Mats Ove Mandt Skjærstein og Alexander Jakobsen

CI/CD for SkyHiGh

Bacheloroppgave i IT-drift og informasjonssikkerhet
Veileder: Erik Hjelmås
Mai 2019

Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Institutt for informasjonssikkerhet og kommunikasjonsteknologi

NTNU
Kunnskap for en bedre verden

Mats Ove Mandt Skjærstein og Alexander Jakobsen

CI/CD for SkyHiGh

Bacheloroppgave i IT-drift og informasjonssikkerhet
Veileder: Erik Hjelmås
Mai 2019

Norges teknisk-naturvitenskapelige universitet
Institutt for informasjonssikkerhet og kommunikajsonsteknologi

NTNU
Kunnskap for en bedre verden

Sammendrag av Bacheloroppgaven

Tittel:	**CI/CD for SkyHiGh**
Dato:	20.05.2019
Deltakere:	Mats Ove Mandt Skjærstein Alexander Jakobsen
Veiledere:	Erik Hjelmås
Oppdragsgiver:	Norwegian University of Science and Technology
Kontaktperson:	Eigil Obrestad, Lars Erik Pedersen
Nøkkelord:	CI/CD pipeline, automatisert testing, OpenStack
Antall sider:	64
Antall vedlegg:	7
Tilgjengelighet:	Åpen

Sammendrag:

I moderne IT-drift som bruker infrastruktur som kode kan det være vanskelig å oppdage feil før koden blir rullet ut til et produksjonsmiljø. NTNU Gjøvik hadde et ønske om å utforske muligheten automatisk testing kan gi for deres skyplattform SkyHiGh. Dette er en openstack implementasjon som driftes av NTNU Gjøvik hvor de bruker Puppet som konfigurasjonsstyringsverktøy til å installere og håndtere plattformen. I dette prosjektet har vi utviklet en CI/CD pipeline for openstack modulen keystone ved bruk av GitLab CI. Pipelinen sjekker all endret puppetkode for feil ved å bruke verktøy for statisk kodeanalyse. Keystone modulen blir installert ved å bruke NTNUs puppetkode og senere akseptanse-, og integrasjonstestet. Dersom testene fullføre uten feil, vil koden gå videre til en annen GitLab branch.

Summary of Graduate Project

Title:	**CI/CD for SkyHiGh**
Date:	20.05.2019
Authors:	Mats Ove Mandt Skjærstein Alexander Jakobsen
Supervisor:	Erik Hjelmås
Employer:	Norwegian University of Science and Technology
Contact Person:	Eigil Obrestad, Lars Erik Pedersen
Keywords:	CI/CD pipeline, automated testing, OpenStack
Pages:	64
Attachments:	7
Availability:	Open

Abstract:

In modern IT operations that use infrastructure as code, it can be difficult to detect bugs before the code is deployed to production. NTNU Gjøvik had a desire to explore the possibility that automatic testing can provide for their cloud platform SkyHiGh. This is an openstack implementation run by NTNU Gjøvik, where they use Puppet as a configuration management tool to install and manage the platform. In this project we have developed a CI/CD pipeline for the openstack module keystone using GitLab CI. The pipeline checks all changed puppetcode for errors using static analysis tools. The keystone module is installed using NTNU's puppetcode and later tested using acceptance and integration tests. The code will then move on to another GitLab branch if all tests are passed.

Forord

Forfatterene av denne bacheloroppgaven, Alexander Jakobsen og Mats Ove Mandt Skjærstein, ønsker å takke vår veileder Erik Hjelmås for mange gode råd, veiledning og samarbeidet gjennom utførelsen av denne oppgaven.

Vi vill også takke vår oppdragsgiver representert ved Eigil Obrestad og Lars Erik Pedersen for all den hjelpen vi har fått av dem til å bruke deres Puppetkode. Og for den tiden de har brukt på dette prosjektet.

Innhold

Figurer

Tabeller

Listings

Terminologiliste

Open Source
Åpen kildekode på norsk. Er en alminnelig tilgjengelig sett av kode som kan bli inspisert, modifisert, endret eller bidratt til av hvem som helst.

Commit
En kommando i ”Git”. Tar valgt kode og sender det opp til et repository.

Repo
Forkortelse av ordet ”repository”. Et repo er et sted hvor kode kan lagres.

SkyHiGh
SkyHiGh er navnet på openstack implementasjonen til NTNU.

Openstack
Openstack er en Open Source skyplattoform som kan installeres og håndteres lokalt.

VM
Forkortelse for ”virtuell maskin”.

Yaml, yml
Forkortelse for ”YAML Ain’t Markup Language”. Yaml er et tekstformat brukt til å representere data.

Branch
En gren, eller en serie med commits i et repository. Et repository kan ha flere uavhenigige branches.

Merge, merge request
Å merge vil si å innføre kode fra en branch inn i en annen. Et merge request er en forespørsel om en merge.

Json
Forkortelse for ”JavaScript Object Notation”. Er et tekstformat brukt til å formatere dokumenter som brukes i datautveksling.

1 Innledning

1.1 Bakgrunn

NTNU benytter openstack som en privat skyløsning. Openstack gir studenter og ansatte en plattform hvor en kan lage virtuelle maskiner og nettverk. Plattformen benyttes av studenter som skal lære seg systemadministrasjon, forskere som har behov for å kjøre simuleringer, samt mye annet. Det er IIK som er ansvarlig for arkitekturen bak de fleste openstack skyene på NTNU.

Selve arkitekturen bak SkyHiGh styres av konfigurasjonsstyringssystemet Puppet, og manifestene som benyttes ligger lagret på GitHub [1]. Det er allerede utarbeidet et dedikert testmiljø for SkyHiGh, og alle endringer som utføres blir rullet ut der før de rulles ut i produksjon. Slik kan det manuelt sjekkes at endringene ikke introduserer feil i andre deler av arkitekturen.

1.2 Fagområde

Hovedområdet for oppgaven er ”continuous integration/continuous delivery” (CI/CD) pipeline. Pipelinen brukes til å sette opp nødvendig infrastruktur og kjøre de automatiske testene for openstack implementasjonen ”SkyHiGh” for NTNU automatisk ved endringer i kodebasen.

CI/CD er en praksis som tar endret kode gjennom en ”pipeline”, denne pipelinen består av en rekke steg hvor forskjellige automatiske tester kjører. Gjennom denne praksisen kan feil oppdages før koden går videre til et produksjonsmiljø.

Under arbeidet med denne oppgaven vil prosjektgruppen komme inn på flere fagområder som:

- **Programvareutvikling:** For utvikling av automatiske tester, installasjon og konfigurering av tjenester og programmer.
- **Drift av infrastruktur:** Design og opprettelse av infrastrukturen som kjører de automatiske testene.

1.3 Oppgavebeskrivelse

NTNU er systemeier og oppdragsgiver for SkyHiGh og denne oppgaven, den originale oppgavebeskrivelse fra NTNU var følgende:

> Da litt tilfeldig manuell testing ikke nødvendigvis dekker all funksjonalitet, og det å systematisk teste alle funksjoner for hver endring er tidskrevende, er det et ønske om å utforske muligheten som automatisk testing kan gi oss. Det har hendt at endringer som er introdusert i arkitekturen har resultert i at deler av arkitekturen har sluttet å

> virke uten at dette ble avdekket i testmiljøet. Dette ønsker man å unngå i fremtiden.
>
> Oppgaven handler dermed om å evaluere hvordan automatisk testing kan benyttes for å øke kvaliteten på de endringer som gjøres. Kan man benytte automatisk verktøy for å utføre:
>
> - Kontroll av kodekvalitet.
> - Integreringstester, for å verifisere funksjonalitet før man godtar at ny kode ”merges” inn i produksjonssystemene.
>
> Da all kodet infrastruktur er styrt av puppet i dag er deg også ønskelig at en eventuell infrastruktur som må settes opp for å gjøre automatisk testing også kan installeres og konfigurasjonsstyres av puppet.

Etter dialog med oppdragsgiver ble oppgaven presisert ytterligere.

NTNU ønsker en CI/CD pipeline for å sjekke kodekvalitet på puppet kode og som tester de forskjellige openstack modulene i SkyHiGh som styres av puppet. Siden SkyHiGh består av mange forskjellige openstack moduler ble prosjektgruppen og oppdragsgivere enige om å utvikle en CI/CD prototype pipeline for openstack modulen keystone.

Oppgaven omhandler å lage en CI/CD pipeline for NTNUs openstack modul keystone. Det er ønskelig at denne pipelinen innholder følgende:

- Kontroll av kodekvalitet på puppetfiler.
- Automatisk testing av at keystone modulen ble installert.
- Automatisk testing av at keystone modulen fungerer.
- Automatisk ”merge” eller ”merge request” av ny kode til et produksjonsmiljø.

Prosjektgruppen skal forsøke å implementere en løsning som inneholder den ønskede funksjonalitet beskrevet over, og å svare på følgende spørsmål:

1. Hvordan kan kontroll av kodekvaliteten utføres?
2. Hvordan kan keystone modulen installeres automatisk ved endring i NTNUs kode?
3. Hvordan kan keystone modulen akseptanse-, og integrasjonstestes?
4. Hvordan kan koden automatisk ”merges” inn i et produksjonsmiljø?

1.4 Avgrensing og rammer

Hovedfokuset på oppgaven er å evaluere hvordan automatisk testing kan benyttes i SkyHiGh. Det har blitt valgt å begrense oppgaven til å lage automatiske tester for NTNUs keystone openstack modul. Prosjektgruppen skal ikke utvikle en fullstendig CI/CD pipeline som tester hele SkyHiGh infrastrukturen.

Denne avgrensningen gir gruppen mer tid til å teste og evaluere forskjellige metoder for å implementere automatiske tester for keystone.

Etter samtale med oppdragsgiver ble det klargjort følgende rammer for prosjektet:

- Det skal ikke kjøpes inn lisenser til programvare.
- Vedlikeholdbarhet og robusthet av løsningen er viktigere enn omfattende funksjonalitet.
- Det er ikke ønskelig at infrastrukturen bruker mye ressurser.
- Det er ønskelig at systemer som kjører de automatiserte testene skal kunne konfigureres ved bruk av puppet.

1.5 Formål

Oppgavens formål er å utforske muligheten for automatisert testing for openstack koden som brukes av NTNU til å konfigurere SkyHiGh. Evaluere hvordan automatisert testing kan benyttes for å øke kvaliteten på denne koden i en CI/CD prosess ved endringer av kode.

Dette innebærer å undersøke, teste og evaluere ulike teknologier og metoder for å sette opp en CI/CD pipeline hvor forskjellige automatiske tester kjører. Det skal settes opp en prototype for å demonstrere mulighetene for automatiserte tester i en CI/CD pipeline for SkyHiGhs keystone modul.

1.6 Målgruppe

Målgruppen for denne oppgaven er i hovedsak NTNUs oppdragsgivere, siden de er systemeiere av SkyHiGh og ønsker å se på de mulighetene testene kan gi. Andre målgrupper for denne rapporten er bedrifter som bruker openstack med puppet og er interessert i muligheten automatisk testing kan gi for deres systemer.

1.7 Prosjektgruppens bakgrunn

Begge medlemmene av prosjektgruppen går samme studieløp, IT-drift og informasjonssikkerhet ved NTNU Gjøvik. Gruppemedlemmene har gjennom studiet fått erfaring innen drift av infrastruktur i skytjenesten openstack, konfigurasjonsstyringssystemet puppet, Docker teknologier, Linux maskiner, drift og oppsett av MySQL databaser, drift av forskjellige tjenestearkitekturer, nettverk, sikkerhetsvurderinger og programvareutvikling.

Prosjektet krever at gruppemedlemmene tilegner seg ny kunnskap om CI/CD prosesser og forskjellige teknologier for å designe og lage automatiske tester og infrastruktur.

1.8 Roller

Eigil Obrestad og Lars Erik Pedersen er systemeiere for SkyHiGh og fungerer som oppdragsgivere for oppgaven. Erik Hjelmås er veileder for bacheloroppgaven.

Prosjektgruppen består av Alexander Jakobsen og Mats Ove Mandt Skjærstein. Mats er

prosjektleder og har ansvaret for kommunikasjon mellom de involverte partene.

1.9 Om rapporten

Vi har i rapporten valgt å bruke flere engelske begreper og betegnelser, da disse er mer gjenkjennelig av fagmiljøet. Denne rapporten bruker klikkbare linker til andre kapitler, kilder, og til internett for videre lesing. Rapporten leses derfor best i en elektronisk utgave. Oppgaven har to forskjellige stiler for å vise kildekode. Hvis deler av kode fra en fil er vist i dokumentet, vises ikke linjenummer. I disse utdragene brukes "..." for å markere om det ligger kode over eller under det fremviste utdraget. Fremvist kildekode med linjenummer viser hele kodefilen og ikke bare et utdrag fra en fil.

Rapportens struktur

Vi har valgt å dele rapporten inn i 7 kapitler. En kort beskrivelse av kapitlene er gitt nedenfor:

1. **Innledning**
   Dette kapittelet inneholder opplysninger, organisering og oppgavebeskrivelsen av prosjektet.
2. **Teori**
   Dette kapittelet inneholder relevant teori til oppgaven.
3. **Kravspesifikasjoner**
   Dette kapittelet inneholder Use Cases av systemet som skal utvikles.
4. **Vurdering og valg av teknologier**
   Dette kapittelet innholder vurderinger og valg av verktøy og teknologier som blir tatt i bruk for å lage CI/CD pipelinen.
5. **System arkitektur**
   Dette kapittelet inneholder en beskrivelse av SkyHiGhs openstack arkitektur og CI/CD pipeline arkitekturen.
6. **Implementasjon**
   Dette kapittelet tar for seg implementasjonen av løsningen som ble utviklet.
7. **Konklusjon**
   Dette kapittelet inneholder resultatet av arbeidet som ble utført av prosjektgruppen. Samt forslag til videre arbeid og prosjektgruppens evaluering av gjennomføringen av oppgaven.

2 Teori

2.1 Infrastruktur som kode

NTNUs openstack implementasjon ”SkyHiGh” blir håndtert av konfigurasjonsstyringsverktøyet Puppet. Puppet er et Open Source verktøy som brukes til å installere, konfigurere og administrere ressurser på et system. Puppet systemet brukes til å beskrive tilstanden på en server ved bruk av et deklarativt domenespesifikt programmeringsspråk, *”domain-specific language (DSL)”*. I motsetning til andre programmeringsspråk beskriver puppets språk tilstanden en maskin skal være i, ikke hvordan maskinen kommer til denne tilstanden. En av styrkene til puppet er idempotent konfigurasjon.

At en operasjon er idempotent vil si at en operasjon kan gjentas flere ganger og ha samme effekt [2]. Det vil si at hvis en maskin har korrekt tilstand, vil ingenting endres på maskinen. Dersom maskinen ikke har korrekt tilstand, vil tilstanden endres til sin korrekte tilstand.

Definering av konfigurasjon i puppet kan gjøres for hver enkelt maskin, eller ved å gruppere maskiner sammen. Konfigurasjonen er beskrevet i filer kalt manifester, disse manifestene styrer hvilke programmer, resurser og konfigurasjon maskinene skal ha. På maskiner som blir håndtert av puppet, kjører puppets tjeneste i bakgrunnen og sjekker om konfigurasjonen har blitt endret fra sin ønsket tilstand i forhåndsdefinerte intervaller. Hvis maskinens konfigurasjon ikke stemmer med hva som er breskrevet i manifestet, vil puppet kjøre og endre tilstanden slik at den blir korrekt. Denne måten å håndtere maskiner på blir kalt ”infrastruktur som kode”.

Infrastruktur som kode er en prosess for å håndtere og sette opp maskiner ved bruk av definisjonsfiler. Dette brukes i stedet for manuelle prosesser ved oppsett eller endringer av et system. Kief Morris beskriver dette som en metode for automasjon av infrastruktur basert på prosesser fra programvareutvikling. Disse prosessene understreker konsistente og gjentagbare rutiner for å sette opp og endre systemer og deres konfigurasjon fra definisjonsfilene som blir rullet ut til maskinene gjennom prosesser som kjører automatisk. Ved bruk av infrastruktur som kode skal infrastrukturelementer bli behandlet som programvare og data [3].

Formålet med infrastruktur som kode er å gjøre endringer av systemer enklere å håndtere og minimere feil når endringer skjer. Å begrense manuelle og gjentagbare prosesser ved bruk av automatisering. Ved å ha systemer definert som kode kan prosesser og verktøy fra programvareutvikling bli brukt, som versjonskontroll, automatisk testing, continuous integration (CI) og continuous delivery (CD).

2.2 Automatisering

For moderne IT-drift som bruker prinsipper fra infrastruktur som kode er automasjon en viktig og stor del av arbeidsprosessen til administratorer. Ved å ta i bruk automatisering frigjøres tid og ressurser til å fokusere på endringer og øke kvaliteten på systemet de drifter. Manuelle og gjentakene prosesser tar lang tid og øker sjansen for at feil oppstår.

Automasjon betyr at en maskin gjør noe for oss. En oppgave er automatisert når et menneske ikke lenger utfører denne oppgaven, men programvaren gjør det for oss [4]. I følge *Store norske leksikon*, defineres automasjon som: *"Teknikken å få systemer til å fungere uten, eller med liten grad av menneskelig medvirkning"* [5].

I boken *"The Practice of Cloud System Administration"*[4] beskrives noen av målene med automasjon som:

- **Skaleringshjelp:** Ved automasjon kan en person gjøre jobben til mange.
- **Forbedret nøyaktighet:** Automasjon er mindre utsatt for feil enn mennesker.
- **Økt repeterbarhet:** Programvare er mer konsist enn mennesker. Hvis en oppgave skal utføres flere ganger vil programvare gjøre det samme hver gang, dette er ikke alltid tilfellet hos mennesker.
- **Pålitelighet:** Når en oppgave er automatisert kan statistikk bli samlet opp. Denne statistikken kan hjelpe til med å gjøre prosessen mer pålitelig over tid.
- **Sparer tid:** Automasjon sparer tid for en bedrift.
- **Gjør prosesser raskere:** Manuelle prosesser tar lengre tid enn automatiske.

Disse målene er viktig for bedrifter som drifter store plattformer med få personer. Konfigurasjonsstyring med puppet og CI/CD pipelines for testing muliggjøres ved å bruke automatiserte prosesser som installerer, konfigurerer og tester kode gjennom hele utviklingsprosessen for et system.

2.3 Continuous Integration/Continuous Delivery (CI/CD)

Når det jobbes med å utvikle eller endre et system som er definert som "infrastruktur som kode" hvor flere personer er involvert i utviklingsprosessen. Da bør det brukes et system som kan hjelpe med å holde høy kodekvalitet og minimere feil. Continuous Integration (CI) er en praksis som hjelper utviklere med dette. CI blir beskrevet av Kief Morris i sin bok *"Infrastructure as Code"* som en praksis for å hyppig integrere og teste alle endringer til et system som er under utvikling [6].

CI lar utviklere integrere og teste alle endringer som er gjort i en kodebase. Endret kode blir commit-et til en felles branch og hver gang dette skjer blir koden testet gjennom CI verktøyet. Dette gir muligheten for rask tilbakemelding når en endring ødelegger ønsket funksjonalitet eller når koden ikke tilfredsstiller god kodekvalitet. For at CI prosessen skal fungere må alle utviklere commit-e sin kode til samme branch og la verktøyet sjekke og teste koden. Hvis tester feiler må disse rettes opp i før nye endringer kan innføres i repo-et. På denne måten vil utviklere være mer sikker på at koden som ligger i denne

”utviklings” branches fungere når den senere skal rulles ut til produksjonsmiljøet.

Continuous Delivery (CD) er en utvidelse av CI som skal sikre at hele systemet er klar for produksjon. CD tar koden som passerte CI fasen og legger det ut til et testmiljø. Dette testmiljøet kan ha manuelle og automatiske tester som tester hele systemet. CD skal validere at systemet er klar for produksjon og finne feil som ikke ble oppdaget av CI prosessen [7].

Continuous Deployement er en annen prosess som kan ble brukt i en pipeline. Continuous Deployement er en forlengesle av Continuous Delivery hvor kode blir automatisk rullet ut og installert i produksjonsmiljøet [8].

En CI/CD pipeline er begrepet som brukes for å ta endret kode gjennom CI og CD prosessene. Det finnes flere forskjellige verktøy for å lage en CI/CD pipeline, eksempel på slike produkter er Jenkins og GitLab CI. Testene som kjører i en pipeline kan implementeres på flere forskjellige måter. En pipeline kan brukes til å installere eller bygge et system, teste og sjekke kvaliteten på koden som er skrevet med manuelle og automatiske steg. Ofte brukes en test pyramide når tester for et system skal implementeres.

2.4 Automatisk testing av infrastruktur

Automatisk testing i en CI/CD pipeline er nødvending hvis tester skal kontinuerlig kjøres ved hver endring i en kodebase. Det vil ta for lang tid hvis manuelle tester må kjøres hver gang en utikler endrer en del av systemet, samtidig som faren for feil øker ved å bare teste systemet manuelt.

Formålet med automatisk testing er å ha repeterbare tester som kjører fortere enn manuelle tester. Flere tester øker sjansen for at feil oppdages tidlig og kan rettes opp i før systemet settes i produksjon. Ved å innføre automatisk testing av et system vil endringer i fremtiden være enklere å innføre, siden en utvikler kan være mer sikker på at feil oppdages av de automatiske testene.

Kief Morris beskriver automatisk testing av infrastruktur i sin bok *”Infrastructure as Code”* slik [9]:

> *”The goal of automated testing is to help a team keep the quality of their system high by identifying errors as soon as they are made so they can be immediately fixed. A team with strong discipline around continuosily testing and fixing is able to make changes quickly and with confidence.”*

Automatisk testing kan implementeres på forskjellige måter. En av disse måtene er den smidige tilnærmingen som fokuserer på å teste ofte og på hver endring som skjer. Denne smidige metoden fungerer når testing er en del av arbeidsmetoden til utviklerne. Testing skal skje hver gang endringer blir gjort, det brukes ofte CI/CD pipelines til å utføre disse testene automatisk. Formålet er å teste endringer samtidig som de utvikles, dette gir en tidlig tilbakemelding om endringene bryter ønsket funksjonalitet. Om en test feiler, så kan feilen rettes opp i med en gang.

Test pyramiden er et begrep fra smidig programvareutvikling som beskriver forskjellige typer tester systemet skal gjennom. Den smidige test pyramiden baserer seg på automatiserte tester som kjører mot koden og systemet. Formålet er å unngå at feil skal komme til produksjonsmiljøet. Pyramiden blir delt opp slik at på bunnen er det mange enkle tester, som unit tester og statisk kodeanalyse. Etter dette kommer tester som tar lengre tid å kjøre og som krever mer ressurser som akseptansetesting og integrasjonstesting [10]. Ved å ta i bruk prinsippene fra test pyramiden inn i en CI/CD pipeline får utviklerne testet store deler av systemet i utviklingsfasen. Endret kode blir sjekket, den nye konfigurasjonen blir testet på nye provisjonerte maskiner og tester kjøres for å sjekke om tjenesten fungerer. I pipelinen blir de raske og enkle testet kjørt først, hvis disse testene ikke finner noen feil kjører pipelinen videre og høyere og mer komplekse tester blir kjørt.

2.5 Programvareutviklingsprosesser for IT-drift

Som beskrevet tidligere handler infrastruktur som kode om ta prinsipper og metoder fra programvareutvikling og bruke det i IT-drift. Disse prinsippene og metodene er godt testet og har blitt utviklet over lang tid i programmeringsmiljøet, på grunn av dette er metodene modne og klare til å bli brukt av personer innen drift.

En metode som ikke allerede har blitt nemt er versjonskontroll. Versjonskontroll tar vare på historien til filer over tid [11]. Dette kan være kodefiler, definisjonsfiler eller dokumentasjon. Ved bruk av automatisering og infrastruktur som kode er det viktig for utviklerne å kunne hente tilbake tidligere versjoner av koden de utvikler om det oppstår feil. Med versjonskontroll er det også mulighet for å se hvem som gjorde endringer på koden og utviklere kan legge til kommentarer på de endringene som er gjort. Dette gjør det lettere å finne feil om de skulle oppstå.

Versjonskontroll sammen med prosesser som CI/CD pipeline og tester som kjører automatisk ved endringer vil hjelpe utviklere med å holde orden på sin kodebase. Det hjelper med å sikre god kvalitet og effektivisering av utviklingsprosessen.

2.6 Virtuelle maskiner og containere

I en CI/CD pipeline for infrastruktur er det behov for å ha servere hvor CI/CD programvaren kjører, og for å provisjonere nye servere hvor programvaren som er under utvikling kan bli testet. For dette brukes det ofte virtuelle maskiner og containere.

Virtualisering er en teknologi som lager en virtuell versjon av en datamaskin. I stedet for å bruke maskinvaren direkte, deles maskinvaren med sin vertsmaskin. Maskinvare, lagringsenheter og nettverk kan virtualiseres. [12].

Virtuelle maskiner (VM-er) emulerer sitt operativsystem og har ingen egne fysiske komponenter. Den fysiske maskinvaren blir delt med VM-en fra en hypervisor. Hypervisorens oppgave er å kjøre flere virtuelle maskiner og isolere dem fra hverandre slik at hver VM

fungerer som en egen fysisk maskin [13].

Fordelene med å bruke VM-er over fysiske maskiner er utnyttelsen av ressursene som er tildelt. Å sette opp nye servere er trivielt og kan automatiseres. I tilegg kan mye av programvare og konfigurasjon ”bakes inn” i oppsettet av ny VM, dette gjør at oppstartstiden er veldig kort.

Containere er en metode å pakke en applikasjons kode, dens konfigurasjon og ”dependencies” i en egen kjørbar enhet [14]. Containere deler kernel med maskinen den kjører på. Containere tar derfor liten plass og kan effektivt utnytte de delte ressursene maskinen har. Docker og Kubernetes er populære produkter som kan lage og kjøre containere.

Containere passer best til å kjøre mikrotjenester, hvor hver tjeneste kjører separat som en egen container eller gruppe med like containere. Disse små tjenestene kommuniserer med hverandre og utgir sammen et større produkt. Dette er en fordel containere har over VM-er. Siden en container pakkes til en egen enhet kan den bli kjørt på flere forskjellige operativsystemer så lenge Docker tjenesten kjører, uavhengig av hva slags programvare operativsystemet har.

3 Kravspesifikasjoner

3.0.1 UML Use Cases

Nedenfor er det laget 3 UML Use Cases. Disse representerer hovedoppgavene for dette prosjektet. For å gjøre dem enklere å lese, er kun hovedaktøren nevnt i hvert Use Case.

Use case nummer: 1
Use case navn: Pipeline for å kjøre automatiserte tester
Beskrivelse: For å gjøre det lettere å teste kode, sikre at testing alltid blir gjort og at testing utføres likt hver gang. Er det ønskelig å ha en pipeline for å kunne kjøre automatiserte tester.
Aktør: Pipelinesystem
Normal hendelsesflyt:

1. Ny kode blir skrevet av driftspersonell
2. Koden pushes opp til et repository
3. Pipelinen starter med å kjøre statisk kodeanalyse for den nye koden
4. Pipelinen deployer relevant infrastruktur til SkyHiGh
5. Pipelinen kjører automatiske tester
6. Pipelinen river ned infrastrukturen den lagde i SkyHiGh.
7. Pipelinen lager et merge request til en annen branch

Variasjoner:

- Mellom punkt 5 og 6 kan det være ønskelig å kjøre mange andre typer tester også
- Til punkt 6, det er kanskje ikke ønskelig å rive ned infrastrukturen. Hvis infrastrukturen blir stående, kan driftspersonell foreta manuell testing eller inspeksjon av infrastrukturen.
- Til punkt 7, i stedet for å sende et merge request kan det være ønskelig å automatisk merge til en annen branch
- Til punkt 7, det kan være ønskelig å ikke ha med dette punktet hvis man vil ha større kontroll selv

Use case nummer: 2
Use case navn: Automatisere tester
Beskrivelse: Nye utrullinger av systemet testes i dag kun via manuell utprøving. Det er ønskelig å lage en kodebase for automatisk testing av systemet.
Aktør: Driftspersonell
Normal hendelsesflyt:

1. Noe ny kode blir skrevet ferdig
2. koden blir satt inn i et testmiljø
3. En serie med automatiske tester blir kjørt
4. Koden blir godkjent
5. koden går videre til annen testing

Variasjoner:

- Til punkt 4, koden kan bli avvist. I så fall avbrytes den normale hendelsesflyten, det gis en fornuftig feilmelding og man går tilbake til punkt 1.

Use case nummer: 3
Use case navn: Automatisk sikring av kodekvalitet
Beskrivelse: Når kode skrives er det opp til hver enkelt utvikler å sikre at det blir kjørt statisk kodeanalyse. Det er ønskelig at statisk kodeanalyse blir kjørt automatisk
Aktør: Driftspersonell
Normal hendelsesflyt:

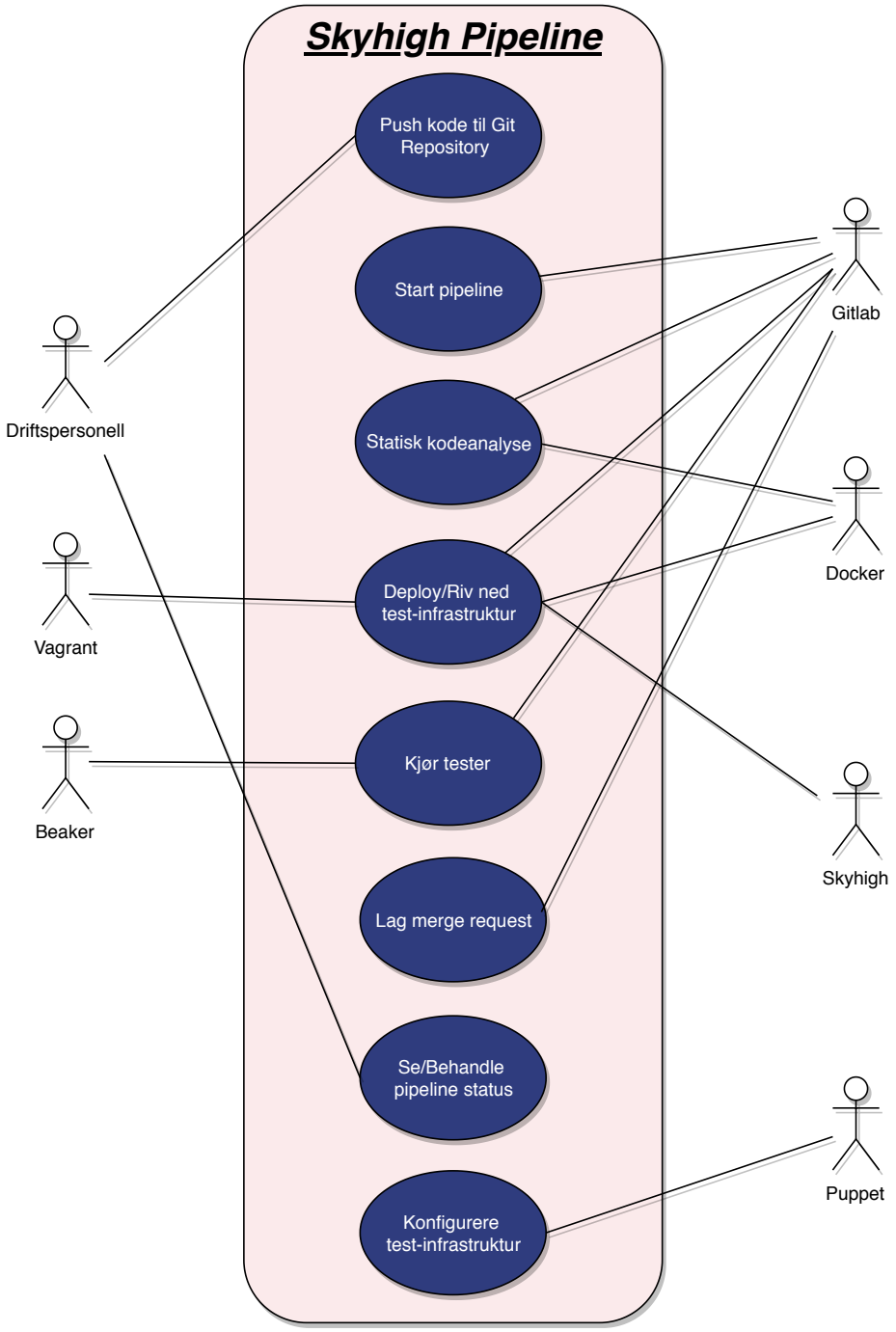
1. Ny kode blir skrevet
2. Statisk kodeanalyse blir kjørt lokalt
3. Statisk kodeanalyse godkjenner den nye koden
4. Kode pushes opp til eksternt repo
5. Statisk kodeanalyse blir kjørt eksternt
6. Statisk kodeanalyse godkjenner den nye koden
7. koden går videre til annen testing

Variasjoner:

- Til punkt 3 og 6, koden kan bli avvist. I så fall avbrytes den normale hendelsesflyten, det gis en fornuftig feilmelding og man går tilbake til punkt 1

3.0.2 UML Use Case Diagram

Se Figur 1 for et UML Use Case diagram som viser hovedtrekkene av hvilke aktører som er involvert og de funksjonene som må utføres. Den eneste menneskelige aktøren er SkyHiGhs driftspersonell. Det er de som skriver, commit-er og pusher puppetkode til et remote repository. Aktøren GitLab vil starte en pipeline. For å utføre alle oppgavene i pipelien benyttes SkyHiGh og Vagrant til deployment, puppet til konfigurasjon og Docker og Beaker til å kjøre tester.



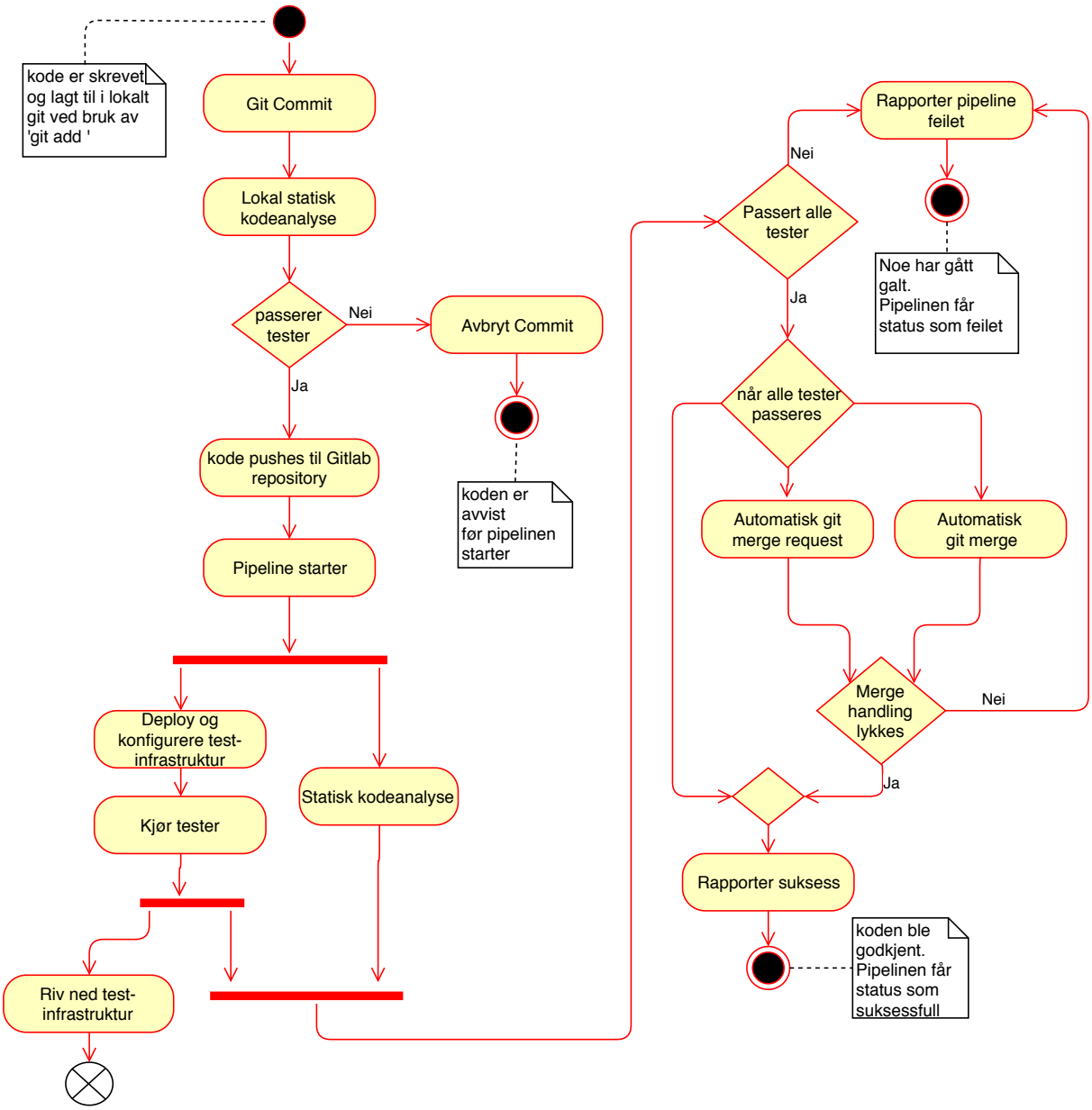
Figur 1: UML Use Case diagram

3.0.3 UML Action Diagram

Se Figur 2 for et UML Action diagram. Diagrammet viser handlingssekvenser og muligheter for systemet. Øverst til venstre er diagrammets start. Når kode commit-es til git

vil det bli kjørt lokale tester. Hvis disse testene passeres vil det bli mulig å pushe nye commits til et sentralt repo.

En pipeline vil bli startet når nye commits er blitt lagt til i repoet. Pipelinen kjører 2 handlingssekvenser som er uavhengige av hverandre. På den ene blir statisk kodeanalyse kjørt. På den andre blir en test-infrastruktur deployet og konfigurert. Deretter blir det kjørt mange tester på systemet som er satt opp. Uavhengig av testenes utfall blir test-infrastrukturen revet ned. Samtidig som det skjer rapporters testenes utfall. Hvis alle testene ikke passeres, blir pipelinen stoppet og får status som feilet. Hvis alle testene passeres kan systemet konfigureres til å gjøre 3 forskjellige ting. Det første rapporterer til pipelinen at alt er OK. Den andre lager et merge request til en annen branch. Det tredje alternativet merger 2 branches og pusher endringene til et remote git repo. Hvis alternativ 2 eller 3 lykkes så rapporteres det at alt er OK og pipelinen avsluttes med status suksessfull. Hvis ikke så rapporteres feilen og pipelinen stopper med status feilet.



Figur 2: UML Action diagram

4 Vurdering og valg av teknologier

4.1 CI/CD pipeline for SkyHiGh

Hvilke CI/CD verktøy som er valgt til å teste SkyHiGh er basert på kriterier satt av oppdragsgiver og gruppens utvalgsprosess. Utvalget av verktøy er hentet fra en blogg som lister opp 8 forskjellige kandidater [15]. I tilegg ble det brukt flere nettsøk for å forsikre om at bloggens innhold er korrekte. 6 av disse 8 kandidatene ble tatt videre til en prosess for å velge det beste verktøyet for oppgaven.

