Teemu Luhtanen

Kuberneteksen, monihaaraisten liukuhihnojen ja jaettujen kirjastojen käyttö Jenkins-töiden hallinnassa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

16.5.2023

Tiivistelmä

Tekijä:	Teemu Luhtanen
Otsikko:	Kuberneteksen, monihaaraisten liukuhihnojen ja jaettujen kirjastojen käyttö Jenkins-töiden hallinnassa
Sivumäärä:	44 sivua + 0 liitettä
Aika:	16.5.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander Järjestelmäsuunnittelija Kalle Immonen

---

Insinöörityön tavoitteena oli ratkaista Jenkins-työsolmujen skaalautuvuusongelma ottamalla käyttöön Kubernetes-klusteri. Jenkins-työsolmujen määrä on staattinen eli niiden määrää ei voida säädellä kysynnän mukaisesti. Tämän vuoksi suuriin kysyntäpiikkeihin ei pystytä vastaamaan niin nopeasti, jos kaikki solmut ovat jo käytössä. Kubernetes puolestaan pystyy säätelemään konttien määrää kysynnän mukaisesti automaattisen skaalauksen ansiosta. Tämän johdosta kysyntään pystytään vastaamaan nopeammin.

Lisäksi tavoitteena oli ottaa Jenkins-töissä käyttöön monihaaraiset liukuhihnat sekä jaetut kirjastot. Monihaaraisten liukuhihnojen avulla Jenkins pystyy samasta projektista tunnistamaan useampia haaroja, joissa on versionhallinnassa mukana Jenkins-tiedosto. Jokainen haara voidaan koota ja kokoamisvaiheessa Jenkins käyttää haaran versionhallinnasta löytyvää Jenkins-tiedostoa liukuhihnana. Jaettujen kirjastojen avulla puolestaan luotiin vakioitu pohja eli liukuhihna, jota jokainen projekti voisi käyttää kokoamisvaiheissaan. Tarkoituksena oli, että vakioidun pohjan vaiheet olisivat kaikissa projekteissa samat, mutta vaiheet suoritettaisiin projektikohtaisten parametrien ja muuttujien mukaisesti.

Kuberneteksen käyttöönotto toteutettiin aluksi yhteen projektiin ilman vakioitua pohjaa. Tämän jälkeen siirryttiin vakioidun pohjan luontiin ja sen käyttöönottoon yhdessä projektissa. Lopuksi vakioitu pohja otettiin käyttöön muissa tarvittavissa projekteissa ja varmistuttiin niiden toiminnasta.

Kuberneteksen käyttöönotossa onnistuttiin hyvin ja skaalautuvuusongelma saatiin ratkaistua. Työn käynnistyessä etsitään sopiva kapseli sen suorittamiseen, jonka johdosta suoritusaika hidastui muutamalla sekunnilla. Monihaaraiset liukuhihnat saatiin toimimaan odotetulla tavalla ja jokaisen haaran, jossa on Jenkins-tiedosto, voi koota. Myös jaetun kirjaston avulla toteutettu vakioitu pohja saatiin toimimaan tarvittavilla projekteilla. Käyttöönottovaiheessa tarvittiin vielä pieniä korjauksia pohjaan, jotta se toimisi oikein jokaisessa projektissa.

Avainsanat:	Kubernetes, Jenkins, monihaaraiset liukuhihnat, jaetut kirjastot

Abstract

Author:	Teemu Luhtanen
Title:	Using Kubernetes, Multibranch Pipelines and Shared Libraries to Manage Jenkins Jobs
Number of Pages:	44 pages + 0 appendices
Date:	16 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer Kalle Immonen, System Designer

---

The goal of the study was to solve the scalability problem of Jenkins worker nodes by implementing a Kubernetes cluster. The number of Jenkins worker nodes is static, meaning that their number cannot be adjusted according to demand. Because of this, it is not possible to respond to large demand spikes as quickly if all the nodes are already in use. Kubernetes, on the other hand, is able to regulate the number of containers according to demand thanks to automatic scaling. As a result, demand can be responded to more quickly.

In addition, the goal was to implement multibranch pipelines and shared libraries in Jenkins jobs. With the help of multibranch pipelines, Jenkins can identify several branches from the same project that have a Jenkinsfile included in the version control. Each branch can be built, and in the building phase, Jenkins uses the Jenkinsfile found in the version control of the branch as a pipeline. Shared libraries, on the other hand, were used to create a standardized base. The standardized base is a pipeline that each project can use in its building phases. The purpose was that the stages of the standardized base would be the same in every project, but the stages would be run according to project-specific parameters and variables.

The implementation of Kubernetes was initially implemented in one project without a standardized base. After that, the creation of the standardized base and its implementation in one project took place. Finally, the standardized base was put into use in other necessary projects and their operation was confirmed.

The implementation of Kubernetes was successful, and the scalability problem was solved. When the job starts, a suitable cluster is searched for its execution, which slows down the execution time by a few seconds. Multibranch pipelines were working as expected and it was possible to build each branch with a Jenkinsfile. The standardized base implemented with the help of a shared library was also working with the necessary projects. During the implementation phase, some minor fixes to the base were needed so that it would work correctly in each project.

Keywords: Kubernetes, Jenkins, multibranch pipelines, shared libraries

Sisällys

Lyhenteet

CI/CD: Jatkuva integrointi ja julkaisu. Lyhenne tulee englannin kielen sanoista *continuous integration* ja *continuous delivery*.

DevOps: Toimintamalli palveluiden tuotantoon. Pyrkii automatisoimaan ohjelmistokehitykseen, testaukseen ja ylläpitoon liittyvät toiminnot. Lyhenne tulee englannin kielen sanoista *development* ja *operations*.

K8s: Kubernetes. Lyhenteen numero 8 tarkoittaa kirjainten lukumäärää K:n ja S:n välissä.

GUI: Graafinen käyttöliittymä. Lyhenne tulee englannin kielen sanoista *grapfical user interface*.

API: Ohjelmointirajapinta. Lyhenne tulee englannin kielen sanoista *application programming interface*.

PR: Lyhenne englannin kielen sanoista *pull request*. Sen avulla voidaan kertoa muille, että koodiin on tehty muutoksia. Kun PR avataan, muut voivat keskustella muutoksista ja antaa parannusehdotuksia. Muut voivat myös hyväksyä PR:n, jos muutokset näyttävät hyvältä.

SCM: Lyhenne tulee englannin kielen sanoista *source control management*. Sillä tarkoitetaan versionhallintajärjestelmää. Versionhallintajärjestelmät tarjoavat jatkuvan koodin kehityshistorian ja auttavat ratkaisemaan ristiriitoja, kun yhdistetään useista lähteistä olevia muutoksia.

1 Johdanto

Insinöörityön tarkoituksena on ratkaista Jenkins-työsolmujen (engl. worker node) skaalautuvuusongelma sekä helpottaa useamman projektin CI/CD-putken hallintaa. Jenkins on jatkuvan integraation ja käyttöönoton (CI/CD) automaatio-ohjelmiston DevOps-työkalu. Sen toimintaa toteuttavat liukuhihnat (engl. pipeline). Liukuhihnat automatisoivat koodikannan muutosten testauksen ja raportoinnin reaaliajassa ja helpottavat koodin eri haarojen (engl. branch) integrointia päähaaraan.

Skaalautuvuusongelma ratkaistaan siirtymällä käyttämään Kubernetes-klusteria Jenkins-työsolmujen sijasta. Kubernetes-klusterin ansiosta töiden työntekijämäärä on dynaaminen staattisten Jenkins-työsolmujen sijasta. Kubernetes (K8s) on järjestelmä, jolla voidaan ottaa käyttöön, skaalata ja hallita konttisovelluksia (engl. containerized applications) missä tahansa. Kubernetes automaattisesti säännöstelee ja sovittaa kontit solmuihin resurssien parhaaseen käyttöön.

Kuberneteksen käyttöönoton lisäksi otetaan Jenkinsissä käyttöön monihaaraiset liukuhihnat sekä jaetut kirjastot. Monihaaraiset liukuhihnat helpottavat Jenkinsin käyttöä usean projektin kanssa jatkossa, sekä Jenkinsiin saadaan myös mukaan useampia haaroja. Jaettuja kirjastoja käytetään puolestaan vakioidun pohjan toteutuksessa. Vakioidulla pohjalla tarkoitetaan tässä tapauksessa liukuhihnaa, joka on jokaiselle projektille yhteinen ja jonka jokainen projekti voi suorittaa sen omilla parametreillaan. Vakioitu pohja sisältäisi kaikille projekteille samat vaiheet, mutta projektikohtaisilla parametreilla. Vakioidun pohjan tarkoituksena on saada uusien töiden lisäämisestä vaivattomampaa ja yksinkertaisempaa.

Insinöörityö toteutetaan Samlinkin projektitiimin jäsenenä, projektitiimin sekä asiakkaan puolen sovelluskehittäjien käyttöön. Tavoitteena on aluksi toteuttaa Kuberneteksen käyttöönotto yhteen projektiin ilman vakioitua pohjaa. Tämän jälkeen siirrytään vakioidun pohjan luontiin ja otetaan se käyttöön aluksi yhdessä projektissa ja varmistutaan sen toimivuudesta. Lopuksi pohja otetaan

käyttöön kaikissa tarvittavissa projekteissa ja varmistetaan toimivuus jokaisessa.

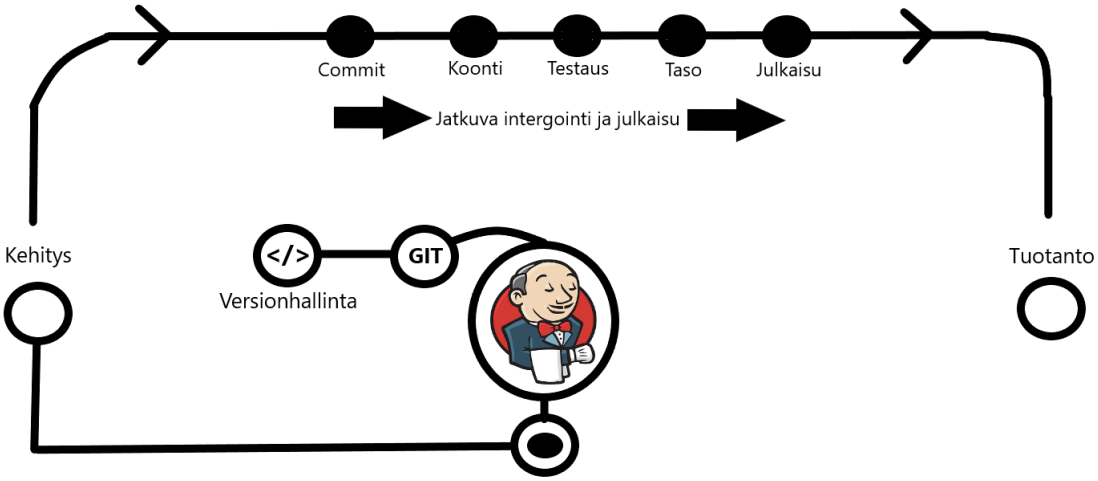
2 Teknologioiden esittely

Tässä luvussa esitellään insinöörityölle ominaiset ja tärkeimmät käytettävät teknologiat kehitystyötä tehtäessä. Tärkeimpinä teknologioina ovat Jenkins, minkä ympärillä insinöörityö suurimmilta osin tehdään, sekä Kubernetes, jonka klusteria siirrytään käyttämään. Näiden teknologioiden lisäksi kehitystyössä käytetään JavaScript- tai TypeScript-ohjelmointikieltä projektin mukaan. Lisäksi Jenkinsin liukuhihnat sekä jaetut kirjastot käyttävät Groovy-ohjelmointikieltä. Useassa jaettujen kirjastojen Groovy-tiedostoissa suoritetaan myös shell-skriptejä.

2.1 Jenkins

Jenkins on avoimeen lähdekoodiin perustuva jatkuvan integraation ja käyttöönoton (CI/CD) automaatio-ohjelmiston DevOps-työkalu. Se on kirjoitettu Java-ohjelmointikielellä ja se tukee useita eri versionhallintajärjestelmiä sekä ohjelmointikieliä. Sitä käytetään toteuttamaan CI/CD-työnkulkua, jota kutsutaan liukuhihnaksi. [1.]

Liukuhihnat automatisoivat koodikannan muutosten testauksen ja raportoinnin sekä helpottavat koodin eri haarojen integrointia päähaaraan. Ne myös havaitsevat nopeasti koodikannan viat, kokoavat (engl. build) ohjelmiston, automatisoivat versioiden testauksen, valmistelevat koodikannan käyttöönottoa toimitusta varten ja lopulta ottavat koodin käyttöön konteissa (engl. container) ja virtuaalikoneissa tai fyysisissä tietokoneissa ja pilvipalvelimissa. [1.] Kuvassa 1 nähdään esimerkkikuva automatisoidusta Jenkins CI/CD -liukuhihnasta.



Kuva 1. Automatisoitu Jenkins CI/CD -liukuhihna. Perustuu lähteeseen [2].

Kuvasta 1 nähdään liukuhihnan vaiheet. Alussa haetaan gitin avulla versionhallinnasta uusi tai olemassa oleva projekti, johon tehdään muutoksia. Jenkins määrittää tälle projektille työn. Työn käynnistyessä Jenkins suorittaa liukuhihnassa annetut vaiheet, jotka ovat kuvassa kohdat commit, koonti, testaus ja taso. Näiden vaiheiden jälkeen muutokset ovat valmiita julkaistavaksi ympäristöihin esimerkiksi tässä tapauksessa tuotantoon.

