UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

IMPLEMENTACIJA PROCESA ZA KONTINUIRANO UVOĐENJE U EKSPLOATACIJU MIKROSERVISA

Master rad

Mentor:
Doc. dr Mihajlo Savić

Kandidat:
Njegoš Railić

Banja Luka, decembar 2021.

UNIVERSITY OF BANJA LUKA

FACULTY OF ELECTRICAL ENGINEERING

THE IMPLEMENTATION OF CONTINUOUS DEPLOYMENT PROCESS FOR MICROSERVICES

Master thesis

Mentor:
Asst. Prof. Mihajlo Savić, PhD

Candidate:
Njegoš Railić

Banja Luka, December 2021.

Tema: **IMPLEMENTACIJA PROCESA ZA KONTINUIRANO UVOĐENJE U EKSPLOATACIJU MIKROSERVISA**

Mentor: **Dr Mihajlo Savić, docent,**
Elektrotehnički fakultet,
Univerzitet u Banjoj Luci

Naučna oblast: **Inženjerstvo i tehnologija**

Naučno polje: **Elektrotehnika, elektronika i informaciono inženjerstvo**

Ključne riječi: automatizacija, CI, CD, CDE, kontinualna isporuka, kontejneri, mikroservisna arhitektura, testiranje, virtuelizacija

Klasifikaciona oznaka (CERIF): **T120**

Sažetak: U ovom radu je predložen i u potpunosti implementiran sistem za kontinuirano uvođenje u eksploataciju i upravljanje isporukom mikroservisa. Predloženi pristup zasnovan je na industrijskim standardima i implementiran korištenjem tehnologija otvorenog koda. Na osnovu predloženog pristupa implementiran je vrlo robustan i modularan CI/CD sistem, koji je olakšao dizajn, implementaciju kao i eventualna buduća proširenja i izmjene sistema. Implementirano rješenje se sastoji od dvije komponente: softverske biblioteke za *Jenkins* alat kao i klijentske aplikacije za orkestraciju isporuke. Pristup je eksperimentalno verifikovan.

Topic: **THE IMPLEMENTATION OF THE CONTINUOUS DEPLOYMENT PROCESS FOR MICROSERVICES**

Mentor: **Dr Mihajlo Savić, Assistant Professor,**
Faculty of Electrical Engineering,
University of Banja Luka

Scientific area: **Engineering and technology**

Scientific field: **Electrical engineering, electronics and information engineering**

Keywords: automation, CI, CD, CDE, continuous delivery, containers, microservices, pipelines, testing, virtualization

Classification label (CERIF): **T120**

Abstract: In this thesis, we proposed and implemented an approach for managing the deployment of microservice-based applications. The proposed approach is based on industry standards and implemented using open-source technologies. Based on the approach a very robust and modular CI-CD system is implemented, which has facilitated the design, implementation as well as future expansions and adaptations of the system. The implementation consists of two parts: a library for the *Jenkins* Continuous Integration system and a CLI application for orchestrating deployments. The approach is experimentally verified.

Mojim roditeljima,

koji su potrebe svoje djece uvijek stavljali ispred svojih.

Supruzi Nini,

za inspiraciju i nesebičnu podršku bez koje ovaj rad nikad ne bi bio završen.

LISTA KORIŠĆENIH SKRAĆENICA

API – Application Programming Interface
ARM – Acorn RISC Machine
ARP – Address Resolution Protocol
BSD – Berkeley Software Distribution
CA – Certificate Authority
CAP – CAP Theorem (Consistency, Availability, Partition Tolerance)
CD – Continuous Deployment
CDE – Continuous Delivery
CI – Continuous Integration
CLI – Command-line Interface
CNI – Container Network Interface
CORBA – Common Object Request Broker Architecture
CR – Container Runtime
CRD – Custom Resource Definition
CRI – Container Runtime Interface
DC – Deployment Controller
DNS – Domain Name System
DS – Deployment Strategy
DVCS – Distributed Version Control Systems
ESB – Enterprise Service Bus
ETCD – Open-source distributed key-value store
GRPC – Google Remote Procedure Call
HA – High Availability
HTTP(S) – Hypertext Transfer Protocol (Secure)
IA64 – Intel Itanium architecture
IaC – Infrastructure as Code
IPVS – IP Virtual Server
ISA – Instruction Set Architecture
ITIL – Information Technology Infrastructure Library
Java RMI – Java Remote Method Invocation
JNLP – Java Network Launching Protocol
JSL – Jenkins Shared Library
JSON – JavaScript Object Notation
K8s – Kubernetes
KCP – Kubernetes Control Plane
KVM – Kernel-based Virtual Machine
MAC – Media Access Control Address
NAT – Network Address Translation
OCI – Open Container Initiative
OS – Operativni sistem
PV – Paravitualizacija
QA – Quality Assurance
RBAC – Role Based Access Control
REST – Representational State Transfer
RPC – Remote Procedure Call
RS – Replica Set
S/390 – IBM Enterprise Systems Architecture/390
SCM – Software configuration Management

SOA – Service Oriented Achitecture
SSH – Secure Shell
SSL – Secure Sockets Layer
SSO – Single Sign On
TCP – Transmission Control Protocol
URI – Uniform Resource Identifier path
UUID – Universally Unique Identifier
VCS – Version Control Systems
VM – Virtuelna mašina
VMM – Monitor virtuelne mašine
YAML – Yet Another Markup Language

SADRŽAJ

Uz rad je priložena elektronska verzija rada.

