructure as code). Käytännön mukaisesti ohjelmiston suoritusympäristö on dokumentoitu skripteinä ja ohjelmakoodina, joiden avulla voidaan asentaa uusia palvelimia täysin automaattisesti. Skriptien käyttö poistaa asennuksista manuaalista työtä vaativat vaiheet ja siten vähentää suoritusympäristön asennusvirheiden mahdollisuutta. (Morris 2016) Automatisaatio myös nopeuttaa palvelinten asennuksia. Perintei-

sesti uusien palvelimien käyttöönotto on saattanut kestää jopa useita päiviä, mutta nykyään automatisoidut työkalut mahdollistavat uusien palvelinten käyttöönoton muutamissa sekunneissa (Humble & Farley 2010, s. 303–304).

2.2 Jatkuva toimitus

Jatkuva toimitus (Continuous Delivery) on ohjelmistokehityksen menetelmä, jonka tavoitteena on DevOps-menetelmän tavoin nopeuttaa ohjelmiston uusien ominaisuuksien käyttöönottoa. Jatkuvan toimituksen ohjelmistoprojekteissa uusia ominaisuuksia ja virheiden korjauksia voidaan toimittaa jopa useita kertoja päivässä. Nopea käyttöönotto mahdollistaa palautteen keräämisen käyttäjiltä mahdollisimman aikaisessa vaiheessa, jolloin saadaan todettua, ovatko ominaisuudet käyttäjien mielestä hyödyllisiä. Tällöin voidaan ketterän kehitysmallin mukaisesti tehdä ohjelmistoon muutoksia käyttäjien palautteen perusteella. Nopean toimitussyklin saavuttaminen vaatii mahdollisimman paljon automatisoitujen työkalujen käyttöä. Ohjelmiston testaus ja asennus tapahtuu täysin automaattisesti ja toimitusprosessi on samanlainen sekä testi- että tuotantoympäristössä. (Humble & Farley 2010)

Ohjelmiston uusien ominaisuuksien toimittamiseen ja julkaisuun liittyy aina riskejä (Humble & Farley 2010, s. 4). Esimerkiksi ohjelmistokoodissa voi olla virheitä, palvelimelta voi puuttua jokin ohjelmiston suorittamiseen tarvittava ohjelmakirjasto tai palvelin voi olla konfiguroitu virheellisesti. Jatkuvan toimitustavan ja automatisoidun julkaisuprosessin käyttö auttaa löytämään nämä ongelmat mahdollisimman aikaisessa vaiheessa, jolloin niiden korjaaminen on helpompaa (Humble & Farley 2010, s. 141).

2.3 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri on arkkitehtuurimalli, jossa järjestelmä koostuu pienistä itsenäisistä sovelluksista eli palveluista. Nämä palvelut on löyhästi sidottu (loose coupling) toisiinsa esimerkiksi HTTP-rajapinnan (Hypertext Transfer Protocol) avulla. (Lewis & Fowler 2014; Zimmermann 2017) Rajapinnan käyttö palveluiden väliseen kommunikointiin mahdollistaa palveluiden toteuttamisen eri ohjelmointikielillä ja teknologioilla toisistaan riippumatta. Mikropalveluarkkitehtuurin vastakohta on monoliittinen arkkitehtuuri, jossa koko järjestelmä koostuu yhdestä suuresta sovelluksesta. (Newman 2015)

Mikropalvelut soveltuvat hyvin jatkuvan toimituksen malliin, sillä yksittäinen palvelu voidaan päivittää automatisoidusti muista palveluista riippumatta (Chen 2018). Suuren monoliittisen järjestelmän tapauksessa sovelluksen sisäiset riippuvuudet saattavat hidastaa

uusien ominaisuuksien toimittamista. Jos pienikin muutos vaatii koko järjestelmän käyttökatkoksen, muutoksia ei voida toimittaa usein. Tällöin muutokset jäävät odottamaan julkaisupäivää, jolloin toimitetaan useita muutoksia kerralla. Suuri määrä muutoksia kasvattaa toimituksen epäonnistumisen riskiä ja vaikeuttaa virheiden löytämistä, jos toimitus epäonnistuu. Mikropalveluista koostuvassa järjestelmässä muutoksia voidaan ottaa käyttöön nopeasti ja pienissä osissa, jolloin saadaan pienennettyä ohjelmiston toimituksen epäonnistumisen riskiä. (Newman 2015)

