OULUN AMMATTIKORKEAKOULU

Mikko Jokipelto

NÄKYVYYDEN RAKENTAMINEN KONTTIPOHJAISEEN MONIKLUSTERIYMPÄRISTÖÖN

NÄKYVYYDEN RAKENTAMINEN KONTTIPOHJAISEEN MONIKLUSTERIYMPÄRISTÖÖN

Mikko Jokipelto
Opinnäytetyö
Kevät 2022
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä: Mikko Jokipelto
Opinnäytetyön nimi: Näkyvyyden rakentaminen konttipohjaiseen moniklusteriympäristöön
Työn ohjaaja: Eino Niemi
Työn valmistumislukukausi ja -vuosi: Kevät, 2022 Sivumäärä: 72 + 1 liite

---

Opinnäytetyön aiheena oli tutkia eri tapoja tuottaa näkyvyyttä konttipohjaiseen moniklusteriympäristön julkaisu- ja hallintajärjestelmän tilaan. Lisäksi opinnäytetyössä käytiin läpi yleisellä tasolla modernia palvelutuotantoa konttipohjaisessa ympäristössä ja tutkittiin malleja, periaatteita sekä teknologioita, jotka toimivat sen mahdollistajina. Lisäksi tutkimus teki syvällisempää selvitystä konttiteknologian juurista ja nykytilasta sekä teknologiasta, joka mahdollistaa konttipohjaisien työkuormien orkestroinnin ja klusterien hallinnoinnin.

Tutkimustyön pohjalta alettiin rakentamaan omaa konseptia, jonka tavoitteena oli tuottaa näkyvyyttä tietyn GitOps-malliin pohjautuvan julkaisu- ja hallintajärjestelmän tilaan. Näkyvyyttä rakennettiin hyödyntämällä Kubernetes-natiivia Operaattori-ohjelmointikaavaa. Luomalla uuden mukautetun resurssin Kubernetes-rajapintapalvelinta voidaan jatkaa lisäämällä siihen uusi päätepiste. Kun mukautetun resurssin tilan sovittelemista varten luodaan mukautettu kontrolleri, saadaan uusi Operaattori. Valmis resurssi muistuttaa normaalia Kubernetes-resurssia toiminnallisuudeltaan, joten sitä pystyy myös hyödyntämään komentorivityökalun avulla. Opinnäytetyössä toteutettiin Operaattori, joka seuraa resurssien tilaa ja sovittelee sen muutokset päivittämällä ne omaan mukautettuun resurssiin. Lisäksi rakennettiin kontrolleri, joka välittää mukautetusta resurssista tilan klusterin ulkopuolelle JSON-sanoman muodossa.

Opinnäytetyön Operaattori-toteutuksen avulla onnistuttiin rakentamaan näkyvyyttä haluttuun resurssiin ja välittämään se automaattisesti klusterin ulkopuolelle standardoidussa muodossa. Tuotettua prototyyppiä voi tulevaisuudessa hyödyntää myös alustana muunlaiselle operointityön automatisoinnille, koska se tarjoaa jo valmiin toiminnallisuuden resurssien seuraamiselle ja seurattavien muutosten sovittamiselle ohjelmallisesti. Lisäksi jatkokehitystyönä voisi tutkia julkaisuautomaation rakentamista prototyyppitoteutuksen pohjalta. Operaattori tarjoaa hyvin voimakkaan työkalun Kubernetes-klusterin räätälöimiseen ja automatisointiin.

Asiasanat:

Ohjelmistokehitys, ketterät menetelmät, pilvipalvelut, Python, tietojenkäsittely, tietoverkot.

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Mikko Jokipelto
Title of thesis: Building Observability to Container-based Multi-cluster Environment
Supervisor(s): Eino Niemi
Term and year when the thesis was submitted: Spring/2022
Number of pages: 72 + 1 appendix

The purpose of this research was to find ways to build observability to a container-based multi-cluster environment deployment and management system. The research also touches on modern service production in container-based environment, and the enabling techniques and models, which allow these even more complex solutions to arise. The technologies covered in the research include microservices and distributed systems, CI/CD and GitOps, containers and Kubernetes. The research makes a deep dive into bases of container technology to build a better understanding of the higher-level abstraction build upon containers e.g., Kubernetes.

The implementation of the research takes advantage of a Kubernetes-native design pattern called Operator. The Operator consists of a custom resource, which extends the Kubernetes API-server and a custom controller or controllers which watch for changes in the resource state and reconciliate the changes with a custom build programming logic. In the implementation the Operator watches for a special resource which belongs to the deployment and management system and reconciles changes to that resource. It reconciles by updating the watched state of the resource to its' own custom resource. There is also another controller which watches for the state of the custom resource and reconciliates the changes to it by sending a JSON-message to a configurable endpoint outside of the cluster. This was done so the custom data could be utilized in a monitoring system, but it also opens an opportunity to take advantage of this message in other systems as well, like Teams, Slack or even email.

The implementation was able to produce more observability to the set system. As a result the information was also very human readable and customizable. The prototype which was born during the proof of concept constructs an extensible platform to build other automated workflows' on. It provides the basic ability to watch a set resource or a state in a particular type of resource and you can program any custom way to reconciliate the observed state. Based on the research an Operator is a very powerful tool for extending your Kubernetes cluster with custom logic or automation.

Keywords:
Agile software development, computer software development, cloud services, Python.

SISÄLLYS

SANASTO

API	Ohjelmointirajapinta (Application Programming Interface) määrittelee, miten applikaatiolle tai palvelulle voi tehdä pyyntöjä ja vaihtaa tietoa, eli kuinka palvelut tai applikaatiot voivat kommunikoida keskenään.
CI	Jatkuva integraatio (Continuous Integration) on versiohallinnan ympärille rakennettu automaatio, jossa voidaan esimerkiksi ajaa matalan tason testejä versiohallintaan julkaisun yhteydessä.
CD	Jatkuva toimitus (Continuous Delivery) on automaattisen julkaisun muoto, jossa applikaatiosta on aina uusin versio valmiina julkaistavaksi, mutta sen julkaisu edellyttää julkaisupäätöksen.
CD	Jatkuva julkaisu (Continuous Deployment) on automaattinen julkaisu muoto, jossa applikaation uusin versio julkaistaan aina automaattisesti.
CNCF	On yhteisö ja ekosysteemi (Cloud Native Computing Foundation), joka kehittää ja ylläpitää maailmanlaajuisesti pilviteknologiaa.
CR	Mukautettu resurssi (Custom Resource), jonka avulla voidaan jatkaa Kubernetes-rajapintapalvelinta uudella päätepisteellä.
CRI	Konttien ajoalustan käyttöliittymä (Container Runtime Interface) on käyttöliittymä, jolla mahdollistetaan Kubelet-komponentille riippumattomuus konttien ajoalustasta.
CRD	Mukautetun resurssin määrittelytiedosto (Custom Resource Definition) kertoo millainen mukautetun resurssin rakenne on.
GitOps	CICD julkaisumalli, joka käyttää Git-versiohallintaa ainoana totuuden lähteenä kuvaamaan järjestelmän toivottua tilaa.
IPC	Prosessien välinen kommunikaatio (Inter-process communication), eri prosessien tai säikeiden väliselle kommunikaatiolle kehitetyt tekniikat.
LXC	Linux-kontit (Linux Containers) on Linux Kernelin tukema käyttöjärjestelmätason virtualisointi, joka ei sisällä laitteiston emulointia.
MNT	Taltioiden ja osioiden liitos (Mount) tiedostojärjestelmässä. Esimerkiksi kaksi konttia voi jakaa saman hakemiston.

OCI	Avoin konttien aloite (Open container initiative) on joukko standardeja, joilla on tarkoitus yhtenäistää konttien pakkaus ja konttien ajoalustat.
PID	Käyttöjärjestelmässä pyörivien prosessien tunnukset (Process Identifier).
SDN	Ohjelmatasolla määritetty verkko (Software defined networking) on määrittely, jota käytetään Kubernetes-resurssien verkkojen määrittelyssä.
SHM	On prosessien välinen kommunikaatiotapa, jossa tietty muistialue varataan useammalle prosessille, jotka kommunikoivat kyseisen muistialueen välityksellä keskenään (Shared Memory).
UTS	Linux-nimiavaruus (Unix-time sharing), jossa määritellään verkkonimet, eli koneen-nimi ja -domain.
Veth	Virtuaalinen ethernet-liittymä (Virtual Ethernet), jota hyödynnetään muun muassa kapselin kytkemiseen verkkoon Kubernetes-klusterin sisällä.
VFS	Virtuaalinen tiedostojärjestelmä (Virtual filesystem) on Linux Kernelin ominaisuus, jonka avulla Kernel voi tukea useita erilaisia tiedostojärjestelmiä.

1 JOHDANTO

Aikoinaan rahtia kuljetettiin laivoissa niin, että lähtösatamassa laivaan lastattiin kuljetettava rahti laatikoissa, ja kun laiva taas saapui perille kohdesatamaansa, laatikot purettiin laivasta kuorma-autoihin. 1950-luvulla rekkakuski nimeltä Malcolm McLean ehdotti, että sen sijaan, että rahti työläästi lastataan ja puretaan satamassa kuorma-autoihin, itse kuorma-auton perävaunu irrotettaisiin ja siirrettäisiin satamissa laivaan, joka kuljettaisi valmiiksi pakatun vaunun kohdesatamaansa. Tämä oli lyhyt tarina rahtikonttien synnystä, konsepti, joka 50 vuotta myöhemmin otettiin käyttöön ohjelmistosovellusten toimittamiseen ja kehittämiseen. Kuten rahtikonttiesi-isänsä, kontit tarjoavat standardoidun tavan pakata ja levittää sisältöään. Kontti paketoi levykuvaksi kaikki riippuvuudet ja ympäristön, joita sen sisältämä applikaatio tarvitsee toimiakseen. Tämä tarkoittaa, että kontin levykuvan avulla sen sisältö voidaan ajaa millä tahansa päätteellä. (Domingus & Arundel 2022, luku 1.)

Kontit helpottavat kehittäjien työtä ja mahdollistavat tiheämmän julkaisutiheyden, järjestelmäresurssien tehokkaamman hyödyntämisen ja palveluiden skaalaamisen. Konttiteknologioiden yleistyttyä alettiin etsimään ratkaisua konttipohjaisten työkuormien ajamiseen ympäristössä, joka koostuisi useasta eri palvelimesta. Kubernetes on yksi tällainen järjestelmä, jolla voidaan rakentaa useammasta koneesta koostuva ympäristö, eli klusteri, ja organisoida konttipohjaisia työkuormia siinä (Domingus & Arundel 2022, luku 1)

Kubernetes on avoimeen lähdekoodiin perustuva alusta, jolla organisoidaan julkaisuja ja ympäristön skaalausta sekä hallinnoidaan konttipohjaisia applikaatioita (Baier, Sayfan & White 2019). Tärkein suunnittelutavoite Kubernetesille oli tehdä monimutkaisten hajautettujen järjestelmien käyttöönotosta ja hallinnasta helppoa, sekä samalla hyötyä konttien tehokkaammasta järjestelmäresurssien hyödyntämisestä (Burns ym. 2016, 14). Kubernetesin suosio on kasvanut niin paljon, että siitä on tulossa alan normi. Se on vakiintumassa alustaksi, jolla kontteja ajetaan aivan, kuten kontit vakinaistivat tavan, jolla applikaatiot pakataan ja julkaistaan (Domingus & Arundel 2022, luku 1).

Kubernetes-moniklusteriympäristöt mahdollistavat palvelujen julkaisemisen useammalle klusterille, jolloin ympäristöjä voidaan replikoida tai jatkaa moduuleihin. Moniklusterisen ympäristön käyttämiseen voi olla useita eri syitä, kuten järjestelmäresurssien tehokkaampi hyödyntäminen ja

lokalisaatio, regulaatioiset vaatimukset, järjestelmän vikasietoisuus ja korkea saavutettavuus sekä liiketoimintamallin mukaileminen. Moniklusteriympäristöt lisäävät järjestelmän monimutkaisuutta, joka johtaa haasteisiin etenkin tietoturvan, verkkoliikennöinnin ja palomuurien määrityksessä sekä julkaisuissa. (RedHat 2021.)

Alexis Richardson (Weave works) on vastuussa termin GitOps lanseerauksesta. Hänen mukaansa, jos komponentti on mahdollista kuvailla deklaratiivisesti, se voidaan myös validoida ja automatisoida. Jos kaikki komponenttisi on määritelty deklaratiivisesti, voit lisätä, ylläpitää ja säilyttää koko järjestelmän tilaa versiohallinnassa, jonka avulla koko järjestelmä voidaan automatisoida (Richardson 2018). GitOps-mallin keskiössä on ajatus siitä, että Git-versiohallinta on yksittäinen totuuden lähde, joka säilyttää toivottua tilaa (engl. desired state), ja GitOps-työkalu sovittelee klusterien nykytilan vastaamaan tätä toivottua tilaa. Käytännössä GitOps mahdollistaa Kubernetesin operoinnin Git-versiohallinnan kautta. (Weave works 2017.)

Operaattori toimii jatkamalla Kubernetes-ohjaustasoa ja rajapintapalvelinta (Dobies & Wood 2020). Yksinkertaisimmillaan Operaattori lisää päätepisteen rajapintapalvelimeen, joka määritellään mukautetun resurssin (engl. custom resource) ja mukautetun kontrollerin (engl. custom controller) muodossa (Dobies & Wood 2020). Operaattorit seuraavat niille määriteltyjen resurssien tilaa ja sovittelevat tilan muutokset niihin ohjelmoidulla logiikalla. (Dobies & Wood 2020.) GitOps-järjestelmät pohjautuvat Operaattoreihin (Weave works 2017).

Opinnäytetyön ensimmäinen tavoite on selvittää eri tapoja rakentaa näkyvyyttä moniklusteriympäristön julkaisu- ja hallintajärjestelmän toimintoihin ja tilaan. Seuraava tavoite on toteuttaa tämä toiminnallisuus hyödyntämällä Kubernetes-natiiveja konsepteja ja toiminnallisuuksia, jotta ratkaisu olisi helposti implementoitavissa missä tahansa Kubernetes-teknologiaa hyödyntävässä ympäristössä ja mahdollisimman vähän sidoksissa ylläpitoa vaativiin ulkopuolisiin resursseihin. Kolmantena tavoitteena toteutukselle asetettiin mahdollisuus hyödyntää tuotettua tila- ja tapahtumatietoa tilaajan monitorointijärjestelmässä. Viimeiseksi tavoitteeksi asetettiin toimivan prototyypin toimitus, joka todistaa suunnitellun konseptin toimivuuden.

Opinnäytetyön tutkimuskysymykseksi asetettiin, miten moniklusteriympäristön julkaisu- ja hallintajärjestelmään voidaan rakentaa näkyvyyttä.

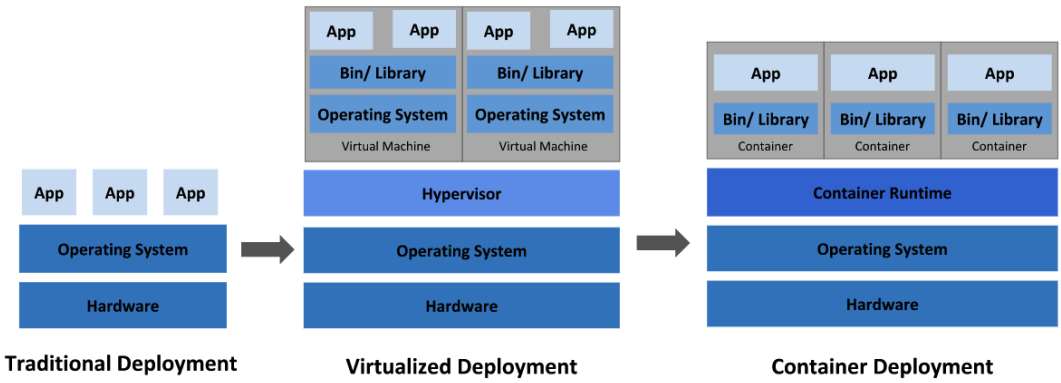
Näkyvyys tämän tutkimuksen kontekstissa on rajattu julkaisu- ja hallintajärjestelmän tilaa indikoiviin tapahtumiin, joita järjestelmästä ei saada perinteisin keinoin kerättyä. Moniklusterisuus tarkoitti tutkimuksen toteuttamisen kannalta sitä, että tapahtumia jouduttaisiin louhimaan, joko keskitetystä lähteestä tai klusteritasolla.

2 PALVELUTUOTANTO JA KONTTITEKNOLOGIAT

Konttiteknologiat ja konttien orkestrointijärjestelmät, kuten Kubernetes, ovat syntyneet vastaamaan modernin palvelutuotannon tarpeita. Palveluiden kuluttajien määrä on kasvanut, mikä on asettanut palveluille uusia vaatimuksia, kuten luotettavuus, skaalautuvuus ja saavutettavuus. Nämä kasvaneet vaatimukset ovat johtaneet liikehdintään pois perinteisistä monoliittisista ympäristöistä, horisontaalisesti skaalautuviin hajautettuihin järjestelmiin. Hajautetut järjestelmät tuovat mukanaan myös muita etuja, kuten mahdollisuuden hyödyntää ketterän ohjelmistokehityksen malleja, nopeammat julkaisusyklit ja palveluiden päivittämisen ilman käyttökatkoja eli rullaavat päivitykset. Hajautetut järjestelmät tuovat mukanaan myös suuren määrän monimutkaisuutta ja haasteita, joihin vastaukseksi on kehitetty konttien orkestrointi- ja hallintajärjestelmät.

2.1 Palvelutuotanto ja hajautetut järjestelmät

Palvelutuotannon kehitys voidaan jakaa julkaisuprosessin ja -alustan mukaan fyysisille palvelimille julkaisemisen aikakauteen, virtuaalikoneille julkaisemisen aikakauteen ja konttien julkaisemisen aikakauteen. Näiden eri julkaisumallien käsittely tarjoaa teknisestä näkökulmasta kuvan palvelutuotannon kehityksestä. Kuviossa 1 on esitelty eri julkaisuprosesseissa hyödynnetyt alustat ja niiden rakenne. (Cloud Native Computing Foundation 2022a.)



KUVIO 1. Palvelutuotannon kehityksen voi jakaa eri aikakausiin järjestelmäalustan mukaan. Kuvassa on esitelty fyysiset, virtuaalikone- ja konttijulkaisualustat (Cloud Native Computing Foundation 2022a):

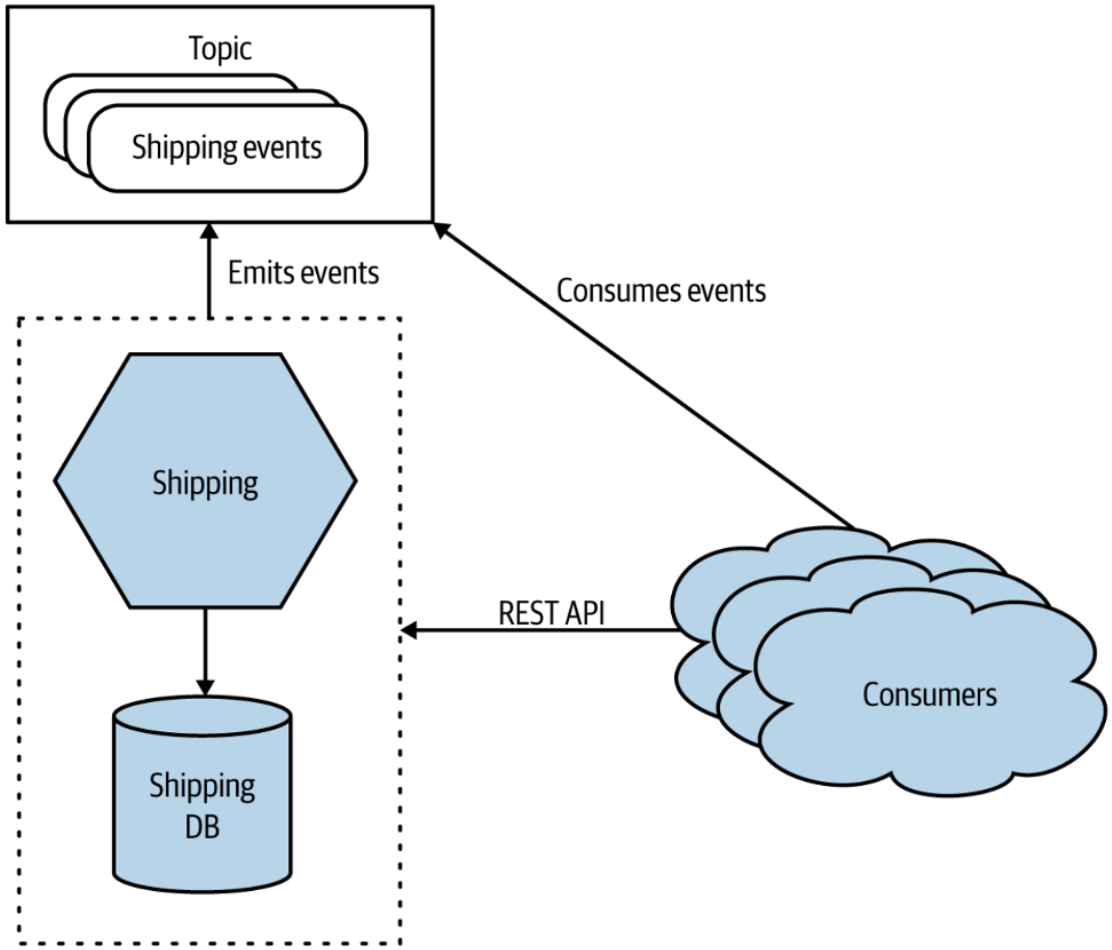
Fyysisille palvelimille julkaisemisen aikakaudella yritykset pyörittivät palveluitaan suoraan servereillä. Laitteistoresurssien jakaminen palveluiden välillä on haasteellista tällaisessa ympäristössä ja johtaa usein tilanteisiin, joissa joku yksittäinen ohjelma varaa eli allokoi suuren osan laitteiston resursseista. Tämän seurauksena muiden samalla palvelimella toimivien ohjelmien suorituskyky kärsii. Vaihtoehtona ja johdatuksena virtualisoinnin aikakaudelle resurssiongelma on mahdollista ratkaista fyysisillä palvelimilla hankkimalla jokaiselle ohjelmalle oman palvelimen. Tämän ratkaisun heikkouksia ovat kuitenkin sen skaalautuvuus, resurssien tyhjäkäynti, ylläpito ja kustannukset. (Cloud Native Computing Foundation 2022a.)

Julkaisu virtuaalikoneille ja virtualisointi konseptina kehitettiin vastaukseksi fyysisille palvelimille julkaisemismallin heikkouksiin. Virtuaalikone (engl. Virtual machine) on täydellisesti virtualisoitu ympäristö, joka on asennettu isäntäkoneen käyttöjärjestelmän päälle. Virtualisointi lisää palvelimen ja palvelun välille kerroksen, joka mahdollistaa tehokkaamman resurssien jaon. Esimerkiksi yhdellä palvelimella voidaan pyörittää kahta virtuaalikoneita, jotka jakavat isäntäkoneen yhden prosessorin ja kovalevyn. Palvelimen ylläpitäjä voi määritellä, miten resursseja jaetaan virtuaalikoneiden välillä. Virtuaalikoneiden avulla ympäristö on helposti skaalattavissa ja uusia palveluita on helppo lisätä. (Cloud Native Computing Foundation 2022a.)

Kontti on eristetty instanssi, joka virtuaalikoneiden tapaan pyörii palvelimen oman käyttöjärjestelmän päällä, kuitenkin sillä erotuksella, että kontit jakavat isäntäkoneen laiteresurssit, Kernelin ja käyttöjärjestelmän. Tästä syystä ympäristöjä, jotka hyödyntävät kontteja, kutsutaan myös kevyesti virtualisoiduiksi ympäristöiksi. Kontti sisältää kaikki riippuvuudet ja konfiguraation palvelun pyörittämiseksi ilman käyttöjärjestelmän ja Kernelin luomaa ylimääräistä taakkaa, mikä tekee niistä kevyitä ja helposti liikuteltavia. Konttien rakennetta käsitellään tarkemmin luvussa 2.2. Kontit helpottavat ja nopeuttavat projektien sisäänajoa kehittäjien päätteille ja vähentävät ympäristön konfiguraatiosta johtuvia eroavaisuuksia kehittäjillä. Kontit mahdollistavat palveluiden jakamisen vielä pienempiin osakokonaisuuksiin ja riippuvuuksien vähentämisen eri palveluiden välillä, erityisesti palveluiden ja infrastruktuurin välillä. Tämä palveluiden eristäminen pienemmiksi, riippumattomiksi osakokonaisuuksiksi on mikropalveluarkkitehtuurin tavoite. Mikropalvelut mahdollistavat entistä tiheämmän kadenssin julkaisujen välillä, koska julkaisuja voidaan tehdä pienemmissä osakokonaisuuksissa. Hajautetut järjestelmät ovat järjestelmiä, jotka koostuvat tällaisista palveluista. (Cloud Native Computing Foundation 2022a.)