2.1.1 Toiminta

Jenkinsiä voidaan käyttää palvelimena useissa käyttöjärjestelmissä, kuten Windowsissa, macOS:ssa, Unix-versioissa ja Linuxissa. Jenkins suoritetaan usein Java-palvelinsovelmassa (engl. servlet), Jetty- tai muissa sovelluspalvelimissa, kuten Apache Tomcatissa. Jenkins on hiljattain muutettu toimimaan myös Docker-kontissa. [3.] Tämä tarkoittaa sitä, että Jenkins voidaan ladata esimerkiksi Docker-kuvina (engl. image), joista jokainen ajetaan Dockerissa konttina. Docker-kontti on käytännössä Docker-kuvan ”esiintymä”. [4.]

Jenkinsin sisältö tallennetaan paikallisesti Jenkins-tiedostoon pelkkänä tekstinä. Jenkins-tiedostossa on JSONia muistuttava aaltosulkeiden syntaksi.

Liukuhihnan vaiheet on suljettu hakasulkeisiin ja määritelty komentoina argumenteilla. Esimerkkikoodissa 1 on esimerkki Jenkins-tiedoston syntaksista. [3.]

```
pipeline {
     agent any
     stages {
          stage ('Stage 1') {
               steps {
                    echo 'Hello world!'
               }
          }
     }
}
```

Esimerkkikoodi 1. Yksinkertainen esimerkki Jenkins-tiedoston syntaksista (deklaratiivinen skripti) [5].

Esimerkkikoodista 1 nähdään, että deklaratiivinen skripti alkaa aina pipeline-lohkolla. Siihen kuuluu aina myös stages-lohko, jonka sisällä läpikäytävät vaiheet ovat. Jokainen vaihe määritellään stage-lohkolla. Deklaratiivisessa skriptissä stage-lohkojen sisällä on aina steps-lohko, jonka sisällä suoritetaan halutut tehtävät.

Jenkins-palvelin lukee Jenkins-tiedoston ja suorittaa tehtävät työntäen koodin kommitoidusta lähdekoodista ajonaikaiseen tuotantoon. Jenkins-tiedostoja voidaan tuottaa käyttämällä graafista käyttöliittymää (GUI) tai kirjoittamalla koodia manuaalisesti projektin lähdekoodissa olevaan Jenkins-tiedostoon, kuten tässä insinöörityössä tehdään. [3.]

Tässä insinöörityössä Jenkins-tiedostot ja liukuhihnat ovat tärkeässä osassa. Jokaiselle projektille luodaan omat Jenkins-tiedostonsa, joissa kutsutaan vakioitua pohjaa projektikohtaisia parametreja käyttäen. Vakioitu pohja puolestaan on liukuhihna, jota kaikki projektit pystyvät käyttämään rakennusvaiheissaan.

2.1.2 Laajennukset

Laajennus (engl. plugin) on Jenkins-järjestelmän parannus. Ne auttavat laajentamaan Jenkinsin ominaisuuksia ja integroivat Jenkinsin muihin ohjelmistoihin.

Laajennukset voidaan ladata Jenkins Plugin -tietovarastosta. Laajennukset kirjoitetaan Java-ohjelmointikielellä ja Jenkinsille on saatavilla satoja laajennuksia erilaisiin käyttötarkoituksiin. [1.]

Laajennukset auttavat integroimaan muita kehittäjätyökaluja Jenkins-ympäristöön, lisäämään uusia käyttöliittymäelementtejä Jenkinsin verkkokäyttöliittymään, auttavat Jenkinsin hallinnassa ja parantavat Jenkinsin koonti- ja lähdekoodin hallintaa. Yksi laajennuksien yleisimmistä käyttötavoista on integrointipisteiden tarjoaminen CI/CD-lähteille ja -kohteisiin. Näitä ovat versionhallintajärjestelmät (SCM), kuten Git ja Atlassian BitBucket, konttiajonkohtaiset järjestelmät, erityisesti Docker, virtuaalikoneen valvojat (engl. hypervisor), kuten VMware vSphere sekä julkiset pilvi-instanssit, kuten Google Cloud Platform ja Amazon AWS. [1.]

Myös tässä insinöörityössä käytetään useita Jenkins-laajennuksia. Jotkut niistä ovat välttämättömiä, jotta ominaisuutta pystytään käyttämään Jenkinsissä, esimerkiksi laajennus monihaaraisen liukuhihnan tai Bitbucket-organisaatiokansion (engl. organization folder) käyttöön. Jotkut taas helpottavat Jenkinsin käytössä tai esimerkiksi Jenkinsin ja Bitbucketin välisessä kommunikaatiossa, mutta eivät ole välttämättömiä.

2.1.3 Hyödyt ja haitat

Jenkinsillä on myös omat hyvät ja huonot puolensa. Yksi sen eduista ovat aiemmin mainitut laajennukset. Niiden ansiosta Jenkinsiä voidaan laajentaa lisäosien avulla, mikä tekee siitä mukautuvan IT-ympäristöjen muutoksiin. Laajennukset lisäävät myös joustavuutta, samoin kuin monipuoliset komentosarja- ja deklaratiiviset kielet, jotka mahdollistavat laajasti muokattavat liukuhihnat. Jenkins sopii hyvin useimpiin ympäristöihin, mukaan lukien monimutkaiset hybridi- ja monipilvijärjestelmät. [1.]

Jenkins on ollut olemassa pidempään kuin muut ratkaisut alalla. Jenkins julkaistiin ensimmäisen kerran vuoden 2011 alussa. Tämä ja sen joustavuus ovat

johtaneet sen laajaan käyttöön. Laajan käytön vuoksi Jenkins tunnetaan hyvin ja sillä on laaja tietopohja, dokumentaatio sekä yhteisön resurssit. Nämä kaikki helpottavat Jenkinsin asentamista, hallintaa ja vianetsintää. [1.]

Vaikka Jenkins on helppo asentaa ohjeiden avulla, Jenkinsin tuotanto voi olla vaikeaa toteuttaa. Tuotantoliukuhihnojen kehittäminen Jenkins-tiedostoilla vaatii koodausta joko deklaratiivisella tai komentosarjakielellä (engl. scripting language). Erityisesti monimutkaisia liukuhihnoja voi olla vaikea koodata, korjata ja ylläpitää. [1.]

Insinöörityön aikana on pystynyt myös itse huomaamaan mainitut Jenkinsin hyvät puolet. Laajennuksien avulla todella saadaan Jenkinsiin joustavuutta, ja laajennuksia löytyykin todella moneen eri tarkoitukseen. Lisäksi lähes kaikkiin Jenkinsin kanssa tuleviin ongelmiin löytyy jonkunlaista apua tai ohjeistusta. Tietenkään monet asiat eivät ole suoraan liitännäisiä omaan ongelmaan, mutta niistä on kuitenkin silti monesti apua ongelman ratkaisemisessa. Vaikka Jenkins-tiedostoissa ja liukuhihnojen koodaamisessa olikin jonkin verran perehtymistä ja opettelua, onnistui se minun osaltani kuitenkin suhteellisen hyvin. Pieniä ongelmia aluksi tietenkin oli, mutta mitään ylitsepääsemätöntä ei missään vaiheessa ilmennyt. Tämän insinöörityön tapauksessa liukuhihnat eivät kuitenkaan ole niin pitkiä ja monimutkaisia kuin jossain tapauksissa voisi olla. Voin kuitenkin hyvin kuvitella, että monimutkaisia sekä pitkiä liukuhihnoja voi olla vaikeampaa luoda ja ylläpitää.

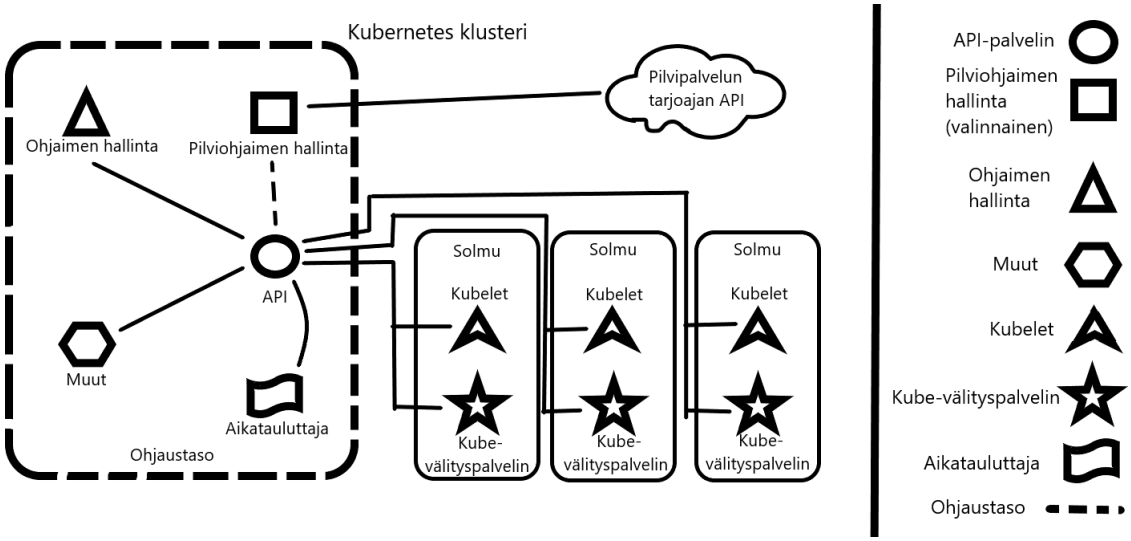
2.2 Kubernetes

Kubernetes (K8s) on avoimen lähdekoodin järjestelmä, jolla voidaan ottaa käyttöön, skaalata ja hallita konttisovelluksia missä tahansa. Kubernetes automatisoi konttihallinnan tehtävät ja sisältää sisäänrakennetut komennot sovellusten käyttöönottoon, sovelluksiin tehtyjen muutosten käyttöönottoon, sovellusten skaalaamiseen ja muuttuviin tarpeisiin, sovellusten valvontaan ja myös moniin muihin asioihin, jotka helpottavat sovellusten hallintaa. [6.] Tämän insinöörityön

tapauksessa Kubernetes hoitaa Jenkins-työn koonnin (engl. build) käyttäen liukuhihnassa määritettyä Kubernetes-kapselia.

2.2.1 Klusteri

Klusteri on keskeinen osa Kuberneteksen perusarkkitehtuurissa. Jos käyttää Kubernetesta, käyttää varmasti myös vähintään yhtä klusteria [7]. Kubernetes-klusteri koostuu joukosta solmuja (engl. node), jotka suorittavat konttisovelluksia. Sovellusten konttipakkaus paketoi sovelluksen riippuvuuksineen ja lisäksi joitakin tarvittavia palveluita. Tällä tavoin Kubernetes-klusterit mahdollistavat sovellusten kehittämisen, siirtämisen ja hallinnan helpommin. Klusterit koostuvat yhdestä pääsolmusta ja useista työsolmuista. Pääsolmun tarkoituksena on ohjata klusterin tilaa. Se koordinoi prosesseja, kuten sovellusten ajoitusta ja skaalausta, klusterin tilan ylläpitämistä ja päivitysten käyttöönottoa. Työntekijäsolmut puolestaan ovat komponentteja, jotka suorittavat pääsolmun määrittämiä tehtäviä. Kubernetes-klusterissa on oltava vähintään yksi pääsolmu ja yksi työntekijäsolmu, jotta se toimii. [8.] Kuvassa 2 näkyy Kubernetes-klusterin komponentit.



Kuva 2. Kubernetes-klusterin komponentteja. Perustuu lähteeseen [9].

Kuten kuvasta 2 näkyy, Kubernetes-klusteri sisältää kuusi pääkomponenttia:

1. API-palvelin: Tarjoaa REST-rajapinnan kaikille Kubernetes-resursseille. Toimii Kubernetes-ohjaustason etupäänä.

2. Aikatauluttaja: Sijoittaa kontteja resurssivaatimusten mukaisesti. Merkitsee muistiin kapselit, joille ei ole määritetty solmua ja valitsee niille solmut, joissa toimia.

3. Ohjaimen hallinta: Suorittaa ohjaimen prosesseja ja sovittaa yhteen klusterin todellisen tilan haluttujen spesifikaatioiden kanssa. Hallitsee ohjaimia, esimerkiksi solmuohjaimia.

4. Kubelet: Varmistaa, että kontit toimivat kapselissa. Ottaa joukon toimitettuja PodSpecs-tietoja ja varmistaa, että niitä vastaavat kontit ovat täysin toimintakunnossa.

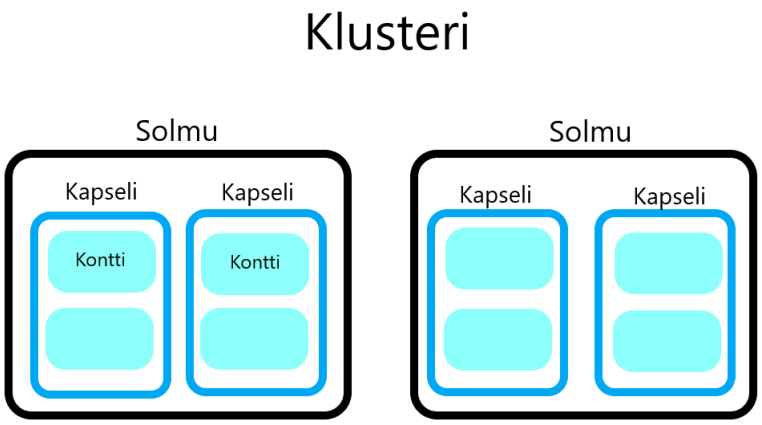
5. Kube-välityspalvelin: Hallitsee verkkoyhteyksiä ja ylläpitää verkkosääntöjä solmujen välillä. Toteuttaa Kubernetes Service -konseptin klusterin jokaisessa solmussa.

6. Muut: Tallentaa kaikki klusteritiedot.

Nämä kuusi komponenttia voivat toimia Linuxissa tai Docker-konttina. Päätsolmu ajaa API-palvelinta, aikatauluttajaa sekä ohjaimen hallintaa. Työsolmut puolestaan ajavat Kubeletia ja Kube-välityspalvelinta. [8.]