1. UVOD

1.1. Predmet istraživanja

Već decenijama, industrijski zahtjevi i stvarne potrebe imaju za posljedicu da se softverski dizajn i arhitekture razvijaju u raznim pravcima. Sve veća složenost poslovnih aplikacija i njihovih zahtjeva, potreba za upravljanjem promjenama i evolucijom softverskih sistema rezultovali su razvojem različitih arhitektura kao što su CORBA (eng. *Common Object Request Broker Architecture*), Java RMI (eng. *Java Remote Method Invocation*) ili ESB (eng. *Enterprise Service Bus*) [1]. Servisno-orijentisana arhitektura SOA (eng. *Service Oriented Achitecture*) postala je odgovor na kompleksne industrijske potrebe velikih preduzeća, preuzimajući u domenu aplikativne arhitekture dominaciju na tržištu. Međutim, uskoro je SOA koncept evoluirao i dobio svog nasljednika, mikroservisnu arhitekturu. Iako fundamentalni pojmovi koji stoje iza mikroservisne arhitekture nisu novi, savremena primjena mikroservisne arhitekture jeste. Njeno usvajanje dijelom je podstaknuto izazovima skalabilnosti, nedovoljnom efikasnošću prethodnih pristupa, sporim razvojem i nedovoljnom brzinom isporuke softvera kao i poteškoćama adaptacije novih tehnologija, koje nastaju kada su složeni softverski sistemi implementirani kao jedna velika monolitna aplikacija. Implementacija aplikacije kao kolekcije mikroservisa, bilo od početka ili dekompozicijom postojeće monolitne aplikacije, u značajnoj mjeri rješava ove probleme. Uz mikroservisnu arhitekturu, takvu aplikaciju je moguće lako horizontalno i vertikalno skalirati, produktivnost i brzina razvoja se drastično povećavaju, a stare tehnologije se lako mogu zamijeniti novim.

Uz sve veću konkurenciju na tržištu softvera, organizacije posvećuju sve više pažnje i ulažu resurse u unapređenje i automatizaciju sistema za isporuku visokokvalitetnog softvera [2]. Linija za uvođenje u eksploataciju (eng. *Deployment pipeline*) predstavlja automatizaciju procesa za isporuku softvera, od sistema za kontrolu verzije (eng. *Version Control System*) izvornog koda do produkcionog okruženja, odnosno krajnjeg korisnika. To podrazumijeva više koraka uključujući kreiranje izvršne verzije (eng. *build*), nakon čega slijedi nekoliko koraka testiranja i uvođenje u eksploataciju. Osnovni elementi ove linije su alati za puštanje u eksploataciju i kontinuiranu integraciju koji omogućavaju da se prati svaka izmjena tokom životnog ciklusa od trenutka kreiranja do isporuke. Kontinuirana integracija CI (eng. *Continuous integration*), kontinuirana isporuka CDE (eng. *Continuos delivery*) i kontinuirano uvođenje u eksploataciju CD (eng. *Continuous deployment*), neke su od praksi usmjerenih na pomoć organizacijama da ubrzaju razvoj i isporuku softverskih funkcionalnosti kroz nove verzije bez narušavanja kvaliteta [3]. Dok se CI koristi za automatizaciju višestrukih koraka, kao što su kreiranje i testiranje novih verzija, CDE i CD imaju mogućnost brzog i pouzdanog uvođenja u eksploataciju aplikacija uz visok nivo automatizacije [3], [4]. Upotreba prethodno navedenih pristupa i praksi donosi nekoliko koristi: (1) brže dobijanje povratnih informacija od procesa razvoja softvera i korisnika; (2) česta i pouzdana isporuka softvera koja dovodi do većeg zadovoljstva kupaca i kvaliteta samog softverskog proizvoda; (3) razvojni i operativni timovi rade zajedno na automatizaciji procesa čime se manuelni koraci mogu gotovo potpuno eliminisati [5].

U modernom pristupu razvoja softvera, CD pomaže u pojednostavljenju procesa isporuke novih verzija aplikacije automatizujući procese i redukujući pri tome rizik. Moderni softver za orkestraciju kontejnera i menadžment klastera pojednostavljuje upravljanje složenim procesima za uvođenje u eksploataciju, ali ujedno povećava i kompleksnost prilikom implementacije i održavanja takvih infrastruktura. Za automatizaciju procesa kreiranja izvršne verzije i testiranja već postoji više metoda i alata. Međutim, integraciono testiranje i koraci za

uvođenje u eksploataciju novih verzija aplikacija su izuzetno složeni za automatizaciju. Kao sastavni dio procesa za uvođenje u eksploataciju u ovom radu biće obrađene različite strategije (DS – *Deployment Strategy*) softvera kao što su plavo-zeleno i A/B testiranje, uključujući i naprednije metode poput strategije kanarinca. Istraživanja pokazuju da strategija kanarinca [6] postaje veoma popularna u kombinaciji sa CD [7], jer omogućava uvođenje novih verzija softverskih komponenata u produkciono okruženje sa minimalnim rizikom.