4.1.1 Kriterier for valg av CI/CD verktøy

Oppdragsgivers kriterier

Lisenser
Det er ikke aktuelt å kjøpe inn lisenser til programvare.

Puppet
Det er ønskelig at systemer og infrastruktur som settes opp skal kunne installeres og konfigurasjonsstyres av Puppet.

Utvalgsprosessen

Verkøyet som blir valgt må følge kriteriene som er satt av oppdragsgiver. Det må også kunne starte servere i openstack, slik at testene blir kjørt i dette miljøet. Det kan ikke bli brukt en CI/CD skyløsning på internett siden SkyHiGh ikke kan nås fra internett. Det var også ønskelig å se på muligheten med å kjøre tester i Docker containere.

Utvalgsprosessen bestod i første omgang i å sammenligne forskjellige CI/CD verktøy. Se vedlegg F.1 for relevant dokumentasjon.

Verktøy	Pris	Kjøres lokalt	OpenStack støtte	Docker støtte	Kan teste Puppet kode
Jenkins [16]	Gratis	Ja	Ja	Ja	Ja
TeamCity [17] [18] [19]	Lisens	Ja	Ja	Ja	Ja
Travis CI [20] [21]	Gratis for Open Source prosjekter	Nei	Bare for Enterprise lisens	Ja	Ja
Go CD [22] [23]	Gratis	Ja	Ja	Ja	Ja
GitLab CI [24] [25]	Lisens og gratis versjon	Lokalt på GitLab server	Ja	Ja	Ja
CircleCI [26] [27]	Begrenset gratis versjon	Bare for betalt versjon	Nei	-	-

Tabell 1: Sammenligning av CI/CD verktøy

Etter sammenligningsprosessen stod Jenkins, Go CD og GitLab CD igjen som kandidater til videre vurdering.

Neste sted i utvalgsprosessen var å se på populariteten av produktene. Populariteten ble målt ved bruk av en nettside som sammenligner bruken av de forskjellige verkøyene [28]. Ut i fra denne fremvisningen, viser det seg at Jenkins er det verktøyet som er mest brukt, ettefulgt av GitLab CI. Go CD er lite brukt og også lite omtalt på sider som Stack Overflow.

Det ble også diskutert valg av CI verktøy med veileder. Veileder mener GitLab er den beste løsningen. Prosjektgruppen har også tilgang til NTNUs egen GitLab installasjon.

På bakgrunn av diskusjon med veileder og av hva andre bedrifter bruker, ble det valgt å undersøke hvilke muligheter Jenkins og GitLab CI har i mer detalj.

4.1.2 Jenkins

Jenkins er et Open Source prosjekt med lang fartstid. Det er nok det mest populære CI verktøyet per dags dato [16]. Med en estimert brukermasse på over 1,5 millioner. Prosjektet er i kontinuerlig utvikling med siste commit til dens GitHub konto, for kun et par dager siden [29].

Jenkins kan brukes til testing av veldig mange språk ettersom den har et stort antall med plugins. Systemet er også kompatibelt med veldig mange versjonskontrollsystemer. Jenkins er avhengig av plugins for at det kan brukes til å lage en pipeline. Systemet ble opprinnelig laget for å bruke CI verktøy for programmering [16]. Dette prosjektet trenger en pipline som skal brukes til infrastruktur som kode. Det er ikke sikkert at alle pluginsene som må benyttes eksisterer, eller fungerer som de skal. Det er derfor en risiko for at Jenkins ikke kan brukes til dette prosjektet.

Jenkins kan brukes til alle CI/CD stadier [16]. Til tradisjonell programvareutvikling kan det brukes til å hente kode fra et versjonskontrollert repository som GitHub/GitLab, kjøre en build av koden, kjøre tester, lage en release og deploye produktet til et miljø [30].

For dette prosjektet er det aktuelt å bruke det til statisk kodeanalyse, deployment og testing. Prosjektgruppen forventer at det er god dokumentasjon på Jenkins generelt, men ikke nødvendigvis så mye på de individuelle plugins-ene som sannsynligvis må brukes for dette prosjektet.

For å lage en Jenkins pipeline trengs det en Jenkinsfile. Dette kan gjøres på 2 forskjellige måter [31]. Den første er ved å bruke en deklarativ syntaks som beskriver hva som skal skje. Den andre er å bruke en scriptet syntaks som likner mer på tradisjonell programmering. Prosjektgruppen forventer at den deklarative er enklere å lære seg. Men siden det er et nyere featurer i Jenkins så er det ikke sikkert at det har like stor funksjonalitet som den scriptede metoden. Hvis Jenkins skal brukes, må det i så fall gjennomfører undersøkelser og tester for å avklare hva som egner seg best.

Jenkins støtter et distribuert oppsett med master og slave noder [32]. Disse slave nodene må ha Jenkins agent installert som kan kommunisere med Jenkins master. Dette kan komme til nytte hvis det kjøres mange tunge tester/builds, som gjør at pipelinen går tregt. Dette uføres ofte med bruk av SSH på Linux noder, men det er mulig å gjøre det mer manuelt ved bruk av egne scripts.

Prosjektgruppen sjekket flere anmeldelser fra nettet, for å kunne lære av andre sine erfaringer. Den første nettsiden har den 18/02/2019, 217 anmeldelser av Jenkins. Totalskåren er 4.3 av 5. [33] Den andre nettsiden har den 18/02/2019, 197 anmeldelser av Jenkins. Totalskåren er 8.4 av 10 [34]. Denne anmelderen mener at jenkins har så mange konfigurasjonsmuligheter at du kun bør bruke den hvis du er sikker på at et mer strømlinjeformet CI/CD verktøy ikke kan brukes [35]. En annen anmelder mener at brukergrensesnittet er tidvis vanskelig [36].

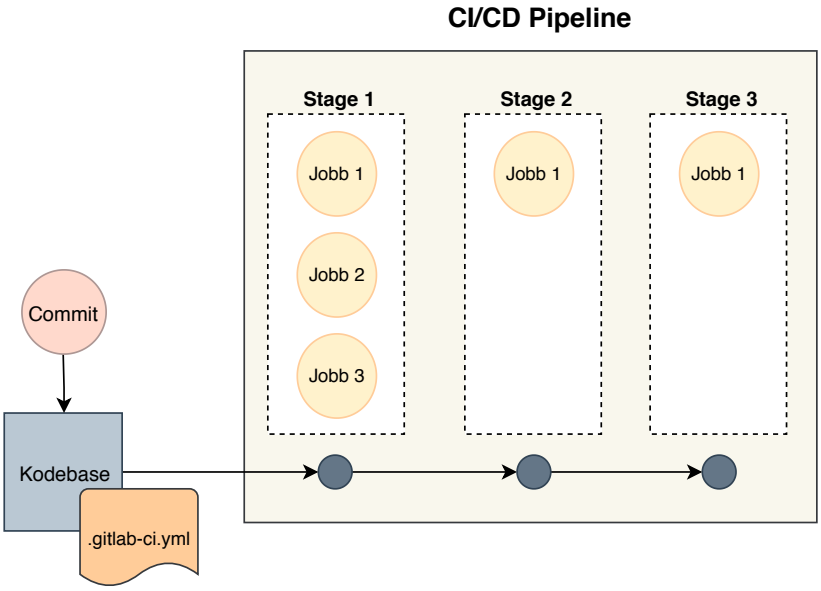
Jenkins er en aktuell kandidat for vårt prosjekt fordi det er gratis, har mange plugins og er et mye brukt system som forbedres kontinuerlig.

4.1.3 GitLab CI

GitLab platformen tilbyr sitt eget CI/CD verktøy kalt GitLab CI. GitLab CI består av flere komponenter som til sammen utgjør CI/CD verktøyet. De viktigste komponentene er pipeline filen som beskriver hele pipelinen. Dette er en yaml basert fil kalt .gitlab-ci.yml, som ligger i roten av repo-et. Det er i denne filen hele pipelinen blir definert. En pipeline er en samling av jobber som blir delt opp i forskjellige deler (stages). Stages i en pipeline er till for å logisk dele opp forskjellige typer jobber, disse jobbene er en samling av instruksjoner som GitLab runner-er skal kjøre. Jobber i en stage kjører parallelt. Hvis en jobb i en pipeline stopper vil også pipelinen stoppe. Alle jobber i en stage må fullføre før neste stage blir påbegynt, hvis ikke annet er beskrevet i pipeline filen [37].

En annen viktig komponent i GitLab CI er GitLab runners. Dette er applikasjonen som kjører jobbene definert i pipelinen. En GitLab runner kan bli installert og håndtert på egne skyplattformer som openstack, men må registreres mot GitLab repo-et for å kunne utføre jobbene i en pipeline.

Eksempel på en GitLab CI/CD pipeline:



Figur 3: Eksempel på en GitLab CI/CD pipeline

Hver gang kode blir commit-ed til repo-et blir en ny pipeline startet basert på innholdet i pipeline filen *.gitlab-ci.yml*. I denne filen kan det være spesifisert flere stager, hvor hver stage kan ha flere jobber. Stagene i pipelinen kjøres sekvensielt som beskrevet i figuren. En GitLab runner henter ned innholdet i repo-et og begynner å kjøre disse jobbene sekvensielt.

4.1.4 Valgt CI/CD verktøy

Vi har valgt å bruke GitLab CI for denne oppgaven.

Jenkins er et veldig fleksibelt verktøy, men med høyere vanskelighetsgrad enn GitLab. Dette er den største grunnen til at Jenkins ble valgt bort. Vi har også mulighet til å bruke NTNUs GitLab server og trenger da ikke å implementere en egen CI servere for oppgaven. På grunn av dette kan mer tid bli satt av til å lage de automatiske testene og pipelinen for oppgaven, i stedet for implementasjon av serveren.

Siden den største delen av oppgaven er å skrive en pipeline-fil er lesbarheten for denne filen viktig. Ved å bruke GitLab CI vil pipelinen ble skrevet i yaml syntax. Yaml syntax er enklere å lese enn Jenkins sin groovy syntax. Dette er viktig da pipelinen beskriver hele prosessen fra når en ny commit kommer inn, til testingen er ferdig.

4.2 Valg av GitLab Runner executor

En GitLab runner kan konfigureres til å kjøre på mange måter [38], blant annet i moduser som Shell, Docker, SSH, kubernetes og mer. Basert på litteraturstudie foretatt så langt, faller valget på enten Shell, eller Dockermodus.

Shell-moduset fungerer som et bash shell, den kan kjøre alle kommandoer som i et vanlig Linux bash shell. Det er også mulig å bygge og kjøre containere i dette moduset hvis Docker tjenesten er installert på GitLab runneren.

Docker-moduset er laget slik at det kun kan bruke ferdige Docker images, det er altså ikke mulighet til å bygge eller oppdatere ett Docker image ved oppstart av pipelinen. Med en GitLab installasjon er det mulig å ha ett eget privat Docker-registry, hvis systemadministratorene har aktiver det [39]. På dette tidspunktet er det ikke aktivert et Docker-registry på GitLab installasjonen som brukes i prosjektet.

For å bruke Docker-moduset må det derfor brukes et offentlig registry som Docker Hub istedet. Dette kan bli et sikkerhetsproblem i det langsiktige perspektiv. En arbeidsprosess basert på en executor i Docker-modus vil si at det må lages og deretter pushes egne images til Docker hub før de kan kjøres i en pipeline. På dette tidspunktet vet vi ikke hvordan Docker imagene skal se ut, så vi må sannsynligvis lage og push nye images ofte. På grunn av dette vil bruken av Docker-modus være mer arbeidskrevende en bruk av Shell-modus. En annen ulempe er at du ikke har full kontroll over hvilke Docker-engine som kjører containerne dine [40]. Docker-moduset har den fordelen at den kan starte og kjøre tjenester (f. eks. MySQL) som alltid vil bli laget før dine egne containere starter.

Valget av executor ble Shell-modus ettersom dette er mer fleksibelt og mindre arbeidskrevende.

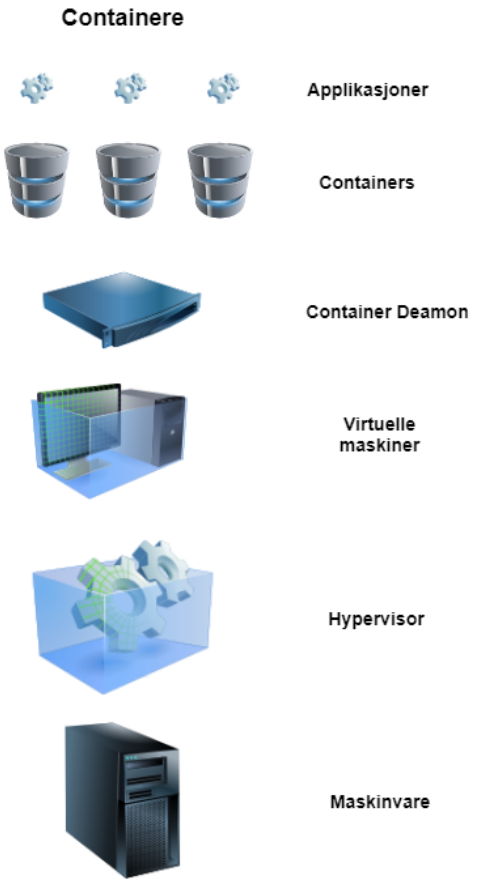
4.3 Container vs. VM til å kjøre tester

Applikasjoner må kjøre oppå et operativsystem. Det er mange måter dette kan gjøres på. På PC-ene folk har hjemme kjører operativsystemet direkte oppå maskinvaren. Dette er greit hvis man er sikker på at det operativsystemet man har kan kjøre alle applikasjonene en trenger. Men hvis man har et behov for å kjøre et sett med applikasjoner som kjører på operativsystem A, og et sett med applikasjoner som kjører på operativsystem B, samtidig som man ikke ønsker å ha flere sett med PC-er. Så kan man løse dette på 2 forskjellige måter. Den første måten er ved å kjøre en eller flere VM-er oppå en hypervisor. Den andre måten er å kjøre en eller flere containere ovenpå operativsystemet. Og siden VM-er har sine egne operativsystemer så kan containere kjøres inni VM-er også.

Containere

Containere derimot er en noe nyere teknologi. I motsetning til VM-er så trenger ikke hver container sin egen instans av et operativsystem. Dette er fordi containere deler kernelen i operativsystemet med vertsoperativsystemet containeren kjører i [41]. Dette gjør at hver container tar mindre plass enn en VM. Selv om det finnes flere typer containerteknologier, så er det Docker som er den mest populære. Ofte brukes ordet Docker som et synonym for containere. Det finnes også flere støtteteknologier til containere som gjør at de blir enda mer praktiske å bruke. Docker Compose gjør det lett å starte mange Docker containere i riktig rekkefølge eller mange samtidig. Docker Swarm og Kubernetes gjør at flere containere kan administreres på en bedre måte.

For å kjøre en pipeline med all den tiltenkte funksjonaliteten som er ønskelig for dette prosjektet, må det benyttes en rekke teknologier. Mange av disse teknologiene er applikasjoner som må kjøres i enten VM-er og/eller containere. I de neste seksjonene vil det bli diskutert diverse fordeler, ulemper og konsekvenser av å velge å benytte enten VM-er eller containere.



Figur 4: Applikasjoner i Containere, operativsystem og maskinvare

4.3.1 Pipeline med containere vs. pipeline med VM-er

Når applikasjoner kjøres i containere så er de mer portable enn når de kjøres i VM-er. Hvis det skulle bli ønskelig å bytte ut det underliggende operativsystemet fra for eksempel Linux Ubuntu til Linux Red Hat. Så vil containerne fremdeles fungere på samme måte. Om det er ønskelig å bytte ut Linux Ubuntu til Windows 10, er dette også mulig, men krever mer arbeid [42]. Containere kjører applikasjoner i et miljø som er tilstrekkelig isolert til det er ufarlig å oppgradere det underliggende operativsystemet. Mange applikasjoner er avhengig av diverse biblioteker for å kunne fungere. Ofte kan flere applikasjoner være avhengig av forskjellige versjoner av samme bibliotek. Å ha forskjellige versjoner av samme bibliotek installert samtidig kan også medføre en del problemer. For eksempel kan en applikasjon bruke en funksjonalitet fra et bibliotek uten først å sjekke om det er riktig versjon av biblioteket. Dette kan resultere i at uforutsette ting skjer, hvilket applikasjonen

ikke er laget for å håndtere. Ved å kjøre en applikasjon i containere kan man sikre at slike konflikter ikke oppstår, dette gjør det tryggere å kjøre flere containere på samme server.

Den kanskje største fordelen fra et GitLab CI pipeline perspektiv, er at du kan installere hva som helst. GitLab runneren har ikke behov for sudo rettigheter og modifikasjoner på filsystemet kan gjøres uten at det oppstår konflikter på den underliggende serveren. For eksempel observerte prosjektgruppen at hvis GitLab runneren hadde sudo rettigheter så kunne den lage filer som var eid av og kun kunne slettes av sudo. Men når en ny pipeline skal kjøres så starter GitLab runneren med å slette alle rester etter forrige gang pipelinen kjørte, denne slettingen ble utført uten sudo rettigheter. Hvis det fantes filer som GitLab runneren ikke hadde tillatelse til å slette, så ville pipelinen feile. En bruker med sudo rettigheter måtte så logge seg inn på serveren for så å manuelt slette alle filene som GitLab runneren ikke kunne slette. Når dette skjer så forårsaker det mer arbeid for de som skal drifte pipelinen. Dette gjør løsningen mindre robust siden den krever med vedlikehold. Dette er begge punkter som oppdragsgiver mener ikke er ønskelig i henhold til teksten om avgrensinger og rammer i kapittel 1.4. Containere gjør det også lettere å sikre at miljøet pipelinen utvikles i og det miljøet det skal brukes i avviker mindre fra hverandre.

En ulempe med å bruke containere er at når man skal lage dem så er det ikke alltid slikt at alt fungerer på samme måtte som det ville gjort når man installerer og kjører applikasjoner rett på en VM. Dette resulterer ofte i at det kan være vanskeligere å lage containere. Å kjøre en pipeline med applikasjoner som kjøres i VM-er istedenfor containere medfører at applikasjoner må installeres på forhånd. Enten med bootscript, systemkonfigurasjonsverktøy eller manuelt via kommandolinje. Hvis man skal installere ting med bootscript, systemkonfigurasjonsverktøy eller manuelt via kommandolinje før pipelinen kjører, så medfører dette et ekstra skritt i arbeidsprosessen. Ettersom det først må sikres at GitLab runnerene har all den programvaren den trenger før pipelinen kjøres. Skulle man derimot installere påkrevet programvare under kjøring av pippelinen, vil dette kreve at GitLab runneren har rettigheter til å installere eller oppdatere programvare. Hvilken kan medføre problemer som ble nevnt ovenfor. Hvis man derimot installerer programvare manuelt så medfører dette at en veldig tidkrevende prosess med betydelig risiko for feil. Og manglende mulighet til å sikre at like oppgaver er løst på en standardisert måte. Hvilket kan resultere i snowflake servere.

For å gjøre VM-er tryggere å bruke så kan man ha en server for hver liten oppgave, men dette krever flere maskiner som må driftes. Hvilket krever mer vedlikehold og mer ressurser.

4.3.2 Valgt testing plattform

Det er prosjektgruppens mening at den beste løsningen for kjøring av pipeline relatere programvare bør så langt det er mulig kjøres i containere. De viktigste argumentene for dette er at pipelinen blir mer robust.

4.4 Verktøy til å kjøre tester

En CI/CD pipeline for infrastruktur som kode er avhengig av støtteverktøy til å sette opp og konfigurere test-maskiner i en skyplattform, installere puppetkoden som ligger i repo-et og programvare som puppet er avhengig av. Det er også behov for verktøy som kan teste og validere denne puppetkoden og om programvaren fungerer etter installasjon.

I kapittel 6 - Implementasjon kan det leses om de forskjellige metodene og verktøyene prosjektgruppen har forsøkt å bruke i pipelinen. Det ble til slutt valgt å bruke Vagrant og Beaker til å sette opp, konfigurere og teste maskinene.

4.4.1 Vagrant

Vagrant er et verktøy for å bygge og håndtere VM-er hvor fokuset er på automasjon. Vagrant har mulighet for å provisjonere maskiner direkte i openstack, verktøyet har også støtte for å kjøre bootscripts etter oppstart av disse maskinene [43]. Maskiner som skal startes kan defineres i en egen fil kalt ”Vagrantfile”. I denne filen blir maskinene definert med navn, hvilke nettverk de tilhører, OS de skal kjøre, sikkerhetsgrupper, bootscripts, osv. Dette gjør provisjonering av maskiner til en enkel prosess som er repeterbar og passer derfor godt i CI/CD pipeline.

Vagrant brukes i dette prosjektet til å starte forhåndsdefinerte servere i openstack og til enkel førstegangskonfigurasjon ved hjelp av et bootscript, dette er definert i en Vagrantfile som ligger i repo-et. Det er på disse keystones puppetkode og støtteservere blir testet.

4.4.2 Beaker

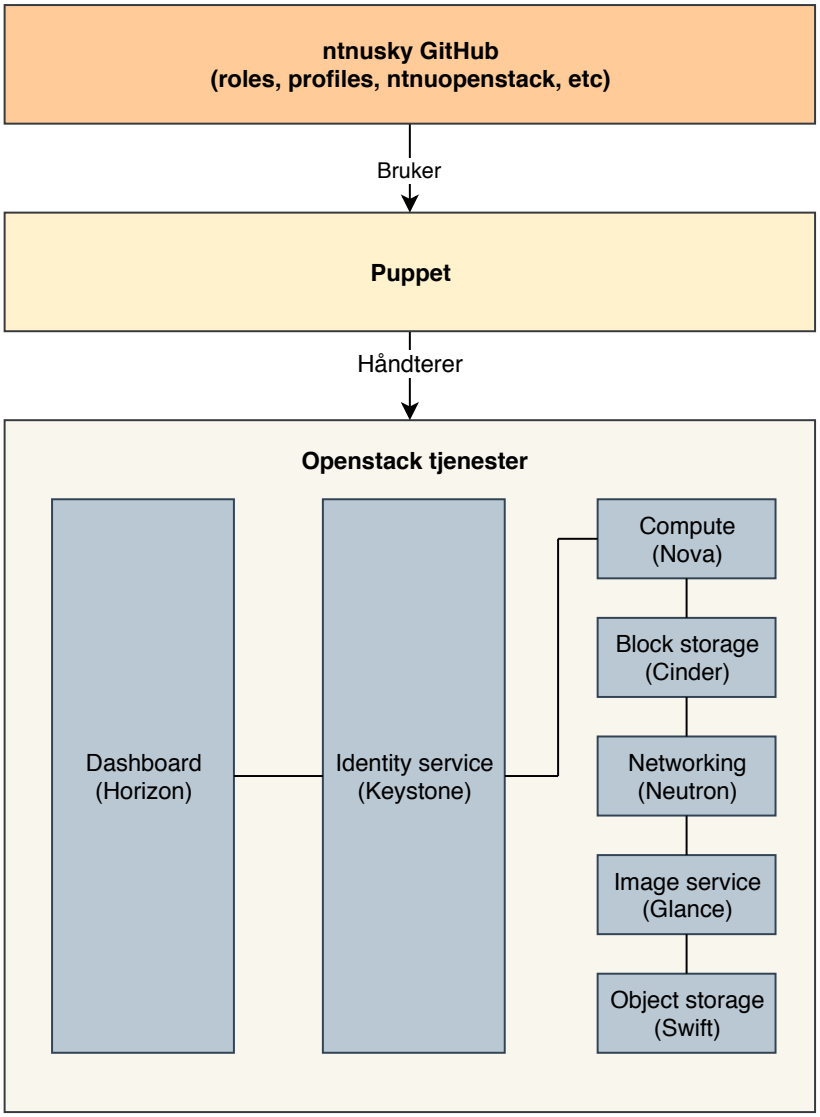
Beaker er et verktøy som blir brukt for akseptansetesting av puppetkode. Beaker kjører tester skrevet i Ruby, men har et eget ”Domain-Specific Language” (DSL) som gir tilgang til spesielle kommandoer for puppet installasjon og testing. Beaker er lagd av samme selskap som puppet og er det verktøyet som er mest brukt til testing av puppetkode [44]. Maskinen Beaker skal kjøre mot blir definert i en egen fil kalt ”HOSTS file”. Denne filen inneholder informasjon om hvilke maskiner Beaker skal SSH-e seg inn mot og hvordan Beaker skal autentisere seg mot maskinene. Roller blir tilegnet hver maskin, som senere blir brukt for å gi de forskjellige maskinene forskjellig konfigurasjon, og for å kjøre forskjellige tester på dem.

Beaker brukes i dette prosjektet til å installere programvare på maskinene som blir startet av Vagrant. Sjekke om maskinenes konfigurasjon er korrekt og installere keystone modulen fra en puppetkatalog i et puppet master/agent oppsett.

5 System arkitektur

5.1 SkyHiGh arkitektur

SkyHiGhs kode ligger i forskjellige GitHub repo-er som er styrt og håndtert av NTNU. NTNU har også en Puppetfile som inneholder alle puppetmoduler SKyHiGh er avhengig av. Puppet bruker disse repo-ene til å rulle ut og håndtere openstack implementasjonen. En illustrasjon av hvordan NTNUs GitHub repo-er, puppet og openstack henger sammen er vist nedenfor i Figur 5.



Figur 5: SkyHiGh arkitektur

Openstack består av flere tjenester som til sammen utgjør en fullstendig skyplattform. En kort beskrivelse av hver tjeneste er hentet fra openstacks egen beskrivelse [45]:

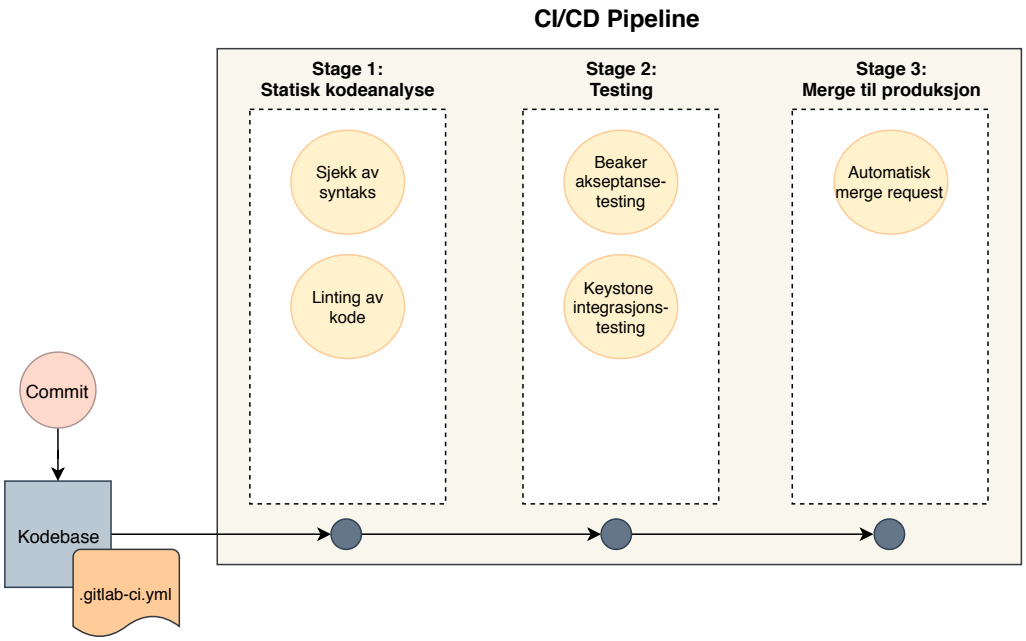
- **Dashboard (Horizon)**
  Web basert grensesnitt for interaksjoner mot openstack tjenestene.
- **Identiy service (Keystone)**
  En API for autentisering av klienter. Støtte for å gi tilgang til ressurser for brukere, grupper, domener, prosjekter. Støtte for å koble seg mot MySQL, LDAP (Microsoft Active Directory, m.m.).
- **Compute (Nova)**
  Provisjonering av VM-er.
- **Block storage (Cinder)**
  Allokering av data, kan brukes som volum i VM-er.
- **Networking (Neutron)**
  Oppretting av nettverk og nettverksenheter i et virtuelt miljø.
- **Image service (Glance)**
  Lagring av ”images” til forskjellige VM-er.
- **Object storage (Swift)**
  Objekt/blob lagring.

Prosjektgruppen har i dette prosjektet kun fokusert på utrulling og testing av keystone tjenesten/modulen.

5.2 CI/CD pipline arkitektur

CI/CD pipelinen har 3 jobber som skal utføres, disse 3 jobbene er delt opp i 3 stages. Stagene blir kjørt i rekkefølge fra stage 1, til stage 3. Hvis en stage feiler, vil pipelinen avlsluttes med en feilkode og neste stage kjøres ikke. Om alle stagene fullfører uten feil vil pipelinen avsluttes uten feil. De raske og mindre kompliserte testene kjører derfor først i pipelinen, slik at disse feilene raskt kan oppdages og rettes opp i. De forskjellige stagene er beskrevet i mer detalj senere i kapittel 6 - Implementasjon.

Figur 6 gir en illustrasjon på hvordan kode går gjennom de forskjellige stagene i pipelinen, og hvilke oppgave hver stage har.



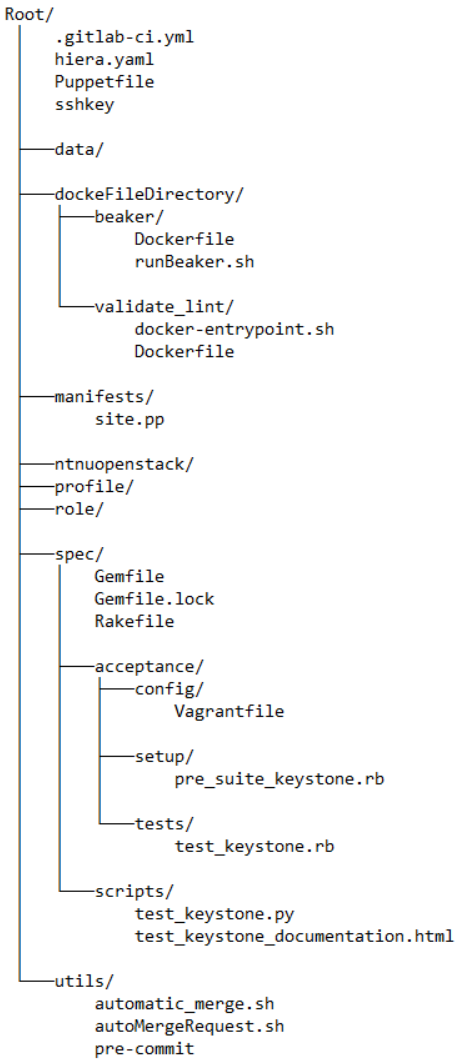
Figur 6: GitLab CI/CD pipeline

Beskrivelse av figuren:
Når kode blir commitet til kodebasen startes pipeline jobbene som beskrevet i filen *.gitlab-ci.yml*. Stage 1 kjører statisk kodeanalyse på alle puppetfiler som har blitt endret siden forrige commit. Hvis ingen feil blir oppdaget i dette stadiet startes stage 2. Denne stagen kjører akseptanse-, og integrasjonstesting av keystone tjenesten. Hvis ingen feil blir oppdaget startes stage 3. Stage 3 lager et merge request til en ny branch. Dette kan være et merge request direkte til produksjonsmiljøet, eller til en branch for manuell testing.

6 Implementasjon

6.1 GitLab repository

GitLab repo-et har koden vi bruker i CI/CD pipelinen for prosjektet. Vi har også hentet inn repo-ene *ntnuopenstack, profile, role* og filene *hiera.yaml og Puppetfile* fra ntnusky (https://github.com/ntnusky). Repo-ene fra ntnusky er i våres repo lagret som mapper.



Figur 7: GitLab repo struktur

Filen .gitlab-ci.yml er pipeline filen som beskriver hele CI/CD pipelinen. Mappen data inneholder all Hiera data som brukes for å konfigurere keystone modulen, disse har vi fått fra oppdragsgiver. dockerFileDirectory inneholder Dockerfiles og entrypoint script

for de containerne som kjører i pipelinen.

I *spec-mappen* ligger testfilene som brukes til akseptanse,- og integrasjonstesting av keystone. I undermappen *acceptance/config/* ligger Vagrantfilen som Vagrant bruker til å installere og starte maskiner i openstack. I *acceptance/setup/* mappen ligger Beaker scriptet som installerer og konfigurerer test stacken for keystone. I *acceptance/tests/* mappen ligger Beaker scriptet som kjører akseptansetestene. I *scripts-mappen* ligger python scriptet som kjører integrasjonstestene.

Mappen *utils* har 2 ”merge scripts” som brukes i siste del av pipelinen. En som utfører automatisk merge til en branch og en som utfører et merge request til en branch. Filen pre-commit er et script som kan brukes lokalt til å utføre linting og validering av puppetkode før det commites til repo-et.

6.2 Utprøving av GitLab CI

Prosjektgruppen gjennomførte flere forsøk for å avdekke hvordan GitLab CI kan brukes. Denne seksjonen består av mange underseksjoner som viser forskjellige metoder å kjøre en pipeline på. På starten er det en henvisning til relevant tekst i vedlegg som inneholder alle filer og konsoll-output fra forsøket.

Hver underseksjon starter med en tekstlig forklaring. Etterfulgt av filen som styrer pipelinen og en tekst som forklarer filen. Til slutt er det en oppsummering av resultatet.

Testene som ble kjørt testet kodefilen for feil med validering av syntaks og linting. Denne filen kan sees under, figur 8, og henvises også til som puppetfilen i seksjonene under.

puppetHelloWorld.pp 176 Bytes

```
# Veldig enkel puppet fil
class helloworld {
  file { '/etc/motd':
    owner   => 'root',
    group   => 'root',
    mode    => '0644',
    content => "hello, world!\n",
  }
}
```

Figur 8: Kodefilen som ble brukt til testing

Denne kodefilen hadde følgende tilstander under testerne:

1. Uten noen feil av typen error eller warning
2. Med syntaks feil av typen error
3. Med linting feil av typen error
4. Med linting feil av typen warning

For at metoden skal kunne brukes så må pipelinens tilstand for hver respektive punkt ovenfor være:

1. Pipeline suksessfull
2. Pipeline feiler
3. Pipeline feiler
4. Pipeline suksessfull

Metode 1

Se vedlegg E.1 for relevant dokumentasjon.

Flerstegs pipeline med containere og kjøring av tester direkte i containeren. I første stadiet bygges det et Docker image. Deretter kjøres tester direkte inne i containeren.

Til denne metoden er det mulig å splitte opp jobbene over 2 GitLab runnere. Hvilket kan være nyttig hvis hver test tar litt tid. Valg av GitLab runner og om testene kjøres på en eller flere GitLab runnere er overlatt til GitLab å bestemme.

.gitlab-ci.yml 839 Bytes

```
1  stages:
2      - build
3      - test
4
5  build1:
6      stage: build
7      script:
8          - echo "Building validate docker image"
9          - docker build --cache-from validate:latest --tag validate:latest dockeFileDirectory/validate/.
10
11 test1:
12     stage: test
13     dependencies:
14         - build1
15     script:
16         - echo "Validating puppet code"
17         - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest
18
19
20 build2:
21     stage: build
22     script:
23         - echo "Building lint docker images"
24         - docker build --cache-from lint:latest --tag lint:latest dockeFileDirectory/lint/.
25
26 test2:
27     stage: test
28     dependencies:
29         - build2
30     script:
31         - echo "Linting puppet code"
32         - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest
33
34
```

Figur 9: GitLab pipeline fil for metode 1

Ovenfor er filen som styrer hva GitLab CI pipelinen skal gjøre.

Fra linje 1-3 blir det erklært at det skal være 2 stages. Det første staget *build*, er byggestadier hvor Docker imagene blir bygget basert på Dockerfilen prosjektgruppen har laget. Hvis alt gå bra i dette stadiet, vil Gitlab CI gå videre til det andre staget *test*, som så kjører en test. Hvis alle stadiene lykkes så avsluttes pipelinen.

På linjene 5 og 20 er det 2 forskjellige jobber som utføres i *build* staget. Og på linjene 11 og 26 er det tilsvarende for *test* staget. Alle jobbene starter men et unikt navn,

dette er viktig ettersom det skal henvises til senere. På linjene 6, 12, 21 og 27 erklæres det at denne jobben skal gjøres i et gitt stage.

Linjene 13 og 28 erklærer en yaml array med alle jobber som denne jobben er avhengig av at har blitt utført suksessfullt før denne jobben kan starte. Det vil si at jobben *test1* er avhengig av jobben *build1*. Og tilsvarende for jobbene *test2* og *build2*

På linjene 7, 15, 22 og 30 starter selve script delen, det er her det blir definert hva denne jobben skal gjøre. Alle script delene starter med en debug melding på linjene 8, 16, 23 og 31.

I linjene 9 og 24 blir det bygget Docker images fra Dockerfilen. Argumentet *–cache-from* betyr at det skal gjenbrukes så mye som mulig av imaget som ble laget forrige gang testen kjørte. Dette resulterer i en betydelig reduksjon i byggetiden. Mens argumentet *–tag* betyr at Docker images som lages nå skal kunne identifiseres ved bruk av et navn (*tag*) som blir gitt. Det siste argumentet er relativ path til Dockerfilen.

I linjene 17 og 32 blir det laget og startet Docker containere. Argumentet *-v* erklærer et delt volum mellom containeren og det underliggende operativsystemet. Altså er innholdet i den gitte mappen tilgjengelig for begge. I dette tilfellet brukes denne mappen til å oppbevare en puppetfil som skal testes. Grunnen til at denne filen blir delt under oppstart av containeren istedenfor å kopiere den inn i image under bygging er fordi det reduserer byggetiden og gjør containeren mindre. Det siste argumentet er navnet på Docker images som skal brukes for å lage containeren.

Resultatet av denne metoden var at:

1. Pipeline var suksessfull når puppetfilen var uten noen feil av typen error eller warning.
2. Pipeline feiler når puppetfilen hadde syntaks feil av typen error.
3. Pipeline feiler når puppetfilen hadde linting feil av typen error.
4. Pipeline var suksessfull når puppetfilen hadde linting feil av typen warning.

Basert på disse resultatene så kan Docker containere brukes i Gitlab CI. Pipelinen klarer å fange opp feilmeldinger korrekt når de kommer fra containerene.

Metode 2

Se vedlegg E.2 for relevant dokumentasjon.

I denne metoden er det kun en container som kjører alle testene via at entrypoint-script.

.gitlab-ci.yml 526 Bytes

```
1  stages:
2      - build
3      - test
4
5  build:
6      stage: build
7      script:
8          - echo "Building docker images"
9          - docker build --cache-from checkcode:latest --tag checkcode:latest dockeFileDirectory/.
10
11 test:
12     stage: test
13     dependencies:
14         - build
15     script:
16         - echo "Validating and linting puppet code"
17         - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
18         - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest linter
```

Figur 10: GitLab pipeline fil for metode 2

Fra linje 1 til 3 erklæres det 2 pipeline stages, *build* og *test*. Linje 5 og 11 er unike navn på jobber. Linje 6 og 12 forteller hvilke stage hver jobb tilhører. Linje 13-14 sier at denne jobben er avhengig av en annen jobb. Dette gjør at jobben *build* alltid kjøres før jobb *test*. Og at jobben *test* alltid kjører på samme GitLab runner som jobben *build*. På linjene 8 og 16 blir det skrevet ut debug meldinger.