2.2.2 Kapselit

Kapselit (engl. pod) ovat pienimpiä käyttöönotettavia laskentayksiköitä, joita voidaan Kuberneteksella luoda ja hallita. Kapseli on ryhmä yhdestä tai useammasta kontista, jossa on jaetut tallennus- ja verkkoresurssit sekä määrittelyt konttien suorittamisesta. Kapselit toimivat solmuissa klusterin sisällä [7]. Kuvassa 3 näkyy kapselien rakenne yksinkertaistettuna klusterin sisällä.



Kuva 3. Yksinkertaistettuna kapselien rakennetta klusterin sisällä. Perustuu lähteeseen [10].

Yksinkertaisimmillaan kapseli on mekanismi, jolla kontti oikeasti kytketään päälle Kuberneteksessa [11]. Kuvassa 4 näkyy yksinkertainen esimerkki kapselista, joka koostuu kontista, jossa pyörii ”nginx-1.14.2”-kuva (engl. image).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Kuva 4. Yksinkertainen esimerkki kapselista [11].

Insinöörityössä käytetään klusteriin määritettyä kapseliä nimeltä “kubernetes-jenkins-build”. Tämä kapseli sisältää Jenkins-agentin Dockerin kanssa sekä rakennustyökalut, kuten maven, nodejs, ant, ansible ja muita. Tämän lisäksi muita

määritettyjä kapseleita oli kolme. Nämä kolme kuitenkin olivat kevyempiä kapseleita, jotka eivät sisältäneet tässä tapauksessa riittäviä rakennustyökaluja töiden suorittamiseen.

2.2.3 Edut

Kubernetes automaattisesti säännöstelee ja sovittaa kontit solmuihin (engl. node) resurssien parhaaseen käyttöön. Kun Kubernetes-klusteri on määritetty, sovellukset voivat toimia minimaalisilla seisokkiajoilla (engl. downtime) ja suoriutua hyvin. Ne vaativat vähemmän tukea, jos solmu tai kapseli epäonnistuu ja se olisi muuten korjattava manuaalisesti. Kuberneteksen konttiorkesterointi (engl. container orchestration) mahdollistaa tehokkaamman työnkulun, jolloin samoja prosesseja ei tarvitse toistaa. Tämä tarkoittaa, että tarvitaan vähemmän palvelimia ja hallintatyötä. [12.]

Kubernetes myös ajoittaa ja automatisoi kontin käyttöönoton useissa laskentasolmuissa (engl. compute nodes), joko pilvessä, paikan päällä olevissa virtuaalikoneissa tai fyysisissä tietokoneissa. Sen automaattinen skaalaus antaa mahdollisuuden skaalata ylös tai alas vastatakseen kysyntään nopeammin. Automaattinen skaalaus käynnistää uusia kontteja tarpeen mukaan raskaille kuormituksille tai piikeille. Piikit saattavat johtua prosessorin käytöstä tai muistikynnyksistä, esimerkkinä verkkotapahtuman käynnistyminen ja pyyntöjen määrän äkillinen lisääntyminen. Kun tarve on ohi, Kubernetes skaalaa resursseja automaattisesti uudelleen. [12.]

Lisäksi Kubernetes auttaa ajamaan konttisovelluksia luotettavasti. Kuten aiemmin on mainittu, se sijoittaa ja tasapainottaa automaattisesti konteissa olevat työkuormat ja skaalaa kapseleita kysynnän mukaisesti. Lisäksi, jos yksi solmu monisolmuklusterissa epäonnistuu, työkuorma jaetaan muille solmuille ilman, että se häiritsee käytettävyyttä. Kubernetes tarjoaa myös itsekorjautumisominaisuuksia ja käynnistää uudelleen, ajoittaa uudelleen tai vaihtaa kontin, kun se epäonnistuu tai kun solmut kuolevat. Tämän avulla voidaan päivittää ohjelmistoa jatkuvasti ilman häiriöitä. [12.]

Näistä Kuberneteksen tuomista eduista varsinkin automaattinen skaalaus on tärkeä tämän insinöörityön kannalta. Tavoitteena oli korvata Jenkins-työsolmut Kuberneteksella, jotta saataisiin automaattinen skaalaus konteissa käyttöön. Tietenkin myös muut mainitut edut ovat hyviä puolia Kubernetekseen siirtymisessä. Varsinkin Kuberneteksen tuomat itsekorjautumisominaisuudet sekä se, että seisokkiajat ovat minimaaliset ja ohjelmistojen päivitys onnistuu sujuvasti ilman häiriöitä, ovat hyviä plussia.

2.3 Ohjelmointikielet

Projektit on toteutettu joko JavaScript- tai TypeScript-ohjelmointikielellä projektin mukaan. Projektit liittyvät aiheeltaan asiakkaan viihdealustasovelluksiin. Joidenkin projektien koodeihin on mahdollisesti tehtävä pieniä muutoksia insinöörityön aikana, mutta näiden ohjelmointikielien käyttö on pienessä osassa tässä insinöörityössä. Insinöörityössä käytetään enemmän Groovy-ohjelmointikieltä, jota käytetään Jenkinsin liukuhihnoissa.

Groovy on olio-ohjelmointikieli, joka perustuu Javaan. Groovy julkaistiin ensimmäisen kerran tammikuun alussa vuonna 2007 [13]. Groovy on staattinen sekä dynaaminen kieli ja sillä on samanlaisia ominaisuuksia kuin esimerkiksi Pythonilla ja Rubylla [14].

3 Ongelman esittely

Tässä luvussa esitellään tiiviisti insinöörityön ongelma. Käsitellään erikseen skaalautuvuusongelma sekä vakioidun pohjan ongelma.

3.1 Skaalautuvuusongelma

Ongelmana on tällä hetkellä käytettyjen Jenkins-työsolmujen skaalautuvuus. Jenkins-työsolmujen määrä on staattinen eli niiden määrää ei voida tarvittaessa helposti säädellä alas tai ylös. Tämän ratkaisemiseksi tavoitteena on korvata Jenkins-työsolmut Kubernetes-klusterilla.

Jenkins-työsolmut ovat koneita, joilla koontiagentit toimivat. Jenkins tarkkailee jokaisen solmun tilaa, esimerkiksi levytilaa ja vapaata tilapäistä tilaa. Solmu siirretään offline-tilaan, jos jokin tarkastelluista arvoista ylittää määritetyn kynnyksen. Jenkins tukee kahden tyyppisiä solmuja: [15.]

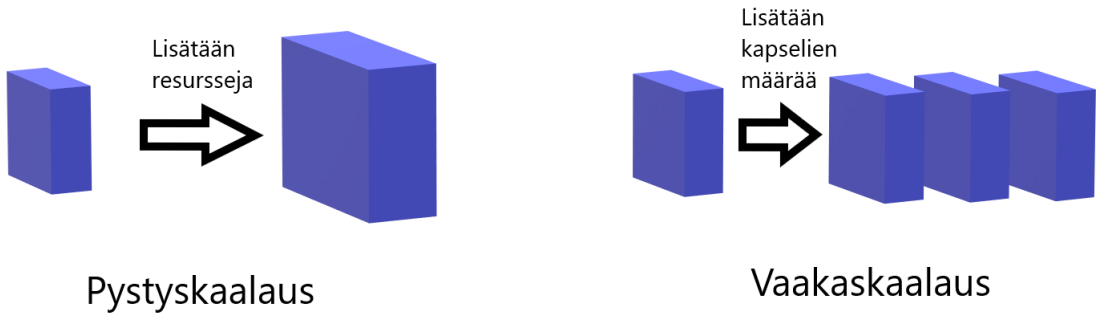
1. Agentit. Ne hallitsevat tehtävien suorittamista Jenkins-ohjaimen puolesta suorittajien (engl. executors) avulla. Agentti on pieni Java-prosessi, joka muodostaa yhteyden Jenkins-ohjaimeen. Agentti voi käyttää mitä tahansa Javaa tukevaa käyttöjärjestelmää. Kaikki koontiin ja testaamiseen tarvittavat työkalut asennetaan solmuun, jossa agentti toimii. Koska työkalut ovat osa solmua, ne voidaan asentaa suoraan tai konttiin, kuten Docker tai Kubernetes. Käytännössä solmut ja agentit ovat olennaisesti samat, mutta ne ovat kuitenkin käsitteellisesti erillisiä. [15.]

2. Sisäänrakennettu solmu. Tämä solmu on ohjainprosessissa oleva solmu. Agenttien lisäksi myös sisäänrakennettua solmua on mahdollista käyttää tehtävien suorittamisessa. Tehtävien suorittamista sisäänrakennetussa solmussa ei kuitenkaan suositella turvallisuus-, suorituskyky- ja skaalautuvuussyistä. Solmulle määritettyjen suorittajien määrä määrittää solmun kyvyn suorittaa tehtäviä. [15.]

Jenkins-agentit voidaan allokoida staattisesti tai dynaamisesti [15]. Tämänhetkiset käytössä olevat Jenkins-työsolmut on allokoitu staattisesti. Ne voidaan kuitenkin allokoida dynaamisesti eri järjestelmien avulla, kuten Kubernetes tässä tapauksessa.

Kuten aikaisemmin mainittiin, automaattisen skaalauksen avulla Kubernetes pystyy skaalaamaan konttien määrää kysynnän mukaisesti. Tämän johdosta kysyntään pystytään vastaamaan nopeammin. Jenkins-solmuja on puolestaan rajoitettu ennalta määritetty määrä, jolloin suuriin kysyntäpiikkeihin ei pystytä niin nopeasti vastaamaan, jos kaikki solmut ovat jo käytössä. Kuberneteksen käytöllä pystytään ehkäisemään tilanteet, jossa kyseisiä tilanteita tapahtuisi, automaattisen skaalauksen ansiosta.

Skaalauksessa Kubernetes käyttää vaakaskaalausta. Kuvassa 5 nähdään vaaka- ja pystyskaalauksen ero.



Kuva 5. Vaaka- ja pystyskaalauksen ero. Perustuu lähteeseen [16].

Kuten kuvasta 5 nähdään, Kuberneteksen tapauksessa vaakaskaalauksella tarkoitetaan sitä, että vastauksena lisääntyneeseen kuormitukseen otetaan käyttöön useampia kapseleita. Tämä eroaa pystyskaalauksesta, jossa puolestaan osoitettaisiin kapseleille lisäresursseja (esimerkiksi muistia tai prosessoritehoa), sen sijaan, että lisättäisiin kapseleiden määrää. [17.]

3.2 Vakioitu pohja

Vakioidulla pohjalla tarkoitetaan liukuhihnaa, jota jokainen projekti voi käyttää kokoamisvaiheissaan. Vakioidun pohjan vaiheet ovat kaikissa projekteissa samat, mutta vaiheet suoritetaan projektikohtaisten parametrien ja muuttujien mukaisesti.

Tällä hetkellä uusien projektien lisääminen on hankalampaa, sillä jokaiselle projektille pitäisi erikseen määritellä omat työt, joita voi olla projektista riippuen 3-5 kappaletta. Jokaisen työn konfiguraatioon pitäisi erikseen määritellä Jenkinsiin muun muassa toiminnot, tunnistetiedot sekä muuttujat, joita toiminnoissa

käytetään. Vakioidun pohjan avulla pystytään uuteen projektiin lisäämään Jenkins-tiedosto, joka käynnistää jokaiselle projektille yhteisen liukuhihnan Jenkins-tiedostossa määritellyillä parametreilla. Myös projektiin liittyvien töiden määrä vähenee, sillä tarvitaan enää vain kaksi työtä: yhteisen liukuhihan suorittamiseen ja julkaiseminen eri ympäristöihin (labra, esituotanto ja tuotanto).

Monihaaraisen liukuhihnan avulla voidaan yhteinen liukuhihna suorittaa mistä tahansa haarasta, jossa vain on Jenkins-tiedosto. Tällöin myös muita kuin viimeisimmäksi koottua (yleensä päähaara) voidaan julkaista testausympäristöihin (labra) tarvittaessa.

Vakioidun pohjan käytöllä saadaan töiden toimintojen määritykset pois Jenkinsistä ja mukaan versiohallintaan. Tällöin määrityksiin voitaisiin helposti tehdä muutoksia projektitasolla muokkaamalla versionhallinnassa olevaa Jenkins-tiedostoa. Näin asiasta kulkeutuu tieto helposti myös muille tiimin jäsenille PR:in muodossa. Itse vakioitu pohja toimii jaetun kirjaston kautta. Pohjan liukuhihna suoritetaan kaikissa projekteissa samalla tavalla, mutta Jenkinsissä olevien konfiguraatiotiedostojen sekä Jenkins-tiedostossa annettujen parametrien mukaisesti. Nämä määritetään projektikohtaisiksi.

4 Toteutus käytännössä

Tässä luvussa käsitellään, miten liukuhihnojen käyttöönotto Kuberneteksella sekä vakioitu pohja on käytännössä toteutettu. Käsitellään myös enemmän jaettuja kirjastoja ja monihaaraisia liukuhihnoja sekä sitä, miten näitä on käytetty toteutuksessa.

4.1 Liukuhihnojen käyttöönotto Kuberneteksella

Lähdin aluksi liikkeelle tutustumalla enemmän Jenkinsiin ja Kubernetekseen ja siihen, miten niitä käytetään yhdessä. Lisäksi perehdyin tarkemmin Jenkinsissä käytettäviin liukuhihnoihin ja niiden eri kirjoitustyyleihin, joita ovat deklaratiivinen skripti (engl. declarative script) ja komentosarjaskripti (engl. scripted script).

Tämän kautta tutustuin enemmän myös Jenkins-tiedostoihin, jotka ovat projektien versionhallinnassa suoritettavia liukuhihnoja.

