Juho Roiha

GitOps-ympäristön asennus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

5.11.2021

Tiivistelmä

Tekijä:	Juho Roiha
Otsikko:	GitOps-ympäristön asennus
Sivumäärä:	43 sivua + 3 liitettä
Aika:	1.11.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaaja:	Yliopettaja Auvo Häkkinen

Insinöörityön tavoitteena oli suunnitella ja toteuttaa GitOps-ympäristö, jonka avulla julkaistaan Java-sovellus. Ympäristön tarkoituksena on suoraviivaistaa sovelluksen julkaisuprosessia ja helpottaa uuden koodin integrointia sovellukseen.

Työn alussa esiteltiin sovelluskonttiteknologiaa, sovellusten jakelun historiaa ja jatkuvaa integrointia ja toimitusta. Seuraavaksi esiteltiin Kubernetes-ajoympäristö ja GitOps-menetelmä.

Esittelyjen jälkeen työssä määriteltiin ympäristön vaatimukset. Vaatimuksien määrittelyn jälkeen suunniteltiin sovelluksen koonnin ja testit suorittava integraatioputki sekä testatusta sovelluksesta Docker-imagen muodostava ja imagen julkaiseva toimitusputki. Putkien suunnittelun jälkeen valittiin Kubernetes-jakelu suorituskykymittauksen pohjalta. Valinnan jälkeen suunniteltiin klusterin rakenne.

Suunnitteluvaiheen jälkeen suoritettiin ympäristön asennus. Asennus suoritettiin ensin virtuaalikoneeseen paikalliselle työasemalle. Paikallista asennusta käytettiin testiympäristönä. Paikallisen asennuksen jälkeen työssä toteutettiin suunniteltu jatkuvan integroinnin ja toimituksen putki. Testiympäristön asentamisen jälkeen tuotantoympäristö asennettiin virtuaalipalvelimille ja sovellus julkaistiin.

Työssä toteutettiin yksinkertainen Kubernetes-ajoympäristö, jonka toimintaa ohjataan GitLab-versiohallinnan kautta. Sovelluksen testaus, koonti ja asentaminen testi- ja tuotantoympäristöihin onnistuu versiohallintaa käyttämällä. Työn valmistuessa sovellus oli julkaistu toteutetussa ympäristössä. Työssä päästiin asetettuun tavoitteeseen. Työstä opituilla taidoilla klusteriin voidaan helposti lisätä uusia web-sovelluksia.

Avainsanat:	Kubernetes, GitOps, Jatkuva integrointi, Jatkuva toimitus, CI/CD

Abstract

Author:	Juho Roiha
Title:	Deploying a GitOps environment
Number of Pages:	43 pages + 3 appendices
Date:	1 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Auvo Häkkinen, Principal Lecturer

The goal of this thesis was to design and deploy a GitOps environment and use it to deploy a Java application. The environment was designed to streamline the deployment process and to ease the integration of new code into the application. The thesis introduces the reader to application containers and continuous integration and deployment processes with Kubernetes and GitOps.

After defining requirements for the system, a continuous integration pipeline for testing and building the application and a continuous deployment pipeline for building and publishing a Docker image of the application were designed. A Kubernetes distribution was chosen based on a performance test. After designing the pipelines and choosing a distribution, the deployment of the cluster on virtual servers was planned.

Once the design was done, a cluster was first deployed on a virtual machine on a local workstation. This cluster was used as a test environment. The CI/CD pipeline was deployed on GitLab and was used to deploy the application to the test environment. After confirming that the CI/CD pipeline and the intended deployment method worked, a cluster was deployed on Hetzner Cloud virtual servers. The cluster became the production environment and was successfully used to publish the application.

In the study a Kubernetes cluster and a CI/CD pipeline were used to successfully deploy a Java application. Git version control system was used initiate the testing, building and deployment of the application via pull requests. The goal set for the thesis was achieved.

Keywords: Kubernetes, GitOps, Continuous integration, Continuous deployment, CI/CD

Sisällys

Liitteet

Liite 1: Jatkuvan integraation ja toimituksen lähdekoodi

Liite 2: Hyväksyntätestauksen tulokset

Liite 3: Suorituskykymittauksen tulokset

Lyhenteet

ACME *Automatic Certificate Management Environment.* Automaattinen varmenteiden hallintaympäristö. Protokolla, jonka avulla palvelinjärjestelmä voi todistaa varmenteen myöntäjälle hallitsevansa varmennettavaa verkkotunnusta.

API *Application Programming Interface*. Ohjelmointirajapinta. Rajapinta, joka mahdollistaa sovelluksen komponenttien tai järjestelmien välisen tiedonsiirron.

CD *Continuous Deployment*. Jatkuva toimitus. Ohjelmistokehityksen käytäntö, jossa ohjelmistoon tehdyt muutokset julkaistaan mahdollisimman usein.

CI *Continuous Integration.* Jatkuva integrointi. Ohjelmistokehityksen käytäntö, jossa kehitystiimin tekemät koodimuutokset yhdistetään mahdollisimman usein.

TLS *Transport Layer Security*. Kuljetuskerroksen tietoturva. Salausprotokolla, jolla voi salata muun muassa HTTP-liikennettä.

VM *Virtual Machine.* Virtuaalikone. Ohjelmallisesti toteutettu näennäinen tietokone.

1 Johdanto

Web-sovellusten pakkaaminen kontteihin (engl. containerization) on Docker-alustan myötä tullut suosittu tapa koota sovellus ja sen tarvitsemat kirjastot eli riippuvuudet jakelupakettiin, jonka ajoympäristö on eristetty isäntäjärjestelmästä [1]. Sovellusten pakkaaminen kontteihin on mahdollistanut sovellusten helpon jakelun ja asentamisen eri järjestelmiin. Docker-säiliöintialustan suosiota heijastelee muun muassa Docker Hub -palvelussa saatavilla olevien imagejen suuri lukumäärä, joita on yli kahdeksan miljoonaa [2]. Aiemmin julkaistuja käyttöjärjestelmätason virtualisointimenetelmiä ovat muun muassa Linux-VServer, LXC ja Solaris Containers.

Aiemmin sovellusten eristäminen isäntäjärjestelmästä on voitu tehdä suorittamalla sovellukset virtuaalikoneissa tai UNIX-käyttöjärjestelmissä linkittämällä sovelluksen ja sen riippuvuudet omaan polkuunsa ja siirtämällä juurihakemiston chroot-komennolla [3]. Virtuaalikoneet ovat ohjelmallisesti käyttöjärjestelmää ja sovelluskoodia suorittavia näennäisiä tietokoneita. Virtuaalikoneiden haittapuolia ovat hidas sovellusten käynnistysaika sekä muisti- ja levytilahukka, sillä virtuaalikoneessa suoritettavan sovelluksen lisäksi virtuaalikoneeseen on myös asennettava käyttöjärjestelmä. Docker ja muut käyttöjärjestelmätason ratkaisut eristävät prosessit isäntäkäyttöjärjestelmästä käyttöjärjestelmän ominaisuuksia käyttäen. Esimerkiksi Docker eristää prosessin tietokoneen tiedostojärjestelmästä, verkoista ja käyttäjätunnuksista. Useimmilla on mahdollista asettaa myös prosessoriaika- ja muistirajoituksia. Käyttämällä Linux-ytimen prosessineristysominaisuuksia Docker-konteissa suoritettavat sovellukset hukkaavat vähemmän resursseja ja käynnistyvät nopeammin kuin virtuaalikoneet. [1.]

Docker-alustalla voi myös asettaa sovelluksille resurssirajoituksia, ohjata portteja ja käynnistää pysähtyneet säiliöt uudelleen. Se ei kuitenkaan tarjoa työkaluja suurten säiliömäärien hallitsemiseen. Esimerkiksi mikropalveluista koostuvan sovelluksen hallinta pelkästään Dockeria käyttäen vaatii paljon käsin konfi-

gurointia. Säiliöiden määrän kasvaessa käsin konfigurointi muuttuu aikaa vieväksi ja virhealttiiksi prosessiksi. Docker ei myöskään tarjoa työkaluja sovelluksen skaalaamiseksi, sillä Docker suoritetaan yhdellä palvelintietokoneella. [4; 5.]

Sovelluskonttien hallintaan on työkaluja, joista suosituimpia ovat Apache Mesos, Docker Swarm ja Kubernetes. Näillä orkestrointityökaluilla on mahdollista hallita suurta määrää kontteja ja palvelintietokoneita. Orkestrointityökalu hallitsee palvelinsolmuista muodostettua klusteria. Työkalu yleensä huolehtii muun muassa konttien välisestä verkkoliikenteestä, vuorottaa (engl. scheduling) konttien suoritusta, hoitaa palvelinsolmujen kuormanjakoa ja tarjoaa suoritettavia sovelluksia palveluna klusterin ulkopuolelle. [3.]

Kontit ja niiden orkestrointi mahdollistavat sovelluksen ja sen riippuvuuksien irrottamisen isäntäkoneen käyttöjärjestelmästä ja ajoinfrastruktuurin irrottamisen palvelininfrastruktuurista. Tämä on lisännyt DevOps-toimintamallin suosiota. DevOps-toimintamallissa sovelluskehityksen, laadunvarmistuksen ja IT-operaatioiden vastuut tuodaan erillisiltä tiimeiltä ketterän sovelluskehitystiimin hoidettavaksi kokonaan tai osittain. DevOps-tiimi vastaa usein sovelluksen koko elinkaaresta, kuten kehittämisestä, testaamisesta, ajoympäristön konfiguroinnista ja valvonnasta. DevOpsin pyrkimyksinä on lyhentää aikaa, joka kuluu sovelluksen toteuttamisesta sovelluksen toimittamiseen, lisätä toimitustiheyttä, pienentää virheiden lukumäärää ja nopeuttaa virhetilanteesta palautumista. [3.]

Työ tehtiin omana toimeksiantona ja työn tavoitteena oli tuottaa helposti ylläpidettävä kehitysympäristö omaan käyttöön. Ympäristön suunnittelussa pyrittiin siihen, että web-sovelluksien kehittäminen, testaus ja tuotantoympäristöön siirtäminen on mahdollisimman sujuvaa ja vaatii mahdollisimman vähän virtuaalipalvelimien käsin konfigurointia. Rakennettua kehitysympäristöä käytetään Java-sovelluksen kehittämiseen, testaamiseen ja lopulta palvelun tuottamiseen.

Luvuissa 2, 3 ja 4 esitellään käytettäviä menetelmiä ja tekniikoita. Luvuissa 5 ja 6 kuvataan ympäristön suunnittelu ja asennus. Lopuksi varmistetaan, että kehitetty ympäristö vastaa vaatimuksiin ja arvioidaan työn onnistumista.

2 Jatkuva integrointi ja jatkuva toimitus

Jatkuva integrointi (engl. continuous integration, CI) tarkoittaa sovelluskehittäjän tai kehitystiimin tekemien muutosten sisällyttämistä osaksi kehitettävää sovellusta mahdollisimman nopeasti ja automatisoidusti. Jatkuvan integroinnin tavoitteena on yksinkertaistaa integrointia tekemällä se usein ja pienissä erissä. Integroimalla usein muutosten määrä pysyy pienenä ja mahdolliset virheet nousevat esiin aikaisemmin. Aikaisin huomatut virheet ovat usein helpompia ja nopeampia korjata kuin myöhemmin ilmenevät virheet. [6; 7.]

Jatkuva toimitus tarkoittaa automaattisesti tapahtuvaa sovelluksen julkaisua. Sovellus joko saatetaan automatisoidusti käyttövalmiiksi ympäristöön, jossa se voidaan testata ennen tuotantoon vientiä (engl. continuous delivery) tai asennetaan automaattisesti tuotantoympäristöön (engl. continuous deployment, CD). Jatkuvan toimittamisen tavoitteena on julkaista sovellus usein ja pienillä muutoksilla. Jatkuvan toimituksen hyödyt ovat samankaltaisia kuin jatkuvan integroinnin. Usein tehtävät pienet muutokset sovelluksen eri osa-alueisiin tuovat mahdolliset virheet esiin nopeammin. [7.]

2.1 Jatkuva integrointi

Jatkuva integrointi suoritetaan yleensä automatisoidulla putkella (engl. pipeline). Putki on sarja työvaiheita, jossa seuraava työvaihe riippuu edellisestä työvaiheesta. Putken suorittaminen on mahdollista aloittaa joko koodimuutosten viennistä versionhallintaan tai koodihaarojen yhdistämispyynnöstä (engl. pull request, merge request). Putkea suorittava palvelin hakee sovelluskoodin, suorittaa koonnin sekä yksikkö- ja integraatiotestit. Usein putki paketoi sovelluksen ja vie sen pakettivarastoon, josta se voidaan asentaa kohdeympäristöön.

Jenkins on suosittu avoimen lähdekoodin ohjelmisto, jolla on mahdollista automatisoida muun muassa sovelluksen koonti ja testaus. Jenkins on erittäin monipuolinen ohjelmisto ja sen toiminnallisuutta voi laajentaa lisäosilla. Jenkins-putki

koostuu vaiheista, jotka voidaan määritellä sovelluksen versionhallinnassa olevalla Jenkinsfilellä. Jenkinsfile kirjoitetaan Groovy-ohjelmointikielen kaltaisella kielellä [8]. Vaikka Jenkinsille on olemassa Kubernetes-lisäosa, soveltuu vuonna 2018 aloitettu Jenkins X -projekti paremmin Kubernetes-ympäristöön. Jenkins X käyttää putken suorittamisessa Tektonia, joka on Kubernetesta varten kehitetty CI/CD-kehys. Jenkins X osaa myös luoda automaattisesti väliaikaisia ympäristöjä sovelluksen testausta varten. Väliaikaiseen ympäristöön asennettua sovellusta on mahdollista testata laajasti ennen koodihaarojen yhdistämistä. [9.]