2.1.1 Mikropalveluarkkitehtuuri ja monoliitit

Mikropalvelut (engl. Microservices) ovat pieniä ja autonomisia palveluita, jotka muodostavat yhdessä laajempia ohjelmakokonaisuuksia. Ne ovat riippumattomia toisistaan ja siten julkaistavissa itsenäisesti. Palvelut pitävät sisällään toiminnallisuuden, jota ne jakavat muille palveluille verkon kautta. Tällaisia palveluita samaan verkkoon lisäämällä voidaan rakentaa yhä monimutkaisempia kokonaisuuksia. Usein mikropalvelukokonaisuus mallinnetaan liiketoimintamallin pohjalta. Esimerkiksi tilausjärjestelmä voi koostua mikropalveluista, joista yksi hallinnoi varastoa, toinen vastaa tilauksen käsittelystä, ja kolmas vastaa toimituksista. Kuviossa 2 on esimerkki, kuinka mikropalveluarkkitehtuuria voisi hyödyntää rahtaustapahtuman käsittelyssä. (Newman 2021, luku 1.)



KUVIO 2. Käyttäjä odottaa vastauksena rahtaustapahtumaa, joka voi olla esimerkiksi selaimen ikkunassa näytettävä sisältö. Kutsu välitetään REST-rajapinnan kautta suoraan mikropalvelulle, joka palauttaa tiedon rahtaustapahtumana kuluttajalle (Newman 2021, luku 1).

Mikropalvelut ovat erityisesti arkkitehtuurillinen valinta, joka tarjoaa useita eri vaihtoehtoja palvelukokonaisuuden rakentamiseen ja muovaamiseen (Newman 2021, luku 1). Mikropalveluarkkitehtuuri on vaihtoehto perinteiselle monoliittiselle arkkitehtuurille.

Monoliittinen arkkitehtuuri on helppo ymmärtää ajattelemalla perinteistä työpöytäohjelmaa, joka asennetaan suoraan käyttäjän omalle tietokoneelle. Tämä ohjelma on kokonaisuus, joka sisältää koko ohjelman muodostavan lähdekoodin ja moduulit paketoituna ja tarjoiltuna yhden suoritettavan tiedoston (engl. executable) kautta. Selkeimmän vertailukohdan monoliitin ja mikropalvelun välille tarjoaa palvelun julkaiseminen, koska monoliitin kaikki toiminnallisuudet täytyy julkaista yhtenä kokonaisuutena samalla kerralla. Monet eri ohjelmointimallit ja -arkkitehtuurit ovat tyypiltään monoliittisiä, joten niissä ei ole kyse mistään tietystä ohjelmointiparadigmasta vaan lopputuotteen ominaisuuksista ja luonteesta.

2.1.2 Palveluiden julkaisu ja GitOps

Automatisoidut julkaisuputket ovat vastaus nopeamman julkaisutiheyden saavuttamiseksi ja yhä suurempien palvelukokonaisuuksien monimutkaisuuden hallitsemiseksi. Automatisoidun ratkaisun avustuksella voidaan taata, että julkaisut suoritetaan jokaisella kerralla samalla tavalla. Tällaisia automatisoituja kokonaisuuksia kutsutaan CICD-järjestelmiksi tai -putkiksi.

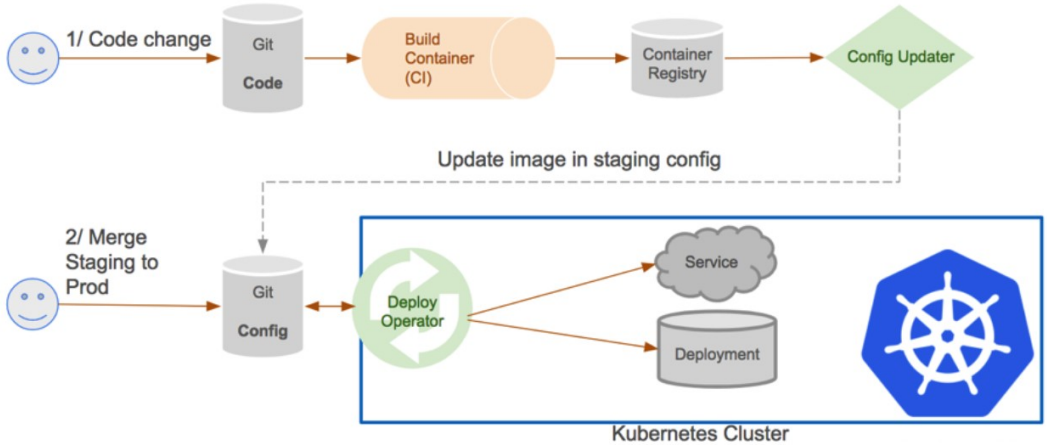
CI/CD

Jatkuva integraatio (engl. Continuous integration) eli CI on versiohallinnan ympärille rakennettu automaatio, jossa lähdekoodiin tehty muutos testataan matalan tason testinipuilla. Jos muutos läpäisee testit, sen lähdekoodi käännetään ja julkaistaan esimerkiksi kuvauskantaan (engl. repository). CD eli jatkuva julkaiseminen (engl. Continuous deployment) tai jatkuva toimitus (engl. Continuous delivery) tarkoittaa sovelluksen tai palvelun julkaisemista kohdeympäristöön. Se ottaa CI-vaiheessa rakennetun sovelluksen ja julkaisee sen ympäristöön, jossa voidaan testata suurempia kokonaisuuksia. Yleensä julkaisut kulkevat eri integraatiotasojen läpi, joissa sovellusta testataan. Kun sovellus läpäisee testit, se siirretään seuraavalle integraatiotasolle, kunnes se saavuttaa tuotannon, jossa se on loppukäyttäjien saatavissa. Testiniput voivat sisältää toiminnallista ja ei-toiminnallista laadunvarmistusta. Jatkuva julkaisu on täysin automatisoitu julkaisu, kun taas jatkuva toimitus on aina valmis julkaistavaksi, mutta sitä ei julkaista automaattisesti. (Red Hat 2018b.)

GitOps

GitOps on CICD-julkaisuprosessiin liittyvä malli, jossa Git-versiohallintaa pidetään yksittäisenä järjestelmän tilan totuuden lähteenä. Tila on esitetty deklaratiivisesti, eli määrittelemällä erillisiin dokumentteihin, minkälainen on ympäristön toivottutila. GitOpsin ytimessä toimii siis kaksi tilaa: järjestelmälle toivottutila (engl. desired state) ja nykytila (engl. current state). Järjestelmä tarkkailee toivottua tilaa ja ylläpitää nykytilaa: jos nykytila poikkeaa toivotusta tilasta, järjestelmä pyrkii sovittelemaan nykytilan vastaamaan toivottua tilaa. Tilamääritelmää säilytetään aina versiohallinnassa, joten samalla järjestelmän tilasta jää historia- ja muutostietoa. (Weave works 2022.)

Kuviossa 3 on esitelty GitOps-julkaisuputken toimintaa. Kuvion yläosassa on esitetty jatkuvan integraation osio julkaisuputkesta, jossa muutos lähdekoodiin julkaistaan versiohallintaan, joka johtaa aplikaation rakentamiseen. Applikaatiokontti siirretään konttirekisteriin, jonka jälkeen ”Config Updater”-prosessi päivittää versiohallinnan konfiguraatiotiedoston. Julkaisu- ja hallintajärjestelmän Operaattori seuraa muutoksia konfiguraation tilassa ja sovittelee ne julkaisemalla konfiguraatiossa määritellyn kontin. (Weave works 2022.)



KUVIO 3. Tyypillinen GitOps julkaisuputki (Weave works 2022.)

GitOps on kehitetty Kubernetes-klusterien hallinnointia ja applikaatioiden julkaisua sinne varten. GitOps-liike ei kuitenkaan enää rajoitu vain Kubernetes alustaan. (Weave works 2022.)

2.1.3 Näkyvyys

Näkyvyys (Observability) on yksi järjestelmän ominaisuus, kuten luetettavuus (engl. reliability), skaalautuvuus (engl. scalability) ja turvallisuus (engl. security). Näkyvyys tulisi ottaa huomioon jokaisessa järjestelmän elinkaaren vaiheessa aina suunnittelusta kehitykseen ja testaukseen. Näkyvyys rakennetaan yhdistelemällä eri lähteistä saatavaa tietoa järjestelmän tilasta. Nämä lähteet ovat tapahtumalokikirjanpito (engl. event logs), metriikka (engl. metrics) ja jäljitystiedot (engl. tracing). Näkyvyys auttaa järjestelmän ylläpitäjää arvioimaan julkaistujen muutoksien vaikutusta järjestelmään ja sen suorituskykyyn. Suorituskyky tässä yhteydessä kattaa myös edellä mainitut ominaisuudet eli luotettavuus, skaalautuvuus ja turvallisuus. Näkyvyys antaa siis järjestelmän ylläpitäjälle silmät järjestelmän sisäiseen tilaan ja elinkaareen. (Yuen ym. 2021, luku 8.1.)

Tapahtumalokikirjanpito

Tapahtumalokit ovat ohjelmien tuottamaa dataa ohjelman sisäisestä suoritusaikaisesta tilasta. Jokaisen tapahtumalokin tulisi olla aikaleimattu ja muuttumaton. Tapahtumalokit tarjoavat hyvin hienojakoista tietoa ohjelman tilasta, ja ne ovat korvaamaton apu ongelmanratkaisutilanteissa. Tapahtumalokien kääntöpuolena on lokituksen taso, koska liian kattavalla ohjelmatason lokeilla on sivuvaikutuksia, kuten levytilan täyttyminen ja ohjelman suorituskyvyn heikentyminen. Lokien hallintaan, selaamiseen ja keräämiseen käytetään yleensä yhtä keskitettyä lokien hallintajärjestelmää, kuten Splunk ja Elasticsearch. (Yuen ym. 2021, luku 8.1.1.)

Metriikka

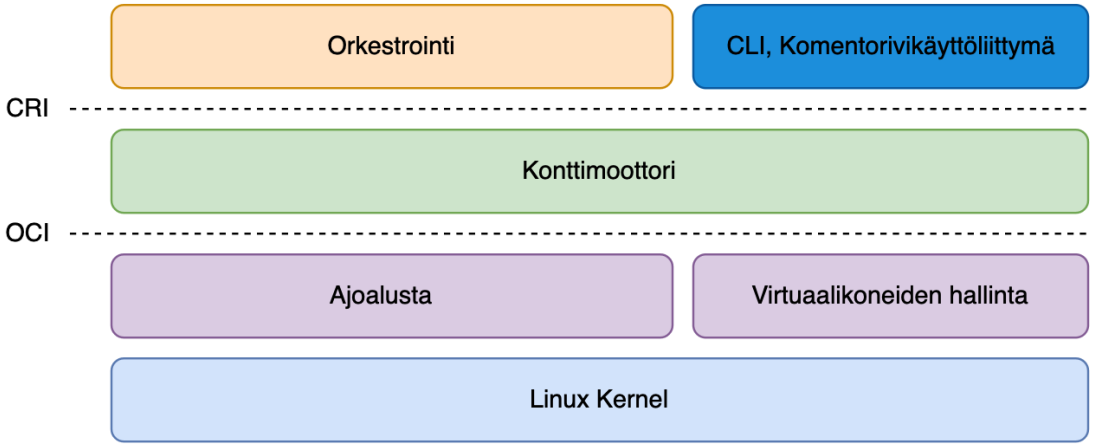
Metriikka on yksinkertaisimmillaan avain-arvolista, joka sisältää tietoa järjestelmän suorituskyvystä ja järjestelmäresurssien hyödyntämisestä. Keskeisimmät metriikan määreet ovat suoritinkäyttö, muistikäyttö, kovalevytila, kovalevyn käyttö ja verkon käyttö. Metriikkaa pystytään normaalisti esittämään järjestelmä- ja järjestelmäkomponenttitasolla sekä ohjelmatasolla. Metriikalla voidaan esittää myös tiettyjen virhetyyppien määrää tai jonon sisältöä. (Yuen ym. 2021, luku 8.1.2.)

Jäljitystieto

Tyypillisesti hajautettujen järjestelmien jäljitystieto vaatii ohjelmakohtaisesti oman prosessin, joka seuraa oikeita ohjelmiston suorituspolkuja tiedon tuottamiseksi. Jäljitystieto on erittäin arvokasta hajautetuissa mikropalveluarkkitehtuuriin perustuvissa järjestelmissä, joissa käyttäjän tekemä kutsu saattaa liikkua kymmenien, ellei jopa satojen eri palveluiden kautta. (Yuen ym. 2021, luku 8.1.3.)

2.2 Kontit

Monet konttiteknologioiden kuluttajat eivät ymmärrä konttien rakennetta, ja osa konttiteknologioiden taiasta onkin juuri siinä, ettei käyttäjän tarvitse ymmärtää kontteja nauttiakseen niiden tuomista hyödyistä. Tämän työn kannalta on kuitenkin ehdottoman tärkeää, että lukija ymmärtää kontteja ja konttiteknologiaa hieman syvällisemmin. Tässä osiossa käsitellään syventävästi Linuxin tarjoamaa teknologiapinoa, jonka päälle konttiteknologiat on rakennettu. Kuviossa 4 on esitelty korkealla tasoilla konttien käytössä tarvittavat komponentit.



KUVIO 4. Kontin rakenne koostuu karkealla tasolla seuraavista komponenteista. Virtuaalikoneiden hallinta liittyy vain teknologiapinoihin, joiden tarkoitus on tarjota normaalia parempaa eristystä konteille ja ajettaville työkuormille.

Kontti on, tiivistettynä yhteen virkkeeseen, nippu prosesseja, jotka toimivat yhdellä palvelimella tai isäntäkoneella omana eristettynä ryhmänä ja toteuttavat loogisen kokonaisuuden eli yhden tai useampia toimintoja, ympäristön tai palvelun. Konteilla on oma tiedostojärjestelmä, verkko, käyttäjä, käyttäjäryhmä, prosessit ja prosessiryhmä. Tämä luo käyttäjälle helposti illuusion, että kontti olisi jotenkin irrallaan isäntäkoneen käyttöjärjestelmästä. Totuus kuitenkin on, että kontin täytyy jakaa resurssinsa isäntäkoneen kanssa, ja se onkin siten ennemmin käyttöjärjestelmän sisäinen kapseli kuin käyttöjärjestelmästä irrallaan oleva instanssi. (Grunert 2019a.)

2.3 Konttiteknologian juuret

Konttien toteutuksessa näkyy läpileikkaus uusia ja vanhoja Linux Kernelin toiminnallisuuksista. Tällaisia toiminnallisuuksia on esimerkiksi Chroot, määrittele juurihakemisto, joka on ollut Kernelissä jo 1980-luvulta asti, kun taas uudemmat toiminnollisuudet, kuten kontrolliryhmät (cgroups) ja nimiavaruudet (namespace) on lisätty Kerneliin vasta 2000-luvulla.

2.3.1 Nimiavaruus

Nimiavaruus (engl. Namespace) kapseloi käyttöjärjestelmän laajuisen resurssin niin, että nimiavaruuden sisällä vaikuttaa, että resurssi kuuluu yksinomaan kyseiselle nimiavaruudelle. Resurssin muutokset ovat näkyviä muille saman nimiavaruuden prosesseille, mutta näkymättömiä nimiavaruuden ulkopuolelle. Nimiavaruus koostuu eri nimiavaruustyypeistä, nimiavaruusrajapinnasta (engl. Namespace API) ja omasta hakemistojärjestelmästä. Nimiavaruuden tarjoama prosessiryhmien ja ohjelmaresurssien eristys on kaiken konttiteknologian perusta. Seuraavaksi esitellään nimiavaruusrajapinta ja järjestelmäkomennot (syscalls), joilla voidaan manipuloida nimiavaruuksia. (Kerrisk 2022.)

Nimiavaruusrajapinta

Nimiavaruusrajapinta sisältää seuraavat käyttöjärjestelmäkutsut (syscalls): kloonaa (clone), aseta nimiavaruus (setns), eriytä (unshare), siirrännän hallinta (ioctl). Näiden kutsujen tehtävänä on käsitellä nimiavaruuksia. Järjestelmäkutsuja on helpompi ajatella funktioina, joilla luodaan tai käsitellään nimiavaruustyyppejä. Seuraavaksi lyhyt esittely eri kutsuista, tarkemmin kutsut on dokumentoitu Linux-manuaalissa. (Kerrisk 2022.)

Kloonaa

Kloonaa (clone) luo uuden prosessin, jolle voidaan antaa argumenttina yksi tai useampi CLONE_NEW*-lippu. Se luo jokaiselle lipulle oman nimiavaruuden ja lapsiprosessin. (Kerrisk 2022.)

Aseta nimiavaruus

Aseta nimiavaruus (setns) liittää sitä kutsuvan prosessin jo olemassa olevaan nimiavaruuteen. Nimiavaruus määritellään tiedoston määrittelijässä (engl. file descriptor), joka viittaa proc-hakemistoon. (Kerrisk 2022.)

Eriytä

Eriytä (unshare) liittää sitä kutsuvan prosessin uuteen nimiavaruuteen. Kuten clone-kutsussa, jos CLONE_NEW*-lippuja on määritelty useita, niille jokaiselle luodaan oma nimiavaruus ja kutsuva prosessi liitetään niihin kaikkiin. (Kerrisk 2022.)

Siirrännän hallinta

Siirrännän hallinnan (ioctl) avulla voidaan määritellä sääntöjä nimiavaruuksien välisiin suhteisiin ja nimiavaruuden näkyvyyteen järjestelmässä. (Kerrisk 2022.)

2.3.2 Nimiavaruustyypit

Nimiavaruustyyppejä voitaisiin ajatella rakenteina tai kaavoina, joissa määritellään rakenteen ominaisuudet, sijainti järjestelmässä, määreet ja monia muita rakenteellisia ominaisuuksia. Ne ovat eristettyjen ympäristöjen konkreettisia rakennuspalikoita, joiden avulla konteille voidaan allokoida käytössä olevia järjestelmäresursseja, kuten muisti ja prosessorikapasiteetti. Nimiavaruustyyppi voi määrittää myös esimerkiksi ip-osoitteet, jotka kontilla on käytettävissä. Seuraavaksi tiivistelmä käytettävistä nimiavaruustyypeistä ja niiden funktioista. (Grunert 2019a.)

Liitos

Liitos (mnt) tarkoittaa isäntäkoneen tiedostojärjestelmästä varattuja liitoskohtia, joita nimiavaruudessa toimiva prosessi voi hyödyntää. Tiedostojärjestelmällä on oma juuri, jonka päälle muut hakemistot on sijoitettu samalla tavalla kuin tyypillisessä Linux-tiedostojärjestelmissä. Liitosta voidaan ajatella samantapaisena tuotteena, jonka chroot-komennolla syntyy, eli eristettynä tiedostojärjestelmänä, jolla on oma juuritason käyttäjä. Juuritason käyttäjän oikeudet rajoittuvat määritellyn liitoksen tiedostojärjestelmän sisälle. Isäntäjärjestelmässä nämä liittymispisteet on kapseloitu virtuaalinen tiedostojärjestelmä - nimisen (VFS) abstraktiokerroksen sisälle. VFS on osa Kerneliä,

ja sitä käytetään eristämään kaikki tiedostojärjestelmät isäntäkoneen sisällä, myös isäntäjärjestelmän oman. (Grunert 2019a.)

UTS-nimiavaruus

UTS-nimiavaruus (UNIX Time-sharing System) mahdollistaa verkkoalueen (engl. domain) ja verkkoaseman tunnuksen (engl. hostname) eristämisen isäntäkoneesta. Tätä ominaisuutta käytetään erityisesti konttien verkoissa ja niiden konfiguroinnissa hyödyksi. (Grunert 2019a.)

IPC-nimiavaruus

IPC-nimiavaruus (Interprocess Communication) mahdollistaa prosessien välisen kommunikaation hallinnoimisen. Käytännössä nämä ovat System V IPC -objekteja ja POSIX-viestijonoja. Esimerkki käyttötapaus IPC:n hyödyntämisestä on jaetun muistin (SHM) erottaminen kahdelle eri prosessille niin, että muistialueen väärinkäyttö estetään. (Grunert2019a.)

PID-nimiavaruus

PID-nimiavaruus mahdollistaa oman prosessi-identifiointi-ryhmän käytön nimiavaruuden sisällä. Eristetyillä prosesseilla on omat prosessitunnuksensa nimiavaruuden ulkopuolella, eli järjestelmätasolla ja sisäpuolella. Taulukossa 1 havainnollistetaan, kuinka prosessilla on eri prosessitunnus nimiavaruuden sisäpuolella ja ulkopuolella, eli isäntäjärjestelmässä. Kontin sisäisessä nimiavaruudessa, prosesseilla on prosessitunnukset 1 ja 2, nimiavaruuden ulkopuolella prosesseilla on isäntäjärjestelmän nimiavaruuden prosessitunnukset 3 ja 4. (Grunert 2019a.)

TAULUKKO 1. Taulukko havainnollistaa prosessitunnuksien eristystä isäntä- ja lapsinimiavaruuksien välillä.

Nimiavaruus	**tunnus:prosessi-nimi**	**tunnus:prosessi-nimi**	**tunnus:prosessi-nimi**	**tunnus:prosessi-nimi**
Isäntäjärjestelmä	1:init	2:sysd	3:init_kontti	4:web-server.py
Kontti			1:init	2:web-server.py

Verkkonimiavaruus

Verkkonimiavaruus (engl. network namespace) mahdollistaa verkkopinon virtualisoinnin. Jokaisella nimiavaruudella on oma verkkopinonsa, joka sisältää ip-osoitteet, reititystiedot, yhteyksien jäljitystiedot, palomuurin määritykset ja muita verkkomäärityksiä. Verkkonimiavaruutta käytetään esimerkiksi Flannel -ohjelmassa, joka on yksi Kubernetesin verkkokomponenteista. Flannel luo ohjelmistolle määritellyn verkon (SDN) käyttämällä virtuaalitietoverkkoparia (veth), joista se kytkee toisen liittymän verkkosiltaan (engl. bridge) ja sen parin halutulle kontille. Verkkosilta voi avata kontille yhteyden esimerkiksi isäntäkoneelle, toiselle kontille tai vapaaseen internetiin. SDN on erityisesti kapselien sisäisessä verkkoliikenteessä hyödynnetty verkkotekniikka. Kapselit ovat Kubernetesin resursseja, joilla voidaan niputtaa useampi yksittäinen kontti samaan eristettyyn ympäristöön. (Grunert 2019a.)

Käyttäjänimiavaruus

Käyttäjänimiavaruus (user namespace) mahdollistaa käyttäjien ja käyttäjäryhmien määrittelemisen nimiavaruuteen. Käyttäjänimiavaruus mahdollistaa samalla käyttäjällä tai käyttäjäryhmällä eri käyttöoikeudet nimiavaruuden ulko- ja sisäpuolella. Käyttäjällä voi olla näin kontin sisällä root-oikeudet, mutta ulkopuolella suppeammat käyttöoikeudet. (Grunert 2019a.)

Kontrolliryhmät

Kontrolliryhmät (cgroups) mahdollistavat resurssien eli prosessien tai prosessiryhmien hallitsemisen. Kontrolliryhmällä voidaan siis rajoittaa, priorisoida ja seurata resurssien käyttöä nimiavaruuden sisällä, ja kontrolliryhmälle voidaan määritellä sillä käytettävissä olevat järjestelmäresurssit, kuten muisti- ja prosessoriosiot. Yksi viimeisimmistä kontrolliryhmän toteutuksista on OOM (Out Of Memory killer), joka mahdollistaa kontrolliryhmän tuhoamisen yhtenä entiteettinä, jolla pyritään varmistamaan, että yksittäisen suoritettavan työkuorman integriteetti säilyy. Merkittävimmät kontrolliryhmät on listattu taulukkoon 2. Kontrolliryhmää pidetään merkittävimpänä konttiteknologioiden mahdollistajana. Kontrolliryhmät ovat myös nimiavaruuden tyyppi ja täten osa nimiavaruuskonseptia. (Grunert 2019a.)

TAULUKKO2. Seitsemän merkittävintä kontrolliryhmää (Baier yms. 2019).