Tavoitteena oli ensiksi tehdä käyttöönotto yhdelle projektille ilman vakioitua pohjaa. Pääsin aluksi testailemaan Kuberneteksen käyttöönottoa testiprojektin liukuhihnassa. Tässä oli jo valmiiksi otettu käyttöön Kubernetes-laajennus sekä agentiksi Kubernetes-kapseli, joiden avulla töiden (engl. job) koontiversiot (engl. build) rakennetaan. Käytettävä agentti määritellään liukuhihnan agent-lohkossa. Esimerkkikoodissa 2 on esimerkki agent-lohkosta.

```
agent {
     label "name-of-agent"
}
```

Esimerkkikoodi 2. Esimerkki liukuhihnan agent-lohkosta.

Esimerkkikoodista 2 nähdään agent-lohko. Esimerkissä sen sisällä on vain label-kohta, jossa määritellään käytettävän agentin nimi. Insinöörityössä tarkoituksena oli siirtyä käyttämään Kubernetes-klusteria, jolloin agentin nimenä voi esimerkiksi olla sen kapseli ”kubernetes-jenkins-build”.

Lähdin aluksi muuttamaan alkuperäisestä komentosarjaskriptistä liukuhihnan deklaratiiviseksi skriptiksi. Deklaratiivisen ja komentosarjaskriptin erot ovat pienet mutta huomattavat:

1. Syntaksi: Komentosarjaskripti perustuu Groovy-skriptikieleen, kun taas deklaratiivinen skripti perustuu YAML-syntaksiin [18].

2. Joustavuus: Komentosarjaskripti tarjoaa enemmän joustavuutta ja hallintaa liukuhihnan työnkulkuun, kun taas deklaratiivinen skripti tarjoaa yksinkertaisemman ja jäsennellymmän syntaksin [18].

3. Virheiden käsittely: Komentosarjaskripti mahdollistaa tarkemman virheidenkäsittely, kun taas deklaratiivinen skripti tarjoaa yksinkertaisemman virheenkäsittelyn, joka on helpompi ymmärtää [18].

4. Koodin uudelleenkäyttö: Komentosarja skripti mahdollistaa enemmän koodin uudelleenkäyttöä, kun taas deklaratiivinen on suunniteltu itsenäisemmäksi ja vähemmän riippuvaiseksi ulkoisista skripteistä ja kirjastoista [18].

5. Luettavuus: Komentosarjaskripti on monimutkaisempaa, kun taas deklaratiivinen skripti on suunniteltu luettavammaksi ja helpommaksi ymmärtää [18].

Näiden asioiden perusteella valitsin deklaratiivisen skriptin käytettäväksi. Esimerkkikoodissa 3 näkyy vielä molempien skriptien syntaksieroja.

```
***Deklaratiivinen skripti
pipeline {
     agent any
     stages {
          stage('Build') {
               steps {

               }
          }
          stage('Test) {
               steps {

               }
          }
          stage('Deploy') {
               steps {

               }
          }
     }
}

***Komentosarjaskripti
node {
     stage('Build') {

     }
     stage('Test') {

     }
     stage('Deploy') {

     }
}
```

Esimerkkikoodi 3. Esimerkit deklaratiivisen ja komentosarjaskriptin syntaksieroista [19].

Kuten esimerkkikoodissa 3 nähdään, syntaksierot ovat vähäiset. Deklaratiivisessa skriptissä aloitetaan pipeline-lohkolla ja komentosarjaskriptissä node-lohkolla. Komentosarjaskriptissä ei myöskään tarvita stage-lohkojen sisäisiä steps-lohkoja.

Siirryin seuraavaksi tutkimaan konfiguraatiotiedostosta lukemista liukuhihnaan. Tarkoituksena oli, että projektikohtaisia arvoja voitaisiin lukea Jenkinsiin määritetystä konfiguraatiotiedostosta. Luettava tiedosto määriteltäisiin projektin Jenkins-tiedostossa liukuhihnakutsun parametrinä. Tutkittuani hieman päätin käyttää ominaisuudet-tiedostosta lukemista. Tein tiedoston lukemiseen liittyen kaksi skriptiä, josta ensimmäinen lukisi halutun tiedoston kaikki arvot muuttujaan. Toisessa skriptissä luettaisiin muuttujasta haluttu arvo käytettäväksi. Tällöin konfiguraatiotiedosto joudutaan käymään vain kerran läpi eikä jokaisen arvon kohdalla uudestaan. Esimerkkikoodissa 4 näkyy konfiguraatiotiedoston lukeminen muuttujaan ja esimerkkikoodissa 5 näkyy halutun arvon lukeminen muuttujasta.

```
***getConfigVars.groovy
def call(String fileName) {
    configFileProvider([configFile(fileId: fileName, variable: 'file-
Content')]) {
        def props = readProperties file: "$fileContent"
        if(!props) {
            error "Provided config file was NOT found"
        }
        return props
    }
}
```

Esimerkkikoodi 4. Koodi konfiguraatiotiedostosta lukemiseen.

Esimerkkikoodissa 4 haluttu tiedosto luetaan koodissa configFileProvider-apumuuttujan avulla, joka on Jenkins-laajennus. Parametrinä syötetään tiedoston nimi, joka halutaan lukea. Funktio palauttaa tiedoston sisällön, jos tiedosto ja sen sisältö löytyy. Jos haluttua tiedostoa ei löytynyt, antaa funktio virheen, ja työn koonti keskeytyy.

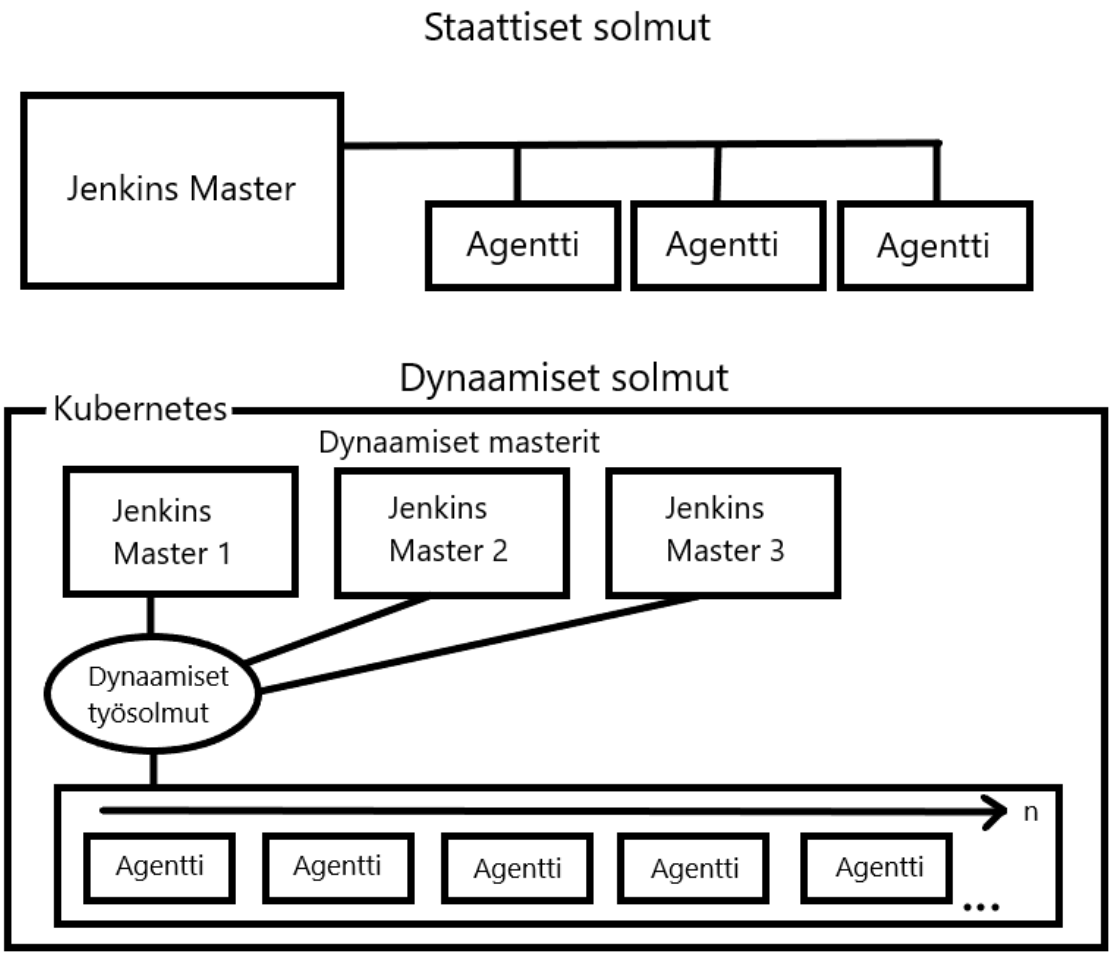
```
***getProp.groovy
def call(Map props, String key) {
    def value = props[key]
    if(value) {
        echo "Provided value was found in config file"
        return value
    } else {
        error "Provided value was NOT found in config file"
    }
}
```

Esimerkkikoodi 5. Koodi konfiguraatiotiedoston halutun arvon lukemisesta.

Esimerkkikoodissa 5 haettaessa tiettyä arvoa konfiguraatiotiedostosta, syötetään funktiolle muuttuja, joka saatiin getConfigVars-funktiosta palautettua, sekä arvo, jota halutaan konfiguraatiotiedostosta etsiä. Funktio palauttaa konfiguraatiotiedoston arvon oikealta kohdalta, jos se haetulla arvolla löytyy. Jos arvoa ei löydy, antaa funktio virheen ja keskeyttää koonnin.

Joissain skripteissä oli tarpeen käyttää myös tunnistetietoja (engl. credentials). Lukuisat kolmannen osapuolen sivustot ja sovellukset voivat olla vuorovaikutuksessa Jenkinsin kanssa. Tällaisen sivuston tai sovelluksen järjestelmävalvoja voi määrittää tunnistetiedot Jenkinsin käyttöön. Tämä tehdään Jenkinsin käytettävissä olevien sovellusten toimintojen ”lukitsemiseksi” yleensä soveltamalla pääsynhallintaa kyseisiin tunnistetietoihin. Kun tunnistetiedot on määritetty, liukuhihnat voivat käyttää tunnistetietoja vuorovaikutuksessa kolmannen osapuolen sovellusten kanssa. Tunnistetietoja voidaan määrittää Jenkinsiin joko globaalille tasolle, kansio/liukuhihna- tai käyttäjätasolle. [20.] Tässä insinöörityössä käytetyt tunnistetiedot määriteltiin kansiotasolle. Tunnistetiedot määriteltiin viihdealustakansioon, jonka alla tarvittavat projektit sijaitsevat. Näin saadaan kaikille tarvittaville viihdealustaprojekteille samat tunnistetiedot käytettäväksi liukuhihnassa.

Kubernetesta käyttämällä saadaan siis korvattua staattiset Jenkins-työsolmut automaattisesti skaalautuvilla Kubernetes-kapseleilla. Kuvasta 6 nähdään erot staattisten ja dynaamisten solmujen käytön välillä.



Kuva 6. Erot staattisten ja dynaamisten solmujen välillä. Perustuu lähteeseen [21].

Kuten kuvasta 6 huomataan, dynaamisissa solmuissa agenttien määrä kasvaa siihen asti, mihin on tarve. Staattisissa solmuissa agentteja on rajoitettu määrä.

4.2 Vakioitu pohja

Vakioitu pohja toteutettiin käyttämällä jaettua kirjastoa. Tässä luvussa esitellään tarkemmin jaettuja kirjastoja ja niiden ominaisuuksia sekä vakioidun pohjan toteutusta ja sen koodia.

4.2.1 Jaetut kirjastot

Jaetulla kirjastolla (engl. shared library) Jenkinsissä tarkoitetaan kokoelmaa Groovy-skriptejä, jotka jaetaan Jenkinsin eri töiden kesken. Skriptien suorittamista varten ne laitetaan Jenkins-tiedostoon. Jaettuja kirjastoja käytetään, jotta voitaisiin käyttää jo kirjoitettua koodia uudelleen uusien projektien kanssa niiden Jenkins-tiedostoissa. Tämä vähentää koodaukseen käytettyä aikaa ja auttaa välttämään koodin päällekkäisyyttä. Jaetun kirjaston luominen yksinkertaistaa myös projektin lähdekoodipäivitysten tekemistä. Kirjaston lähdekoodin päivittäminen päivittää myös jokaisen kirjastoa käyttävän projektin koodin. [22.]