På linje 9 blir det bygget Docker images fra Dockerfilen. Argumentet *–cache-from* betyr at det skal gjenbrukes så mye som mulig av imaget som ble laget forrige gang testen ble kjørt. Mens argumentet *–tag* betyr at Docker images som lages nå skal kunne identifiseres ved bruk av et navn (*tag*) som blir gitt. Det siste argumentet er relativ path til Dockerfilen.

På linje 17-19 blir det laget og startet Docker containere. Argumentet *-v* erklærer et delt volum mellom containeren og det underliggende operativsystemet. argumentet *-t* er navnet på Docker images som skal brukes for å lage containeren. Det siste argumentet på linje 17 og 18 forteller containerens enrypoint-script om den skal utføre syntaks validering, parsing. Eller lintingsjekk, linter.

Resultatet av denne metoden var at:

1. Pipeline var suksessfull når puppetfilen var uten noen feil av typen error eller warning.
2. Pipeline feiler når puppetfilen hadde syntaks feil av typen error.
3. Pipeline feiler når puppetfilen hadde linting feil av typen error.

4. Pipeline var suksessfull når puppetfilen hadde linting feil av typen warning.

Basert på disse resultatene så kan containere med entrypoint-script brukes i GitLab CI. Pipelinen klarer å fange opp feilmeldinger korrekt når de kommer fra containerene.

Kommentar

Det som er viktigst å få med seg her er at pipelinen feiler når bash kommandoer returnerer en feilkode. Dette er relevant å vite når man skriver scripts. Det er altså ikke nødvendig å ha *if* tester for å sjekke returverdier for så å manuelt kalle exit 1.

Metode 3

Se vedlegg E.3 for relevant dokumentasjon.

I dette forsøke ble det bruk Docker Compose til å starte flere containere samtidig. Containerene kjører så tester i henhold til det som ble spesifisert i seksjonen 6.2 - Utprøving av GitLab CI.

.gitlab-ci.yml 238 Bytes

```
1  stages:
2      - test1
3
4  test:
5      stage: test1
6      script:
7          - echo "testing 1"
8          - cd dockeFileDirectory/
9          - docker-compose up --build
10
11 after_script:
12   - cd dockeFileDirectory/
13   - docker-compose down
14
15
```

Figur 11: GitLab pipeline fil for metode 3

Ovenfor er filen som styrer hva GitLab CI pipelinen skal gjøre.

Linjene 1-2 erklærer et stage *test1*.
Linje 4 erklærer et unikt navn på en jobb.
Linje 5 forteller hvilket stage jobben skal gjøres i.
Linjene 6-9 er selve script delen, det er her man definerer hva denne jobben skal gjøre.

Linjene 11-13 er et globalt *after_script*. Dette er et script som alltid kjører etter alle jobber i alle stages. Dette brukes til å sikre at det blir gjort en opprydding etter at en jobb er kjørt.

Linje 7 er en debug melding. På linje 8 flyttes workingdirektory til en mappe som inneholder en Docker Composefil. Dette gjøres fordi den påfølgende kommandoen på linje 9 krever at man står i en mappe som inneholder en Docker Composefil for å fungere. På linje 9 brukes docker-compose til å bygge 2 containere som så vil utføre syntaks-validering og linting sjekk på en enkel puppetfil. Argumentet *–build* må brukes for å sikre at docker-compose alltid bygger containere basert på den nyeste versjonen av en Dockerfil. På linje 13 stoppes containerene.

Resultatet av denne metoden var at:

1. Pipeline var suksessfull når puppetfilen var uten noen feil av typen error eller warning.
2. Pipeline var suksessfull når puppetfilen hadde syntaks feil av typen error.
3. Pipeline var suksessfull når puppetfilen hadde linting feil av typen error.
4. Pipeline var suksessfull når puppetfilen hadde linting feil av typen warning.

Basert på disse resultatene så kan ikke Docker Compose brukes i GitLab CI. Pipelinen klarer ikke å fange opp feilmeldinger korrekt fra containerene.

Metode 4

Se vedlegg E.4 for relevant dokumentasjon.

Til denne metoden blir det kjør en pipeline på et stage, men det kjøres en script før stagen blir kjørt. Tester blir kjørt direkte fra pipelinefilen. Dette er den eneste metoden som ikke bruker containere.

.gitlab-ci.yml 644 Bytes

```
1  stages:
2      - test
3
4  # Installer programmvare
5  before_script:
6      - sudo apt-get update
7      - sudo apt-get install wget -y
8      - tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXXX) || exit 1
9      - wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb
10     - sudo dpkg -i "$tempdeb"
11     - sudo apt-get update
12     - sudo apt-get install puppetserver -y
13     - sudo apt-get install puppet-lint -y
14
15 # Kjør tester
16 test:
17     stage: test
18     script:
19         - echo "Validating and linting puppet code"
20         - /opt/puppetlabs/bin/puppet parser validate puppetFiles/*
21         - puppet-lint --no-autoloader_layout-check puppetFiles/*
```

Figur 12: GitLab pipeline fil for metode 4

Linje 1 og 2 erklærer at det kun er 1 stage i pipelinen. Linje 5-13 er et globalt *before_script* som alltid kjører før alle stages. Her blir all påkrevet programvare installert før hoved-scriptet skal kjøre. På linje 7 installeres *wget*. Dette er et verktøy som blir brukt til å laste ned annen programvare. Linjene 8-12 installerer riktig versjon av *puppetserver*. Den brukes til å kjøre syntaks validering av puppetkode. Linje 13 installerer *puppet-lint*. Den brukes til linting-sjekk av puppetkode.

Når programvare skal installeres via kommandolinje så blir alltid kommandoen *apt-get* med parametere *-y* brukt. *-y* sier at under installasjon så skal eventuelle installasjonsvalg svares med yes/ja. Kommandoen *apt-get* blir brukt istedenfor apt. Dette er fordi apt er en nyere kommando som er mer tiltenkt menneskelig interaksjon. For eksempel for å vise en fremdriftsbar. Det er ikke umulig at måten den fungerer på kan endres i fremtiden. *Apt-get* på den andre siden er en gammel kommando som er forventet å være mer stabil i måten den fungerer på.

På linje 16 er det en jobb med et unikt navn, *test*. Linje 17 forteller hvilket stage denne jobben tilhører. Vi ser her at det går fint om en jobb og et stage har samme navn. Til slutt på linjene 18 – 21 er hovedscriptet. Det starter med en debug melding på linje 19. Linje 20 utfører syntaks-validering på alle puppetfiler som ligger i mappen som spesifiserer som parameter. Linje 21 utfører linting-sjekk på alle puppetfiler som ligger i mappen som ble spesifisert som siste parameter. Parameteret *–no-autoloader_layout-check* gjør at linting-sjekken utfører uten en spesiell test som sjekker om alle puppetfiler ligger i en korrekt mappestruktur. Å fjerne denne testen gjør det enklere å kjøre linting-sjekken, ettersom man slipper å lage en kompleks mappestruktur.

Resultatet av denne metoden var at:

1. Pipeline var suksessfull når puppetfilen var uten noen feil av typen error eller warning.
2. Pipeline feiler når puppetfilen hadde syntaks feil av typen error.
3. Pipeline feiler når puppetfilen hadde linting feil av typen error.
4. Pipeline var suksessfull når puppetfilen hadde linting feil av typen warning.

Basert på disse resultatene kan pipelinen kjøres med bash kommandoer kjørt direkte fra pipelinefilen. Pipelinen klarer å fange opp feilmeldinger korrekt.

6.3 CI/CD pipelinen

6.3.1 Statisk kodeanalyse

Første del av CI/CD pipelinen er å bruke verktøy for statisk kodeanalyse av puppet koden som blir comitted til repo-et. Formålet med dette er å øke kodekvaliteten og minimere sannsynligheten for at feil kommer videre til produksjonsmiljøet. Denne delen av pipelinen har to jobber som skal utføres, syntaks-validering og linting av endret kode.

Syntaks-validering sjekker alle puppetfiler for feil i puppetkodens syntaks med puppets valideringsprogram. Hvis ingen feil blir funnet startes neste jobb. Hvis feil blir funnet i syntaksen til puppet, vil pipelinen stoppe og feilene som ble oppdaget skrevet ut til pipeline-konsollet i GitLab.

Linting sjekker om puppetfilene følger puppets egen stilguide for hvordan koden skal se ut. Denne jobben kjører bare dersom syntaks-sjekkingen fullførte uten feil. Grunnen til dette er at linting-programmet ikke kjører korrekt hvis det syntaksfeil i koden. Hvis feil blir oppdaget av linteren, vil disse bli skrevet ut til pipeline-konsollet i GitLab.

Både syntaks-validering og linting kjører i containere bygget av samme Dockerimage. Det ble valgt å kjøre disse testene i containere siden de er avhengig av egne programmer som ikke allerede er installert på GitLab runner-en. Containeren bygges i pipelinen og de forskjellige jobbene kjøres basert på parametre sendt til containeren ved oppstart. Containeren bruker ett script som ”entrypoint”, her utføres enten syntaks-validering eller linting på filene basert på det parameteret som er sendt med.

Det er ønskelig at denne prosessen kjører raskt, da dette skal skje hver gang kode blir endret. Flere tilltak er gjort for å øke hastigheten på dette:

- Bygging av container.
  Containeren bygges i pipelinen, men bruker ”caching” mekanismen til Docker når en ny versjon skal bygges.

- Innhenting av kode til containeren.
  Siden programmene kjører i en container, trenger denne containeren tilgang til puppetfilene som skal sjekkes. I stedet for å kopiere inn disse filene til containeren blir ”Docker volume” brukt til å gjøre alle filene fra GitLab runner-en tilgjengelig fra containeren.

- Sjekk av endret kode.
  Bare endret kode blir sjekket, dette øker hastigheten på kjøretiden til programmene. Det blir brukt ”git diff” kommandoen til å finne filene som har endret seg, og det er bare disse filene som blir sjekket.

Se vedlegg D.2 for de komplette scriptene og filene som brukes i denne delen av pipelinen.

Kodeutdrag for GitLab pipeline seksjonen for statisk kodeanalyse:

```
...
# Validating and linting all puppet code that has changed since last commit
statictest:
    stage: test_changed_code
    before_script:
        - echo "Building validate and linting docker image"
        - docker build --cache-from checkcode:latest --tag checkcode:latest \
            dockeFileDirectory/validate_lint/.
        - if [[ $(git diff HEAD^ HEAD --name-only | grep '.pp') = "" ]]; \
            then echo "No files to check" && exit 0; fi
        - FILES=$(git diff HEAD^ HEAD --name-only | grep '.pp')

    script:
        - echo "Validating and linting puppet code"
        - docker run -v $(pwd):/root/puppet/ -t checkcode:latest parser "$FILES"
        - docker run -v $(pwd):/root/puppet/ -t checkcode:latest linter "$FILES"
...
```

Listing 1: Statisk kodeanalyse stage i pipelinen

Dersom det ikke er noen filer i ”FILES” arrayen som vist i kodesnutten ovenfor, vil pipelinen avslutte med ”exit 0” og pipelinen vil fortsette til neste steg. Pipelinen skal ikke fortsette dersom feil blir oppdaget av programmene.

Docker ENTRYPOINT script for containeren som utfører statisk kodeanalyse:

```
#!/bin/bash
cd /root/puppet/

TYPE=$1; shift
ARRAY=($@)

# The command to run is chosen from the first parameter.
if [ "$TYPE" == "parser" ]; then
  echo "Running puppet parser validate"
  COMMAND="/opt/puppetlabs/bin/puppet parser validate"

elif [ "$TYPE" == "linter"  ]; then
  echo "Running puppet lint"
  COMMAND="/usr/bin/puppet-lint --no-autoloader_layout-check \
    --fail-on-warnings --no-80chars-check --no-140chars-check"

else
  echo "Error: No parameter given in second position, must provide parser og linter"
  exit 1
fi

# Loops through all .pp files that have changed since last commit
#  and runs the specified command on them.
for file in "${ARRAY[@]}"
do
  echo "$file"
  $COMMAND "$file"
done
```

Listing 2: Entrypoint script for statisk kodeanalyse

Entrypoint scriptet kjører forskjellige programmer basert på parameter sent til containeren. Hvis feil oppdages i noen av programmene, vil jobben avsluttes med en ”exit kode”, slik at pipelinen ikke fortsetter.

Tester før pipelinen

Å vente på at en pipeline skal fullføre kan ta lang tid eller være ressurskrevende. Det kan være ønskelig å kjøre enkle tester som syntaks-validering og linting lokalt før man starter en pipeline. Dette vill forhindre unødvendig oppstart av pipelinen. For å utføre dette ble et feature i git kalt ”git hooks” benyttet. Git hooks gjør at man kan kjøre script når visse hendelser skjer, som ved commit eller push.

Prosjektgruppen modifiserte et script fra internett for å utføre dette [46]. Det opprinnelige scriptet kjørte syntaks validering av PHP kode. Dette ble endret til syntaks-validering av puppetkode. Det ble også lagt til puppet lintingsjekk. Kodestilen ble endret til å være i samsvar med andre script i dette prosjektet. Til slutt ble koden i scriptet kvalitetsikret i henhold til prosjektgruppens standarder.

Koden under er prosjektgruppens versjon av dette scriptet.

```
1  #!/bin/bash
2
3  ###   Functions  ###
4
5  # prints red text
6  function errorPrint() {
7      echo -e "\e[1;31m\t $1 \e[0m" >&2
8  }
9
10 ### Script Start ###
11
12 # for all files changed since last commit
13 git diff --cached --name-only | while read FILE
14 do
15   # find all files ending in .pp
16   if [[ "$FILE" =~ ^.+(pp)$ ]]
17   then
18     # if the file still exits
19     if [[ -f $FILE ]]
20     then
21       # run syntax validarion on the file and check errors
22       if ! puppet parser validate "$FILE"
23       then
24         # if there are syntax errors, print error message and exit with errorconde 1
25         errorPrint "Aborting commit due to files with syntax errors."
26         exit 1
27       fi
28       # run linting check on file and check for errors
29       if ! errorText=$(puppet-lint "$FILE")
30       then
31         # if there are lining errors, print error message and exit with errorconde 1
32         errorPrint "$errorText"
33         errorPrint "Aborting commit due to files with linting errors."
34         exit 1
35       fi
36     fi
37   fi
38 # needed in some type of shells
39 done || exit $?
```

Listing 3: Script for validering og linting av kode lokalt

På linje 6-8 er en funksjon som printer rød tekst med 1 TAB innrykk til konsollet. Dette er gjort for å gjøre koden senere mer leselig. Samt at det gjør det enklere å forbedre og gjøre endringer på koden hvis dette skulle være ønskelig i fremtiden. På linje 13 finner man alle filer som har blitt endret og lagt till via *git add*. Disse filenes relative path pipes så til en while loop. På linje 19 blir det sjekket om filen er av riktig type. Det vil si at den ender med *.pp*. Hvis ikke, så går loopen videre uten å gjøre noe med filen. På linje 19 blir det sjekket at filen faktisk eksisterer. Dette er nødvendig fordi det er mulig filen har blitt endret selv etter siste gang den ble lagt til via *git add* [46].

På linje 22 blir det kjørt syntaks-validering av filen. Hvis scriptet oppdager feil så vil

scriptet gå til linje 25 og 26 som printer en feilmelding og avslutter scriptet med feilkode 1. Det samme skjer på linje 29-35 for sjekking med linting. På linje 29 lages utskriften til liningsjekken i en variabel. Dette gjøres for at scriptet på linje 32 skal skrive ut til konsollet på samme format som det gjøres i syntaks-valideringen ovenfor. På linje 39 er det en || *exit 1* denne trengs i visse spesielle situasjoner [46].

For å bruke dette scriptet må man stå i en mappe som inneholder en *.git* mappe og man må kopiere scriptet inn i riktig mappe ved bruk av denne kommandoen:

```
cp pre-commit .git/hooks/pre-commit
```

Kodekvalitet

Scriptet er kvalitetssikret ved bruk av https://www.shellcheck.net/, et verktøy for statisk kodeanalyse av shell scripts. Shellsheck klagde på måten det opprinnelige scriptet utførte sjekk av returverdier etter en kommando. Den foreslo at sjekk av typen *if [$? -ne 0]* blir byttet til *if ! kommando* dette ble gjort på linje 22 og 29. Videre mente shellsheck at på linje 13 så burde *while read FILE* endres til *while read -r FILE*. Men det står på forklaringssiden [47] til denne feilmeldingen, at *read -r* brukes for at backslashes assosiert med line feeds ikke skal tolkes. Det er i dette tilfellet nødvendig å gjøre dette, fordi den foregående *git diff –cached –name-only*. Returnerer relative fil paths avskilt av line feeds. Denne linjen blir derfor ikke endret.

6.3.2 Testing av keystone

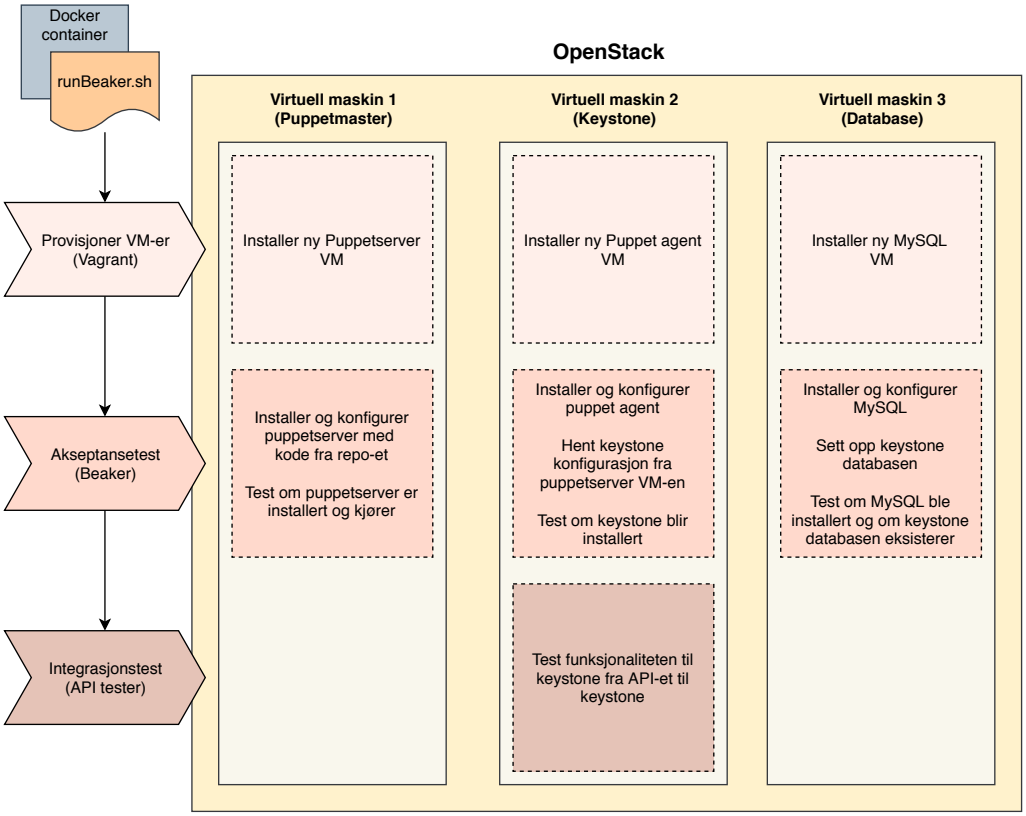
Andre del av CI/CD pipelinen er å teste keystone modulen ved bruk av akseptanse-, og integrasjonstester. Formålet med testing er å sjekke om keystone modulen kan installeres og at den fungerer med den koden som nå ligger i repo-et. Testingen skjer uansett hva slags type kode som er endret, om det er puppetkode, Hiera filer, nye Dockerfiler eller testene selv.

For å teste hele keystone ”stacken” må nye VM-er settes opp og konfigureres. Puppet programvaren må installeres og keystone installeres på en av maskinene med kommandoen ”puppet agent -t”. Keystone er også avhengig av en MySQL server for lagring. Keystone stacken vi bruker består da av tre VM-er som alle kjører i openstack.

For å provisjonere nye VM-er brukes Vagrant med en openstack plugin for å kunne starte serverne i SkyHiGh. Beaker brukes til å installere de nødvendige programmene på serverene og for å sjekke at programmene har blitt installert og konfigurert korrekt. Et python script ble utviklet for å kjøre tester mot keystone API-et. Alt dette blir kjørt sekvensielt i et script som kjører i en Docker container.

Se vedlegg D.3 for de komplette scriptene og filene som brukes i denne delen av pipelinen.

Figuren under illustrerer hvordan scriptet *runBeaker.sh* sekvensielt kjører og utfører de forskjellige oppgavene.



Figur 13: CI/CD testing stage illustrert

Provisjoner VM-er

Scriptet *runBeaker.sh* kjører først ”Vagrant up” på en Vagrantfile som definerer hvilke maskiner som skal settes opp i openstack miljøet. Vagrant setter opp alle maskinene i openstack og kjører et enkelt bootscript for å gjøre maskinene klare for å bli koblet til fra Beaker.

Akseptansetesting

Hvis vagrant klarte å starte maskinen uten feil, starter akseptansetestingen av keystone modulen. Her blir Beaker kjørt med 2 ruby scripts mot serverene.

Det første scriptet installerer og konfigurer Puppet og MySQL på maskinene. Scriptet kopierer over puppetkoden (*roles, profiles* og *ntnuopenstack*), Hiera filene og Puppetfilen fra repo-et til den nye Puppetmaster serveren. Programmet r10k blir kjørt på Puppetfilen etter at den har blit kopiert over for å installere alle puppet moduler som keystone er avhengig av.

Det andre scriptet sjekker om puppet programvaren ble installert og om MySQL har

fått opprettet ”keystone” databasen. Keystone modulen blir hentet og installert fra Puppetmasteren til keystone serveren ved bruk av kommandoen ”puppet agent -t”. Scriptet sjekker om denne kommandoen klarer å installere keystone korrekt uten feilmeldinger.

Integrasjonstesting

Hvis Beaker testene fullførte uten feil startes integrasjonstestingen av den installerte keystone modulen. Til dette blir det brukt et python script som tester funksjonaliteten til keystone. Python scriptet ble utviklert av prosjektgruppen siden vi ikke fant noen gode verktøy eller rammeverk som testet keystone for oss. Scriptet tester store deler av keystones funksjonalitet.

Første del av scriptet forsøker å lage keystone ressursene *token, domene, gruppe, prosjekt, region, rolle* og *bruker*. Ressursene blir sjekket om de inneholder de korrekte verdiene før de blir slettet. Scriptet sjekker om de forventede tjenestene er listet opp i keystone sin ”service catalog”. Alle testene kjører mot keystone sin API, så det blir også sjekket om keystone gir de korrekte HTTP status kodene når metodene *”GET”, ”POST”, ”PUT”, ”PATCH”, ”DELETE”* kjøres.

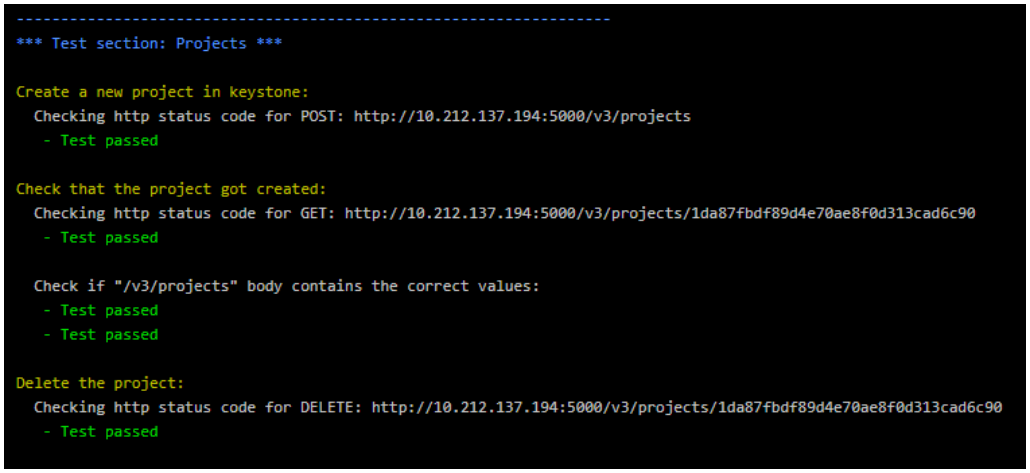
Hvis disse testene passerer kjøres en større test mot keystone. Her blir flere ressurser opprettet og satt sammen. Denne testen gjør følgende:

- Oppretter et nytt domene.
- Oppretter et nytt prosjekt.
- Oppretter en ny bruker.
- Oppretter en ny gruppe og legger til brukeren i gruppen.
- Oppretter en ny rolle, legger til gruppen i rollen og legger til gruppen i prosjektet.

Når disse ressursene er opprettet, sjekkes det at de ble opprettet og om de inneholder de korrekte verdiene. Disse sjekkene består av følgende:

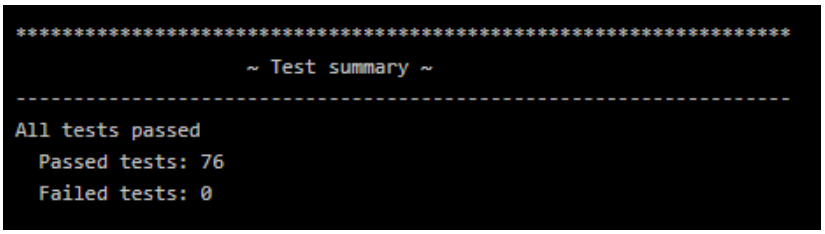
- Sjekk om prosjektet ble opprettet og er en del av domenet.
- Sjekk om brukeren ble opprettet og om brukeren er en del av prosjektet og domenet.
- Sjekk om brukeren er en del av gruppen og om gruppen er en del av domenet.

Til slutt blir alle opprettede ressurser slettet. En utskrift av hvilke tester som kjører og om det feiler eller passerer blir vist i pipeline konsollet. Under ligger et eksempel på utskriften.



Figur 14: Integrasjonstest pipeline utskrift

Hvis en test feiler blir utskriften endret til *"- Test failed"* og fargen endret til rødt. Når alle testene er ferdig med å kjøre blir en oppsummering skrevet ut til konsollet som viser hvor mange tester som ble kjørt og hvor mange som feilet eller passerte. Et eksempel på dette er vist i figuren under.



Figur 15: Integrasjonstest pipeline utskrift oppsummering

Hvis en eller flere tester feiler avsluttes pipelinen med feil.

Docker container

Scriptet *runBeaker.sh* kjører som et entrypoint script i en Docker container. Vi ønsket å minimere kompleksiteten på GitLab runner-en og valgte derfor å lage et Docker image til å ha den nødvendige programvaren som Vagrant, Beaker og python scriptet er avhengig av installert. Denne containeren blir bygget i pipelinen på samme måte som containeren for statisk kodeanalyse som vist i kodeutdraget nedenfor.

```
...
# Acceptance and integration testing with Beaker and API tests
acceptance_and_integration:
    stage: testing
    before_script:
        - echo "Building acceptance and integration docker image"
        - docker build --cache-from beaker:latest \
          --tag beaker:latest dockeFileDirectory/beaker/.

    script:
        - echo "Running acceptance and integration tests on keystone"
        - docker run -v $(pwd):/root/puppet/ -t beaker:latest
...
```

Listing 4: Testing stage i pipelinen

Hvis Vagrant, Beaker eller Python scriptet feiler vil pipelinen ikke gå videre til neste stage.

Hvordan keystone får sin konfigurasjon

Keystone blir installert med konfigurasjonen som er definert i Hiera filene som ble kopiert over til Puppetmasteren. Keystone sin autentiserings URL, endepunkt URL-er og MySQL serverens IP står beskrevet i Hiera filene. Siden IP-adresser endrer seg hver gang nye servere blir startet i openstack, må vi fylle inn disse dataene før de blir kopiert over til Puppetmasteren.

Vi har behov for å finne floating og lokale IP-adresser som så må settes inn i Hiera filene. Et eksempel på hvordan vi henter ut disse IP-adressene:

```
...
# Get IP adresses from the vagrant machines.
IPS=$(vagrant ssh-config | grep HostName)
VAGRANT_IP_MASTER=$(echo $IPS | awk '{ print $2 }')
VAGRANT_IP_AGENT=$(echo $IPS | awk '{ print $4 }')
VAGRANT_IP_MYSQL=$(echo $IPS | awk '{ print $6 }')
MYSQL_LOCAL_IP=$(vagrant ssh mysql \
    -c "ip address show ens3 | grep -w 'inet' | sed -e 's/^.*inet //' -e 's/\/.*$//'")
KEYSTONE_LOCAL_IP=$(vagrant ssh agent \
    -c "ip address show ens3 | grep -w 'inet' | sed -e 's/^.*inet //' -e 's/\/.*$//'")
export VAGRANT_IP_AGENT=$VAGRANT_IP_AGENT
...
```

Listing 5: Uthenting av IP-adresser fra Vagrant i runBeaker.sh

Disse blir senere i scriptet lagt inn i Hiera filene ved bruk av regex kommandoen *"sed"*. Ved å sette IP og URL-er på denne måten trenger vi ikke å bruke statiske IP-adresser på testing serverne vi setter opp. Det kan også brukes samme Hiera data på test serverne som i produksjon, siden dette scriptet tar seg av å sette de korrekte verdiene selv. Et eksempel på en slik kommando er vist nedenfor.

```
...
sed -i "s/^.*ntnuopenstack::keystone::mysql::ip:.*$/ntnuopenstack::keystone::mysql::ip:
        '$MYSQL_LOCAL_IP'/" $CURR_PATH/data/common.yaml
...
```

Listing 6: Setting av MySQL IP i Hiera-data i runBeaker.sh

Hvordan serverne har blitt sikret

Med vår implementasjon av testing delen av pipelinen så fikk vi noen utfordringer med å sikre løsningen. Vi kunne ikke bruke Beakers openstack-, eller Vagrant hypervisor plugin til å starte og konfigurere maskiner direkte på grunn av problemer som oppsto med disse plugins-ene. Våres løsning bruker Vagrant og Beaker som egne programmer, som holdes sammen av scriptet *runBeaker.sh*. Problemet som oppsto med denne løsningen er at Beaker trenger full root tilgang på Vagrant serverne for å kunne installere programvare. Det hjelper ikke å kjøre kommandoer som ”sudo” i Beaker scriptene siden Beakers DSL kommandoer ikke støtter dette. For å få brukt Beaker måtte vi tillate SSH innlogging som root med passord til våres Vagrant maskiner som kjører i openstack.

Vi trengte da en måte å lage passord som kunne brukes av Beaker til å logge inn på maskinene som startes av Vagrant. Til dette bruker vi programmet *'pwgen'* og eksporterer passordet før Vagrant blir kjørt. Et eksempel på hvordan dette gjøres er vist nedenfor.

```
...
# Generates random passwords between PASS_LENGTH_MIN - PASS_LENGTH_MAX.
PASS_LENGTH_MIN=20
PASS_LENGTH_MAX=30

# Password to log into the vagrant machines over SSH as root.
VAGRANT_PASS=$(pwgen -c -n -N 1 $(shuf -i $PASS_LENGTH_MIN-$PASS_LENGTH_MAX -n 1))
export VAGRANT_PASS=$VAGRANT_PASS
...
```

Listing 7: Generering av passord i runBeaker.sh

Dette passordet blir hentet inn i Vagrantfilen som en ”environment variabel”. Alle maskinene kjører et ”provision” script for å tillate SSH og setter passord på root brukeren, koden under viser hvordan dette gjøres i Vagrantfilen.

```
...
# Script to run on the vagrant machine.
$script = <<-SCRIPT

# Change ssh config to allow root ssh access with password.
sudo sed -i -e 's/PermitRootLogin prohibit-password/PermitRootLogin yes/g' \
    /etc/ssh/sshd_config
sudo sed -i -e 's/PasswordAuthentication no/PasswordAuthentication yes/g' \
    /etc/ssh/sshd_config
sudo sed -i -e 's/StrictModes yes/StrictModes no/g' /etc/ssh/sshd_config

# Add password to root user.
sudo usermod --password $(echo $password | openssl passwd -1 -stdin) root

# Restart ssh service.
sudo service ssh restart
SCRIPT


# Vagrant configuration.
Vagrant.configure('2') do |config|
  config.vm.synced_folder ".", "/vagrant", disabled: true
  config.ssh.username                    = 'ubuntu'
  config.ssh.forward_agent               = true

  # Run the script on the provisioned vagrant machines.
  # Uses environmental variable for password.
  config.vm.provision 'shell', inline: $script, env: {"password" => ENV['VAGRANT_PASS']}
...
```

Listing 8: Konfigurering av SSH og passord i Vagrantfile

Det samme passordet må settes inn i Beaker HOSTS filen, slik at Beaker vet hvilke passord den skal bruke i SSH tillkoblingen. Denne HOSTS filen blir generert av *runBeaker.sh* scriptet og lagret i Docker containeren.

```
...
# Creates the beaker hosts file.
cat <<EOF > $BEAKER_HOSTFILE
---
HOSTS:
  Master:
    platform: ubuntu-16.04-amd64
    hypervisor: none
    ip: $VAGRANT_IP_MASTER
    roles:
      - master
    ssh:
      user: 'root'
      password: $VAGRANT_PASS
      auth_methods:
        - password
...
```

Listing 9: Generering av Beaker HOSTS fil i runBeaker.sh

MySQL root passordet blir også generert med *"sed"* kommandoen på samme måte som "VAGRANT_PASS" og satt inn i Beaker scriptene.

Data fra Hiera

Keystones admin passord, token og MySQL passord for keystone databasen befinner seg i Hiera filene. For å forenkle bruken av testing scriptene valgte vi å hente ut disse dataene fra Hiera filene som ligger i repo-et, i stedet for å hardkode dem inn i scriptene. All variabel data som brukes til å konfigurere keystone i produksjonsmiljøet vil da også brukes i testingsmiljøet. Disse verdiene blir hentet ut i *runBeaker.sh* scriptet, eksportert som environment variabler og brukt i Beaker-, og Python scriptene. Eksempelet under viser hvordan Python integrasjonstest scriptet bruker disse dataene.

```
...
# Set local variables from ENV variables if the exist.
if os.environ.get("VAGRANT_IP_AGENT") is None:
    print_color(TermColor.RED, "ERROR: Environmental variable
            'VAGRANT_IP_AGENT' is not set.")
    exit(1)
KEYSTONE_URL = "http://" + os.environ["VAGRANT_IP_AGENT"] + ":5000"

if os.environ.get("KEYSTONE_PASSWORD") is None:
    print_color(TermColor.RED, "ERROR: Environmental variable
            'KEYSTONE_PASSWORD' is not set.")
    exit(1)
KEYSTONE_PASSWORD = os.environ["KEYSTONE_PASSWORD"]

if os.environ.get("KEYSTONE_TOKEN") is None:
    print_color(TermColor.RED, "ERROR: Environmental variable
            'KEYSTONE_TOKEN' is not set.")
    exit(1)
KEYSTONE_TOKEN = os.environ["KEYSTONE_TOKEN"]
...
```

Listing 10: Bruk av Hiera verdier i test_keystone.py

Hiera data blir også brukt i Beaker scriptene som vist under.

```
...
  mysqlRootPass = ENV['MYSQL_ROOT_PASS']
  mysqlKeystonePass = ENV['MYSQL_KEYSTONE_PASS']
...
```

Listing 11: Bruk av Hiera verdier i pre_suite_keystone.rb

Ved å bruke data fra Hiera, og å lage egne passord i kjøretiden for scriptet kan data fra Hiera endres uten at scriptene selv trenger å endres. Dette vil forenkle vedlikehold av testene som kjører.

6.3.3 Avslutt pipeline

Automatisk merge request

Etter en pipeline har fullført suksessfullt, kan det være ønskelig å gjennomføre et automatisk merge request til en annen branch. Prosjektgruppen fant et script på en av GitLab sine nettsider som kunne utføre dette [48].

Men etter utprøving viste det seg at scriptet ikke hadde tilstrekkelige feilhåndteringer.

For eksempel hvis et API kall utført via *curl* returnerte feil statuskode etter at det var prøvd å lage et merge request. Så fortsatte scriptet bare som om alt var OK. Pipelinen rapporterte ikke at det automatiske merge requestet hadde feilet. Scriptet er avhengig av at flere miljøvariabler er tilgjengelige. Hvis disse ikke var satt, så fikk man ikke noen fornuftige feilmeldinger. En slik sjekk og feilmeldingen ble lagt til. Det var også kun mulig å merge fra den branchen man var i, til default branch. Dette var lite fleksibel. Så prosjektgruppen la til muligheten til å merge til hvilken som helst branch. Det et var heller ikke gjennomført statisk analyse av kodekvalitet i henhold til prosjektgruppens standarder. Scriptet ble derfor analysert ved bruk av *ShellCheck* [49]. Modifikasjoner ble så uført basert på anbefalingene analyseverktøyet gjorde. Til slutt ble det testet for å se at det fungerte.