Kontrolliryhmä	Tehtävä
Memory cgroup	Seuraa kontrolliryhmän muistinkäyttöä ja sen avulla voidaan määritellä rajoitukset fyysisen muistin, Kernel-tason muistin ja kokonaismuistin käytölle.
Blkio cgroup	Seuraa ja hallinnoi kontrolliryhmäkohtaista tietoa lohkolaitteiden (engl. Block devices) I/O:n käytön osalta.
CPU cgroup	Seuraa käyttäjän ja järjestelmän prosessorikapasiteetin käyttöä.
Freezer cgroup	Aikatauluttaa resursseja, työkuormille, jotka vaativat prosessien pysäytystä ja käynnistystä.
CPUset cgroup	Mahdollistaa prosessin tai applikaation kiinnittämisen tiettyihin prosessoriytimiin.
Net_cls/net_prio cgroup	Merkkaa kontrolliryhmän verkkoliikenteen (net_cls), jonka avulla voi suorittaa liikenteen priorisointia (net_prio).
Devices cgroup	Hallitsee mitä kirjoitus- ja lukuoikeuksia kontrolliryhmällä on laitteisiin.

2.3.3 Proc-hakemisto

Proc-hakemisto on nimiavaruuden sijainti isäntäkäyttöjärjestelmässä. Nimiavaruuteen liittyviä määrityksiä voidaan suoraan tarkastella proc-hakemistosta. Esimerkiksi kuviossa 5 on esitelty, kuinka ”/proc/$PID/ns”-hakemiston avulla on mahdollista tarkastella, missä nimiavaruudessa prosessit ovat. Tässä esimerkissä ”$PID” on haettavan prosessin prosessitunnus.

```
> ls -Gg /proc/self/ns/
total 0
lrwxrwxrwx 1 0 Feb  6 18:32 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 0 Feb  6 18:32 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 0 Feb  6 18:32 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 0 Feb  6 18:32 net -> 'net:[4026532008]'
lrwxrwxrwx 1 0 Feb  6 18:32 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 0 Feb  6 18:32 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 0 Feb  6 18:32 user -> 'user:[4026531837]'
lrwxrwxrwx 1 0 Feb  6 18:32 uts -> 'uts:[4026531838]'
```

KUVIO 5. Jokaisen prosessin nimiavaruus voidaan löytää proc-hakemistosta.

Isäntäkoneen proc-hakemiston kautta löytyvät kaikki isäntäkoneella pyörivät prosessit. Tämän ansiosta proc-hakemistoa voidaan käyttää hyödyksi esimerkiksi vianselvitystehtävissä. Kontit ovat eristettyjä järjestelmiä isäntäjärjestelmän sisällä, eivätkä ne ehkä olekaan niin irrallaan isäntäjärjestelmästä kuin monet luulevat. (Grunert 2019a.)

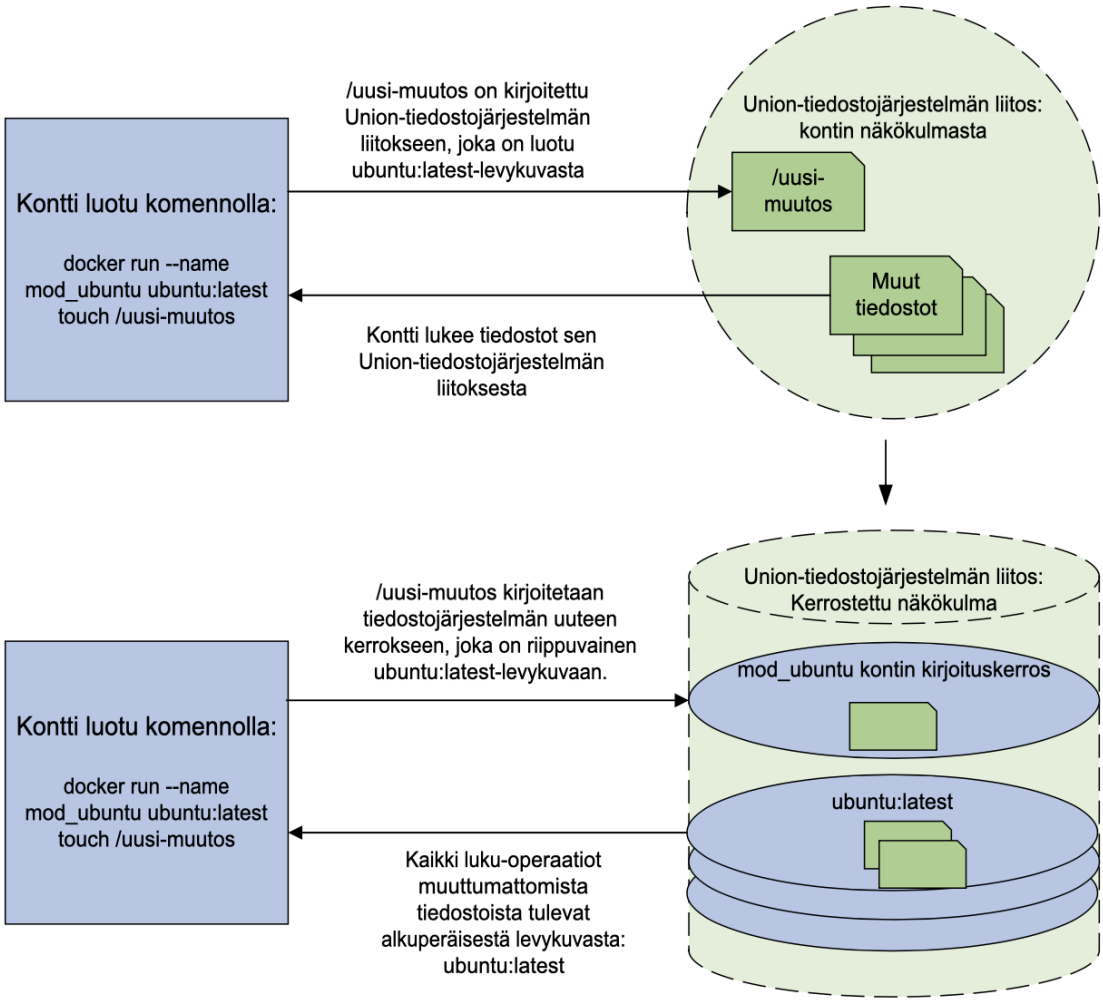
2.3.4 Konttien pakkaus ja tiedostojärjestelmä

Docker oli ensimmäinen konttipalvelutarjoaja, joka kehitti konttien pakkaamisen. Käyttäjät voisivat liikutella kontteja vapaasti levykuvan (engl. Image) muodossa päätteeltä toiselle. Tämä merkitsi myös konttipohjaisten julkaisujen syntyä. Docker loi alkuperäisestä skeemasta kaksi erillistä versiota, joista toinen lahjoitettiin OCI, Open Container Initiative, OCI-levykuvastandardin pohjaksi. (Grunert 2019b.)

Levykuva on tallenne kontin tiedostojärjestelmästä ja sen tilasta tietyllä ajan hetkellä (Baier ym. 2019). Kontin levykuva koostuu yhdestä tai useasta kerroksesta, jotka ovat keskenään vanhempi-lapsisuhteessa (engl. Parent-child relation). Jokainen levykuvan kerros edustaa muutosta vanhempi- ja lapsikerroksen välillä. Konttien tietojärjestelmät ja levykuvat hyödyntävät Union-tiedostojärjestelmää. (Nikoloff & Kuenzli 2019, luku 7.2.1.)

Union-tiedostojärjestelmä

Union-tiedostojärjestelmä (engl. Union filesystem) koostuu kerroksista. Aina kun tiedostojärjestelmään tehdään muutos, järjestelmä luo uuden kerroksen kaikkien muiden kerrosten päälle. Kerrostetun tiedostojärjestelmän muutosta on esitelty kuvan 6 alaosassa, jossa kontin tiedostojärjestelmään kirjoitetaan uusi-muutos-tiedosto ”touch”-komennolla. Kaikkien näiden kerroksien yhdistelmä muodostaa tiedostojärjestelmän, jonka kontti ja kontin käyttäjät näkevät. Kontin ja käyttäjän näkökulma tiedostojärjestelmään on havainnollistettu kuvion 6 yläosassa. (Nikoloff & Kuenzli 2019, luku 7.2.1.)



KUVIO 6. Ylemmässä kuvassa on Union-tiedostojärjestelmän liitos, kuten kontti ja käyttäjä sen näkevät. Alemmassa kuvassa on esitelty, kuinka Union-tiedostojärjestelmä on kerrostettu ja miten tiedostojärjestelmään kohdistuvat muutokset kirjoitetaan omaan kirjoituskerrokseensa (Nikoloff & Kuenzli 2019, luku 7.2.1).

Kun tiedosto luetaan Union-tiedostojärjestelmästä, se luetaan aina ylimmästä kerroksesta, jossa tiedosto on olemassa. Muutokset tiedostoihin, kuten tiedoston muokkaus ja poisto, kirjoitetaan myös aina tiedostojärjestelmän ylimpään kerrokseen. Tiedostojärjestelmän ylin kerros on kirjoituskerros ja sen alapuolella olevat kerrokset ovat lukukerroksia, joihin ei voi kirjoittaa. Union-tiedostojärjestelmät käyttää tekniikkaa nimeltä kopio-kirjoitettaessa (engl. copy-on-write). Tämä tarkoittaa, että koko lukukerros kopioidaan kirjoituskerrokseen, kun tiedostoa, joka on lukukerroksessa, muokataan. (Nikoloff & Kuenzli 2019, luku 7.2.1.)

OCI-standardi

OCI-standardi esiteltiin vuonna 2015. Standardin tarkoitus oli yhtenäistää konttien pakkaus ja konttien ajoalusta. Standardi määrittää tavan toteuttaa konttien pakkaus ja ajoalusta niin, että kaikki halukkaat voisivat implementoida ne. Tässä vaiheessa on hyvä muistaa, että suuret teknologiayritykset, kuten Google ja Amazon, olivat hyödyntäneet konttiteknologiaa paljon ennen kuin Docker toi teknologian kaikkien saatavaksi. Docker oli keskeisessä roolissa OCI-standardin tuottamisessa ja antoi muun muassa oman ajoalustansa, runC, ja pakkausformaatin, appC, standardin malleiksi. Pakkaus- ja ajoalustateknologiat ovat nykyään enemmän tai vähemmän OCI-standardin mukaisia, mutta tällä hetkellä trendi on kuitenkin se, että teknologioita muutetaan seuraamaan standardia aktiivisesti. Tämä tietenkin johtuu siitä, että myös orkestrointialustat ja niiden rajapinnat on rakennettu käsittelemään OCI-standardin mukaisia rakenteita. (Grunert 2019b.)

2.3.5 Konttien ajaminen

Kontin käynnistäminen on hyvin samankaltainen tapahtuma kuin minkä tahansa Linux-prosessin käynnistäminen. Tässä kohtaa on hyvä pitää mielessä, että käynnissä oleva kontti ei eroa normaalista käynnissä olevasta prosessista. Konttimoottorin (engl. Container engine) tehtävä on käsitellä käyttäjältä tai orkestrointialustan rajapinnalta tulevia komentoja, ladata konttien levykuvat levykuvarekisteristä, purkaa levykuvat ja järjestää ne levylle, valmistella tiedostojärjestelmän liitospisteet (mnt) isäntäkoneelle ja käsitellä kontin ajoalustalle siirrettävä metadata. Konttien ajamista varten on oma prosessinsa, jota kutsutaan kontin ajoalustaksi (engl. Container runtime). Konttimoottori keskustelee suoraan ajoalustan kanssa. (Red Hat 2018a.)

Kontin ajoalusta

Kontin ajoalusta (engl. Container runtime) on matalamman tason komponentti, jota tyypillisesti konttimoottori hyödyntää. Ajoalusta käsittelee tiedostojärjestelmän liitokset (mnt) ja metadatan, jotka sille välittää konttimoottori. Ajoalustan tehtäviä on myös ottaa käyttöön kontrolliryhmät, SE-Linux-käytännöt ja App Armor -säännöt sekä lähettää tarvittavat järjestelmäkutsut isäntäjärjestelmän Kernelille kontin käynnistämiseksi. (Red Hat 2018a.)

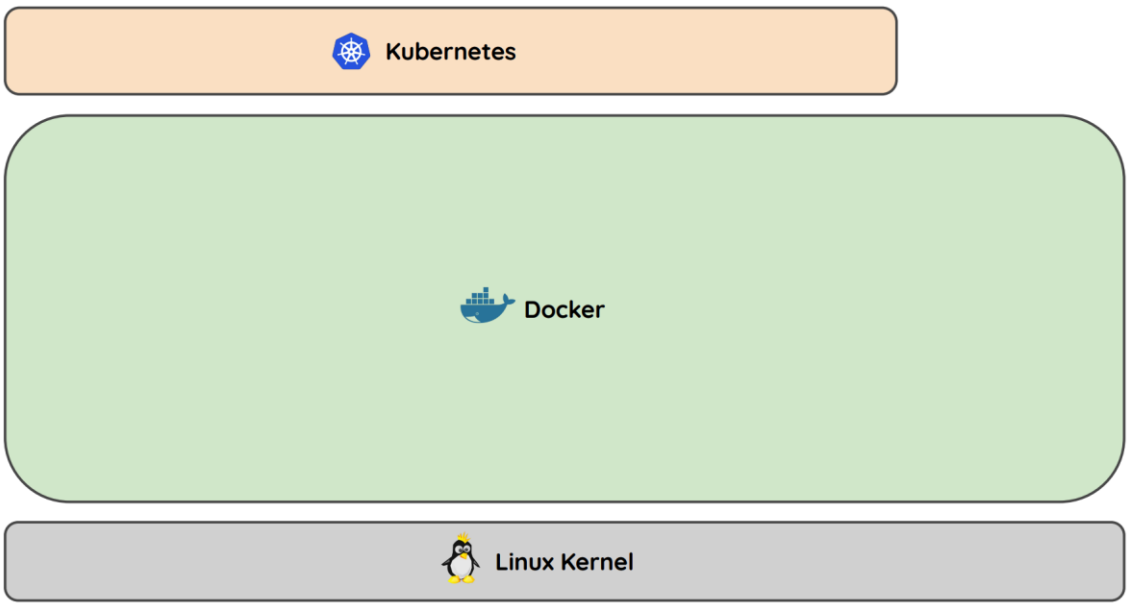
CRI-standardi

Kontin ajoalustan käyttöliittymä (engl. Container runtime interface), CRI, on Kubernetesiin kehitetty rajapinta, joka määrittelee, kuinka kontin ajoalusta voi keskustella Kubernetesin kanssa. CRI siis määrittelee, miten kontteja voi luoda ja tuhota Kubernetes-klusterissa. Tämä on CNCF:n hyväksymä projekti, joka sai alkunsa Podman-konttiteknologiapinon Kubernetes käyttöönottoprojektista. Koska Podmanin implementointi Kubernetesiin vaati niin suuria muutoksia Kubernetesin ydinkomponentteihin, Kubelet, päätettiin luoda rajapinta ja -säännöt, jotka ajoalustan rajapintakutsujen tulee toteuttaa. (Alibaba Cloud 2020.)

2.4 Konttiteknologiapinot ja niiden kehitys

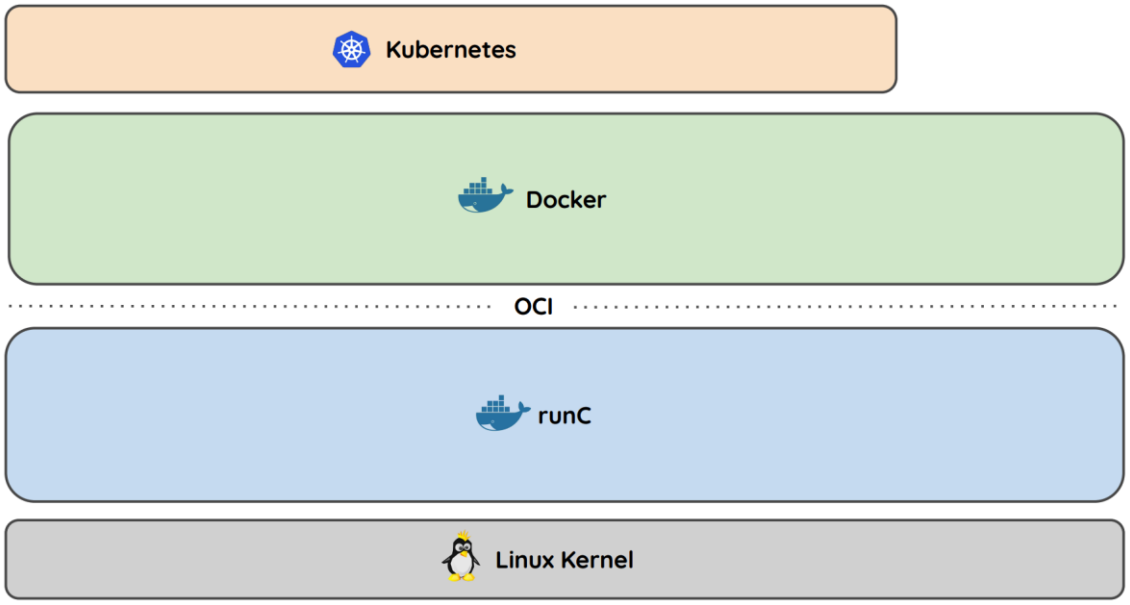
Tämä osio punoo yhteen konttiteknologian osa-alueita, joita edellä on käsitelty. Teknologianipuilla on omat käyttökohteensa ja tarkoituksensa. Uudet standardit, kuten OCI ja CRI, ovat mahdollistaneet konttiteknologiakentän kasvamisen. Lähestytään teknologiapinoja seuraamalla niiden kehitystä. Kuviot 7–10 helpottavat visualisoimaan muutokset konttiteknologia pinossa.

Aluksi oli vain Kubernetes, Linux Kernel ja Docker. Vuonna 2013 julkaistiin ensimmäinen versio Dockerista, joka sisälsi itsenäisen ajoalustan. Kaikki toiminnot konttien hyödyntämiseen oli pakattu yksiin kuoriin eli Docker:iin. Docker:in ensimmäinen versio hyödynsi LXC:n konttiajoalustaa, josta Docker myöhemmin luopui, koska se ei tarjonnut tarpeeksi optioita konttien eristyksen hienosäätämiseen. (Carrez 2020.)



KUVIO 7. Ensimmäinen versio teknologiapinosta sisälsi vain Dockerin, Kubernetesin ja Linux Kernelin (Carrez 2020).

Vuoden 2015 OCI-standardin myötä konttien ajoalusta irrotettiin konttimoottorista, ja samalla standardiin luotiin yhtenevä säännöstö konttien pakkaamisesta. Tämä tarkoitti runC-ajoalustan syntymistä, joka oli C-kielellä koodattu hyvin suorituskykyinen ja yksinkertainen ajoalusta. Monet nykyiset konttien ajoalustat, kuten rkt ja containerD käyttävät runC-ajoalustaa pohjalla ja lisäävät sen ympärille oman toteutuksensa. RunC on Dockerin kehittämä ajoalusta, joka julkaistiin OCI-standardin myötä avoimena lähdekoodina toimimaan eräänlaisena malliajoalustana. (Carrez 2020.)

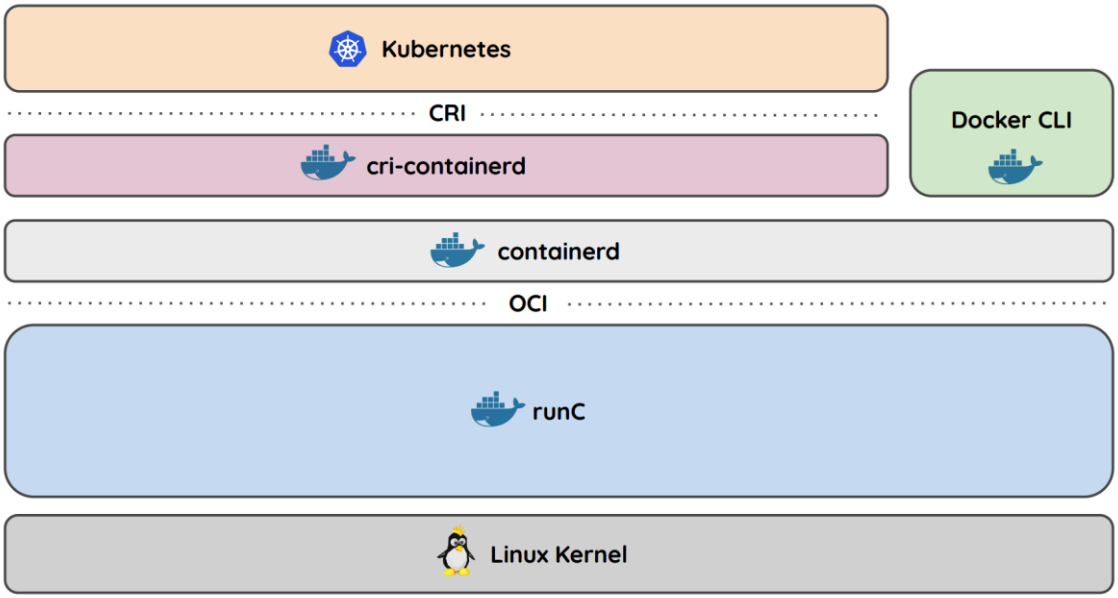


KUVIO 8. OCI-standardin myötä konttien ajoalusta irrotettiin konttimoottorista. RunC oli ensimmäinen OCI-standardin mukainen ajoalusta (Carrez 2020).

Vuonna 2016 julkaistiin CRI-standardi. CRI-standardi määrittelee rajapinnan, jonka avulla konttien ajoalustat voivat keskustella konttien orkestrointityökaluille. Dockerin tapauksessa tämä tarkoitti konttimoottorin pilkkomista osiin:

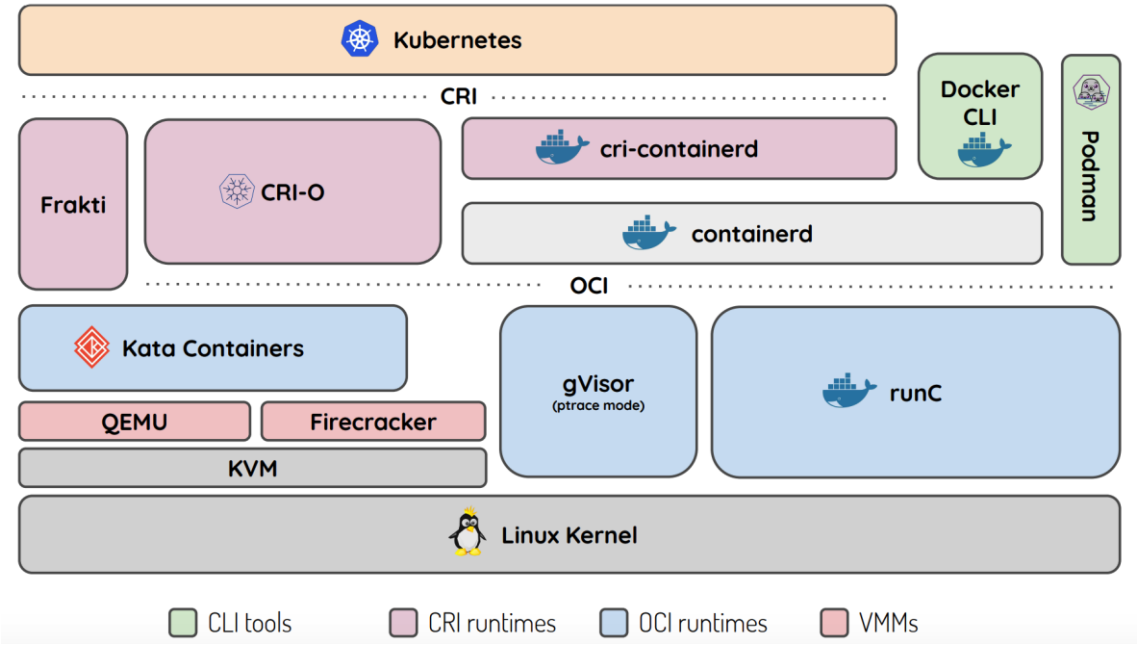
- Containerd, joka on itse konttimoottori.
- CRI-containerd, joka on ikään kuin adapteri, joka toteuttaa CRI-standardin rajapinnan.
- Docker CLI, joka on komentorivikäyttöliittymä Dockerin ohjaamiseen suoraan päätteeltä.

Samoihin aikoihin ilmestyi myös rkt-konttialusta. Rkt on Googlen hautomosta syntynyt tuote, jolla muun muassa on mahdollista käsitellä natiivisti kapseleita. (Carrez 2020.)



KUVIO 9. CRI-standardin syntyminen johti Docker:in konttimoottorin pilkkomiseen containerd-, cri-containerd- ja Docker CLI -moduuleihin (Carrez 2020).

Konttiteknologian standardointi ja avoimen lähdekoodin implementointi on kasvattanut merkittävästi saatavilla olevien konttiteknologioiden määrää ja kirjoa. Monet suuremmat yritykset, kuten AWS ja Google, ovat julkaisseet omia sisäisiä projekteja ja konttiteknologiaa avoimena lähdekoodina, koska siten heidän teknologiansa voidaan istuttaa avoimien konttiteknologioiden standardeihin, joka vaikuttaa myös standardien kehitykseen. Tällaisia teknologioita ovat muun muassa konttien ajoalustat gVisor (Google), Firecracker (AWS) ja Kata (IBM, Intel). (Carrez 2020.)