Jaetut kirjastot koostuvat usein kolmesta kansiosta, joita ovat src, vars ja resources. Kuvassa 7 on esimerkki jaettujen kirjastojen dokumentaatiosta.

```
(root)
+- src                     # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

Kuva 7. Esimerkki jaettujen kirjastojen kansioista ja sisällöistä [23].

Src-hakemiston tulisi näyttää tavalliselta Java-lähdehakemistorakenteelta. Tämä hakemisto lisätään luokkapolkuun (engl. classpath) suoritettaessa liukuhihnoja. [23.]

Vars-hakemisto sisältää komentosarjatiedostoja (engl. script files), jotka näkyvät muuttujina liukuhihnassa. Tiedoston nimi on liukuhihnamuuttujan nimi. Jos

olemassa on tiedosto nimeltä info.groovy, jossa on funktio def tervehdys(viesti)…, voidaan käyttää tätä funktiota kutsumalla info.tervehdys ”terve” liukuhihnassa. Tähän tiedostoon voidaan lisätä niin monta funktiota kuin halutaan. Lisäksi jos tiedostoon sisällytetään def call()-funktio, suoritetaan se automaattisesti, jos tiedostoa kutsutaan liukuhihnassa näin: info(). [23.]

Resources-hakemisto mahdollistaa LibraryResource-vaiheen käyttämisen ulkoisesta kirjastosta siihen liittyvien ei-Groovy-tiedostojen lataamisen. Tällä hetkellä tätä ominaisuutta ei kuitenkaan tueta sisäisissä kirjastoissa. [23.]

Jaettuja kirjastoja on kahdenlaisia: globaaleja sekä kansiotason. Jokaiseen Jenkinsissä olevaan kansioon voidaan liittää jaettu kirjasto. Tämä mahdollistaa sen, että tiettyjä kirjastoja voidaan käyttää kaikissa kansion tai alikansion sisällä olevissa liukuhihnoissa. Kansiopohjaisia kirjastoja ei pidetä ”luotettavina”, sillä ne toimivat Groovy-hiekkalaatikossa (engl. sandbox) aivan kuten tyypillisetkin liukuhihnat. [23.]

Globaaleissa kirjastoissa puolestaan mikä tahansa järjestelmän liukuhihna voi hyödyntää kirjastossa toteutettua toimintoa. Globaaleita kirjastoja pidetään ”luotettuina”, sillä ne voivat suorittaa mitä tahansa metodeita Java-, Groovy- tai kolmannen osapuolen kirjastoissa, Jenkinsin sisäisissä API-liittymissä sekä Jenkins-laajennuksissa. Globaaleiden kirjastojen kanssa pitää kuitenkin olla varovainen, sillä kuka tahansa, joka pystyy tekemään muutoksia projektihallintaan, voi saada rajoittamattoman pääsyn Jenkinsiin. Globaalien kirjastojen konfigurointiin tarvitaan ”Overall/RunScripts”-oikeudet, jotka myönnetään yleensä vain Jenkinsin järjestelmävalvojille. [22.] Päädyin itse käyttämään käyttöoikeuksien vuoksi kansiotason kirjastoja. Tämä ei kuitenkaan haitannut, sillä jaetun kirjaston pystyi määrittelemään suoraan pääkansioon, jolloin kaikki sen sisällä olevat työt ja projektit pystyvät käyttämään määritettyä jaettua kirjastoa.

Jaettu kirjasto otetaan käyttöön @Library-annotaatiolla sen jälkeen, kun se on määritelty Jenkinsin konfiguraatioon. Esimerkkikoodissa 6 on esimerkki siitä, miten jaettua kirjastoa käytetään Jenkins-tiedostossa. [22.]

```
@Library('pipeline-library-demo')_

stage('Demo') {
     sayHello 'Alex'
}
```

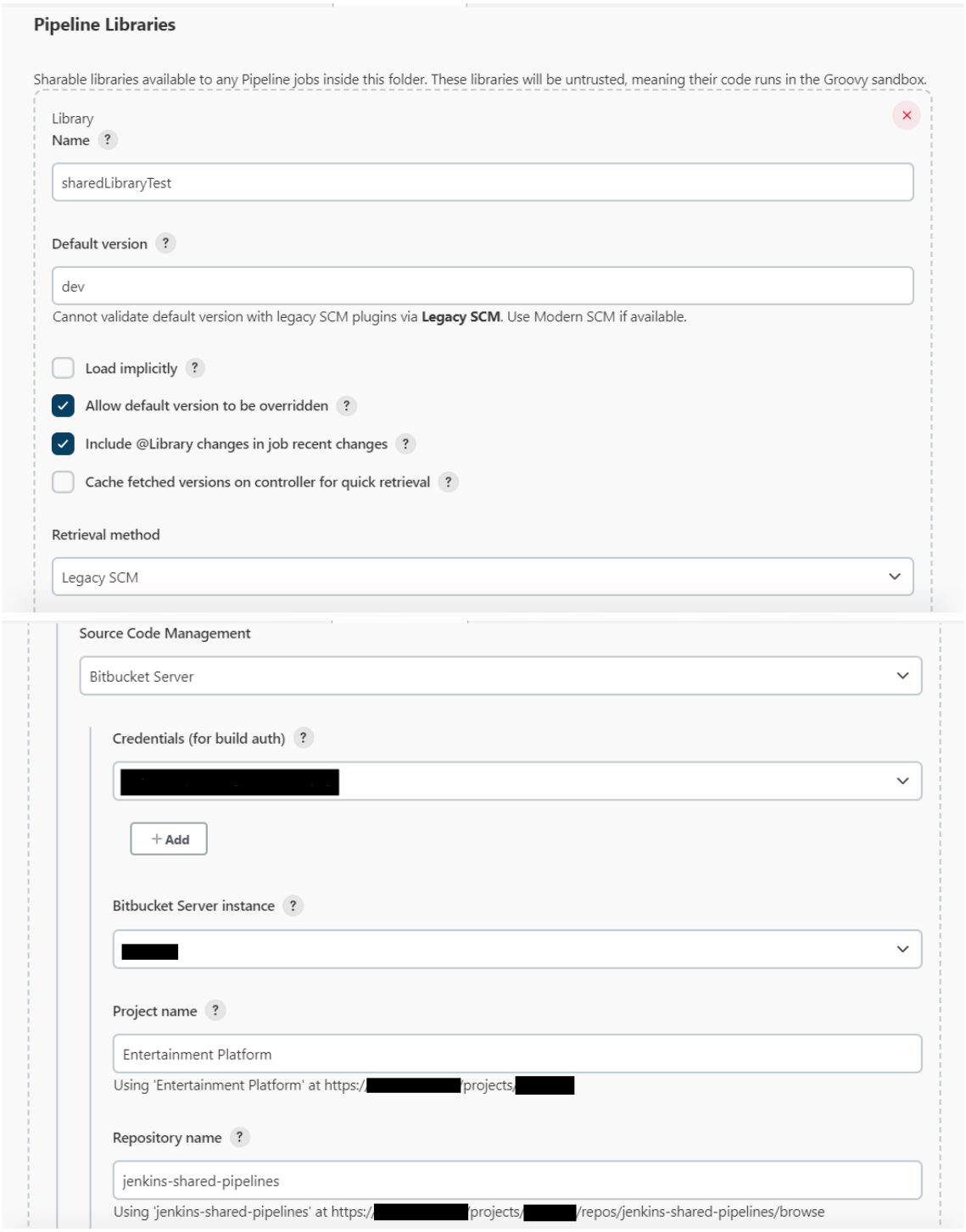
Esimerkkikoodi 6. Esimerkki jaetun kirjaston annotaatiosta ja käytöstä liukuhihnaskriptissä [22].

Esimerkkikoodissa 6 nähdään @Library-annotaatio ja parametrina jaetun kirjaston nimi. Stage-lohkossa kutsutaan sayHello 'Alex', joka tarkoittaa, että jaetusta kirjastosta kutsutaan sayHello.groovy-tiedoston call-metodia parametrilla "Alex".

4.2.2 Toteutus

Vakioidun pohjan toteutuksessa käytin jaettua kirjastoa. Alkuperäisenä tavoitteena oli käyttää oletus Jenkins-tiedostoa. Oletus Jenkins-tiedosto määriteltäisiin Jenkinsin globaalille tasolle tiedostona, joka voitaisiin projektissa määritellä suoritettavaksi. Oikeudet Jenkinsissä eivät kuitenkaan riittäneet, että olisimme saaneet globaalille tasolle muutoksia.

Päätimme tehdä jaetulle kirjastolle oman projektin, jota muut projektit voivat käyttää. Jaettu kirjasto määriteltiin Jenkinsin GUI:lla osoittamaan oikeaan projektiin. Kuvassa 8 on esimerkki jaetun kirjaston määrityksistä.



Kuva 8. Jaetun kirjaston määritys Jenkins GUI:lla.

Jaettua kirjastoa määriteltäessä on muutama kohta, joita voidaan muuttaa, kuten kuvasta 8 nähdään. Näistä tärkeimpänä tämän insinöörityön kannalta on projektin määritys, jota jaettu kirjasto käyttää. Konfiguraation alussa jaetulle kirjastolle annetaan nimi, jota käytetään liukuhihnan alussa @Library-annotaation

parametrina. Default version -arvoksi annetaan haara, jota jaetussa kirjastossa halutaan käyttää. Lopuksi määritellään projekti haettavaksi halutusta versionhallintajärjestelmästä. Bitbucketin tapauksessa syötetään käytettävät tunnistetiedot sekä halutun projektin nimi.

Kun jaetulle kirjastolle oli luotu oma projekti ja määritykset tehty, jokaiselle projektille luotiin versionhallintaan oma Jenkins-tiedosto, jossa kutsutaan jaetussa kirjastossa olevaa tiedostoa, joka suorittaa liukuhihnan. Suoritettava tiedosto on pohja kaikille projektien liukuhihnoille, joka suoritetaan projektikohtaisesti parametrien mukaisesti. Tällä tavoin saadaan itse projektiin tuleva Jenkins-tiedosto mahdollisimman lyhyeksi ja yksinkertaiseksi. Lisäksi se vaatii tulevaisuudessa vähemmän muutoksia. Esimerkkikoodissa 7 näkyy esimerkki eräässä projektissa olevasta Jenkins-tiedostosta.

```
***Jenkinsfile
@Library("sharedLibraryTest") _

startCommonPipeline(
    build:[[project:"drm", name:"user", path:"api", image:"drm-api-
user"],
           [project:"drm", name:"encode", path:"api", image:"drm-api-
encode"]],
    environments:["lab", "pre", "pro", "aws"]
)
```

Esimerkkikoodi 7. Esimerkki Jenkins-tiedosto eräästä projektista.

Esimerkkikoodissa 7 tiedoston ensimmäisellä rivillä otetaan jaettu kirjasto käyttöön. Annotaation parametriksi annetaan jaetun kirjaston nimi, joka määritellään Jenkinsin konfiguraatiossa. Kolmannelta riviltä alkava kutsu suorittaa jaetusta kirjastosta startCommonPipeline.groovy-nimisen (vakioitu pohja) tiedoston call()-metodin. Kutsun parametrina on arvoja, joita käytetään Docker-kuvan rakentamisessa sekä Teams-ilmoituksessa.

Itse vakioituun pohjaan on environment-lohkoon sekä parameters-lohkoon määritelty arvot, joita käytetään samoin kaikissa projekteissa. Lisäksi parameters-lohkoon määritellyt arvot ovat muutettavissa, kun työ halutaan koota (engl. build) Jenkinsin GUI:n kautta. Käydään läpi vakioitu pohja kohta kohdalta.

```
***startCommonPipeline.groovy
def call(Map jenkinsConfig = [:]) {
    currentBuild.displayName = ”#$BUILD_NUMBER $GIT_NAME”
    pipeline {
        agent {
            label 'kubernetes-jenkins-build'
        }
```

Esimerkkikoodi 8. Vakioidun pohjan kutsu ja agentti.

Esimerkkikoodissa 8 annetaan parametrina Map-tyyppinen muuttuja. Mapin avulla voidaan kutsussa ”nimetä” parametreja, jotta annettujen parametrien määrittäminen olisi helpompaa. Esimerkkinä on esimerkkikoodi 7, jossa on annettu startCommonPipeline-kutsulle parametreiksi ”build” ja ”environments”. currentBuild.displayNamen avulla määritellään Jenkins GUI:ssa näkyvä koonnin nimi. Ilman erillistä määritystä koonnin nimeksi tulee vain koonnin numero (BUILD_NUMBER koodissa). Agent-lohkossa määritetään liukuhihnan suorittamiseen käytettävä työsolmu tai tässä tapauksessa kapseli. Alussa käytin ”kubernetes-jenkins”-kapselia, joka on kevyt, mutta tarkoitettu vain yksinkertaisiin tehtäviin ja yksinkertaisten shell-skriptien suorittamiseen. Tämä kapseli ei sisältänyt esimerkiksi curl-komentoa, joten siirryimme käyttämään hieman raskaampaa ”kubernetes-jenkins-build”-kapselia, joka sisältää Jenkins-agentin Dockerin kanssa sekä rakennustyökalut, kuten maven, nodejs, ant, ansible ja muita.

```
***startCommonPipeline.groovy
environment {
             TZ='Europe/Helsinki' // Timezone

             // Project
             repositoryName = sh(script: “basename $GIT_URL .git”, re-
turnStdout: true).trim()
             allVars = getConfigVars(repositoryName+'-config')
             projectName = getProp(allvars, 'projectName’)
             schemaScript = getProp(allvars, 'schemaScript’)
             testScript = getProp(allvars, 'testScript’)
             installScript = getProp(allvars, 'installScript’)
             summary = getProp(allVars, 'summary')
             themeColor = getProp (allVars, 'themeColor')
             displayName = getProp (allVars, 'displayName')
             actionURI = getProp (allVars, 'actionURI')
             actionlab = getProp (allVars, 'actionlab')
             actionpre = getProp (allVars, 'actionpre')
             actionpro = getProp (allVars, 'actionpro')
             actionaws = getProp (allVars, 'actionaws')
)

             // Credentials
             TEAMS_URL = credentials('TeamsWebhookSamlinkJenkins')
             ARTIFACTORY = credentials('samlink-artifact')
}
```

Esimerkkikoodi 9. Vakioidun pohjan environment-muuttujat.

Esimerkkikoodissa 9 ovat liukuhihnan environment-muuttujat. Tässä määritettyjä muuttujia tarvitaan jokaisessa projektissa. Ensimmäisenä määritetty TZ-muuttuja määrittelee liukuhihnalle aikavyöhykkeen. Vyöhyke täytyi määrittää erikseen, sillä osassa projekteissa testit eivät menneet läpi, koska aikavyöhyke oli kapselissa väärä. Project-osion muuttujat luetaan konfiguraatiotiedostoista. Projektikohtaisten muuttujien arvot vaihtelevat sen mukaan, mikä projekti on kyseessä tai minkä projektin konfiguraatiotiedosto luetaan. Tarvittavat muuttujat ovat kuitenkin jokaisessa projektissa samat. Lopussa vielä liukuhihnassa käytettävät tunnistetiedot. Tunnistetieto luetaan muuttujaan käyttämällä tunnistetiedon ID-arvoa. Tunnistetiedot määritellään Jenkinsin GUI:ssa. Environment-lohkon muuttujia voidaan käyttää jokaisessa liukuhihnan vaiheessa haluttaessa.