Prosjektgruppens versjon av dette scriptet kan ses under.

```
#!/usr/bin/env bash

# Base script source(1.4.19):
#   https://about.gitlab.com/2017/09/05/how-to-automatically-create-a-new-mr-on-gitlab-with-gitlab-ci/

# For ease of debugging,
# uncomment the line below(!!! passwords/sectet tokens may be printed to console !!!)
# set -x

#########################################################
########               Functions               ########
#########################################################

# prints red text
function errorPrint() {
    echo -e "\e[1;31m\t $1 \e[0m" >&2
}

# prints green text
function succsessfulPrint() {
    echo -e "\e[32m\t $1 \e[0m" >&2
}

# Checks if a variable is set
function isSet () {
    if [ -z "${1}" ]
    then
        # Print error message and exit with error code
        errorPrint  "$2 not set"
        exit 1
    fi
}

#########################################################
########             Script Start              ########
#########################################################

```

```
38  # Check if all required variables are set
39  isSet "${CI_PROJECT_URL}" CI_PROJECT_URL
40  isSet "${CI_PROJECT_ID}" CI_PROJECT_ID
41  isSet "${CI_COMMIT_REF_NAME}" CI_COMMIT_REF_NAME
42  isSet "${GITLAB_USER_ID}" GITLAB_USER_ID
43  isSet "${PRIVATE_TOKEN}" PRIVATE_TOKEN
44
45  # Extract the host where the server is running, and add the URL to the APIs
46  [[ $HOST =~ ^https?://[^/]+ ]] && HOST="${BASH_REMATCH[0]}/api/v4/projects/"
47
48  # If no target_branch is provided by paramater, use default_branch
49  if [ $# -eq 0 ]
50  then
51      # Look which is the default branch
52      TARGET_BRANCH=$(curl --silent "${HOST}${CI_PROJECT_ID}" --header
        ↪  "PRIVATE-TOKEN:${PRIVATE_TOKEN}" | python3 -c "import sys, json;
        ↪  print(json.load(sys.stdin)['default_branch'])");
53  else
54      TARGET_BRANCH="$1"
55  fi
56
57  # Check if required variable is set
58  isSet "${TARGET_BRANCH}" TARGET_BRANCH
59
60  # The description of our new merge requests, we do not want to remove the branch after the
     ↪  MR has
61  # been closed
62  BODY="{
63      \"id\": ${CI_PROJECT_ID},
64      \"source_branch\": \"${CI_COMMIT_REF_NAME}\",
65      \"target_branch\": \"${TARGET_BRANCH}\",
66      \"remove_source_branch\": false,
67      \"title\": \"WIP: ${CI_COMMIT_REF_NAME}\",
68      \"assignee_id\":\"${GITLAB_USER_ID}\"
69  }";
70
71  # Require a list of all the merge request and take a look if there is already
72  # one with the same source branch
73  LISTMR=$(curl --silent "${HOST}/${CI_PROJECT_ID}/merge_requests?state=opened" --header
     ↪  "PRIVATE-TOKEN:${PRIVATE_TOKEN}");
74  # Check if last last call was succsessful
75  ifNotSuccessful "${?}" "bad responce form gitlab server"
76  # Count number of existing merge requests
77  COUNTBRANCHES=$(echo "${LISTMR}" | grep -o "\"source_branch\":\"${CI_COMMIT_REF_NAME}\"" |
     ↪  wc -l);
78
79  # If no merge requests found, let's create a new one
80  if [ "${COUNTBRANCHES}" -eq "0" ]
81  then
82      status=$(curl -s -o /dev/null -w '%{http_code}' -X POST
        ↪  "${HOST}${CI_PROJECT_ID}/merge_requests" \
83          --header "PRIVATE-TOKEN:${PRIVATE_TOKEN}" \
84          --header "Content-Type: application/json" \
85          --data "${BODY}")
86
87      # Check if merge requests call was succsessful
```

```
88      if [ "${status}" -ne 201  ]
89      then
90          errorPrint  "unable to create new merge_requests. curl returnd starus: ${status}"
91          exit 1
92      fi
93      # Everything OK, exit with no error code
94      succsessfulPrint "Opened a new merge request: WIP: ${CI_COMMIT_REF_NAME} and assigned
          ↪  to you"
95      exit 0
96  fi
97
98  # Everything OK, exit with no error code
99  succsessfulPrint "No new merge request opened, because one already exits"
100 exit 0
```

Listing 12: Automatisk merge request script

Det opprinnelige scriptet

På linje 46 lagets en variabel for å lagre gitlabserverens URL. På linje 52 finner man repoets default branch ved å sende et HTTP GET request til gitlabserveren. Svaret man får tilbake pipes så til et inline python script som parser json og henter ut ønsket string, altså *default_branch*. På 62-69 lages det en json string som senere vil bli brukt i et HTTP POST request til gitlabserverens API. Linjene 73 henter ut alle eksisterende merge request ved å sende et HTTP GET request til gitlabserverens API. Linje 77 teller antall merge request for denne branchen som pipelinen kjører i nå.

På linje 80 sjekkes det at det ikke er noen eksisterende merge request. Og hvis det ikke er noen, så lages et HTTP POST request på linjene 82-85.

Nye deler av scriptet

På linje 15-17 defineres en hjelpefunksjon *errorPrint* denne får et argument *$1* som er en forklarende feilmelding(string). Denne feilmeldingen blir på linje 16 formatert i rød tekst og så skrevet til pipelines konsoll output log. Tekstfargen blir så resatt tilbake til default. Linje 20-22 gjør det samme med grønn tekst. Disse formateringene gjøres for at det skal bli lettere å finne meldinger i pipelines konsoll output log.

Fra linje 25-31 er en funksjon *isSet* som brukes til å sjekke om en påkrevet variabel, gitt ved parameter *$1*, er initialer. Hvis den ikke er det så printes en forklarende feilmelding, gitt ved parameter *$2*, til pipelines konsoll output log. Scriptet kaller så *exit 1* for å fortelle pipelien at den skal feile. Prosjektgruppen fant ut at hvis de glemte å initialisere noen miljøvariabler, så ville det opprinnelige scriptet bare kjører videre og erklære at alt var OK. Selv om et merge request ikke var laget. Ved å alltid sjekke at visse variablene er initialisert, så gjør det det veldig enkelt å debugge og å finne ut hvorfor en merge request ikke har blitt laget ettersom pipelien feiler med en fornuftig feilmelding.

På linjene 39-43 og 58 sjekkes alle variablene som scriptet bruker.

- CI_PROJECT_URL er URL-en til den aktuelle GitLab serveren. Som hoster prosjektet.
- CI_PROJECT_ID er den unke ID-en til prosjektet.
- CI_COMMIT_REF_NAME er git branchen som pipeline kjører I akkurat nå.
- GITLAB_USER_ID er bruker ID-en til den GitLab brukeren som startet pipelinen denne gangen.
- PRIVATE_TOKEN er autorisasjons tokenet som kreves for å gjøre et merge request via GitLab API-et.
- TARGET_BRANCH er branchen som det skal merges til.

På linje 49-55 settes branchen som det skal merges til (*TARGET_BRANCH*). Linje 49 sjekker om det er sendt med noen argumenter til scriptet. Hvis det er det, så settes *TARGET_BRANCH* til denne, linje 54, hvis ikke så blir repoets default branch, ofte kalt 'master', satt til *TARGET_BRANCH*. I linje 52 er det mulighet til å endre branchen som det skal merges til, lagt til av prosjektgruppen fordi det ved eksperimentering med brancher kan være ønskelig å kunne utføre en merge request til annet enn default branch.

Mens på linjene 88-92 sjekkes det om foregående *curl* kommando, som sendte en *HTTP POST request* til gitlabserverens API lykkes. Hvis den har lykkes så skal et merge request ha blitt laget. For å muliggjøre denne sjekken ble det på linje 82 gjort en endring i *curl* kommandoen slik at HTTP responskoden ble hentet ut og lagret i variabelen *status*.

På linjene 111 og 117 blir grønn tekst skrevet til pipelines konsoll output log. Dette gjøres fordi det øker lesbarheten til loggen. Basert på at andre ting som har lykkes i pipelien får en grønn melding, så forventer en bruker at også seksjonen av pippelien returnerer en grønn tekst hvis den lykkes. Eller en rød tekst hvis den feiler.

Kodekvalitet

Shellcheck ble brukt for å sikre god kodekvalitet [49]. Dette verktøyet bemerket at notasjonen *'kommando'* bør byttes til *$(kommando)*. Dette fordi den mente at den foregående notasjonen var en Legacy feature. Slike endringer ble gjort på linje 52, 73 og 77.

Videre sa den at det på linje 77 og 80 bør variabler som *$COUNTBRANCHES* settes inn i dobbel quotes(") for å forhindre globbing og ord splitting. Altså bør det stå "*$COUNTBRANCHES*". Dette ble gjort. Scriptet passerer nå shellsheck uten noen klager.

Alternativer og forbedringer

Slik som scriptet er nå vill pipelinen feile hvis et merge request ikke kan gjennomføres, gitt at det ikke eksisterer et allerede. Hvis man skulle være enda strengere så burde kanskje pipelinjen feile også hvis man prøver å gjøre et merge request når det allerede eksisterer et. For å gjøre den kan linje 99 byttes fra successfulPrint til errorPrint og linje 100 kan byttes til å kalle exit med feilkode 1 istedenfor 0.

Den store faren med dette scriptet er at for å kunne gjøre et automatisk merge re-

quest via gitlabserverens API. Så må man benytte et autorisasjonstoken [48] med scope nivå *API*. Dette gir lese og skrive tilgang til alt i GitLab prosjektet. Hvis dette token-et skulle komme på avveie så er det en risiko for at uønskede aktører vil kunne angripe gitlabserverens repo. Det ville blant annet være mulig å slette alt i repoet, eller endre gruppemedlemmers tilgangsnivå.

Automatisk merge

Etter en pipeline har fullført suksessfullt, kan det være ønskelig å gjennomføre et automatisk merge fra en branch til en annen branch. Prosjektgruppen fant et script på stackoverflow som ble brukt utgangspunkt for dette arbeidet [50].

Det opprinnelige scriptet var laget for å kjøre med en GitLab runner i Docker executor modus. Det var ikke utført kvalitetssikring av koden i henhold til dette prosjektets standarder. Det var hardkodet slik at det kun kunne merge til master branchen. På grunn av konfigurasjonsproblemer på SHH tjenesten til den serveren som NTNU brukte til å hoste gitlabinstallasjon, så fungerte ikke dette scriptet i sin daværende form. Det var derfor nødvendig å gjøre mange endringer på scriptet.

Prosjektgruppens versjon av scriptet som automatisk merger to brancher kan ses under.

```
#!/bin/bash

########         Prescript check         ########

# If no branch is provided as first argument, then give error message and exit with
 ↪  errorcode 1
if [ "$#" -ne 1 ]
 then
 # Print red text
   echo -e "\e[1;31m\t missing required parameter: to_branch \e[0m" >&2
   exit 1                                  # exit script
 fi

########             Variables            ########

# Git server + repo address
gitlab_repo="git@gitlab.stud.idi.ntnu.no:alexajak/automatiset_openstack_testing.git"
# GitLab server
gitlab_server="gitlab.stud.idi.ntnu.no"
# Name of repository and name for directory
repo_name="automatiset_openstack_testing"
# Name of the branch to be megre to
to_branch=$1
# The branch that the pipeline was started form
from_branch="$CI_BUILD_REF"

########               Functions              ########

# prints green text
function print() {
```

```
30      # Format text
31      greenText="\e[32m"
32      resetText="\e[0m"
33      echo -e "$greenText \t $1 $resetText" >&2
34  }
35
36  ########             Script Start             ########
37
38  # Clone git repo
39  print "clone git repo"
40  chmod 600 sshkey
41  sudo rm -rf $repo_name
42  sudo ssh-keyscan "$gitlab_server" >> /home/gitlab-runner/.ssh/known_hosts
43  ssh-agent bash -c "ssh-add sshkey && git clone $gitlab_repo"
44  cd automatiset_openstack_testing || exit 1
45
46  # Configure git
47  print "Configureing git"
48  git config --global user.email "$GITLAB_USER_EMAIL"
49  git config --global user.name "$GITLAB_USER_ID"
50  git remote set-url origin "$gitlab_repo"
51
52  # Merge to branch and push to origin
53  print "merge to branch $to_branch"
54  git checkout "$to_branch"
55  git reset --hard origin/"$to_branch"
56  git merge "$from_branch"
57  ssh-agent bash -c "cd .. && ssh-add sshkey && cd $repo_name && git push origin $to_branch"
```

Listing 13: Automatisk merge script

På linje 6 sjekkes det om en det er sendt med et argument til scriptet. Hvis det ikke er det så printes en feilmelding og scriptet avbryter med feilkode 1. Funksjonen print på linjene 29-34 får en tekst som argument *$1*. Denne teksten formateres så til en rød/grønn tekst og printes så i pipelinens konsoll output log. Dette gjøres for å gjøre loggen mer lesbar.

Linjene 16-24 setter noen variabler som enten skal brukes flere ganger. Eller som det ikke er ønskelig er hardkodet inni kommandoer. Linjene 39, 47 og 53 er debugg meldinger som som brukes til å gjøre det lettere å lese pipeline loggen. På linje 40 settes en privat SSH nøkkel til å ha riktig rettighetsnivå for denne brukeren. Det er viktig å sikre dette ettersom en nøkkel med feil rettighetsnivå vil bli avvist av subsekvente SSH kommandoer.

På linje 42 legges domenet til GitLab-serveren inn i SSH-filen *known_hosts*. Hensikten med dette er å sikre at scriptet kan kjøre automatisk. Første gang man kontakter en ny host via SSH, må en bruker manuelt godkjenne hosten via kommandolinje input. Linje 43 bruker relevant SSH nøkkel til å klone ned det samme repo-et som scriptet ligger i. Det lages her en kopi av repo-et inni en kopi av repo-et. Dette er nødvendig fordi den første kopien som ble hentet ned når pipelinen startet kan ha blitt utsatt for endringer før dette scriptet kjører. Slike endringer på filstrukturen kan føre til at et forsøk på merge blir avist av git. En ny kopi av repo-et sikrer at man jobber på en ren

versjon. Før man kloner ned et repo er det viktig at det ikke en en mappe med samme navn som repo-et allerede der men kloner ned. Dette er spesielt aktuelt hvis et tidligere run av pipelinen feilet. I så fall kan det være igjen filer og mapper fra forrige gang. På linje 41 sikres dette ved å prøve å rekursivt slette en mappe med samme navnet som repo-et. Hvis det ikke er en slik mappe der fra før så fortsetter bare scriptet videre. På linje 44 flytter man seg inn i det nylig klonede repo mappen eller scriptet avsluttes hvis det ikke lykkes. På linjene 48-50 konfigureres med et brukernavn, epost og et repo origin. Dette er nødvendig for å kunne utføre siste del av scriptet.

På linje 22 hentes det fra argumentet *$1*, ut navnet på branchen det skal merges til. Dette gjøres for å gjøre koden lenger ned i scriptet mer leselig. På linje 54 flytter man seg til den branchen det skal merges til. På linje 55 sikres det at hvis det har skjedd noen endringer på denne branchen så blir disse resatt til en tidligere tilstand. På linje 56 utføres en lokal merge og på linje 57 pushes disse endringene opp til origin repo-et.

Kodekvalitet

Scriptet er kvalitetssikret ved bruk av *Shellcheck* [49]. Den ga en melding om at linje 42 bør endres. Den gir følgende feilmelding:
"*- SC2024: sudo doesn't affect redirects. Use .. | sudo tee -a file*"
Men det står også på forklaringssiden til denne feilmeldingen at det er et unntak for det: "*If you want to run a command as root but redirect as the normal user, you can ignore this message.*"[51]. Det er dette som er hensikten her så denne linjen blir ikke endret.

Alternativer og forbedringer

Slik som scriptet er nå så er det avhengig av at det er en SSH nøkkel med navnet *sshkey* i rotmappen til git repo-et. Det er ikke trygt å lagre slike ting i et git repo. Det blir gjort her for å forenkle utviklingsprosessen. En mer ordentlig løsning ville være å ha SSH nøkkelen lagret på GitLab runneren, ikke i repo-et.

Linje 43 og 57 bruker noen one-line SSH kommandoer som kan være vanskelige å lese. Dette var nødvendig på grunn av konfigurasjonsproblemer i SSH tjenesten på GitLab-serveren. Hadde ikke disse problemene vært tilstede er det kanskje mulig å bryte opp kommandoene over flere linjer. Konfigurasjonsproblemer på SSH tjenesten ble etterhvert løst av NTNUs driftspersonell. Men prosjektgruppen hadde ikke tid til å skrive om scriptet.

6.4 Utfordringer

6.4.1 Førstegangsinstallasjon av keystone

Tidlig i prosjektet oppsto det problemer med å installere keystone modulen ved bruk av NTNUs puppetkode. Det var veldig mye kode som måtte gjennomgås. Koden var fordelt over 3 GitHub repo-er og det var vanskelig å holde oversikt.

Prosjektgruppen brukte oppdragsgivers dokumentasjon til å prøve å deploye keystone [52]. Det var i utgangspunktet ønskelig å kunne skille ut keystone modulen fra

alle de andre modulene i puppetkoden slik at vi enklere kunne fokusere på en mindre kodebase. Det viste seg at det ikke lot seg gjøre uten å skrive om en del kode. Å gjøre dette ville være veldig tidskrevende og vanskelig, noe som ikke var ønskelig fra vår side. Alternative var å deploye keystone uten å bruke oppdragsgivers puppetkode, men da ville det bli vanskelig å lage tester som skulle teste oppdragsgivers puppetkode.

Keystone modulen er avhengig av mange forskjellige parametre fra Hiera for å kunne bli installert, mange av disse er ikke spesifisert eller dokumentert i NTNUs openstack dokumentasjon [53].

Etter hvert forsøk på å installere modulen feilet den på grunn av manglende eller feilkonfigurerte parametre. Dette ble forsøkt rettet opp ved å implementere parameteret i neste installasjon, men vi fikk aldri til en fungerende installasjon. Siden det ikke er beskrevet hva parametrene skal inneholde, er det veldig vanskelig å konfigurere keystone korrekt.

Etter mye prøving og feiling bestemte prosjektgruppen for å ta kontakt med oppdragsgiver, slik at vi kunne få tilgang til deres Hieara data. Vi fikk tilgang til en versjon av disse dataene og klarte å sette opp en keystone modul som kjørte.

6.4.2 Forsøkt implementert

I dette underkapittelet beskriver vi forsøkte løsninger på installasjon og testing av keystone som ikke ble med i den endelige implementasjonen.

Openstack HEAT med bootscripts

Gruppens første test av å installere keystone stacken automatisk i openstack var å bruke openstack HEAT med bootscripts. HEAT filene opprettet nettverket og startet 3 servere i openstack (*Puppetmaster, Keystone og MySQL*). Bootscriptene installerte og konfigurerte de 3 serverne som ble startet.

Etter mye testing fikk vi til en løsning som klarte å installere programvaren og konfigurere serverne.

Problemet med denne løsninger var at vi ikke kunne sette statisk floating og lokale IP-adresser på serverne. På grunn av dette måtte vi endre Hiera data manuelt på Puppetmaster serveren hver gang vi startet en ny stack. Puppetmasterens IP-adresse måtte også legges til i Keystone serverens Hosts file slik at serveren skulle få sitt sertifikat. Vi valgte derfor å gå videre og prøve å finne andre løsninger på hvordan keystone automatisk kunne bli installert.

Docker compose

Etter å ha prøvd å installere keystone med bruk av openstack HEAT, forsøkte vi oss på å sette opp serverne som containere. Vi lagde Dockerimages for keystone tjensten og Puppetmasteren, og brukte et mysql image for databasen. Vi brukte Docker compose

til å sette alt sammen og lagde et nytt nettverk i compose filen. Siden vi brukte et eget nettverk mellom containere, trengte vi ikke å konfigurere IP-adresser manuelt, løsningen kunne derfor bli helt automatisk.

Vi fikk allikevel problemer med denne løsningen. Når keystone fikk sin puppet konfigurasjon fra Puppetmasteren fikk vi mange feilmeldinger. Siden puppet ikke er lagd for å kjøre i containere gikk vi ikke videre med denne løsningen.

Beaker openstack hypervisor plugin

Beaker har en egen plugin for å starte maskiner direkte i openstack [54]. Vi forsøkte å bruke denne plugin-en siden det hadde gjort installasjon og testing lettere hvis alt ble definert i Beaker HOSTS filen. Vi fulgte oppskriften på GitHub, men vi fikk feilmeldinger.

Beaker klarte ikke å autentisere seg mot SkyHiGh keystone tjeneste. URL-en til denne tjenesten var definert i HOSTS filen som */v3/auth/tokens*. Vi vet at denne URL-en er korrekt siden den fungerer hvis vi bruker Vagrant med samme URL.

Problemet er at Beaker endrer URL-en automatisk ved å legge på */v2/tokens* bakerst i URL-en som medfører at Beaker ikke klarer å autentisere seg. Den fulle URL-en til keystone ble da */v3/auth/tokens/v2/tokens*. Etter mye feilsøking har vi ikke funnet en løsning på dette problemet.

Beaker vagrant_custom hypervisor plugin

Siden openstack plugin-en ikke fungerte, prøvde vi å bruke en forhåndsdefinert Vagrantfile som blir tilkalt av Beaker direkte. Til dette brukte vi Beakers vagrant_custom plugin. Den eneste dokumentasjonen vi fant på hvordan denne skulle brukes er fra en commit på Beaker sin GitLab [55]. Ved å sette hypervisor feltet i Beaker HOSTS filen til ”vagrant_custom” og angi banen til Vagrantfilen i CONFIG feltet skal dette fungere i følge commit-en.

Vi testet løsningen og maskinene ble startet i openstack, men vi fikk feilmeldinger. Feilmeldingene oppsto når Beaker kjørte kommandoen ”vagrant ssh-config”. Denne kommandoen fungerer hvis Vagrant kjøres mot samme Vagrantfile.

Etter mye feilsøking har vi ikke funnet en løsning eller årsak til hvorfor Beaker ikke klarer å kjøre denne kommandoen uten feil, vi måtte derfor velge den ”workarounden” som ble implementert.

Keystone testing med Tempest

For integrasjonstesting av keystone forsøkte vi først å bruke et allerede ferdig rammeverk fra openstack [56]. Tempest testing rammeverket er utviklet for testing mot hele openstack løsningen, men kan også brukes til å bare kjøre tester mot keystone.

Prosjektgruppen fulgte instruksjonene som var beskrevet, men fikk ikke til å kjøre testene. På dette tidspunktet hadde vi lite tid igjen og valgte derfor å lage våres egne

tester.

7 Konklusjon

7.1 Resultat

NTNU ønsket å utforske muligheten automatisk testing har for SkyHiGh. Hvordan automatisk testing kan benyttes for å øke kvaliteten på endringer i kodebasen. Samt om verktøy kan brukes til å utføre kontroll av kodekvalitet og integreringstester. Prosjektgruppen har evaluert og testet en rekke verktøy for å oppnå dette.

Det har blitt utviklet en fullstendig CI/CD pipeline ved bruk av GitLab CI. Pipelinen kvalitetssikrer all endret puppetkode hver gang kode commit-es til repo-et. Til dette brukes verktøy for statisk kodeanalyse som sjekker koden for syntaks-, og stilfeil. Keystone modulen blir installert ved bruk av provisjoneringsverktøyet Vagrant og puppetkoden for keystone akseptansetestet med Beaker. Integrasjonstestene er utviklet av prosjektgruppen som tester modulens API med et Python script. Hvis keystone består alle tester går koden videre til en ny branch ved å bruke automatisk merge, eller merge request. I denne branchen ligger bare kode som har bestått alle tester, og er derfor klar til å enten bli testet manuelt eller bli satt i produksjon. Prosjektgruppen har oppnådd målene ved å implementere den ønskede funksjonaliteten og besvart spørsmålene fra kapittel 1.3 - Oppgavebeskrivelse.

7.2 Videre arbeid

Prosjektets pipeline dekker kun keystone modulen i openstack. Det hadde vært ønskelig om alle moduler i openstack ble deployet og testet. Men for å få til en slik fullstendig dekning kreves en stor mengde arbeidstid. Dette kan bli vanskelig ettersom driften av SkyHiGh hovedsakelig utføres av 2 personer. En mer praktisk løsning vil være å kartlegge hvilke moduler som har det største behovet for grundig testing før de settes ut i produksjon. Kanskje de modulene som feiler oftest. Deretter bør det lages mocks- og stub-systemer som integrerer med disse openstack modulene. Til slutt må det lages tester og modulene må deployes via pipelinen.

Prosjektgruppen undersøke muligheten til å lage unit-tester til SkyHiGh sin puppetkode. Men kom til den konklusjonen at å skrive unit-tester ville bli for tidkrevende. Ettersom det krever et høyt kompetansenivå i Puppet og Ruby. Prosjektgruppen tror de ville måtte bruke en til to uker bare for å lære seg disse verktøyene. Unit-tester til Puppet er noe som bør gjøres av folk som jobber med Puppet og Ruby regelmessig. Et annet problem er at puppetkoden til SkyHiGh ikke er laget for å være kompatibel med Puppet Development Kit (PDK) [57]. PDK er et system som kan styre kjøringen av mange forskjellige tester til puppetkoden. Prosjektgruppen valgte å ikke bruke den begrensede tiden de hadde på å undersøke mulighetene til å gjører eksisterende kode kompatibel med PDK.

For å forbedre brukervennligheten til pipelinen, må pipelinens kjøretid reduseres betydelig. Nåværende versjon tar som regel minst 19 minutter før den er ferdig. Det hadde vært ønskelig om tiden var maks fem minutter. Får å oppnå dette kan man undersøke verdien av å ha programvare allerede installert på serveren. Spesielt installasjon av Puppet og alle Puppemodulene tar lang tid. Men dette vil kreve at man bruker en del tid på å lage gode server-templates som så må vedlikeholdes og versjonskontrolleres. Hver gang en ny programvare eller ny versjon av en programvare brukers, må server-templaten oppdateres.

7.3 Evaluering

7.3.1 Gruppearbeid

Samarbeidet i gruppen har fungert bra. Prosjektgruppen har hatt ukentlige statusmøter hvor de har delt opp oppgaver og diskutert hva som skal gjøres fremover. Trello, et digitalt Kanban brett har blitt brukt som verktøy til å holde oversikt over oppgaver og fordeling av disse. Prosjektgruppen har også hatt jevnlige møter med veileder, som har vært til stor hjelp for oss. Vi har fokusert mye på utvikling av CI/CD platformen og er fornøyd med det endelige resultatet som har blitt oppnådd.

7.3.2 Gjennomføring av oppgaven

Vi hadde en del problemer i starten av oppgaven med å få installert keystone. Vi brukte lang tid på å prøve selv før vi fikk hjelp av oppdragsgiver med Hiera data. Når vi fikk begynne å teste med forskjellige verktøy, gikk det bedre med oppgaven og vi begynte å få resultater av arbeidet. Har også hatt en del problemer med Beaker verktøyet, som beskrevet i kapittel 6.4.2. På grunn av disse problemene har implementasjonen av løsningen tatt lengre tid enn forventet.

7.3.3 Hva har gruppen lært

I denne oppgaven har vi fått bruk for mye av det vi har lært gjennom studiet. Vi har brukt mye Docker for bygging og kjøring av forskjellige containere, bash scripts til å kjøre programmer og kommandoer i pipelinen. Dette er noe vi har brukt tidligere i studiet, men føler vi har lært mye nytt om disse teknologiene. Vi har også lært oss nye teknologier, som bruk av GitLab CI for å lage en pipeline. Dette hadde vi ikke gjort før og her har vi lært mye. Beaker og Vagrant var også teknologier som vi ikke tidligere har brukt. Denne oppgaven har vært en blanding av å bruke tidligere kunnskaper som vi har lært ved NTNU, og å ta i bruk ukjente teknologier for å utføre oppgaven. Vi har også fått mer erfaring av å jobbe sammen i en gruppe for å utvikle et større system. Dette har vært en utfordrende oppgave hvor vi har lært mye.

7.3.4 Alternative løsninger

Dette prosjektet har brukt Vagrant med en openstack plugin for å deploye nye servere i SkyHiGh. Dette var fordi prosjektgruppen opprinnelig prøvde å bruke Beaker til å styre både deployment av servere og testing av keystone. Og Beaker bruker Vagrant til å deploye servere. Det ble etterhvert klart at Beaker ikke kunne brukes med den versjonen

av keystone som ble brukt i SkyHiGh. Men på det tidspunktet hadde gruppen allerede investert mye tid i å få Vagrant til å fungere i Skyhigh. Som et alternativ til Vargrant kan det brukes enten openstack kommandolinje API klient for å deploye et færre antall servere. Eller openstack HEAT til å deploye et større antall servere.

Når det skulle velges en GitLab executor så falt valget på shell modus. Dette var blant annet fordi NTNU sin GitLab instans ikke hadde et tilgjengelig GitLab container-registry på det tidspunktet dette prosjektet vurderte mulighetene som fantes. Hvis et container-registry skulle bli tilgjengelig i fremtiden, kan det være aktuelt å vurdere om det er bedre å bruke Docker executor modus istedenfor shell executor modus.

Noen av de tekniske utfordringene i dette prosjektet var hvordan openstack moduler skulle depoyes og konfigureres. Blant annet på grunn av at openstack ikke har en god måte å dele IP-adresser mellom servere. Det er mulig at dette ville vært lettere hvis openstack modulene kjørte inni containere istedenfor direkte på servere som i dag. Fordi det det da ville være mulig å kjøre flere moduler på en server hvilket betyr at færre servere må lages og færre IP-adresser må deles mellom servere. Skulle man i fremtiden klare å kjøre SkyHiGh inni Kubernetes, så kan man bruke GitLab sin Kubernetes integrasjon [58] til å undersøke om deployment eventuelt blir lettere.

7.4 Avslutning

Prosjektgruppen føler at de har oppnådd det de ønsket med denne oppgaven. Oppgaven viser hvilke muligheter automatisk testing har og prosjektgruppen håper NTNU får nytte av det som har utviklet og undersøkt i dette prosjektet.

Bibliografi

[1] ntnusky. Github repositories. [Internett] Tilgjengelig fra https://github.com/ntnusky. [Besøkt 10. april 2019].

[2] Wikipedia. Idempotence. [Internett] Tilgjengelig fra https://en.wikipedia.org/w/index.php?title=Idempotence&oldid=882959878. [Besøkt 12. april 2019].

[3] Morris, K. 2016. *Infrastructure as Code*. O'Reilly Media, p. 5.

[4] Thomas A. Limoncelli, Strata R. Chalup, C. J. H. 2015. *The Practice of Cloud System Administration. DevOps and SRE Practices for Web Services. Volume 2*. Addison-Wesley, p. 243-272.

[5] Andersen, P. B. automatisering. [Internett] Tilgjengelig fra https://snl.no/automatisering. [Besøkt 15. april 2019].

[6] Morris, K. 2016. *Infrastructure as Code*. O'Reilly Media, p. 183.

[7] aws. What is continuous delivery? [Internett] Tilgjengelig fra https://aws.amazon.com/devops/continuous-delivery/. [Besøkt 12. april 2019].

[8] aws. Continuous delivery vs. continuous deployment. [Internett] Tilgjengelig fra https://aws.amazon.com/devops/continuous-delivery/. [Besøkt 25. april 2019].

[9] Morris, K. 2016. *Infrastructure as Code*. O'Reilly Media, p. 195.

[10] Brown, R. The agile testing pyramid. [Internett] Tilgjengelig fra https://www.agilecoachjournal.com/2014-01-28/the-agile-testing-pyramid. [Besøkt 12. april 2019].

[11] git. Om versjonskontroll. [Internett] Tilgjengelig fra https://git-scm.com/book/no-nb/v1/Komme-i-gang-Om-versjonskontroll. [Besøkt 15. april 2019].

[12] Microsoft. Hva er virtualisering? [Internett] Tilgjengelig fra https://azure.microsoft.com/nb-no/overview/what-is-virtualization/. [Besøkt 16. april 2019].

[13] Wikipedia. Virtuell maskin. [Internett] Tilgjengelig fra https://no.wikipedia.org/w/index.php?title=Virtuell_maskin&oldid=16620888. [Besøkt 16. april 2019].

[14] aws. What is a container? [Internett] Tilgjengelig fra https://aws.amazon.com/containers/. [Besøkt 16. april 2019].

[15] Pecanac, V. Top 8 continuous integration tools [20. februar 2016]. [Internett] Tilgjengelig fra https://code-maze.com/top-8-continuous-integration-tools. [Besøkt 18. februar 2019].

[16] Cloudbees. About jenkins. [Internett] Tilgjengelig fra https://www.cloudbees.com/jenkins/about. [Besøkt 3. mai 2019].

[17] JetBrains. Buy teamcity. [Internett] Tilgjengelig fra https://www.jetbrains.com/teamcity/buy/#license-type=new-license. [Besøkt 18. feb 2019].

[18] JetBrains. Technology awareness. [Internett] Tilgjengelig fra https://www.jetbrains.com/teamcity/features/technology_awareness.html. [Besøkt 18. feb 2019].

[19] JetBrains. Teamcity cloud openstack plugin. [Internett] Tilgjengelig fra https://github.com/yandex-qatools/teamcity-openstack-plugin. [Besøkt 18. feb 2019].

[20] CI, T. Built for every team. [Internett] Tilgjengelig fra https://travis-ci.com/plans. [Besøkt 18. feb 2019].

[21] CI, T. Why choose enterprise over travis-ci.com? [Internett] Tilgjengelig fra https://docs.travis-ci.com/user/enterprise/. [Besøkt 18. feb 2019].

[22] CD, G. [Internett] Tilgjengelig fra https://www.gocd.org/. [Besøkt 18. feb 2019].

[23] S.V, A. What's new in gocd. [Internett] Tilgjengelig fra https://www.gocd.org/2018/10/16/new-gocd-features.html. [Besøkt 18. mai 2019].

[24] GitLab. [Internett] Tilgjengelig fra https://about.gitlab.com/pricing/#self-managed. [Besøkt 18. mai 2019].

[25] GitLab. Using docker images. [Internett] Tilgjengelig fra https://docs.gitlab.com/ee/ci/docker/using_docker_images.html. [Besøkt 18. mai 2019].

[26] CircleCI. Hosting options. [Internett] Tilgjengelig fra https://circleci.com/product/#hosting-options. [Besøkt 18. mai 2019].

[27] CircleCI. Great teams trust use circleci. [Internett] Tilgjengelig fra https://circleci.com/pricing/. [Besøkt 18. feb 2019].

[28] stackshare. Jenkins vs. gitlab ci vs. go.cd. [Internett] Tilgjengelig fra https://stackshare.io/stackups/gitlab-ci-vs-go-cd-vs-jenkins. [Besøkt 18. februar 2019].

[29] Jenkins. Jenkins git repo. [Internett] Tilgjengelig fra https://github.com/jenkinsci. [Besøkt 18. februar 2019].

[30] Simplilearn. What is jenkins? | what is jenkins and how it works? | jenkins tutorial for beginners | simplilearn. [Internett] Tilgjengelig fra https://www.youtube.com/watch?v=LFDrDnKPOTg&feature=youtu.be&t=751. [Besøkt 18. februar 2019].

[31] Jenkins. Pipeline syntax. [Internett] Tilgjengelig fra https://jenkins.io/doc/book/pipeline/syntax/. [Besøkt 18. februar 2019].

[32] Jenkins. Distributed builds. [Internett] Tilgjengelig fra https://wiki.jenkins.io/display/JENKINS/Distributed+builds. [Besøkt 18. februar 2019].

[33] Tong, A. Jenkins reviews product details. [Internett] Tilgjengelig fra https://www.g2crowd.com/products/jenkins/reviews. [Besøkt 18. februar 2019].

[34] TrustedRadius. Jenkins reviews. [Internett] Tilgjengelig fra https://www.trustradius.com/products/jenkins/reviews. [Besøkt 18. februar 2019].

[35] S, A. Your friend, if you are a hacker. [Internett] Tilgjengelig fra https://www.g2crowd.com/products/jenkins/reviews/jenkins-review-131863. [Besøkt 18. februar 2019].

[36] Heusen, K. V. Review: "jenkins provides solid support for continuous integration and continuous delivery". [Internett] Tilgjengelig fra https://www.trustradius.com/reviews/jenkins-2018-04-06-19-21-18. [Besøkt 18. februar 2019].

[37] Padovani, R. A beginner's guide to continuous integration [22. januar 2018]. [Internett] Tilgjengelig fra https://about.gitlab.com/2018/01/22/a-beginners-guide-to-continuous-integration. [Besøkt 18. februar 2019].

[38] Gitlab. Selecting the executor. [Internett] Tilgjengelig fra https://docs.gitlab.com/runner/#selecting-the-executor. [Besøkt 5. mai 2019].

[39] Pundsack, M. Gitlab container registr. [Internett] Tilgjengelig fra https://about.gitlab.com/2016/05/23/gitlab-container-registry/. [Besøkt 5. mai 2019].

[40] Gitlab. The docker executor. [Internett] Tilgjengelig fra https://docs.gitlab.com/runner/executors/docker.html. [Besøkt 5. mai 2019].

[41] Bauer, R. What's the diff: Vms vs containers. [Internett] Tilgjengelig fra https://www.backblaze.com/blog/vm-vs-containers/. [Besøkt 18. mai 2019].

[42] Microsoft. Linux containers on windows. [Internett] Tilgjengelig fra https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers. [Besøkt 23. april 2019].

[43] HashiCorp. Introduction to vagrant. [Internett] Tilgjengelig fra https://www.vagrantup.com/intro/index.html. [Besøkt 7. mai 2019].

[44] Puppet. beaker. [Internett] Tilgjengelig fra https://github.com/puppetlabs/beaker. [Besøkt 7. mai 2019].

[45] openstack. Openstack services. [Internett] Tilgjengelig fra https://www.openstack.org/software/project-navigator/openstack-components/#openstack-services. [Besøkt 9. mai 2019].

[46] wadmiraal. How git hooks made me a better (and more lovable) developer. [Internett] Tilgjengelig fra http://wadmiraal.net/lore/2014/07/14/how-git-hooks-made-me-a-better-and-more-lovable-developer/. [Besøkt 5. mai 2019].

[47] koalaman. read without -r will mangle backslashes. [Internett] Tilgjengelig fra https://github.com/koalaman/shellcheck/wiki/SC2162. [Besøkt 5. mai 2019].

[48] Padovani, R. How to automatically create a new mr on gitlab with gitlab ci. [Internett] Tilgjengelig fra https://about.gitlab.com/2017/09/05/how-to-automatically-create-a-new-mr-on-gitlab-with-gitlab-ci/. [Besøkt 3. mai 2019].

[49] koalaman. Shellcheck. [Internett] Tilgjengelig fra https://www.shellcheck.net/. [Besøkt 3. mai 2019].

[50] Kania, J. How can i make gitlab runner merge code into a branch on a successful build. [Internett] Tilgjengelig fra https://stackoverflow.com/questions/42113402/how-can-i-make-gitlab-runner-merge-code-into-a-branch-on-a-successful-build. [Besøkt 3. mai 2019].

[51] koalaman. sudo doesn't affect redirects. use ..| sudo tee file. [Internett] Tilgjengelig fra https://github.com/koalaman/shellcheck/wiki/SC2024. [Besøkt 4. mai 2019].

[52] Obrestad, E. Bootstrapping - skyhigh [7. desember 2018]. [Internett] Tilgjengelig fra https://www.ntnu.no/wiki/display/skyhigh/Bootstrapping. [Besøkt 6. februar 2019].

[53] Obrestad, E. Hieradata - skyhigh [10. desember 2018]. [Internett] Tilgjengelig fra https://www.ntnu.no/wiki/display/skyhigh/Hieradata. [Besøkt 6. februar 2019].

[54] Puppet. beaker-openstack. [Internett] Tilgjengelig fra https://github.com/puppetlabs/beaker-openstack. [Besøkt 4. mai 2019].

[55] Puppet. Add vagrant_custom hypervisor. [Internett] Tilgjengelig fra https://github.com/puppetlabs/beaker/pull/1215/commits/bb596fa14026fe9b3d9b75ff6b4220f5d6ccbb6c. [Besøkt 4. mai 2019].

[56] openstack. Openstack testing (tempest) of an existing cloud. [Internett] Tilgjengelig fra https://opendev.org/openstack/tempest/. [Besøkt 4. mai 2019].

[57] Puppet. Validating and testing modules. [Internett] Tilgjengelig fra https://puppet.com/docs/pdk/1.x/pdk_testing.html. [Besøkt 16. mai 2019].

[58] Gitlab. Kubernetes + gitlab. [Internett] Tilgjengelig fra https://about.gitlab.com/solutions/kubernetes/. [Besøkt 16. mai 2019].

[59] Nemytchenko, I. Gitlab ci: Run jobs sequentially, in parallel or build a custom pipeline. [Internett] Tilgjengelig fra https://about.gitlab.com/2016/07/29/the-basics-of-gitlab-ci/. [Besøkt 23. april 2019].

A Forprosjekt

Forprosjekt - CI/CD for SkyHiGh

Mats Ove Mandt Skjærstein (997831)
Alexander Jakobsen (473151)

30. januar 2019

Innhold

1 Mål og rammer

1.1 Bakgrunn

NTNU gjøvik drifter en instans av Openstack. Periodisk blir det rullet ut nye versjoner av Openstack, dette gjøres ved bruk av system-konfigurasjonssystemet Puppet. For å sikre at alle systemets funksjoner fungerer som de skal, er det hensiktsmessig å teste funksjonene til den nye versjonen før systemet settes ut i produksjon. All slik testing av funksjonalitet gjøres i dag via manuell testing. Dette er en tidskrevende og feilutsatt prosess. Det er oppdragsgivers ønske at fremtidig testing gjøres automatisk.