KUVIO 10. Konttien teknologiapinot (Carrez 2020).

Monissa käyttötapauksissa on huomattu, että konttien kevyt virtualisaatio ei eristä konttia tarpeeksi ympäristöstään. KVM (engl. Kernel-based Virtual Machine) eli Kernel-pohjainen virtuaalikone on teknologia, jota kehitetään vastaamaan lisääntyneeseen eristyksen tarpeeseen. Teknologiat, kuten Kata Containers, kehittävät konttien virtualisointia, jotta erityisen sensitiivisiä työkuormia voidaan ajaa turvallisesti. Kata Containers kuitenkin häviää suorituskyvyssä huomattavasti runC:lle. (Carrez 2020.)

Konttien ympärille tehdyt standardit ovat luoneet mahdollisuuden ajaa eri työkuormia hyödyntämällä eri konttiteknologiapinoja. Voimme siis ajaa työkuormia, jotka eivät sisällä sensitiivistä informaatiota, hyödyntämällä suorituskykyistä ja kevyttä teknologiapinoa, esimerkiksi cri-O ja runC. Tällainen työkuorma voi olla, vaikka palvelu, joka esittää pörssikurssit. Sensitiivistä informaatiota sisältävä työkuorma voidaan ajaa turvallisella teknologiapinolla, kuten cri-O ja Kata ja QEMU. Tällainen työkuorma voi olla palvelu, joka palauttaa käyttäjän osakesalkun sisällön. (Carrez 2020.)

2.5 Konttien organisointi

Konttien organisoija (engl. Container orchestrator) on järjestelmä, joka on suunniteltu keräämään useita koneita yhdeksi tai useaksi klusteriksi. Klusteri muodostaa yhtenäisen alustan, joka näyttäytyy käyttäjälle yhtenä hyvin tehokkaana tietokoneena, jonka päällä kontteja voidaan ajaa. Or-

ganisoijalla on käytännössä kolme tehtävää: dynaamisesti aikatauluttaa konttipohjaisia työkuormia klusteroidussa ympäristössä, hallita ja ylläpitää klustereita sekä tarjota yhteinen määrittelytapa ympäristössä ajettaville työkuormille. Käytännössä tämä tarkoittaa sitä, että orkestroijalle määritetään deklaratiivisesti ympäristön haluttu tila (engl. desired state) ja orkestrointityökalu pyrkii sovittamaan nykytilan niin, että se vastaa haluttua tilaa. (Domingus & Arundel 2022, luku 1.)

2.6 Kubernetes

Kubernetesin juuret ovat Googlen sisäisissä konttien organisointijärjestelmissä Borg ja Omega. Googlella, kuten monilla muillakin teknologiajäteillä, oli yli vuosikymmenen kokemus konttien orkestroinnista, ennen Kubernetesin julkaisua avoimen lähdekoodin ratkaisuna. Kubernetesin suunnitteluperiaatteet ovat skaalautuvuus, turvallisuus ja liikuteltavuus. Kaikki sen komponentit ovat riippumattomia toisistaan (engl. loosely coupled) ja räätälöitävissä. (Domingus & Arundel 2022, luku 1.)

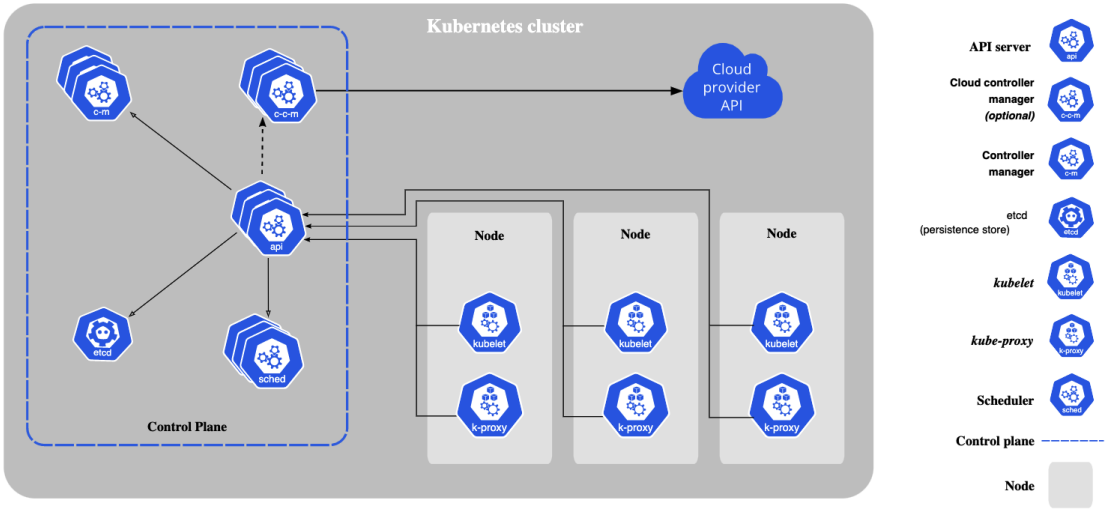
Kubernetes koostuu useista eri abstraktioista kuten noodit (engl. Nodes), jotka vastaavat palvelimia tai niiden virtualisoituja ositteita. Komponentit (engl. Components), jotka vastaavat työkuormien ja noodien orkestroimisesta. Objektit (engl. Objects) ja resurssit (engl. Resources), jotka esittävät osia, jotka muodostavat työkuorman. (Luksa 2018, luku 1.3.2.)

Kubernetesin toiminta perustuu deklaratiivisuuteen, eli klusterin haluttu tila (engl. desired state) määritellään erillisiin dokumentteihin ja Kubernetes pyrkii sovittelemaan klusterin nykytilan vastaamaan tätä haluttua tilaa. Kaikki Kubernetes-resurssit ovat määriteltävissä yaml-dokumenttina. (Cloud Native Computing Foundation 2022a.)

2.6.1 Kubernetesin arkkitehtuuri

Kubernetes koostuu työnoodeista (engl. Worker nodes), ja niiden ohjaamista varten olevasta ohjaustasosta (engl. Control plane). Ohjaustaso koostuu komponenteista, jotka muodostavat Kubernetesin suoritus- ja ohjauslogiikan sekä käyttöliittymän. Kuviossa 11 on esitelty eri ohjaustason komponentit: rajapintapalvelin (API server), aikatauluttaja (Scheduler), kontrolleriohjaaja (Controller manager), etcd-tietovarasto (etcd) ja pilvipalvelun tarjoajan kontrolleri ohjaaja (Cloud controller

manager). Kaikki komponentit esitellään tarkemmin luvussa 2.6.2. (Cloud Native Computing Foundation 2022b.)



KUVIO 11. Kubernetesin arkkitehtuuri koostuu ohjaustasosta, joka sisältää Kubernetesin ohjauslogiikan, ja työnoodeista, jotka vastaavat ajettavista työkuormista, kuten kapselit (Cloud Native Computing Foundation 2022b).

Työnoodit muodostuvat seuraavista komponenteista: Kube-välityspalvelimesta (Kube-proxy) ja Kubelet-komponenteista, joiden avulla työnoodit keskustelevat ohjaustason kanssa ja suorittavat työkuormia. Yhdessä ohjaustaso ja noodit muodostavat klusterin. Nykyään myös moniklusteriympäristöt ovat yleistyneet, mikä on vaatinut vielä korkeamman tason abstraktioita, kuten klusteriryhmä. (Cloud Native Computing Foundation 2022b.)

2.6.2 Ohjaustaso

Ohjaustaso (engl. Control plane) on abstraktio, joka sisältää Kubernetesin komponentit, jotka käsittelevät ja ohjaavat koko klusterin toimintaa ja siinä suoritettavia työkuormia. Ohjaustasoa on vanhemmissa lähteissä ja dokumentaatiossa kutsuttu päänoodiksi (engl. Master Node). Ohjaustaso koostuu seuraavista komponenteista: rajapintapalvelimesta (API Server), aikatauluttaja (Scheduler), kontrollerimanageri (Controller Manager), pilvikontrollerijohtaja (Cloud Control Manager), etcd-tietovarasto (etcd). (Cloud Native Computing Foundation 2022b.)

Rajapintapalvelin

Rajapintapalvelin (ApiServer) vastaa kaikesta viestinnästä ohjaustason ja muun järjestelmän välillä. Se tarjoaa RESTful-tyyppisen rajapinnan, jonka avulla klusterin tilaa voidaan käsitellä (Luksa 2018). Klusterin tila säilytetään etcd-tietovarastossa, jota rajapintapalvelin päivittää. Rajapintapalvelimen yleisin implementaatio Kubernetes-klusterissa on kube-apiserver. Kube-apiserver on suunniteltu skaalautumaan horisontaalisesti, eli se skaalautuu käynnistämällä uusia instansseja itsestään. On siis mahdollista ajaa useita instansseja rajapintapalvelimesta samassa klusterissa hyödyntämällä kuormanjakoa. (Cloud Native Computing Foundation 2022b.)

Etcd

Etcd-tietovarastossa säilytetään Kubernetes-klusterin tilaa, eli tietoa klusterin komponenttien konfiguraatiosta. Etcd on hyvin vikasietoinen ja hajautettu avain-arvo-tietokanta. Etcd muodustuu sanoista etc, joka on Linux-hakemisto, missä normaalisti säilytetään konfiguraatiotietoja, ja d, joka tulee sanasta distributed. Näin ollen etcd-tietovaraston suunnittelun lähtökohteena oli tuottaa hajautettu etc -hakemisto. (Cloud Native Computing Foundation 2022f.)

Aikatauluttaja

Aikatauluttaja (engl. Scheduler) on ohjaustason komponentti, joka vastaa uusien kapselien (engl. Pod) sijoittamisesta työnoodeille. Aikatauluttaja tekee päätöksiä, jotka perustuvat työnoodien ja yksittäisten kapselien resursseihin ja ympäristön asettamiin rajoitteisiin, määrittelyyn ja sääntöihin, kuten yhteenkuuluvuus- (engl. affinity) ja yhteenkuulumattomuusehdot (engl. anti-affinity). Yhteenkuuluvuus ja -kuulumattomuus määrittävät samalle ja eri työnoodeille kuuluvat kapselit. Aikatauluttaja huomioi myös tiedon paikallisuuden (engl. localization), työkuormien väliset häiriöt ja niiden elinkaaren. (Cloud Native Computing Foundation 2022b.)

Kontrolleriohjain

Kontrolleriohjain (engl. Controller manager) on ohjaustason komponentti, joka käynnistää kontrolleriprosesseja. Jokainen kontrolleri on käytännössä erillinen prosessi, mutta monimutkaisuuden vähentämiseksi kaikki kontrollerit on niputettu yhdeksi binääriksi, jolloin sitä myös ajetaan yhtenä prosessina. Kontrollerit valvovat Kubernetes-resurssien tilaa ja pyrkivät ylläpitämään resurssien nykytilaa (engl. current state) niin että se vastaa mahdollisimman hyvin toivottua tilaa (engl. desired state). Kontrolleri on käytännössä prosessi, joka seuraa loppumattomasti resurssin tilaa ja

reagoi, jos nykytila ei vastaa toivottua tilaa. Tätä kutsutaan myös sovitussilmukaksi (engl. reconciliation loop). Kontrolleriohjainprosessi sisältää kontrollereita, kuten noodikontrollerin, jonka vastuulla on reagoida ja huomata, jos työnoodi häviää. (Cloud Native Computing Foundation 2022b.)

Pilvipalvelutarjoajien kontrolleriohjain

Pilvipalvelutarjoajien kontrolleri ohjein in ohjaustason komponentti, joka sisältää pilvipalvelutarjoajakohtaista ohjauslogiikkaa. Tämä komponentti erottaa palvelutarjoajakohtaiset komponentit klusterin sisäisistä komponenteista. Kontrolleriohjaimen tapaan pilvipalveluiden kontrolleriohjain (engl. Cloud controller manager) kokoaa nipun kontrollereita yhdeksi binääriksi, kuten kontrolleriohjain. (Cloud Native Computing Foundation 2022b.)

2.6.3 Työnoodit

Kubelet

Kubelet on prosessi, joka toimii klusterin kaikissa kapseleissa (Pods). Sen tehtävänä on varmistaa, että kapselin sisällä olevat kontit ovat toiminnassa. Kubelet siis käytännössä vastaa kapselien ajosta, luomisesta ja tuhoamisesta. Kubelet sijaitsee ohjaustason rajapintapalvelimen ja konttien ajoalustan välissä. Kapselispesifikaatio (PodSpec) sisältää määrittelyt ajettavista konteista. (Cloud Native Computing Foundation 2022b.)

Kube-välipalvelin

Kube-välipalvelin (Kube-proxy) on verkkovälityspalvelin, joka toimii klusterin jokaisella noodilla. Se kuuluu Kubernetesin palvelukonseptiin (engl. service concept). Välityspalvelin ylläpitää verkkomäärityksiä ja -sääntöjä noodeilla. Välityspalvelin on vastuussa kapselien klusterin sisäisestä ja ulkopuolisesta verkkoliikenteessä. (Cloud Native Computing Foundation 2022b.)

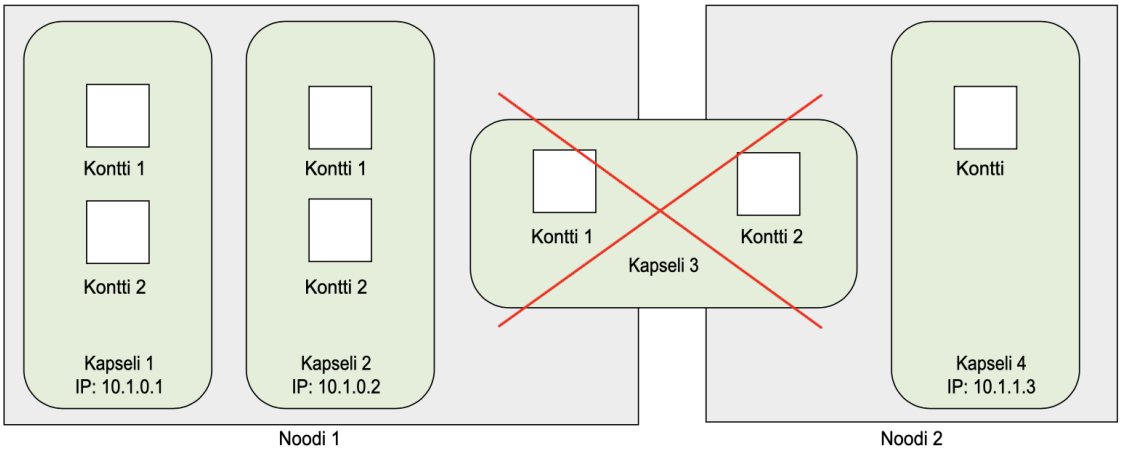
2.6.4 Kubernetes-resurssit

Kuberentes-konfiguraatio koostuu entiteeteistä, joita kutsutaan resursseiksi. Näistä resursseista muodostetaan ympäristön haluttu tila (engl. Desired state). Resurssit määrittelevät, mitä konttipohjaisia sovelluksia ympäristössä toimii, millä työnoodilla niitä ajetaan sekä niiden käytettävissä

olevat järjestelmän resurssit ja niiden ominaisuudet sekä säännöt. Kubernetes-resursseja ovat kapselit (Pod), kopiosarjat (Replica sets), julkaisut (Deployments), tilalliset sarjat (StatefulSet), palvelut (Services), säilyvät taltiot (Persistent Volumes) ja monia muita. (Bilgrin & Huß 2019, luku 1.)

Kapseli

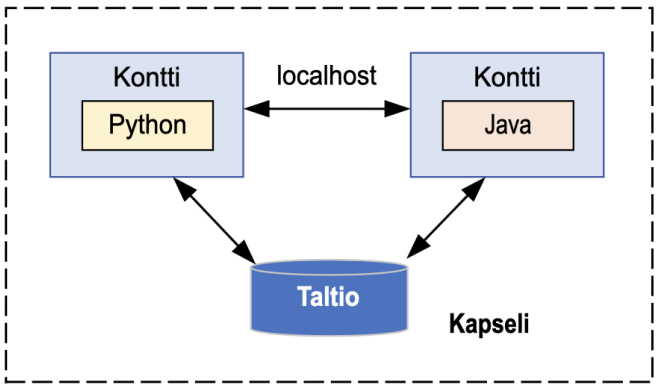
Kapseli (Pod) sisältää yhden tai useita kontteja. Kapseli on pienin atominen yksikkö, jota Kubernetes voi aikatauluttaa, julkaista ja eristää ajossa. Kuten on esitelty kuviossa 12, kaikki kapselin sisällä olevat kontit aikataulutetaan aina samalle työnoodille. Ne myös julkaistaan aina yhtenä kokonaisuutena. Kapseli on yksikkö, jolla hallitaan kontin tai konttiryhmän elinkaarta. (Bilgrin & Huß 2019, luku 1.)



KUVIO 12. Kapselin sisältämät kontit julkaistaan aina samalle työnoodille (Luksa 2019).

Kapseli hyödyntää konttien eristyksessä samoja Linux Kernelin komponentteja, joita kontit hyödyntävät. Kapselin sisällä kontit jakavat IP-osoitteet ja portit, eli kapselin eristämiseen hyödynnetään Linux Kernelin verkkonimiavaruutta (engl. Network namespace). Konteilla on myös sama isäntänimi (engl. hostname) ja UTS-nimiavaruus. Ne pystyvät kommunikoimaan keskenään hyödyntämällä prosessiensisäistä viestintää eli IPC-nimiavaruutta. Kapselilla on oma PID-nimiavaruus. PID-nimiavaruus voi olla eristetty myös konttien välillä, jolloin kontit näkevät vain omat prosessinsa. Kapselin sisäinen tiedostojärjestelmä poikkeaa konttien tiedostojärjestelmästä. Levykuvissa konteille määritellyt tiedostojärjestelmän liitospisteet säilyvät, mutta kapselin sisällä muut kontit eivät näe toistensa tiedostojärjestelmiä liitoksista huolimatta. On kuitenkin mahdollista, että kontit jakavat hakemistoja keskenään taltion (engl. Volume) avulla. Kuviossa 13 on esimerkki,

kuinka kontit voivat jakaa taltion ja tietoliikenneyhteyden kapselin sisällä. (Beda ym. 2017, luku 5.)



KUVIO 13. Kontit voivat kommunikoida Kapselin sisällä verkon yli eli localhostin kautta, jaetun levyn eli tiedostojärjestelmän kautta tai hyödyntämällä prosessien sisäistä kommunikaatiota eli IPC-mekanismia (Bilgrin & Huß 2019, luku 1).

Suunnitellessa kapselien sisältöä on hyvä muistaa, että jos applikaatio ei toimisi niin, että sen palvelut ovat eri fyysisillä palvelimilla, ne tulee sijoittaa myös samaan kapseliin (Beda ym. 2017, luku 5).

Merkkaus ja annotaatiot

Tunnisteet (engl. Label) ovat avain-arvopareja, jotka liitetään Kubernetes-resursseihin niiden tunnistamiseksi. Tunnisteet mahdollistavat käyttäjien oman organisaatiostruktuurin kartoittamisen Kubernetes-resursseihin. Tunnisteet toimivat perustana resurssien ryhmittelylle. Annotaatiot toisaalta tarjoavat tiedon säilytysmekanismin, joka muistuttaa tunnisteita. Annotaatiot ovat avain-arvo-pareja, mutta toisin kuin tunnisteet ne on suunniteltu pitämään tietoa, jota ei käytetään indeksointiin tai yksittäisen resurssin identifioimiseen. (Beda ym. 2017, luku 6.)

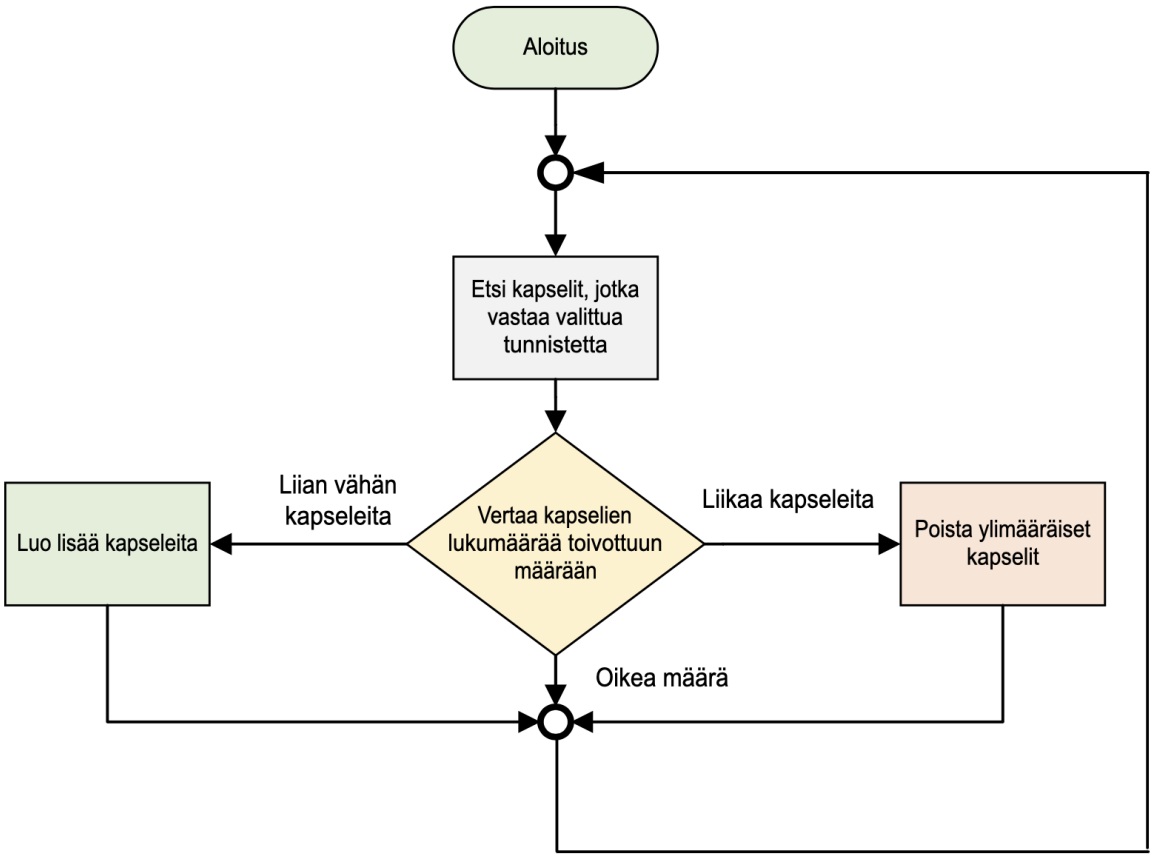
Palvelu

Monet asiat, kuten konttien käynnistyminen, ympäristön skaalaaminen, noodin vaihto ja muutokset, aiheuttavat kapselin tuhoamisen ja uudelleen rakentamisen – kapselit ovat lyhytikäisiä. Kapselille voidaan määritellä IP-osoite vasta sen aikataulutuksen yhteydessä, joten tarvitaan oma resurssi vastaamaan kapselin löytämisestä ja kuormanjaosta. Palvelu (engl. Service) sitoo IP-osoit-

teen ja portin pysyvästi omaan nimeensä tai osoitteeseensa, joka toimii ja näkyy sen jälkeen sisääntulopisteenä applikaatioon. Palvelu siis tarjoaa käyttäjälle yhden osoitteen, josta käyttäjä voi kommunikoida yhden tai useamman kapselin kanssa. (Bilgrin & Huß 2019, luku 1.)

Kopioinnin ohjain

Kopioinnin ohjain (engl. Replication controller) tehtävänä on yksinkertaisesti varmistaa, että kapselit ovat aina käynnissä ja niitä on oikea määrä. Ohjaimelle määritellään, mitä kapseleita se tarkkailee ja kuinka monta instanssia kyseistä kapselia tulee olla olemassa. Ohjain alkaa tarkkailla sen jälkeen määriteltyjä kapseleita ja niiden lukumäärää. Kuviossa 14 on selvitetty, mitä tapahtuu, jos ohjain huomaa nykytilan poikkeavan toivotusta tilasta, eli että kapseleita on liikaa tai liian vähän. (Luksa 2018, luku 4.2.1.)