```
***startCommonPipeline.groovy
parameters {
                string(name: 'GIT_REF', defaultValue: '**', descrip-
    tion: 'Branch reference to checkout')
                string(name: 'GIT_NAME', defaultValue: '**', descrip-
    tion: '')
                choice(name: 'NODE_VERSION', choices: ['14.17.0',
    '12.20.2', '10.8.0', '8.11.3', '7.3.0', '6.9.2'], description:
    'NodeJs version')
}
```

Esimerkkikoodi 10. Vakioidun pohjan parametrimuuttujat.

Esimerkkikoodin 10 parameters-lohkossa on työn käynnistämisessä määriteltävät parametrit. Parametreihin voidaan antaa erityyppisiä muuttujia, kuten string, text, booleanParam, choice tai password. Näitä parametreja voidaan käyttää jokaisessa liukuhihnan vaiheessa haluttaessa.

```
***startCommonPipeline.groovy
stages {
            stage('Checkout') {
                steps {
                    checkoutCommon()
                }
            }

            stage('Install') {
                steps {
                    runInstall()
                }
            }
            stage('Schemas') {
                steps {
                    compileSchemas()
                }
            }
            stage('Tests') {
                steps {
                    runTests()
                }
            }
            stage('Build Image') {
                steps {
                    buildDockerImage(jenkinsConfig.build)
                }
            }
}
```

Esimerkkikoodi 11. Vakioidun pohjan vaiheet.

Esimerkkikoodin 11 stages-lohkossa määritellään vaiheet, jotka liukuhihnan suorituksessa käydään läpi kohta kohdalta. Jokaisessa vaiheessa voidaan suorittaa halutut asiat ja nimetä tasot sen mukaan. Tässä tapauksessa tasoissa kutsutaan jaetun kirjaston tiedostonimeä kuten runInstall() tai compileSchemas(). Kutsu suorittaa näissä tiedostoissa olevan call()-metodin.

```
***startCommonPipeline.groovy
post {
            success {
                sendTeamsDeployMessage(jenkinsConfig.environments)
            }
        }
```

Esimerkkikoodi 12. Vakioidun pohjan post-lohko.

Jos esimerkkikoodissa 11 näkyvät tasot menivät läpi onnistuneesti, suorittaa esimerkkikoodin 12 post-lohkon success-lohko annetun tehtävän. Tässä tapauksessa kutsuu sendTeamsDeployMessagea, joka lähettää ilmoituksen Teams-kanavalle uudesta koontiversiosta. Sen parametriksi annetaan projektilla mahdolliset julkaisuympäristöt, jotka määritetään projektin Jenkins-tiedostossa.

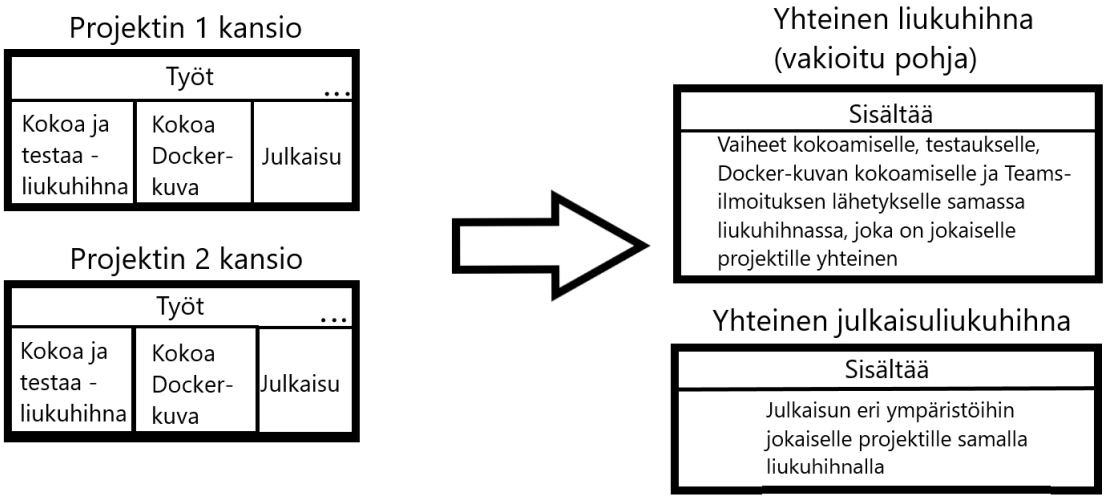
Vakioidussa pohjassa käytetään jaetun kirjaston tekniikoita kutsumalla tiedostonimeä (esim. runInstall()). Kutsuttavat tiedostot sisältävät call-metodin, joka suoritetaan oletusarvoisesti tiedostoa kutsuttaessa. Esimerkkikoodissa 13 esimerkiksi runInstall.groovy-tiedosto.