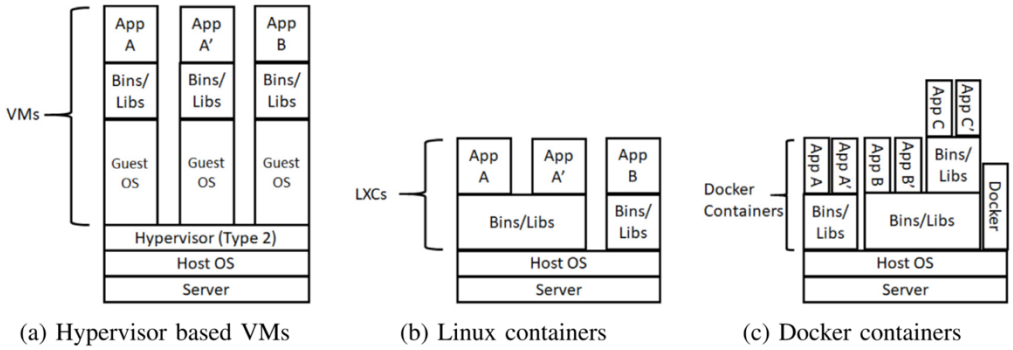
Sovelluksen jakaminen pienikokoisiin palveluihin helpottaa palveluiden kehittämistä, sillä löyhän sidonnan ansiosta yhden palvelun muuttaminen ei aiheuta muutoksia muihin palveluihin. Mikropalvelut mahdollistavat sovelluksen skaalaamisen suuremmalle käyttäjämäärälle. Jos yksittäisen palvelun kuorma kasvaa liian suureksi, voidaan samasta palvelusta käynnistää useampia rinnakkaisia palveluita jakamaan kuormaa. Palveluiden rinnakkainen suoritus lisää myös järjestelmän vikasietoisuutta. Jos yksi palvelu ei ole käytettävissä virhetilanteen takia, muut rinnakkaiset palvelut pystyvät toimimaan sen sijalla. (Newman 2015)

3 VIRTUALISOINTITEKNOLOGIAT

3.1 Virtuaalikoneet ja kontit

Virtuaalikone (virtual machine) on ohjelmistotasolla toteutettu tietokone, joka emuloi fyysistä tietokonelaitteistoa. Käyttäjän näkökulmasta virtuaalikone käyttäytyy fyysisellä laitteistolla suoritettavan tietokoneen tavoin. Virtualisoinnin avulla palvelimen käyttöastetta saadaan suuremmaksi, sillä yhden palvelimen kapasiteettia voidaan jakaa usealle virtuaalikoneelle. Tällöin tarvitaan vähemmän fyysisiä palvelimia, mikä pienentää laitteiston ylläpitokustannuksia. (Portnoy 2016, s. 11–16) Kustannussäästön lisäksi virtualisointi tuo joustavuutta palvelinten hallintaan. Virtuaalikoneita voidaan luoda lisää tarpeiden mukaan ja niitä voidaan siirtää fyysiseltä palvelimelta toiselle (Humble & Farley 2010, s. 303–304).