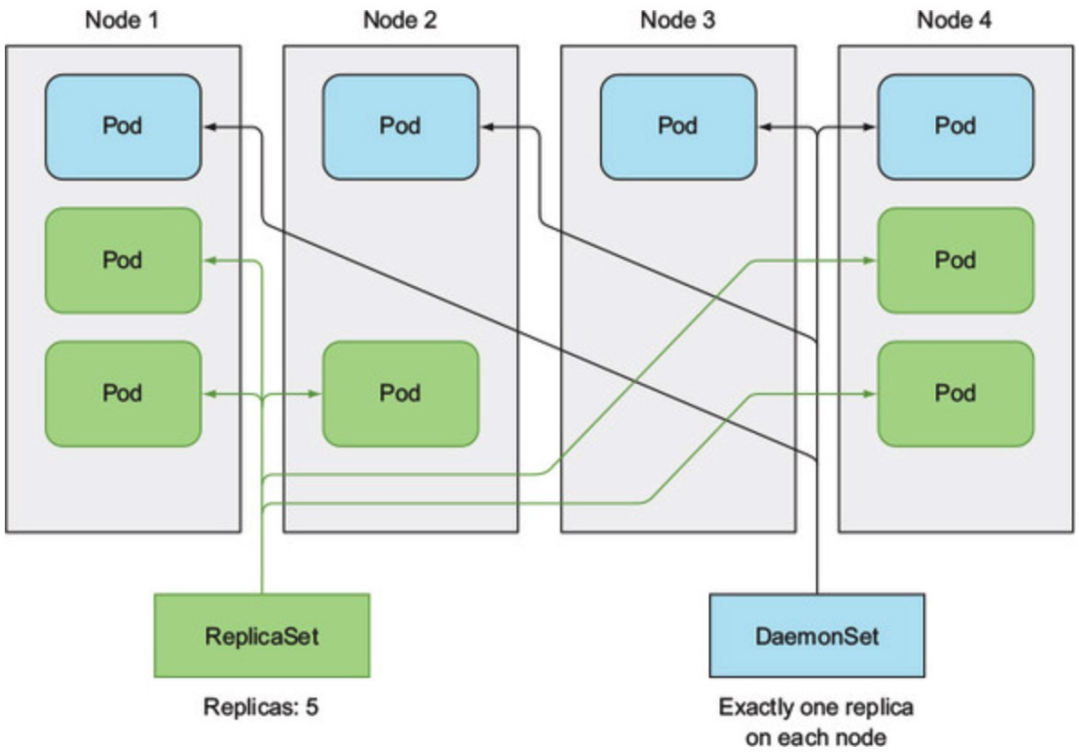
KUVIO 14. Kopioinnin ohjaajan sovittelusilmukka (engl. Reconsiliation loop) (Luksa 2018, luku 4.2.1).

Kopiojoukko

Kopiojoukon (ReplicaSet) tehtävä on replikoida ja uudelleen aikatauluttaa kapseleita, kun niiden noodi kaatuu. Kopiojoukko luotiin jatkamaan kopioinnin ohjainta, mutta nykyään se on korvannut kokonaan kopioinnin ohjaajan käytön. Kopiojoukko toimittaa kaikki samat tehtävät kuin kopioinnin ohjaaja, mutta sen kapselivalitsija (engl. Pod selector) on kehittyneempi. Kopioinnin ohjaajan valitsija etsi aina vain ne kapselit, joiden tunnisteen avain-arvo-pari vastaavaa täysin määriteltyä avain-arvo-paria. Kopiojoukko taas voi hyväksyä myös kapseleita, joiden tunnisteen pelkkä avain vastaa määriteltyä avainta. Se ei ota kantaa avaimen sisältämään arvoon tai siihen, että kaikkia määriteltyjä avain-arvopareja ei löydy kapselin tunnisteista. (Luksa 2018, luku 4.3.1.)

Daemon-joukko

Daemon-joukko mahdollistaa tietyn kapselin pyörittämisen jokaisella noodilla, ja jokaisella on tasan yksi instanssi kyseistä kapselia. Esimerkki tällaisesta sovelluksesta voisi olla lokitietojen kerääminen: tietoja pitää kerätä joka noodilta, mutta niitä keräämiä prosesseja ei saa olla useampaa kuin yksi noodia kohde, jotta lokeista ei synny duplikaatteja. Kuvio 15 auttaa hahmottamaan daemon- ja kopiosarjan eron. (Luksa 2018, luku 4.4.)



KUVIO 15. Kopiosarjan ja daemon-sarjan ero esitettynä (Luksa 2018, luku 4.4).

Työ

Työ (Job) on Kubernetes-resurssi, joka mahdollistaa työkuorman ajamisen kerran, jolloin se suoritetaan alusta loppuun, eikä sitä enää sen jälkeen ajeta. Käytännössä tämä tarkoittaa sitä, että kun kapselin sisällä olevien konttien työkuormat valmistuvat, kapselin tila muuttuu valmiiksi, eikä Kubernetes ala käynnistämään kontteja uudestaan. Töiden sisälle on mahdollista asettaa myös ajastettuja Cron-tehtäviä (engl. CronJob). Cron-tehtävä on oma Kubernetes-resurssinsa. (Luksa 2018, luku 4.5.1.)

Konfiguraatiokartta

Konfiguraatiokartta (ConfigMap) on Kubernetes resurssi, joka tallentaa ja säilyttää konfiguraatiodataa. Data on kerätty avain-arvo-pareihin. Kapseli käyttää konfiguraatiokartan tietoja joko ympäristömuuttujien (engl. environment variable) avulla tai kokonaisina konfiguraatiotiedostoina, jotka on jaettu yhteisen levyn avulla. Kapseli yhdistyy konfiguraatiokarttaan sen nimen avulla, mikä tarkoittaa, että konfiguraatiokartan sisältöä voidaan määritellä esimerkiksi eri integraatiotasojen mukaan (kehitys, testi, tuotanto), jolloin muutoksia kapseliin ei tarvita eri integraatiotasoissa. (Luksa 2018, luku 7.4.1.)

Salaisuus

Salaisuus (Secret) toimii samalla tavalla kuin konfiguraatiokartta, eli se voi jakaa kapselille tietoa tiedoston kautta tai suoraan ympäristömuuttujana. Salaisuus-resurssiin tulisi tallettaa sensitiivistä tietoa, kun taas konfiguraatiokartta on normaalia konfiguraatiotietoa varten. Kubernetes jakaa salaisuuden tiedot vain noodille, jossa sitä käytetään. Lisäksi tietoa käytetään ainoastaan suoraan muistista, joten sitä ei tallenneta fyysiselle levylle missään vaiheessa. Salaisuus-resurssien sisältämä tieto säilytetään salattuna (engl. encrypted) ohjaustason etcd-tietovarastoissa. (Luksa 2018, luku 7.5.1.)

Pysyvä taltio

Pysyvä taltio (Persistent volume) on Kubernetes-taltiohallinnan resurssi, joka abstraktoi käyttäjältä ja ylläpitäjältä taltion käyttöön liittyvät yksityiskohtaiset tiedot. Pysyvä taltio muodostuu kahdesta resurssista: pysyvä taltio ja vaatimukset pysyvälle taltiolle (engl. Persistent volume claim). Klusterissa se on samankaltainen klusterin komponentti kuin noodi. Pysyvä taltio on samankaltainen lisäosa kuin taltiot (Volumes), mutta sillä erotuksella, että se ei ole riippuvainen kapselin elin-

kaaresta. Vaatimus pysyvälle taltiolle on resurssin käyttäjän tekemä tallennustila pyyntö. Pyyntöön voidaan määritellä haluttuja oikeuksia levylle, kuten luku- ja kirjoitusoikeudet. (Cloud Native Computing Foundation 2022c.)

Tilallinen joukko

Tilallinen joukko (engl. StatefulSet) on konsepti, joka on rakennettu tilallisia työkuormia varten. Sen tehtävä on hallita tilallisten palveluiden julkaisua ja skaalausta klusterissa. Resurssina tilallinen joukko on samankaltainen kuin julkaisu, mutta tilallinen joukko ylläpitää ja säilyttää kapseleita identifioivaa tietoa. (Cloud Native Computing Foundation 2022c.)

RBAC

Rooleihin perustuva käyttöoikeus (engl. Role based access control) on keino, jolla voidaan määritellä erilaisia käyttöoikeuksia eri resursseihin käyttäjille. Kubernetes ei alunperin sisällä käyttäjä konseptia, minkä takia käyttäjän lisääminen täytyy rakentaa hyödyntämällä rooleihin perustuvaa käyttöoikeutta. Se muodostuu neljästä eri resurssista: rooli (Role), klusterirooli (ClusterRole), roolin kiinnite (Role binding) ja klusteriroolin kiinnite (ClusterRoleBinding). (Cloud Native Computing Foundation 2022d.)

Rooli ja klusterirooli sisältää säännöt, joilla määritellään oikeuksia. Niissä voidaan määritellä ainoastaan oikeuksia, ei siis estoja tai rajoituksia. Roolit määritellään aina nimiavaruuden mukaan, mutta klusterirooli taas ei sisällä nimiavaruutta, koska Kubernetes-resurssit voivat olla joko nimiavaruuden sisällä tai kuulumatta nimiavaruuteen. Roolien sitominen on roolimäärityksen eli roolin tai klusteriroolin kiinnittämistä tiettyyn palveluun, käyttäjään tai ryhmään. (Cloud Native Computing Foundation 2022d.)

3 PROTOTYYPIN SUUNNITTELU JA TOTEUTUS

Tässä luvussa pureudutaan tutkimuskysymykseen ja sen pohjalta rakennettuun käytännön toteutukseen. Tutkimuskysymyksenä oli, miten konttipohjaiseen moniklusteriympäristön julkaisu- ja hallintajärjestelmän tilaan voidaan rakentaa näkyvyyttä. Tämän lisäksi tavoitteena oli, että kysymystä voitaisiin lähestyä ensisijaisesti Kubernetes-natiivien toiminnallisuuksien avulla ja että saataisiin rakennettua toimiva prototyyppi.

Toteutus on rakennettu Kubernetesin tarjoamilla toiminnollisuuksilla sekä laajentamalla sen ydinkomponentteja, jotta työn tulokset eivät olisi sidoksissa muihin tekniikkoihin, Kubernetesin ajoympäristöön tai esimerkiksi ohjelmointikieleen. Toteutuksessa käytetty ohjelmointikehys (engl. Framework) tukee useita eri ohjelmointikieliä, mutta itse toteutuksen lähdekoodi on kirjoitettu Python-ohjelmointikielellä. Luvussa esitellään myös toteutuksen suunnittelua ja itse suunnitteluprosessia, Operaattori-ohjelmointikaava (engl. Operator design pattern) ja toteutuksessa hyödynnetyt työkalut.

3.1 Suunnittelu

Työn tavoitteena on selvittää mahdollisia tapoja rakentaa näkyvyyttä Kubernetes-klusterin sisällä tapahtuvista julkaisuista, joita hallinnoi ja julkaisee kolmannen osapuolen tuottama konttien ohjaus- ja julkaisujärjestelmä (engl. container management and deployment engine). Kyseessä on järjestelmä, jonka päätehtävinä on ylläpitää tietoa kaikkien klusterien resurssien nykytilasta sekä tarkkailla muutoksia versiohallinnassa määriteltyyn klusterien resurssien haluttuun tilaan. Järjestelmä sovittaa nykytilan ja halutun tilan välisiä muutoksia julkaisemalla, päivittämällä tai poistamalla resursseja. Näitä tapahtumia hyödyntämällä lähdettiin toteuttamaan näkyvyyttä järjestelmään.

Näkyvyys koostuu kolmesta peruspilarista: tapahtumalokit, metriikka ja jäljitys. Näkyvyyttä käsitellään tarkemmin luvussa 2.1.3. Toteutuksen tavoitteena on luoda räätälöityä tapahtumalokitietoa. Metriikka ja jäljitys ovat osa-alueita, joista tällä hetkellä saadaan tilaajan monitorointijärjestelmään tietoa. Tilaajalla olisi mahdollisuus hyödyntää myös Kubernetesin tapahtumalokeja tiedon kokoamiseksi monitorointijärjestelmään, mutta kyseinen tapahtumatieto sijaitsee monien eri klusterien

sisällä. Lisäksi sen kohdistaminen, kerääminen ja käsittely olisi hyvin haasteellista ja vikaherkkää. Tästä syystä työn suunnitteluvaiheessa pyrittiin löytämään yksi keskeinen lähde, joka sisältäisi kaiken tarpeellisen tiedon tapahtumatietojen rakentamiseksi. Kubernetesin jäljityslokiominaisuus (engl. Audit logs) tarjoaa yhden vaihtoehdon tiedon keräämiseen yksittäisestä keskitetystä lähteestä.

Jäljityslokit kerätään suoraan Kubernetes-ohjaustason rajapintapalvelimelta. Rajapintapalvelin käsittelee kaiken viestinnän klusterin sisällä, joten jäljityslokit sisältävät kirjaimellisesti kaikki tapahtumat klusterissa. Jäljityslokeja ei tuoteta klusterista vakiona vaan se on ominaisuus, joka täytyy erikseen määritellä ja ottaa käyttöön. Rajapintalokit tarvitsevat myös oman tietokannan, johon lokit tallennetaan. Koska lokimateriaalia syntyy erittäin paljon, jäljityslokien käyttöönotto lisää rajapintapalvelimen järjestelmäresurssien tarvetta, joten sillä on negatiivisia vaikutuksia rajapintapalvelimen suorituskykyyn. Keskitettyä lokimateriaalia voitaisiin hyödyntää esimerkiksi lokitiedon käsittelyyn erikoistuneella hakumoottorilla.

Toinen vaihtoehto on seurata julkaisujärjestelmän omien komponenttien tilaa ja rakentaa sen pohjalta räätälöityä tapahtumatietoa. Yksi vaihtoehto toteuttaa tämän kaltainen ratkaisu on Operaattori-suunnittelukaavaa, jossa komponenttien tilan muutoksia seurataan kontrollerien avulla. Operaattori koostuu mukautetusta resurssista (engl. Custom resource), joka jatkaa Kubernetes-rajapintapalvelinta uudella päätepisteellä ja kontrollerista (engl. Controller), joka sovittelee mukautetun resurssin tilaa. Käytännön toteutukseen valittiin Operaattori-suunnittelukaava, koska se on Kubernetes-natiivi konsepti, joka tämän toteutuksen lisäksi mahdollistaa myös muunlaisen ohjelmointilogiikan rakentamisen Kubernetesiin.

3.2 Operaattori-ohjelmointikaava

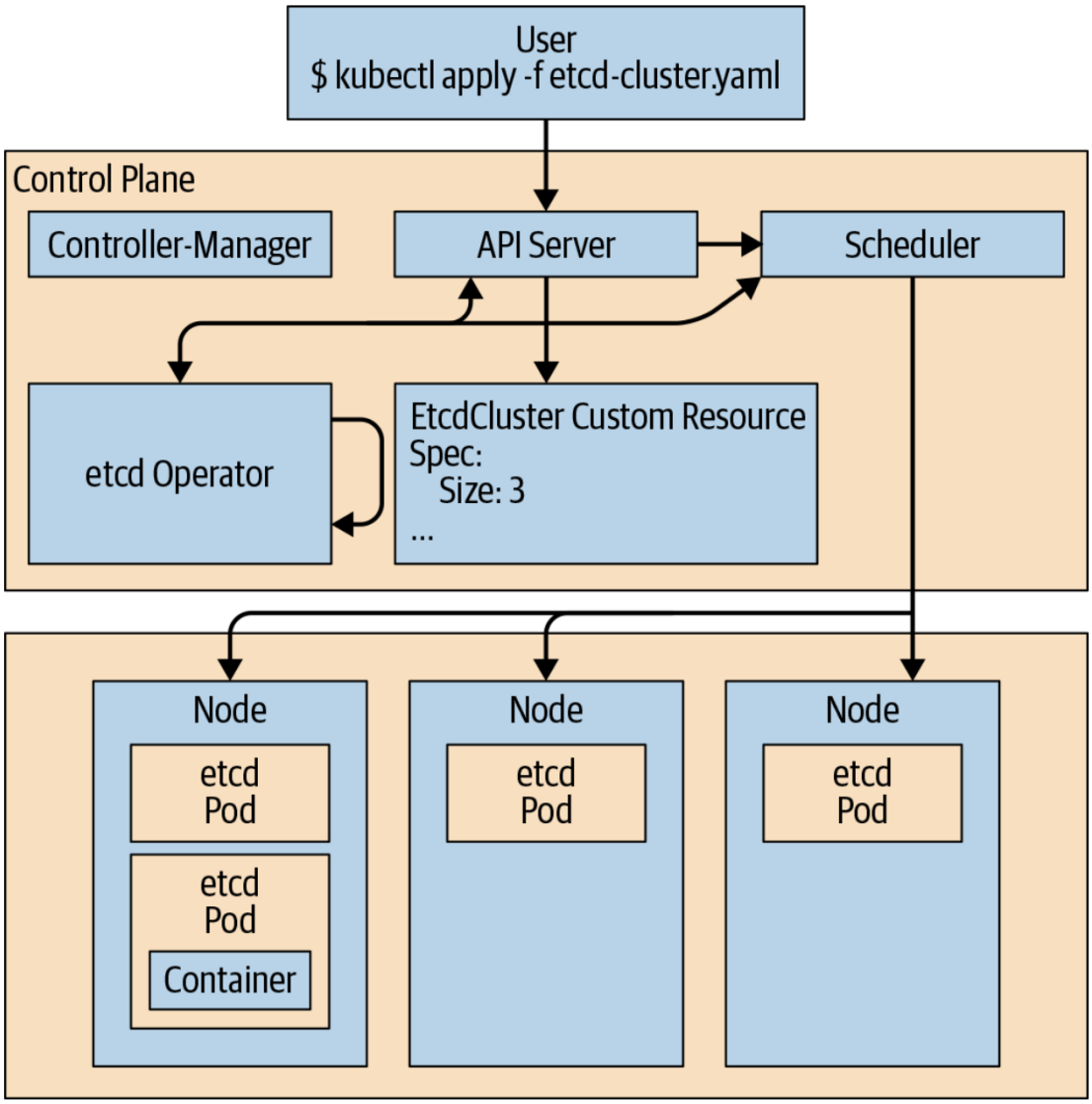
> An operator is a Kubernetes controller that understands two domains: Kubernetes and something else. By combining knowledge of both areas, it can automate tasks that usually require a human operator that understands both domains. (Zelinskie 2018.)

Operaattorin avulla Kubernetes-ympäristössä voidaan automatisoida ylläpitotyön tehtäviä. SRE (engl. Site Reliability Engineer) on Googlen kehittämä ammattinimike ja kokoelma käytäntöjä,

sääntöjä ja toimintatapoja. Ne ovat suunniteltu helpottamaan tai jopa mahdollistamaan suuren järjestelmän hallinnoimisen. Merkittävä osa SRE:n työstä koostuu järjestelmän pääkäyttäjien ja muiden toistuvien manuaalisten tehtävien automatisoinnista. Operaattori on yksi työkalu, jota voi hyödyntää automatisoinnissa. Operaattori on käytännössä mukautetun resurssin (engl. Custom resource), CR:n, ja mukautetun kontrollerin (engl. Custom controller) yhdistelmä. (Dobies & Wood 2020.)

Operaattori

Operaattori toimii laajentamalla Kubernetesin rajapintapalvelinta, joka sijaitsee klusterin ohjaustasossa. Yksinkertaisessa käyttötapauksessa operaattori lisää yhden päätepisteen (engl. Endpoint) rajapintapalvelimeen. Nämä päätepisteet ovat mukautettuja resursseja, joiden lyhenne on CR. Operaattori on käytännössä mukautettu resurssi, CR, jonka parina toimii ohjaustason komponentti, yleensä sitä varten rakennettu kontrolleri, joka seuraa ja ylläpitää tätä uutta resurssia. Kuviossa 16 on esitelty esimerkki operaattorista. Etcd-Operaattori seuraa EtcdCluster-mukautetun resurssin tilaa ja sovittaa tilan aikatauluttajalle. (Dobies & Wood 2020, luku 1.)



KUVIO 16. ”Etcd Operator” seuraa ja ylläpitää ”EtcdCluster” -mukautetussa resurssissa määriteltyä tilaa. Esimerkissä tila on määritelty niin, että etcd-kapselien toivottu lukumäärä on kolme.”etcd Operator” välittää Kubernetes aikatauluttajalle tiedon, joka sovittaa halutun tilan. (Dobies & Wood 2020, luku 1.)

Kontrolleri

Kontrolleri (engl. Controller) aktiivisesti monitoroi ja ylläpitää sille määriteltyjen Kubernetes-resurssien haluttua tilaa (engl. Desired state). Tällaista loppumatonta silmukkaa kutsutaan sähkötekniikan puolella säätöpiireiksi (engl. Control loop). Säätöpiirit ovat hyvin keskeinen konsepti Kubernetesin arkkitehtuurissa. Ohjaustason ytimessä toimivat Kubernetesin omat kontrollerit, kontrollerimanageri (engl. Controller manager). (Kubernetes dokumentaatio 2022)

Mukautettu resurssi

Operaattori koostuu kahdesta osasta: kontrollerista ja mukautetusta resurssista (engl. Custom Resource). Mukautetun resurssin avulla voidaan jatkaa Kubernetesin rajapintapalvelinta uudella päätepisteellä (engl. Endpoint), jolloin uusi resurssi on kutsuttavissa samalla tavalla, kuin Kubernetesin alkuperäiset resurssit. Esimerkiksi jos resurssin nimi on EtcdCluster, Kubectl CLI-työkalun avulla resurssia voidaan kutsua suoraan, kubectl get etcdclusters -komennolla. (Dobies & Wood 2020, luku 1.)

Mukautetun resurssin määritelmä

Mukautetun resurssin määritelmä (engl. Custom Resource Definition), tai lyhyesti CRD, kertoo nimensä mukaan Kubernetesille, minkälainen mukautettu resurssi on. Muiden resurssimäärittelyiden kaltaisesti CRD on yaml-tiedosto, joka sisältää resurssin rakenteen. Kuvio 17 on esimerkki mukautetun resurssin määrittelystä. Taulukko 3 esittää, mistä elementeistä mukautetun resurssin määrittely koostuu.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: prometheuses.monitoring.coreos.com      ❶
spec:
  group: monitoring.coreos.com                  ❷
  names:
    kind: Prometheus                            ❸
    plural: prometheuses                        ❹
  scope: Namespaced                             ❺
  versions:                                     ❻
  - name: v1                                    ❼
    storage: true                               ❽
    served: true                                ❾
    schema:
      openAPIV3Schema: ....                     ❿
```

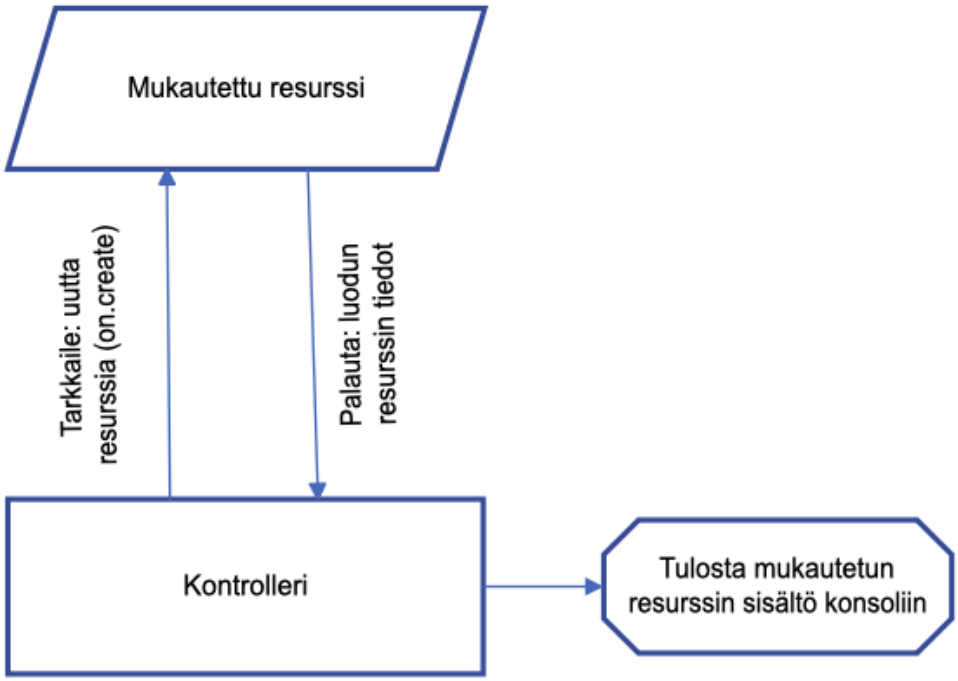
KUVIO 17. CRD-rakenne (Ibryam & Hub, 2019)

TAULUKKO 3. Kuvan 17 CRD-rakenteen selitteet.

Kentän numero	Selite
1	Nimi.
2	Rajapintaryhmä, johon se kuuluu.
3	Tyypin nimi (engl. Kind), jota käytetään tyypin instanssin tunnistamisessa.
4	Tyypin nimi monikkomuodossa, käytetään kun tyypin instansseja listataan.
5	Kohde, johon instansseja voi luoda (namespace tai cluster-wide).
6	Resurssin saatavilla olevat versiot.
7	Tuetun version nimi.
8	Yhden version täytyy olla ”storage”-versio, jota käytetään resurssin määritelmän säilyttämiseen alustaan.
9	Julkaistaanko kyseinen versio klusterin rajapinnassa.
10	OpenAPI V3 -skeema validointia varten.

3.3 Minimalistinen kontrolleri

Tilaa seurataan kontrollereilla, jotka sovittavat muutokset tilassa niille määritellyille kohderesurs-seille. Tämä kontrollerin ja resurssin yhdistelmä muodostaa Operaattorin, Operaattori-ohjelmointi-kaavaa on käsitelty yksityiskohtaisemmin luvussa 3.2. Kopf-ohjelmointikehys (engl. Framework) tarjoaa valmiit funktiot ja komentorivityökalun operaattorien kehitykseen. Kopf on suunniteltu hel-posti lähestyttäväksi työkaluksi, joka suoraviivaistaa operaattorien kehittämistyötä. Kuvioissa 18 ja 19 esitellään minimalistinen toteutus, joka seuraa resursseja tyyppiä kapseli (Pod), ja tulostaa konsoliin resurssin sisällön, kun uusi kyseisen tyyppinen resurssi luodaan.



KUVIO 18. Minimalistisen toteutuksen kaavio.

first-operator > src > first-operator.py > ...