```
***runInstall.groovy
def call() {
    nodejs("node.js ${params.NODE_VERSION}") {
        sh '''
        npm --version
        node --version
        $installScript
        '''
    }
}
```

Esimerkkikoodi 13. runInstall.groovy-tiedoston koodi.

Kun vakioidussa pohjassa (startCommonPipeline.groovy) kutsutaan runInstall(), se kutsuu runInstall.groovy-tiedoston call()-metodia, joka näkyy

esimerkkikoodissa 13. Tässä tapauksessa funktio suorittaa nodejs-lohkon, jonka parametriksi annetaan työn käynnistyksessä määritelty node-versio. Lohkon sisällä suoritetaan shell-skripti, joka tarkistaa npm- sekä node-versiot ja suorittaa environment-muuttujista installScriptin. Jakamalla liukuhihna pienempiin osiin useampaa tiedostoa käyttämällä saadaan itse liukuhihna selvempään muotoon ja helpommin ymmärrettäväksi. Kuvassa 9 nähdään vielä muutos nykyisen ratkaisun ja vakioidun pohjan käytön välillä.



Kuva 9. Muutos nykyisen ratkaisun ja vakioidun pohjan käytön välillä.

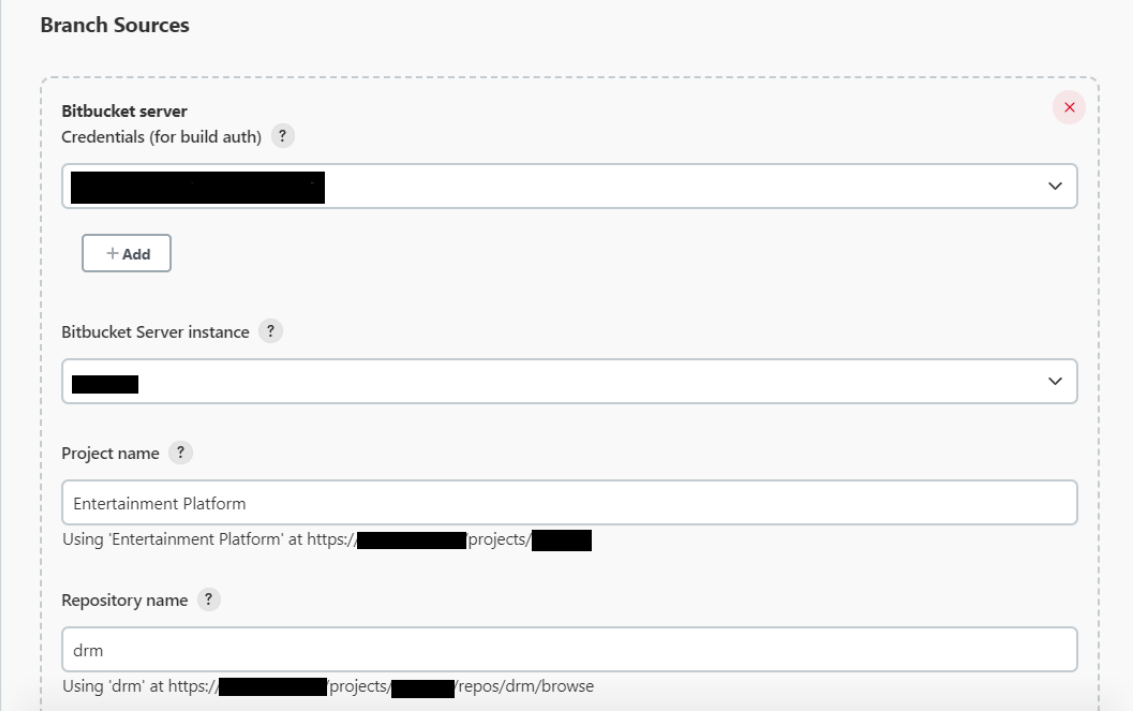
Kuten kuvasta 9 nähdään, nykyisellä ratkaisulla tarvitaan jokaiselle projektille omat työnsä Jenkinsin projektikansion sisällä. Vakioidun pohjan avulla saatiin useampi työ saamaan liukuhihnaan ja kaikille projekteille yhteiseksi. Lisäksi julkaisuliukuhihna saatiin yhteiseksi kaikille projekteille, josta on lisää luvussa 4.4.

4.3 Monihaaraiset liukuhihnat

Monihaaraisilla liukuhihnoilla (engl. multibranch pipeline) tarkoitetaan konseptia, jolla luodaan automaattisesti Jenkins-liukuhihnoja Git-haaroihin perustuen. Se voi löytää automaattisesti uusia haaroja, jos niitä tulee, ja luoda automaattisesti liukuhihnan kyseiselle haaralle. Kun liukuhihnan rakentaminen alkaa, Jenkins käyttää kyseisen haaran Jenkins-tiedostoa kokoamisvaiheisiin. Valitut haarat

voidaan myös jättää pois, jos tiettyjen haarojen ei haluta olevan automaattisesti liukuhihnassa. Tämä toteutetaan Jenkinsin GUI:n kautta monihaaraisen liukuhihnan konfiguraatiossa. [24.]

Monihaaraisten liukuhihnojen käyttöönotto oli melko helppoa. Jenkinsin GUI:n kautta luominen onnistui yksinkertaisesti ja konfiguraatioon löytyi hyviä ohjeita. Käydään vielä konfiguraatiosta läpi pari tärkeää kohtaa insinöörityöhön liittyen. Kuvassa 10 nähdään monihaaraisen liukuhihnan projektin määrittely.

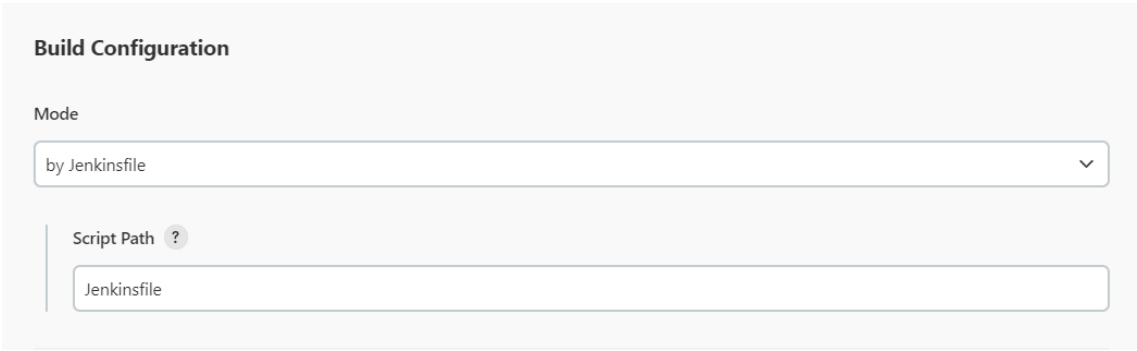


Kuva 10. Monihaaraisen liukuhihnan projektin määrittely.

Kuvassa 10 näkyy konfiguraation projektin määrittely. Konfiguraatiosta määritellään, mitä projektia kyseinen monihaarainen liukuhihna käyttää. Sen määrittelyssä syötetään tarvittavat tunnistetiedot versionhallintaan sekä projektin nimi. Kuvassa 11 nähdään kohta konfiguraatiosta, jossa määritellään mukana olevat haarat.

Kuva 11. Mukana olevien haarojen määritys monihaaraisen liukuhihnan konfiguraatiossa.

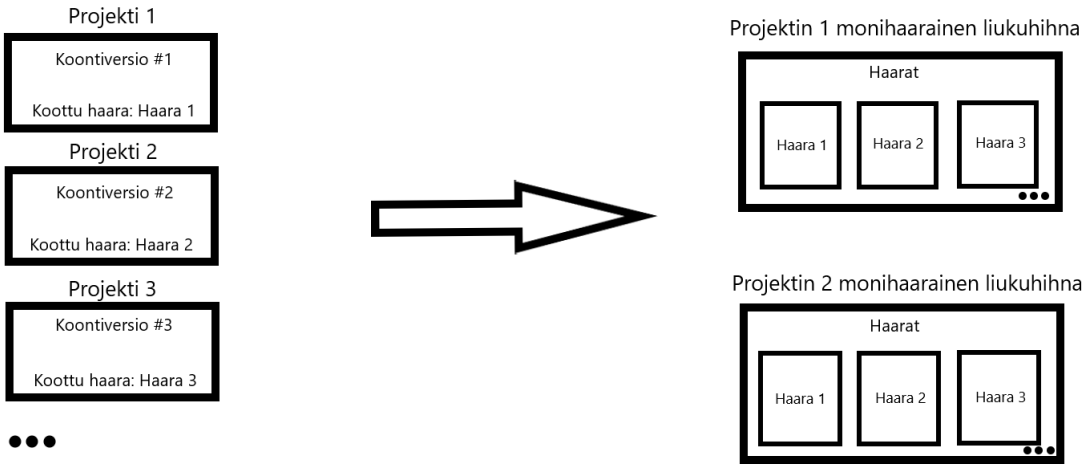
Monihaaraisen liukuhihnan konfiguraatioon voidaan määritellä ”Filter by name (with wildcards)” -kohta, jonka avulla voidaan erikseen määritellä, mitkä haarat tulevat mukaan monihaaraiseen liukuhihnaan. Vaihtoehtoisesti voidaan sulkea tiettyjä haaroja ulos. Kuvasta 11 nähdään, että kyseiseen monihaaraiseen liukuhihnaan on luettu vain haara, jonka nimi on ”merge44master”. Kuvassa 12 on Jenkins-tiedoston määritys.



Kuva 12. Monihaaraisen liukuhihnan Jenkins-tiedoston määritys.

Kuvassa 12 on kohta konfiguraatiosta, jossa määritellään se, minkä tiedoston perusteella työn suorittaminen tehdään. Kuvassa 12 on valittu vaihtoehdoksi Jenkins-tiedosto. Skriptin polkuna on oletusarvoisesti ”Jenkinsfile”. Tämä tarkoittaa, että Jenkins etsii projektin juuresta ”Jenkinsfile”-nimistä tiedostoa, jonka suorittaa työn käynnistyessä.

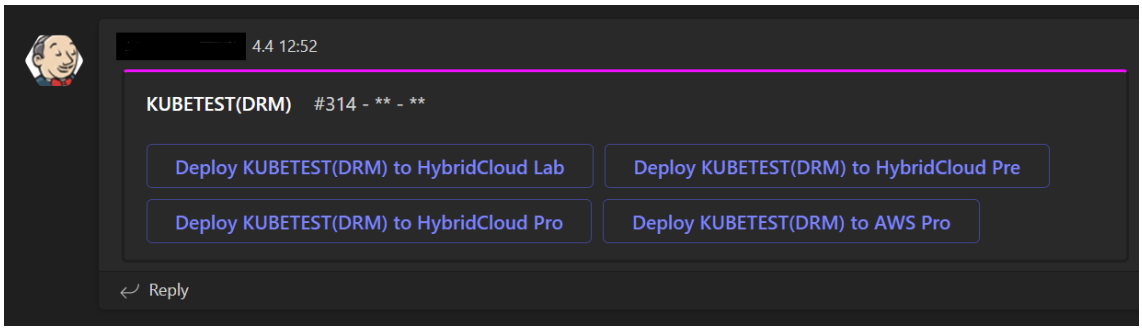
Tällä hetkellä käytössä olevat liukuhihnat suorittavat projektin koonnin parametreissa määritetylle haaralle. Monihaaraisten liukuhihnojen avulla voidaan koota mikä tahansa haaroista ja jokaisella niistä on oma näkymä Jenkins GUI:ssa. Kuvassa 13 nähdään erot nykyisen ratkaisun ja monihaaraisten liukuhihnojen käytön välillä.



Kuva 13. Liukuhihnat tällä hetkellä verrattuna monihaaraisten liukuhihnojen käytön kanssa.

4.4 Julkaisu

Yhdeksi ongelmaksi insinöörityön aikana ilmeni projektiversion julkaisu eri ympäristöihin, kuten labraan, esituotantoon ja tuotantoon. Tällä hetkellä julkaisu tapahtuu erillisestä työstä Jenkinsissä. Jokaiselle projektille on omat julkaisutyönsä, jotka käynnistämällä ajetaan versio haluttuun ympäristöön. Haluttu ympäristö valitaan Jenkinsin GUI:sta työtä käynnistettäessä. Jokaiselle projektille on myös omat työnsä julkaisua varten. Julkaisutyöt voidaan myös käynnistää Teams-viestistä. Onnistuneen koontiversion tuloksena tulee Teams-kanavalle ilmoitus, jossa on vaihtoehdot eri ympäristöihin julkaisusta. Kuvassa 14 on esimerkki erään projektin ilmoituksesta.

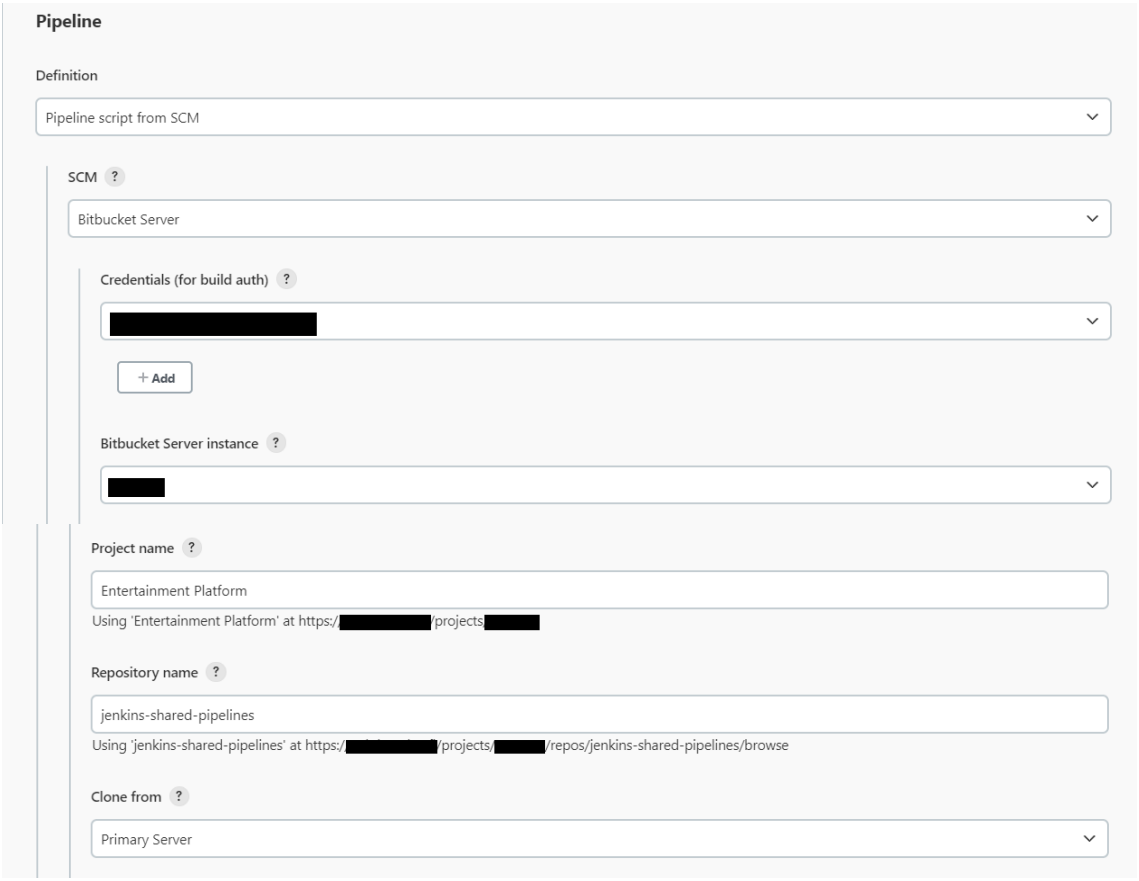


Kuva 14. Teams-ilmoitus eräästä projektista.

Kuvasta 14 nähdään, että ilmoituksessa kerrotaan koontiversion numero, projektin nimi sekä haara (testi-ilmoituksessa **). Jokaiselle mahdolliselle julkaisuympäristölle on myös oma painike, josta avautuu ikkuna Jenkinsiin ja työn käynnistämiseen. Teams-viestien linkit on täydennetty parametreilla, jotka automaattisesti täytetään Jenkinsin GUI:hin työtä käynnistettäessä.

Mietin tähän erilaista ratkaisua, jossa julkaisu olisi voitu yhdistää projektien yhteiseen liukuhihnaan (vakioitu pohja). Harkitsin esimerkiksi Jenkinsin input-kenttää, joka pyytäisi käyttäjältä suostumuksen julkaisuun, jos työ olisi ajettu onnistuneesti. Tämä ratkaisu ei kuitenkaan tyydyttänyt, sillä työn eteneminen jäisi tauolle odottaen vastausta. Lisäksi koontiversion julkaisu useampaan ympäristöön ei olisi onnistunut ilman uuden koontiversion luomista jokaiselle ympäristölle. Etsinnästä huolimatta ei Jenkinsistä löytynyt tähän sopivaa työkalua, joka toimisi halutusti liukuhihnojen kanssa.

Tämän vuoksi päätin pysyä kutakuinkin nykyisessä ratkaisussa. Muutin julkaisutyön universaaliksi, jotta kaikki projektit voisivat käyttää julkaisuissaan samaa työtä. Kuvassa 15 on julkaisutyön konfiguraatiota.



Kuva 15. Projektin määritys osa julkaisuliukuhihnan konfiguraatiosta Jenkinsin GUI:ssa.

Kuvassa 15 nähdään julkaisutyön määritystä konfiguraatiosta. Definition-kohtaan valitaan ”Pipeline script from SCM”, joka tarkoittaa, että suoritettava liukuhihna otetaan versionhallinnasta. Tämän jälkeen syötetään tiedot projektista, jota työssä halutaan käyttää. Tässä tapauksessa versionhallintajärjestelmäksi valittiin Bitbucket ja syötettiin käytettävät tunnistetiedot sekä käytettävän projektin nimi. Kuvasta 16 nähdään työn suorittamisessa käytettävän liukuhihnan määritys konfiguraatiosta.

Kuva 16. Käytettävän tiedoston määritys julkaisuliukuhihnaan.

Kuvassa 16 määritettiin vielä konfiguraatiosta projektin haara, jota työssä käytetään kohdassa ”Branches to build”. ”Script path” -osioon pystyttiin erikseen määrittämään tarkempi polku liukuhihnaan, jos se ei ole projektin juuressa nimellä ”Jenkinsfile”. Tässä tapauksessa julkaisuliukuhihna sijaitsi vars-kansion alla nimellä startDeployPipeline.groovy. Tällä määrityksellä työn käynnistyessä Jenkins tietää, mitä tiedostoa projektista etsiä.

Lisäsin aiempaan versioon myös uuden parametrin, jossa määritellään projekti, joka halutaan julkaista. Tämän parametrin avulla luetaan Jenkinsin konfiguraatiotiedostoista oikean projektin tiedot, jota käytetään julkaisuskriptin muuttujissa. Käydään läpi vielä julkaisuliukuhihnan koodin eri osiot. Esimerkkikoodissa 14 on julkaisuliukuhihnan aloitus.

```
***startDeployPipeline.groovy
@Library('sharedLibraryTest') _

currentBuild.displayName = "#$BUILD_NUMBER $PROJECT $TARGET $GIT_NAME"
pipeline {
    agent {
        label 'kubernetes-jenkins-build'
    }
```

Esimerkkikoodi 14. Julkaisuliukuhihnan aloitus ja agentti.

Esimerkkikoodissa 14 nähdään julkaisuliukuhihnan aloitus. @Library-annotaatiolla otetaan käyttöön jaettu kirjasto. currentBuild.displayName:n avulla määritellään Jenkins GUI:ssa näkyvä koonnin nimi. Kuten vakioidussa pohjassakin, deklaratiivinen liukuhihna alkaa pipeline-lohkolla, jota seuraa agent-lohko, jossa agentiksi määritellään tässäkin tapauksessa ”kubernetes-jenkins-build”-kapseli.

```
***startDeployPipeline.groovy
environment {
        // Project
        allVars = getConfigVars('test'+params.PROJECT+'file')
        projectName = getProp(allVars, 'projectName')
        repositoryName = getProp(allVars, 'repositoryName'
        displayName = getProp(allVars, 'displayName')
        namespaceLab = getProp(allVars, 'namespaceLab')
        namespacePre = getProp(allVars, 'namespacePre')
        namespacePro = getProp(allVars, 'namespacePro')
        tag1 = getProp(allVars, 'tag1')
        tag2 = getProp(allVars, 'tag2')

        // Credentials
        TEAMS_URL = credentials('TeamsWebhookSamlinkJenkins')
        KUBECONFIG = credentials('hybridcloud-kubeconfig')
    }
```

Esimerkkikoodi 15. Julkaisuliukuhihnan environment-muuttujat

Esimerkkikoodista 15 nähdään julkaisuliukuhihnan environment-muuttujat. Kuten myös vakioidussa pohjassa oli, julkaisuliukuhihnassakin Project-osion muuttujat luetaan konfiguraatiotiedostosta ja arvot vaihtelevat projektin mukaan. Credentials-osiossa on tunnistetiedot, jotka luetaan käyttämällä tunnistetiedon ID-arvoa. Environment-lohkon muuttujia voidaan käyttää jokaisessa liukuhihnan vaiheessa haluttaessa.

```
***startDeployPipeline.groovy
parameters {
        choice(name: 'PROJECT', choices: ['drm', 'hubdater'…], de-
scription: 'The name of the project to be deployed.')
        string(name: 'GIT_REF', defaultValue: '**', description:
'Branch reference to checkout')
        string(name: 'GIT_NAME', defaultValue: '**', description: '')
        choice(name: 'TARGET', choices: ['lab', 'pre', 'pro', 'aws'],
description: 'Choose where the build will be deployed.')
        string(name: 'EXAMPLE_MANIFEST_VERSIONS', defaultValue: '',
description: 'Leave empty not apply example manifests. Space seperated
list of versions to be installed of the example manifests. Example: v1
v3')
        booleanParam(name: 'DRY_RUN', defaultValue: true,  descrip-
tion: 'Simulates running configuration. Typically, you set this
false.')
        booleanParam(name: 'TRIM_EXAMPLE_PREFIX', defaultValue: false,
description: 'Removes "example-" prefix from example manifests to in-
stall them directly, Note: This only trims on LAB environment')
    }
```

Esimerkkikoodi 16. Julkaisuliukuhihnan parametrit.

Esimerkkikoodista 16 nähdään julkaisuliukuhihnan parametrit. Nämä syötetään työtä käynnistettäessä. Jos julkaisutyö käynnistetään Teams-ilmoituksesta, ei näitä parametreja tarvitse muuttaa, sillä ne tulevat viestin linkistä automaattisesti. Myös parametreja voidaan käyttää liukuhihnan jokaisessa vaiheessa.