1.2. Cilj istraživanja

Osnovni cilj istraživanja bio je da se na osnovu teorijskih postavki, industrijskih standarda i praktičnih primjera modeluje i implementira modularan CD sistem za kontinuirano uvođenje mikroservisa u eksploataciju.

Teorijski doprinos rada predstavljen je u sistematizovanoj analizi problema sa kojima se susrećemo prilikom automatizacije CD procesa kao i u detaljnom pregledu metodologija i tehnologija za uvođenje u eksploataciju novog softvera. Na osnovu analize rezultata i ograničenja odabran je pristup za implementaciju CD rješenja.

Praktični doprinos rada ogleda se u razvoju i konfigurisanju softverskih komponenata, kako bi se implementirao predloženi pristup za automatizaciju CD procesa za mikroservisnu arhitekturu. S obzirom na fleksibilnost, te mogućnosti za nadogradnju i prilagođavanje koje ovo rješenje potencijalno pruža, njegova praktična primjena je moguća u širokom spektru organizacija, bez obzira na veličinu i tehnologije koje su korištene.

1.3. Metodologija

Početak rada čini teorijski uvod u mikroservisnu arhitekturu, njeno poređenje sa drugim arhitekturama informacionih sistema, kao i kratak pregled savremenih tehnologija neophodnih da bi se razumjela mikroservisna arhitektura. Navedeno obuhvata definiciju i teoretske principe vezane za kontejnere i orkestraciju kontejnera, uključujući i kratak opis dodatnih softverskih i infrastrukturnih komponenata sistema koji nisu sastavni dio alata za orkestraciju mikroservisne arhitekture, ali su neophodne da bi se jedan ovakav sistem implementirao u praksi.

Nakon teorijskog uvoda prezentovan je i ukratko objašnjen proces automatizacije isporuke softvera sa detaljnim osvrtom na CI, CDE i CD prakse, kao komponente jednog automatizovanog CI/CD procesa. Na osnovu prikaza strategija za uvođenje u eksploataciju, kao i strategija za testiranje prilikom uvođenja u eksploataciju, formulisane su preporuke i smjernice.

Nakon izbora odgovarajućih alata i tehnologija projektovan je i implamentiran CD sistem, koji automatizujući korake kreiranja artifakata, testiranja i isporuke obezbjeđuje visok kvalitet koda bez unošenja dodatnih grešaka. Verzija artifakta kreirana u ovakom sistemu je uvijek spremna za isporuku u produkciono okruženje, čime se omogućava visokofrekventna, efikasna i pouzdana isporuka mikroservisnih aplikacija na dnevnoj bazi.

Praktična primjena implementiranog alata ilustrovana je na realnom primjeru.

1.4. Struktura rada

Ovaj rad se bavi implementacijom procesa za kontinuirano uvođenje u eksploataciju mikroservisa u korporativnom okruženju. Istraživanje je usmjereno na identifikaciju i analizu problema sa kojima se organizacije suočavaju ukoliko žele da automatizuju proces razvoja i isporuke softvera. Bez obzira na veličinu razvojnih timova i praksi za razvoj softvera koje se koriste, implementacija CI/CD sistema predstavlja veoma kompleksan i vremenski zahtjevan proces. Glavni cilj istraživanja je da se u okviru predložene teme detaljno analiziraju i obrade različiti problemi i izazovi sa kojima se susrećemo prilikom uvođenja jednog ovakvog sistema, a zatim i da prikaže prijedlog rješenja i implementira alate za automatizaciju ovog procesa. U radu je detaljno opisan proces uvođenja u eksploataciju u okviru mikroservisne arhitekture, a potom na osnovu sprovedenog istraživanja odabrane najpogodnije tehnologije za implementaciju efikasnog i automatizovanog CD procesa.

Nakon uvodne glave, druga glava obuhvata teorijski uvod u mikroservisnu arhitekturu, njeno poređenje sa drugim arhitekturama informacionih sistema, kao i kratak pregled savremenih tehnologija neophodnih da bi se razumjela mikroservisna arhitektura. Navedeno obuhvata definiciju i teoretske principe vezane za virtuelizaciju, kontejnerske tehnologije i orkestraciju kontejnera. Potom je dat detaljan opis arhitekture *Kubernetes* platforme za orkestraciju kontejnera sa osvrtom na distribuirane sisteme i pojam eventualne konzistentnosti. Poseban osvrt dat je na umrežavanje kontejnera i sam način rada *Kubernetes* platforme.