Virtualisointiteknologiat voidaan jakaa karkeasti kahteen kategoriaan: virtuaalikoneisiin ja virtualisoituihin kontteihin (container virtualization) (Bui 2014). Virtuaalikoneiden tapauksessa luodaan hypervisor-hallintasovelluksen avulla fyysistä tietokonelaitteistoa emuloiva virtuaalikone, jossa suoritettava sovellus on täysin eristetty laitteistosta ohjelmistokerroksen avulla (Martin et al. 2018). Tämä mahdollistaa useiden virtuaalikoneiden yhtäaikaisen suorittamisen samalla fyysisellä palvelimella. Virtualisointia voidaan tarkastella abstraktiokerrosten avulla (Kuva *1*). Alimpana kuvassa on fyysinen laitteisto, jonka yläpuolella toimii isäntäkäyttöjärjestelmä (host operating system). Isäntäkäyttöjärjestelmän yläpuolella toimiva hypervisor-hallintasovellus säätelee virtuaalikoneiden vieraskäyttöjärjestelmien (guest operating system) tekemiä laitteistokutsuja ja ohjaa ne isäntäkäyttöjärjestelmälle (Portnoy 2016, s. 21–22).



Kuva 1. Virtualisointiteknologiat (Wan et al. 2018)

Vaihtoehtona hypervisor-hallintasovellukseen perustuvalle virtualisoinnille on käyttöjärjestelmätason virtualisointi, jossa isäntäkäyttöjärjestelmään luodaan toisistaan eristettyjä

virtuaaliympäristöjä eli kontteja. Isäntäkäyttöjärjestelmän resursseja suoraan käyttävät kontit eivät tällöin vaadi erillistä vieraskäyttöjärjestelmää tai hypervisor-kerrosta. Tämän ansiosta kontit vaativat vähemmän resursseja ja levytilaa kuin vastaava virtuaalikone. (Bui 2014) Konttien eristämisessä toisistaan hyödynnetään käyttöjärjestelmän ytimen tarjoamia ominaisuuksia, kuten nimiavaruudet (namespaces) ja cgroups eli control groups (Grattafiori 2016).

Käynnistettävälle kontille luodaan omat nimiavaruudet, joiden kautta kontti käyttää käyttöjärjestelmän resursseja, kuten verkkoyhteyksiä ja tiedostojärjestelmää. Resurssien jakaminen nimiavaruuksiin mahdollistaa useiden konttien suorittamisen saman isäntäkäyttöjärjestelmän alaisuudessa, siten että kontit eivät pääse näkemään tai käsittelemään toistensa resursseja. Nimiavaruuksien avulla voidaan esimerkiksi käynnistää useita prosesseja samalla prosessitunnisteella (process id, PID), kunhan ne sijaitsevat eri nimiavaruuksissa. Tämän nimiavaruuksien eriyttämisen ansiosta kontit vaikuttavat käyttäjän näkökulmasta kokonaiselta käyttöjärjestelmältä, joka pystyy suorittamaan omia prosessejaan isäntäkäyttöjärjestelmästä riippumatta. Nimiavaruuksien lisäksi cgroups-ominaisuus rajoittaa ja säätelee käyttöjärjestelmän resursseja. Cgroups säätelee konttien käytössä olevaa suoritinaikaa, keskusmuistin määrää ja levylle kirjoittamista (Heo 2015). Säätelyllä varmistetaan, ettei yksittäinen kontti käytä kaikkea suoritinaikaa tai muistia (Grattafiori 2016; Docker Inc 2019e).

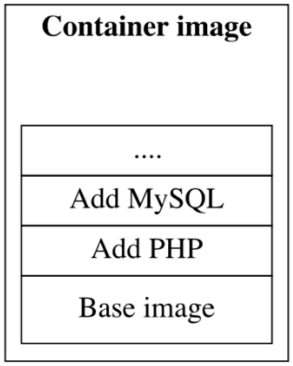
3.2 Docker Engine

Docker on avoimen lähdekoodin virtualisointiteknologia, joka mahdollistaa käyttöjärjestelmätason virtualisoinnin eli konttien suorittamisen isäntäkäyttöjärjestelmän alaisuudessa ilman hypervisor-ohjelmistoa. Virtualisoinnin toteuttavan Docker Engine -sovelluksen lisäksi ekosysteemiin sisältyy Docker Hub -palvelu, jonka avulla kehittäjät voivat jakaa tekemiään Docker-levykuvia (image) muiden käyttöön. (Docker Inc 2019d)