```
***startDeployPipeline.groovy
stages {
        stage('Checkout') {
            steps {
                checkoutDeploy()
            }
        }
        stage('Deploy') {
            steps {
                deployToCloud()
            }
        }
    }
    post {
        success {
            sendTeamsDeployedMessage()
        }
    }
```

Esimerkkikoodi 17. Julkaisuliukuhihnan vaiheet.

Esimerkkikoodista 17 nähdään julkaisuliukuhihnan vaiheet. Julkaistavaksi haluttu projekti täytyy ensimmäisessä vaiheessa hakea (engl. checkout) versionhallinasta checkoutDeploy()-kutsulla. Tämän jälkeen siirrytään Deploy-vaiheeseen, jossa varsinainen julkaisu tapahtuu parametreissa valittuun ympäristöön (TARGET-parametri). Lopuksi lähetetään Teams-kanavalle vahvistusilmoitus onnistuneesta julkaisusta, jos aikaisemmat vaiheet ovat menneet läpi.

5 Käyttöönoton lopputulokset

Tässä luvussa käsitellään lyhyesti Kuberneteksen, monihaaraisten liukuhihnojen sekä vakioidun pohjan käyttöönoton lopputuloksia. Lisäksi pohditaan myös mahdollisia jatkokehityskohteita tai mitä voisi vielä parantaa.

5.1 Kuberneteksen käyttöönotto

Kuberneteksen käyttöönotto kaikissa projekteissa sujui melko mutkattomasti. Aluksi piti tietenkin selvittää, mitä kaikkea tarvitsee tehdä käyttöönottoa varten. Kuitenkin suurin osa Kubernetekseen liittyen oli tehty Jenkinsissä jo valmiiksi, muun muassa määritelty käytettävä klusteri ja sen kapselit, kuten ”kubernetes-jenkins” ja ”kubernetes-jenkins-build”. Käyttöönotto toteutettiin lyhykäisyydessään lisäämällä liukuhihnan alkuun agent-lohko ja nimettiin käytettävä klusteri. Tässä tapauksessa klusterina käytettiin Kubernetes-klusteria ”kubernetes-jenkins-build”.

Kuberneteksen osalta ei jäänyt juurikaan jatkokehityskohteita. Mahdollisesti voisi lisätä erilaisia kapseleita, jotka sopisivat paremmin tiettyjen tehtävien suorittamiseen. Tällöin kapselit voisivat sisältää vain kontteja, joita kyseiseen tehtävään tarvitaan ja kapseleista saataisiin näin hieman kevyempiä.

5.2 Monihaaraisten liukuhihnojen käyttöönotto

Monihaaraisten liukuhihnojen käyttöönotto oli yksinkertaista ja hoitui helposti. Päädyimme loppujen lopuksi käyttämään Jenkinsin organisaatiokansiota (engl.

organization folder). Organisaatiokansioiden avulla Jenkins voi valvoa koko Bitbucket-tiimiä/projektia. Tämän avulla voidaan luoda Jenkinsiin automaattisesti uusia monihaaraisia liukuhihnoja projekteille, jotka sisältävät Jenkins-tiedoston. Organisaatiokansion käyttöä varten tarvitaan Bitbucketin tapauksessa ”Bitbucket Branch Source” -laajennus. [25.]

Tämän avulla siis saimme jokaiselle tarvittavalle projektille automaattisesti monihaaraisen liukuhihnan eikä niitä tarvinnut luoda jokaiselle projektille erikseen. Organisaatiokansion konfigurointikin onnistui yksinkertaisesti. Konfiguraatiosta täytyi vain määritellä Bitbucketin kansio, jota haluttiin tarkkailla. Konfiguroinnin jälkeen organisaatiokansio skannasi määritellyn kansion projektit. Skannauksen jälkeen organisaatiokansion alla näkyivät projektit, joissa oli vähintään yksi haara, josta löytyi Jenkins-tiedosto. Organisaatiokansion voi milloin tahansa uudelleen skannata uusia projekteja sekä haaroja käynnistämällä skannaus manuaalisesti Jenkinsin GUI:n kautta.

5.3 Jaettujen kirjastojen käyttöönotto

Vakioidun pohjan käyttöönottoon tarvittiin eniten työtä, sillä eri vaiheiden universalointi vei paljon aikaa. Se kuitenkin saatiin toimimaan muutamissa projekteissa pienten viilausten jälkeen. Vakioidun pohjan ansiosta uusien töiden lisääminen on helpompaa, ja uudet projektit voidaan laittaa käyttämään kaikille yhteistä liukuhihnaa.

Julkaisuliukuhihnan käyttöönotto vei myös suhteellisen paljon aikaa. Kuitenkin verrattuna vakioituun pohjaan vaiheita ja universalointia vaativia kohtia oli vähemmän. Myös julkaisuliukuhihna saatiin toimimaan hyvin tarvittavilla projekteilla pienten viilausten jälkeen.

Jatkokehityskohteena on saada vakioitu pohja toimimaan myös kaikilla muilla viihdealustaprojekteilla ja myöhemmin myös mahdollisesti radioverkkoprojekteissa. Sama koskee myös julkaisuliukuhihnaa, mutta sen kohdalla tarvittavat muutokset ovat pienempiä. Lisäksi vakioidun pohjan sekä julkaisuliukuhihnan

koodeissa on refaktoroitavaa esimerkiksi konfiguraatiotiedostosta lukemisen kanssa.

6 Yhteenveto

Insinöörityön tarkoituksena oli ratkaista Jenkins-työsolmujen skaalautuvuusongelma ottamalla käyttöön Kubernetes-klusteri. Lisäksi Jenkinsissä otettiin käyttöön monihaaraiset liukuhihnat sekä jaetut kirjastot. Monihaaraisten liukuhihnojen avulla oli tarkoitus mahdollistaa projektien eri haarojen koonti Jenkinsissä sekä helpottaa Jenkinsin käyttöä useamman projektin kanssa jatkossa. Jaettujen kirjastojen avulla toteutettiin vakioitu pohja, jonka tarkoituksena oli luoda liukuhihna, jota jokainen projekti voisi käyttää.

Insinöörityön tavoitteisiin päästiin ja skaalautuvuusongelma saatiin ratkaistua Kuberneteksen avulla. Työn käynnistyessä etsitään sopiva kapseli sen suorittamiseen, jonka johdosta suoritusaika hidastui muutamalla sekunnilla. Monihaaraisten liukuhihnojen käyttöönotto sujui mutkattomasti ja jokaisen haaran, jossa on Jenkins-tiedosto, voi koota. Myös jaetun kirjaston avulla toteutettu vakioitu pohja saatiin toimimaan tarvittavilla projekteilla. Käyttöönottovaiheessa tarvittiin vielä pieniä korjauksia pohjaan, jotta se toimisi jokaisessa projektissa oikein.

Muutosten ansiosta Jenkinsin töiden suorittamisessa ei enää tarvita staattisia Jenkins-työsolmuja. Monihaaraisten liukuhihnojen käytöllä saatiin Jenkinsin projektirakennetta yksinkertaistettua ja helpotettua muiden kuin master-haaran julkaisua eri ympäristöihin. Vakioidun pohjan avulla, joka toteutettiin jaetulla kirjastolla, saatiin vähennettyä projektien Jenkins-töiden määrää. Aikaisemmin jokaisella projektilla oli 3-5 työtä eri tarkoituksiin, mutta vakioidun pohjan avulla saatiin työt sisällytettyä projektin monihaaraiseen liukuhihnaan sekä jokaiselle projektille yhteiseen julkaisutyöhön. Julkaisutyön avulla jokainen projekti voi samaa liukuhihnaa käyttäen julkaista koontiversion haluttuun ympäristöön. Aikaisemmin jokaisella projektilla oli oma julkaisutyönsä. Myös julkaisutyö toteutettiin jaettua kirjastoa käyttämällä.

Kubernetes, monihaaraiset liukuhihnat ja jaetut kirjastot saatiin käyttöönotettua muutamassa projektissa. Jatkokehityskohteena on saada nämä käyttöön myös muissa viihdealustaprojekteissa ja myöhemmin myös radioverkkoprojekteissa. Lisäksi vakioidun pohjan koodissa on hieman refaktoroitavaa esimerkiksi konfiguraatiotiedostosta lukemisen ja parin projektin erityistapausten käsittelemisen kanssa.

Lähteet

1 Riglian, Adam. 2019. What is Jenkins and How Does It Work. Verkkoaineisto. TechTarget. <https://www.techtarget.com/searchsoftwarequality/definition/Jenkins>. Päivitetty 11.2019. Luettu 25.2.2023.

2 Automated CI/CD with Jenkins. Verkkoaineisto. Medium. <https://medium.com/avmconsulting-blog/automated-ci-cd-with-jenkins-39b21c7c8035>. Päivitetty 2.6.2020. Luettu 1.5.2023.

3 BasuMallick, Chiradeep. What is Jenkins? Working, uses, pipelines, and features. Verkkoaineisto. Spiceworks. <https://www.spiceworks.com/tech/devops/articles/what-is-jenkins/>. Päivitetty 20.7.2022. Luettu 25.2.2023.

4 Docker. Verkkoaineisto. Jenkins Documentation. <https://www.jenkins.io/doc/book/installing/docker/>. Luettu 10.4.2023.

5 Getting started with pipeline. Verkkoaineisto. Jenkins Documentation. < https://www.jenkins.io/doc/book/pipeline/getting-started/>. Luettu 18.3.2023.

6 What is Kubernetes. Verkkoaineisto. Google Cloud. <https://cloud.google.com/learn/what-is-kubernetes>. Luettu 25.2.2023.

7 Casey, Kevin. What's the difference between a pod, a cluster, and a container. Verkkoaineisto. The Enterprises Project. <https://enterprisersproject.com/article/2020/9/pod-cluster-container-what-is-difference>. Päivitetty 1.9.2020. Luettu 10.4.2023.

8 What is Kubernetes Cluster. Verkkoaineisto. Vmware. <https://www.vmware.com/topics/glossary/content/kubernetes-cluster.html>. Luettu 10.4.2023.

9 Kubernetes Components. Verkkoaineisto. Kubernetes Documentation. <https://kubernetes.io/docs/concepts/overview/components/>. Luettu 1.5.2023.

10 Palmer, Matthew. Kubernetes networking guide for beginners. Verkkoaineisto. Matthewpalmer.net. <https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-networking-guide-beginners.html>. Luettu 1.5.2023.

11 Pods. Verkkoaineisto. Kubernetes Documentation. <https://kubernetes.io/docs/concepts/workloads/pods/>. Luettu 30.3.2023.

12 IBM Cloud Education. Top 7 Benefits of Kubernetes. Verkkoaineisto. IBM. <https://www.ibm.com/cloud/blog/top-7-benefits-of-kubernetes>. Päivitetty 6.9.2022. Luettu 25.2.2023.

13 Groovy Tutorial. Verkkoaineisto. Tutorialspoint. <https://www.tutorialspoint.com/groovy/index.htm>. Luettu 27.3.2023.

14 Apache Groovy Tutorial. Verkkoaineisto. JavaTpoint. <https://www.javatpoint.com/groovy>. Luettu 27.3.2023.

15 Managing nodes. Verkkoaineisto. Jenkins Documentation. <https://www.jenkins.io/doc/book/managing/nodes/>. Luettu 30.4.2023.

16 Brinley, Dennis. Horizontal Scaling of Event-Driven Microservices in Kubernetes. Verkkoaineisto. Solace. <https://solace.com/blog/horizontal-scaling-of-event-driven-microservices-in-kubernetes/>. Luettu 1.5.2023.

17 Horizontal Pod Autoscaling. Verkkoaineisto. Kubernetes Documentation. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Luettu 10.4.2023.

18 Dandu, Praveen. Understanding the Differences Between Jenkins Scripted and Declarative Pipeline: A Comprehensive Guide with Real-World Examples. Verkkoaineisto. Blog Devgenius. <https://blog.devgenius.io/understanding-the-differences-between-jenkins-scripted-and-declarative-pipeline-a-comprehensive-960826e26c2>. Luettu 2.5.2023.

19 McKenzie, Cameron. Declarative vs. scripted pipelines: What's the difference. Verkkoaineisto. TheServerSide. <https://www.theserverside.com/answer/Declarative-vs-scripted-pipelines-Whats-the-difference>. Päivitetty 18.12.2020. Luettu 11.3.2023.

20 Using credentials. Verkkoaineisto. Jenkins Documentation. <https://www.jenkins.io/doc/book/using/using-credentials/>. Luettu 27.3.2023.

21 Chapman, Eric. Implementing Ephemeral Jenkins Masters with Kubernetes. Verkkoaineisto. Liatrio. <https://www.liatrio.com/blog/ephemeral-jenkins>. Päivitetty 8.1.2020. Luettu 7.5.2023.

22 Kovacevik, Aleksandar. Jenkins Shared Library: How to Create, Configure and Use. Verkkoaineisto. PhoenixNAP. <https://phoenixnap.com/kb/jenkins-shared-library>. Päivitetty 3.2.2022. Luettu 11.3.2023.

23 Extending with shared libraries. Verkkoaineisto. Jenkins Documentation. <https://www.jenkins.io/doc/book/pipeline/shared-libraries/>. Luettu 11.3.2023.

24 Wilson, Bibin. Jenkins Multibranch Pipeline Tutorial For Beginners. Verkkoaineisto. Devopscube. <https://devopscube.com/jenkins-multibranch-pipeline-tutorial/>. Päivitetty 6.8.2020. Luettu 11.3.2023.

25 Branches and pull requests. Verkkoaineisto. Jenkins Documentation. <https://www.jenkins.io/doc/book/pipeline/multibranch/>. Luettu 1.5.2023.