```python
import kopf

@kopf.on.create('pod')
def create_fn(body, **kwargs):
    print(f"Käsittelijän on laukaissut resurssi, jonka sisältö on seuraava: {body}")

```

KUVIO 19. Minimalistinen toteutus Kopf-ohjelmointikirjaston avulla.

Minimalistisen operaattorin testistä syntynyt tuloste yhden kapselin sisällöstä löytyy liitteenä (liite 1). Resurssin tiedot ovat sanakirjamuodossa (engl. Dictionary). Testissä klusteriin lisättiin hello-minikube-julkaisu, joka sisältää pienen testiohjelman, joka julkaistaan kapseliin.

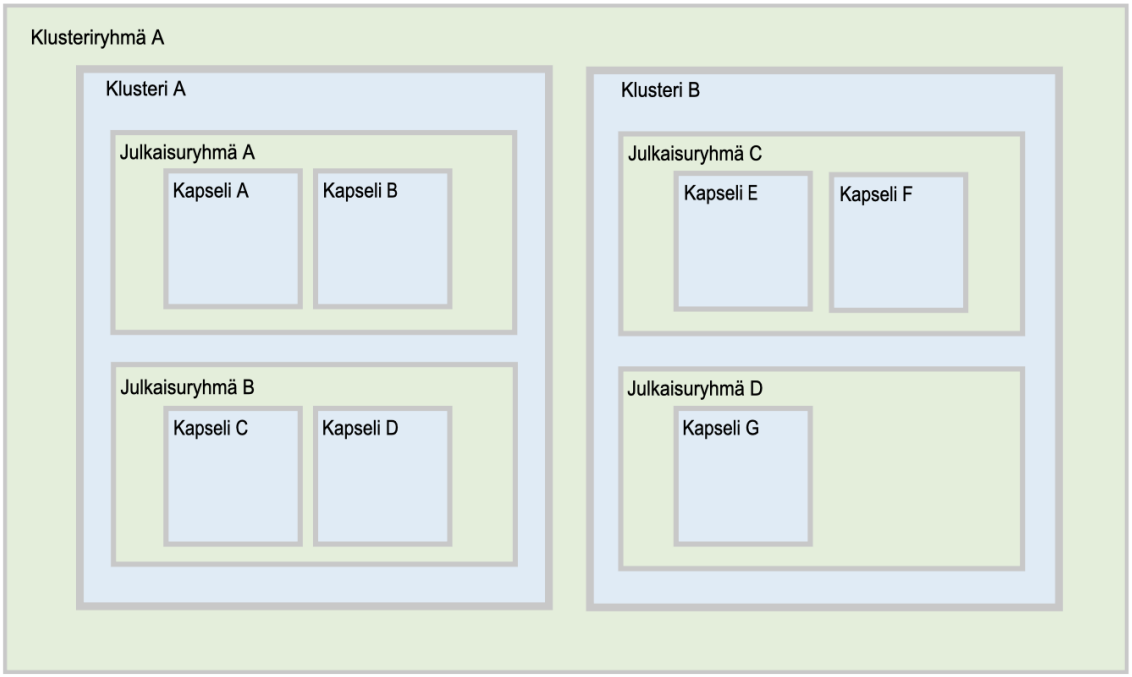
3.4 Moniklusteriympäristön julkaisu- ja hallintajärjestelmän ymmärtäminen

Julkaisu- ja hallintajärjestelmä on rakennettu GitOps-mallia mukaillen, joten se tarkkailee muutoksia versiohallinnassa, ja sovittaa muutokset hallinnoimiinsa klustereihin. GitOps-mallin mukaan versiohallinnassa on määritelty järjestelmän tila, jota kutsutaan halutuksi tilaksi (engl. Desired state). Julkaisujenhallintajärjestelmä ylläpitää tietoa järjestelmän nykytilasta ja vertaa sitä haluttuun tilaan, jos tilat eroavat toisistaan järjestelmä pyrkii sovittamaan halutun tilan. Lisätietoa GitOps-mallista on luvussa 2.1.2.

Julkaisu- ja hallintajärjestelmä koostuu Operaattoreista, jotka seuraavat järjestelmän sisäisten resurssien tilaa, kuten julkaisut, julkaisuryhmät ja klusterit, sekä järjestelmän ulkopuolista tilaa eli versiohallintaa. Samaa Operaattori-ohjelmointikaavaa hyödynnetään toteutuksessa näkyvyyden rakentamiseen. Operaattori-toteutuksessa rakennetaan ryhmä kontrollereita, jotka seuraavat niille osoitettujen resurssien tilaa ja sen muutoksia. Näistä muutoksista syntyvästä tiedosta rakennetaan järjestelmän toiminnasta kokonaiskuva, joka lopuksi lähetetään klusterin ulkopuolelle, missä sitä voidaan hyödyntää esimerkiksi monitoroinnissa.

3.5 Julkaisu- ja hallintajärjestelmän mukautetut resurssit

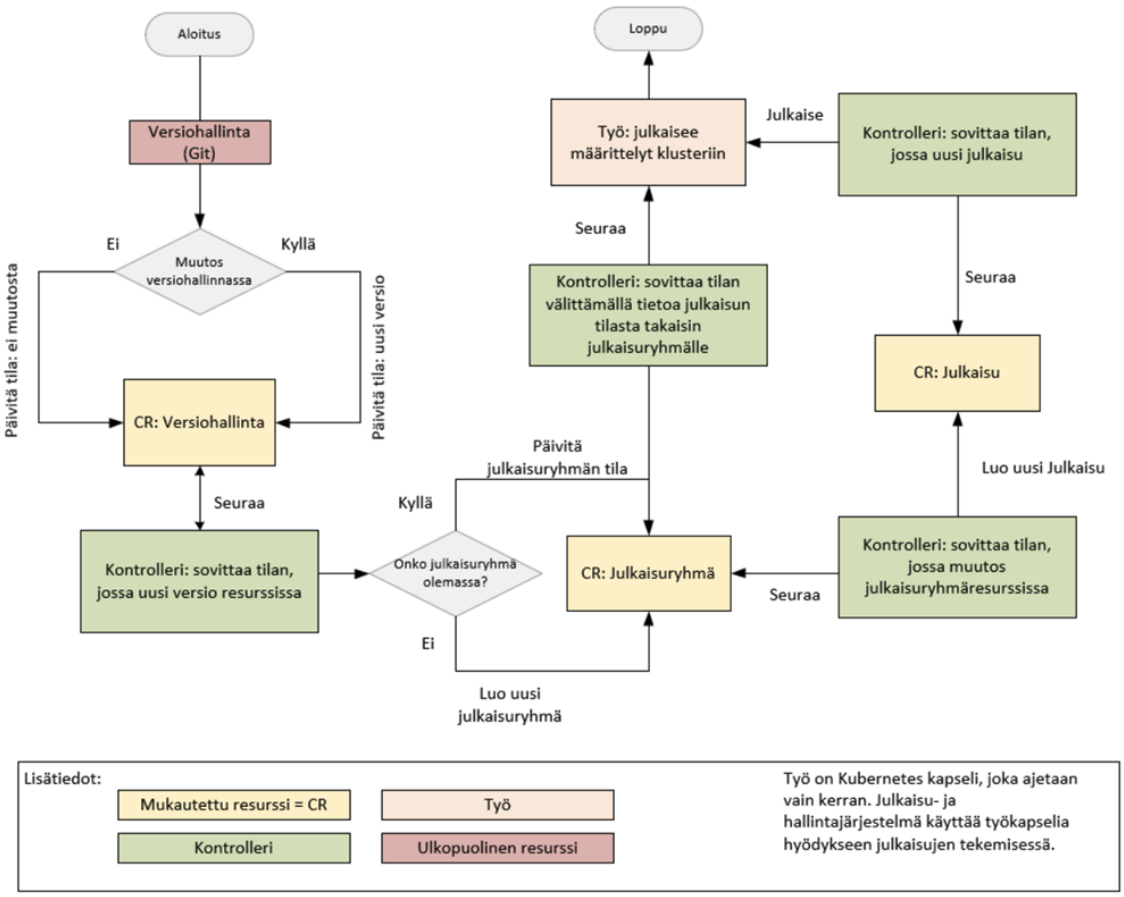
Julkaisu- ja hallintajärjestelmä hyödyntää useita erityyppisiä mukautettuja resursseja oman toiminnallisuutensa toteuttamiseksi. Kuviossa 20 on esitetty, kuinka nämä resurssit lisäävät kaksi uutta abstraktion kerrosta klusteriin: julkaisuryhmä ja klusteriryhmä. Klusteriryhmä kokoaa kaikki ympäristön klusterit, joihin julkaisemista järjestelmä hallinnoi. Järjestelmä ylläpitää myös tietoja kaikkien klusterien tilasta mukautetussa resurssissa.



KUVIO 20. Kaaviossa on esitetty vihreällä värillä abstraktiot, jotka hallintajärjestelmä tuo lisäksi. Sinisellä on kuvattu alkuperäiset Kubernetesin abstraktiot ja niiden sijoittuminen hallintajärjestelmän abstraktioihin.

Julkaisuryhmä muodostuu julkaistavista resursseista, jotka luetaan samasta versiohallinnan rekisteristä. Julkaisuryhmän sisältö julkaistaan aina kokonaisuudessaan; kun julkaisuryhmän tilaan päivitetään muutos, hallintajärjestelmä vie muutoksen kaikkiin julkaisuryhmälle osoitettuihin klustereihin. Uusi julkaisuryhmä luodaan, kun havaitaan uusi versiohallinnan rekisteriä edustava resurssi. Julkaisuryhmää päivitetään aina julkaisun tilan ja versiohallinnassa havaitun muutoksen yhteydessä.

Versiohallinnan rekisteriä edustaa oma mukautettu resurssi, joka säilyttää sisällään versiohallinnan tilaa. Järjestelmällä on kaksi tapaa seurata versiohallinnan muutoksia pollaus ja webhook-integraatio, joista pollaus on oletusasetus. Jokaisella pollauksella järjestelmä päivittää versiohallintaa edustavan mukautetun resurssin. Kuviossa 21 on esitelty yksinkertaistettu malli, kuinka järjestelmän komponentit toimivat yhteen muutosten julkaisussa klustereihin.



KUVIO 21. Yksinkertaistettumalli klusterihallintajärjestelmän toimintaperiaatteesta.

3.6 Operaattori-toteutuksen suunnittelu

Osiossa 3.4 käsiteltiin sitä, kuinka Operaattori-ohjelmointikaavaa hyödyntämällä voimme rakentaa ohjelmalogiikka, jonka avulla voidaan kerätä tietoa klusterin sisäisten resurssien toiminnasta ja tilasta. Käytännön toteutuksessa täytyy kuitenkin ottaa huomioon vielä asioita, kuten operaattorin julkaisu, käyttöoikeudet sekä operaattorien tilan ja muistin hallinta. Kubernetes-klusterissa Operaattori toimii kapselin sisällä, kuten kaikki muutkin työkuormat. Kubernetes voi poistaa tai luoda uusia kapseleita, joten Operaattorien tilaa pitää pystyä hallitsemaan myös tällaisissa tilanteissa.

3.6.1 Arkkitehtuuri

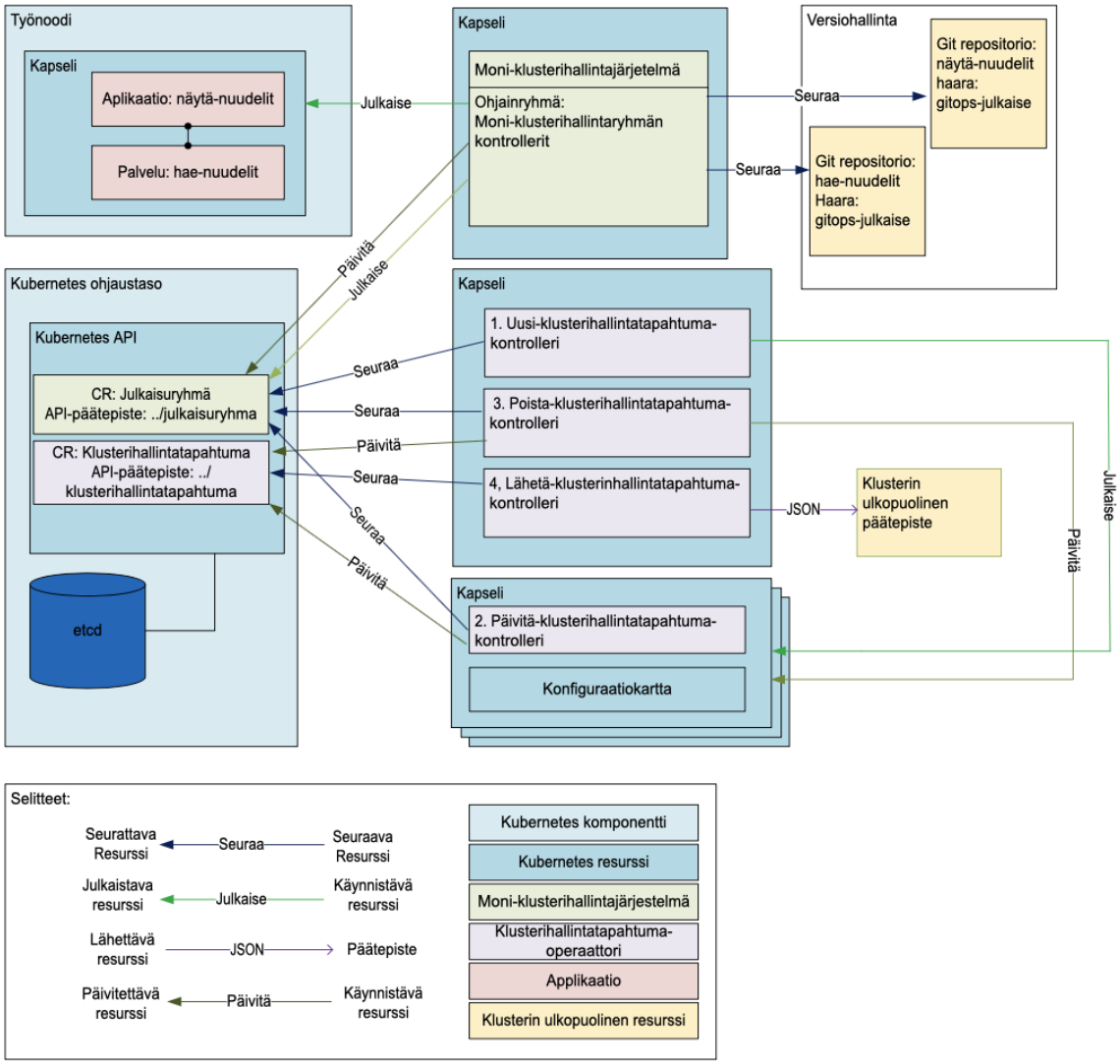
Operaattori-toteutuksen arkkitehtuuri koostuu mukautetusta resurssista Klusterihallintatapahtuma (engl. Cluster management event) sekä neljästä eri kontrollerista, jotka muodostavat yhdessä

Operaattorin. Operaattori julkaistaan yhdessä kapselissa klusteriin, jossa toimii Kubernetesin ohjaustaso. Operaattorin kontrolleri seuraa julkaisu- ja hallintajärjestelmän omaa mukautettua resurssia, joka on julkaisuryhmä. Kun uusi julkaisuryhmä luodaan, uusi-klusterihallintatapahtuma-kontrolleri luo uuden klusterinhallintatapahtuma-resurssin ja julkaisee kapselin, joka sisältää julkaisuryhmän tilaa seuraavan päivitä-klusterihallintatapahtuma-kontrollerin. Päivitä-klusterihallintatapahtuma-kontrollereita on aina yksi jokaista julkaisuryhmää kohden, julkaisuryhmän tiedot julkaistaan kontrollerin kanssa konfiguraatiokarttaresurssin muodossa. Taulukko 4 kuvaa kaikki Operaattorin kontrollerien tehtävät.

Kuviossa 22 esitellään operaattoritoteutuksen arkkitehtuuria ja sen istumista Kubernetesin alkuperäisiin komponentteihin ja resursseihin. Kuviosta käy ilmi myös, miten kontrollerit hyödyntävät moniklusterihallinta- ja julkaisujärjestelmän omien resurssien tilaa.

TAULUKKO 4. Operaattori hyödyntää neljää eri kontrolleria halutun toiminnallisuuden muodostamiseksi.

Kontrolleri	**Tapahtuma**	**Seurattava kohde**	**Tehtävä**
Uusi-klusterinhallintatapahtuma-kontrolleri	Uusi	Julkaisuryhmä	Luo uusi klusterihallintatapahtuma resurssi. Luo uusi päivitä-klusterinhallintatapahtuma-kontrolleri.
Päivitä-klusterinhallintatapahtuma-kontrolleri	Päivitys	Julkaisuryhmä	Päivitä klusterinhallintatapahtuma resurssi.
Poista-klusterinhallintatapahtuma-kontrolleri	Poisto	Julkaisuryhmä	Poista klusterinhallintatapahtuma resurssi. Poista päivitä-klusterinhallintatapahtuma-kontrolleri.
Lähetä-klusterinhallintatapahtuma-kontrolleri	Päivitys	Klusterinhallintatapahtuma	Lähetä JSON-viesti klusterin ulkopuoliseen päätepisteeseen.



KUVIO 22. Operaattori-toteutuksen arkkitehtuurikuvaus.

3.6.2 Mukautetun resurssin määrittely

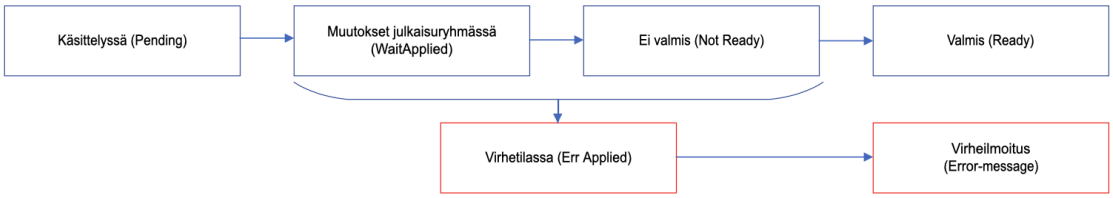
Mukautetun resurssin määrittelyssä on otettava huomioon julkaisu- ja hallintajärjestelmän julkaisuryhmän elinkaari. jota käsiteltiin tarkemmin luvussa 3.3.1. Kun järjestelmä saa uuden versiohallinnan osoituksen, se luo uuden julkaisuryhmän, jonka uusi-klusterihallintatapahtuma-kontrolleri rekisteröi. Se sovittelee uuden julkaisuryhmän julkaisemalla päivityksestä huolehtivan kontrollerin ja luomalla uuden klusterihallintatapahtuma-resurssin. Kun kyseiseen julkaisuryhmään julkaistaan uusia muutoksia, julkaisu- ja hallintajärjestelmä päivittää julkaisuryhmän tilan, jonka päivitä-klusterihallintatapahtuma-kontrolleri rekisteröi ja välittää sille osoitettuun klusterihallintatapahtuma-resurssiin. Mukautetun resurssin suunnittelussa oli otettava huomioon julkaisuryhmän käyttäytyminen. Taulukko 5 esittelee klusterihallintatapahtumaresurssin kentät: nimi, nimiavaruus, kohdeklusteri ja luotu, jotka asetetaan vain ja ainoastaan, kun uusi klusterihallintatapahtuma luodaan.

TAULUKKO 5. Mukautetun resurssin kentät.

Kenttä	Tyyppi	Sisältö	Pakollinen kenttä (kyllä/ei)
Nimi (Name)	Teksti	Julkaisuryhmälle määritelty nimi.	kyllä
Nimiavaruus (Namespace)	Teksti	Julkaisuryhmän nimiavaruus.	kyllä
Kohdeklusteri (Target-cluster)	Teksti	Nimiavaruus johon julkaisuryhmä on kohdennettu.	kyllä
Julkaisut (Deployments)	Lista	Yksittäisen julkaisun tiedot ja tila. Lista sisältää julkaisuolioita, joiden tiedot purettu taulukkoon 3.	kyllä
Luotu (Created)	Teksti	Aikaleima, kun resurssi on luotu.	kyllä

Julkaisut-kentän julkaisuolio sisältää yksittäisen julkaisun tiedot. Sitä päivitetään aina julkaisuryhmäresurssiin päivitettyjen muutoksien mukaisesti. Julkaisut-kenttä sisältää listan, johon aina uuden julkaisun alettua lisätään uusi julkaisuolio. Näin ollen klusterihallintatapahtumaresurssi kerää myös historiatietoja tehdyistä julkaisuista.

Taulukon 6 rivit, jotka on merkattu vihreällä, kuvaavat julkaisuryhmän välittämää tilatietoa julkaisusta. Tämä kertoo, ollaanko uutta julkaisua käynnistämässä, missä vaiheessa julkaisu on ja onko julkaisu onnistunut vai epäonnistunut. Näin ollen julkaisun tilaa voidaan seurata vaihe vaiheelta. Kuvio 23 esittelee normaalin julkaisun aikana tapahtuvat tilan muutokset, ja jos julkaisu jää virhetilaan, myös virheilmoitus otetaan talteen.



KUVIO 23. Julkaisuryhmän tilaa seurataan klusterihallintatapahtumaresurssin julkaisuoliossa hyödyntämällä seuraavia julkaisuryhmän tilan muutoksia. Jos julkaisussa tapahtuu virhe, niin virheilmoitus tallennetaan omaan virheilmoitus kenttään.

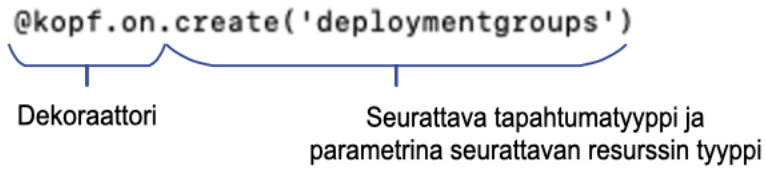
TAULUKKO 6. Mukautetun resurssin julkaisujen tilaa kuvaavan Deployments -kentän tietueet. Julkaisun tilaa esittävät kentät, joita päivitetään julkaisun edistyessä, on maalattu vihreäksi.

Kenttä	Tyyppi	Sisältö	Pakollinen kenttä (kyllä/ei)
Virhetilassa (Err Applied)	Olio	Osa julkaisuista on päätynyt virhetilaan. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Muokattu (Modified)	Olio	Klustereihin on julkaistu muutokset ja ne ovat valmiina, mutta julkaisun jälkeen niihin on tehty käsin muutoksia. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Ei valmis (Not Ready)	Olio	Julkaisuryhmät on julkaistu, mutta kaikki julkaisut ei ole vielä valmiina. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Ei synkronissa (Out of Sync)	Olio	Klustereihin on julkaistu muutokset, mutta niiden valmiutta ei tiedetä vielä. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Käsittelyssä (Pending)	Olio	Julkaisuryhmää käsitellään/tila on tuntematon. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Valmis (Ready)	Olio	Kaikki resurssit on julkaistu ja valmiina. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Muutokset julkaisuryhmässä (Wait Applied)	Olio	Muutokset on synkronoitu julkaisuryhmään ja odottavat vielä lähetystä klustereihin. Tila on 1 jos ehto toteutuu, ja 0 jos ei.	kyllä
Klusterin tila (Cluster-state)	Olio	Julkaisun tila klustereissa, muotoa: valmiina kpl/kokonaislukumäärä.	kyllä
Julkaistun resurssin versio (Deployed-resource-version)	Teksti	Julkaistun resurssin versio.	kyllä
Virheilmoitus (Error-message)	Teksti	Jos julkaisussa syntyy virhe sen virhesanoma.	ei
Sukupolvi (Generation)	Luku	Montako kertaa resurssia on päivitetty.	kyllä
Julkaisu alkoi (Started)	Teksti	Koska viimeisin julkaisu on aloitettu.	kyllä
Julkaisu päivitetty (Updated)	Teksti	Koska julkaisun tilaa on viimeksi päivitetty.	kyllä

3.7 Operaattori-toteutuksen rakentaminen

Operaattorin rakentamiseen käytetään Kopf-ohjelmointikehystä, jonka avulla toteutettiin kontrollerit ja niiden toiminnallisuudet. Kopf sisältää valmiina erityyppisiä seurattavia tilan muutoksia, kuten: uusi resurssi, resurssin päivitys, resurssin poisto ja resurssin tapahtuma. Kuviossa 24 on esitelty yksinkertainen Kopf-ohjelmointikehyksen avulla tuotettu funktio, joka reagoi uusien resurssien luontiin. Kontrollerien suunnittelussa on pyritty toteuttamaan yksittäinen vastuu -periaatetta

(engl. Single responsibility principle), joten jokaisella mukautetun resurssin toiminnolla on oma kontrolleri.



KUVIO 24. Yksinkertainen funktio seuraamaan uusia julkaisuja klusteriin.

Kubernetes-rajapintapalvelinta vaativat toiminnot toteutettiin hyödyntämällä kontrollerin sisällä Kubernetes API client -kirjastoa. Kubernetes API client -kirjasto tarjoaa suuren määrän valmiita funktioita, joiden avulla voi hyödyntää Kubernetes-rajapintapalvelinta. Luvussa 3.5 on esitelty Kubernetes API client -kirjaston implementaatiota käytännön toteutuksessa, uusi-klusterinhallintata-pahtuma-kontrolleri.

3.7.1 Operaattori-toteutuksen kontrollerit ja resurssin elinkaari

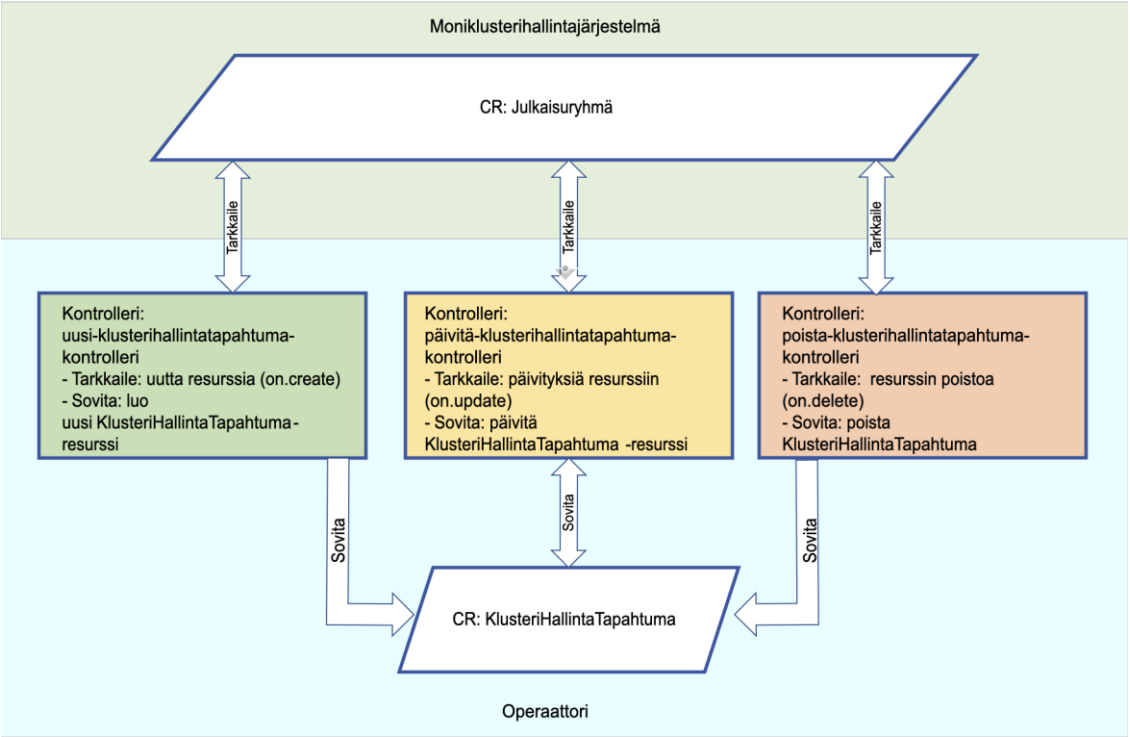
Operaattorin tulee seurata uusien julkaisuryhmien julkaisua klusteriin. Kun uusi julkaisuryhmä luodaan, Operaattori reagoi muutokseen luomalla klusteriin uuden mukautetun resurssin. Kubernetes Python client -ohjelmointikirjastoa hyödynnetään uuden resurssin ja päivitä-klusterihallintata-pahtuma-kontrollerin julkaisemiseen kontrollerin sisältä. Kubernetes Python client on avoimen lähdekoodin kirjasto, joka paketoi Kubernetes-rajapintapalvelimen rajapintakutsuja suoraan koodista kutsuttaviksi funktioiksi.

Operaattorin suunnittelussa hyödynnetään periaatteita olio-ohjelmoinnin SOLID-ohjelmointikäytänteistä, etenkin yksivastuuperiaatetta (engl. The single-responsibility principle) ja avoin-suljettu-periaatetta (engl. Open-closed principle). Yksivastuuperiaatteessa määritellään, että yksittäinen luokka, tässä tapauksessa kontrolleri, pitäisi olla muuttumaton ja sillä pitäisi olla vain yksi tehtävä. Avoin-suljettu-periaate määrittelee, että olion, tässä tapauksessa kontrollerin, pitäisi olla avoin sen toiminnallisuuden jatkamiselle, mutta suljettu muutokselle. Käytännössä tämä tarkoittaa sitä, että jokaista resurssia ja resurssin tilan muutosta havainnoimaan luodaan oma kontrolleri. Nämä kontrollerit päivittävät yhtä resurssia, joka edustaa julkaisu- ja hallintajärjestelmän tilaa.

Mukautettu resurssi on rakennettu niin, että resurssin perustiedot ovat julkaisuryhmälle määritelty nimi, julkaisuryhmän nimiavaruus, julkaisuryhmän kohdeklusteri ja resurssin luomisaika. Näitä kenttiä käsitellään vain uuden resurssin luomisen yhteydessä. Lisäksi resurssilla on Julkaisut-niminen lista, joka sisältää tiedot kaikista yksittäisistä julkaisuista, joita julkaisuryhmä on tehnyt. Tarkempi määrittely mukautetusta resurssista löytyy luvusta 3.4.2.

Uusi julkaisuryhmä syntyy, kun julkaisu- ja hallintajärjestelmälle annetaan uusi versiohallinnan osoitus. Uusi-klusterihallintatapahtuma-kontrolleri seuraa uusia julkaisuryhmiä, kun uusi julkaisuryhmä luodaan, kontrolleri luo uuden klusterihallintatapahtuma-resurssin, johon se välittää julkaisuryhmän perustiedot ja lähtötilan. Julkaisuryhmä päivitetään aina, kun versiohallinnassa havaitaan muutos ja sen hallitsemien julkaisujen tila muuttuu. Päivitä-klusterihallintatapahtuma-kontrolleri seuraa julkaisuryhmän päivitystä, ja päivittää julkaisuryhmän muutokset klusterihallintatapahtuma-resurssille. Mukautetun resurssin elinkaareen kuuluu myös se, että resurssi voidaan poistaa, joten poista-klusterihallintatapahtuma-kontrolleri lisättiin seuraamaan poistuvia julkaisuryhmiä. Se poistaa klusterihallintatapahtuma-resurssin, kun julkaisuryhmä poistetaan.

Kuviossa 25 on esitelty kontrollerit, jotka seuraavat julkaisuryhmän tilaa. Operaattori pystyy nyt luomaan uusia klusterihallintatapahtuma-resursseja. Se pystyy myös päivittämään resurssia, kun julkaisuryhmässä havaitaan muutoksia, ja myös poistamaan resurssin, jotta se ei jää orvoksi, kun julkaisuryhmä poistetaan.



KUVIO 25. Julkaisuryhmän poistoa varten lisättiin poista-klusterihallintatapahtuma-kontrolleri. Nyt koko resurssin elinkaari on hallinnassa, kun KlusteriHallintaTapahtuma poistetaan poistettavan julkaisuryhmän mukana.

3.7.2 Uusi-klusterinhallintatapahtuma-kontrolleri

Käsitellään seuraavaksi uusi-klusterinhallintatapahtuma-kontrollerin rakenne sen lähdekoodista käsin.

Ensimmäisessä kohdassa (kuvio 26, kohta 1) kontrollerille on määriteltävä käytettävät Python-kirjastot. Tässä kontrollerissa käytetään neljää eri kirjastoa:

- ”Datetime”, jota käytetään aikaleimojen luomiseen
- ”Kopf”, joka tarjoaa Kopf-ohjelmointikehyksen kirjastot operaattorin rakentamiseen
- ”Kubernetes”, joka sisältää Kubernetes API client -kirjaston, jota käytetään rajapintakutsujen tuottamiseen
- Yaml, jota käytetään yaml-dokumenttien käsittelyssä.

Seuraavaksi määritellään ”@kopf”-dekoraattoriin seurattava toiminto (kuvio 26, kohta 2), eli tässä tapauksessa ”on.create”-funktio. ”On.create”-funktio kutsuu käsittelijäfunktiota, kun se havaitsee uuden resurssin, joka on määritelty funktion parametrina. Seurattavan resurssin tyypin voi olla

joko yksikkö tai monikko muodossa. Jos haluttaan seurata kaikkia tapahtumia, jossa luodaan uusi kapseli, tyyppi voi olla esimerkiksi ”pod” tai ”pods”.

Käsittelijäfunktiolle (kuvio 26, kohta 2) määritellään parametrina, mitä tietoa se palauttaa resurssista. Tässä tapauksessa ”body”, joka vastaa koko resurssin sisällöstä, ja ”spec”, joka vastaa vain resurssin spesifikaatioavaimen alta löytyvää tietoa. ”**kwargs” tulee olla aina määriteltynä, sillä se vastaa versioiden taaksepäin yhteensopivuudesta.

Kohdassa kolme (kuvio 26, kohta 3) määritellään uusi olio ”CustomObjectApi” ja asetetaan se ”api”-nimiseen muuttujaan. Kubernetes API client -kirjasto tarjoaa useita eri rajapintaluokkia riippuen halutusta toiminnallisuudesta. Tässä tapauksessa käytetään ”CustomObjectApi” -luokkaa, koska myöhemmin koodissa (Kuva 26, kohta 7) oliota käytetään uuden mukautetun resurssin luomiseen.

Uuden klusterihallintatapahtuman tiedot annetaan kirjasto muotoisessa muuttujassa (kuvio 26, kohta 5 ja 6). Julkaisut olioon alustetaan kaikki julkaisun tilaa kuvaavat muuttujat ja klusterin tilaa kuvaavan ”cluster-state”-muuttujan arvolla nolla. Päivitä-klusteri-tapahtuma-kontrolleri alkaa päivittää julkaisuolion tilaa eli muokkaa arvon nollasta yhdeksi, kun tila saavutetaan. Aloitusaika eli ”Started”-muuttujaan annetaan aikaleima merkitsemään ensimmäistä muutosta julkaisuolioon. Julkaisun tiedot asetetaan resurssin spesifikaatioon; ”spec”-avaimen alle asetetaan myös resurssin: julkaisuaika, nimi ja nimiavaruus. Kohdassa 6 asetetaan myös resurssin rungon (engl. Body) sisältö eli versio rajapinnalle, luotavan resurssin tyyppi ja metadata, joka sisältää resurssin tunnistetietoja.

Lopuksi uusi resurssi luodaan kohdassa 7 käyttämällä ”create_namespaced_custom_object”-funktiota, jolle annetaan parametrina: mukautetun resurssin versio, nimen monikkomuoto (engl. Plural), sovellusryhmä, nimiavaruus ja aiemmin määritelty resurssin runko.