Travis CI on alun perin GitHubin kanssa käytettäväksi julkaistu jatkuvan integroinnin työkalu. Myöhemmin tuetuiksi alustoiksi on lisätty myös Bitbucket, Gitlab ja Assembla. Travis CI on Travis CI GmbH:n tuottama palvelu ja asiakkaan omille palvelimille tehtävä asennus hinnoitellaan erikseen. Palvelun ilmaiseen versioon sisältyy kuukaudessa 10000 krediittiä, joka vastaa suunnilleen tuhatta minuuttia suoritusaikaa Linux-ympäristössä. Travis CI:n putket määritellään versionhallinnan juuressa olevalla yaml-dokumentilla. Travis CI tarjoaa valmiin pohjan 35 ohjelmointikielelle, joiden kanssa työkalun käyttöönotto on erittäin suoraviivaista. [10; 11.]

GitLab CI/CD on GitLabin avoimen lähdekoodin jatkuvan integroinnin ja toimituksen tuote. GitLab CI/CD:tä on mahdollista käyttää myös muiden Git-repositorioiden kanssa. GitLab CI/CD on tarjolla SaaS-tuotteena ja omille palvelimille asennettuna. Omille palvelimille asennettaessa voi asentaa joko koko GitLab-versionhallintapalvelimen tai vain GitLab Runner -palvelimen, joka suorittaa CI/CD-putkea. GitLab CI/CD:stä on myös tarjolla palveluna, jonka ilmainen versio sisältää 400 minuuttia suoritusaikaa kuukaudessa. GitLab CI/CD-putket määritellään Travis CI:n tapaan versiohallinnan juuressa olevalla yaml-dokumentilla. [12; 13.]

2.2 Jatkuva toimitus

Jatkuva toimitus suoritetaan integroinnin jälkeen. CI-putki tuottaa yksilöllisellä tunnisteella varustetun paketin, joka asennetaan kohdeympäristöön automaattisesti. Jatkuvassa toimituksessa kohdeympäristön konfiguraatiota muutetaan automaattisesti ottamaan käyttöön uusin versio sovelluksesta. Asennusta ei yleensä tehdä suoraan tuotantoon, vaan niin kutsuttuun staging-ympäristöön. Staging-ympäristö vastaa yleensä tarkasti tuotantoympäristöä. Ympäristössä varmistetaan sovelluksen toimista tuotantoympäristöä vastaavalla konfiguraatiolla. Tavoitteena on varmistaa, ettei asennus tuotantoon aiheuta regressioita ja että sovellus toimii suunnitellusti. Asennuksen jälkeen putki voi suorittaa automatisoidut hyväksyntätestit ja korottaa asennuksen tuotantoympäristöön joko automaattisesti tai manuaalisesti. [7.]

Paketin yksilöitävyys ja muuttumattomuus ovat tärkeitä ominaisuuksia, sillä vain yksilöimällä jokaisen koonnin voidaan varmistua palvelimella suoritettavan koodin versiosta. Tapoja yksilöidä koonnit ovat esimerkiksi koonnin numerointi, muutoksen SHA-tiivistefunktion arvon käyttäminen, semanttinen versiointi ja semanttisen versioinnin ja koontinumeron yhdistäminen. Semanttisen versioinnin käytössä tulee huomioida, että CI/CD-putki ei osaa tehdä semanttista versiointia automaattisesti, vaan muutosten versiointi jää kehittäjän vastuulle. [3; 7.]

Aiemmin esitellyt Jenkins, Travis CI ja GitLab CI/CD pystyvät suorittamaan myös jatkuvan toimituksen osan putkesta. Markkinoilla on lisäksi työkaluja, joiden tarkoituksena on automatisoida vain toimituksen osuus. Esimerkiksi ArgoCD ja FluxCD ovat jatkuvan toimituksen ohjelmistoja, jotka asennetaan Kubernetes-klusterille. Ne tarkkailevat Git-repositorion muutoksia ja asettavat klusterin tilan vastaamaan versiohallinnassa yaml-dokumentein kuvattua tilaa automaattisesti. [3.]

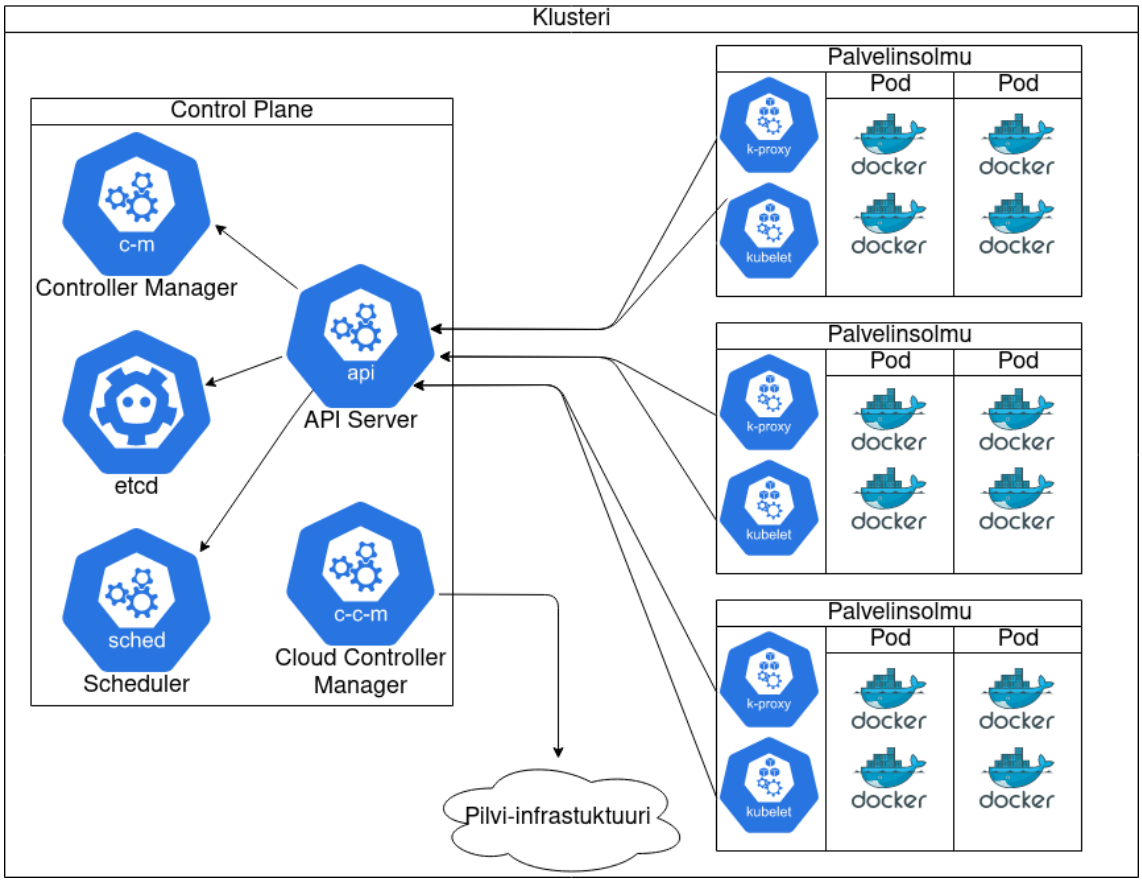
3 Kubernetes - konttien hallintajärjestelmä

Kubernetes on Googlen vuonna 2015 julkaisema avoimen lähdekoodin sovelluskonttien orkestrointiohjelmisto, jota kehittää nykyään Cloud Native Computing Foundation -säätiö. Kubernetes on konttien orkestrointiin (engl. container orchestration) tarkoitettu työkalu, joka luo abstraktiokerroksen palvelintietokoneiden päälle ja mahdollistaa infrastruktuurin määrittelemisen ohjelmallisesti. Palvelintietokoneista muodostetaan klustereita, joilla suoritetaan nimiavaruuksiin jaettuja kontteja. Kubernetes tarjoaa rajapinnat ja työkalut konttien ja infrastruktuurin kuten verkkojen, kuormanjaon ja resurssirajoitusten hallinnointiin. Kun joukko palvelimia on abstrahoitu klusteriksi ja nimiavaruuksiksi, ei sovelluskehittäjän tarvitse enää tietää, millä tai kuinka monella palvelimella sovellus toimii ja hallita palvelimia itse. Sovelluskehittäjän tarvitsee vain määritellä, kuinka montaa rinnakkaista instanssia kontista suoritetaan. Klusterin kuorman kasvaessa on mahdollista liittää klusteriin lisää palvelinsolmuja (engl. node) saatavuuden ja suorituskyvyn takaamiseksi. Myös automaattinen skaalaaminen on mahdollista, jolloin klusteriin voidaan automaattisesti liittää palvelinsolmuja kuorman kasvaessa. Kubernetes-asennusta voi laajentaa myös lisäämällä klusterien määrää. Tämä mahdollistaa suuremman maantieteellisen alueen kattamisen. [14.]

Kubernetesia hallitaan yleisesti deklaratiivisesti yaml-merkintäkielellä kirjoitetuilla tiedostoilla. Deklaratiivisuudella tarkoitetaan tässä yhteydessä sitä, että yaml-kielellä kuvataan tiedostoihin Kubernetesin haluttu tila. Ympäristön halutun tilan saavuttaminen suorittamalla sarja komentoja tietyssä järjestyksessä olisi imperatiivinen lähestymistapa. Deklaratiivisen määrittelyn etu on idempotenttisuus, jolloin komennon lopputulos on aina sama riippumatta komennon suorituskertojen määrästä. Klusterin voidaan ajatella olevan virhetilassa silloin, kun klusterin tila ei vastaa konfiguraatiota. Deklaratiivisen määrittelyn ansiosta klusterin tilan voi palauttaa suorittamalla yhden komennon. [3.]

3.1 Klusterin rakenne

Kubernetes-klusteri koostuu vähintään yhdestä palvelinsolmusta ja palvelinsolmuja hallitsevasta ohjaustasosta (engl. control plane). Ne voivat sijaita samalla palvelimella. Klusterin rakennetta ja komponenttien välistä liikennettä on havainnollistettu kuvassa 1. [3.]



Kuva 1. Havainnekuva Kubernetes-klusterin rakenteesta.

Palvelinsolmut suorittavat kapseleiksi (engl. pod) kutsuttuja konttijoukkoja ja tarjoavat kapseleille resursseja, kuten prosessoriaikaa, keskusmuistia ja levytilaa. Klusterin kapselit ja muut oliot on jaettu nimiavaruuksiin. Nimiavaruuksilla voi ryhmitellä toisiinsa liittyviä olioita, jolloin monesta mikropalvelusta koostuva kokonaisuus voi toimia yhden nimiavaruuden sisällä. Nimiavaruuksien välistä verkkoliikennettä voi rajoittaa ja niille voi asettaa prosessoriaika- ja muistirajoitteita.

Ohjaustason tehtävä on saattaa klusteri haluttuun tilaan ja ylläpitää sitä. Ohjaustaso tarkkailee palvelinsolmujen kuormitusastetta ja jakaa kapselit klusterin palvelinsolmujen kesken. Ohjaustaso säilyttää klusterin tilaa etcd-tietovarastossa. Ohjaustaso voi hallita myös pilvipalveluntarjoajan alustaa rajapinnan kautta ja esimerkiksi lisätä palvelinsolmuja klusteriin. [3.]

Palvelinsolmut kommunikoivat ohjaustason kanssa Kubernetes-rajapintapalvelimen kautta. Kubelet on palvelinsolmulle asennettava komponentti, joka kommunikoi ohjaustason kanssa. Kubelet rekisteröi palvelinsolmun osaksi klusteria ja vastaanottaa ohjaustason vuorottajalta PodSpec-oloita. PodSpec-oliot määrittelevät palvelinsolmulle vuorotetut kapselit. Kubelet huolehtii, että palvelinsolmu suorittaa sille osoitettuja kapseleita. [3.]

3.2 Oliot

Kubernetes-klusterilla suoritetaan olioita. Olio määritellään joko JSON- tai yaml-dokumentilla, tai se on mahdollista luoda kubectl-komennolla. Käyttäjä kuvaa dokumenttiin olion, jonka Kubernetes pyrkii toteuttamaan. Koodiesimerkki 1 määrittelee ensin Deployment-olion, joka luo klusterille kaksi rinnakkaista konttia, joissa suoritetaan nginx-imagen uusin versio. Katkoviivan jälkeen määritellään Service-olio, joka tarjoaa nginx-kontit kuormanjakajan portista 80 ulkoverkkoon. Kubernetes pyrkii asentamaan tarvittavat Docker-kontit ja reitittämään verkkoliikenteen niin, että määrittelyä vastaava ympäristö toteutuu. [14.]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nginx
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: service-nginx
spec:
  externalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

Koodiesimerkki 1. Deployment- ja Service-oliot määrittelevä yaml-dokumentti.

Olio koostuu vähintään neljästä kentästä. ApiVersion-kenttä kertoo Kubernetekselle käytettävän rajapinnan version. Kind-kenttä kertoo luotavan olion tyypin. Metadata-kentässä oliolle annetaan yksilöivät tiedot ja spec-kentässä käyttäjä määrittelee olion toivotun tilan. Oliot ovat Kuberneteksen keskeisin käsite, sillä niiden avulla käyttäjä määrittelee muun muassa mitä sovelluksia klusterilla suoritetaan, kuinka paljon resursseja sovelluksille annetaan ja kuinka sovellus tarjotaan klusterista ulkoverkkoon. [15.]