1.2 Prosjektmål

Målet for prosjektet er å undersøke til hvilken grad automatisert testing kan brukes til å teste utrulling av Openstack via oppdragsgivers eksisterende Puppet kode.

Hvis automatisert testing kan gjennomføres, så skal vi vurdere forskjellige metoder dette kan gjøres på. For så å lage en prototype av et automatisert testsystem. Til slutt må det lages en vedlikeholdsplan for systemet

1.3 Rammer

Etter en samtale med oppdragsgiver har vi kommet frem til følgende rammer:

- Det er ikke aktuelt å kjøpe inn lisenser til programvare.
- Vedlikeholbarhet og robusthet er viktigere en omfattende funksjonalitet.
- Det er ikke aktuelt å bruke store ressurser på å drifte testsystemene.
- Systemer som skal kjøre automatiserte tester skal konfigureres ved bruk av Puppet.

2 Omfang

2.1 Fagområde

Hovedområdet for oppgaven er continuous integration/continuous delivery (CI/CD) pipeline. Dette innebærer bruken av automatiske tester for OpenStack implementasjonen "SkyHiGh" for NTNU.

CI/CD er en praksis som tar endret kode gjennom en pipeline som består av en rekke automatiserte tester. Gjennom denne praksisen kan flere feil oppdages før de går ut i produksjonsmiljøet.

OpenStack er en Open Source skytjeneste som består av et sett med programvare verktøy for bygging og kontrollering av skytjeneste platformer. NTNU bruker dette for sin private skytjeneste, som gir studenter og ansatte en platform hvor den kan lage virtuelle maskiner og nettverk.

Vi vil komme inn på flere fagområder under oppgaver, som:

- Programvareutvikling:
  For utvikling av automatiske tester, og oppretting av tjenester ved bruk av konfigurasjonssyringsverktøyet Puppet.

- Drift av skytjenester:
  Design og opprettelse av infrastrukturen som kjører de automatiske testene.

2.2 Avgrensninger

Vi begrenser oss til å skrive automatiske tester for deler av OpenStack kodenbasen, ikke å gjøre hele kodebasen klar for en CI/CD pipeline.

2.3 Oppgavebeskrivelse

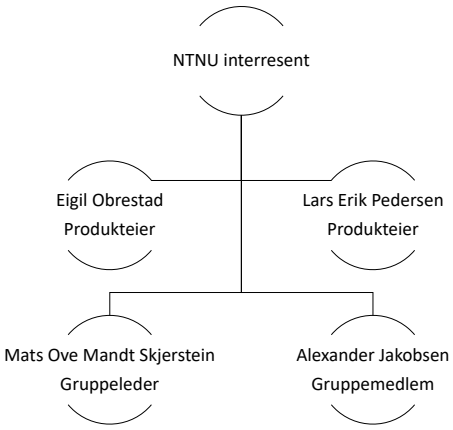
Oppgavens formål er å utforske muligheten for automatisk testing for OpenStack koden. Evaluere hvordan automatisk testing kan benyttes for å øke kvaliteten på denne koden i en continuous integration/continuous delivery (CI/CD) prosess ved endringer av kode. Dette innebærer å undersøke, teste, og evaluere ulike teknologier og metoder for å sette opp en CI/CD pipeline hvor forskjellige automatiske tester kjører. Formålet med CI/CD prosessen er å forsikre seg for at endringer ikke bryter ønsket funksjonalitet i OpenStack, og for å øke kvaliteten på de endringene som skjer.

Det skal utvikles en prototype som bruker en CI og/eller CD pipeline til å rulle ut nye endringer til produksjonsmiljøet ved endringer i kodebasen. Det er ønskelig at en slik prosess skal minimere bruk av manuelle tester og fange opp feil før de blir rullet ut i produksjonsmiljøet.

All infrastruktur som settes opp for å gjøre automatisk testing skal kunne installeres av konfigurasjonssystemet Puppet.

3 Prosjektorganisering

3.1 Ansvarsforhold



Organisasjonskart:
Eigil og Lars Erik er produkteierne til SkyHigh og SkyLow. De fungerer som oppdragsgivere for prosjektet.

Mats vil fungere som prosjektleder og vanlig gruppemedlem. Og vil med det ha ansvaret for kommunikasjonen mellom gruppen og produkteierne og prosjektveileder.

Alexander vil fungere som gruppemedlem.

3.2 Rutiner og regler for gruppen

Gruppen har blitt enige om en sett med regler for gjennomføringen av prosjektet:

- Gruppemedlemmene skal bruke minimum 27 timer pr. arbeidsuke på prosjektet.
- Gruppemedlemmene skal på mandag og fredag jobbe fra kl. 09:00 til 16:00. Ukens resterende 13 arbeidstimer kan gjennomføres når som helst ellers i den aktuelle uken.
- Fritak fra de faste tidene må bli avtalt 1 uke i forveien, og den tapte arbeidstiden må jobbes inn innen maks 2 uker.
- Prosjektkostnader som ikke blir dekt av NTNU deles opp likt mellom gruppemedlemmene. Kostnadene må dokumenteres ved kvitering, og innkjøpet må være skriftlig godkjent av alle gruppemedlemmer på forhånd.
- Et gruppemedlem kan jobbe inn fri ved å jobbe mer en 27 timer på en uke. Men maks 14 timer.
- Sykdom anses ikke som tapt tid, men må jobbes inn senere.
- Gruppemedlemmene skal loggføre arbeidestiden.
- Hver fredag kl. 9:00 skal gruppen ha et statusmøte. Det skal skrives møtereferat.
- Hver onsdag kl. 9:30 skal gruppen ha et møte med veileder. Det skal skrives møte-

referat og alle gruppemedlemmer skal være tilstede om ikke annet er avklart på forhånd.

4 Planlegging, oppfølging og rapportering

4.1 Hovedinndeling av prosjektet

Oppdragsgiver har ingen faste krav til hvordan oppgaven skal utføres. Det er opp til oss å undersøke og velge ulike teknologier og oppsett som vi kan evaluere. Siden oppgaven er åpen og vi kan velge de teknologiene vi mener passer best, så vil en smidig utviklingsmodell passe oss best.

4.2 Valg av SU-modell/prosessrammeverk med argumentasjon

Vi har valgt å bruke en smidig utviklingsprosess som består av Kanban, med ekstra kontroll på hvilke faser hver oppgave skal bestå av. Vi vil da bruke en hybrid versjon av Kanban som gir mer struktur i prosjektperioden, men som samtidig er fleksibel nok til at 2 personer kan klare å levere resultater.

Kanban gir oss muligheten til å være mer åpen og teste ut flere muligheter når vi jobber med oppgavene vi setter oss. I motsetning til Scrum, så har vi mulighet til å endre og reprioritere oppgaver underveis. Siden oppgaven baserer seg på mye undersøking av ulike teknologier, er det en fordel å bruke et rammeverk som Kanban til både prosjektstyring og oppfølging av oppgavene underveis.

Valg av metode og tilnærming

Vi tar i bruk Trello som et digitalt Kanban brett, hvor vi fordeler og legger inn nye oppgaver underveis i Kanban kort. Vi har ingen faste tidsfrister på oppgavene, men forsøker å forholde oss til Gantt-skjemat for å ikke bruke for mye tid på oppgavene.

Hver oppgave som defineres skal legges inn i Kanban brettet. Brettet består av de vanlige Kanban kolonnene som «Backlog», «ToDo», «In progress», «To review» og «Done». Kortene skal alltid flyttes til de korrekte kolonnene, slik at vi kan holde god oversikt over oppgavene vi jobber med og de som gjenstår.

I tillegg til disse kolonne, har vi implementert 3 nye kolonner som skal gi mer struktur når vi jobber med de tekniske oppgave. Disse tre fasene består av:

- Fase 1 - Lag tester:
  I denne fasen skriver vi alle testene for modulen/tjenesten vi har valgt.

- Fase 2 - Automatiser:
  Modulen/tjenesten legges inn i CI/CD pipelinen slik at testene kjører automatisk ved endringer.

- Fase 4 - Skriv rapport:

Vi skriver bacheloroppgaven underveis slik at vi ikke glemmer eller utaleter noe i den endelige rapporten.

Alle tekniske oppgaver som til slutt skal legges inn i en CI/CD pipeline skal bevege seg gjennom disse fasene. En teknisk oppgave består av å implementere en av OpenStacks moduler/tjenester inn i en CI/CD pipeline.

Vi bruker Kanbans «Work In Progress» (WIP), med maks 4 kort i hver kolonne om gangen. Ved å bruke WIP skal vi unngå å sitte med for mange oppgaver samtidig.

4.3 Plan for status møter og besluttnigspunkter i perioden

Gruppen har planlagt møte med veileder hver uke i startfasen. Det er avklart at hyppigheten av møter kan endres underveis i prosjektet.

Det er ikke planlagt faste møter med oppdragsgivere, men disse kan planlegges underveis. Oppdragsgiver vil være med på noen av veileder-møtene underveis.

Gruppen vil ha statusmøter 1 gang i uken hvor vi diskuterer fremgangen vi har gjort. Flere møter blir planlagt ved behov. Beslutninger underveis som ikke krever avklaring med veileder eller oppdragsgivere vil gjøres underveis i gruppens egne kommunikasjonskanaler.

5 Organisering av kvalitetssikring

5.1 Dokumentasjon, standardbruk og kildekode

All kode som vi lager selv, skal versjonkontrolleres i github, og koden skal være godt kommentert.

All tekst som skal brukes i den endelige rapporten skal skrives inn i overleaf. Gruppemedlemmene kan bruke andre tekstbehandlingsverktøy til arbeidsnotater, men gruppemedlemmene må i så fall ukentlig konvertere relevante tekster til overleaf sitt latex format. Dette er for sikre en god arbeidsflyt samtidig som det forhindre at det lages store mengder med tekst som ikke er i et inkompatibelt format.

Verktøytabell:

Navn	Formål
GitHub	Versjonskontroll
Overleaf	Latex text editor
Toggle	Timelogging
Openstack	Skyplatform
Puppet	Systemkonfigurasjon
TeamGantt	Gantt-skjema

5.2 Risikoanalyse

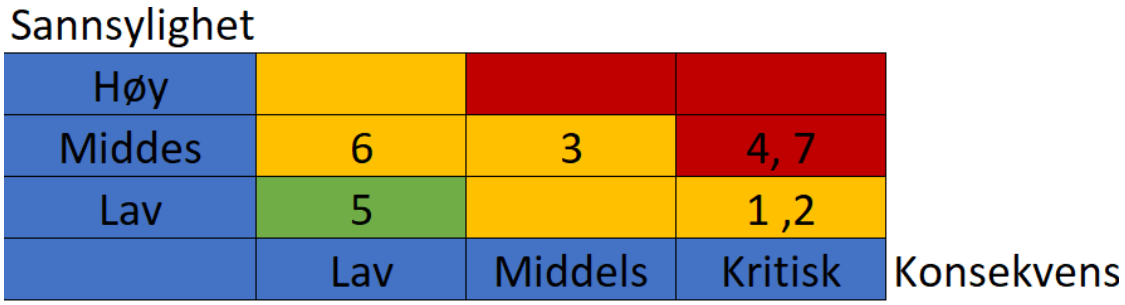
Denne seksjonen omhandler hendelser som utgjør en risiko for at prosjektet ikke når sine mål.

I tabbelen under er hendelser omtalt som 'Risiko'. Kollonen 'Nummer' brukes til å koble hendelsestabellen med tiltaksplan tabellen. Kollonen 'Sannsynlighet' viser sannsynligheten for at en hendelse forekommer. Sannsynlighet er delt i 3 forskjellige kategorier. Fra minst til mest sannsynlig har vi: 'Lav', 'Middels' og 'Høy'. Kollonen 'Konsekvens' kan ha 3 forskjellige verdier. Fra minste til høyeste konsekvens: 'Lav','Middels' og 'kritisk'.

Risikotabell

Nummer	Risiko	Sannsynlighet	Konsekvens	Tiltak
1	Sykdom blant gruppemedlemmene	Lav	Kritisk	Ja
2	Oppdragsgiver avbryter prosjektet	Lav	Kritisk	Nei
3	Deler av arbeidet er for komplisert for oss	Middels	Middels	Ja
4	Arbeidsdokumenter / artifakter går tapt	Middels	kritisk	Ja
5	Store endringer i prosjektmålene	Lav	Lav	Nei
6	Uenigheter om hvordan oppgaver skal løses, eller veien fremover resulterer i at betydelig arbeidstid går tapt, hvilket gjør at vi ikke når tidsfrister	Middels	Lav	Ja
7	Vi klarer ikke å lage et representativt testmiljø	Middels	Kritisk	Ja

Risikomatrise

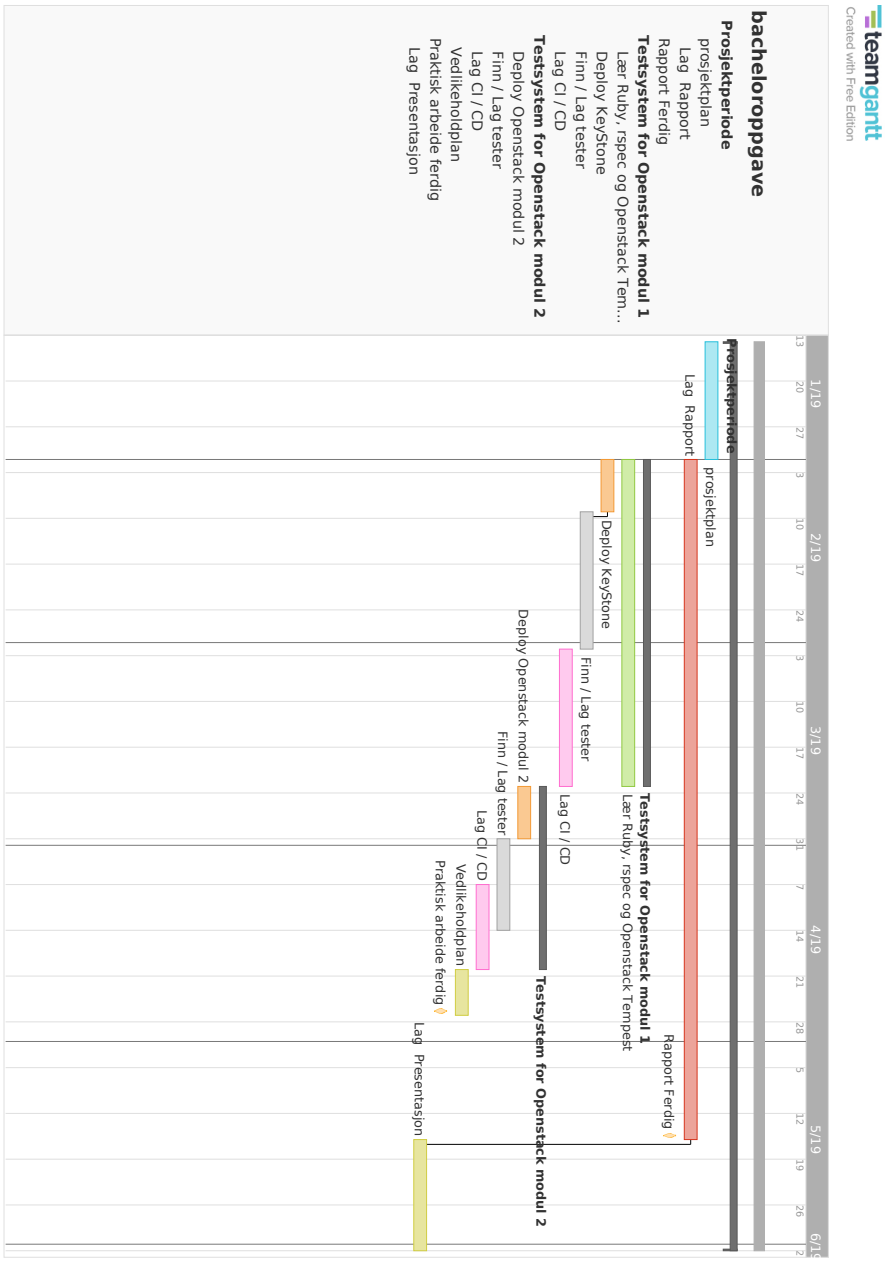


TiltaksTabell

Nummer	Tiltak
1	Ved sykdom i gruppen vil resterende gruppemedlem og oppdragsgiver måtte lage en liste over hvilke oppaver som skal flyttes fra produkt backlogtil en "hvis det blir tid backlog"
3	Hvis betydelige deler av arbeidet blir for komplekst for oss, så må vi gå enten søke ekstern kompetanse eller skalere ned prosjektets ambisjoner
4	For å forhindre at arbeidsdokumenter går tapt, så skal gruppemedlemmene ta hver sin lokale kopi av hele prosjektet hver uke. Alt som lagres på Overleaf skal bli tatt backup av til GitHub minst en gang pr. arbeidsdag
6	Betydelige uenigheter innad i gruppen må tas opp men veileder. Med det målet at et kompromiss kan oppnåes.
7	Hvis vi har problemer med å etablere et testmiljø, så vil første tiltak være å utvide tidsfristen for få laget det. Det andre tiltaket vil være å søke ekstern kompetanse

6 Plan for gjennomføring

6.1 Gantt-skjema



6.2 Liste over aktiviteter(work breakedown structure)

I denne seksjonen listes opp alle hovedaktiviteter som vil gjøres i prosjektet.

- Finn og vurder relevante teknologier relatert til automatisert testing av Openstack.
- Kartlegge hva som skal testes og prioriteringsrekkefølgen.
- Opparbeid relevant teknisk kompetanse.
- Lag eget eller få tilgang til eksisterende testmiljø.
- Kartlegge hva slags type tester som er hensiktsmessing for oppdragsgiver.
- kartlegge hva som kan gjenbrukes fra Openstacks eksisterende test systemer.
- Integrer eksisterende elle lag egne tester for utrulling av Openstack.
- Lag vedlikeholdsplan for testmiljø.
- Lære oss Ruby, rspec, Openstack Tempest.

6.3 Milepærer

- Når alt det praktiske arbeidet er ferdig.
- Når rapporten er ferdig.

6.4 Tids- og ressursplan

Det er vanskelig å estimere tidsforbruket til hver oppgave. Dette er fordi vi ikke har utført en liknende oppgave tidligere. Vi klarer heller ikke å finne en mal for hvordan en liknende oppgave har blitt utført tidligere. Vi finner eksempler på hvordan automatisert testing blir brukt for programvareutvikling, men ikke for å verifisere utrulling av et system via et systemkonfigurasjonsverktøy som Puppet. Vi har tidlige hørt om bruk av CI og CD fra noen systemadministrasjons kurs vi har hatt. Men ingen av dem har gitt en guide på hvordan denne oppgaven skal løses.

Vi har en fleksibel timeplan. Hvis det viser seg at dette er ekstremt vanskelig så kommer vi til å kun fokusere på testsystemet til en Openstack modul, istedenfor to. Vi har etter samtaler med oppdragsgiver bestemt oss for å starte med modulen ”KeyStone”. Den skal være den enkleste modulen, og krever kun at vi setter opp en database for at den skal kunne fungere.

Vi vil bruke Puppetkoden(manifest) til oppdragsgiver for å rulle ut Openstack modulene. Hvis ting går greit så vil vi prøve det vi har lært på en annen Openstack modul. Da med en mye kortere tidsfrist.

Det er sannsynlig at vi må lære oss Ruby og testrammeverket rspec for å kjøre tester med Puppet. Vi skal undersøke et eksisterende open-source integrasjons testrammeverk for Openstack som heter Tempest. Hvis det er lovende vil vi måtte lære oss nok Python til å kunne lese koden til Tempest.

bacheloroppgave

			Start	Due
▼ **Prosjektperiode**	**0%**			
prosjektplan	0%		14 January 2019	Thursday
Lag Rapport	0%		1 February 2019	15 May 2019
Rapport Ferdig	☐		15 May 2019	15 May 2019
▼ **Testsystem for Openstack modul 1**	**0%**			
Lær Ruby, rspec og Openstack Tempest	0%		1 February 2019	22 March 2019
Deploy KeyStone	0%		1 February 2019	8 February 2019
Finn / Lag tester	0%		9 February 2019	1 March 2019
Lag CI / CD	0%		2 March 2019	22 March 2019
▼ **Testsystem for Openstack modul 2**	**0%**			
Deploy Openstack modul 2	0%		23 March 2019	30 March 2019
Finn / Lag tester	0%		31 March 2019	13 April 2019
Lag CI / CD	0%		7 April 2019	19 April 2019
Vedlikeholdplan	0%		20 April 2019	26 April 2019
Praktisk arbeide ferdig	☐		26 April 2019	26 April 2019
Lag Presentasjon	0%		16 May 2019	1 June 2019

1 Kontrakt

Mellom Alexander Jakobsen og Mats Ove Mandt Skjerstein

1.1 Regler for bacheloroppgave

- gruppe medlemmene skal bruke minimum 27 timer pr arbeidsuke på prosjektet
- Prosjektkostnadder som ikke blir dekt av NTNU deles opp likt mellom gruppemedlemmene. Men kostnaddene må dokumenteres ved kvitering. og innkjøpet må være skriftlig godkjent av alle gruppe medlemmer på forhånd.
- gruppemedlemmene skal på mandag og fredag jobbe fra kl. 09:00 til 16:00. ukens resterende 13 arbeidstimer kan gjennomføres når som helst ellers i den aktuelle uken.
- fritak fra de faste tidene må bli avtalt 1 uke i forveien. og den tapte arbeidstiden må jobbes inn innen maks 2 uker.
- et gruppemedlem kan jobbe inn fri ved å jobbe mer en 27 timer på en uke. Men maks 14 timer.
- sykdom anses ikke som tapt tid man må jobbe inn senere
- gruppemedlemmene skal loggføre arbeidestiden
- Hver fredag kl. 9:00 skal gruppen ha et statusmøte. Det skal skrives møtereferat.
- Hver onsdag kl. 9:30 skal gruppen ha et møte med veileder. Det skal skrives møtereferat. Alle gruppemedlemmer skal være med.

1.1.1 Signaturer

Alexander Jakobsen Alexander Jakobsen
Dato 30/1-19

Mats Ove Mandt Skjerstein Mats Ove Mandt Skjerstein
Dato 30/1-19

B Møtereferater

Møte med Eigil, Lars Erik og Erik Hjelmås 16.01.19

Deltagere: Alexander, Mats Ove, Lars Erik,Erik Hjelmås
Tid: 09:30 til 10:00
Sted: NTNU Gjøvik, rom A117

Agenda

Diverse spørsmål om oppgaven.

Referat

Vi spurte om vi kunne få tilsendt noen relevante tidligere bacheloroppgaver. Erik skulle sende oss noen og mener Evry oppgaven for 2 år siden var mest relevant for oss. Det finnes ingen oppgaver som har tatt for seg CI/CD før.
Vi forespurte om relevant litteratur til oppgave. Erik mener bøkene ”Infrastructure as Code” og ”Cloud Native DevOps with Kubernetes” er mest relevant. Sistnemte bok kommer ut i februar 2019.
Bacheloroppgaven kan skrives på Norsk hvis vi ønsker dette, men Erik anbefaller at vi skriver på engelsk.
Vi spurte om de hadde noen anbefalinger av hvilke prosessrammeverk som vi burde bruke. De mener vi burde velge en smidig modell, med bruk av Kanban, gjerne en hybrid.

Møte Med Egil og Lars Erik 21.01.19

Deltagere: Alexander, Egil, Lars Erik
Tid: 13:00 til 13:20
sted: NTNU gjøvik rom A117

Agenda

- OpenStack Tempest
- SkyLow, eventuelt hvordan vi skal lage vårt eget testmiljø
- prosjektavtale

Referat

Egil anbefalte at vi forker deres repo og bruker det til å starte diverse openstack tjenester i SkyHigh. deretter skal vi kjøre tester mot disse tjenestne.
Egil forelo at vi starter med å teste module til KeyStone først. han mente at det var den enklese modulen. alt den trenger en DB (f.eks. MySql).
vi har fått et prosjekt i SkyHigh. Med kvote for 16 instansr. hvis vi trenger større kvote så skal vi kontakte dem.
Egil anbefalte at vi tar en grundig vurdering av nytten til OpenStack Tempest før vi dedikerer oss fult til det. Lars Erik var usikker på om det ville være nyttig vårt bruk. han

trodde det var mer egnet til utviklig av OpenStack enn validering av konfigurasjonen til en instanser av OpenStack.
egil og Lars Erik sa vi skulle stikke innom når som helt med en fysik kopi av Prosjektavtalen så skulle de signere den. de hadde ingen punkter de ville tilføre avtalen.

Møte Med Erik hjelmås 22.01.19

Deltagere: Alexander, Erik Hjelmås
Tid: 9:30 til 10:00
sted: NTNU gjøvik rom A117

Agenda

- OpenStack Tempest
- utviklingsmodell
- fremdriftsplan
- prosjektavtale

Referat

vi ble enige om at OpenStack Tempest var en kandidat for test automatisering, men han trodde ikke det var fasiten for hele oppgaven.
erik trodde at valg av git platform var veldig relevant. noen pipeline platformer kan kun brukes for containere. mens vi må ha en som kan kjøre VM-er. han nevnte at Beakerbbrukes til dette. for eksempel git bruker Travis CI". er det mulig vi må ha en egen instans av GitLab for å kunne kjøre ønsket pipeline?
Erik mente at målet vårt burde være å få til en testpyramide
erik mener vi bør skrive raporten kontinuelig gjennom hele prosjetet, altså ikke bare på slutten for eksempel kan vi ha som mål å skreve 2 sider i uken.
Erik mente det var en god ide å starte med å rulle ut en enkel OpenStack modul som KeyStone"han foreslo følgende arbeids form:
-> sett opp openstack modul -> finn/lag tester -> automatisering av tester ->
|———<————————<————————————<————————————<————————<————|

dette medfører at vi må endere eksisterende fremdriftsplan.
erik mener bør ha en mer strukturert utviklingsmodell en det vi har i dag. det er bedere å ha for mye struktur i begynnelsen. for så å slakke av senere.
Erik sa at for publisering av batcheloroppgaver så måtte det lages en separat avtale om det. det er altså ikke noe vi må ta med i prosjektavtalen.
erik nevnter 2 rammeverk for tesing som vi burde se på: inspec : https://www.inspec.io/ serverspec : https://serverspec.org/

Møte med Erik Hjelmås og Egil 30.01.19

Deltagere: Alexander, Mats Ove, Lars Erik,Erik Hjelmås
Tid: 09:30 til 10:00
Sted: NTNU Gjøvik, rom A117

Agenda

prosjektavtale
prosjektplan

div om bacelorrapport.

Referat

Vi er ferdige med prosjektporten, Eirik sa at vi skulle maile den til han, så skulle han vurdere/ godkjenne den. Erik sa at det vil være lettere å gjøre tester med kontainere. Men skyHigh er pr i dag drevet uten kontainere. Men han nevnte at vi måtte være kritiske til kontainere vi finner på dockerhub. Han mente at det var mye dårlig der. Vi spurte om hva slags lengde bachelor rapporten skal ha. Erik sa ca. 50 sider. Men generell regel er 30 til 70 sider. Egil viste oss et diagram over komponentene i skyhigh. Og as at hvis vi googlet openstack arkitektur så finner vi diagrammet. Han sa også at de brukte MySql som deres keystone database. Han nevnte også at automatisert testing av puppet koden deres var et mål.
Erik sa at den ultimate integrasjonstest på slutten var at testsystemet brukte et token fra keystone til å gjøre noe med an annen modul Egil signerte prosjektavtalen. Lars Erik signerte etter møtet.

Møte med Erik Hjelmås og Egil 6.02.19

Deltagere: Alexander, Mats Ove, Lars Erik, Erik Hjelmås
Tid: 09:30 til 10:00
Sted: NTNU Gjøvik, rom A117

Agenda

problemer med å deploye keystone
rapportstruktur

Referat

Alexander spurte hvordan vi burde jobbe med rapporten underveis. Lurte på om vi burde skrive tekster som kan settes rett inn i rapporten. Eller om vi bare skulle skrive arbeidsnotater. Erik sa at som et minimum så må vi lage punktlister på hva vi driver med. Vi kunne gjerne lage tegninger for å forklare ting. Mange tegninger var bra sa han.

Alexander og Mats fortalte at de sto fast i forsøket på å deploye Openstack modulen KeyStone. Hver gang de prøvde å deploye fikk de feilmelding på missing parametre. Vi ble enige om at vi måtte ha en sesjon sammen med enten Egil eller Lars Erik for å se hvordan de løste det i deres utrulling. Erik foreslo at vi lagde en liten case av dette problemet som vi skulle bruke i rapporten under kapitelet implementasjon.

Erik sa at vi burde bestemme strukturen på rapporten(kapitler) på et tidlig tidspunkt. Mats nevnte en mal fra innsida. Erik så at vi kunne bruke den som utgangspunkt.

Møte med Egil og Lars Erik 12.2.19

Deltagere: Alexander, Mats, Egil og Lars Erik
Tid: 12:30 til 13:30
Sted: NTNU gjøvik rom A117

Agenda

Konfigurasjon av Openstack

Referat

Egil og Lars Erik hadde vasket konfigurasjonsfilene sine til Openstack infrastrukturen i løpet av den siste uken. De presenterte oss med en link til filene (filer.rothaugane.com/hiera.tgz) resten av møtet brukte vi på å gå gjennom filene. Vi fikk beskjed om at vi skulle kontakte dem hvis det var noe vi lurte på.

Møte Alexander, Mats, Erik 20.2.19

Deltagere: Alexander, Mats
Tid: 09:30 til 10:00
Sted: NTNU gjøvik rom A117

Agenda

Skriv om CI platformer
veien videre

Referat

Erik sa at vi ikke skulle bruke tid på a sette opp en ci server. Det er nå et ønske om at vi fokuserer på gitlab som CI platform. Han foreslo at vi bruker tid på å finne eksempler på pipline filer. Kanskje CERN har gode eksempler
Han mente at vi kun bør skrive kort og konsist om CI platformer. En mulighet kunne være at vi vurder om vi burde lager et eget REPO som koordinerer/ snakker med andre REPOER.
Til veien videre foreslo Begynn å jobbe med pipelinefilen. Erik mente at å skrive en kompleks pipeline fil bør være målet for dette prosjektet. Vi må finne ut hva en CI server skal gjøre (jobber, stages, fra kode til deploy?) lær terminologien til pipeline leverandører.

Møte Mats og Erik 27.02.19

Deltagere: Mats
Tid: 09:30 til 10:00
Sted: NTNU gjøvik rom A117

Agenda

GitLab CI og hva som skal gjøres videre.

Referat

Erik mener vi burde prøve å få til GitLab CI, da dette er den letteste CI-en å bruke. Erik mener NTNU has sin egen GitLab som vi kan prøve å bruke (http://gitlab.stud.iie.ntnu.no). Hvis dette ikke fungerer, prøv en GitLab installasjon i SkyHigh (https://about.gitlab.com/install/), dette skal være gratis.

Erik mener dette bude være fokuset våres, vi burde sette av litt tid fremover til å få GitLab til å kjøre med runners i SkyHigh. Det finnes puppet moduler for å installere både GitLab og GitLab runners, men å få dette til å fungere med Puppet burde ikke være førsteprioriteten nå.

Hovedfokuset i oppgaven er å lage en storpipeline fil, helst i GitLab som tester mer enn bare linting, validering og unit-testing".

Vi burde prøve å installere og teste keystone med beaker etter at pipelinen er satt opp.

Møte Alexander, Mats, Erik 06.03.19

Deltagere: Alexander, Mats
Tid: 09:30 til 10:00
Sted: NTNU gjøvik rom A117

Agenda

Tilbakemelding på skriving, måte og innhold.
Hva vi burde fokusere på fremover.

Referat

Tilbakemeldinger fra Erik på rapportskriving: Begrunn alle valg som har blitt tatt. Påstander i teksten burde støttes opp med kilder. Ikke skriv "vi", "oss", osv. Rapporten skal ikke være en historie over hva som har blitt gjort. Pass på skrivefeil, spesielt ord fordeling. Bruk tid på å formulere setninger, skal ikke være på et muntlig språk.

Erik mener vi burde fortsette videre med den øvrige delen av test pyramiden. Dette innebærer stating av koden til et testings miljø hvor det blir utført smoke-tester og integrasjonstester. Her må vi se hvordan andre gjør det, slik at vi har noe å basere oss på.

Møte Alexander og Erik 13.3.19

Deltagere: Alexander, Erik
Tid: 09:30 til 10:00
Sted: NTNU gjøvik rom A117

Agenda

vagrant
baker
dokumentere feilsøking

Referat

Alexander fortalte Erik at vi hadde klart å deploye til skyHigh ved bruk av vagrant både i og uten container. Men han sa at hvis man brukte container så var det ikke mulig å gjøre server provisionering ved bruk av puppet. Man måtte bruke shell. PGA dette var det best å kjøre vagrant direkte på en VM. Men at vi slet med beaker.

Erik foreslo at vi kunne søke hjelp fra andre. Men han visste ikke om noen på skolen som kunne noe om beaker. Og hvis vi ikke lyktes så måtte vi dokumentere at vi hadde foretatt en systematisk feilsøkingsprosess. Hvordan forenkle, googling, søke hjelp fra andre og div forsøk på løsninger. Det kunne kanskje være lurt å ha en tidsfrist for når

vi skal gi oss hvis vi ikke kommer noe lengre.

Møte Alexander, Mats og Erik 20.03.19

Deltagere: Alexander, Mats, Erik
Tid: 09:30 til 10:00
Sted: NTNU gjøvik ukjent møterom

Agenda

Beaker problemer og dagens løsning

Referat

Viste fremgangen vi har hatt i Beaker til Erik, og snakket om problemer som har oppstått ved bruk av Beaker.

Mats har oppdaget flere feil i Beaker som har gjort prosessen vanskelig. Beaker fungerer ikke med ”openstack plugin” og ”vagrant_custom”. På grunn av dette ble det utviklet en ”workaround” metode som vi i dag bruker (shell script).

Erik mener at vi må dokumentere hvordad vi kom frem til denne løsningen og hvordan den fungerer. Kunne også melde inn bugs til openstack slik at dette kan bli fikset i fremtiden.

Vi spurte Erik om hvordan vi kan hente ut floating IP fra openstack fra en server, Erik ga oss en løsning på dette ved å bruke facts i Puppet. Eksempel: ”$ip = $::facts[’ec2_metadata’][’public-ipv4’]”.

Erik ønsker at vi inviterer oppdragsgivere til neste uke og viser frem en demo av hva vi har så langt.

Møte demo 27.3.19

Deltagere: Alexander, Erik, Egil og Lars Erik
Tid: 09:30 til 10:00
Sted: NTNU gjøvik ukjent møterom

Agenda

demonstrasjon av arbeidet så langt
tilbakemeldinger fra oppdragsgiver

Referat

Viste frem det vi hadde laget så langt. Alexander startet pipelinen, og forklarte koden vår

Tilbakemeldinger: Eirik foreslå å bytte ”;” med ”&&” i .gitlab-ci.yml filen, seksjon

'merge'. Eigil ønsker seg noe som kan kjøre lokalt, slik at hver gang. Hvis vi skulle skrive tester framover så ønsket de at vi brukte bash, python eller ruby.

Alexander nevnte at vi har igjen en del arbeid på områder som sikkerhet, ytelse og kodekvalitet.

Møte Alexander, Mats og Erik 10.4.19

Deltagere: Alexander, Mats
Tid: 09:30 til 10:00
Sted: NTNU gjøvik rom A117

Agenda

forbedringer av kode
rapport
sikkerhet

Referat

alexander og mats fortalete hva de hadde gjort siden forrige gang vi møttes(Demoen), de hadde laget api tester, gitlab merge request scrip, git pre-commit script og div forbedring av eksisterende kode.

erik ønsket at vi lagde et uttrdrag av rapporten så lang og leverer den til han innen neste torsdag. vi ble enige om at vi skulle gjøre det.

erik mente at vi ikke burde ha noe hemelig i gitrepoet. vi burde istede ha det på serverne(gitlab-runnerene). erik foreslo at vi eventuelt skulle lage feature request til gitlab for å få bedere sikkerhetsfearues i fremtiden.

C Statusmøter

Møte Alexander og Mats 25.01.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

møter med oppdragsgiver og veileder
prosjektplan
automatiserte testverktøy

Referat

Vi diskuterte veileders forslag til forbedring av forprosjektplan. Og ble enige om å gjøre noen endringer. Alexander vil lage nytt Gantt-skjema. På bakgrunn av de punktene vi ble enige om på møtet. Alexander lager en risikomatrise til eksisterende risikotabell. Mats vil finne og skrive om hvordan vi kan få mer struktur på prosjektet.

Deretter diskuterte vi verktøy for automatisert testing. Blant annet Serverspec, inSpec, Chef kitchen, Puppet-kitchen. Vi ser det som sannsynlig at vi må lære oss Ruby og rspec for å lage/kjøre tester til Puppet.

Møte Alexander og Mats 01.02.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

installasjon av keystone
karrieredag

Referat

Vi snakket om hvordan vi skulle gå frem for å få installert keystone ved bruk av oppdragsgivers puppet kode. Vi bestemte oss for å prøve å bruke oppdragsgivers dokumentasjon på hvordan dette kunne gjøres https://www.ntnu.no/wiki/display/skyhigh/Bootstrapping. Vi bestemte oss for å jobbe sammen inntil vi trodde vi kunne begynne å rulle ut infrastruktur. Deretter skulle vi dele oss for å prøve oss frem hver for oss.
Vi bestemte oss for å forskyve mandagens arbeidsøkt til onsdag, slik at vi begge kunne delta på karrieredagen.

Møte Alexander og Mats 08.02.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

Problemer med å deploye Keystone
Arbeidsplan for helgen

Referat

Vi konkluderete med at vi ikke kunne komme noe videre med utrullingen av keystone før vi fikk hjelp av oppdragsgiver.
Vi diskuterte om vi skulle gå videre i prosjektet, altså starte med å finne/lage tester. Eller om vi skulle jobbe med oppgaver i andre fag, for så å jobbe inn den tapte tiden så fort vi hadde fått hjelp av oppdragsgiver. Vi bestemte oss for å jobbe med andre fag. Men vi kunne lage strukturen på den endelige rapporten slik som veileder Hadde anbefalt på siste møte med han.

Møte Alexander og Mats 15.02.19

Deltagere: Alexander, Mats
Tid: 17:42 til 19:00
Sted: Discord

Agenda

Problemer med å deploye keystone og bootstrap klassene.
Hva vi skal jobbe med videre.

Referat

Etter at vi fikk Hiera dataene fra oppdragsgiver har vi i noen dager jobbet med å installere keystone og bootstrap klassene. Vi har nå konkludert med at dette ikke lar seg gjøre ut i fra de filene vi har fått. Det er for dårlig spesifisert hvordan og hvilke parametre vi trenger for å få installert noe som helst.

Vi diskuterte om hva vi nå skulle gjøre videre, siden vi ligger bak ifht Gantt-skjemaet.

Vi ble enige om at vi sender en mail til oppdragsgiver hvor vi forklarer om problemet vi har, og at vi trenger hiera data som er klare for en installasjon. Vi har ikke lenger tid til å gjette på hvilke parametre vi trenger og hva som skal stå i dem for at installasjonen skal fungere. Dersom vi ikke får disse filene, kan vi ikke lage tester for NTNU sin Openstack installasjon.