Kaikki Docker-konttien hallintaan ja suorittamiseen liittyvät toiminnot tapahtuvat Docker Engine -sovelluksen avulla, joka koostuu daemon-taustaprosessista, client-komentorivisovelluksesta ja niiden välisestä ohjelmointirajapinnasta (Application programming interface, API), jonka avulla pyynnöt välitetään komentorivisovelluksen ja taustaprosessin välillä. Taustaprosessi on vastuussa konttien hallinnoinnista, kuten niiden käynnistämisestä, pysäyttämisestä ja valvonnasta. Taustaprosessi toimii pyyntöjen välittäjänä kontin ja isäntäkäyttöjärjestelmän välissä. Se tarjoaa konteille pääsyn verkkoyhteyksiin ja mahdollisuuden kirjoittaa tiedostoja levylle. Ohjelmointirajapintaan perustuva arkkitehtuuri mahdollistaa taustaprosessin ja komentorivisovelluksen asentamisen eri tietokoneille, jolloin taustaprosessia voidaan hallita etänä verkkoyhteyden yli. (Docker Inc 2019d)

3.3 Docker-levykuva

Jokainen kontti perustuu levykuvaan, jossa on määritelty kontin käyttöjärjestelmä ja sen päälle asennettavat sovellukset ja muut konfiguraatiomuutokset, kuten ympäristömuuttujat. Levykuva on kontin malli (template), jolloin saman levykuvan pohjalta voidaan käynnistää useita eri kontti-instansseja. Levykuvat rakentuvat yleensä jonkin toisen levykuvan pohjalta, jonka päälle tehdään muutoksia ja lisäyksiä (Kuva 2). Näitä peruslevykuvia (base image) ovat esimerkiksi Ubuntu- ja Alpine Linux. Levykuvat rakentuvat kirjoitussuojatuista datakerroksista (read-only layer), joissa voidaan esimerkiksi asentaa sovelluksia tai kopioida tiedostoja. Jokainen muutos tallennetaan omaan datakerrokseensa, mikä mahdollistaa datakerrosten uudelleenkäytön useissa eri levykuvissa. (Docker Inc 2019d)



Kuva 2. Docker-levykuva (Bui 2014)

Levykuva määritellään Dockerfile-tiedostossa, jonka perusteella luodaan build-komentoa käyttäen lopullinen levykuva. Käännösvaiheessa Dockerfile-tiedostossa olevat komennot suoritetaan rivi kerrallaan ja jokainen muutos tallennetaan omaan datakerrokseensa. Nämä kerrokset tallentuvat välimuistiin (cache) ja niitä voidaan hyödyntää saman levykuvan luomiseen uudestaan. Jos Dockerfile-tiedostoon tehdään muutos, vain muuttuneiden rivien jälkeiset komennot pitää suorittaa uudestaan, sillä muilta osin voidaan hyödyntää välimuistissa olevia datakerroksia, mikä nopeuttaa levykuvien luomista. Lisäksi datakerrosten tallentaminen välimuistiin vähentää levytilan tarvetta, sillä eri levykuvat käyttävät yhteisiä datakerroksia, jos kerrokset ovat identtiset. Useat levykuvat voivat perustua esimerkiksi samaan Alpine Linux -peruslevykuvaan. Tällöin peruslevykuva on tallennettuna yhteiseen välimuistiin, joten peruslevykuvaa käyttävät levykuvat eivät tarvitse omaa kopiota siitä. (Docker Inc 2019a)

Dockerfile-tiedoston lopussa määritellään aloituspiste (entrypoint) eli skripti tai sovellus, joka käynnistetään kontin käynnistymisen yhteydessä. Kontin elinikä on sidottu tähän prosessiin ja prosessin pysähtyessä myös kontin suoritus pysähtyy. On mahdollista

käynnistää konttiin myös useita prosesseja samaan aikaan, mutta mikropalveluarkkitehtuurin mukaisesti on suositeltavaa tehdä jokaiselle sovellukselle oma kontti. (Docker Inc 2019g)