Treće poglavlje daje detaljan pregled komponenata linija za uvođenje u eksploataciju softvera. Prvo je dat uopšten pregled sastavnih komponenata jednog CI/CD sistema, a zatim je svaka od njih detaljno opisana. Nakon toga, objašnjena je važnost upotrebe sistema za kontrolu verzija i dat je kratak pregled organizacije repozitorijuma izvornog koda. Zatim su opisani sistemi za kontrolu verzija nakon čega slijede strategije grananja. Kako je osnovna namjena CI linije kreiranje artifakta spremnog za isporuku u produkciju, dat je pregled šema za verzionisanje artifakata i metoda za digitalno potpisivanje koda. Iako je testiranje softvera zasebna disciplina, uzimajući u obzir njenu važnost, dat je kratak pregled testiranja i objašnjena je uloga testiranja u CI/CD sistemima.

Četvrta glava daje pregled strategija za testiranje i uvođenje u eksploataciju sa posebnim osvrtom na razlike između njih. Potom su opisane strategije zasnovane na ponovnom kreiranju, postepenoj zamjeni i plavo-zelenoj isporuci, kao i njihove prednosti i nedostaci. Iako strategije za testiranje prilikom uvođenja u eksploataciju imaju sličnu namjenu, one se koriste prvenstveno za testiranje novih funkcionalnosti softverskih proizvoda pa je dat i kratak pregled A/B testiranja, strategije kanarinca, kao i strategije sjenke. Glava se završava kratkim poređenjem, nakon kojeg slijede preporuke i smjernice.

U petoj glavi dat je prijedlog rješenja za implementaciju CD procesa. Ovdje je data arhitektura rješenja sa opisom osnovnih dijelova. Kako se sistem sastoji od dvije logičke cjeline, prije svega CI linije za kreiranje artifakata uz očuvanje kvaliteta i ispravnosti koda, a zatim i CD linije čija je osnovna uloga pouzdana isporuka softvera, dat je detaljan opis predloženih koraka. Potom je dat je detaljan opis razvijenih komponenata.

Šesta glava opisuje implementaciju predloženog rješenja, prije svega infrastrukture koja je korištena prilikom implementacije i validacije praktičnog dijela rada. Ovdje je dat kratak pregled korištenih alata i način na koji je konfigurisana platforma za orkestraciju mikroservisnih aplikacija. Potom je dat opis implementacije biblioteke za *Jenkins* CI server, kao i klijentske aplikacije za orkestraciju uvođenja u eksploataciju mikroservisnih aplikacija.

Na kraju je opisan način na koji je implementirana progresivna isporuka čime je validirane implementirano rješenje.

Na kraju rada data su zaključna razmatranja kao i preporuke i smjernice za buduća istraživanja.

Nakon zaključka, navedena je korištena literatura.

1.5. Objavljeni rezultati istraživanja

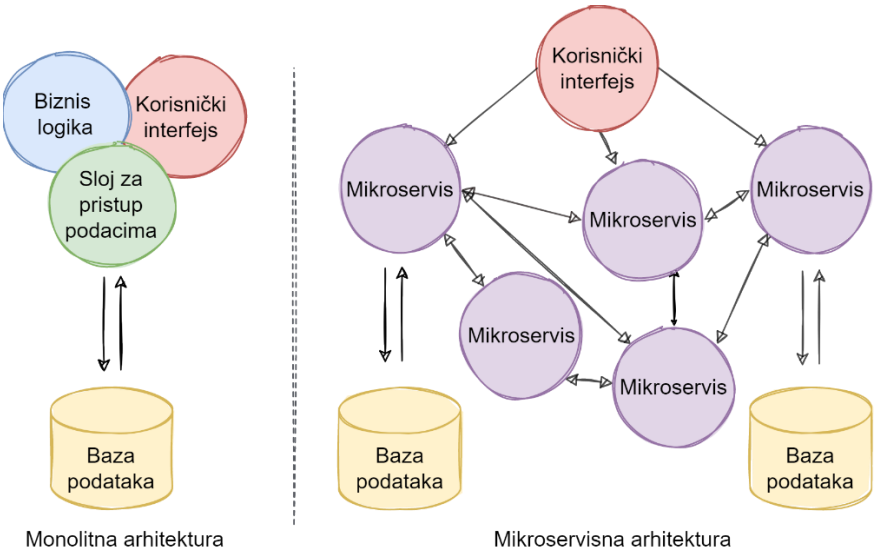
Najznačajniji rezultati istraživanja sprovedenog u okviru izrade ovog master rada, predstavljeni su u radu:

Njegoš Railić, Mihajlo Savić: „Architecting Continuous Integration and Continuous Deployment for Microservice Architecture“, ISBN: 978-1-7281-8230-8, 20th International Symposium INFOTEH, Jahorina 2021.