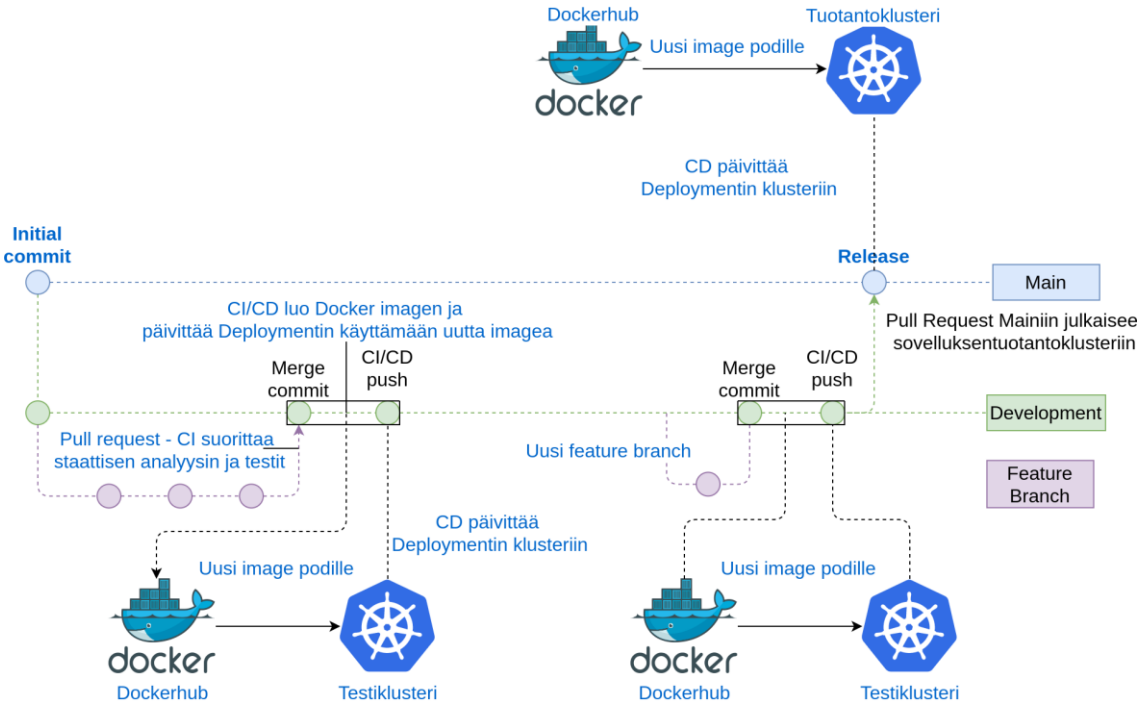
3.3 Nimiavaruudet

Nimiavaruudet ovat virtuaalisia klustereita klusterin sisällä. Nimiavaruuksien avulla Kubernetes-olioista voi muodostaa joukkoja. Tästä on hyötyä etenkin silloin, kun klusteria käyttää useampi itsenäinen sovelluskehitystiimi. Sovellusten jakaminen nimiavaruuksiin helpottaa palvelinsolmujen resurssien jakamista sovellusten kesken ja mahdollistaa hienojakoisemman käyttöoikeuksien säätelyn. Käyttäjälle on mahdollista sallia resurssien hallinnointi vain tietyissä nimiavaruuksissa. [3; 14.]

Myös verkkoliikennettä nimiavaruuksien välillä voi rajoittaa. Oletuksena Kubernetes sallii nimiavaruuksien välisen liikenteen. NetworkPolicy-oliolla voi joko estää liikenteen nimiavaruuksien välillä kokonaan tai sallia liikenteen vain tiettyihin nimiavaruuksiin. Tästä on hyötyä etenkin multi-tenant-klustereissa, joissa samalla klusterilla on useita käyttäjiä. Käyttäjien välisen liikenteen estäminen parantaa tietoturvaa, sillä mikäli liikenne on estetty, pahantahtoinen käyttäjä ei voi vaikuttaa muihin nimiavaruuksiin. [3; 14.]

4 GitOps -versionhallinta- ja julkaisujärjestelmä

Kubernetes-ajoympäristön käyttöönotto tuottaa suuren määrän yaml-dokumentteja, jotka määrittelevät järjestelmän halutun tilan. Tila voi kuvata muun muassa säiliöihin asennetun sovelluksen versiota, järjestelmän resurssirajoituksia, verkkomäärityksiä, klusterin tarjoamia palveluita, levyvarauksia ja kuormanjakoa. GitOps-menetelmässä ajoympäristön konfiguraatiota käsitellään ohjelmakoodina ja siihen kohdistetaan Git-versionhallintatyökalun toimintatapoja [3]. Kuvan 2 esimerkissä koodihaarojen yhdistäminen käynnistää jatkuvan integroinnin ja jakelun.



Kuva 2. Koodihaarojen yhdistämien Development- ja Main-haaroihin käynnistää integraatio- ja jakeluputken suorittamisen.

Main-haarassa säilytetään tuotantoympäristössä suoritettava koodi ja Development-haarassa testiympäristössä suoritettava koodi. Sovelluskehitys tehdään luomalla koodihaarojen yhdistämispyyntöjä erillisissä haaroissa toteutetuista ominaisuuksista Development-haaraan, jolloin jatkuva integrointi kääntää ja testaa sovelluksen ennen haarojen yhdistämistä. Yhdistämisen jälkeen jatkuva toimitus julkaisee uuden Docker-imagen ja päivittää konfiguraation viittaamaan tähän uuteen imageen. Integrointi ja jakeluprosessi on automatisoitu, ja koko järjestelmää ohjataan versiohallintaa käyttäen.

GitOps-menetelmässä ajoympäristö on kuvattu versionhallintaan deklaratiivisesti. Tämä kuvaus toimii ainoana lähteenä ajoympäristön konfiguraatiolle. Kun ajoympäristö saa konfiguraationsa vain yhdestä lähteestä, vältytään tilanteilta, joissa manuaaliset konfiguraatiomuutokset jäävät dokumentoimatta. Versiohallittu konfiguraation lähde lisää myös ajoympäristön vikasietoisuutta, sillä ympäristö voidaan häiriötilanteissa automaattisesti palauttaa mihin tahansa sen edellisistä tiloista. Versionhallintaan viedyn konfiguraation etuja ovat myös muun

muassa muutoshistorian näkyminen ja muutosten helppo peruuttaminen, konfiguraatioiden helppo haaroitus testi- ja tuotantoympäristöjä varten sekä muutosten katselmoinnit ja hyväksymisen koodihaarojen avulla. Vaikka tässä insinöörityössä GitOpsia käytetään Kubernetesin konfigurointiin, on GitOpsin käyttö mahdollista lähes minkä tahansa ympäristön kanssa, jonka tilaa voidaan hallita deklaratiivisesti. GitOpsilla on mahdollista myös käyttää muita versionhallintaohjelmistoja kuin Gitiä. GitOpsin tärkein teesi on se, että kaikki muutokset järjestelmään tai sovelluskoodiin tehdään versionhallinnan kautta automaattisesti. [3.]

4.1 GitOpsin edut

GitOps pyrkii tarjoamaan ratkaisuja konfiguraatiotiedostoina määritellyn infrastruktuurin automaattiseen käyttöönottoon ja tiedostojen säilyttämiseen. GitOpsissa sovelluskehittäjä ei hallitse järjestelmän tilaa asettamalla konfiguraatioita itse esimerkiksi komentoriviltä ja suorita skriptiä, vaan Kubernetesiin asennettu operaattori vertaa versionhallinnassa olevaa toivottua tilaa (engl. desired state) järjestelmän nykyiseen tilaan ja tekee tarvittavat muutokset. Versionhallintatyökalun käyttö tuo myös mahdollisuuden kumota tehdyt muutokset helposti, mikäli muutoksesta seuraa ei toivottuja seurauksia. GitOps tarjoaa merkittäviä etuja manuaaliseen konfiguraation hallintaan verrattuna muutosten hallintaan käytetään koodihaarojen yhdistämispyyntöjä. Konfiguraatiomuutosten näkyvyys paranee, sillä konfiguraation koko muutoshistoria on tallennettuna versionhallinnassa. Myös konfiguraatioiden keskitetty säilytys parantaa näkyvyyttä. Muutosten lisäksi versiohallintaan tallentuu muutosten tekijä. Konfiguraatiomuutokset voi myös testata automaattisesti ennen koodihaarojen yhdistämispyynnön hyväksyntää. [3.]

Kun ohjelmallinen operaattori hoitaa infrastruktuurin muutokset, voidaan tietoturvaa parantaa rajaamalla ihmisten joukkoa, jolla on oikeudet tehdä muutoksia klusteriin. Esimerkiksi ArgoCD:n avulla sovelluskehittäjä voi hallita sovelluksen konfiguraatiota ilman, että hänellä on pääsyä klusterille. Tietoturvaa parantaa myös versionhallinnan asetukset, joista voidaan muun muassa estää muutosten

tekeminen suoraan main- tai development-haaroihin ja asettaa yhdistämispyynnölle vähimmäismäärä hyväksyjiä.

Automaattinen CI/CD-putki yhdistettynä hallittuihin muutoksiin konfiguraatiossa ja sovelluskoodissa mahdollistavat erittäin nopean iteroinnin ja lyhentää ominaisuuden toteuttamisen läpimenoaikaa. Nopea iterointi ja lyhyt väli ohjelmoinnista tuotantoon mahdollistaa pienempien muutosten viennin tuotantoon, mikä helpottaa testausta ja pienentää epäonnistuneiden muutosten osuutta. [3.]

4.2 GitOpsin haasteet

Vaikka versionhallinnan käyttö DevOpsin toteuttamiseen on pääosin suoraviivaista, on versionhallinnan käyttämisessä konfiguraation säilyttämiseen ja iterointiin myös omat ongelmansa. Yksi niistä on ’salaisuuksien’ säilyttäminen. Monet sovellukset tarvitsevan ajon aikana tietokantatunnuksia, API-avaimia, TLS-sertifikaatteja ja muita tietoja, joita ei ole suositeltavaa säilöä selkokielisenä versionhallinnassa. Tällaiset tiedot on joko säilöttävä versionhallinnan ulkopuolella tai säilöttävä versionhallinnassa salattuina. Mikäli salaisuudet säilytetään versionhallinnan ulkopuolella, järjestelmän tila ei ole kokonaisuudessaan kuvattu versionhallinnassa. Mikäli tiedot salataan ennen versionhallintaan vientiä, täytyy purkamiseen sopiva avain silti säilöä jossain. [3.]

Toinen GitOpsin mukanaan tuoma haaste on repositorioiden määrän kasvu. Sovelluskehitystä varten on usein pystytetty monia rinnakkaisia ympäristöjä, usein vähintään kaksi, toinen laadunvarmistukselle ja toinen tuotannolle. Jos jokaiselle sovelluksen komponentille luodaan ympäristökohtainen repositorio, kasvaa hallittavien repositorioiden määrä suureksi, mikä vaikuttaa negatiivisesti luettavuuteen. Vaikka kaikki muutokset jäävät versionhallinnan historiaan, on monen repositorion muutosten läpikäyminen erittäin hidasta, etenkin jos sovellus koostuu useista mikropalveluista asennettuna moneen eri ympäristöön. Pienentämällä repositorioiden määrää läpikäytävien muutosten määrä ei välttämättä pienene, mutta tietoa joutuu etsimään pienemmästä määrästä lähteitä. Pienempi repositorioiden määrä altistaa kuitenkin ristiriidoille, sillä mikäli useampi

CI/CD-putki käyttää repositoriota samaan aikaan, voi repositorio vanhentua putken suorituksen aikana, mistä voi seurata manuaalista ratkaisemista vaativa ristiriita. [16; 17.]

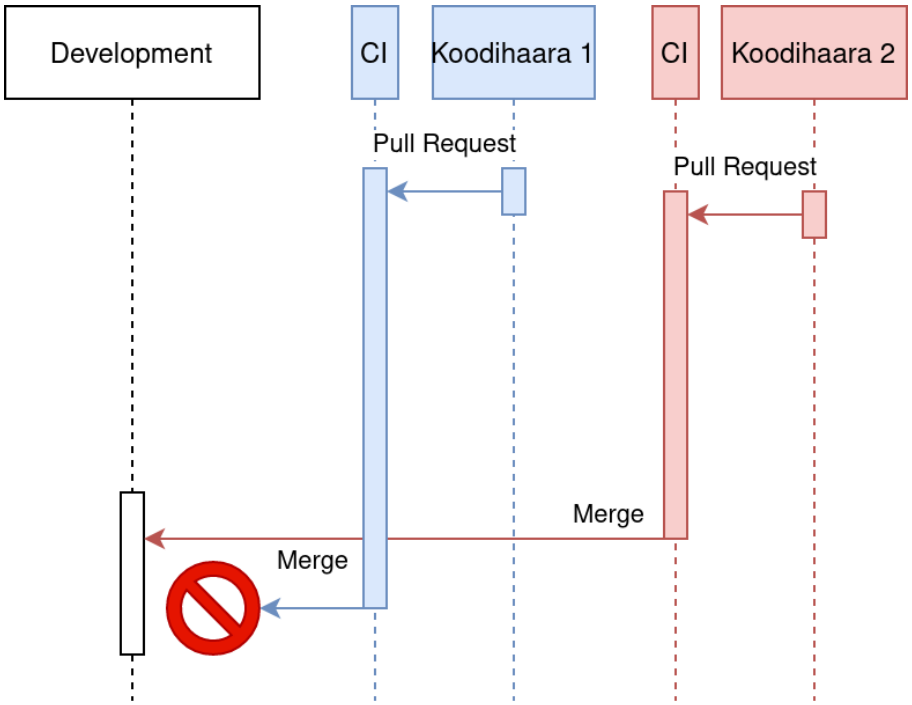
4.3 Ratkaisuja haasteisiin

Edellisessä luvussa kuvatut GitOpsin haasteet on yleisesti tiedostettu ja niihin on toteutettu erilaisia ratkaisuja. Salaisuuksien hallintaan käytettyjä ratkaisuja, joissa salaisuus salataan klusterilla ja tallennetaan salattuna versionhallintaan, ovat mm. SealedSecret, Kamus ja Helm Secrets. Esimerkiksi Bitnamin SealedSecret asentaa klusterille kontrollerin, jolla se luo klusterille avainparin. Purkamiseen sopiva yksityinen avain on vain kontrollerin tiedossa. SealedSecret jakaa rajapinnan kautta julkista avainta, jolla salaisuudet salataan kubeseal-työkalun avulla. Koska yksityiseen avaimeen on pääsy vain kontrollerilla, versionhallintaan viedyt salatut salaisuudet ovat klusterikohtaisia eikä niitä voida käyttää uudelleen muissa klustereissa. Tästä syystä on tärkeää säilyttää salaisuuksien tietosisältö myös jossain muualla, sillä katastrofin sattuessa salaisuuksien sisältöä voi olla mahdoton palauttaa. Näkyvyyttä (engl. scope) voi säätää salausvaiheessa, ja oletuksena salaisuuksia ei voi salaamisen jälkeen enää uudelleennimetä tai siirtää nimiavaruudesta toiseen. Salaisuuden uudelleennimeämisen voi halutessaan sallia nimiavaruuden sisällä tai koko klusterin sisällä, jolloin salaisuuden voi nimetä uudelleen ja siirtää nimiavaruudesta toiseen. Klusterin järjestelmänvalvoja voi myös hakea avainparin klusterilta, mikäli avaimista halutaan varmuuskopiot tai ne halutaan siirtää toiseen klusteriin. [3.]