Vi ble enige om hva vi skulle fokusere på videre, som var:

- Finne ut hvilke CI/CD verktøy vi skal bruke og begrunne valget.

Som en del av denne prosessen, tester vi ut 2 forskjellige verktøy og gir våres meninger om hva som passer best.

- Lage en CI/CD pipeline for linting og validering av Puppet kode.
  Det er ønskelig at hvis denne CI/CD pipelinen er en lokal installasjon, at den håndteres av Puppet kode. Vi skal også undersøke hvordan pipelinen skal gjøre automatisk, og om det er hensiktsmessig at dette skjer ved hver commit eller merge.

Vi kommer til å jobbe med disse punktene, og skriving av rapporten fremover til vi får svar fra oppdragsgiver ifht. keystone hiera dataene.

Møte Alexander og Mats 25.2.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

hva har vi gjort i helgen
temaer for neste møte med veileder

Referat

Mats snakket om det han hadde gjort i helgen. Han hadde problemer med å kjøre Gitlab runners i Openstack. Han hadde også gjort ferdig all den automatiseringen av KeyStone deployment som vi jobbet med etter møtet med Egil og Lars Erik i forrige uke. Alexander hadde klart å kjøre en enkel Gitlab pipeline. Deretter snakket vi om temaer til neste møte med veileder. Aktuelle punkter var: skal vi utelukke Gitlab eller kan vi få en Gitlab lisens.

Til slutt ble vi enige om å jobbe videre men Gitlab pipeline og å lage en Docker Compose fil for utrulling av KeyStone.

Møte Alexander og Mats 1.3.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

møtet mellom mats og Eirik på onstag
hva skal vi gjøre fremover

Referat

mats nevnte at Erik mente vi kun burde fokusere på Gitlab. Ettersom han mente Jenkins var utdatert og at ingen brukte det lenger. Deretter så vi på og diskuterte NTNU sin andre, kanskje ofisielle, instalasjon. vi bestemte oss for å fokusere på Gitlab. At et av målene våre var å lage en større pipeline file.

Vi er usikre på hvordan vi skal få delt opp oppgaver mellom oss slik at vi ikke jobber med det samme, vi kom ikke frem til noen løsning på dette møtet. vi bestemte oss for å jobbe sammen i dag for at vi begge ble godt kjent med gitlab.

Møte Alexander og Mats 08.03.19

Deltagere: Alexander, Mats
Tid: 19:00 til 20:00
Sted: Discord

Agenda

Hva vi skal jobbe med videre.

Referat

Snakket om hva vi skulle gjøre videre etter å ha fått til validering og linting i pipelinen. Mats mener vi burde starte med beaker for akseptanse testing av keystone modulen. Siden beaker er et veldig omfattende verktøy, er det best å prøve å sette opp mindre Puppet moduler med dette først. Målet videre er få beaker til å starte OpenStack servere, og smoke teste en enkel modul.

Når vi har fått beaker til å fungere kan vi starte med å lage tester for keystone. Er mulig at GitLab repo-et må gjøres om til å være mer lik Puppets ”control repo” for at dette skal fungere.

Møte Alexander og Mats 29.3.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

demoen som jeg holdt på onsdag
teste Keystone

Referat

Snakket om demoen som alexander holdt på onsdag. Deretter satt vi opp en rekke oppgaver som vi skulle gjøre fremover på Trello. Vi så på et lite python script som mats hadde laget for å teste Keystone

Møte Alexander og Mats 5.4.19

Deltagere: Alexander, Mats
Tid: 09:00 til 10:00
Sted: Discord

Agenda

hva vi har gjort siden siste møte
veien videre
hva vi skal gjøre fremover

Referat

Mats sa at han var ferdig med Python API testene til Keystone, men at han hadde igjen å integrere det in i Beaker. Alexander sa han var ferdig med autoMergeRequest. scriptet.

Vi ble så enige om at vi skal prøve å bli ferdige med alt det tekniske til påske, litt over en uke unna. Vi så over møterapportene vi hadde på Overleaf, og la merke til at vi hadde glemt å skrive rapporter for et par uker i mars, Mats sa han hadde noen notater fra et møte og sa han skulle skrive en rapport basert på det.

Mats vil bruke tiden fremover på å å integrere Python API testene til Keystone in i Beaker. Alexander skal jobbe med å få alt med vegrant in i Docker containere. Slik at vi ikke lenger trenger å håndtere directoryes som er umulige å slette uten sudo rettigheter.

D Scripts og filer brukt i CI/CD pipelinen

D.1 Pipeline fil

```
stages:
    - test_changed_code
    - testing
    - merge

# Validating and linting all puppet code that has changed since last commit
statictest:
    stage: test_changed_code
    only:
        - Development
    before_script:
        - echo "Building validate and linting docker image"
        - docker build --cache-from checkcode:latest --tag checkcode:latest
          ↪  dockeFileDirectory/validate_lint/.
        - if [[ $(git diff HEAD^ HEAD --name-only | grep '.pp') = "" ]]; then
          ↪  echo "No files to check" && exit 0; fi
        - FILES=$(git diff HEAD^ HEAD --name-only | grep '.pp')

    script:
        - echo "Validating and linting puppet code"
        - docker run -v $(pwd):/root/puppet/ -t checkcode:latest parser "$FILES"
        - docker run -v $(pwd):/root/puppet/ -t checkcode:latest linter "$FILES"


# Acceptance and integration testing with Beaker and API tests
acceptance_and_integration:
    stage: testing
    only:
        - Development
    before_script:
        - echo "Building acceptance and integration docker image"
        - docker build --cache-from beaker:latest --tag beaker:latest
          ↪  dockeFileDirectory/beaker/.

    script:
        - echo "Running acceptance and integration tests on keystone"
        - docker run -v $(pwd):/root/puppet/ -t beaker:latest


# If all other stages where succssfull, make new brach based on this one and push
 ↪  to origin
to_manual_testing:
    stage: merge
    only:
        - Development
    dependencies:
        - statictest
        - acceptance_and_integration
    script:
    # run automatic_merge_request script
```

```
47      - chmod +x utils/autoMergeRequest.sh
48      # set all required variabels and call script with or without a target branch
         ↪  as a parameter
49      - HOST=${CI_PROJECT_URL} CI_PROJECT_ID=${CI_PROJECT_ID}
         ↪  CI_COMMIT_REF_NAME=${CI_COMMIT_REF_NAME}
         ↪  GITLAB_USER_ID=${GITLAB_USER_ID} PRIVATE_TOKEN=${PRIVATE_TOKEN}
         ↪  ./utils/autoMergeRequest.sh  Production
```

D.2 Scripts og filer brukt i stage 1: Statisk kodeanalyse

Dockerfile for stage 1

```
FROM ubuntu:18.04

# Install utility tool.
RUN apt-get update && \
    apt-get -y install wget

# Adds new repo to apt-get.
RUN tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXXX) || exit 1 && \
    wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb && \
    dpkg -i "$tempdeb"

# Installs puppet and puppet lint.
RUN apt-get update && \
    apt-get -y install puppetserver && \
    apt-get -y install puppet-lint

# Set workdir and copy in script.
WORKDIR /root
RUN mkdir /root/puppet
COPY docker-entrypoint.sh /root/docker-entrypoint.sh

RUN chmod +x docker-entrypoint.sh

ENTRYPOINT ["./docker-entrypoint.sh"]
```

Entrypoint script for container i stage 1

```
#!/bin/bash

# Info:
#  Bash script that runs as ENTRYPOINT in the docker container "checkcode".
#  Used to run static code analysis on puppet code.
#
# Parameters:
#  Param 1: Which command to run, accepts "validate" or "parser".
#  Param 2: Array with .pp files that the command run against.
#
# Example:
#  Start container that runs 'puppet parser' on '$FILES':
#  $ docker run -v $(pwd):/root/puppet/ -t checkcode:latest parser "$FILES"
#
#  "$FILES" is created by the command:
#  $ FILES=$(git diff HEAD^ HEAD --name-only | grep '.pp')
#  This list all files with the 'pp' extension that has changed since last commit
 ↪  on the repo.


cd /root/puppet/

TYPE=$1; shift
ARRAY=($@)

# The command to run is chosen from the first parameter.
if [ "$TYPE" == "parser" ]; then
  echo "Running puppet parser validate"
  COMMAND="/opt/puppetlabs/bin/puppet parser validate"

elif [ "$TYPE" == "linter"  ]; then
  echo "Running puppet lint"
  COMMAND="/usr/bin/puppet-lint --no-autoloader_layout-check --fail-on-warnings
   ↪  --no-80chars-check --no-140chars-check"

else
  echo "Error: No parameter given in second position, must provide parser og
   ↪  linter"
  exit 1
fi

# Loops through all .pp files that have changed since last commit
#  and runs the specified command on them.
for file in "${ARRAY[@]}"
do
  echo "$file"
  $COMMAND "$file"
done
```

D.3 Scripts og filer brukt i stage 2: Testing

Dockerfile for stage 2

```
FROM ubuntu:18.04

# Installs python3.7 and dependencies for test_keystone.py.
RUN apt-get update && \
    apt-get install python3.7 -y && \
    apt-get install python3-pip -y && \
    pip3 install requests

# Install utility tools.
RUN apt-get install pwgen -y && \
    apt-get install sshpass -y && \
    apt-get install wget -y

# Install vagrant and vagrant plugin.
RUN wget -c https://releases.hashicorp.com/vagrant/2.0.3/vagrant_2.0.3_x86_64.deb
 ↪  && \
    dpkg -i vagrant_2.0.3_x86_64.deb && \
    rm vagrant_2.0.3_x86_64.deb && \
    vagrant plugin install vagrant-openstack-provider

# Install ruby bundler.
RUN apt install ruby-dev libffi-dev build-essential -y && \
    apt-get install ruby-bundler -y

# Set workdir and copy in script.
WORKDIR /root
RUN mkdir /root/puppet
COPY runBeaker.sh /root/runBeaker.sh

RUN chmod +x runBeaker.sh

# make seperate vagrant dir not used by volume
RUN mkdir /vagrant

ENTRYPOINT ["./runBeaker.sh"]
```

Entrypoint script for container i stage 2

```
#!/bin/bash

# Script to provision machines with vagrant and run tests.


# ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Functions

# prints red text
function errorPrint() {
    echo -e "\e[1;31m\t $1 \e[0m" >&2
}


# ''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Variables:

# Current path:
CURR_PATH="$(pwd)/puppet"
export CURR_PATH=$CURR_PATH

# copy vagrant config to directory not shared by host machine
cp  "$CURR_PATH/spec/acceptance/config/Vagrantfile" /vagrant/Vagrantfile

# Directory of the Vagrantfile.
VAGRANT_DIR="/vagrant"

# Beaker files.
BEAKER_HOSTFILE="/vagrant/default.yaml"
BEAKER_PRE_SUITE="$CURR_PATH/spec/acceptance/setup/pre_suite_keystone.rb"
BEAKER_TEST="$CURR_PATH/spec/acceptance/tests/test_keystone.rb"

# Generates random passwords between PASS_LENGTH_MIN - PASS_LENGTH_MAX.
PASS_LENGTH_MIN=20
PASS_LENGTH_MAX=30

# Password to log into the vagrant machines over SSH as root.
VAGRANT_PASS=$(pwgen -c -n -N 1 $(shuf -i $PASS_LENGTH_MIN-$PASS_LENGTH_MAX -n 1))
export VAGRANT_PASS=$VAGRANT_PASS

# Password to log into mysql server as root.
MYSQL_ROOT_PASS=$(pwgen -c -n -N 1 $(shuf -i $PASS_LENGTH_MIN-$PASS_LENGTH_MAX -n 1))
export MYSQL_ROOT_PASS=$MYSQL_ROOT_PASS


#''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
# Script:

# Installs gems from Gemfile.
cd $CURR_PATH/spec/
bundle install

# Starts vagrant machines.
```

```
54 cd $VAGRANT_DIR
55 vagrant up
56 if [ $? -ne 0 ]; then
57   errorPrint "Error in running vagrant up"
58   exit 1
59 fi
60
61 # Get IP adresses from the vagrant machines.
62 IPS=$(vagrant ssh-config | grep HostName)
63 VAGRANT_IP_MASTER=$(echo $IPS | awk '{ print $2 }')
64 VAGRANT_IP_AGENT=$(echo $IPS | awk '{ print $4 }')
65 VAGRANT_IP_MYSQL=$(echo $IPS | awk '{ print $6 }')
66 MYSQL_LOCAL_IP=$(vagrant ssh mysql -c "ip address show ens3 | grep -w 'inet' |
    ↪  sed -e 's/^.*inet //' -e 's/\/.*$//'")
67 KEYSTONE_LOCAL_IP=$(vagrant ssh agent -c "ip address show ens3 | grep -w 'inet' |
    ↪  sed -e 's/^.*inet //' -e 's/\/.*$//'")
68 export VAGRANT_IP_AGENT=$VAGRANT_IP_AGENT
69
70 # Gets the mysql password for the database "keystone" from hiera.
71 MYSQL_KEYSTONE_PASS=$(grep -w "ntnuopenstack::keystone::mysql::password"
    ↪  $CURR_PATH/data/ntnuopenstack.yaml | sed 's/"//g' | awk '{ print $2 }')
72 export MYSQL_KEYSTONE_PASS=$MYSQL_KEYSTONE_PASS
73
74 # Add IP adresses to hiera data.
75 sed -i
    ↪  "s/^.*ntnuopenstack::keystone::mysql::ip:.*$/ntnuopenstack::keystone::mysql::ip:
    ↪  '$MYSQL_LOCAL_IP'/" $CURR_PATH/data/common.yaml
76 sed -i
    ↪  "s/^.*ntnuopenstack::keystone::auth::url:.*'http:\/\/.*:35357'/ntnuopenstack::keystone::auth::url:
    ↪  'http:\/\/$KEYSTONE_LOCAL_IP:35357'/" $CURR_PATH/data/ntnuopenstack.yaml
77 sed -i
    ↪  "s/^.*ntnuopenstack::keystone::auth::uri:.*'/ntnuopenstack::keystone::auth::uri:
    ↪  'http:\/\/$KEYSTONE_LOCAL_IP:5000'/" $CURR_PATH/data/ntnuopenstack.yaml
78 sed -i
    ↪  "s/^.*profile::openstack::endpoint::admin:.*'http:\/\/.*'.*$/profile::openstack::endpoint::admin:
    ↪  'http:\/\/$KEYSTONE_LOCAL_IP'/" $CURR_PATH/data/openstack.yaml
79 sed -i
    ↪  "s/^.*profile::openstack::endpoint::internal:.*'http:\/\/.*'.*$/profile::openstack::endpoint::interr
    ↪  'http:\/\/$KEYSTONE_LOCAL_IP'/" $CURR_PATH/data/openstack.yaml
80 sed -i
    ↪  "s/^.*profile::openstack::endpoint::public:.*'http:\/\/.*'.*$/profile::openstack::endpoint::public:
    ↪  'http:\/\/$VAGRANT_IP_AGENT'/" $CURR_PATH/data/openstack.yaml
81
82 # Removes carriage return (CR) in files.
83 sed -i 's/\r//' $CURR_PATH/data/common.yaml
84 sed -i 's/\r//' $CURR_PATH/data/ntnuopenstack.yaml
85 sed -i 's/\r//' $CURR_PATH/data/openstack.yaml
86
87 # Creates the beaker hosts file.
88 cat <<EOF > $BEAKER_HOSTFILE
89 ---
90 HOSTS:
91   Master:
92     platform: ubuntu-16.04-amd64
93     hypervisor: none
94     ip: $VAGRANT_IP_MASTER
95     roles:
96       - master
```

```
97      ssh:
98        user: 'root'
99        password: $VAGRANT_PASS
100       auth_methods:
101         - password
102
103   Agent:
104     platform: ubuntu-16.04-amd64
105     hypervisor: none
106     ip: $VAGRANT_IP_AGENT
107     roles:
108       - agent
109       - default
110     ssh:
111       user: 'root'
112       password: $VAGRANT_PASS
113       auth_methods:
114         - password
115
116   Mysql:
117     platform: ubuntu-16.04-amd64
118     hypervisor: none
119     ip: $VAGRANT_IP_MYSQL
120     roles:
121       - database
122     ssh:
123       user: 'root'
124       password: $VAGRANT_PASS
125       auth_methods:
126         - password
127
128 CONFIG:
129   type: foss
130 EOF
131
132 # Run beaker.
133 beaker --hosts $BEAKER_HOSTFILE --pre-suite $BEAKER_PRE_SUITE --tests
    ↪  $BEAKER_TEST
134 if [ $? -ne 0 ]; then
135   vagrant destroy --force
136   errorPrint "Error in running beaker"
137   exit 1
138 fi
139
140 # Create ENV variables for keystone_test.py from hiera data.
141 KEYSTONE_PASSWORD=$(grep -w "ntnuopenstack::keystone::admin_password:"
    ↪  $CURR_PATH/data/ntnuopenstack.yaml | sed 's/"//g' | awk '{ print $2 }')
142 KEYSTONE_TOKEN=$(grep -w "ntnuopenstack::keystone::admin_token:"
    ↪  $CURR_PATH/data/ntnuopenstack.yaml | sed 's/"//g' | awk '{ print $2 }')
143 export KEYSTONE_PASSWORD=$KEYSTONE_PASSWORD
144 export KEYSTONE_TOKEN=$KEYSTONE_TOKEN
145
146 # Run Keystone API tests.
147 python3 $CURR_PATH/spec/scripts/test_keystone.py
148 if [ $? -ne 0 ]; then
149   vagrant destroy --force
150   errorPrint "Error in running test_keystone.py"
151   exit 1
```

```
152 fi
153
154 # Clean up: destroy vagrant machines.
155 vagrant destroy --force
```

Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :


# Uses OpenStack specific configuration for vagrant plugin 'vagrant-openstack-provider'.
require 'vagrant-openstack-provider'


# Script to run on the vagrant machine.
$script = <<-SCRIPT

# Change ssh config to allow root ssh access with password.
sudo sed -i -e 's/PermitRootLogin prohibit-password/PermitRootLogin yes/g' /etc/ssh/sshd_config
sudo sed -i -e 's/PasswordAuthentication no/PasswordAuthentication yes/g' /etc/ssh/sshd_config
sudo sed -i -e 's/StrictModes yes/StrictModes no/g' /etc/ssh/sshd_config

# Add password to root user.
sudo usermod --password $(echo $password | openssl passwd -1 -stdin) root

# Restart ssh service.
sudo service ssh restart
SCRIPT


# Vagrant configuration.
Vagrant.configure('2') do |config|
  config.vm.synced_folder ".", "/vagrant", disabled: true
  config.ssh.username                   = 'ubuntu'
  config.ssh.forward_agent              = true

  # Run the script on the provisioned vagrant machines. Uses environmental variable for password.
  config.vm.provision 'shell', inline: $script, env: {"password" => ENV['VAGRANT_PASS']}

  # Puppetmaster configuration.
  config.vm.define :master do |master|
    master.vm.provider :openstack do |os, ov|
      os.server_name                    = 'beaker-puppetmaster'
      os.identity_api_version           = '3'
      os.openstack_auth_url             = 'https://api.skyhigh.iik.ntnu.no:5000/v3/auth/tokens'
      os.project_name                   = 'BITSEC3_V19_SkyhighCICD'
      os.user_domain_name               = 'Default'
      os.project_domain_name            = 'NTNU'
      os.username                       = 'BITSEC3_V19_SkyhighCICD_service'
      os.password                       = 'gGNUojWEF4dp'
      os.region                         = 'SkyHiGh'
      os.floating_ip_pool               = 'ntnu-internal'
      os.floating_ip_pool_always_allocate = false
      os.flavor                         = 'm1.small'
      os.image                          = 'Ubuntu Server 16.04 LTS (Xenial Xerus) amd64'
      os.networks                       = ['test']
```

```
51        os.security_groups                  = ['default','linux']
52
53        ov.nfs.functional                   = false
54      end
55    end
56
57    # Puppet agent configuration (keystone).
58    config.vm.define :agent do |agent|
59      agent.vm.provider :openstack do |os, ov|
60        os.server_name                      = 'beaker-keystone'
61        os.identity_api_version             = '3'
62        os.openstack_auth_url               =
          ↪  'https://api.skyhigh.iik.ntnu.no:5000/v3/auth/tokens'
63        os.project_name                     = 'BITSEC3_V19_SkyhighCICD'
64        os.user_domain_name                 = 'Default'
65        os.project_domain_name              = 'NTNU'
66        os.username                         = 'BITSEC3_V19_SkyhighCICD_service'
67        os.password                         = 'gGNUojWEF4dp'
68        os.region                           = 'SkyHiGh'
69        os.floating_ip_pool                 = 'ntnu-internal'
70        os.floating_ip_pool_always_allocate = false
71        os.flavor                           = 'm1.small'
72        os.image                            = 'Ubuntu Server 16.04 LTS (Xenial
          ↪  Xerus) amd64'
73        os.networks                         = ['test']
74        os.security_groups                  = ['default','linux']
75
76        ov.nfs.functional                   = false
77      end
78    end
79
80    # Mysql configuration.
81    config.vm.define :mysql do |mysql|
82      mysql.vm.provider :openstack do |os, ov|
83        os.server_name                      = 'beaker-mysql'
84        os.identity_api_version             = '3'
85        os.openstack_auth_url               =
          ↪  'https://api.skyhigh.iik.ntnu.no:5000/v3/auth/tokens'
86        os.project_name                     = 'BITSEC3_V19_SkyhighCICD'
87        os.user_domain_name                 = 'Default'
88        os.project_domain_name              = 'NTNU'
89        os.username                         = 'BITSEC3_V19_SkyhighCICD_service'
90        os.password                         = 'gGNUojWEF4dp'
91        os.region                           = 'SkyHiGh'
92        os.floating_ip_pool                 = 'ntnu-internal'
93        os.floating_ip_pool_always_allocate = false
94        os.flavor                           = 'm1.tiny'
95        os.image                            = 'Ubuntu Server 16.04 LTS (Xenial
          ↪  Xerus) amd64'
96        os.networks                         = ['test']
97        os.security_groups                  = ['default','linux']
98
99        ov.nfs.functional                   = false
100     end
101   end
102 end
```

Beaker installasjonsscript

```
# Configures vagrant machines to test keystone.

# Install packages for puppet and mysql.
# Configures puppet and mysql.
test_name "Install and configure keystone stack" do

  # Gets environment variables.
  path = ENV['CURR_PATH']
  mysqlRootPass = ENV['MYSQL_ROOT_PASS']
  mysqlKeystonePass = ENV['MYSQL_KEYSTONE_PASS']

  # Installs puppetserver and puppet agent on machines with the 'master' and
   ↪  'agent' vagrant roles.
  step "Install puppet" do
    hosts.each do |host|
      if host['roles'].include?('master') || host['roles'].include?('agent')
        on(host, "wget https://apt.puppet.com/puppet5-release-xenial.deb")
        on(host, "dpkg -i puppet5-release-xenial.deb")
        on(host, "rm puppet5-release-xenial.deb")
        on(host, "apt-get update")
      end

      # Installs puppetserver on machines with the 'master' vagrant role.
      if host['roles'].include?('master')
        install_package(host, 'puppetserver') unless check_for_package(host,
         ↪  'puppetserver')
      end

      # Installs puppet-agent on machines with the 'agent' vagrant role.
      if host['roles'].include?('agent')
        install_package(host, 'puppet-agent') unless check_for_package(host,
         ↪  'puppet-agent')
      end
    end
  end

  # Configures puppet for keystone.
  step "Configure puppet" do
    mastername = "beaker-puppetmaster"

    hosts.each do |host|
      # Confiugres the puppetmaster.
      if host['roles'].include?('master')
        puppetPath = "/etc/puppetlabs/code/environments/master"

        on(host, "echo '127.0.0.1 #{mastername}.openstacklocal #{mastername}' >>
         ↪  /etc/hosts")
        on(host, "/opt/puppetlabs/bin/puppet config set server
         ↪  #{mastername}.openstacklocal --section main")
        on(host, "/opt/puppetlabs/bin/puppet config set autosign true --section
         ↪  master")
        on(host, "mkdir #{puppetPath}")

        # Copy over directories and files to the puppetmaster.
        files = [ 'hiera.yaml', 'data/', '/manifests' ]
        files.each do |file|
          scp_to host, "#{path}/#{file}", "#{puppetPath}/#{file}"
```

```
52          end
53
54          # Install r10k, copy over Puppetfile and install dependencies.
55          on(host, "/opt/puppetlabs/puppet/bin/gem install r10k")
56          scp_to host, "#{path}/Puppetfile", "#{puppetPath}/Puppetfile"
57          on(host, "cd #{puppetPath} && /opt/puppetlabs/puppet/bin/r10k puppetfile
             ↪  install")
58
59          # Copy over the puppet code.
60          folders = [ 'ntnuopenstack/', 'profile/', 'role/' ]
61          folders.each do |folder|
62            scp_to host, "#{path}/#{folder}", "#{puppetPath}/modules/#{folder}"
63          end
64          on(host, "service puppetserver start")
65        end
66
67        # Configures puppet agent (keystone).
68        if host['roles'].include?('agent')
69          on(host, "echo '#{master.get_ip} #{mastername}.openstacklocal
             ↪  #{mastername}' >> /etc/hosts")
70          on(host, "/opt/puppetlabs/bin/puppet config set server
             ↪  #{mastername}.openstacklocal --section main")
71          on(host, "/opt/puppetlabs/bin/puppet config set environment master
             ↪  --section main")
72        end
73      end
74    end
75
76    # Installs mysql on machines with the 'database' vagrant role.
77    step "Install mysql" do
78      hosts.each do |host|
79        if host['roles'].include?('database')
80          on(host, "apt-get update")
81          on(host, "debconf-set-selections <<< \"mysql-server
             ↪  mysql-server/root_password password #{mysqlRootPass}\"")
82          on(host, "debconf-set-selections <<< \"mysql-server
             ↪  mysql-server/root_password_again password #{mysqlRootPass}\"")
83          on(host, "apt-get install -qq mysql-server")
84        end
85      end
86    end
87
88    # Configures mysql to be used by keystone.
89    step "Configure mysql" do
90      hosts.each do |host|
91        if host['roles'].include?('database')
92          on(host, "sed -i 's/bind-address.*= 127.0.0.1/bind-address          =
             ↪  0.0.0.0/' /etc/mysql/mysql.conf.d/mysqld.cnf")
93          on(host, "mysql --user root --password=#{mysqlRootPass} -e \"CREATE
             ↪  DATABASE keystone;GRANT ALL PRIVILEGES ON keystone.* TO
             ↪  'keystone'@'%' IDENTIFIED BY '#{mysqlKeystonePass}';\"")
94          on(host, "mysql --user root --password=#{mysqlRootPass} -e \"flush
             ↪  privileges;\"")
95          on(host, "systemctl enable mysql")
96          on(host, "service mysql restart")
97        end
98      end
99    end
```

```
100 end
```

Beaker akseptansetest script

```
# Tests the keystone installation.

require 'beaker'
require 'beaker-puppet'

test_name "Check installation of keystone stack" do

  # Gets environment variable.
  mysqlRootPass = ENV['MYSQL_ROOT_PASS']

  step "Checks if packages got installed and if services are running" do
    hosts.each do |host|
      if host['roles'].include?('master')
        assert check_for_package(host, 'puppetserver')
        on(host, "systemctl is-active puppetserver") do |result|
          assert_equal(0, result.exit_code)
        end
      end

      if host['roles'].include?('agent')
        assert check_for_package(host, 'puppet-agent')
      end

      if host['roles'].include?('database')
        assert check_for_package(host, 'mysql-server')
        on(host, "systemctl is-active mysql") do |result|
          assert_equal(0, result.exit_code)
        end
      end
    end
  end

  step "Check if mysql got keystone DB" do
    on database, "mysql --user root --password=#{mysqlRootPass} -e \"show
     ↪  databases;\" | grep keystone" do |result|
      assert_match result.stdout, "\/keystone\n/"
    end
  end

  step "Check if puppet can apply keystone configuration" do
    hosts.each do |host|
      if host['roles'].include?('agent')
        shell '/opt/puppetlabs/bin/puppet agent -t -v --detailed-exitcodes',
         ↪  :acceptable_exit_codes => [0,2]
      end
    end
  end
end
```

Python integrasjonstest script

```
"""
Program to test the keystone API.
API calls described at: https://developer.openstack.org/api-ref/identity/v3/

Uses Python 3.7

Dependencies can be installed with:
$ pip3 install requests

To create a documentation in HTML:
$ python3 -m pydoc -w ./test_keystone.py

This only works if the script returns 0.
"""

import os
import json
import requests


# ********************************************************************************************
# CLASS:

class TermColor:
    """
    Contains colors to be used when printing text.
    """

    PURPLE = '\033[95m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    NORMAL = '\033[0m'
    UNDERLINE = '\033[4m'


# ********************************************************************************************
# GLOBAL VARIABLES:

FAILED_TESTS = False                            # Changes to True if one or more tests fails.
NUM_PASSED_TESTS = 0                            # Number of passed tests.
NUM_FAILED_TESTS = 0                            # Number of failed tests.


# ********************************************************************************************
# FUNCTIONS:

def print_color(color_obj, message):
    """
    Prints text with colors.

    :parameter:
```

```
54        - color_obj: (TermColor obj) The color of the text.
55        - message:   (string) The message to be printed.
56      """
57
58      print(color_obj + message + TermColor.NORMAL)
59
60
61  def test_failed(status):
62      """
63      Updates the test scores.
64
65      :parameter:
66        - status: (Boolean) True if the test failed, False if it passed.
67      """
68
69      global FAILED_TESTS, NUM_PASSED_TESTS, NUM_FAILED_TESTS
70
71      # Updates global variables based on param.
72      if status:
73          NUM_FAILED_TESTS += 1
74          FAILED_TESTS = True
75      else:
76          NUM_PASSED_TESTS += 1
77
78
79  def print_message(method, url, reason, actual_status_code, expected_status_code):
80      """
81      Prints if test failed or not,
82      calls 'test_failed' function to update global variables.
83
84      :parameter:
85        - method:               (string) The HTTP method used.
86        - url:                  (string) The url that was used in the request.
87        - reason:               (string) HTTP status message.
88        - actual_status_code:   (int)    The HTTP status code that the website
      ↪  displayed.
89        - expected_status_code: (int)    The HTTP status code that was expected to
      ↪  show.
90      """
91
92      print("  Checking http status code for " + method + ": " + url)
93      if actual_status_code != expected_status_code:
94          test_failed(True)
95          message = "   - Test failed: Wrong http status response code: got " + \
96              str(actual_status_code) + " want " + str(expected_status_code) + \
97              "\n   - HTTP status message: " + reason
98          print_color(TermColor.RED, message)
99
100     else:
101         test_failed(False)
102         print_color(TermColor.GREEN, "   - Test passed")
103
104
105 def check_values(uri_path, dictionary):
106     """
107     Checks if values match in the dictionary.
108     (Compares keys and values).
109
```

```
110     :parameter:
111       - uri_path:   (string)     URI path used in the test.
112       - dictionary: (dictionary) Contains key-value pairs that are compared.
113     """
114
115     print("\n  Check if", uri_path, "body contains the correct values:")
116
117     # Loops through dictionary and compares the content of key and value.
118     for i in range(len(dictionary)):
119         key = list(dictionary.keys())[i]
120         value = list(dictionary.values())[i]
121
122         if value != key:
123             test_failed(True)
124             message = "   - Test failed: Wrong value: got " + \
125                 key + " want " + value
126             print_color(TermColor.RED, message)
127
128         else:
129             test_failed(False)
130             print_color(TermColor.GREEN, "   - Test passed")
131
132
133 def get_request(url, header):
134     """
135     Does a GET request and checks if status code returns as expected.
136
137     :parameter:
138       - url:    (string) The URL that the request runs against.
139       - header: (Response obj) HTTP header to be used in the GET request.
140
141     :returns:
142       - (Response obj) HTTP body.
143     """
144
145     response = requests.get(url, headers=header)
146     print_message("GET", url, response.reason, response.status_code, 200)
147
148     return json.loads(json.dumps(response.json()))
149
150
151 def post_request(url, header, payload):
152     """
153     Does a POST request and checks if status code returns as expected.
154
155     :parameter:
156       - url:     (string) The URL that the request runs against.
157       - header:  (request header) HTTP header to be used in the POST request.
158       - payload: (request body) HTTP body to be used in the POST request.
159
160     :returns:
161       - (Response obj) HTTP header.
162       - (Response obj) HTTP body.
163     """
164
165     response = requests.post(url, headers=header, data=json.dumps(payload))
166     print_message("POST", url, response.reason, response.status_code, 201)
167
```

```
168     return response.headers, json.loads(json.dumps(response.json()))
169
170
171 def patch_request(url, header, payload):
172     """
173     Does a PATCH request and checks if status code returns as expected.
174
175     :parameter:
176       - url:     (string) The URL that the request runs against.
177       - header:  (request header) HTTP header to be used in the PATCH request.
178       - payload: (request body) HTTP body to be used in the PATCH request.
179     """
180
181     response = requests.patch(url, headers=header, data=json.dumps(payload))
182     print_message("PATCH", url, response.reason, response.status_code, 200)
183
184
185 def put_request(url, header):
186     """
187     Does a PUT request and checks if status code returns as expected.
188
189     :parameter:
190       - url:    (string) The URL that the request runs against.
191       - header: (request header) HTTP header to be used in the PUT request.
192     """
193
194     response = requests.put(url, headers=header)
195     print_message("PUT", url, response.reason, response.status_code, 204)
196
197
198 def delete_requests(url, header):
199     """
200     Does a DELETE request and checks if status code returns as expected.
201
202     :parameter:
203       - url:    (string) The URL that the request runs against.
204       - header: (request header) HTTP header to be used in the DELETE request.
205     """
206
207     response = requests.delete(url, headers=header)
208     print_message("DELETE", url, response.reason, response.status_code, 204)
209
210
211 # ****************************************
212 # Basic tests:
213
214 def test_token(url, password, token):
215     """
216     Tests the "/v3/auth/tokens" URI in the keystone API.
217
218     :parameter:
219       - url:      (string) The URL to test.
220       - password: (string) Keystone password.
221       - token:    (string) Keystone token.
222     """
223
224     print_color(TermColor.BLUE,
```

```
225
                ↪  "\n\n------------------------------------------------------------------------")
226     print_color(TermColor.BLUE, "*** Test section: Authentication and token
         ↪  management ***")
227
228     # Set the authentication token in the header.
229     post_headers = {
230         'Content-Type': 'application/json',
231         'X-Auth-Token': token
232     }
233
234     # Payload to be sent in the POST requests that creates a new keystone token.
235     post_payload = {
236         "auth": {
237             "identity": {
238                 "methods": [
239                     "password"
240                 ],
241                 "password": {
242                     "user": {
243                         "name": "admin",
244                         "domain": {
245                             "name": "Default"
246                         },
247                         "password": password
248                     }
249                 }
250             }
251         }
252     }
253
254     # Sends the POST request to create a new token.
255     print_color(TermColor.YELLOW, "\nGenerate a new keystone token:")
256     resp_header, resp_body = post_request(url, post_headers, post_payload)
257
258     # Return if 'token' is not in the response body.
259     if 'token' not in resp_body:
260         return
261
262     # Gets the subject token from the POST header.
263     subject_token = resp_header.get('X-Subject-Token')
264
265     # Create the new header to be sent with a GET request.
266     get_headers = {
267         'X-Auth-Token': token,
268         'X-Subject-Token': subject_token
269     }
270
271     # Sends a GET request to check if the token got created.
272     print_color(TermColor.YELLOW, "\nCheck if the token got created:")
273     get_request(url, get_headers)
274
275     # Deletes the generated token.
276     print_color(TermColor.YELLOW, "\nDeletes the token:")
277     delete_requests(url, get_headers)
278
279
280 def test_domain(url, token):
```

```
281     """
282     Tests the "/v3/domains" URI in the keystone API.
283
284     :parameter:
285       - url:    (string) The URL to test.
286       - token:  (string) Keystone token.
287     """
288     print_color(TermColor.BLUE,
289
                 ↪  "\n\n----------------------------------------------------------------------")
290     print_color(TermColor.BLUE, "*** Test section: Domain ***")
291
292     domain_description = "Test domain"
293     domain_name = "TestDomain"
294
295     headers = {
296         'Content-Type': 'application/json',
297         'X-Auth-Token': token,
298     }
299
300     # Payload to be sent in the POST request that creates a new domain.
301     post_payload = {
302         "domain": {
303             "description": domain_description,
304             "enabled": True,
305             "name": domain_name
306         }
307     }
308
309     # Sends a POST request to create a new domain in keystone.
310     print_color(TermColor.YELLOW, "\nCreate a new domain in keystone:")
311     _, resp_body = post_request(url, headers, post_payload)
312
313     # Return if 'domain' is not in the response body.
314     if 'domain' not in resp_body:
315         return
316
317     # Gets the resource URL from the POST body.
318     get_url = resp_body['domain']['links']['self']
319
320     # Sends a GET request and checks if the values was set correctly.
321     print_color(TermColor.YELLOW, "\nCheck if the domain got created:")
322     data = get_request(get_url, headers)
323
324     # Create a dictionary of what fields that should contain the same data and
       ↪  checks if they do.
325     dictionary = {
326         data['domain']['description']: domain_description,
327         data['domain']['name']: domain_name
328     }
329     check_values("\"/v3/domains\"", dictionary)
330
331     # Updates the payload for the PATCH request.
332     patch_payload = {
333         "domain": {
334             "enabled": False
335         }
336     }
```

```
337
338     # Sends a PATCH request to mark the domain as inactive.
339     print_color(TermColor.YELLOW, "\nSet the domain as inactive:")
340     patch_request(get_url, headers, patch_payload)
341
342     # Sends a DELETE request to delete the created domain.
343     print_color(TermColor.YELLOW, "\nDelete the domain:")
344     delete_requests(get_url, headers)
345
346
347 def test_groups(url, token):
348     """
349     Tests the "/v3/groups" URI in the keystone API.
350
351     :parameter:
352       - url:    (string) The URL to test.
353       - token:  (string) Keystone token.
354     """
355
356     print_color(TermColor.BLUE,
357
                 ↪  "\n\n------------------------------------------------------------------------")
358     print_color(TermColor.BLUE, "*** Test section: Groups ***")
359
360     group_description = "Test group"
361     group_name = "TestGroup"
362
363     headers = {
364         'Content-Type': 'application/json',
365         'X-Auth-Token': token,
366     }
367
368     # Payload to be sent in the POST request that creates a new group.
369     post_payload = {
370         "group": {
371             "description": group_description,
372             "domain_id": "default",
373             "name": group_name
374         }
375     }
376
377     # Sends the POST request to create the group.
378     print_color(TermColor.YELLOW, "\nCreate a new group in keystone:")
379     _, resp_body = post_request(url, headers, post_payload)
380
381     # Return if 'group' is not in the response body.
382     if 'group' not in resp_body:
383         return
384
385     # Gets the resource URL from the POST body.
386     get_url = resp_body['group']['links']['self']
387
388     # Sends a GET request and checks if the values was set correctly.
389     print_color(TermColor.YELLOW, "\nCheck that the group got created:")
390     data = get_request(get_url, headers)
391
392     # Create a dictionary of what fields that should contain the same data and
       ↪  checks if they do.
```