2. MIKROSERVISNA ARHITEKTURA

Monolitne aplikacije sastoje se od komponenata koje su čvrsto povezane i posmatraju se obično kao jedan entitet prilikom razvoja i procesa isporuke softvera. S vremenom ove aplikacije postaju veće, odnosno kompleksnije, a veze između komponenata čvršće pa samim tim i proces održavanja postaje teži. Nakon nekog vremena, kako kompleksnost sistema raste, kao izazov postavlja se pitanje kako efikasno razvijati i isporučivati nove verzije softvera [8]. Više timova radi na različitim funkcionalnostima i na kraju integrišu veliki broj izmjena u isti repozitorijum izvornog koda. Ciklusi isporuke softvera se obično mjere u danima, sedmicama pa u nekim slučajevima i mjesecima. Čak i najmanja izmjena, na primjer samo jedne linije koda, zahtijeva da se kreira i isporuči nova verzija kompletne aplikacije. Osim toga, sam proces isporuke često nije automatizovan i radi se ručno, čime se povećava mogućnost greške, a u slučaju da je neophodna isporuka neke od starijih verzija aplikacije obično je potrebno još više vremena, ako je to uopšte izvodivo.

Da bi jedna ovakva aplikacija bila isporučena, obično je potreban mali broj servera ili fizičkih mašina koje imaju dovoljno resursa za izvršavanje aplikacije. U slučaju da se broj korisnika poveća obično se pribjegava metodama vertikalnog skaliranja dodavanjem više memorije i procesora, ili horizontalnog skaliranja dodajući više identičnih servera na kojima se izvršava ista verzija aplikacije. Vertikalno skaliranje obično ne zahtijeva promjene u aplikaciji ali veoma brzo postaje skupo i vrlo lako se dostižu limiti dostupnih rješenja. Horizontalno skaliranje, u drugu ruku, je sa aspekta dodatnih hardverskih resursa veoma jeftino, jer skalira gotovo linearno, ali može zahtijevati velike promjene u aplikativnom kodu i nije uvijek izvodivo. Na primjer, veoma teško je horizontalno skalirati relacione baze podataka koje nisu predviđene za takav rad.



Slika 2.1. Uporedni prikaz monolitne i mikroservisne arhitekture

Monolitna aplikacija može biti uspješna ako nije složena i nema potrebe za čestim isporučivanjem novih funkcionalnosti [9]. Međutim, uporedo s rastom poslovne organizacije i razvojem poslovnih procesa, potreba za isporukom novih softverskih funkcionalnosti obično se povećava, kao i složenost samog softvera. Dodavanje novih funkcionalnosti postojećem softveru znači dodavanje novog koda i proširenje repozitorijuma izvornog koda. Da bi prevazišle ove probleme, većina organizacija, nezavisno od veličine, postepeno rekonstruišu svoje monolitne aplikacije u male, nezavisne komponente koje se nazivaju mikroservisi.

Ne postoji jedinstvena definicija mikroservisa, odnosno mikroservisne arhitekture, već postoji opšteprihvaćeno stanovište koje je vremenom evoluiralo u industriji prema kojem je mikroservis jedinica za razvoj softvera [10]. On predstavlja artifakt savršene veličine za kontinualnu isporuku u produkciju. Zbog njegove jednostavnosti i veličine, moguće je vrlo lako izvršiti validaciju, odnosno utvrditi da li određeni mikroservis pravilno izvršava funkcionalnost koju implementira, kako bi se osigurao ispravan rad. Sa stanovišta mikroservisa postoji uvjerenje da ovi aspekti pružaju brz, praktičan i efikasan način za stvaranje poslovne vrijednosti pomoću softvera.

Mikroservisna arhitektura je jedan od pristupa za razvoj softvera, koji podrazumijeva razvoj jedne aplikacije kao skupa mikroservisa koji se izvršavaju nezavisno i komuniciraju upotrebom jednostavnih mehanizama, kao što je HTTP protokol, preko odgovarajućeg programskog interfejsa [9]. Svaki od mikroservisa implementira jedan jednostavan dio poslovne logike i može se isporučiti u produkciju nezavisno od drugih mikroservisa, pri tome koristeći potpuno automatizovan proces. Mikroservisi mogu biti implementirani upotrebom različitih programskih jezika i koristiti različite tehnologije za skladištenje podataka, pri čemu se njima upravlja centralizovano.

Kako su mikroservisi razdvojeni i nezavisni jedni od drugih, mogu se razvijati, isporučivati i skalirati individualno, bez bitnog uticaja na ostatak sistema. Ovim se omogućava izmjena pojedinih komponenata sistema nezavisno, ali i veoma brzo i u skladu sa promjenama korisničkih zahtjeva. Svaki mikroservis izvršava se kao nezavisan proces i komunicira sa drugim servisima preko jednostavnog i precizno definisanog interfejsa, kao što je prikazano na slici 2.1.

Protokoli koji se koriste su jednostavni, većina programera ih dobro razumije, a istovremeno nisu zavisni od ili ograničeni na određeni programski jezik. Kako je svaki mikroservis nezavisan, može biti napisan u programskom jeziku koji je najprikladniji za implementaciju tačno tog mikroservisa. Budući da se komunikacija ostvaruje uglavnom preko API-ja, moguće je razviti, implementirati i testirati svaki mikroservis zasebno. Promjena jednog od njih ne zahtijeva promjene ili rekonstruisanje bilo kojeg drugog mikroservisa, pod uslovom da se API ne mijenja ili mijenja samo na način da se ostvaruje kompatibilnost unazad.