Toinen suosittu ratkaisu on käyttää salaisuuksienhallintaohjelmistoa. Tällaisen tarjoavat Kubernetes-alustaa myyvät palveluntarjoajat kuten AWS Secrets Manager, GKE Secrets Manager ja Azure Key Vault. Vaihtoehtoisesti sovelluskehittäjä voi asentaa valitsemansa toteutuksen, esimerkiksi HashiCorp Vaultin. Näissä ratkaisuissa sovelluskehittäjä luopuu ajatuksesta säilöä salaisuudet Gitiin. Sovelluskehittäjä säilöö salaisuudet erilliseen säilöön, josta sovellukset pyytävät salaisuuksia tarvittaessa. Salaisuudet voidaan myös injektoida kapseleille, jolloin sovelluskoodiin ei tarvitse tuoda koodia salaisuuden hallintatoteutukseen

tai Kubernetekseen sidottua koodia. HashiCorp Vaultia käytettäessä salaisuudet eivät ole Gitissä, vaan ne on käsin asetettava Vaultiin. Vault on kuitenkin luonteeltaan pysyvä säilytyspaikka luottamukselliselle tiedolle. Kun yaml-pohjat on luotu ja kontrolleri asennettu Kubernetekseen, ei salaisuuksien tallentamisesta tarvitse huolehtia. Vault osaa myös luoda kapselikohtaisia väliaikaisia käyttäjätunnuksia esimerkiksi tietokantayhteyksiä varten ja luoda ja kierrättää salasanoja. [3.]

Tietovarastojen hallintaan ei ole olemassa helppoa ratkaisua. Näkyvyyttä voi parantaa antamalla jokaiselle koonnille uniikin tunnisteen ja käyttämällä sitä sovelluksen asentamisessa. Latest-tunnistetta ja snapshot-versioita kannattaa välttää, sillä ne peittävät suorituksessa olevan sovelluksen version. Kuvassa 3 sininen putki alkaa ennen punaisen putken suoritusta, mutta päättyy punaisen putken suorituksen jälkeen.



Kuva 3. Samoihin tiedostoihin kohdistuneet muutokset voivat aiheuttaa ristiriitoja koodihaarojen välillä.

Jos molemmat koodihaarat tuovat muutoksia samoihin tiedostoihin, syntyy yhdistämisvaiheessa ristiriita. Tämä ristiriita huomataan vasta putken suorittamisen jälkeen, jolloin putken suoritus on turha ja aikaa menee hukkaan. Koodihaarojen välisten ristiriitojen välttämiseksi sovelluskehittäjien on tehtävä yhteistyötä ja jaettava työ niin, että samoihin tiedostoihin tulee mahdollisimman vähän muutoksia eri koodihaarojen välillä. Myös CI/CD-putken suoritusajan lyhentäminen voi auttaa.

5 Ympäristön suunnittelu

Ympäristön suunnittelu aloitettiin käyttäjän tarpeiden tunnistamisesta. Koska ympäristön toteutti sen pääasiallinen käyttäjä, työn laajuus oli rajattava tarkasti. Huonosti suunniteltuna työn laajuus kasvaa sen edetessä niin suureksi, että sen toteuttaminen ja ylläpitäminen on mahdotonta kohtuullisessa ajassa.

Työ alkoi käyttäjätarinoiden kirjaamisesta. Käyttäjätarinoiden kirjoittaminen auttaa hahmottamaan ympäristöön kohdistuvia vaatimuksia ja valitsemaan käytettävät teknologiat ja tuotteet. Varsinaista vaatimusmäärittelydokumenttia tässä insinöörityössä ei tehty.

- Sovelluskehittäjänä haluan, että koodimuutoksen vienti versionhallintaan käynnistää automaattisen asennuksen, jotta näen versionhallinnasta, mikä versio sovelluksesta on asennettu.
- Sovelluskehittäjänä haluan, että sovelluksen konfiguraatioita voi hallita versionhallinnasta, jottei sovelluksen konfigurointia varten tarvitse yhdistää palvelimelle.
- Sovelluskehittäjänä haluan asentamisen ja muiden operaatioiden olevan mahdollisimman automatisoituja, jotta voin keskittyä koodaamiseen.
- Sovelluskehittäjänä haluan automaattisen haavoittuvuus- ja laatuskannauksen lähdekoodille ja riippuvuuksille, jotta sovellusten ylläpito helpottuu.
- Ylläpitäjänä haluan, että kaatuneet sovellukset käynnistyvät uudelleen automaattisesti, jotta käyttökatkot pysyvät lyhyinä.
- Ylläpitäjänä haluan, että sertifikaatit päivittyvät automaattisesti, jottei sovellukselle tule käyttökatkoja.

- Ylläpitäjänä haluan lisätä klusterin laskentakykyä tarvittaessa, jotteivat palveluiden vasteajat kasva liian suureksi.
- Ylläpitäjänä haluan saada nopean yleiskuvan suoritettavista sovelluksista ja niiden tilasta, sillä sovellusten tilan selvittäminen yksi kerrallaan vie aikaa.

Käyttäjätarinoista johdettiin ympäristöön kohdistuvia vaatimuksia ja ne koottiin taulukkoon 1. Vaatimusten avulla työmäärää ja valmiusastetta on helpompi arvioida. Vaatimusten avulla suunniteltiin hyväksyntätestit, joiden avulla varmistettiin, että ympäristö on tarkoituksenmukainen.

Taulukko 1. Ympäristön vaatimukset

1	Tuotantoympäristön tietokantayhteys ei ole käytössä testiympäristössä.	Pakollinen
2	Koodihaarojen yhdistämispyyntö development- tai main-haaraan käynnistää sovelluksen testit.	Pakollinen
3	Mikäli testit epäonnistuvat, koodia ei voi integroida.	Toivottava
4	Koodihaarojen yhdistäminen development-haaraan käynnistää JVM-koonnin sovelluksesta.	Toivottava
5	Koodihaarojen yhdistäminen main-haaraan käynnistää natiivikoonnin sovelluksesta.	Toivottava
6	Buildin luoma Docker-image viedään julkiseen Docker-repositorioon.	Pakollinen
7	Koodille suoritetaan staattinen analyysi.	Toivottava
8	Versionhallinnan development-haarassa oleva koodi on asennettuna testiympäristöön.	Pakollinen
9	Versionhallinnan main-haarassa oleva koodi on asennettuna tuotantoympäristöön.	Pakollinen
10	Tuotantoklusteriin voi lisätä palvelinsolmuja.	Pakollinen
11	Sovellus käynnistyy automaattisesti uudelleen kaatuessaan.	Pakollinen
12	Sovellus käyttää suojattua HTTPS-yhteyttä.	Pakollinen
13	Sertifikaatit uusitaan automaattisesti.	Toivottava

Vaatimukset jaoteltiin pakollisiin ja toivottaviin. Työ hyväksytään, kun kaikki pakolliset vaatimukset on täytetty. Hyväksyntätestaus on mahdollista monilta osin automatisoida, mutta tässä työssä hyväksyntätestaus tehtiin manuaalisesti. Hyväksyntätestauksen tulokset on esitelty liitteessä 2.

5.1 Git-repositorion rakenne

Asennettavan sovellukset lähdekoodi oli jo valmiiksi GitLabin ylläpitämässä versiohallinnassa. Repositorion hakemistorakenne järjesteltiin niin, että sovelluksen lähdekoodi on kuvan 4 mukaisesti juurihakemistossa sijaitsevassa ggl-api/ -hakemistossa ja yaml-konfiguraatiot hakemistossa yaml/.

```
[juiffi@fedora Programming]$ tree ggl_java/ -d
ggl_java/
├── ggl-api
│   └── src
│       ├── main
│       │   ├── docker
│       │   ├── java
│       │   │   └── org
│       │   │       └── juiffi
│       │   │           └── ggl_api
│       │   │               ├── mapper
│       │   │               ├── repository
│       │   │               └── resources
│       │   └── resources
│       │       └── META-INF
│       │           └── resources
│       └── test
│           └── java
│               └── org
│                   └── juiffi
│                       └── ggl_api
│                           └── test
│                               └── resources
└── yaml
    ├── development
    │   ├── deployment
    │   ├── secrets
    │   └── services
    └── main
        ├── deployment
        ├── ingress
        ├── secrets
        └── service
```

Kuva 4. Repositorion hakemistorakenne.

Yaml-hakemiston sisällä tiedostot on jaettu ensin kohdeympäristön ja sitten tyypin mukaan. Kun kaikkien ympäristöjen yaml-dokumentit lisättiin samaan repositorioon, erillistä repositoriota konfiguraatiotiedostoille ei tarvinnut luoda. Sovelluksen koodit ja konfiguraatiot voitiin säilöä sovelluskohtaiseen repositorioon.

Tällä vastattiin myös vaatimuksiin 8 ja 9 konfiguroimalla ArgoCD lukemaan testi- ja tuotantoympäristöissä määritykset koodihaaraa vastaavasta hakemistosta.

5.2 CI/CD-työkalujen valinta

Jatkuvan integroinnin työkaluksi valittiin GitLab CI/CD. Se on avoimen lähdekoodin työkalu, joka integroituu erittäin hyvin osaksi GitLabia. GitLab tarjoaa 400 minuuttia ilmaista suoritusaikaa kuukaudessa jaetuilla palvelimilla. Suoritusajan rajoitus ei koske käyttäjän itse asentamia suorituspalvelimia, joita voi käyttää jaettujen palvelimien lisäksi tai sijasta. Jaetut suorituspalvelimet pienentävät käyttökustannuksia verrattuna esimerkiksi Jenkinsiin, josta ei ole saatavana ilmaista palvelua, vaan se on suoritettava omalla laitteistolla. GitLab CI/CD:n dokumentaatio on myös hyvällä tasolla, ja suorituspalvelimen asentaminen Kubernetes-klusteriin oli Helm-pakettienhallinnalla helppoa.

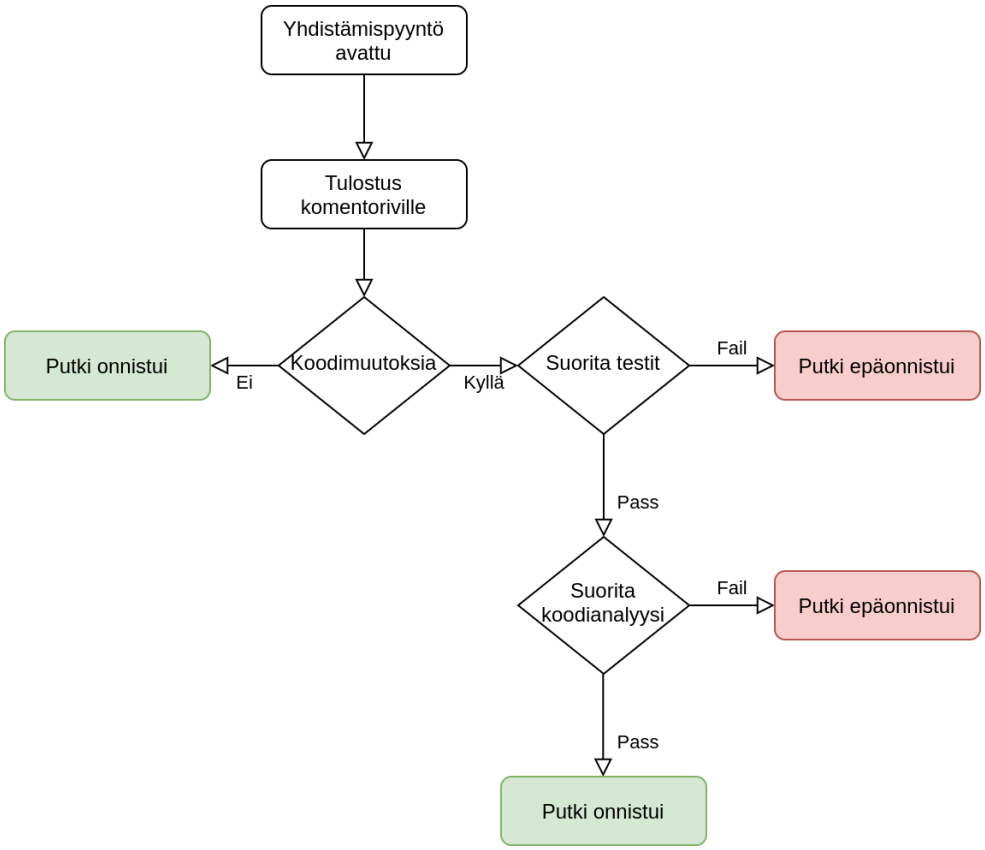
Jenkins X ei sopinut jatkuvan integroinnin ratkaisuksi tässä projektissa, sillä se ei tue Bitnamin SealedSecreteja. TravisCI karsiutui, sillä siitä on tarjolla vain SaaS-ratkaisu ja GitLab-integraatio ei ole yhtä hyvä kuin GitLab CI/CD:ssä. Jenkins ja Tekton olivat molemmat erittäin varteenotettavia vaihtoehtoja, mutta käyttökustannuksissa ja käyttöönoton helppoudessa ne eivät pärjänneet GitLab CI/CD:lle. Mitä vähemmän klusterille tarvitsee asentaa työkaluja, sitä suurempi osa resursseista jää sovellusten suorittamiseen.

Jatkuvan toimituksen työkaluksi valittiin ArgoCD, vaikka myös GitLab CI/CD olisi voinut sen tehdä. ArgoCD tukee SealedSecreteja ja tarjoaa tehokkaat komentorivi- ja web-käyttöliittymät. Ratkaisulla pienennettiin toimittajaloukun riskiä eli tilannetta, jossa järjestelmän osan vaihtaminen toisen toimittajan tuotteeseen on työlästä tai mahdotonta.