Käynnistymisen yhteydessä konttiin liitetään kirjoituskerros (write layer), johon kontin suorituksen aikana tehtävät muutokset tallentuvat. Jokaisella kontilla on oma kirjoituskerroksensa, joka poistuu samalla kun kontti poistetaan. Levykuvasta käynnistettävän uuden kontin suoritus alkaa aina samasta lähtötilanteesta, sillä aiemmin käynnistetyt kontit eivät tee muutoksia levykuvaan. Pysyvää tietojen tallennusta varten konttiin voidaan liittää looginen levy (volume), joka voi sijaita esimerkiksi isäntäkäyttöjärjestelmän kiintolevyllä tai verkkolevyllä. Loogiselle levylle tallennettu tieto säilyy kontin poistamisen jälkeenkin. (Wang 2016)

3.4 Docker-rekisteri

Docker-rekisteri (registry) on tietovarasto levykuvien jakelua varten. Kehittäjät voivat julkaista tekemiään levykuvia tietovarastoon ja ladata sieltä muiden levykuvia Docker client-sovelluksen avulla. Rekisterin voi asentaa omalle palvelimelle tai käyttää SaaS-palveluna (software as a service) tarjottavaa Docker Hubia (Docker Inc 2019c; Boettiger 2014). Palveluun luodulla ilmaisella käyttäjätilillä voi julkaista yhden yksityisen tietovaraston ja rajattoman määrän julkisia tietovarastoja. Yksityiset tietovarastot mahdollistavat levykuvien jakelun rajatulle joukolle kehittäjiä, kun taas julkiset tietovarastot ovat kaikkien nähtävissä ja käytettävissä. Maksullisena palveluna on myös mahdollista saada käyttöön useampia yksityisiä tietovarastoja.

Docker Hubissa olevat julkiset tietovarastot ovat usein liitettynä Github-versionhallintapalveluun, jossa on mahdollista tarkastella levykuvien Dockerfile-määrittelyjä. Docker ylläpitää virallisia levykuvia suosituista palveluista, kuten Ubuntu-käyttöjärjestelmästä, Python-ohjelmointikielestä ja PostgreSQL-tietokannasta (Docker Inc 2019f).

4 DOCKER OHJELMISTOKEHITYKSESSÄ

4.1 Hyödyt

Viime vuosien aikana Dockerista on tullut yleisesti käytössä oleva konttien virtualisointiteknologia. Se ratkaisee useita ohjelmiston kehitykseen, julkaisuun ja testaamiseen liittyviä ongelmia, jonka ansiosta erityisesti suurta määrää palvelimia hallinnoivat yritykset ovat ottaneet sen käyttöön. (Datadog 2018)

Dockerin ensisijainen käyttötarkoitus on mikropalveluarkkitehtuurin mukaisista pienistä palveluista koostuvan sovelluksen hallinnointi ja julkaisu palvelimille (Martin et al. 2018). Suositellun käytännön mukaisesti jokainen palvelu sijoitetaan omaan konttiinsa, joka sisältää palvelun tarvitsemat ohjelmakirjastot (Docker Inc 2019b). Nämä kontit voidaan julkaista automaattisesti palvelimille, mikä mahdollistaa ohjelmiston jatkuvan toimituksen useita kertoja päivässä. Automatisoitu julkaisuprosessi vähentää suoritusympäristöön liittyviä asennus- ja konfigurointivirheitä, sillä kontti on suoritusympäristöstään riippumaton, joten se toimii samalla tavoin sekä kehitys- että tuotantopalvelimilla. Kontin levykuvan muodostavaa Dockerfile-tiedostoa säilytetään versionhallintajärjestelmässä ohjelmakoodin mukana, jolloin levykuvaan tehtävät muutokset tallentuvat versiohistoriaan. Jos kontin toiminnassa tai konfiguraatiossa huomataan ongelmia, versiohistorian avulla voidaan tunnistaa, milloin konfiguraatioon on tehty muutoksia ja voidaan palata edelliseen toimivaan versioon.