Iako mikroservisna arhitektura rješava određene probleme, istraživanja pokazuju da sa porastom broja mikroservisa dolazi do pojave drugih izazova koji se prvenstveno odnose na upotrebu različitih programskih jezika i biblioteka, testiranje mikroservisa, kao i skaliranje odnosno orkestraciju istih [8]. Sam prelazak na novu arhitekturu povlači sa sobom niz promjena. Tu se prvenstveno misli na okruženje u kom će se aplikacija izvršavati, način isporuke novih verzija, nadgledanje izvršavanja aplikacije (eng. *monitoring*) kao i izmjene postojeće prakse. Međutim, najvažnije pitanje ovdje je u kakvom okruženju će se izvršavati mikroservisne aplikacije. Kao što je već pomenuto u prethodnom tekstu, različite softverske komponente, odnosno mikroservisi, prilikom izvršavanja često zahtijevaju različite, ponekad i konfliktne, verzije biblioteka ili imaju specifične zahtjeve u pogledu konfiguracije okruženja. Konfiguracija okruženja koja, između ostalog, podrazumijeva i instaliranje biblioteka, kontrolisanje njihovih konfiguracija, kao i ažuriranje istih, često može biti veoma komplikovana čak i upotrebom specijalizovanih alata za ovu namjenu kao što su *Ansible* [11], *Puppet* [12], *Terraform* [13] ili *SaltStack* [14].

Ukoliko posmatramo sa istorijskog stanovišta, prva opcija bi bila da se koriste fizički serveri. Ako bi se više mikroservisnih aplikacija izvršavalo u ovakvom okruženju bilo bi jako teško ograničiti dostupne resurse što bi uzrokovalo određene probleme sa raspodjelom resursa.

To bi moglo dovesti do situacija u kojima bi jedna aplikacija zauzimala većinu resursa, a kao rezultat toga, druge aplikacije bi imale probleme sa performansama. Korišćenje jednog servera za mikroservis takođe nema smisla, jer hardverski resursi ne bi bili potpuno iskorišćeni.

Kada se aplikacija sastoji od više komponenata, pri čemu svaka treba zasebno okruženje, sasvim je logično da se koristi virtuelizacija radi boljeg iskorišćenja serverskih resursa i da se svakoj komponenata dodijeli jedna ili više virtuelnih mašina. U tom slučaju, može se na svakoj virtuelnoj mašini nezavisno instalirati odgovarajući operativni sistem, biblioteke te odgovarajuća konfiguracija. Kako vremenom broj komponenata sistema raste, raste i broj virtuelnih mašina a ukoliko se uzme u obzir da za svaku od tih komponenata obično treba obezbijediti nekoliko izolovanih okruženja, jasno se vidi da ovaj pristup ima ozbiljna ograničenja kada je u pitanju pokretanje i izvršavanje mikroservisa.

Uzimajući u obzir da mikroservisi implementiraju jednostavne i potpuno izolovane funkcionalnosti, pri čemu svaki od njih treba zasebno okruženje, kao logičan izbor nameće se upotreba virtuelnih mašina koje obezbjeđuju emulaciju kompletnog sistema odnosno virtuelizacije. Kako virtuelizacija obuhvata veoma širok i ne čvrsto definisan pojam, ukratko ćemo dati kratak uvod i pregled osnovnih tipova virtuelizacije, što će proizvesti zaključak koji tip virtuelizacije predstavlja optimalan izbor za primjenu u mikroservisnom okruženju.

2.1. Virtuelizacija

Za potrebe ovog rada virtuelizaciju možemo posmatrati kao izvršavanje virtuelne instance računarskog sistema u sloju koji je apstrahovan od stvarnog hardvera. Najčešće se odnosi na istovremeno pokretanje više operativnih sistema na jednom računarskom sistemu [15]. Ove virtuelne instance se nazivaju virtuelne mašine (VM) pri čemu svaka VM izvršava svoj operativni sistem i ponaša se poput nezavisne mašine, iako se izvodi u okviru dijeljenog realnog hardvera.