5.3 CI/CD-putken suunnittelu

Jatkuvan integroinnin ja toimituksen putki toteutettiin niin, että integraatioputki suoritetaan vain yhdistämispyynnön avaamisen tai päivittämisen yhteydessä. Lisäksi yhdistämispyynnön täytyy kohdistua development- tai main-haaraan. Tällä menettelyllä pyrittiin välttämään ylimääräisiä integraatioputken suorituksia. Tärkeintä on suorittaa testit ja analyysi sellaiselle versiolle, joka on asentumassa testi- tai tuotantoympäristöön. Sovelluskehittäjä voi halutessaan suorittaa staattisen analyysin ja yksikkö- ja integraatiotestit omalla työasemallaan.

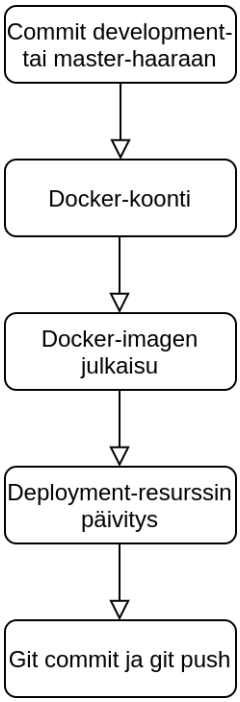
Testien ja analyysin läpäiseminen on vaatimus yhdistämispyynnön hyväksymiselle. Koodihaarojen yhdistämisen voi estää GitLabissa kahdella tavalla. Ensimmäinen vaihtoehto on asettaa repositorion asetuksista päälle asetus, joka vaatii jokaiselta yhdistämispyynnöltä putken onnistuneen suorituksen. Tämän haittapuolena on se, että putki täytyy suorittaa, vaikka sovelluskehittäjän muutokset eivät vaatisi uutta koontia. Tällainen tilanne voi syntyä esimerkiksi dokumentaation päivittämisestä. Toinen vaihtoehto on lisätä repositoriolle ylläpitäjän rooliin uusi tunnus. Tunnuksen ssh-avain lisätään CI/CD-putkeen muuttujaksi, jonka avulla yhdistämispyyntö hyväksytään GitLabin rajapinnan kautta. Haittapuolena on se, että GitLab sallii pyyntöjen yhdistämisen ilman hyväksyntää. Hyväksynnän asettaminen pakolliseksi on maksullinen ominaisuus. Työssä päädyttiin käyttämään ensimmäistä vaihtoehtoa, sillä testaamattoman koodin vienti development- ja main-haaroihin haluttiin estää vaatimuksen 3 mukaisesti. Kuvassa 5 integraatioputken vaiheet on esitelty vuokaaviona. [18.]



Kuva 5. Integraatioputken vuokaavio.

Putki on suunniteltu vastaamaan vaatimuksiin 2, 3, 4 ja 7. Suoritus käynnistyy vain development- ja main-haaroihin kohdistuvasta yhdistämispyynnöstä. Putkeen luotiin vaihe, joka suoritetaan aina. Vaihe ainoastaan tulostaa komentoriville merkkijonon eikä siksi voi epäonnistua. Koodimuutosten tarkistamisen jälkeen suoritetaan testit ja koodianalyysi. Ne suoritetaan vain, mikäli sovelluskehittäjä on tehnyt muutoksia, jotka vaativat uuden koonnin. Mikäli testit epäonnistuvat tai koodin laatu ei ole riittävällä tasolla, putki epäonnistuu. Epäonnistunut putki estää koodihaarojen yhdistäimisen.

Onnistuneen suorituksen jälkeen koodihaarat voi yhdistää. Mikäli kohdehaara on development tai main, GitLab CI/CD aloittaa uuden putken suorittamisen. Putken vaiheet on esitelty kuvassa 6.



Kuva 6. Koodihaarojen yhdistämisen jälkeen suoritettavat vaiheet.

Putki paketoi sovelluksen Docker-imageksi ja vie sen Docker Hub -repositorioon. Docker Hub on vaatimuksen 6 mukainen julkinen repositorio. Putki merkitsee luodun Docker-imagen Git-versiohallinnan luomalla tiivisteellä. Koska tiiviste on uniikki, Docker-imagesta tulee käytännössä muuttumaton. Tiivisteellä merkitty image sisältää aina tiivistettä vastaavat lähdekoodit. On kuitenkin huomioitava, että mikäli käyttäjällä on kirjoitusoikeus Docker-imaget sisältävään repositorioon, hän voi ylikirjoittaa merkityn imagen. Tilanteen syntymistä pyrittiin estämään rajoittamalla kirjoitusoikeudet vain itselle ja putken Docker-tunnuksille.

Imagen luonnin jälkeen putki päivittää repositorion yaml/-hakemistossa sijaitsevan Deployment-olion määrittelyn. Resurssissa on määritelty klusterille asennettavan imagen versio. Putki korvaa versionumerona käytettävän tiivisteen ja luo muutoksesta uuden pysyvän muutoksen. Putki varastoi muutokset versiohallintaan viestillä “CI-CD-BOT Image tag updated”, ja putken suoritus päättyy. Putki käynnistyy development- ja main-haaroihin tehdystä muutoksesta vain, jos

pysyvän muutoksen viesti ei ala merkkijonolla “CI-CD-BOT" ja muutoksia on tehty lähde koodit sisältävään hakemistoon.

Klusterille asennettu ArgoCD tarkkailee muutoksia git-repositoriossa. Mikäli yaml/-hakemistossa sijaitsevat määritystiedostot muuttuvat, ArgoCD päivittää muutokset klusterille. Kun putki on päivittänyt käytettävän imagen version, ArgoCD aloittaa muutoksen toimeenpanon klusterilla. Koodihaaran yhdistämispyynnön hyväksyminen päivittää putken ja ArgoCD:n avulla klusterilla suoritettavan sovelluksen ilman käyttäjän toimia.

5.4 Kubernetes-jakelun valinta

Työssä päätettiin käyttää Kubernetesta sovelluksen jakeluun, sillä Kubernetes-klusterin suorituskykyä voi parantaa lisäämällä siihen palvelinsolmuja. Kubernetes myös pyrkii säilyttämään klusterin tilan konfiguraatiota vastaavana muun muassa käynnistämällä kaatuneet sovellukset uudelleen.

Kuberneteksen asentamiseen käytetään kubeadm-työkalua. Kuberneteksen dokumentaatiossa suositellaan, että palvelinsolmuilla olisi vähintään kaksi gigatavua muistia ja kaksi prosessoriydintä [19]. Canonicalin jakelu Microk8s suosittelee neljää gigatavua keskusmuistia [20]. Rancherin k3s-jakelun minimivaatimuksiksi on ilmoitettu 512 megatavua keskusmuistia ja yksi prosessoriydin [21]. Koska Kuberneteksen komponenttien suorittamiseen kuluvat resurssit ovat pois sovellusten suorittamiselta, päätettiin jakeluiden välisiä suorituskykyeroja mitata erikokoisilla virtuaalipalvelimilla. Tavoitteena oli löytää jakelu, joka kuluttaisi vähiten resursseja ja olisi siten käyttökustannuksiltaan edullisin. Samalla verrattiin eroja Quarkus-sovelluksen JVM-koonnin ja natiivikoonnin välillä.

Koska kubeadm-työkalu vaatii valmiiksi paketoitua jakelua enemmän konfigurointia, se jätettiin pois vaihtoehdoista ja suorituskykymittauksesta. Suorituskykymittaus suoritettiin Microk8s ja k3s Kubernetes-jakeluille ja valinta tehtiin niiden välillä.

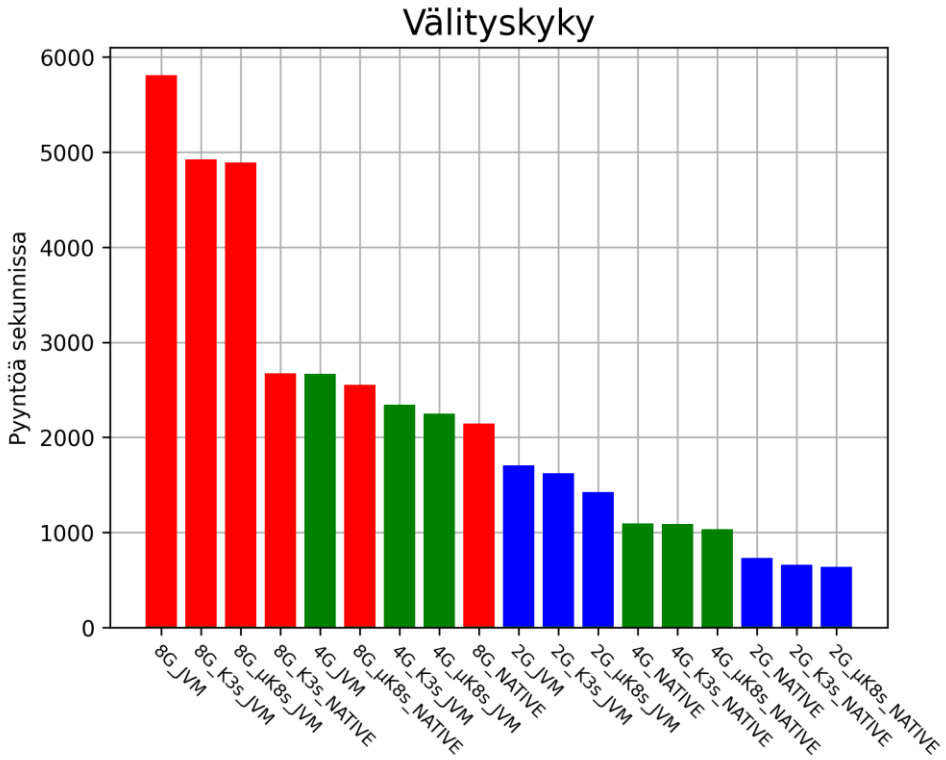
5.5 Suorituskykymittaus

Suorituskykymittausta varten asennettiin kolmelle virtuaalipalvelimelle kullekin kuusi erilaista kokoonpanoa. Palvelimet olivat yhden, kahden ja neljän prosessoriytimen virtuaalipalvelimia ja niillä oli vastaavasti kaksi, neljä ja kahdeksan gigatavua keskusmuistia. Palvelimille asennettiin vuorotellen seuraavat kokoonpanot:

1. Sovelluksen JVM-koonti ilman Kubernetesta
2. Sovelluksen Natiivikäännös ilman Kubernetesta
3. K3s + Sovelluksen JVM-koonti
4. K3s + Sovelluksen Natiivikäännös
5. Microk8s + Sovelluksen JVM-koonti
6. Microk8s + Sovelluksen Natiivikäännös

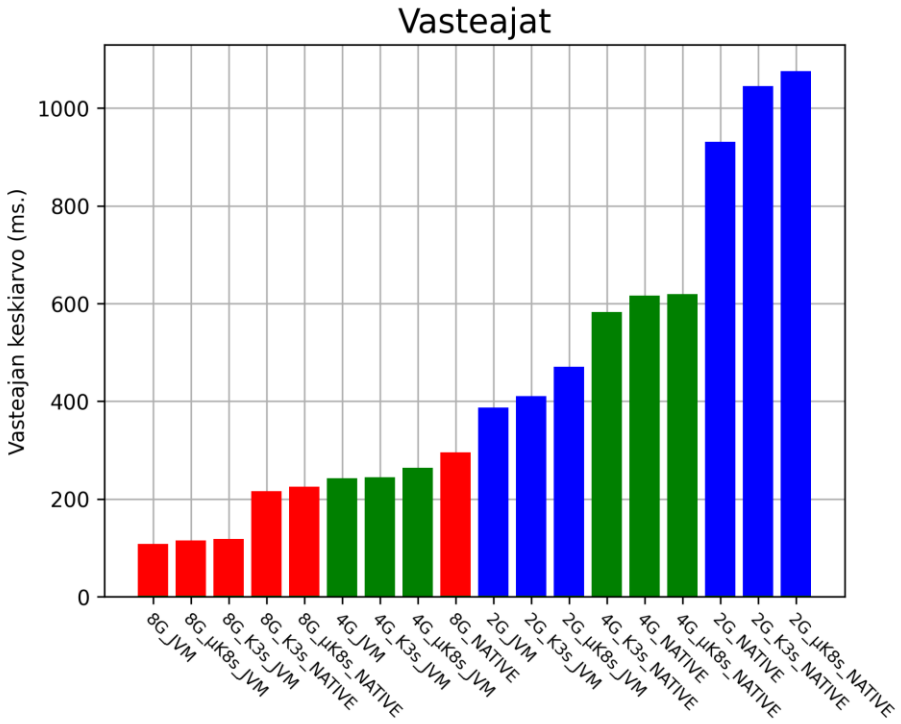
Palvelimia kuormitettiin erilliseltä virtuaalipalvelimelta JMeter-ohjelmalla. JMeter asetettiin kutsumaan sovellusta 700 rinnakkaisella säikeellä. Jokainen säie kutsui sovellusta kolmetuhatta kertaa. Säikeiden määrä valittiin niin, että jokainen testattava kokoonpano oli täydessä prosessorikuormassa testin aikana. Sovellus haki erilliseltä tietokantapalvelimelta kutsua vastaavat rivit ja palautti ne JMeterille.

Ero jakeluiden välillä ei ollut suuri välityskyvyssä ja vasteajassa mitattuna. Kuva 7 havainnollistaa välityskyvyn kasvua resurssien kasvaessa. Suorituskykymittauksen tulokset on taulukoitu liitteessä 3.



Kuva 7. Pylväsdiagrammi kokoonpanojen keskimääräisestä välityskyvystä.

Tulosten perusteella K3s oli keskimäärin noin 4 prosenttia suorituskykyisempi kuin Microk8s. Ero jakeluiden välillä kapeni, kun virtuaalipalvelimen resursseja lisättiin. Ilman Kubernetesta palvelimen välityskyky oli keskimäärin 15 % suurempi. Poikkeuksena tehokkaimmalla virtuaalipalvelimella suoritettu mittaus, jossa suoritettiin sovelluksen natiivikäännöstä ilman Kubernetesta. Ilman Kubernetesta palvelimelta mitattiin lähes 18 % pienempi välityskyky. Kuvassa 8 on esitetty pylväsdiagrammina eri kokoonpanojen vasteajat.