Mikropalveluiden julkaisun lisäksi useat yritykset ovat ottaneet kontit käyttöön lisätäkseen kehittäjien tehokkuutta (Forrester Consulting 2017). Usein ohjelmiston kehittäjä ei saa toisen kehittäjän tekemiä muutoksia toimimaan omalla tietokoneellaan kehitysympäristöjen erojen takia. Syynä voi olla esimerkiksi puuttuva ohjelmistokirjasto tai versioltaan eroava ohjelmointikielen kääntäjä. Ohjelmiston kehitysympäristöjen eroavaisuus on yleinen ongelma myös tutkimusprojektien toistettavuudessa. Arizonan yliopiston tutkimuksen mukaan vain alle 50% vanhojen tutkimusprojektien koodista kääntyy ilman virheitä (Collberg et al. 2014). Nämä riippuvuusongelmat ympäristöjen välillä voidaan ratkaista käyttämällä Dockerin avulla standardisoitua kehitysympäristöä. Ohjelmistoa kehitetään Docker-konteissa, jolloin jokaisella kehittäjällä on käytössään samat versiot kirjastoista ja ohjelmointikielistä. Docker toimii Windows-, MacOS- ja Linux-käyttöjärjestelmissä, joten kehittäjät voivat työskennellä haluamassaan ympäristössä.

Dockerin suosion ansiosta sen ympärille on rakentunut vahva avoimen lähdekoodin yhteisö, jossa valmiita levykuvia jaetaan Docker Hub -palvelun kautta (Docker Inc 2019c).

Levykuvien modulaarisuus tekee niiden uudelleenkäytöstä helppoa. Esimerkiksi Elasticsearch-hakukoneohjelmiston levykuva perustuu Java-ohjelmointikielen sisältävään OpenJDK-levykuvaan, joka perustuu käyttöjärjestelmän sisältävään Linux-levykuvaan. Useille ohjelmointikielille, tietokannoille ja muille työkaluille on olemassa valmiit levykuvat, joiden avulla on mahdollista kokeilla uusia työkaluja nopeasti ilman niiden asentamista.

4.2 Haitat

Dockerin huonona puolena voidaan pitää sen käyttöönoton vaatimaa työmäärää. Uusi työkalu vaatii opiskelua, sillä kehittäjien pitää ymmärtää miten kontit toimivat, jotta he osaavat hyödyntää niitä. Kontit lisäävät järjestelmän monimutkaisuutta tuomalla yhden abstraktiokerroksen lisää. Esimerkiksi kehittäjien pitää osata konfiguroida verkkoyhteydet konttien välille. Joissain yrityksissä erillinen ylläpitotiimi asentaa ja konfiguroi palvelimia. Konttien käyttöönotto ohjaa kehityskulttuuria devops-mallin suuntaan, jossa samat henkilöt kehittävät sovellusta ja ylläpitävät ympäristöä. Kehittäjien vastuulla on silloin levykuvien luominen, mikä sisältää asennus- ja konfigurointityötä.

Sovellusten tietoturvan ylläpitoon tulee uusia haasteita konttiympäristöön siirtymisen myötä. Docker Hubista ladattavat valmiit levykuvat saattavat sisältää haitallista koodia. Erityisesti monimutkaiset riippuvuusketjut levykuvien välillä vaikeuttavat niiden turvallisuuden tarkistamista. Tietoturvauhkien minimoimiseksi on suositeltavaa käyttää Docker Hubin tarjoamia virallisia levykuvia, joille on tehty tietoturvaskannaukset (Docker Inc 2019f). Levykuvien tietoturvan lisäksi on tärkeää huolehtia myös Docker Engine-alustan tietoturvapäivitysten asennuksesta. Helmikuussa 2019 paljastui Dockerin käyttämästä runc-teknologiasta haavoittuvuus, jonka avulla haittaohjelman sisältävä kontti voi saada pääkäyttäjän oikeudet isäntätietokoneelle (Chanana 2019).