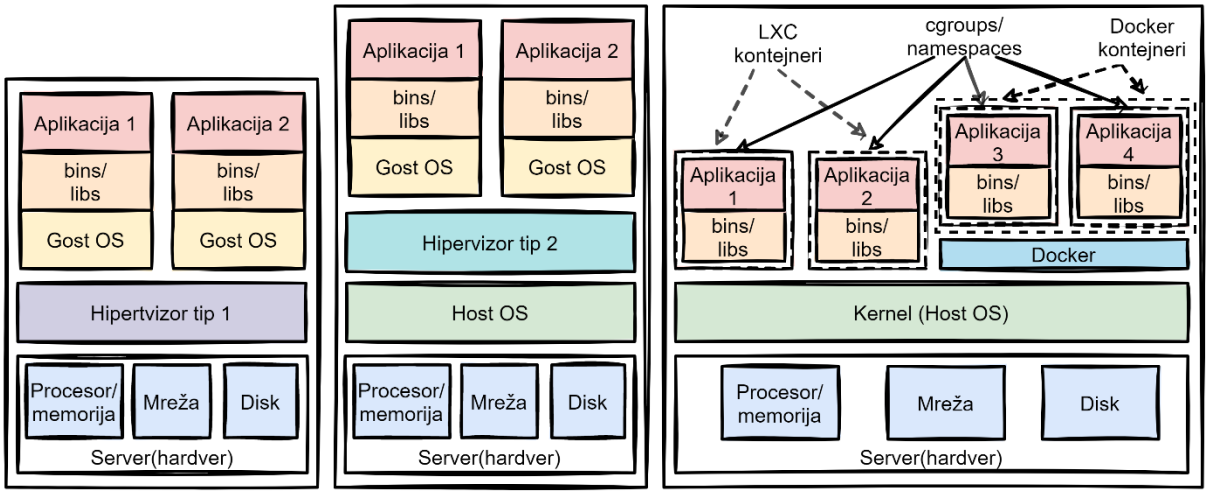
Pojam virtuelizacije pominje se prvi put 1960. godine, kao metoda logičke podjele sistemskih resursa velikih centralnih računara (eng. *mainframe computers*) između različitih aplikacija, ali je od tada značenje pojma značajno prošireno [10]. Prve teoretske osnove i uslove dovoljne da jedna računarska arhitektura efikasno podržava virtuelizaciju računarskog sistema postavili su Popek i Goldberg 1974. godine [16]. Sistemske virtuelne mašine sposobne su za virtuelizaciju čitavog skupa hardverskih resursa, uključujući procesor, memoriju te resurse za skladištenje podataka, kao i periferne uređaje. VMM, koji se takođe naziva i hipervizor (eng. *hypervisor*), je dio softvera koji obezbjeđuje apstrakciju virtuelne mašine [16]. VMM izoluje operativni sistem i resurse hipervizora od virtuelnih mašina i omogućava kreiranje i upravljanje tim VM-ovima [17]. Osnovna uloga mu je da bude interfejs između hardvera i virtuelnih mašina obezbjeđujući da svaka ima pristup neophodnim fizičkim resursima, ali istovremeno tačno onoliko koliko joj je potrebno odnosno dodijeljeno. Prema istraživanju Popeka i Goldberga, tri svojstva su od interesa za analizu okruženja stvorenog VMM-om:

- ekvivalentnost – program koji se izvršava u okviru VM treba da se ponaša na identičan način kao da se izvršava na stvarnoj mašini;
- kontrola resursa – VMM mora imati potpunu kontrolu na virtuelizovanim resursima;
- efikasnost – statistički značajna većina mašinskih instrukcija mora se izvršiti bez intervencije VMM-a.

Popek i Goldberg su u istraživanju kreirali i podjelu hipervizora, koji se prema njima dijele u dvije osnovne grupe. Hipervizori tipa 1 su oni koji se izvršavaju direktno na hardveru i služe samo da obezbijede virtuelnim mašinama okruženje za izvršavanje. U modernim okvirima u ovaj tip se obično ubrajaju *VMWare vSphere* [18], *Microsoft Hyper-V* [19] i *Oracle VM Server* [20]. S druge strane, hipervizori tipa 2 su hipervizori koji se izvršavaju pod već postojećim operativnim sistemom, uporedo sa drugim aplikacijama i servisima. Neki od predstavnika su *VirtualBox* [21], *VMWare Player* [22], *QEMU* [23] itd.

Hipervizor tip 1 se naziva još i *Standalone* VMM i izvršava se direktno na fizičkom hardveru "ispod" svih operativnih sistema (koji su "gosti" u virtuelnim mašinama). S obzirom na to da se on izvršava direktno na hardveru i da mu nije potreban dodatni operativni sistem, ovaj tip hipervizora smatra se najefikasnijim. Osim što su efikasni, oni su takođe i vrlo sigurni, jer je osnovni operativni sistem eliminisan i nema nedostataka i ranjivosti koji su tipični za klasične operativne sisteme ili su oni u značajnoj mjeri eliminisani. Obično se koriste kao serverska rješenja, jer je podrška u vidu drajvera usko specijalizovana i orijentisana prema određenim proizvodima vodećih proizvođača hardvera.

Hipervizori tipa 2 se još nazivaju i *Hosted,* jer se izvršavaju u okviru postojećeg operativnog sistema koji tada ima ulogu "domaćina", odnosno hosta. U ovom slučaju hipervizor se oslanja na operativni sistem da poduzme odgovarajuće operacije kao što je upravljanje resursima (procesor, memorija, disk itd.) čime se omogućava podrška za širok spektar hardvera. Međutim, određeni dio resursa je neminovno dodijeljen i osnovnom operativnom sistemu pa te resurse ne možemo da koristimo za potrebe virtuelnih mašina. Oni koriste hardversku podršku za virtuelizaciju ukoliko je dostupna, u suprotnom se prebacuju na softversku emulaciju i dinamičko rekompajliranje, koje je izvodivo bez obzira na hardversku podršku.