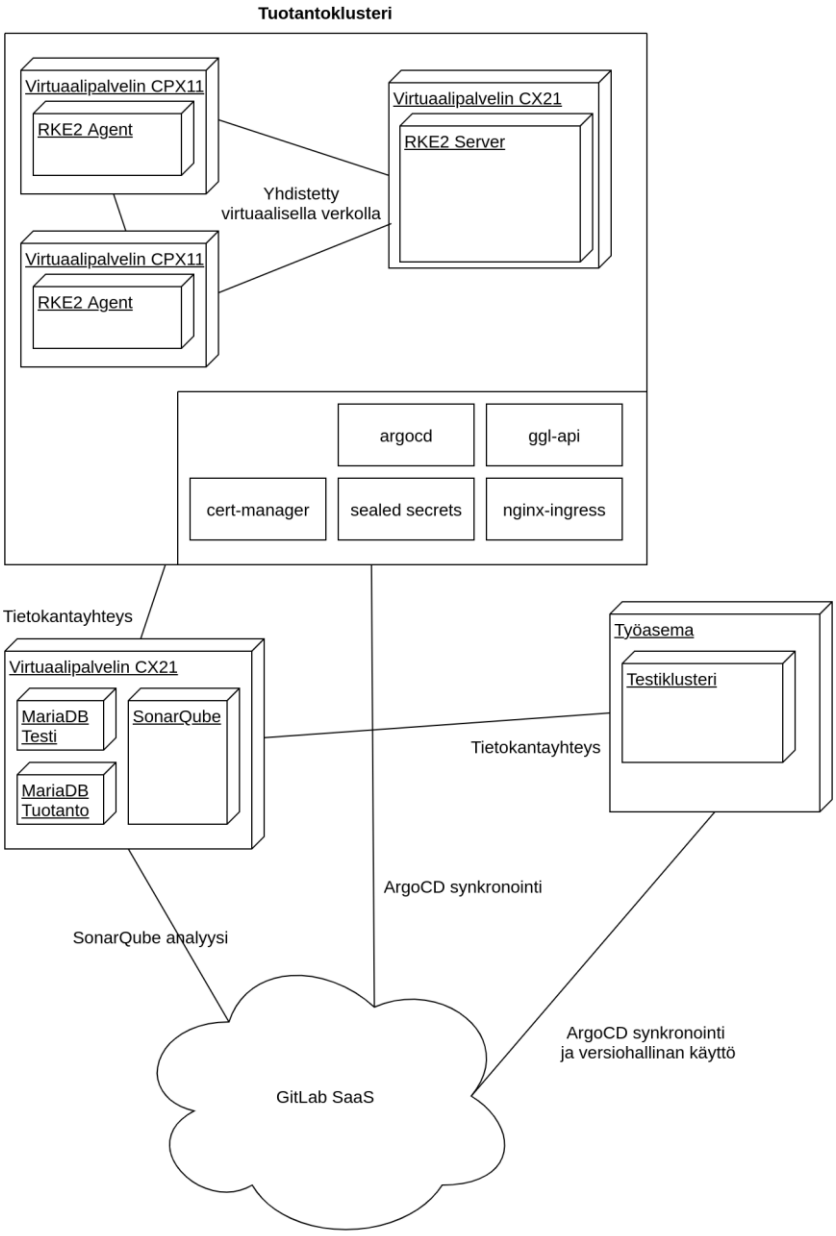
Kuva 8. Pylväsdiagrammi kokoonpanojen keskimääräisistä vasteajoista.

Vasteaikamittauksen tulokset ovat hyvin samansuuntaiset, kuin välityskykymittauksen tulokset. Resurssien kaksinkertaistaminen karkeasti puolitti vasteajan. Jakeluiden välillä ei ollut suurta Microk8s ja K3s vaihtoivat paikkaa tehokkaimmalla virtuaalipalvelimella verrattuna välityskykymittaukseen. Tehokkaimmalla virtuaalipalvelimella vasteaika JVM-koonnilla parani vain n. 8 % jättämällä Kubernetes pois, vaikka ero välityskyvyssä oli n. 18 %.

Mittausten isoin löydös oli Quarkus-sovelluksen suorituskyky GraalVM-natiivikäännöksenä. JVM-käännös oli huomattavasti nopeampi kuin natiivikäännös. Ero oli neliydinpalvelimella yli kaksinkertainen. Tämän perusteella vaatimus natiivikäännöksen asentamisesta tuotantoympäristöön jätettiin pois. Vaikka natiivikäännös tarvitsee ajon aikana paljon vähemmän keskusmuistia, on pidentynyt vasteaika käyttökokemuksen kannalta erittäin huono asia. Pienimmällä virtuaalipalvelimella vasteajat kasvoivat kuormassa yli sekunnin mittaisiksi, mikäli sovelluksesta oli käytössä natiivikäännös. JVM-käännös samalla palvelimella vastasi kutsuihin noin 400 millisekunnissa.

5.6 Klusterin rakenteen suunnittelu

Tietokannat olivat suunnittelun alkaessa entuudestaan omalla palvelimellaan. SonarQuben asennusohjeet suosittelevat Kubernetesiin asennettaessa palvelinsolmun varaamista pelkästään SonarQubelle [22]. Koska SonarQubelle ei haluttu varata kokonaista palvelinsolmua, SonarQube asennettiin tietokantojen kanssa samalle palvelimelle. Tietokannat ja SonarQube jäivät klusterin ulkopuolelle kuvan 9 sijoittelukaavion mukaisesti.



Kuva 9. Asennettavan ympäristön sijoittelukaavio

Sijoittelukaaviossa näkyy suoritettavien sovellusten jakautuminen eri virtuaalipalvelimille. Klusterin ulkopuolinen palvelin suorittaa SonarQubea ja MariaDB-tietokantoja Docker-konteissa ilman Kubernetesta. GitLab on yhteydessä tähän palvelimeen vain silloin, kun jatkuvan integroinnin putki suorittaa SonarQube-analyysin. Palvelimen konfiguraatiota ei hallita versionhallinnasta. Sen sijaan kehittäjän työasemalle asennettua testiklusteria ja useasta virtuaalipalvelimesta koostuvaa tuotantoklusteria ohjataan GitLabin versionhallinnasta käsin.

Tuotantoklusterin virtuaalipalvelimet on yhdistetty samaan verkkoon, jolloin yksittäisen palvelinsolmun julkisesta IP-osoitteesta ei tarvitse avata portteja ulkoverkkoon. Klusterille asennettu kuormanjako reitittää liikenteen klusterin sisällä ja Kubernetes vuorottaa suoritettavat sovelluskontit palvelinsolmuille. Sovelluskehittäjän ei tarvitse tietää, millä palvelinsolmuilla sovellusta suoritetaan.

Klusterille asennettavat työkalut ja sovellukset asennettiin omiin nimiavaruuksiinsa. Käyttäjien ja palvelutunnusten (engl. service account) käyttöoikeuksia voi hallita nimiavaruuskohtaisesti ja nimiavaruuksien välistä tietoliikennettä on helppo rajoittaa. Esimerkiksi GitLab Runner tarvitsee oikeuden luoda ja poistaa olioita CI/CD-putken suoritusta varten [23]. Kun GitLab Runnerin asentaa omaan nimiavaruuteensa, voi oikeuden rajata koskemaan vain GitLab Runnerin nimiavaruutta.

6 Ympäristön asennus

Suorituskykymittausten ja asennuksen työmääräarvion perusteella Kubernetes-jakeluksi valittiin Rancherin k3s:n paikallista testiympäristöä varten ja RKE2:n tuotantoympäristöä varten. RKE2 käyttää sisäisesti k3s:ää, mutta on suunnattu tuotantokäyttöön [24].

6.1 Klusterin paikallinen asennus

Rancher tarjoaa k3s:n asentamiseen valmiin skriptin. Sen avulla klusterin asennus on helppo tehdä. Klusterin asennuksen aikana tarvitaan pääkäyttäjän oikeuksia, joten klusteri asennettiin virtuaalikoneeseen. Klusterin asennuksen jälkeen asennettiin Bitnamin SealedSecrets-controller suorittamalla komento:

```
kubectl apply -f https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.16.0/controller.yaml
```

ArgoCD-controllerin asennus tehtiin suorittamalla komennot:

```
kubectl create namespace argocd && kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Komentojen suorittamisen jälkeen ArgoCD:n service-olion määrittelyä muokattiin vaihtamalla ServiceTypeksi NodePort. NodePort paljastaa sovelluksen klusterin ulkopuolelle sattumanvaraisesta portista välillä 30000–32768 [25]. ArgoCD sai virtuaalikoneen portin 31361, joka ohjattiin samaan porttiin työasemalla. ArgoCD:lle kirjauduttiin selaimen kautta ja varmistettiin, että asennus oli onnistunut.

Klusterin, SealedSecretsin ja ArgoCD:n asentamisen jälkeen sovelluksen git-repositorio lisättiin ArgoCD:lle graafisen käyttöliittymän kautta. Lisäämisen jälkeen ArgoCD alkoi tarkkailla määrittelymuutoksia repositorion yaml/deployment-hakemistossa. Seuraavaksi määriteltiin Secret-tyyppinen Kubernetes-olio. Olio säilyttää testiympäristön tietokannan osoitteen, käyttäjätunnuksen ja salasanan. Tiedot salattiin kubeseal-työkalulla ja vietiin versiohallintaan.

ArgoCD haki määrittelyt repositoriosta ja purki salauksen onnistuneesti. Seuraavaksi lisättiin Deployment- ja Service-oliot ja vietiin ne versiohallintaan. K3s sisältää oletuksena Klipper-kuormanjakajan. Kun Service-olion tyypiksi asetetaan LoadBalancer, Klipper tarjoaa sovelluksen käyttäjän haluamasta portista. Ggl-api Docker-image paljastaa kontista portin 8080, joka asetettiin myös Service-olion portiksi. Kun uudet määritykset oli viety versiohallintaan, ArgoCD haki

määritykset versiohallinnasta ja loi oliot. Sovellus käynnistyi ja oli saatavilla virtuaalikoneen portista 8080. Työaseman portti 8888 ohjattiin virtuaalikoneen porttiin 8080 ja sovelluksen toimivuus varmistettiin.

6.2 CI/CD-putken toteutus

Luvussa 5.4 suunnitellun putken toteuttaminen aloitettiin toteuttamalla vaihe, jonka GitLab CI/CD suorittaa silloin, kun koodimuutoksia ei ole tehty. Koodiesimerkissä 2 määritellään “empty-run”-niminen työn CI/CD-putki “cicd-pipeline-requirement"-vaiheeseen.

```
empty-run:
  stage: cicd-pipeline-requirement
  image: busybox:latest
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  script:
    - echo "Empty pipeline run."
    - echo true
```

Koodiesimerkki 2. CI/CD-putken työvaihe, joka suorittaa tulostuksen komentoriville.

Työ suoritetaan busybox Docker -imagessa ja vain jos putken käynnistäjä on koodinhaarojen yhdistämispyyntöön liittyvä tapahtuma. Esimerkiksi tavallinen git push ei käynnistä putken suorittamista. Script-osuudessa määritellään suoritettavaksi kaksi echo-komentoa. Komentojen suorittamisen jälkeen putken suoritus on valmis.

Seuraava toteutettava vaihe oli laadunvarmistus. Sekä testit että koodianalyysi suoritetaan CI/CD-putken samassa vaiheessa koodiesimerkin 3 mukaisesti.

```
sonarqube-job:
  rules:
   - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
     changes:
      - ggl-api/**/*
  stage: qa
  image: maven:3.6.3-jdk-11
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: one-single-cache-key-for-everything
    paths:
    - .m2/repository
    - .sonar/cache
  script:
    - cd ggl-api
    - mvn org.owasp:dependency-check-maven:check
    - mvn verify sonar:sonar
```

Koodiesimerkki 3. CI/CD-putken työvaihe, joka suorittaa yksikkötestit, riippuvuuksien haavoittuvuusanalyysin ja staattisen SonarQube-analyysin.

Työn suorittamiseen käytettävä Docker-image vaihdettiin busyboxista maveniksi. Maven-image sisältää tarvittavat sovellukset ja riippuvuudet työn suorittamiseksi. Työn suoritus alkaa vain yhdistämispyynnöissä ja vain, mikäli sovelluskehittäjä on tehnyt muutoksia ggl-api-hakemiston sisällä oleviin tiedostoihin. Ehtojen jälkeen määritellään image, muuttujat Sonarqube-analyysia varten ja otetaan käyttöön välimuisti. Työn aikana ladatut tiedostot, kuten Maven-riippuvuudet ja haavoittuvuusanalyysin tarvitsemat tiedot, tallennetaan välimuistiin. Välimuistin käyttö nopeuttaa putken suoritusta. Työn skripti koostuu kolmesta komennosta. Ensimmäinen komento siirtyy repositorion hakemistoon ggl-api, ja seuraavat suorittavat riippuvuuksien haavoittuvuusanalyysin, testit ja staattisen analyysin.

Kun koodihaarojen yhdistämispyyntöjen kontekstissa suoritettava osuus oli toteutettu, siirryttiin toteuttamaan imagen luontia. Koodiesimerkissä 4 Docker-image luodaan mvn-komennolla.

```
create-docker-image:
  stage: build-image
  image: maven:3.6.3-jdk-11
  rules:
    - if: '$CI_COMMIT_MESSAGE !~ /CI-CD-BOT Image tag updated/ &&
($CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "development")'
      changes:
        - ggl-api/**/*
  cache:
    key: one-single-cache-key-for-everything
    paths:
      - .m2/repository
  script:
    - cd ggl-api
    - mvn package -Dquarkus.container-image.build=true  -Dquarkus.con-
tainer-image.tag=$CI_COMMIT_SHA -Dquarkus.container-image.push=true -
Dquarkus.container-image.username=$DOCKER_USER -Dquarkus.container-im-
age.password=$DOCKER_PASS -Dquarkus.container-image.group=$DOCKER_USER
```

Koodiesimerkki 4. CI/CD-putken työvaihe, jossa sovelluksesta luodaa Docker Hubiin vietävä Docker-image.

Työn suoritus alkaa, kun sovelluskehittäjä vie muutoksia development- tai main-haaraan. Tämä koskee myös tilanteita, joissa muutokset haaraan tulevat yhdistämispyynnön seurauksena. Suoritusta ei aloiteta, mikäli muutoksen viesti on “CI-CD-BOT Image tag updated”. Työ käyttää edellisen tavoin välimuistia nopeuttamaan koontia. Skriptiosiossa luetaan GitLabiin määritellyt ympäristömuuttujat $DOCKER_USER ja $DOCKER_PASS, joilla skripti tunnistautuu Docker Hubiin.

CI/CD-putken viimeisessä vaiheessa päivitetään koodihaaraa vastaavan yaml-hakemiston Deployment-olion imagen tunniste. Koodiesimerkissä 5 suoritukseen lisätään GitLabissä määritelty CICD_PRIVATE_KEY, jonka arvo on base64 koodattu yksityinen avain. Yksityisellä avaimella on oikeus viedä muutoksia suoraan development- ja main-haaroihin.