```
from datetime import datetime
import kopf
import kubernetes
from kubernetes.client.rest import ApiException

import yaml


@kopf.on.create('deploymentgroups')
def create_fn(spec, body, **kwargs):

    api = kubernetes.client.CustomObjectsApi()

    name = body.get('metadata').get('name')
    namespace = body.get('metadata').get('namespace')

    deployments = [{
        "Ready": {"state":0},
        "NotReady": {"state":0},
        "WaitApplied": {"state":0},
        "ErrApplied": {"state":0},
        "OutOfSync": {"state":0},
        "Pending": {"state":0},
        "Modified": {"state":0},
        "started": str(datetime.now()),
        "generation": body.get('metadata').get('generation'),
        "cluster-state": {"ready":0, "desired_ready":0}
    }]

    crd_body = {
        "apiVersion": "kopf.dev/v1",
        "kind": "ClusterManagementEvent",
        "metadata": {"name": f"cluster-management-event-{body.get('metadata').get('name')}"},
        "spec": {
            "name": name,
            "namespace": namespace,
            "created": body.get('metadata').get('creationTimestamp'),
            "target-cluster": spec.get('targets')[0].get('clusterGroup'),
            "deployments": deployments,
        },
    }

    try:
        created_resource = api.create_namespaced_custom_object(
            group="kopf.dev",
            version="v1",
            namespace="default",
            plural="clustermanagementevents",
            body=crd_body,
        )
    except ApiException as e:
        print(f"Exception when calling CustomObjectsApi->create_namespaced_custom_object: {e}\n")

    kopf.info(crd_body, reason="New", message="A new resource created")
```

KUVIO 26. Uusi-klusterihallintatapahtuma-kontrollerin lähdekoodi.

3.7.3 Kontrollerin julkaisu kontrollerin sisältä

Klusterin sisällä kontrollerit toimivat kapseleissa. Kontrollerit, jotka seuraavat resurssien luontia ja poistoa, ovat saman kapselin sisällä. Päivityksestä vastaava kontrolleri on aina omassa erillissessä kapselissaan, koska se julkaistaan uusi-klusterihallintatapahtuma-kontrollerin sisältä. Tämä uusi kontrolleri saa oman kapselinsa ja konfiguraatiokartan, joka sisältää seurattavan kohteen tunnistamiseen tarvittavat tiedot. Lisätietoja kontrollerin julkaisusta toisen kontrollerin sisältä on annettu luvussa 3.6.2.

Kuviossa 27 on esitelty otanta uusi-klusterinhallintatapahtuma-kontrollerin lähdekoodista, jossa luodaan uusi päivitä-klusterinhallintatapahtuma-kontrolleri Kubernetes API client -kirjaston avulla.

Kohdassa 1 määritellään julkaistavat resurssit, eli kontrolleri ja konfiguraatiokartta. Python mahdollistaa arvojen injektoinnin dokumenttiin syntaksilla: ”{muuttujan arvo}”. Tätä hyödynnetään konfiguraatiokarttaan arvojen asettamisessa. Kohdassa 3 julkaistaan konfiguraatiokartta ja kohdassa 4 julkaistaan uusi kontrolleri. Kohta 2 osoittaa kuinka konfiguraatiokartta ja julkaisu käyttävät eri luokkaa Kubernetes API client -kirjastosta.

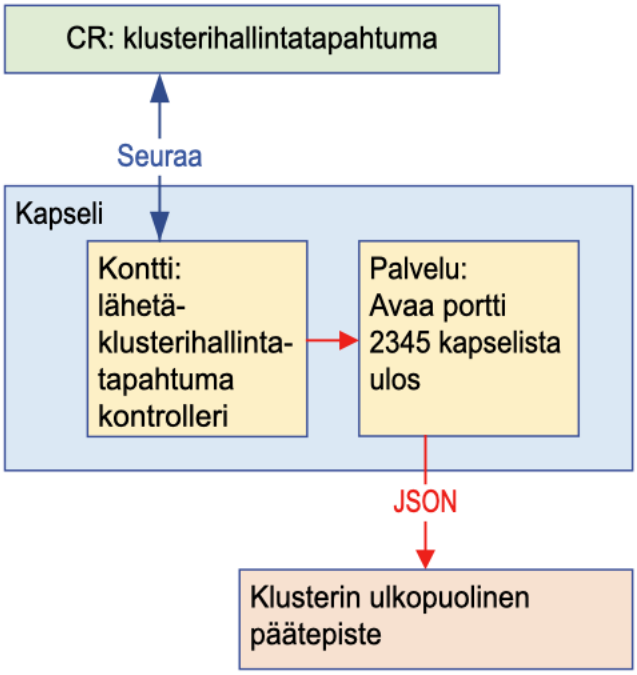
```
❶ controller_doc = yaml.safe_load(f"""
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: {name}-update-operator
        namespace:   <nimiavaruus>
        labels:
          app: {name}-update-operator
      spec:
        replicas: 1
        selector:
          matchLabels:
            app: {name}-update-operator
        template:
          metadata:
            labels:
              app: {name}-update-operator
          spec:
            serviceAccountName: cluster-management-operator-account-new
            containers:
            - name: {name}-update-operator
              image:   <käytettävä levykuva>
              volumeMounts:
                - name: config-volume
                  mountPath: /configs/
            volumes:
            - name: config-volume
              configMap:
                name: cme-config
      """)

  config_doc = yaml.safe_load(f"""
      apiVersion: v1
      kind: ConfigMap
      metadata:
        name:   <nimi>
        namespace:    <nimiavaruus>
      data:
        config.json: |
          {
            "resource_of_interest": "{name}"
          }
      """)

❷ config_map_api = kubernetes.client.CoreV1Api()
  deployment_api = kubernetes.client.AppsV1Api()

  kopf.adopt(config_doc)
  try:
❸     config_map_deploy = config_map_api.create_namespaced_config_map(namespace="         ", body=config_doc)
  except ApiException as e:
      print("Exception when calling CoreV1Api->create_namespaced_config_map: %s\n" % e)

  kopf.info(config_doc, reason="New", message="A new resource created")

  kopf.adopt(controller_doc)
  try:
❹     controller_deploy = deployment_api.create_namespaced_deployment(namespace="           ", body=controller_doc)
  except ApiException as e:
      print("Exception when calling AppsV1Api->create_namespaced_deployment: %s\n" % e)

  kopf.info(controller_doc, reason="New", message="A new resource created")
```

KUVIO 27. Otanta uusi-klusteritapahtuma-kontrollerin lähdekoodista, jossa käytetään ”Kubernetes API client”-kirjastoa, uuden kapselin julkaisemiseksi.

Kubernetes API client -kirjasto mahdollistaa kontrollerin sisältä uusien kontrollerien julkaisun. Tätä ratkaisua käytettiin hyödyksi päivitä-klusterihallintatapahtuma-kontrollerin julkaisussa.

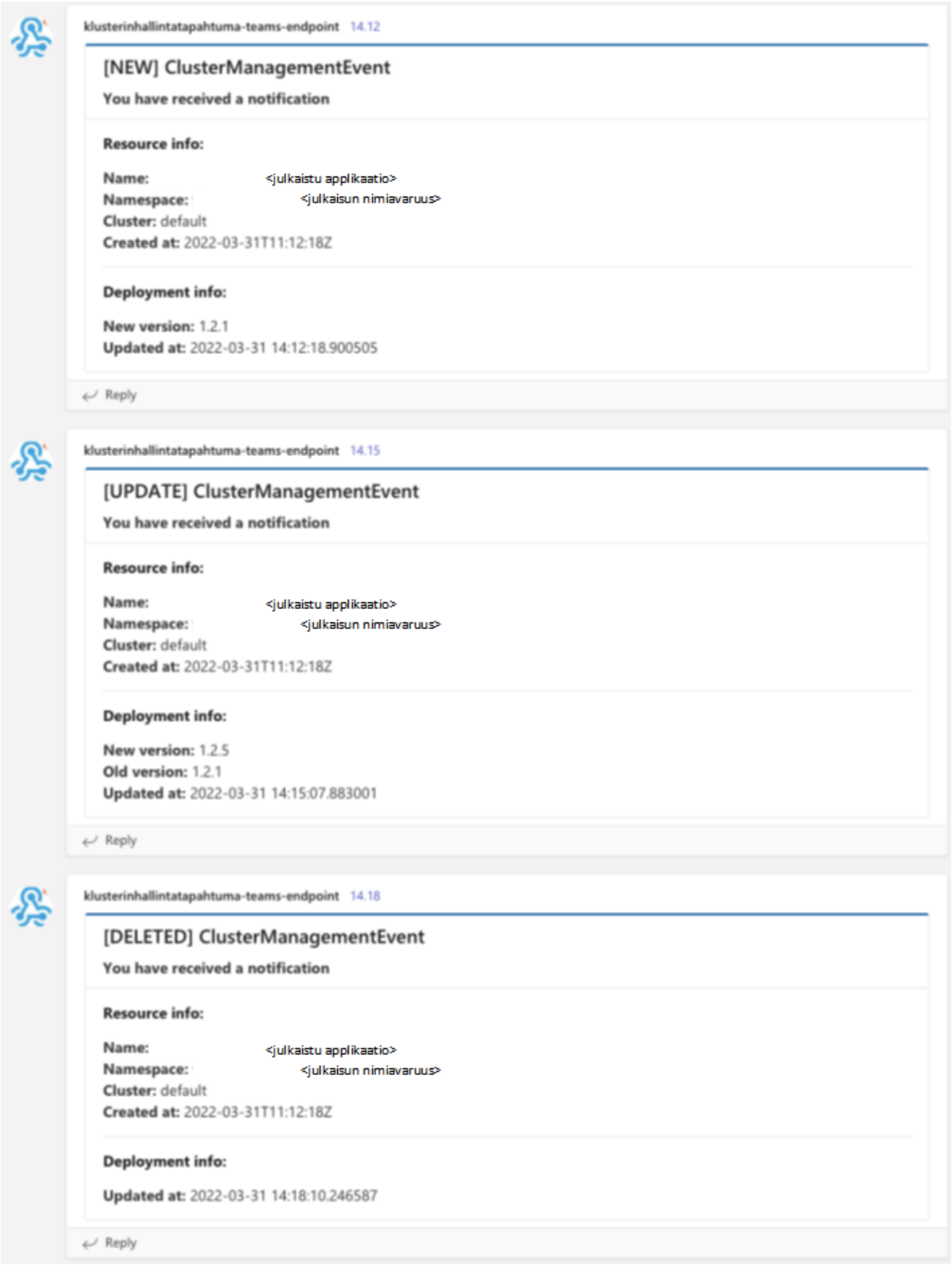
3.8 Tietojen lähetys

Lähetä-klusterihallintotapahtuma-kontrollerin tehtävä on lähettää JSON-sanoma klusterin ulkopuolelle esimerkiksi monitorointijärjestelmän hyödynnettäväksi. Seurattavan resurssin tiedot annetaan julkaistavalle kontrollerille konfiguraatiokartan avulla. Konfiguraation määrittelyyn asetetaan arvot dynaamisesti ja resurssi julkaistaan Kubernetes-rajapintapalvelimen avulla. Kyseinen kontrolleri tulkitsee lisäksi mukautetun resurssin tilaa ja luo sen pohjalta kuvan seurattavan julkaisu- ja hallintajärjestelmän nykytilasta ja toiminnasta. Klusterin ulkopuolelle kommunikoimista varten luotiin palvelu, johon määriteltiin portti, jota kontrolleri voi käyttää. Kuviossa 28 on esitetty tarkemmin tietojen lähetyksen toteutusta. Lisätietoa Kubernetes-resursseista, kuten palvelu, konfiguraatiokartta ja kapseli, löytyy luvusta 2.5.4.



KUVIO 28. Lähetä-klusterihallintatapahtuma kontrolleri seuraa klusteritapahtuma mukautettua resurssia ja välittää muutokset JSON-formaatissa klusterin ulkopuoliseen päätepisteeseen. Kontrolleri hyödyntää palveluresurssia liikenteen reitittämiseksi ulos klusterista.

Esimerkkitoteutuksessa käytettiin hyödyksi Microsoft Teams -ohjelmaa ja sen ”incoming webhook”-integraatiota. Kuviossa 29 on esitelty kolme resurssin eri tilaa Teams-ohjelmassa, jotka ovat: uusi (new), päivitys (update) ja poisto (delete). Päivitä-tilassa näkyy myös versionumeron muutos.



KUVIO 29. Esimerkkitoteutuksessa hyödynnettiin Teams -ohjelman integraatiota, jonka avulla lähetä-klusteritapahtuma-kontrollerin lähettämä tieto resurssin tilan muutoksesta näytettiin omalla Teams-kanavalla.

3.9 Operaattorin julkaisu klusteriin

Operaattoritoteutuksen julkaistava versio toteutetaan Helm-työkalun avulla. Helm on Kubernetes-liitännäinen, joka toimii pakettien hallinta järjestelmänä (engl. Packet Manager). Se mahdollistaa Kubernetes-resurssien eli yaml-dokumenttien paketoinnin niin, että ohjelman voi kuka hyvänsä ladata ja asentaa helposti. Helm-paketteja kutsutaan ”chart”:eiksi, ne ovat käytännössä kokoelmia määrittelydokumentteista, jotka ovat tarpeellisia applikaation julkaisemiseksi Kubernetes-klusteriin. (Cloud Native Computing Foundation 2022e.)

Kuviossa 30 esitetään, kuinka uusi Helm-projekti aloitetaan. ”Helm create”-komento luo uuden projektin ja sen hakemistorakenteen, joka sisältää esimerkki applikaation määrittelyt. Kuviossa 31 näkyy esimerkkiprojektin rakenne. Määrittelyjen sisään on kommentoitu ohjeita, kuten tiedostojen merkitys ja rooli projektissa.