Suorituskyvyltään Docker-konteissa suoritettavat ohjelmat ovat lähes yhtä tehokkaita kuin ilman virtualisointia suoritettavat ohjelmat. IBM:n tutkimuksen mukaan Docker-kontit eivät vaikuta prosessointitehoa tai keskusmuistia vaativiin operaatioihin, mutta saattavat hidastaa kiintolevylle kirjoittamista tai lisätä verkkokutsujen latenssia. Jos suorituskyvyltään kriittinen sovellus halutaan siirtää kontissa suoritettavaksi, sille tulisi suorittaa kuormitustestaus, jonka avulla selvitetään kontin vaikutus sovelluksen suorituskykyyn. Erityisesti sovellukset, jotka tekevät paljon pieniä levykirjoitusoperaatioita, voivat suoriutua hitaammin Docker-kontissa. (Felter et al. 2014)

5 YHTEENVETO

Tässä tutkimuksessa esiteltiin Docker-virtualisointiteknologian toimintaperiaate ja verrattiin sitä virtuaalikoneella toteutettuun ohjelmiston virtualisointiin. Tutkimuksessa selvitettiin myös Dockerin hyötyjä ja haittoja ohjelmistokehityksessä kirjallisuuskatsauksen avulla. Virtuaalikoneisiin verrattuna Docker-konteilla on paljon hyödyllisiä ominaisuuksia. Konttien levykuvat ovat pienikokoisia, joten ne eivät vaadi paljoa levytilaa ja levykuvia on nopeaa kopioida palvelimille. Kontit käynnistyvät nopeasti, joten niitä voidaan käyttää ohjelmiston suorituskyvyn skaalaamiseen käynnistämällä uusia rinnakkaisia kontteja.

Näiden ominaisuuksien ansiosta Docker-kontit soveltuvat erityisesti mikropalveluarkkitehtuuria noudattavien järjestelmien virtualisointiratkaisuksi. Konteissa suoritettavia itsenäisiä palveluita voidaan päivittää automatisoidusti toisistaan riippumatta. Ohjelmiston julkaisuprosessin automatisointi vähentää virheitä ja mahdollistaa nopean julkaisusyklin. Tuotantokäytössä olevan ohjelmiston hallinnoinnin lisäksi Docker helpottaa kehitysympäristön yhtenäistämistä eri kehittäjien välillä. Kontteja käytettäessä jokaisella kehittäjällä on käytössään samat versiot ohjelmakirjastoista. Docker helpottaa kehitystä myös tarjoamalla suuren määrän valmiita levykuvia Docker Hub -palvelun kautta.

Dockerin haittapuolena voidaan nähdä sen monimutkaisuus, joka lisää kehittäjien työmäärää ja vaatii uuden työkalun opiskelua. Lisäksi virtuaalikoneisiin verrattuna Docker on heikommin eristetty isäntäkäyttöjärjestelmästä, mikä altistaa Docker-kontit tietoturvauhkille. Docker tarjoaa merkittäviä hyötyjä erityisesti mikropalveluista koostuvien järjestelmien ylläpitoon ja kehitykseen, minkä ansiosta Docker on saavuttanut suosiota kehittäjien keskuudessa. Se ei kuitenkaan ole välttämätön työkalu jokaisessa ohjelmistoprojektissa. Uuden sovelluksen kehitystä aloitettaessa tulisi pohtia Dockerin hyötyjä ja haittoja ja valita niiden perusteella otetaanko Docker käyttöön.

LÄHTEET

Boettiger, C. (2014). An introduction to Docker for reproducible research, with examples from the R environment, ACM SIGOPS Operating Systems Review, Special Issue on Repeatability and Sharing of Experimental Artifacts, vol. 49, no. 1, pp. 71–79.

Bui, T. (2014). Analysis of Docker Security, Aalto University T-110.5291 Seminar on Network Security.

Chanana, B. (2019). Docker Security Update: cve-2019-5736 and Container Security Best Practices, verkkosivu Saatavissa (viitattu 28.3.2019): https://blog.docker.com/2019/02/docker-security-update-cve-2018-5736-and-container-security-best-practices/.