```
change-image-tag:
  stage: deploy
  rules:
    - if: '$CI_COMMIT_MESSAGE !~ /CI-CD-BOT Image tag updated/ &&
($CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "development")'
      changes:
        - ggl-api/**/*
  script:
    - 'command -v ssh-agent >/dev/null || ( apt-get update -y && apt-
get install openssh-client -y )'
    - eval $(ssh-agent -s)
    - ssh-add <(echo "$CICD_PRIVATE_KEY" | base64 -d)
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$CICD_PRIVATE_KEY" > ~/.ssh/id_ed25519
    - ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
    - git config --global user.email "juhoroiha@gmail.com"
    - git config --global user.name "Botti Veijari"
    - mkdir ~/ggl-gitti
    - cd ~/ggl-gitti
    - git clone git@gitlab.com:juiffi/ggl_java.git
    - cd ggl_java
    - git checkout $CI_COMMIT_BRANCH
    - (sed -i "s/ggl-api:.*/ggl-api:$CI_COMMIT_SHA/g" yaml/$CI_COM-
MIT_BRANCH/deployment/deployment.yml)
    - git add yaml
    - git commit -m "CI-CD-BOT Image tag updated"
    - git push
```

Koodiesimerkki 5. CI/CD-putken deploy-vaiheessa Deployment-olion imagen tunniste vaihdetaan ja muutos viedään versiohallinnan development- tai main-haaraan.

Ensiksi skriptiosuudessa asennetaan ssh-agent -komentorivisovellus, mikäli se ei ole jo asennettuna. Sen avulla ympäristömuuttujassa oleva yksityinen avain lisätään ssh-komennon käyttöön. Seuraavaksi git-repositorio kloonataan, valitaan oikea koodihaara ja imagen tunniste vaihdetaan sed-editorilla. Muutokset viedään versiohallintaan viestillä "CI-CD-BOT Image tag updated", jotta putken suoritus ei käynnisty uudelleen.

Kun putken toimivuus oli testattu, asetettiin GitLabin asetuksista koodihaarojen suojaus päälle development- ja main-haaroihin. Suojaus estää koodimuutosten viemisen haaroihin suoraan. Muutosten on tultava koodihaarojen yhdistämispyyntöjen kautta. Sovelluskehittäjän yrittäessä viedä muutoksia suojattuihin haaroihin, GitLab ilmoittaa suojauksesta kuvan 10 mukaisesti.

```
juiffi@fedora ~/Programming/ggl_java (development)$ git push
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 16 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 430 bytes | 430.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0), pack-reused 0
remote: GitLab: You are not allowed to push code to protected branches on this project.
To gitlab.com:juiffi/ggl_java.git
 ! [remote rejected] development -> development (pre-receive hook declined)
error: failed to push some refs to 'gitlab.com:juiffi/ggl_java.git'
juiffi@fedora ~/Programming/ggl_java (development)$
```

Kuva 10. Virheilmoitus, jonka käyttäjä saa yrittäessään viedä koodia suojattuun haaraan.

Poikkeukseksi lisättiin CI/CD-putken käyttämä yksityisen avain, sillä muuten putki ei voisi vaihtaa käytettävän Docker-imagen tunnistetta viimeisessä työvaiheessa. GitLabin käyttöliittymässä havaittiin epäjohdonmukaisuus yksittäisten avainten sallimisessa. Poikkeuksen lisääjä saa oikeuden viedä muutoksia suojattuihin haaroihin myös muilla tunnuksilleen lisätyillä ssh-avaimilla. Ongelman ratkaisemiseksi GitLab-projektille lisättiin käyttäjä, jota ei käytetä sovelluskehitykseen. Poikkeus suojaukseen lisättiin tämän käyttäjän avulla.

6.3 Tuotantoklusterin asennus

Hetznerin CX21-virtuaalipalvelimelle asennettiin RKE2 Kubernetes -jakelu. Asennusohjeina käytettiin RKE2:n dokumentaatiota. Valmis skripti asentaa jakelun systemd-palveluna. Klusterille asennettiin ArgoCD ja Bitnamin Sealed-Secrets samalla tavoin kuin testiklusterille. Asennuksen jälkeen suoritettiin kube-bench-työkalu, joka tarkistaa yleisimmät tietoturvapuutteisiin johtavat konfiguraatiovirheet. Kube-bench ei osannut etsiä RKE2:n konfiguraatiotiedostoja, sillä ne sijaitsivat eri hakemistoissa kuin tavallisella Kubernetes-asennuksella. Useimmat kohdat jouduttiin tarkistamaan käsin.

Kube-benchin tulosten tarkistamisen jälkeen klusterille asennettiin MetalLB-kuormanjako. MetalLB on ohjelmallisesti toteutettu kuormanjako, joka osaa reitittää liikenteen palvelinsolmujen välillä niiden kuormitukseen perustuen. MetalLB tarvitsee kuormanjakoa varten joukon IP-osoitteita [26]. Koska Hetznerillä

virtuaalipalvelimet eivät oletuksena ole samassa verkossa, palvelimia varten luotiin virtuaalinen verkko Hetzner Cloud -palvelun hallintapaneelin kautta. Vuokrattu virtuaalipalvelin lisättiin tähän verkkoon ja kuormanjaolle varattiin verkosta IP-osoitteet 10.0.0.240 - 10.0.0.250. Kuormanjaon asentaminen helpottaa palvelinsolmujen lisäämistä ja palvelujen tarjoamista porteista 80 ja 443. Hetzneriltä voi myös vuokrata kuormanjakopalvelua, mutta se on maksullinen palvelu.

Kuormanjaon asentamisen jälkeen klusterille asennettiin Nginx ingress -kontrolleri. Ingress-kontrollerin tarkoituksena on tarjota klusterille asennettuja sovelluksia klusterin ulkopuolelle. Ilman ingress-kontrolleria vain yksi sovellus voi varata palvelinsolmun portit 80 ja 443. Ingress-kontrolleri toimii käänteisenä välityspalvelimena ja reitittää liikenteen joko alidomainin tai polun perusteella oikeaan sovellukseen. Ingress-kontrolleri ja kuormanjako mahdollistavat sovelluksen skaalaamisen palvelinsolmuja ja rinnakkaisia suorituksia lisäämällä. [27.]

Ingress-kontrollerin jälkeen viimeinen puuttuva palanen oli varmenteiden hankkiminen. NoIP-palvelusta varattiin dynaaminen verkkotunnus juiffi.ddns.net. NoIP:n tarjoamat verkkotunnukset ovat ilmaisia ja niihin liitettyä IP-osoitetta voi vapaasti vaihtaa. Sen jälkeen klusterille asennettiin cert-manager. Cert-manager on Kuberneteksen varmenteiden hallinnan lisäosa. Tämän insinöörityön kannalta sen tärkein ominaisuus on vaatimusten 12 ja 13 täyttäminen, eli automaattinen varmenteiden hakeminen ja uusinta. Ingress-kontrollerin kanssa cert-managerilla voi helposti tarjota klusterille asennettuja sovelluksia https-yhteyden yli. ClusterIssuer ottaa yhteyttä varmenteen myöntäjätahoon ja suorittaa tarvittavat toimet varmenteen hankkimiseksi. Cert-managerin käyttöönotossa klusterille määriteltiin koodiesimerkin 6 mukainen ClusterIssuer-olio.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: juho.roiha@gmail.com
    preferredChain: ""
    privateKeySecretRef:
      name: letsencrypt-prod
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
    - http01:
        ingress:
          class: nginx
```

Koodiesimerkki 6. ClusterIssuer-olion määrittelevä yaml-dokumentti.

Esimerkin ClusterIssuer hakee ACME-protokollan avulla Let’s Encryptiltä TLS-varmenteen. ACME-protokollalla palvelimen haltija todistaa varmenteen myöntäjälle hallitsevansa verkkotunnusta vastaavaa palvelinta. Se estää varmenteen hankkimisen verkkotunnukselle, johon varmenteen hakijalla ei ole oikeutta. [28.]