```
393     dictionary = {
394         data['group']['description']: group_description,
395         data['group']['name']: group_name
396     }
397     check_values("\"/v3/groups\"", dictionary)
398 
399     # Sends a DELETE request to delete the created domain.
400     print_color(TermColor.YELLOW, "\nDelete the group:")
401     delete_requests(get_url, headers)
402 
403 
404 def test_projects(url, token):
405     """
406     Tests the "/v3/projects" URI in the keystone API.
407 
408     :parameter:
409       - url:    (string) The URL to test.
410       - token:  (string) Keystone token.
411     """
412 
413     print_color(TermColor.BLUE,
414 
                    ↪  "\n\n----------------------------------------------------------------------")
415     print_color(TermColor.BLUE, "*** Test section: Projects ***")
416 
417     project_description = "Test project"
418     project_name = "TestProject"
419 
420     headers = {
421         'Content-Type': 'application/json',
422         'X-Auth-Token': token,
423     }
424 
425     # Payload to be sent in the POST request that creates a new project.
426     post_payload = {
427         "project": {
428             "description": project_description,
429             "domain_id": "default",
430             "enabled": True,
431             "is_domain": False,
432             "name": project_name
433         }
434     }
435 
436     # Sends the POST request to create the project.
437     print_color(TermColor.YELLOW, "\nCreate a new project in keystone:")
438     _, resp_body = post_request(url, headers, post_payload)
439 
440     # Return if 'project' is not in the response body.
441     if 'project' not in resp_body:
442         return
443 
444     # Gets the resource URL from the POST body.
445     get_url = resp_body['project']['links']['self']
446 
447     # Sends a GET request and checks if the values was set correctly.
448     print_color(TermColor.YELLOW, "\nCheck that the project got created:")
449     data = get_request(get_url, headers)
```

```
450
451     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
452     dictionary = {
453         data['project']['description']: project_description,
454         data['project']['name']: project_name
455     }
456     check_values("\"/v3/projects\"", dictionary)
457
458     # Sends a DELETE request to delete the project.
459     print_color(TermColor.YELLOW, "\nDelete the project:")
460     delete_requests(get_url, headers)
461
462
463 def test_regions(url, token):
464     """
465     Tests the "/v3/regions" URI in the keystone API.
466
467     :parameter:
468       - url:    (string) The URL to test.
469       - token:  (string) Keystone token.
470     """
471
472     print_color(TermColor.BLUE,
473
                    ↪  "\n\n----------------------------------------------------------------------")
474     print_color(TermColor.BLUE, "*** Test section: Regions ***")
475
476     region_id = "newTestRegion"
477     region_description = "New test region"
478
479     headers = {
480         'Content-Type': 'application/json',
481         'X-Auth-Token': token,
482     }
483
484     # Payload to be sent in the POST request that creates a new region.
485     payload = {
486         "region": {
487             "description": region_description,
488             "id": region_id,
489             "parent_region_id": None
490         }
491     }
492
493     # Sends the POST request to create the region.
494     print_color(TermColor.YELLOW, "\nCreate a new region in keystone:")
495     _, resp_body = post_request(url, headers, payload)
496
497     # Return if 'region' is not in the response body.
498     if 'region' not in resp_body:
499         return
500
501     # Gets the resource URL from the POST body.
502     get_url = resp_body['region']['links']['self']
503
504     # Sends a GET request and checks if the values was set correctly.
505     print_color(TermColor.YELLOW, "\nCheck that the region got created:")
```

```
506     data = get_request(get_url, headers)
507
508     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
509     dictionary = {
510         data['region']['id']: region_id,
511         data['region']['description']: region_description
512     }
513     check_values("\"/v3/regions\"", dictionary)
514
515     # Sends a DELETE request to delete the project.
516     print_color(TermColor.YELLOW, "\nDelete the region:")
517     delete_requests(get_url, headers)
518
519
520 def test_roles(url, token):
521     """
522     Tests the "/v3/roles" URI in the keystone API.
523
524     :parameter:
525       - url:    (string) The URL to test.
526       - token:  (string) Keystone token.
527     """
528
529     print_color(TermColor.BLUE,
530
                    ↪  "\n\n----------------------------------------------------------------------")
531     print_color(TermColor.BLUE, "*** Test section: Roles ***")
532
533     role_description = "Test role"
534     role_name = "TestRole"
535
536     headers = {
537         'Content-Type': 'application/json',
538         'X-Auth-Token': token,
539     }
540
541     # Payload to be sent in the POST request that creates a new role.
542     payload = {
543         "role": {
544             "description": role_description,
545             "name": role_name
546         }
547     }
548
549     # Sends the POST request to create the region.
550     print_color(TermColor.YELLOW, "\nCreate a new role in keystone:")
551     _, resp_body = post_request(url, headers, payload)
552
553     # Return if 'role' is not in the response body.
554     if 'role' not in resp_body:
555         return
556
557     # Gets the resource URL from the POST body.
558     get_url = resp_body['role']['links']['self']
559
560     # Sends a GET request and checks if the values was set correctly.
561     print_color(TermColor.YELLOW, "\nCheck that the role got created:")
```

```
562     data = get_request(get_url, headers)
563
564     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
565     dictionary = {
566         data['role']['description']: role_description,
567         data['role']['name']: role_name
568     }
569     check_values("\"/v3/roles\"", dictionary)
570
571     # Sends a DELETE request to delete the project.
572     print_color(TermColor.YELLOW, "\nDelete the role:")
573     delete_requests(get_url, headers)
574
575
576 def test_services(url, token):
577     """
578     Tests the "/v3/services" URI in the keystone API.
579
580     :parameter:
581       - url:    (string) The URL to test.
582       - token:  (string) Keystone token.
583     """
584
585     print_color(TermColor.BLUE,
586
                 ↪  "\n\n------------------------------------------------------------------------")
587     print_color(TermColor.BLUE, "*** Test section: Service catalog ***")
588     print_color(TermColor.YELLOW, "\nCheck if the expected services are in the
        ↪  service catalog:")
589
590     headers = {
591         'Content-Type': 'application/json',
592         'X-Auth-Token': token,
593     }
594
595     # Sends the GET request and gets the OpenStack services.
596     data = get_request(url, headers)
597
598     # Return if 'services' is not in the response body.
599     if 'services' not in data:
600         return
601
602     # Create an array of expected OpenStack services that should be listed in the
        ↪  API.
603     expected_services = [
604         "glance",
605         "heat-cfn",
606         "neutron",
607         "cinderv3",
608         "placement",
609         "keystone",
610         "nova",
611         "cinder",
612         "heat",
613         "swift",
614         "cinderv2"
615     ]
```

```
616
617     # Test fail if list of actual services are not the same size as expected.
618     print("\n  Check if number of services are as expected:")
619     if len(data['services']) != len(expected_services):
620         test_failed(True)
621         message = "    - Test failed: Wrong size of service list: got " + \
622                   str(len(data['services'])) + " want " +
                      ↪  str(len(expected_services))
623         print_color(TermColor.RED, message)
624         return
625
626     test_failed(False)
627     print_color(TermColor.GREEN, "    - Test passed")
628
629     # Turn 'actual service' names into a list.
630     actual_services = []
631     for name in data['services']:
632         actual_services.append(name['name'])
633
634     # Sorts the lists before comparing them.
635     actual_services.sort()
636     expected_services.sort()
637
638     # Creates a dictionary to compare actual to expected.
639     dictionary = dict(zip(actual_services, expected_services))
640     check_values('\"/v3/services\"', dictionary)
641
642
643 def test_users(url, token):
644     """
645     Tests the "/v3/users" URI in the keystone API.
646
647     :parameter:
648       - url:    (string) The URL to test.
649       - token:  (string) Keystone token.
650     """
651
652     print_color(TermColor.BLUE,
653
                   ↪  "\n\n------------------------------------------------------------------------")
654     print_color(TermColor.BLUE, "*** Test section: Users ***")
655
656     headers = {
657         'Content-Type': 'application/json',
658         'X-Auth-Token': token,
659     }
660
661     user_name = "Test SkyHigh"
662     user_description = "Test SkyHigh user"
663     user_mail = "test@skyhigh.test"
664
665     # Payload to be sent in the POST request that creates a new user.
666     payload = {
667         "user": {
668             "domain_id": "default",
669             "enabled": True,
670             "name": user_name,
671             "password": "verySecretPassword",
```

```
672             "description": user_description,
673             "email": user_mail,
674             "options": {
675                 "ignore_password_expiry": True
676             }
677         }
678     }
679 
680     # Sends the POST request to create the user.
681     print_color(TermColor.YELLOW, "\nCreate a new user in keystone:")
682     _, resp_body = post_request(url, headers, payload)
683 
684     # Return if 'user' is not in the response body.
685     if 'user' not in resp_body:
686         return
687 
688     # Gets the resource URL from the POST body.
689     get_url = resp_body['user']['links']['self']
690 
691     # Sends a GET request and checks if the values was set correctly.
692     print_color(TermColor.YELLOW, "\nCheck that the user got created:")
693     data = get_request(get_url, headers)
694 
695     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
696     dictionary = {
697         data['user']['description']: user_description,
698         data['user']['name']: user_name,
699         data['user']['email']: user_mail
700     }
701     check_values("\"/v3/users\"", dictionary)
702 
703     # Sends a DELETE request to delete the project.
704     print_color(TermColor.YELLOW, "\nDelete the user:")
705     delete_requests(get_url, headers)
706 
707 
708 # ****************************************
709 # Advanced tests:
710 
711 def test_functionality(url, token):
712     """
713     Tests the functionality of the keystone API.
714 
715     :parameter:
716       - url:    (Dictionary) Dictionary of all URL paths that the test runs
      ↪  against.
717       - token:  (string)     Keystone token.
718     """
719 
720     print_color(TermColor.BLUE,
721 
                 ↪  "\n\n----------------------------------------------------------------------")
722     print_color(TermColor.BLUE, "*** Test section: Keystone functionality ***")
723 
724     headers = {
725         'Content-Type': 'application/json',
726         'X-Auth-Token': token,
```

```
727     }
728
729     # ------------------------------------------
730     # Domain creation:
731     print_color(TermColor.YELLOW, "\nCreating the domain:")
732
733     domain_description = "new test domain"
734     domain_name = "NewTestDomain"
735
736     # Payload to be sent in the POST request that creates a new domain.
737     create_domain_payload = {
738         "domain": {
739             "description": domain_description,
740             "enabled": True,
741             "name": domain_name
742         }
743     }
744
745     # Sends a POST request to create a new domain in keystone.
746     _, domain_body = post_request(url['domains'], headers, create_domain_payload)
747
748     # Return if 'domain' is not in the response body.
749     if 'domain' not in domain_body:
750         return
751
752     # Gets the ID of the domain.
753     domain_id = domain_body['domain']['id']
754
755     # ------------------------------------------
756     # Project creation:
757     print_color(TermColor.YELLOW, "\nCreating the project:")
758
759     project_description = "New Test project"
760     project_name = "NewTestProject"
761
762     # Payload to be sent in the POST request that creates a new project.
763     create_project_payload = {
764         "project": {
765             "description": project_description,
766             "domain_id": domain_id,
767             "enabled": True,
768             "is_domain": False,
769             "name": project_name
770         }
771     }
772
773     # Sends the POST request to create the project.
774     _, project_body = post_request(url['projects'], headers,
        ↪  create_project_payload)
775
776     # Return if 'project' is not in the response body.
777     if 'project' not in project_body:
778         return
779
780     # Gets the ID of the project.
781     project_id = project_body['project']['id']
782
783     # ------------------------------------------
```

```
784     # User creation:
785     print_color(TermColor.YELLOW, "\nCreating the user:")
786 
787     user_name = "New Test SkyHigh"
788     user_description = "New Test SkyHigh user"
789     user_mail = "newTest@skyhigh.test"
790 
791     # Payload to be sent in the POST request that creates a new user.
792     create_user_payload = {
793         "user": {
794             "default_project_id": project_id,
795             "domain_id": domain_id,
796             "enabled": True,
797             "name": user_name,
798             "password": "verySecretPassword",
799             "description": user_description,
800             "email": user_mail,
801             "options": {
802                 "ignore_password_expiry": True
803             }
804         }
805     }
806 
807     # Sends a POST request to create a new user in keystone.
808     _, user_body = post_request(url['users'], headers, create_user_payload)
809 
810     # Return if 'user' is not in the response body.
811     if 'user' not in user_body:
812         return
813 
814     # Gets the ID of the user.
815     user_id = user_body['user']['id']
816 
817     # ---------------------------------------------
818     # Group creation:
819     print_color(TermColor.YELLOW, "\nCreating the group and adding the user to
        ↪  the group:")
820 
821     group_description = "New Test group"
822     group_name = "NewTestGroup"
823 
824     # Payload to be sent in the POST request that creates a new group.
825     create_group_payload = {
826         "group": {
827             "description": group_description,
828             "domain_id": domain_id,
829             "name": group_name
830         }
831     }
832 
833     # Sends the POST request to create the group.
834     _, group_body = post_request(url['groups'], headers, create_group_payload)
835     print("")
836 
837     # Return if 'group' is not in the response body.
838     if 'group' not in group_body:
839         return
840 
```

```
841     # Gets the ID of the group.
842     group_id = group_body['group']['id']
843
844     # Adds user to the group.
845     put_request(url['groups'] + "/" + group_id + "/users/" + user_id, headers)
846
847     # ------------------------------------------
848     # Role creation:
849     print_color(TermColor.YELLOW,
850                 "\nCreating the role and assigning it to the group on the
                     ↪  project:")
851
852     role_description = "New Test role"
853     role_name = "NewTestRole"
854
855     # Payload to be sent in the POST request that creates a new role.
856     create_role_payload = {
857         "role": {
858             "description": role_description,
859             "domain_id": domain_id,
860             "name": role_name
861         }
862     }
863
864     # Sends the POST request to create the region.
865     _, role_body = post_request(url['roles'], headers, create_role_payload)
866     print("")
867
868     # Return if 'role' is not in the response body.
869     if 'role' not in role_body:
870         return
871
872     # Gets the ID of the role.
873     role_id = role_body['role']['id']
874
875     # Assign role to group on project.
876     put_request(url['projects'] + "/" + project_id
877                 + "/groups/" + group_id + "/roles/" + role_id, headers)
878
879     # ------------------------------------------
880     # Check config:
881
882     # *** Make sure the project got is part of the correct domain. ***
883     # Sends the GET request and gets the OpenStack project.
884     print_color(TermColor.YELLOW,
885                 "\nChecking that the project got created and is part of the
                     ↪  domain:")
886
887     project_data = get_request(url['projects'] + "/" + project_id, headers)
888
889     # Create a dictionary of what fields that should contain the same data and
         ↪  checks if they do.
890     project_dictionary = {
891         project_data['project']['domain_id']: domain_id,
892         project_data['project']['description']: project_description,
893         project_data['project']['name']: project_name
894     }
895     check_values("\"/v3/projects\"", project_dictionary)
```

```
896
897     # *** Make sure the user is part of the project and domain. ***
898     print_color(TermColor.YELLOW,
899                 "\nChecking that the user got created "
900                 "and that the user is part of the project and domain:")
901
902     # Sends the GET request and gets the OpenStack user.
903     user_data = get_request(url['users'] + "/" + user_id, headers)
904
905     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
906     user_dictionary = {
907         user_data['user']['default_project_id']: project_id,
908         user_data['user']['domain_id']: domain_id,
909         user_data['user']['name']: user_name
910     }
911     check_values("\"/v3/users\"", user_dictionary)
912
913     # *** Make sure the user is part of the group and the group is part of the
        ↪  domain. ***
914     print_color(TermColor.YELLOW,
915                 "\nChecking that the user is part of the group, "
916                 "and that the group is part of the domain:")
917
918     # Sends the GET request and gets the OpenStack users groups.
919     user_groups_data = get_request(url['users'] + "/" + user_id + "/groups",
        ↪  headers)
920
921     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
922     user_groups_dictionary = {
923         user_groups_data['groups'][0]['description']: group_description,
924         user_groups_data['groups'][0]['domain_id']: domain_id,
925         user_groups_data['groups'][0]['name']: group_name
926     }
927     check_values("\"/v3/users/<user_id>/groups\"", user_groups_dictionary)
928     print("")
929
930     # *** Make sure that the role is assigned to the group on the project. ***
931     # Sends the GET request and gets the OpenStack role for the group on
        ↪  project.
932     role_data = get_request(url['projects'] + "/" +
933                             project_id + "/groups/" + group_id + "/roles",
                                ↪  headers)
934
935     # Create a dictionary of what fields that should contain the same data and
        ↪  checks if they do.
936     role_dictionary = {
937         role_data['roles'][0]['id']: role_id,
938         role_data['roles'][0]['name']: role_name
939     }
940     check_values("\"/v3/projects/<project_id>/groups/<group_id>/roles\"",
        ↪  role_dictionary)
941
942     # ---------------------------------------------
943     # Cleanup:
944     print_color(TermColor.YELLOW, "\nDeletes all resources that where created:")
945
```

```
946     # Deletes the role.
947     delete_requests(url['roles'] + "/" + role_id, headers)
948     print("")
949 
950     # Deletes the group.
951     delete_requests(url['groups'] + "/" + group_id, headers)
952     print("")
953 
954     # Deletes the user.
955     delete_requests(url['users'] + "/" + user_id, headers)
956     print("")
957 
958     # Deletes the project.
959     delete_requests(url['projects'] + "/" + project_id, headers)
960     print("")
961 
962     # Update the domain to be disabled.
963     patch_payload = {
964         "domain": {
965             "enabled": False
966         }
967     }
968 
969     # Sends a PATCH request to mark the domain as inactive.
970     patch_request(url['domains'] + "/" + domain_id, headers, patch_payload)
971     print("")
972 
973     # Deletes the domain.
974     delete_requests(url['domains'] + "/" + domain_id, headers)
975 
976 
977 #
    ↪  ********************************************************************************************
978 # MAIN:
979 
980 print("*********************************************************************"
981       "\n                    ~ Keystone API tests ~"
982       "\n---------------------------------------------------------------------")
983 
984 # Set local variables from ENV variables if the exist.
985 if os.environ.get("VAGRANT_IP_AGENT") is None:
986     print_color(TermColor.RED, "ERROR: Environmental variable 'VAGRANT_IP_AGENT'
        ↪  is not set.")
987     exit(1)
988 KEYSTONE_URL = "http://" + os.environ["VAGRANT_IP_AGENT"] + ":5000"
989 
990 if os.environ.get("KEYSTONE_PASSWORD") is None:
991     print_color(TermColor.RED, "ERROR: Environmental variable 'KEYSTONE_PASSWORD'
        ↪  is not set.")
992     exit(1)
993 KEYSTONE_PASSWORD = os.environ["KEYSTONE_PASSWORD"]
994 
995 if os.environ.get("KEYSTONE_TOKEN") is None:
996     print_color(TermColor.RED, "ERROR: Environmental variable 'KEYSTONE_TOKEN' is
        ↪  not set.")
997     exit(1)
998 KEYSTONE_TOKEN = os.environ["KEYSTONE_TOKEN"]
999 
```

```
1000 # Dictionary of URL paths that should be tested.
1001 URL_PATHS = {
1002     "tokens":   KEYSTONE_URL + "/v3/auth/tokens",
1003     "domains":  KEYSTONE_URL + "/v3/domains",
1004     "groups":   KEYSTONE_URL + "/v3/groups",
1005     "projects": KEYSTONE_URL + "/v3/projects",
1006     "regions":  KEYSTONE_URL + "/v3/regions",
1007     "roles":    KEYSTONE_URL + "/v3/roles",
1008     "services": KEYSTONE_URL + "/v3/services",
1009     "users":    KEYSTONE_URL + "/v3/users"
1010 }
1011
1012 # Tests to run:
1013 test_token(URL_PATHS['tokens'], KEYSTONE_PASSWORD, KEYSTONE_TOKEN)
1014 test_domain(URL_PATHS['domains'], KEYSTONE_TOKEN)
1015 test_groups(URL_PATHS['groups'], KEYSTONE_TOKEN)
1016 test_projects(URL_PATHS['projects'], KEYSTONE_TOKEN)
1017 test_regions(URL_PATHS['regions'], KEYSTONE_TOKEN)
1018 test_roles(URL_PATHS['roles'], KEYSTONE_TOKEN)
1019 test_services(URL_PATHS['services'], KEYSTONE_TOKEN)
1020 test_users(URL_PATHS['users'], KEYSTONE_TOKEN)
1021 test_functionality(URL_PATHS, KEYSTONE_TOKEN)
1022
1023
1024 # Finished running tests.
1025 print("\n\n**********************************************************************"
1026       "\n                    ~ Test summary ~"
1027       "\n----------------------------------------------------------------------")
1028
1029 if FAILED_TESTS:
1030     print("One or more tests have failed \n  "
1031           "Passed tests:", NUM_PASSED_TESTS, "\n  Failed tests:",
              ↪  NUM_FAILED_TESTS, "\n\n")
1032     exit(1)
1033
1034 print("All tests passed \n  "
1035       "Passed tests:", NUM_PASSED_TESTS, "\n  Failed tests:", NUM_FAILED_TESTS,
           ↪  "\n\n")
```

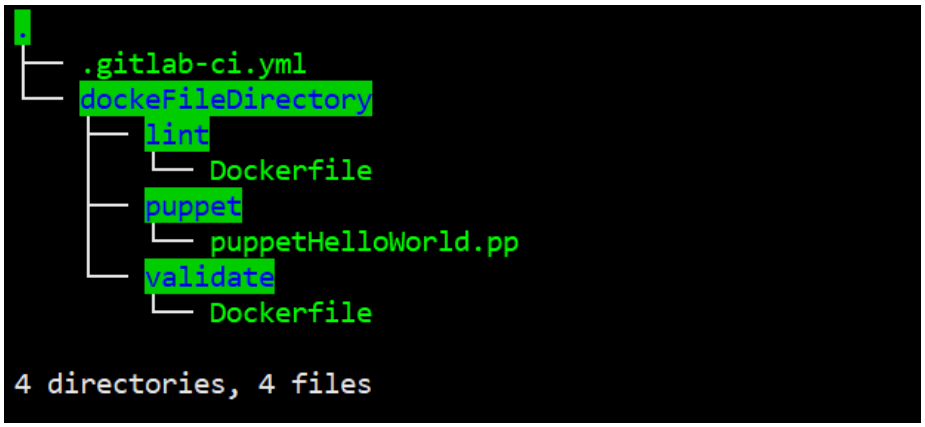
E Utprøving av GitLab CI

E.1 Metode 1

Flerstegs pipeline med containere og kjøring av tester direkte i containeren. I første stadiet bygger man et docker image. Deretter kjører man tester direkte inne i containeren

Til denne metoden er det mulig å splitte opp jobbene over 2 gitlab-runnere. Hvilket kan være nyting hvis hver test tar litt tid. Valg av gitlab-runner og om testene kjøres på en eller flere gitlab-runnere er overlat til gitlab å bestemme.

Figur 16: mappe og filstruktur



Ovenfor er et screenshot av alle mapper og filer som ble brukt til å prøve denne metoden.

Figur 17: puppet fil

puppetHelloWorld.pp 176 Bytes

```
1  # Veldig enkel puppet fil
2  class helloworld {
3    file { '/etc/motd':
4      owner   => 'root',
5      group   => 'root',
6      mode    => '0644',
7      content => "hello, world!\n",
8    }
9  }
```

Ovenfor er en veldig enkel puppetfil som brukes til å utføre tester

Figur 18: .gitlab-ci.yml

.gitlab-ci.yml 839 Bytes

```
stages:
    - build
    - test

build1:
    stage: build
    script:
        - echo "Building validate docker image"
        - docker build --cache-from validate:latest --tag validate:latest dockeFileDirectory/validate/.

test1:
    stage: test
    dependencies:
        - build1
    script:
        - echo "Validating puppet code"
        - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest


build2:
    stage: build
    script:
        - echo "Building lint docker images"
        - docker build --cache-from lint:latest --tag lint:latest dockeFileDirectory/lint/.

test2:
    stage: test
    dependencies:
        - build2
    script:
        - echo "Linting puppet code"
        - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest


```

Ovenfor er filen som styrer hva gitlab ci pipelinen skal gjøre

Figur 19: lint/Dockerfile

Dockerfile 207 Bytes

```
FROM ubuntu:18.04

# Install puppet-lint
RUN apt-get update && \
    apt-get -y install puppet-lint

WORKDIR /root/puppet/

ENTRYPOINT ["/usr/bin/puppet-lint"]
CMD ["--no-autoloader_layout-check", "." ]
```

Ovenfor er Dockerfilen som blir brukt til å lage en container som skal kjøre linting test på tidligere nent puppetfil

Figur 20: validate/Dockerfile

Dockerfile 415 Bytes

```
1  FROM ubuntu:18.04
2
3  RUN apt-get update && \
4      apt-get install wget -y
5
6  # Installs puppet.
7  RUN tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXX) || exit 1 && \
8      wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb && \
9      dpkg -i "$tempdeb" && \
10     apt-get update && \
11     apt-get -y install puppetserver
12
13 WORKDIR /root/puppet/
14
15 CMD ["/opt/puppetlabs/bin/puppet","parser",  "validate", "."]
```

Ovenfor er Dockerfilen som blir brukt til å lage en container som skal kjøre syntaks validering av tidligere nent puppetfil

Case: ingen feil

Pipeline konsoll output med ingen feil av typen ERROR eller WARNING I filen
Introdusert feil: ingen

Figur 21: Runner: Test

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on Test _k1yS5Ya
Using Shell executor...
Running on gitlab-runner-test...
Fetching changes...
HEAD is now at e0115f6 Update Dockerfile
Checking out e0115f67 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Linting puppet code"
Linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest
Job succeeded
```

Figur 22: Runner: second runner

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on second runner 6Do2CRuc
Using Shell executor...
Running on gitlab-runner-1...
Fetching changes...
HEAD is now at e0115f6 Update Dockerfile
Checking out e0115f67 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Validating puppet code"
Validating puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest
Job succeeded
```

Case: syntaks feil

Pipeline konsoll output med syntaks feil av typen ERROR I filen
Introdusert feil (fjernet en ’): file { ’/etc/motd’: " -> file { /etc/motd’: "

Figur 23: Runner: Test

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on Test _k1yS5Ya
Using Shell executor...
Running on gitlab-runner-test...
Fetching changes...
HEAD is now at 8359fe5 Update puppetHelloWorld.pp
Checking out 8359fe5f as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Linting puppet code"
Linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest
ERROR: Syntax error on line 6
Try running `puppet parser validate <file>`
ERROR: Job failed: exit status 1
```

Figur 24: Runner: second runner

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on second runner 6Do2CRuc
Using Shell executor...
Running on gitlab-runner-1...
Fetching changes...
HEAD is now at 8359fe5 Update puppetHelloWorld.pp
Checking out 8359fe5f as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Validating puppet code"
Validating puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest
Error: Could not parse for environment production: Syntax error at 'motd' (file: /root/puppet/puppetHelloWorld.pp, line: 3, column: 15)
ERROR: Job failed: exit status 1
```

Case: linting feil ERROR

Pipeline konsoll output med linting feil av typen ERROR I filen
Introdusert feil (whitspace): "owner => 'root'," -> "owner => 'root', "

Figur 25: Runner: Test

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on Test _k1yS5Ya
Using Shell executor...
Running on gitlab-runner-test...
Fetching changes...
HEAD is now at 2f96ba5 Update puppetHelloWorld.pp
Checking out 2f96ba51 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Linting puppet code"
Linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest
ERROR: trailing whitespace found on line 4
ERROR: Job failed: exit status 1
```

Figur 26: Runner: second runner

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on second runner 6Do2CRuc
Using Shell executor...
Running on gitlab-runner-1...
Fetching changes...
HEAD is now at 2f96ba5 Update puppetHelloWorld.pp
Checking out 2f96ba51 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Validating puppet code"
Validating puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest
Job succeeded
```

Case: linting feil WARNING

Pipeline konsoll output med linting feil av typen WARNING I filen
Introdusert feil (flyttet =>): "owner => 'root', " -> "owner=> 'root'"

Figur 27: Runner: Test

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on Test _k1yS5Ya
Using Shell executor...
Running on gitlab-runner-test...
Fetching changes...
HEAD is now at 9b16459 Update puppetHelloWorld.pp
Checking out 9b164597 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Validating puppet code"
Validating puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t validate:latest
Job succeeded
```

Figur 28: Runner: second runner

```
Running with gitlab-runner 11.8.0 (4745a6f3)
  on second runner 6Do2CRuc
Using Shell executor...
Running on gitlab-runner-1...
Fetching changes...
HEAD is now at 9b16459 Update puppetHelloWorld.pp
Checking out 9b164597 as Metode_1_flere_stegs_pipeline_med_docker_og_kjøring_med_direkte_tester_i_containeren...
Skipping Git submodules setup
$ echo "Linting puppet code"
Linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t lint:latest
WARNING: indentation of => is not properly aligned (expected in column 13, but found it in column 10) on line 4
Job succeeded
```

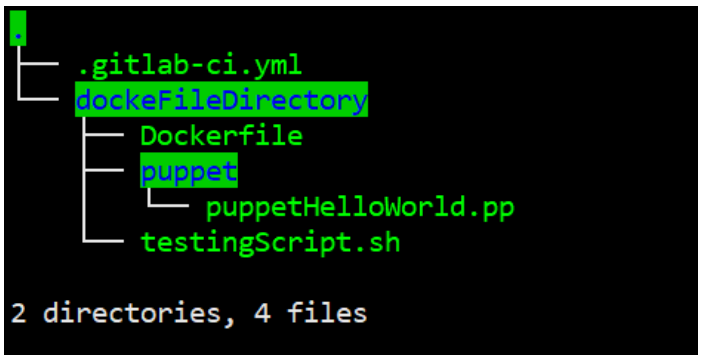
Metode 1 observasjoner

Buil1 og test1, kjører på en runner mens, build2 og test2 kjører på en annen
Build1 tar lang tid, og siden build1 og build2 er i samme stage(build), må begge bli ferdige før noen av dem kan gå videre til neste stage(test). Dette er ineffektivt.
En feil av typen WARNING får ikke pipelinen til å erklære jobben som feilet.
Det ser ut som om puppet-lint også utfører en form som syntaks sjekking av koden, ettersom den også reagerte når vi introduserte en enkel syntaks feil.

E.2 Metode 2

I denne metoden er det kun 1 container som kjører alle testene via at entrypoint-scrip

Figur 29



Ovenfor er et screenshot av alle mapper og filer som ble brukt til å prøve denne metoden.

Figur 30

puppetHelloWorld.pp 176 Bytes

```
1  # Veldig enkel puppet fil
2  class helloworld {
3    file { '/etc/motd':
4      owner   => 'root',
5      group   => 'root',
6      mode    => '0644',
7      content => "hello, world!\n",
8    }
9  }
```

Figur 31

.gitlab-ci.yml 526 Bytes

```
stages:
    - build
    - test

build:
    stage: build
    script:
        - echo "Building docker images"
        - docker build --cache-from checkcode:latest --tag checkcode:latest dockeFileDirectory/.

test:
    stage: test
    dependencies:
        - build
    script:
        - echo "Validating and linting puppet code"
        - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
        - docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest linter
```

Ovenfor er filen som styrer hva gitlab ci pipelinen skal gjøre

Figur 32

Dockerfile 615 Bytes

```
FROM ubuntu:18.04

# Install utility tool.
RUN apt-get update && \
    apt-get -y install wget

# Adds new repo to apt-get.
RUN tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXXX) || exit 1 && \
    wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb && \
    dpkg -i "$tempdeb"

# Installs puppet and puppet lint.
RUN apt-get update && \
    apt-get -y install puppetserver && \
    apt-get -y install puppet-lint

# Set workdir and copy in script.
WORKDIR /root
RUN mkdir /root/puppet
COPY testingScript.sh /root/testingScript.sh

RUN chmod +x testingScript.sh

ENTRYPOINT ["./testingScript.sh"]
```

Ovenfor er Dockerfilen som blir brukt til å lage en container som skal kjøre et script som utfører syntaks validering og lintingsjekk på tidligere nent puppetfi

Figur 33

testingScript.sh 391 Bytes

```
#!/bin/bash

cd /root/puppet/

if [ "$1" == "parser" ]; then
  echo "Running puppet parser validate"
  find $(pwd) -name '*.pp' | xargs -n 1 -t /opt/puppetlabs/bin/puppet parser validate

elif [ "$1" == "linter"  ]; then
  echo "Running puppet lint"
  find $(pwd) -name '*.pp' | xargs -n 1 -t /usr/bin/puppet-lint --no-autoloader_layout-check

else
  echo "Error: No parameters given"

fi
```

Ovenfor er et entrypoint-script som utfører syntaks validering og lintingsjekk

Case: ingen feil

Pipeline konsoll output med ingen feil av typen ERROR eller WARNING I filen
Introdusert feil: ingen

Figur 34

```
$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
Running puppet parser validate
/opt/puppetlabs/bin/puppet parser validate /root/puppet/puppetHelloWorld.pp
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest linter
Running puppet lint
/usr/bin/puppet-lint --no-autoloader_layout-check /root/puppet/puppetHelloWorld.pp
Job succeeded
```

Case: syntaks feil

Pipeline konsoll output med syntaks feil av typen ERROR I filen
Introdusert feil (fjernet en ’): file { ’/etc/motd’: " -> file { /etc/motd’: "

Figur 35


$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
Running puppet parser validate
/opt/puppetlabs/bin/puppet parser validate /root/puppet/puppetHelloWorld.pp
Error: Could not parse for environment production: Syntax error at 'motd' (file: /root/puppet/puppetHelloWorld.PP, line: 3, column: 15)
ERROR: Job failed: exit status 1

Case: linting feil ERROR

Pipeline konsoll output med linting feil av typen ERROR I filen
Introdusert feil (whitspace): "owner => 'root'," -> "owner => 'root', "

Figur 36

```
$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
Running puppet parser validate
/opt/puppetlabs/bin/puppet parser validate /root/puppet/puppetHelloWorld.pp
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest linter
Running puppet lint
/usr/bin/puppet-lint --no-autoloader_layout-check /root/puppet/puppetHelloWorld.pp
ERROR: trailing whitespace found on line 4
ERROR: Job failed: exit status 1
```

Case: linting feil WARNING

Pipeline konsoll output med linting feil av typen WARNING I filen
Introdusert feil (flyttet =>): "owner => 'root', " -> "owner=> 'root'"

Figur 37

```
$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest parser
Running puppet parser validate
/opt/puppetlabs/bin/puppet parser validate /root/puppet/puppetHelloWorld.pp
$ docker run -v $(pwd)/dockeFileDirectory/puppet:/root/puppet/ -t checkcode:latest linter
Running puppet lint
/usr/bin/puppet-lint --no-autoloader_layout-check /root/puppet/puppetHelloWorld.pp
WARNING: indentation of => is not properly aligned (expected in column 13, but found it in column 10) on line 4
Job succeeded
```

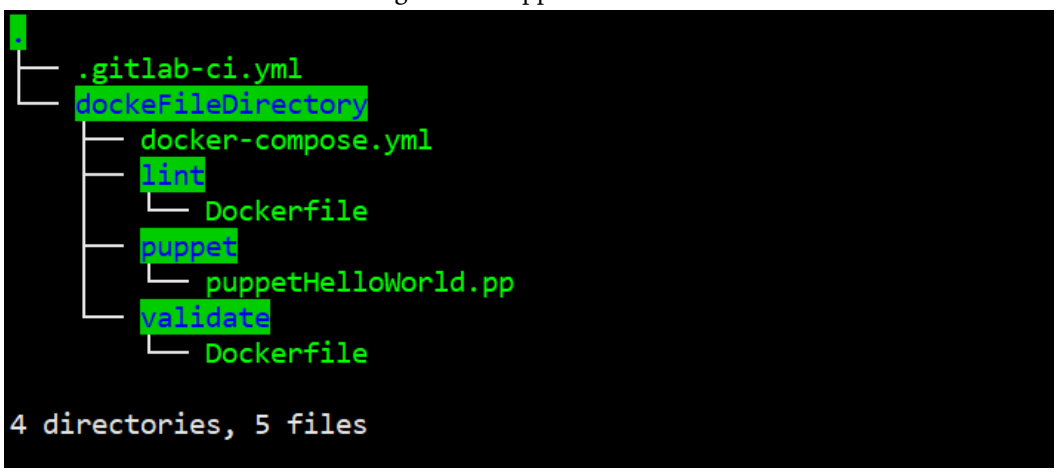
Metode 2 observasjoner

En feil av typen WARNING får ikke pipelinen til å erklære jobben som feilet.
Flere linjer med kode en ved metode 5
Flere filer enn ved metode 5
Trenger ikke å installere mer programvare på host server
Gitlab CI vil kjøre stager sekvensielt[59], men stagene kan kjøres på flere forskjellige gitlab-runnere, og disse kan befinne seg på forskjellige servere. Så for å være sikker på det som lages i build stage, et docker image i vårt tilfelle, er tilgjengelig for den gitlab-runnere som skal kjøre tester staget. Må man definere dependencies og/eller artifacts i .gitlab-ci.yml.
Krever kunnskap om bash, gitlab ci og docker
Trenger ikke å rydde opp på serveren etterpå

E.3 Metode 3

I dette forsøke ble det bruk docker-compose til å starte flere containere samtidig. Containerene kjører så tester i henhold til det som ble spesifisert i seksjonen Utprøving avgitlab CI.