Chen, L. (2018). Microservices: Architecting for Continuous Delivery and DevOps, IEEE International Conference on Software ArchitectureIEEE, Seattle, USA.

Collberg, C., Proebsting, T., Moraila, G., Shankaran, A., Shi, Z. & Warren, A.M. (2014). Measuring Reproducibility in Computer Systems Research.

Datadog (2018). 8 Surprising Facts About Real Docker Adoption, verkkosivu Saatavissa (viitattu 30.3.2019): https://www.datadoghq.com/docker-adoption/.

Docker Inc (2019a). About storage drivers, verkkosivu Saatavissa (viitattu 17.5.2019): https://docs.docker.com/storage/storagedriver/.

Docker Inc (2019b). Best practices for writing Dockerfiles, verkkosivu Saatavissa (viitattu 19.4.2019): https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

Docker Inc (2019c). Docker Hub, verkkosivu Saatavissa (viitattu 11.2.2019): https://hub.docker.com/.

Docker Inc (2019d). Docker overview, verkkosivu Saatavissa (viitattu 3.2.2019): https://docs.docker.com/engine/docker-overview/.

Docker Inc (2019e). Docker Security, verkkosivu Saatavissa (viitattu 31.3.2019): https://docs.docker.com/engine/security/security/.

Docker Inc (2019f). Official Images on Docker Hub, verkkosivu Saatavissa (viitattu 23.3.2019): https://docs.docker.com/docker-hub/official_images/.

Docker Inc (2019g). Run multiple services in a container, verkkosivu Saatavissa (viitattu 31.3.2019): https://docs.docker.com/config/containers/multi-service_container/.

Felter, W., Ferreira, A., Rajamony, R. & Rubio, J. (2014). An Updated Performance Comparison of Virtual Machines and Linux Containers, IBM Research Division, Austin, TX, USA.

Forrester Consulting (2017). Containers: Real Adoption And Use Cases In 2017.

Grattafiori, A. (2016). Understanding and Hardening Linux Containers.

Heo, T. (2015). Control Group v2, verkkosivu Saatavissa (viitattu 25.3.2019): https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

Humble, J. & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, Upper Saddle River, NJ, 463 p.

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V., Itkonen, J., Mäntyla, M.V. & Männisto, T. (2015). The highways and country roads to continuous deployment, IEEE Software, vol. 32, no. 2, pp. 64–72.

Lewis, J. & Fowler, M. (2014). Microservices – a definition of this new architectural term, verkkosivu Saatavissa (viitattu 23.3.2019): https://martinfowler.com/articles/microservices.html.

Martin, A., Raponi, S., Combe, T. & Di Pietro, R. (2018). Docker ecosystem – Vulnerability Analysis, Computer Communications, vol. 122, pp. 30–43.

Morris, K. (2016). Infrastructure as Code: Managing Servers in the Cloud, O'Reilly Media, 362 p.

Newman, S. (2015). Building Microservices, O'Reilly Media, Sebastopol, CA, 282 p.

Novak, A. (2016). Going to Market Faster: Most Companies Are Deploying Code Weekly, Daily, or Hourly, verkkosivu Saatavissa (viitattu 18.4.2019): https://blog.newrelic.com/technology/data-culture-survey-results-faster-deployment/.

Portnoy, M. (2016). Virtualization Essentials, 2nd edn, John Wiley & Sons, Incorporated, 334 p.

Stober, T. & Hansmann, U. (2010). Agile Software Development: Best Practices for Large Software Development Projects, Springer-Verlag, Berlin, Heidelberg, 179 p.

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, vol. 119, pp. 97–109.

Wang, C. (2016). Containers 101: Docker fundamentals, verkkosivu Saatavissa (viitattu 31.3.2019): https://www.infoworld.com/article/3077875/containers-101-docker-fundamentals.html.

Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment, Computer Science - Research and Development, vol. 32, no. 3–4, pp. 301–310.