Työkalujen asentamisen jälkeen määriteltiin ArgoCD:tä varten Ingress-olio. Koodiesimerkissä 7 määritellään Ingress-olio, joka ohjaa osoitteeseen argocd.juiffi.ddns.net osoitetut pyynnöt klusterin ArgoCD-asennukselle.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
  name: argocd-server-ingress
  namespace: argocd
spec:
  rules:
  - host: argocd.juiffi.ddns.net
    http:
      paths:
      - backend:
          service:
            name: argocd-server
            port:
              name: https
        path: /
        pathType: Prefix
  tls:
  - hosts:
    - argocd.juiffi.ddns.net
    secretName: argocd-secret
```

Koodiesimerkki 7. Ingress-olion määrittely, jossa TLS/SSL-päättely tehdään Ingress-kontrollerin sijaan kohdepalvelulla.

Usein liikenne ingress-kontrollerilta sovellukselle on salaamatonta ja vain käyttäjän ja ingress-kontrollerin välinen liikenne salataan. Koska ArgoCD käyttää portin 443 kautta myös gRPC-protokollaa, täytyy TLS-päättäminen (engl. termination) tehdä vasta ArgoCD:n kapselilla.

Koska MetalLB-kuormanjakaja tarjoaa porteista 80 ja 443 ingress-kontrollerin, testiympäristön asennusta varten määriteltyä Service-oliota jouduttiin muokkaamaan. Tyyppi vaihdettiin LoadBalancerista ClusterIP:ksi ja muokattu määrittely lisättiin hakemistoon yaml/main/services. ClusterIP-tyyppinen palvelu ei näy klusterilta ulkoverkoon. Sovellus paljastettiin ulkoverkkoon luomalla Ingress-olion avulla. Olion määrittely lisättiin versiohallinnan hakemistoon yaml/main/ingress. Tuotantoympäristön tietokantayhteyttä varten luotiin uusi Secret-määrittely. Testiympäristön Deloyment-määrittelyä voitiin käyttää tuotan-

toympäristössä. Uudet yaml-dokumentit vietiin versiohallintaan ja sovellus lisättiin ArgoCD:n hallittavaksi. ArgoCD haki onnistuneesti yaml-dokumentit versiohallinnasta ja sovellus oli saatavilla osoitteesta https://ggl-api.juiffi.ddns.net.

7 Hyväksymistestaus

Toteutuksen jälkeen varmistettiin, että toteutettu ympäristö vastaa vaatimuksia. Vaatimuksista johdettiin hyväskymistestit ja testit suoritettiin käsin. Monet testitapauksista olisi mahdollista myös automatisoida esimerkiksi Robot Frameworkin avulla.

Vaatimus 1 testattiin syöttämällä testiympäristön tietokantaan rivejä ja kyselemällä tietokantaa tuotantoympäristöön asennetun rajapintasovelluksen kautta. Rajapintasovellus ei palauttanut testiympäristön tietoja.

Seuraavaksi varmistettiin CI/CD-putken toimivuus. Testaaminen aloitettiin lisäämällä sovellukseen epäonnistuva testi. Epäonnistuva testi vietiin uudessa koodihaarassa versiohallintaan ja koodihaarasta luotiin yhdistämispyyntö development-haaraan. GitLabin käyttöliittymältä tarkistettiin, että putken suoritus alkaa ja putken lokitiedoista varmistettiin, että testit epäonnistuivat. Ympäristö toimi vaatimusten 2 ja 3 mukaisesti.

Epäonnistuva testi poistettiin koodihaarasta ja muutos vietiin versiohallintaan. Putken suoritus alkoi uudelleen, ja nyt testit menivät läpi. Testien jälkeen putki suoritti koodianalyysin, ja tulokset tarkistettiin SonarQuben käyttöliittymältä. Vaatimuksia 4 ja 5 ei testattu natiivikäännöksen huonon välityskyvyn takia, vaan sovelluksesta asennettiin JVM-käännös molempiin ympäristöihin. Putken suorituksen jälkeen varmistettiin, että uusi Docker-image oli viety Docker Hubiin ja että ArgoCD oli asentanut uuden imagen. ArgoCD:n synkronointia täytyi odottaa, sillä oletuksena synkronointi tapahtuu kolmen minuutin välein. Uusi image asentui testiklusterille, ja ympäristö toimi siten vaatimusten 6, 7, ja 8 mukaisesti. Development-haarasta tehtiin yhdistämispyyntö main-haaraan ja varmistettiin,

että putki suorittaan asennuksen vaatimuksen 10 mukaisesti myös tuotantoympäristöön.

Vaatimus 10 testattiin vuokraamalla Hezneriltä virtuaalipalvelin ja asentamalla siihen RKE2 Agent. Palvelin liitettiin osaksi klusteria ja klusterille asennetun sovelluksen rinnakkaisia suorituksia lisättiin. Määrää kasvatettiin, kunnes suoritusten yhteenlaskettujen resurssivarausten määrä ylitti yhden palvelinsolmun klusterin kapasiteetin. ArgoCD:n käyttöliittymältä varmistettiin, että kaikki rinnakkaiset kapselit olivat päällä. Vaatimus 11 testattiin poistamalla kubectl-komennolla yksi pod klusterilta ja varmistamalla, että kapselien määrä palaa takaisin asetettuun arvoon.

Viimeisenä testattiin varmenteisiin kohdistuvat vaatimukset 12 ja 13. Vaatimus 12 testattiin avaamalla asennettu sovellus selaimen kautta ja varmistamalla, että yhteys muodostetiin käyttämällä https-protokollaa. Samalla tarkistettiin, että varmenne oli Let’s Encryptin myöntämä, eikä palvelimen itse allekirjoittama. Vaatimus 13 testattiin asettamalla cert-managerin renew-before arvo 89 päivään. Koska Let’s Encryptin varmenteet ovat voimassa kolme kuukautta kerrallaan, cert-manager haki uuden varmenteen myötämispäivää seuraavana päivänä. Hyväksyntätestauksen perusteella toteutettu ympäristö täyttää sille asetetut vaatimukset.

8 Yhteenveto

Insinöörityön tavoitteena oli tuottaa sovelluksen julkaisua ja ylläpitoa helpottava järjestelmä, jossa mahdollisimman suuri osa sovelluksen julkaisuprosessista on automatisoitu. Insinöörityössä esiteltiin konttiteknologian historiaa, Kubernetesta ja GitOps-menetelmää. Pohjustuksen jälkeen suunniteltiin GitOps-menetelmällä käytettävä ympäristö ja toteutettiin se. Suunnitteluvaiheessa valittiin käytettävät palvelut ja työkalut sekä suunniteltiin jatkuvan integroinnin ja jatkuvan toimituksen putket. Valinnat tehtiin asetettujen vaatimusten, suorituskykymittauksen ja lähdekirjallisuuden pohjalta.

Toteutuksessa asennettiin Kubernetes-klusteri työasemalla sijaitsevalle virtuaalikoneelle ja Hetzneriltä vuokratulle virtuaalipalvelimelle. Työasemalle asennettu klusteri toimii kehitysympäristönä ja vuokrapalvelimen klusteri tuotantoympäristönä, josta sovellus on julkaistu internetiin. CI/CD-putken toteutus julkaisee testatut koodimuutokset automaattisesti versiohallinnan haaroista klustereille.

Työn tuloksena syntyi automatisoitu ympäristö, jonka avulla voi julkaista Docker-kontteihin paketoituja sovelluksia ja ohjelmointirajapintoja. Ympäristö on laajennettavissa, mikäli kuorman tai sovellusten määrä kasvaa klusterin kapasiteettia suuremmaksi. Paikallinen testiklusteri toimii hyvin kehittämisen tukena ja siitä oli työn aikana paljon hyötyä. Testiklusterin konfiguraatioiden uudelleenhyödyntäminen tuotantoklusterissa oli kohtuullisen helppoa. Työtä voidaan pitää onnistuneena ja työn alussa asetettua tavoitetta saavutettuna.

Ympäristöä voi jatkokehittää esimerkiksi asentamalla tietokanta klusterin sisälle. Nykyratkaisussa yhdelle virtuaalipalvelimelle asennettu tietokanta jää kuormituksen kasvaessa pullonkaulaksi. Toinen hyvä kehittämiskohde olisi sovelluksen kapseleiden ja klusterin palvelinsolmujen määrän automaattinen skaalaaminen kuorman perusteella. Nykytoteutuksessa ympäristö ei reagoi kasvaneeseen kuormaan lisäämällä sovellukselle resursseja tai liittämällä klusteriin lisää solmuja.

Lähteet

1 Sean P. Kane, Karl Matthias. 2018. O'Reilly Media. Docker: Up & Running, 2nd Edition. Verkkoaineisto. https://learning.oreilly.com/library/view/docker-up/9781492036722/. Luku 1. Luettu 24.8.2021.

2 Docker Inc. Explore Docker's Container Image Repository. Verkkoaineisto. https://hub.docker.com/search?type=image. Luettu 24.8.2021.

3 Billy Yuen, Jesse Suen, Alex Matyushentsev, Todd Ekenstam. 2021. O'Reilly Media. GitOps and Kubernetes. Verkkoaineisto. https://learning.oreilly.com/library/view/gitops-and-kubernetes/9781617297274. Luettu 13.7.2021.

4 Docker Inc. Docker Documentation. Verkkoaineisto. https://docs.docker.com/config/containers/resource_constraints/. Luettu 29.7.2021.

5 Microsoft. Kubernes vs. Docker. Verkkoaineisto. https://azure.microsoft.com/en-us/topic/kubernetes-vs-docker/. Luettu 24.8.2021.

6 Martin Fowler. 2006. Continuous Integration. Verkkoaineisto. https://www.martinfowler.com/articles/continuousIntegration.html. Luettu 4.9.2021.

7 Mikael Krief. 2019. Learning DevOps. Packt Publishing. Verkkoaineisto. https://learning.oreilly.com/library/view/learning-devops/9781838642730/. Luettu. 4.9.2021.

8 Brian Proffitt. 2011 Hudson devs vote for name change; Oracle declares fork. Computerworld. Verkkoaineisto. https://www.computerworld.com/article/2746627/hudson-devs-vote-for-name-change--oracle-declares-fork.html. Luettu 4.9.2021.

9 Jenkins.io. 2021. User Handbook. Verkkoaineisto. https://www.jenkins.io/doc/book/pipeline/syntax/. Luettu 8.10.2021.

10 Jenkins X Authors. Introduction to what Jenkins X is. 2021. Verkkoaineisto. https://jenkins-x.io /v3/about/what/. Luettu 4.9.2021.

11 Alona Hlobina, Paul Gordon, Nicolas Rios. 2020. The new pricing model for travis-ci.com. Verkkoaineisto. https://blog.travis-ci.com/2020-11-02-travis-ci-new-billing. Luettu 4.9.2021.

12 Travis CI GmbH. Languages. 2021. Verkkoaineisto. https://docs.travis-ci.com/user/languages/. Luettu 8.10.2021.

13 GitLab Inc. GitLab CI/CD for external repositories. 2021. Verkkoaineisto. https://docs.gitlab.com/ee/ci/ci_cd_for_external_repos/. Luettu 8.10.2021.

14 GitLab Inc. Pricing. 2021. Verkkoaineisto. https://about.gitlab.com/pricing/. Luettu 8.10.2021.

15 Marko Lukša. 2018. Kubernetes in Action. Manning Publications. Verkkoaineisto. https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/. Luettu 25.8.2021.

16 The Kubernetes Authors. 2021. Understanding Kubernetes Objects. Verkkoaineisto. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. Luettu 8.10.2021.

17 Ádám Sándor. 2020. GitOps: The Bad and the Ugly. Verkkoaineisto. https://blog.container-solutions.com/gitops-limitations. Luettu 8.10.2021.

18 Chris Ward. 2021. The Pros and Cons of GitOps. Verkkoaineisto. https://humanitec.com/blog/gitops-pros-and-cons. Luettu 8.10.2021.

19 Gitlab Inc. 2021. Merge request approvals. Verkkoaineisto. https://docs.gitlab.com/ee/user/project/merge_requests/approvals/. Luettu 8.10.2021.

20 The Kubernetes Authors. 2021. Minikube start. Verkkoaineisto. https://minikube.sigs.k8s.io/docs/start/. Luettu 8.10.2021.

21 Canonical. 2021. Introduction to MicroK8s. Verkkoaineisto. https://microk8s.io/docs. Luettu 8.10.2021.

22 Rancher. 2021. Installation Requirements. Verkkoaineisto. https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/. Luettu 8.10.2021.

23 SonarSource S.A. 2021. Deploy SonarQube on Kubernetes. Verkkoaineisto. https://docs.sonarqube.org/latest/setup/sonarqube-on-kubernetes/. Luettu 8.10.2021.

24 GitLab Inc. 2021. GitLab Runner Helm Chart. Verkkoaineisto. https://docs.gitlab.com/runner/install/kubernetes.html. Luettu 8.10.2021.

25 Rancher. 2021. RKE2 Documentation. Verkkoaineisto. https://docs.rke2.io/. Luettu 8.10.2021.

26 The Kubernetes Authors. 2021. Service. Verkkoaineisto. https://kubernetes.io/docs/concepts/services-networking/service/. Luettu 8.10.2021.

27 MetalLB. 2021. Configuration. Verkkoaineisto. https://metallb.universe.tf/configuration/. Luettu 8.10.2021.

28 The Kubernetes Authors. 2021. Ingress. Verkkoaineisto. https://kubernetes.io/docs/concepts/services-networking/ingress/. Luettu 8.10.2021.

29 Patrick Nohe. 2019. ACME Protocol: What it is and how it works. Verkkoaineisto. https://www.thesslstore.com/blog/acme-protocol-what-it-is-and-how-it-works/. Luettu 8.10.2021.

Jatkuvan integraation ja toimituksen lähdekoodi

```
image: maven:3.8.2-openjdk-11

variables:
  MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.repo.local=$CI_PRO-
JECT_DIR/.m2/repository -Dorg.slf4j.simpleLogger.log.org.apache.ma-
ven.cli.transfer.Slf4jMavenTransferListener=WARN -Dorg.slf4j.simp-
leLogger.showDateTime=true -Djava.awt.headless=true"

stages:
  - cicd-pipeline-requirement
  - qa
  - build-image
  - deploy

empty-run:
  stage: cicd-pipeline-requirement
  image: busybox:latest
  rules:
    - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
  script:
    - echo " Empty pipeline run."
    - echo true

sonarqube-job:
  rules:
   - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
     changes:
      - ggl-api/**/*
  stage: qa
  variables:
    SONAR_USER_HOME: "${CI_PROJECT_DIR}/.sonar"
    GIT_DEPTH: "0"
  cache:
    key: one-single-cache-key-for-everything
    paths:
      - .m2/repository
      - .sonar/cache
  script:
    - cd ggl-api
    - mvn org.owasp:dependency-check-maven:check
    - mvn verify sonar:sonar

create-docker-image:
  stage: build-image
  rules:
    - if: '$CI_COMMIT_MESSAGE !~ /CI-CD-BOT Image tag updated/ &&
($CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "development")'
      changes:
        - ggl-api/**/*
  cache:
    key: one-single-cache-key-for-everything
    paths:
      - .m2/repository
  script:
    - cd ggl-api
```

```
    - mvn package -Dquarkus.container-image.build=true  -Dquarkus.con-
tainer-image.tag=$CI_COMMIT_SHA -Dquarkus.container-image.push=true -
Dquarkus.container-image.username=$DOCKER_USER -Dquarkus.container-im-
age.password=$DOCKER_PASS -Dquarkus.container-image.group=$DOCKER_USER

change-image-tag:
  stage: deploy
  rules:
    - if: '$CI_COMMIT_MESSAGE !~ /CI-CD-BOT Image tag updated/ &&
($CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "development")'
      changes:
        - ggl-api/**/*
  script:
    - 'command -v ssh-agent >/dev/null || ( apt-get update -y && apt-
get install openssh-client -y )'
    - eval $(ssh-agent -s)
    - ssh-add <(echo "$CICD_PRIVATE_KEY" | base64 -d)
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$CICD_PRIVATE_KEY" > ~/.ssh/id_ed25519
    - ssh-keyscan gitlab.com >> ~/.ssh/known_hosts
    - git config --global user.email "juhoroiha@gmail.com"
    - git config --global user.name "Botti Veijari"
    - mkdir ~/ggl-gitti
    - cd ~/ggl-gitti
    - git clone git@gitlab.com:juiffi/ggl_java.git
    - cd ggl_java
    - git checkout $CI_COMMIT_BRANCH
    - (sed -i "s/ggl-api:.*/ggl-api:$CI_COMMIT_SHA/g" yaml/$CI_COM-
MIT_BRANCH/deployment/deployment.yml)
    - git add yaml
    - git commit -m "CI-CD-BOT Image tag updated"
    - git push
```

Hyväksyntätestauksen tulokset

Vaatimus	Testi	Tulos
1	Syötetään testikantaan Mock-dataa ja varmistetaan ettei niitä saada haettua tuotantoympäristöstä	OK
2	Tarkistetaan CI/CD-putken lokeilta, että testit on suoritettu.	OK
3	Lisätään epäonnistuva yksikkötesti versiohallintaan ja suoritetaan putki	OK
4	Vaatimusta ei testata, sillä sovelluksesta suoritetaan vain JVM-koonti	-
5	Vaatimusta ei testata, sillä sovelluksesta suoritetaan vain JVM-koonti	-
6	Kirjautudutaan putken suorituksen jälkeen Docker Hubiin ja varmistetaan, että uusi image on saatavilla	OK
7	Tarkistetaan SonarQuben käyttöliittymältä analyysin tulokset	OK
8	Viedään muutos development-haaraan ja varmistetaan, että vastaava muutos asentuu testiklusterille	OK
9	Viedään muutos main-haaraan ja varmistetaan, että vastaava muutos asentuu tuotantoklusterille	OK
10	Asennetaan RKE2 Agent uudelle virtuaalipalvelimelle, liitetään se klusteriin ja skaalataan sovellusta suuremmaksi kuin yksittäinen palvelinsolmu kykenee suorittamaan. Varmistetaan ArgoCD:ltä, että käynnissä olevien kapseleiden määrä kasvaa yhden palvelinsolmun kapasiteetin yli.	OK
11	Poistetaan kubectl-työkalulla kapseli klusterilta ja varmistetaan, että kapseli käynnistyy uudelleen	OK
12	Avataan sovellus verkkoselaimella ja varmistetaan että käytetty protokolla on https.	OK
13	Asetetaan cert-managerin renewBefore -arvo 89 päivään ja varmistetaan, että varmenne on vaihtunut päivän kuluttua	OK

Suorituskykymittauksen tulokset

Kokoonpano	Kesimääräinen vaste-aika	Keskimääräinen väli-tyskyky (pyyntöä se-kunnissa)
8G_JVM	107.68 ms.	5808.07
8G_µK8s_JVM	115.6 ms.	4891.03
8G_K3s_JVM	118.56 ms.	4921.95
8G_K3s_NATIVE	216.01 ms.	2673.14
8G_µK8s_NATIVE	225.05 ms.	2552.58
4G_JVM	242.33 ms.	2668.59
4G_K3s_JVM	244.37 ms.	2344.06
4G_µK8s_JVM	263.49 ms.	2250.79
8G_NATIVE	295.15 ms.	2146.7
2G_JVM	387.32 ms.	1705.04
2G_K3s_JVM	410.26 ms.	1621.58
2G_µK8s_JVM	471.12 ms.	1426.1
4G_K3s_NATIVE	583.12 ms.	1089.88
4G_NATIVE	616.64 ms.	1093.29
4G_µK8s_NATIVE	619.57 ms.	1032.15
2G_NATIVE	931.54 ms.	732.06
2G_K3s_NATIVE	1044.77 ms.	657.83
2G_µK8s_NATIVE	1075.61 ms.	639.95