```
→  helm (* |minikube:cme-local)helm create cluster-management-event-operator
Creating cluster-management-event-operator
```

KUVIO 30. ”Helm create” -komennolla voidaan luoda uusi Helm-projekti. Helm luo esimerkki kansiorakenteen ja tiedostot.

```
→  helm (* |minikube:cme-local)tree
.
└── cluster-management-event-operator
    ├── Chart.yaml
    ├── charts
    ├── templates
    │   ├── NOTES.txt
    │   ├── _helpers.tpl
    │   ├── deployment.yaml
    │   ├── hpa.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   ├── serviceaccount.yaml
    │   └── tests
    │       └── test-connection.yaml
    └── values.yaml
```

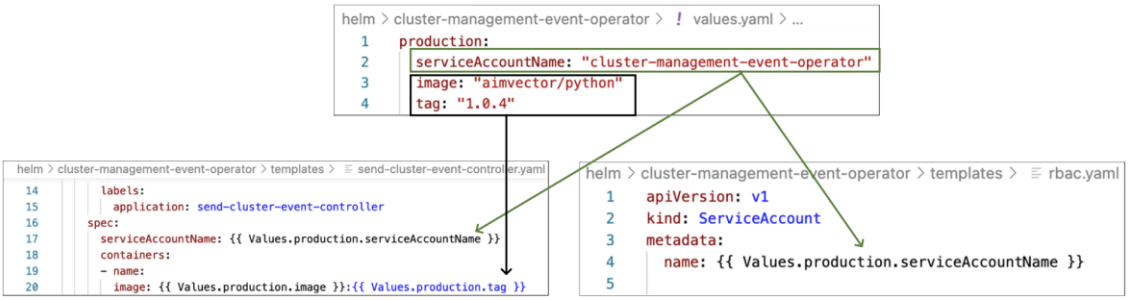
KUVIO 31. Helm create luo esimerkki projektin ja hakemistorakenteen.

Yksinkertaisin tapa viedä omat määrittelytiedostot Helm-projektiin on poistaa templates-kansiosta kaikki muu sisältö lukuunottamatta ”_helpers.tpl” -tiedostoa. Seuraavaksi voidaan siirtää kaikki omat Kubernetes-resurssien määrittelytiedostot kyseisen kansion sisälle. Tämän jälkeen

”Chart.yaml”-tiedosto tulee tyhjentää ja ajaa komento ”helm template <projektin-nimi> <julkaisun-nimi> ”. Tämä komento luo templates-kansion tiedostoista yhden ”Chart.yaml”-tiedoston. Käytännössä projekti on jo käyttökelpoinen tässä vaiheessa, mutta se ei hyödynnä vielä esimerkiksi arvojen injektointia ”values.yaml”-tiedoston avulla.

Kuviossa 32 on esitelty toteutuksen Helm-projektin rakennetta. Kohdassa yksi projektin juurihakemiston nimi määritellään, ja sen on oltava projektin nimi. Templates-kansion alla ovat kaikki projektissa julkaistavat resurssit. Kohdat neljä, viisi ja seitsemän ovat projektin kontrollerit, joiden toiminnot on tarkemmin selvitetty osiossa 3.5.1. Kohdassa kolme määritellään ohjelman mukautettu resurssi. Kohdassa kuusi rooliin perustuvien käyttöoikeuksien (engl. Role based access control) määrittelyt julkaistaan samassa projektissa. Tämä määrittely antaa operaattorille oikeuden käsitellä, seurata ja muokata resursseja klusterin eri nimiavaruuksissa. Kohdan seitsemän ”Service.yaml”-tiedosto on lähetä-klusterihallintatapahtuma-kontrollerin reitityksen määrittelyä varten. Käytännössä se avaa portin klusterista ulos, jotta kontrolleri voi lähettää viestinsä klusterin ulkopuolelle. Viimeisenä on kohta yhdeksän, joka on ”values.yaml”-tiedosto, johon voi asettaa arvoja, joita haluaa injektoida muihin määrittelydokumentteihin. Kuviossa 33 on esitelty, kuinka ”serviceAccountName”-avaimen arvo on määritelty ”values.yaml”-dokumentissa, ja kuinka siihen viitataan kontrollerin määrittelyn ja ”rbac.yaml”-tiedoston sisällä.

KUVIO 32. Operaattorin Helm-hakemisto sisältää kaikki julkaistavat resurssit.



KUVIO 33. "Values.yaml"-tiedosto mahdollistaa arvojen injektoinnin toisiin resurssien määrittely tiedostoihin.

4 TULOKSET

Opinnäytetyön tutkimuskysymyksenä oli, miten konttipohjaiseen moniklusteriympäristön julkaisu- ja hallintajärjestelmän tilaan voidaan rakentaa näkyvyyttä. Tämän lisäksi tavoitteena oli, että kysymystä voitaisiin lähestyä ensisijaisesti Kubernetes-natiivien toiminnallisuuksien avulla ja että saataisiin rakennettua toimiva prototyyppi. Taustaselvityksen ja julkaisu- ja hallintajärjestelmään perehtymisen jälkeen toteutus päätettiin tehdä hyödyntämällä Operaattori-ohjelmointikaavaa.

Operaattori-toteutuksen avulla onnistuttiin rakentamaan näkyvyyttä julkaisu- ja hallintajärjestelmän sisäiseen toimintaan. Operaattorin keräämää tietoa yhdistetään ja rikastetaan ohjelmallisesti, joten seurattavan resurssin tilasta pystyttiin rakentamaan hyvin tarkkaa kuvaa. Esimerkiksi tilan muutoksia voidaan rakentaa usean eri tapahtuman ja tilan yhdistelmän pohjalta. Operaattori tuottaa tilasta JSON-sanoman, joka on laajasti tuettu eri alustoilla. Tämä mahdollistaa sanoman hyödyntämisen monitoroinnin lisäksi useissa muissakin sovelluksissa. Operaattorin tuottama tieto on myös hyvin luettavissa verrattuna esimerkiksi jäljityslokien tuottaman lokitietoon, joka on käytännössä raakadataa, jota on mahdollista käsitellä hakumoottorin avulla. Toteutus onnistuttiin lisäksi paketoimaan ja julkaisemaan klusteriin.

Opinnäytetyön tuloksena syntyi toimiva prototyyppi, jolla pystyttiin tuottamaan näkyvyyttä Kubernetes-klusterin julkaisu- ja hallintajärjestelmään. Lisäksi työ tuotti tärkeää lisätietoa mahdollisuuksista rakentaa näkyvyyttä Kubernetes-klusterin sisällä toimivien resurssien tapahtumiin ja elinkaareen. Toteutuksen Operaattori voi lisäksi toimia alustana muille Operaattori-toteutuksille, koska se sisältää jo perustoiminnallisuudet resurssien tilan muutosten seuraamiseen. Tämän alustan päälle voi rakentaa monenlaista logiikkaa, kuten epäonnistuneita julkaisuja voidaan seurata ja kerätä niistä tietoa, tai jopa reagoida esimerkiksi palauttamalla automaattisesti vanhempi versio julkaistavasta resurssista.

5 POHDINTA

Opinnäytetyön aiheena oli tutkia eri tapoja tuottaa näkyvyyttä konttipohjaiseen moniklusteriympäristön julkaisu- ja hallintajärjestelmän tilaan. Työ aloitettiin mittavalla taustatutkimuksella syventymällä konttiteknologioiden perustuksiin Linux Kernelin toiminnallisuuksissa. Tämän pohjalta työtä jatkettiin perehtymällä tutkimuskehykseen, joka sisälsi Kubernetesin, Helmin, GitOps-julkaisumallin, moniklusteriset ympäristöt, näkyvyyden, mikropalveluarkkitehtuurin ja toteutukseen valitun Operaattori-ohjelmointikaavan. Tutkimusta toteutettiin katsauksena alan kirjallisuuteen, artikkeleihin ja valmistajien dokumentaatioon sekä perehtymällä muihin julkaisuihin, kuten blogikirjoituksiin ja videoihin. Tutkimuksen sekä julkaisu- ja hallintajärjestelmään perehtymisen jälkeen toteutukseen valittiin Operaattori-ohjelmointikaava. Toteutettu Operaattori koostuu neljästä erityyppisestä kontrollerista ja mukautetusta resurssista, jonne julkaisu- ja hallintajärjestelmän tilaa päivitetään. Lisäksi tilan muutokset välitetään klusterin ulkopuolelle standardoidussa tiedostomuodossa, jossa sitä voidaan hyödyntää esimerkiksi monitoroinnissa.

Tutkimus tuotti erityisesti merkittävää lisäarvoa Operaattori-ohjelmointikaavan mahdollistamasta tavasta jatkaa Kubernetes-natiiveja komponentteja omalla ohjelmistologiikalla. Toteutuksessa rakennettu ohjelmistokokonaisuus seurasi sille osoitetun resurssin tilaa ja tilan muutoksia, joiden pohjalta luotiin eri tapahtumia. Näiden tapahtumien sisältämän tiedon pohjalta rakennettiin JSON-muotoisia sanomia, joilla tieto välitettiin klusterin ulkopuolelle. Toteutus muodostaa alustan, joka seuraa resurssin elinkaarta ja sovittelee muutokset resurssien tilassa omalla ohjelmistologiikallaan. Tämän alustan päälle voi rakentaa monenlaista logiikkaa, kuten epäonnistuneita julkaisuja voidaan seurata ja kerätä niistä tietoa, tai jopa reagoida esimerkiksi palauttamalla automaattisesti vanhempi versio julkaistavasta resurssista.

Yhdistämällä Kubernetesin muita toiminnallisuuksia Operaattoriin, voidaan toteuttaa hyvin moninaisia kokonaisuuksia. Tällaisia toiminnallisuuksia ovat esimerkiksi pääsy-webhookit (engl. Admission webhooks), jotka sisältävät mutatoivan webhookin (engl. Mutating webhook) ja validoivan webhookin (engl. Validating webhook). Mutatoivaa webhookia voidaan käyttää merkitsemään Operaattorilla havaittu tila julkaistavaan resurssiin, jonka jälkeen validoivaa webhookia voidaan käyttää arvioimaan kyseinen tila ja päättämään sen pohjalta, voidaanko resurssia julkaista vai ei. Tällainen käyttötapaus voisi olla esimerkiksi julkaisussa ajettujen haavoittuvuustestien tietojen liit-

täminen resurssiin: jos resurssista löytyy haavoittuvuus, sitä ei julkaista. Operaattorilla on mahdollista tarkkailla käynnissä olevia resursseja, joten haavoittuvuuksia voitaisiin mahdollisesti kaivaa esille myös ajonaikaisesti ja merkata mutatoivalla webhookilla.

Operaattori-ohjelmointikaavasta löytyy paljon materiaalia ja kirjallisuutta, mutta käytännössä sen hyödyntämistä pidetään silti haasteellisena. Omassa toteutuksessani erityisesti Operaattorin julkaisu ja käyttöoikeudet aiheuttivat haasteita, minkä takia suosittelisinkin vastaavaan tehtävään ryhtyvää perehtymään Kubernetes-klusterin tietoverkkojen määrittelyyn, applikaation tilan hallintaan ja RBAC-määrittelyyn. Suosittelen myös Helmin hyödyntämistä, koska klusteriin julkaistava-applikaatio sisältää useita määrittelytiedostoja, kuten esimerkiksi julkaisu, konfiguraatiokartta, RBAC, palvelu ja säilyvätaltio. Helmin ”template”-ominaisuuden avulla on helppo varmistaa, että määrittelyjen arvot ovat yhteneväisiä.

Tutkimuksen ja toteutuksen tavoitteiksi asetettiin kartoittaa eri tapoja, joilla voidaan tuottaa näkyvyyttä julkaisu- ja hallintajärjestelmään, Kubernetes-natiivin teknologian hyödyntäminen toteutuksessa, mahdollisuus hyödyntää toimittajan omaa monitorointijärjestelmää tuotetun tiedon esittämisessä ja prototyypin toteutuksen onnistuminen. Operaattori on Kubernetes-natiiviteknologia, joten ensimmäinen tavoite saavutettiin. Tutkimus onnistui selvittämään kaksi vaihtoehtoista tapaa tuottaa tietoa julkaisu- ja hallintajärjestelmän tilasta. Jäljityslokien hyödyntäminen on teoriassa mahdollista, mutta käytännön toteutusta siitä ei tehty. Toteutuksen prototyyppi on todettu toimivaksi, joten myös tämä tavoite saavutettiin. Monitorointijärjestelmän hyödyntämistä ei valitettavasti ehditty todentamaan, mutta JSON-muotoisia sanomia varten monitorointijärjestelmässä on rajapinta, jonka avulla tiedon syöttäminen järjestelmään pitäisi olla mahdollista.

Tutkimuksen pohjalta mielenkiintoisia jatkotutkimuskohteita olisi ylläpitoautomatiikan rakentaminen hyödyntämällä Operaattori-ohjelmointikaavaa, esimerkiksi edellä mainittu validointilogiikan toteuttaminen webhookien ja Operaattorin yhteistoiminnalla.

LÄHTEET

Alibaba Cloud 2020. Getting started with Kubernetes | Kubernetes Container Runtime Interface. Hakupäivä 13.03.2022. https://www.alibabacloud.com/blog/getting-started-with-kubernetes-%7C-kubernetes-container-runtime-interface_596339.

Baier, Jonathan & Sayfan, Gigi & White, Jesse 2019. The Complete Kubernetes Guide. Packt Publis- hing Ltd. Hakupäivä 8.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Beda, Joe & Hightower, Kelsey & Burns, Brendan 2017. Kubernetes: Up and Running. O'Reilly Media Inc. Hakupäivä 1.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Bilgrin, Ibryam & Huß, Roland 2019. Kubernetes Patterns. O'Reilly Media Inc. Hakupäivä 8.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Burns, Brendan, Grant, Brian, Oppenheimer, David, Brewer, Eric & Wilkes, John 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue 16(14), 70-93. Hakupäivä 11.3.2022. ACM Digital Library -tietokanta.

Carrez, Thierry 2020. Below Kubernetes, Demystifying container runtimes. Hakupäivä 12.2.2022. https://www.youtube.com/watch?v=MDsjINTL7Ek.

Cloud Native Computing Foundation 2022a. What is Kubernetes. Versio 1.23. Hakupäivä 1.2.2022https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

Cloud Native Computing Foundation 2022b. Kubernetes Components. Versio 1.23. Hakupäivä 1.2.2022. https://kubernetes.io/docs/concepts/overview/components/.

Cloud Native Computing Foundation 2022c. Workload Resources. Versio 1.23. Hakupäivä 1.2.2022. https://kubernetes.io/docs/concepts/workloads/controllers/.

Cloud Native Computing Foundation 2022d. Using RBAC Authorization. Versio 1.23. Hakupäivä 1.2.2022. https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

Cloud Native Computing Foundation 2022f. Data Model. Versio 3.5. Hakupäivä 1.2.2022. https://etcd.io/docs/v3.5/learning/data_model/.

Cloud Native Computing Foundation 2022e. Charts. Versio 3.8.1. Hakupäivä 12.2.2022. https://helm.sh/docs/topics/charts/.

Dobies, Jason & Wood, Joshua 2020. Kubernetes Operators. O'Reilly Media Inc. Hakupäivä 18.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Domingus, Justin & Arundel, John 2022. Cloud Native DevOps with Kubernetes, 2nd Edition. O'Reilly Media Inc. Hakupäivä 18.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Grunert, Sascha 2019a. Demystifying Containers – Part I: Kernel Space. Suse. Hakupäivä 8.2.2022. https://www.suse.com/c/demystifying-containers-part-i-kernel-space/.

Grunert, Sascha 2019b. Demystifying Containers – Part III: Container images. Suse. Hakupäivä 8.2.2022. https://www.suse.com/c/demystifying-containers-part-iii-container-images/.

Kerrisk, Micheal 2022. Man7. Hakupäivä 1.2.2022. https://man7.org/linux/man-pages/index.html.

Luksa, Marko 2018. Kubernetes in Action. Manning Publications Co. Hakupäivä 1.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Newman, Sam 2021. Building Microservices, 2nd Edition. O'Reilly Media Inc. Hakupäivä 8.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Nikoloff, Jeffrey & Kuenzli, Stephen 2019. Docker in Action, Second edition. Manning Publications Co. Hakupäivä 18.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Red Hat 2018a. A Practical introduction to container terminology. Hakupäivä 23.2.2022. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.

Red Hat 2018b. What is CI/CD? Hakupäivä 23.2.2022. https://www.redhat.com/en/topics/devops/what-is-ci-cd.

RedHat 2021. 3 questions to answer when considering a multi-cluster Kubernetes architecture. Hakupäivä 27.3.2022. https://www.redhat.com/architect/multi-cluster-kubernetes-architecture.

Richardson, Alexis 2018. GitOps – Git push all the things. Hakupäivä 27.3.2022. https://www.youtube.com/watch?v=uWzgmmCzdF4.

Yuen, Billy & Suen, Jesse & Matyushentsev, Alex & Ekenstam, Todd 2021. GitOps and Kubernetes. Manning Publications. Hakupäivä 15.2.2022. O'Reilly learning. Vaatii käyttöoikeuden.

Zelinskie, Jimmy 2018. Kommentti keskusteluketjussa [Discussion] Operator vs. controller pattern. Hakupäivä 1.2.2022. https://github.com/kubeflow/training-operator/issues/300#issuecomment-357527937.

Weave works 2017. GitOps - Operations by Pull Request. Hakupäivä 23.2.2022. https://www.weave.works/blog/gitops-operations-by-pull-request.

Weave works 2022. Guide to GitOps. Hakupäivä 23.2.2022. https://www.weave.works/technologies/gitops/.

Käsittelijän on laukaissut resurssi, jonka sisältö on seuraava: {'metadata': {'name': 'hello-minikube-7bc9d7884c-t5g8k', 'generateName': 'hello-minikube-7bc9d7884c-', 'namespace': 'default', 'uid': 'f0c1e9d1-2828-4407-b5da-1f1f521b5595', 'resourceVersion': '564279', 'creationTimestamp': '2022-03-08T11:45:08Z', 'labels': {'app': 'hello-minikube', 'pod-template-hash': '7bc9d7884c'}, 'ownerReferences': [{'apiVersion': 'apps/v1', 'kind': 'Kopiosarja', 'name': 'hello-minikube-7bc9d7884c', 'uid': '83ae5029-b483-42f9-9c39-8458c58e26b9', 'controller': True, 'blockOwnerDeletion': True}], 'managedFields': [{'manager': 'kube-controller-manager', 'operation': 'Update', 'apiVersion': 'v1', 'time': '2022-03-08T11:45:08Z', 'fieldsType': 'FieldsV1', 'fieldsV1': {'f:metadata': {'f:generateName': {}, 'f:labels': {'.': {}, 'f:app': {}, 'f:pod-template-hash': {}}, 'f:ownerReferences': {'.': {}, 'k:{"uid":"83ae5029-b483-42f9-9c39-8458c58e26b9"}': {}}}, 'f:spec': {'f:containers': {'k:{"name":"echoserver"}': {'.': {}, 'f:image': {}, 'f:imagePullPolicy': {}, 'f:name': {}, 'f:resources': {}, 'f:terminationMessagePath': {}, 'f:terminationMessagePolicy': {}}}, 'f:dnsPolicy': {}, 'f:enableServiceLinks': {}, 'f:restartPolicy': {}, 'f:schedulerName': {}, 'f:securityContext': {}, 'f:terminationGracePeriodSeconds': {}}}}, {'manager': 'Go-http-client', 'operation': 'Update', 'apiVersion': 'v1', 'time': '2022-03-18T21:48:09Z', 'fieldsType': 'FieldsV1', 'fieldsV1': {'f:status': {'f:conditions': {'k:{"type":"ContainersReady"}': {'.': {}, 'f:lastProbeTime': {}, 'f:lastTransitionTime': {}, 'f:status': {}, 'f:type': {}}, 'k:{"type":"Initialized"}': {'.': {}, 'f:lastProbeTime': {}, 'f:lastTransitionTime': {}, 'f:status': {}, 'f:type': {}}, 'k:{"type":"Ready"}': {'.': {}, 'f:lastProbeTime': {}, 'f:lastTransitionTime': {}, 'f:status': {}, 'f:type': {}}}, 'f:containerStatuses': {}, 'f:hostIP': {}, 'f:phase': {}, 'f:podIP': {}, 'f:podIPs': {'.': {}, 'k:{"ip":"172.17.0.4"}': {'.': {}, 'f:ip': {}}}, 'f:startTime': {}}}, 'subresource': 'status'}]}, 'spec': {'volumes': [{'name': 'kube-api-access-dkmnn', 'projected': {'sources': [{'serviceAccountToken': {'expirationSeconds': 3607, 'path': 'token'}}, {'configMap': {'name': 'kube-root-ca.crt', 'items': [{'key': 'ca.crt', 'path': 'ca.crt'}]}}, {'downwardAPI': {'items': [{'path': 'namespace', 'fieldRef': {'apiVersion': 'v1', 'fieldPath': 'metadata.namespace'}}]}}], 'defaultMode': 420}}], 'containers': [{'name': 'echoserver', 'image': 'k8s.gcr.io/echoserver:1.4', 'resources': {}, 'volumeMounts': [{'name': 'kube-api-access-dkmnn', 'readOnly': True, 'mountPath': '/var/run/secrets/kubernetes.io/serviceaccount'}], 'terminationMessagePath': '/dev/termination-log', 'terminationMessagePolicy': 'File', 'imagePullPolicy': 'IfNotPresent'}], 'restartPolicy': 'Always', 'terminationGracePeriodSeconds': 30, 'dnsPolicy': 'ClusterFirst', 'serviceAccountName': 'default', 'serviceAccount': 'default', 'nodeName': 'minikube', 'securityContext': {}, 'schedulerName': 'default-scheduler', 'tolerations': [{'key': 'node.kubernetes.io/not-ready', 'operator': 'Exists', 'effect': 'NoExecute', 'tolerationSeconds': 300}, {'key': 'node.kubernetes.io/unreachable', 'operator': 'Exists', 'effect': 'NoExecute', 'tolerationSeconds': 300}], 'priority': 0, 'enableServiceLinks': True, 'preemptionPolicy': 'PreemptLowerPriority'}, 'status': {'phase': 'Running', 'conditions': [{'type': 'Initialized', 'status': 'True', 'lastProbeTime': None, 'lastTransitionTime': '2022-03-08T11:45:08Z'}, {'type': 'Ready', 'status': 'True', 'lastProbeTime': None, 'lastTransitionTime': '2022-03-16T19:33:36Z'}, {'type': 'ContainersReady', 'status': 'True', 'lastProbeTime': None, 'lastTransitionTime': '2022-03-16T19:33:36Z'}, {'type': 'PodScheduled', 'status': 'True', 'lastProbeTime': None, 'lastTransitionTime': '2022-03-08T11:45:08Z'}], 'hostIP': '192.168.49.2', 'podIP': '172.17.0.4', 'podIPs': [{'ip': '172.17.0.4'}], 'startTime': '2022-03-08T11:45:08Z', 'containerStatuses': [{'name': 'echoserver', 'state': {'running': {'startedAt': '2022-03-16T19:33:34Z'}}, 'lastState': {'terminated': {'exitCode': 255, 'reason': 'Error', 'startedAt': '2022-03-08T12:38:14Z', 'finishedAt': '2022-03-16T19:33:02Z', 'containerID': 'docker://41ce79d8c175014ed1b934f03047bf0bdc1fddfae5c266cf0c7b9670965865e3'}}, 'ready': True, 'restartCount': 3, 'image': 'k8s.gcr.io/echoserver:1.4', 'imageID': 'docker-pullable://k8s.gcr.io/echoserver@sha256:5d99aa1120524c801bc8c1a7077e8f5ec122ba16b6dda1a5d3826057f67b9bcb', 'containerID': 'docker://8aac96f578df434a8a6ef99e368d489b8544583eea595b33aef2d5b269c2d52d', 'started': True}], 'qosClass': 'BestEffort'}, 'kind': 'Pod', 'apiVersion': 'v1'}

Tulosteen tiedot ovat kirjasto -muodossa (engl. Dictionary), joten tietoa ei ole esimerkiksi validia JSONia. JSON-formatointityökalulla tiedon saa kuitenkin muutettua luettavampaan muotoon.