Figur 38: mappestruktur



Figur 39

docker-compose.yml 247 Bytes

```yaml
version: '3.5'

services:
  lint:
    build:
      context: .
      dockerfile: ./lint/Dockerfile
    container_name: puppet_lint

  validate:
    build:
      context: .
      dockerfile: ./validate/Dockerfile
    container_name: puppet_validate
```

Figur 40

puppetHelloWorld.pp 176 Bytes

```
# Veldig enkel puppet fil
class helloworld {
  file { '/etc/motd':
    owner   => 'root',
    group   => 'root',
    mode    => '0644',
    content => "hello, world!\n",
  }
}
```

Figur 41

Dockerfile 240 Bytes

```
1  FROM ubuntu:18.04
2
3  RUN mkdir /puppetfiles
4
5  COPY /puppet/*.pp /puppetfiles
6
7  RUN apt-get update && \
8      apt-get -y install puppet-lint
9
10 WORKDIR /puppetfiles
11
12 ENTRYPOINT ["/usr/bin/puppet-lint"]
13 CMD ["--no-autoloader_layout-check", "." ]
```

Figur 42

Dockerfile 470 Bytes

```
FROM ubuntu:18.04

RUN mkdir /puppetfiles

COPY /puppet/*.pp /puppetfiles

RUN apt-get update && \
    apt-get install wget -y

# Installs puppet.
RUN tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXXX) || exit 1 && \
    wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb && \
    dpkg -i "$tempdeb" && \
    apt-get update && \
    apt-get -y install puppetserver

WORKDIR /puppetfiles

CMD ["/opt/puppetlabs/bin/puppet","parser",  "validate", "."]
```

Case: ingen feil

Pipeline konsoll output med ingen feil av typen ERROR eller WARNING I filen
Introdusert feil: ingen

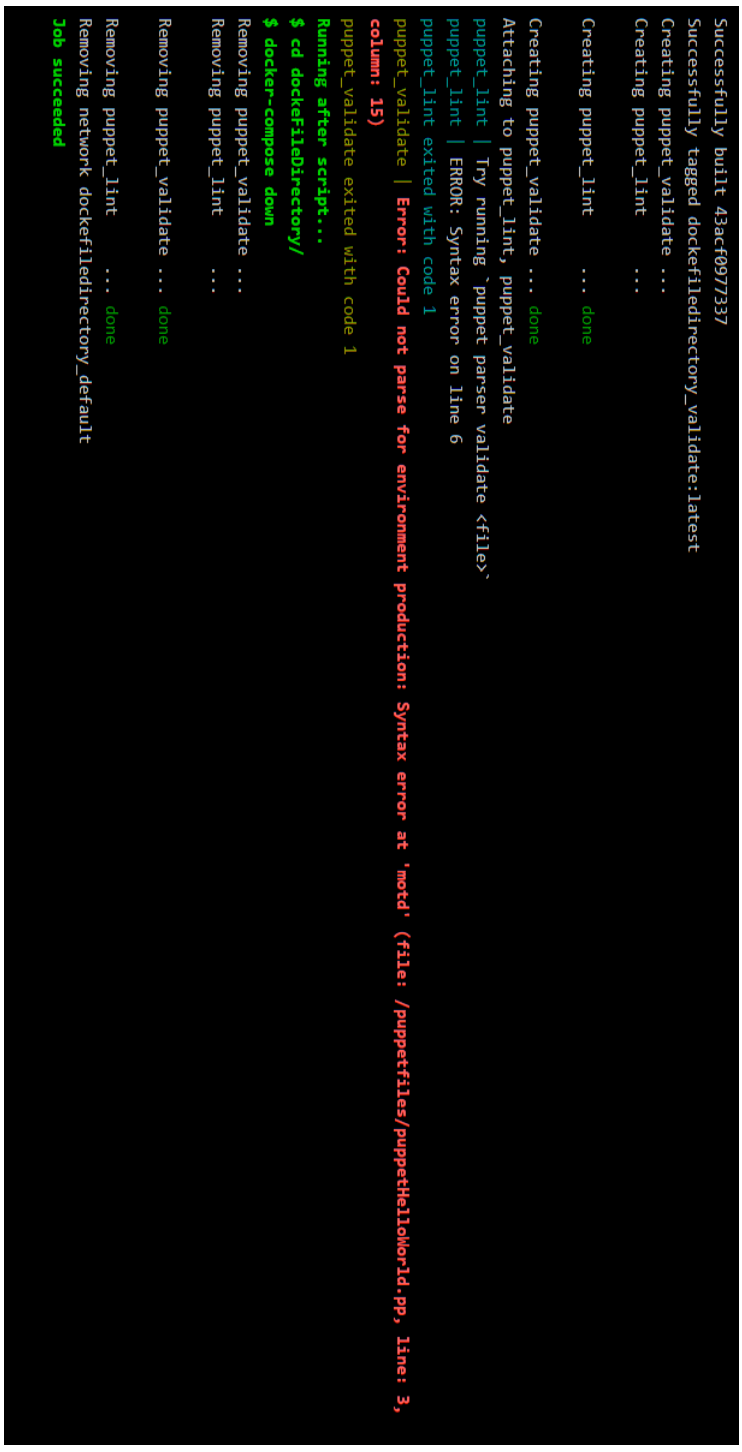
Figur 43

```
Successfully built 213b991458df
Successfully tagged dockefiledirectory_lint:latest
Creating puppet_lint ...
Creating puppet_validate ...
Creating puppet_lint
Creating puppet_validate

Creating puppet_lint ... done

Creating puppet_validate ... done
Attaching to puppet_lint, puppet_validate
puppet_lint exited with code 0
puppet_validate exited with code 0
Running after script...
$ cd dockeFileDirectory/
$ docker-compose down
Removing puppet_validate ...
Removing puppet_lint     ...

Removing puppet_lint     ... done

Removing puppet_validate ... done
Removing network dockefiledirectory_default
Job succeeded
```

Case: syntaks feil

Pipeline konsoll output med syntaks feil av typen ERROR I filen
Introdusert feil (fjernet en ’): file { ’/etc/motd’: " -> file { /etc/motd’: "

Figur 44



Case: linting feil ERROR

Pipeline konsoll output med linting feil av typen ERROR I filen
Introdusert feil (whitspace): "owner => 'root'," -> "owner => 'root', "

Figur 45

```
Successfully built 8ae16c1020dc
Successfully tagged dockefiledirectory_validate:latest
Creating puppet_validate ...
Creating puppet_lint     ...

Creating puppet_validate ... done

Creating puppet_lint     ... done
Attaching to puppet_validate, puppet_lint
puppet_lint | ERROR: trailing whitespace found on line 4
puppet_validate exited with code 0
puppet_lint exited with code 1
Running after script...
$ cd dockeFileDirectory/
$ docker-compose down
Removing puppet_lint     ...
Removing puppet_validate ...

Removing puppet_validate ... done

Removing puppet_lint     ... done
Removing network dockefiledirectory_default
Job succeeded
```

Case: linting feil WARNING

Pipeline konsoll output med linting feil av typen WARNING I filen
Introdusert feil (flyttet =>): "owner => 'root', " -> "owner=> 'root'"

Figur 46

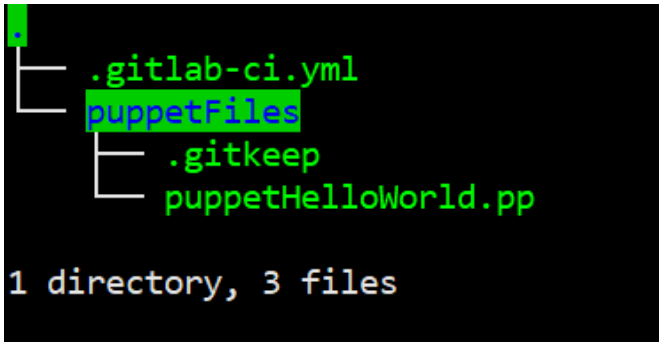
```
Successfully built dc2a76976944
Successfully tagged dockefiledirectory_validate:latest
Creating puppet_lint ...
Creating puppet_validate ...

Creating puppet_lint     ... done

Creating puppet_validate ... done
Attaching to puppet_lint, puppet_validate
puppet_lint | WARNING: indentation of => is not properly aligned (expected in column 13, but found it in column 10) on line 4
puppet_lint exited with code 0
puppet_validate exited with code 0
Running after script...
$ cd dockeFileDirectory/
$ docker-compose down
Removing puppet_lint     ...
Removing puppet_validate ...

Removing puppet_lint     ... done

Removing puppet_validate ... done
Removing network dockefiledirectory_default
Job succeeded
```

Observasjoner metode 4

Hverken feil av typen ERROR eller WARNING vil forårsake at pipeline jobben feiler
Mer overhead
Krever kunnskap om bash, gitlab ci, docker og docker-compose
Tregere å kjøre
Gjør .gitlab-ci.yml: mye mindre og oversiktlig
Trenger ikke å rydde opp på serveren etterpå
Trenger ikke å installere mer programvare på host server

E.4 Metode 4

Figur 47



Ovenfor er et screenshot av alle mapper og filer som ble brukt til å prøve denne metoden.

Figur 48

puppetHelloWorld.pp 176 Bytes

```
1  # Veldig enkel puppet fil
2  class helloworld {
3    file { '/etc/motd':
4      owner   => 'root',
5      group   => 'root',
6      mode    => '0644',
7      content => "hello, world!\n",
8    }
9  }
```

Figur 49

.gitlab-ci.yml 644 Bytes

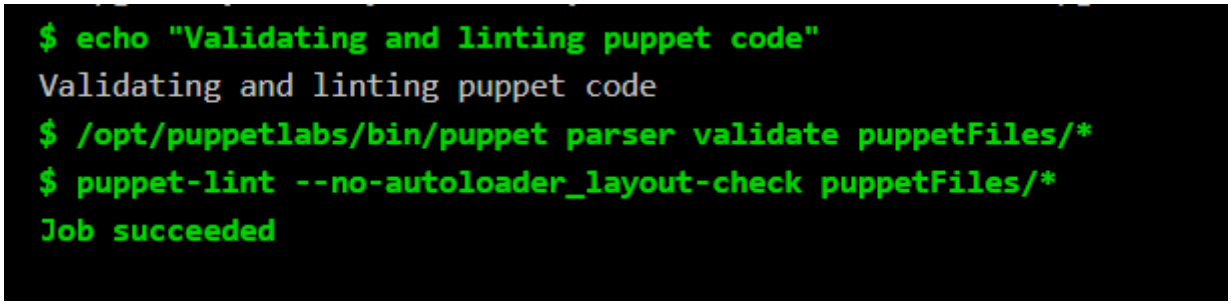
```
stages:
    - test

# Installer programmvare
before_script:
    - sudo apt-get update
    - sudo apt-get install wget -y
    - tempdeb=$(mktemp /tmp/debpackage.XXXXXXXXXXXXXXXXX) || exit 1
    - wget -O "$tempdeb" https://apt.puppet.com/puppet5-release-bionic.deb
    - sudo dpkg -i "$tempdeb"
    - sudo apt-get update
    - sudo apt-get install puppetserver -y
    - sudo apt-get install puppet-lint -y

# Kjør tester
test:
    stage: test
    script:
        - echo "Validating and linting puppet code"
        - /opt/puppetlabs/bin/puppet parser validate puppetFiles/*
        - puppet-lint --no-autoloader_layout-check puppetFiles/*
```

Ovenfor er filen som styrer hva gitlab ci pipelinen skal gjøre

Case: ingen feil

Pipeline konsoll output med ingen feil av typen ERROR eller WARNING I filen
Introdusert feil: ingen

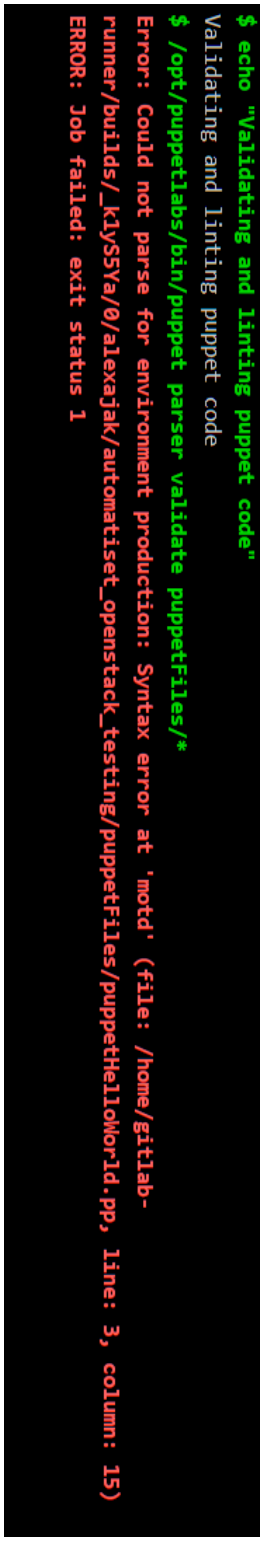
Figur 50



Case: syntaks feil

Pipeline konsoll output med syntaks feil av typen ERROR I filen
Introdusert feil (fjernet en ’): file { ’/etc/motd’: " -> file { /etc/motd’: "

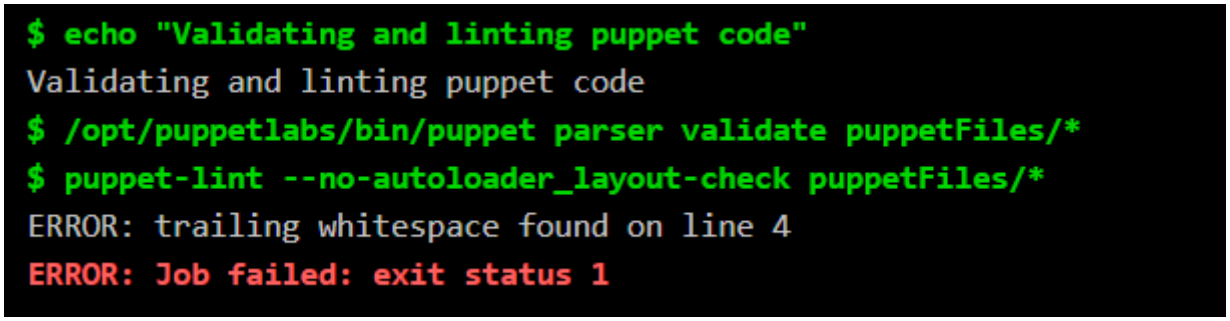
Figur 51


$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ /opt/puppetlabs/bin/puppet parser validate puppetFiles/*
Error: Could not parse for environment production: Syntax error at 'motd' (file: /home/gitlab-runner/builds/_k1ySSYa/0/alexajak/automatiset_openstack_testing/puppetFiles/puppetHelloWorld.pp, line: 3, column: 15)
ERROR: Job failed: exit status 1

Case: linting feil ERROR

Pipeline konsoll output med linting feil av typen ERROR I filen
Introdusert feil (whitspace): "owner => 'root'," -> "owner => 'root', "

Figur 52



Case: linting feil WARNING

Pipeline konsoll output med linting feil av typen WARNING I filen
Introdusert feil (flyttet =>): "owner => 'root', " -> "owner=> 'root'"

Figur 53

```
$ echo "Validating and linting puppet code"
Validating and linting puppet code
$ /opt/puppetlabs/bin/puppet parser validate puppetFiles/*
$ puppet-lint --no-autoloader_layout-check puppetFiles/*
WARNING: indentation of => is not properly aligned (expected in column 13, but found it in column 10) on line 4
Job succeeded
```

Observasjoner metode 5

En feil av typen WARNING får ikke pipelinen til å erklære jobben som feilet.
Host serveren må installere programvare
Host server trenger mer konfigurasjon
Host server brukeren gitlab-runnertrenger flere rettigheter
Færrest antall linjer med kode
Færrest antall filer i repoet
Krever kunnskap om bash og gitlab ci
Kan være behov for å rydde opp på serveren etterpå

F Sammenlikninger av pipelinesystemer

F.1 Sammenlikninger

Valg av CI/CD platform

Krav

- Er best om den kjører lokalt.
- kunne installeres og håndteres med Puppet.
- Det er ikke aktuelt å kjøpe inn lisenser til programvare.
- Vedlikeholdbarhet og robusthet er viktigere en omfattende funksjonalitet.
- Det er ikke aktuelt å bruke store ressurser på å drifte testsystemene.
- Systemer som skal kjøre automatiserte tester skal konfigureres ved bruk av Puppet.

Utvalgte CI/CD plattformer hentet fra: https://blog.aquasec.com/10-essential-container-ci/cd-tools

1. Jenkins

Hoved Kilde: https://www.cloudbees.com/jenkins/about
Jenkins er et Open Source prosjekt med lang fartstid. Det er nok det mest populære CI verktøyet per dags dato. Med en estimert brukermasse på over 1,5 millioner. Prosjektet er i kontinuerlig utvikling med siste kommit til dens github konto, for kun et par dager siden(https://github.com/jenkinsci). Jenkins kan brukes til testing av veldig mange språk ettersom den har et stort antall med plugins. Og det er kompatibelt med veldig mange versjon kontrollsystemer.

Jenkins kan brukes til alle CI/ CD stadier. For vår er det aktuelt å bruke det til statis kode analyse, deployment og testing. Docker kan brukes til deployment hvis det viser seg ønskelig. Vi forventer at det er god dokumentasjon på Jenkins generelt, men ikke nødvendigvis så mye på individuelle plugins.

Jenkins er en aktuell kandidat for vårt prosjekt fordi det er gratis, har mange plugins og er et modent system.

2. Microsoft VSTS

VSTS is now Azure DevOps!

kilde: https://www.youtube.com/watch?v=JhqpF-5E10I
Kan brukes med alle språk

vanskelig å finne relevant informasjon, kun marketing info

3. Bamboo (Atlassian Dev Tools)

Kilde: https://www.atlassian.com/software/bamboo

Bamboo et CI verktøy laget av Atlassian. Og er godt integrert med deres andre produkter som Jira og Bitbucket. Bamboo har også bedre native støtte for ting som git og noen andre ting en det Jenkins har.

Det er uvisst om dette verktøyet er egnet for bruk men puppet.

Det er gratis for bruk opp til 10 jobber. Deretter er prisen 550 dollar (https://www.atlassian.com/purchase/product/bamboo). Det har støtte for conteinere.

4. GitLab CI/CD

Guide:https://about.gitlab.com/2016/07/29/the-basics-of-gitlab-ci/
Dokumentasjon: https://docs.gitlab.com/ee/ci/

GitLab CI/CD er en integrert del av GitLab Prosjektet. Det er ikke gratis, men Det brukes allerede på NTNU Gjøvik. Og vår veileder Erik Hjelmås har nevnt at det kan i fremtiden være aktuelt for NTNU å benytte det mer enn de gjør i dag. Det er derfor ikke umulig at prisen er overkommelig.

GitLab CI/CD støtter bruk av Docker og kubernetes ifølge deres egen dokumentasjon. Det kan kjøres lokalt slik at NTNU har den kontrollen de ønsker. Men det er ukjent hvorvidt dette er egnet for bruk men puppet og openstack

Kan være aktuel.

5. Codeship

Basic version:
Deployment via egendefinert script på pre konfigurere maskiner(Ubuntu 14.04)(https://documentation.codeship.com/basic/builds-and-configuration/packages/). Puppet er ikke installert. Støtter Ruby men ikke puppet. Så det er usikkert hvordan dette vil fungere. Rspec er et testrammeverk for Ruby som brukes av puppet. Støtter konteinere. Kjøres ikke lokalt
Tror ikke den kan deploye til openstack. Codeship er ikke gratis

Pro version:
Bruker egendefinerte Docker Files

6. Codefresh

Kilde: https://codefresh.io/features/
Codefresh har et stort fokus på conteinertjenester, og da kubernetes spesielt. Codefresh kan ikke brukes med
VMer(https://codefresh.io/docs/docs/configure-ci-cd-pipeline/introduction-to-codefresh-pipelines/)

Det har en kompleks prismodell(mer en 2 typer) der en versjon er gratis, men da kan man kun ha 1 build om gangen og kun 120 builds i måneden. Vi tror dette kan bli vanskelig å bruke sammen med dette prosjektet ettersom Codefresh er bygget fra bunnen av med konteinere som deployment løsning. Mens det er ikke puppet, og siden SkyHigh ikke bruker containere pr dags dato kan det kanskje gjøre det vanskelig å teste ting. Det er ukjent om dette kjøres lokalt eller ikke.

7. TeamCity

Støtter Ruby og Rspec(https://www.jetbrains.com/teamcity/features/technology_awareness.html), støtter ikke openstack natively. En begrenset utgave er tilgjengelig gratis hvis du allerede har en lisens.(https://www.jetbrains.com/teamcity/buy/#license-type=new-license) vi finner ingen dokumentasjon angående docke. Ser ut til å ha en openstack plugin til å skape VMer i openstack(https://github.com/yandex-qatools/teamcity-openstack-plugin). Denne pluginen er under aktiv utvikling pr dags dato. Dokumentasjonen er forteller ikke hvordan dette fungerer teknisk. Det er ukjent om det kan brukes docker. Eller om man kan deploye

Kan være aktuel.

8. Travis CI

Dokumentasjon: https://docs.travis-ci.com/
Integrert inn i Github. Rene VMer for hvert build, gratis for open source prosjekter. Kompleks prismodell ellers(mer en 2) . gratis for open source. Travis støtter både builds i containere og VMer. støtter ruby og mange andre språk.

Veldig usikker på om dette er en kandidat eller ikke

9. GoCD

Dokumentasjon: https://docs.gocd.org/current/
Har veldig kompleks konfigurasjonsmuligheter. Kan bruke kommandolinjen til å kjøre ting. Virker som et kompleks verktøy med stort potensiale. Men sannsynligvis mye å lære, kanskje vanskelig å bruke kan ha stor fleksibilitet. GoCD er gratis, men det er en Enterprise utgave med støtte for add-ons og plugins. Docker kan brukes med plugin støtte. Den kjører lokalt.

GoCD har en openstack plugin som kan starte og stoppe build agenter i openstack(https://www.gocd.org/2018/10/16/new-gocd-features.html)

Kan være en aktuell kandidat, men forvent en del arbeide + vanskelighetsgrad

10. CircleCI

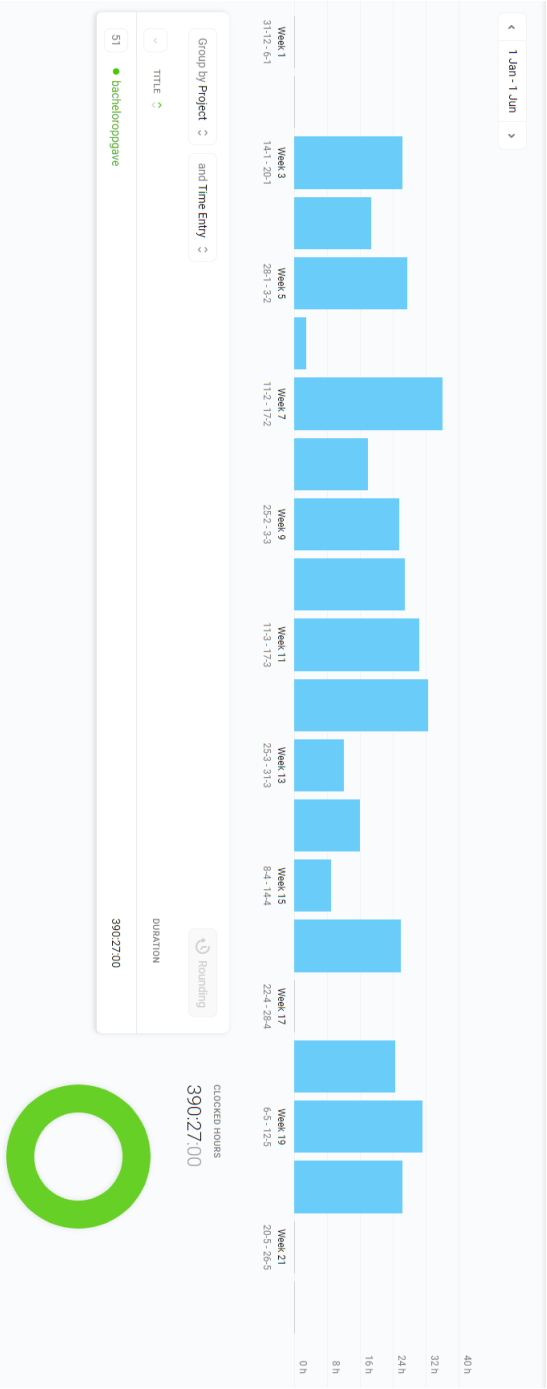
Støtter kun container builds. Kun 1 gratis container. Kan kjøres lokalt mot betaling
Finner ingen dokumentasjon for bruk med puppet eller openstack

Platform	Docker	VM	Kjøres lokalt	Pris	Installeres og håndteres med Puppet.	Resurs Krevende	Generell Fleksibilitet	Enkelt å bruke	OpenStack støtte
Jenkins	ja		ja	gratis	ja		stor		plugins
Microsoft VSTS									
Bamboo (Atlassian Dev Tools)	ja	ukjent	ukjent	Enkel gratis versjon					
GitLab	ja	ja	ja	Mange prismodeller					
Codeship	ja	ja	nei	Nei					
Codefresh	ja	nei	ukjent	Begrenset gratis versjon					
TeamCity	ukjent	ukjent	ukjent	Gratis hvis du allerede har lisens					plugins
Travis CI	ja	ja		Gratis for					

				open source					
GoCD	ukjent	ja	Nei	Gratis			stor		plugins
CircleCI	ja								

G Timelister

G.1 Timeliste Alexander Jakobsen



Detailed report

2019-01-01 - 2019-12-31

Total 390 h 27 min

Date	Description	Duration	User
01-14	**gjennomgang av tom røyse forelesning**	**2:00:00**	Alexajak
	bacheloroppgave	09:00-11:00	
01-14	**lest artikler om automatisert testing Automated testing**	**2:00:00**	Alexajak
	bacheloroppgave	11:00-13:00	
01-14	**gjennomgår andre bacheloroppgaver**	**1:00:00**	Alexajak
	bacheloroppgave	13:00-14:00	
01-14	**gjennomgår toggle som et prosjetverktøy**	**1:00:00**	Alexajak
	bacheloroppgave	14:00-15:00	
01-14	**satt opp prosjekt på overleafe / sherelatex**	**2:30:00**	Alexajak
	bacheloroppgave	16:00-18:30	
01-14	**gjennomgår andre bacheloroppgaver**	**1:00:00**	Alexajak
	bacheloroppgave	19:00-20:00	
01-16	**møte med veileder og oppdragsgiver**	**0:30:00**	Alexajak
	bacheloroppgave	09:30-10:00	
01-18	**skrive forprosjekt**	**10:00:00**	Alexajak
	bacheloroppgave	08:15-18:15	
01-20	**researched Openstack Tempest og CI/CD**	**6:15:00**	Alexajak
	bacheloroppgave	09:15-15:30	
01-21	**researched ci / cd, laget gantt-skjema, prosjektavtale, skrevet forprosjekt, forbredelser til møte**	**6:30:00**	Alexajak
	bacheloroppgave	08:30-15:00	
01-21	**researched ci / cd, laget gantt-skjema, prosjektavtale, skrevet forprosjekt, forbredelser til møte**	**1:45:00**	Alexajak
	bacheloroppgave	17:00-18:45	
01-22	**møte + referat Egil og Lars Erik**	**1:00:00**	Alexajak
	bacheloroppgave	13:00-14:00	
01-23	**møte med veileder + referat**	**1:00:00**	Alexajak
	bacheloroppgave	09:30-10:30	
01-25	**research, statusmøte, prosjektplan**	**8:30:00**	Alexajak
	bacheloroppgave	07:00-15:30	
01-28	**research, prosjektavtale, regelavtale, Ruby**	**6:00:00**	Alexajak
	bacheloroppgave	10:00-16:00	
01-30	**møte + referat Egil og Lars Erik, prosjektavtale forprosjekt**	**3:00:00**	Alexajak
	bacheloroppgave	09:15-12:15	
02-01	**statusmøte, studert og prøvd å bruke oppdragsgivers puppet kode, satt opp puppet master - agent inftastructur.**	**9:30:00**	Alexajak
	bacheloroppgave	08:00-17:30	

02-03	**div forsøk på å installere keystone** bacheloroppgave	**9:00:00** 09:00-18:00	Alexajak
02-06	**møte + referat Egil og Lars Erik,** bacheloroppgave	**1:00:00** 09:30-10:30	Alexajak
02-08	**statusmøte, laget rapport struktur** bacheloroppgave	**2:00:00** 09:00-11:00	Alexajak
02-12	**møte + referat Egil og Lars Erik,** bacheloroppgave	**1:15:00** 12:30-13:45	Alexajak
02-13	**prvde å installere keystone** bacheloroppgave	**10:00:00** 13:00-23:00	Alexajak
02-14	**pøvde å installere keystone** bacheloroppgave	**8:15:00** 12:00-20:15	Alexajak
02-15	**installere keystone via apt, lagde ci/cd tegning på draw.io** bacheloroppgave	**6:30:00** 09:00-15:30	Alexajak
02-15	**statusmøte** bacheloroppgave	**0:40:00** 17:40-18:20	Alexajak
02-17	**gjenomgått features av CI/CD platformer** bacheloroppgave	**9:30:00** 09:00-18:30	Alexajak
02-18	**gjenomgått features av CI/CD platformer, Discord møte med mats, skrevet om Jenkins** bacheloroppgave	**9:15:00** 07:00-16:15	Alexajak
02-20	**møte med vileder og møterefferat** bacheloroppgave	**1:00:00** 09:30-10:30	Alexajak
02-21	**møte + referat Egil og Lars Erik. jobbet med mats** bacheloroppgave	**5:00:00** 09:00-14:00	Alexajak
02-21	**jobben med å dokumentere infrastrukturn vi fikk laget sammen med Eigil og Lars Erik** bacheloroppgave	**2:45:00** 15:15-18:00	Alexajak
02-25	**skriving, Gitlab CI** bacheloroppgave	**2:30:00** 07:30-10:00	Alexajak
02-25	**skriving, Gitlab CI, møte med mats** bacheloroppgave	**3:45:00** 12:15-16:00	Alexajak
02-25	**skriving** bacheloroppgave	**1:45:00** 16:00-17:45	Alexajak
03-01	**møte med mats, gitlab utprøving** bacheloroppgave	**9:00:00** 09:00-18:00	Alexajak
03-03	**jobben med å dokumentere CI infrastrukturen vi jobbet med på fredag** bacheloroppgave	**8:30:00** 07:00-15:30	Alexajak
03-04	**jobben med å dokumentere CI infrastrukturen vi jobbet med fredag, så på bruk av docker executor, vagrant og rspec** bacheloroppgave	**9:00:00** 07:00-16:00	Alexajak
03-06	**researched rspec** bacheloroppgave	**0:30:00** 08:00-08:30	Alexajak

03-06	**møte med veileder** bacheloroppgave	**0:30:00** 09:30-10:00	Alexajak
03-08	**rettet skrivefeil i testcase 1 fra forrige uke. jobbet med å deplye servere via vagrant til openstack(ikke lyktes så langt)** bacheloroppgave	**7:30:00** 08:00-15:30	Alexajak
03-08	**statusmøte med mats** bacheloroppgave	**0:30:00** 17:30-18:00	Alexajak
03-10	**fikk vagrant til å starte instanser i SkyHigh, fikk vagrant til å kjøre i en docker container, fikk vagrant containeren til å køre i gitlab CI.** bacheloroppgave	**9:00:00** 07:00-16:00	Alexajak
03-11	**jobbet med vagrant povoisjonering og passord håndtering** bacheloroppgave	**10:00:00** 07:00-17:00	Alexajak
03-13	**møte med veileder + møterefferat** bacheloroppgave	**1:00:00** 09:30-10:30	Alexajak
03-15	**jobbet med å deploye til openstack** bacheloroppgave	**8:00:00** 07:00-15:00	Alexajak
03-17	**jobbet med å deploye til openstack** bacheloroppgave	**11:30:00** 08:30-20:00	Alexajak
03-18	**jobbet med å deploye til openstack** bacheloroppgave	**15:37:00** 07:00-22:37	Alexajak
03-21	**jobbet med å deploye til openstack** bacheloroppgave	**8:00:00** 10:00-18:00	Alexajak
03-22	**status møte, automatisk branch mege ved ci suksess** bacheloroppgave	**9:00:00** 09:00-18:00	Alexajak
03-25	**jobbet med å merge brancher** bacheloroppgave	**4:00:00** 08:00-12:00	Alexajak
03-27	**domo med veileder og oppdragsgiver + møterefferat** bacheloroppgave	**1:00:00** 09:30-10:30	Alexajak
03-29	**status møte, lagde git pre-commit script som kjører tester ved commit** bacheloroppgave	**7:00:00** 09:00-16:00	Alexajak
04-01	**refakrorerte et script(auto_merge), møte med mats, lagde et auto_merge_request script** bacheloroppgave	**9:30:00** 08:30-18:00	Alexajak
04-05	**status møte** bacheloroppgave	**2:00:00** 09:00-11:00	Alexajak
04-07	**refakrorerte et script, slik at vi ikke lenger har problemer med filer som ikke kan slettes** bacheloroppgave	**4:30:00** 10:00-14:30	Alexajak
04-14	**Rapportskriving** bacheloroppgave	**9:00:00** 09:00-18:00	Alexajak
04-15	**Rapportskriving** bacheloroppgave	**7:00:00** 12:00-19:00	Alexajak
04-16	**Rapportskriving** bacheloroppgave	**9:00:00** 09:00-18:00	Alexajak

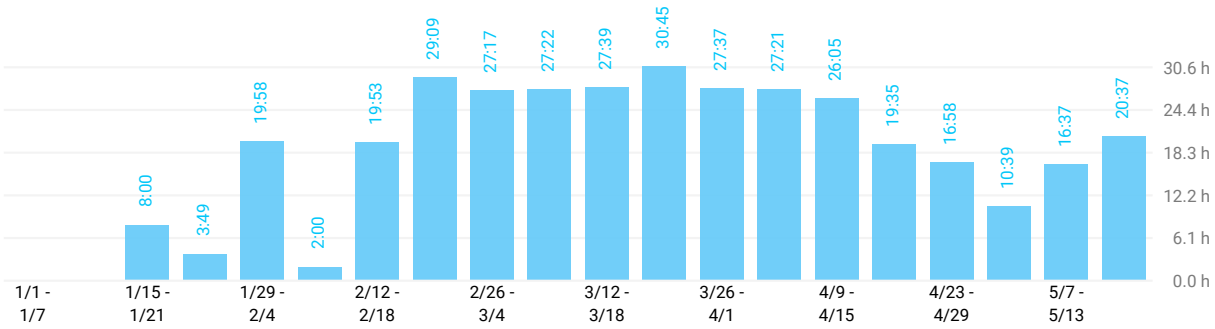
04-17	**Rapportskriving**	**10:00:00**	Alexajak
	bacheloroppgave	08:00-18:00	
05-03	**Rapportskriving**	**10:30:00**	Alexajak
	bacheloroppgave	09:00-19:30	
05-04	**Rapportskriving**	**6:00:00**	Alexajak
	bacheloroppgave	11:30-17:30	
05-05	**Rapportskriving**	**8:10:00**	Alexajak
	bacheloroppgave	08:00-16:10	
05-06	**Rapportskriving**	**8:45:00**	Alexajak
	bacheloroppgave	08:30-17:15	
05-07	**Rapportskriving**	**7:00:00**	Alexajak
	bacheloroppgave	09:00-16:00	
05-08	**Møte med veileder**	**0:30:00**	Alexajak
	bacheloroppgave	09:30-10:00	
05-09	**Rapportskriving**	**10:30:00**	Alexajak
	bacheloroppgave	09:00-19:30	
05-10	**Rapportskriving**	**4:30:00**	Alexajak
	bacheloroppgave	09:00-13:30	
05-14	**Rapportskriving**	**8:00:00**	Alexajak
	bacheloroppgave	08:00-16:00	
05-15	**Rapportskriving**	**6:00:00**	Alexajak
	bacheloroppgave	09:00-15:00	
05-16	**Rapportskriving**	**4:15:00**	Alexajak
	bacheloroppgave	08:00-12:15	
05-18	**Rapportskriving**	**6:00:00**	Alexajak
	bacheloroppgave	11:00-17:00	
05-19	**Innleveringsarbeide**	**2:00:00**	Alexajak
	bacheloroppgave	10:00-12:00	

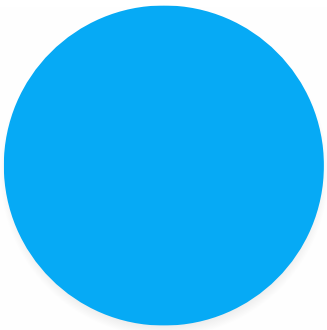
G.2 Timeliste Mats Ove Mandt Skjærstein

Summary Report

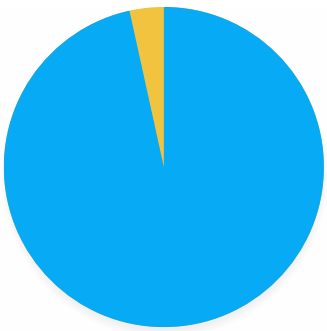
January 01, 2019 – May 19, 2019

TOTAL HOURS: 361:28:57



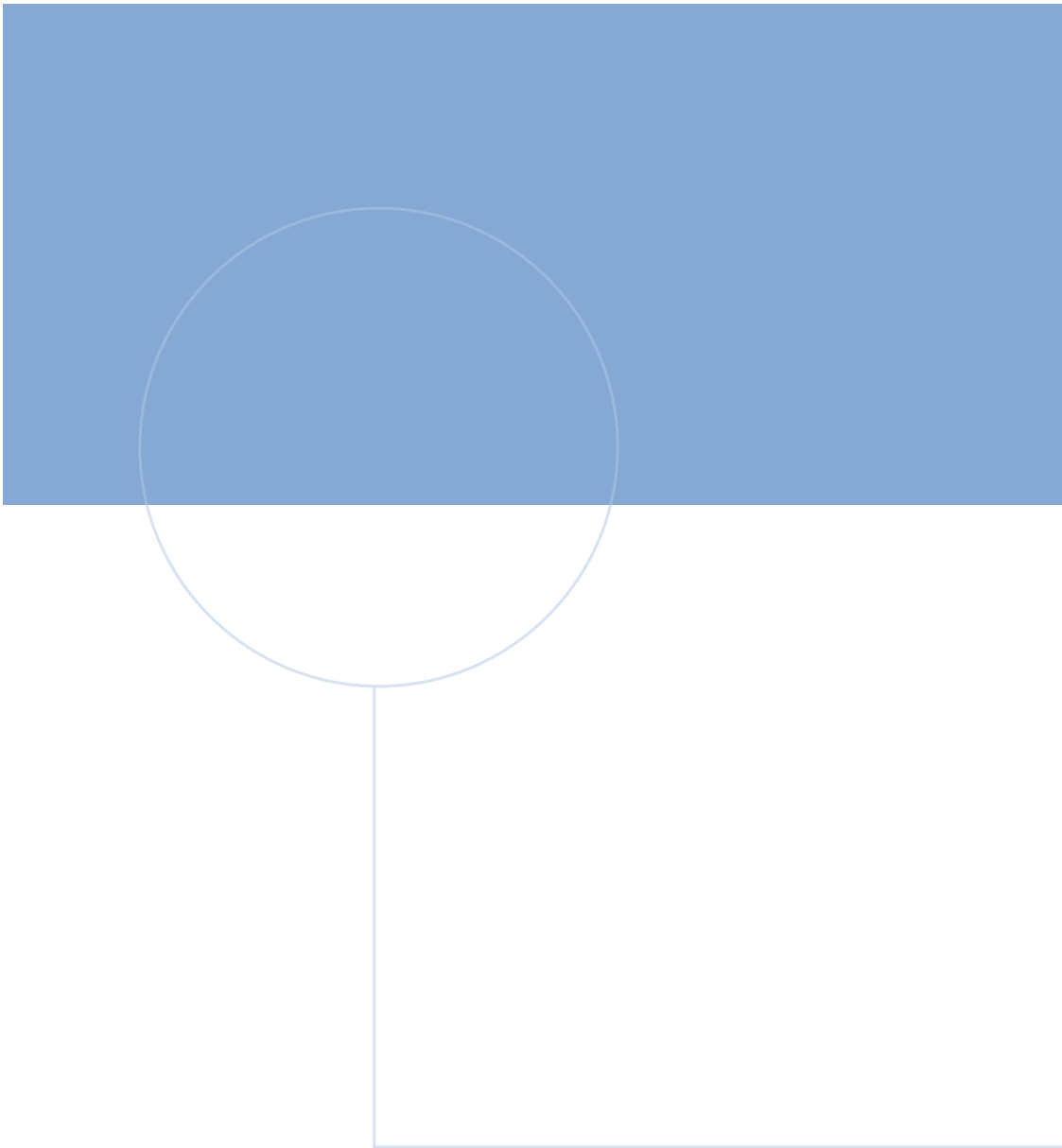


PROJECT	DURATION
Bachelor	361:28:57



TIME ENTRY	DURATION
Bachelor oppgave	348:47:01
Forprosjekt	12:41:56

PROJECT - TIME ENTRY	DURATION
Bachelor	361:28:57
Bachelor oppgave	348:47:01
Forprosjekt	12:41:56



NTNU

Kunnskap for en bedre verden