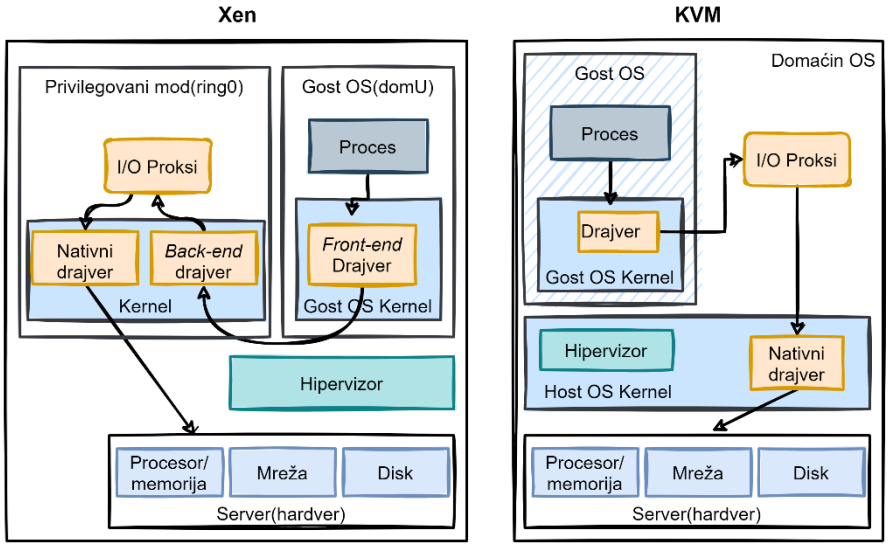
Slika 2.2. Uporedni prikaz hipervizora tip 1, tip 2 i virtuelizacije, sa virtuelizacijom na nivou operativnog sistema odnosno Docker-om

2.1.1. Paravirtuelizacija

Paravirtuelizacija (PV) je način virtuelizacije kod kojeg se gostujući operativni sistem modifikuje prije nego što se instalira unutar virtuelne mašine, kako bi se omogućilo svim gostujućim operativnim sistemima da uspješno sarađuju i dijele resurse, radije nego da se emulira kompletno hardversko okruženje [24]. Glavno ograničenje PV je činjenica da gostujući OS mora biti prethodno prilagođen za izvršavanje na VMM, ali sa druge strane eliminiše

potrebu za upotrebom privilegovanih instrukcija koje mogu da utiču na performanse sistema koji koriste potpunu virtuelizaciju. U ovom slučaju OS je svjestan da se ne izvršava na hardveru, pa će komunicirati sa hipervizorom koji će za njega izvršiti privilegovane instrukcije.

Komunikacija sa hipervizorom se ostvaruje preko interfejsa koji su obično pozivi API-ja koji je implementiran na strani hipervizora. Na ovaj način je posao hipervizora bitno pojednostavljen i moguće je ostvariti odlične performanse. Pri upotrebi paravirtuelizacije nije potrebna i ne koristi se hardverska podrška za virtuelizaciju, tako da je upotreba ovog pristupa moguća i na računarima koji je ne podržavaju. Najpoznatiji hipervizori predstavnici paravirtuelizacije su KVM (*Kernel-based Virtual Machine*) [25] i Xen [26]. Xen koristi veći broj procesorskih prstenova, jer se Xen izvršava u nultom prstenu, kernel VM-a u prvom, a aplikacije u drugom prstenu. U slučajevima kada nije moguće izvršiti modifikaciju kernela, pribjegava se upotrebi nemodifikovanog kernela OS (mora postojati i hardverska podrška za virtuelizaciju) i namjenski napravljenih drajvera za virtuelne (emulirane) periferije. Ovo je često dobar kompromis, jer virtuelizacija procesora i memorije nije toliko problematična, dok to nije slučaj sa perifernim jedinicama. Važno je napomenuti da ovaj pristup nije vezan za Xen i predstavlja standardan način rada za gotovo sve hipervizore.



Slika 2.3. Uporedni prikaz arhitekture Xen i KVM hipervizora

KVM način virtuelizacije zasnovan je na *Linux* operativnom sistemu i klasifikuje se kao hipervizor tipa 1 (pretvara *Linux* kernel u hipervizor). U ovom slučaju hipervizor je sastavni dio kernela, tačnije kernel modul. Istovremeno operativni sistem je potpuno funkcionalan i omogućava korisniku da pokreće druge aplikacije uporedo sa virtuelnim mašinama, što ga klasifikuje kao hipervizor tipa 2. KVM zahtijeva postojanje hardverski podržane virtuelizacije i trenutno podržava x86-64, S/390, PowerPC, IA64 i ARM ISA. Osim potpune virtuelizacije, za određene operativne sisteme postoji i podrška za paravirtuelizaciju u vidu drajvera za mrežnu i grafičku karticu, disk kontroler itd. Takođe, ovdje nije potrebno vršiti modifikaciju drajvera pa upotreba nativnih drajvera obezbjeđuje nešto bolje performanse, jer sama I/O putanja ima manje koraka nego kod Xen-a.

2.1.2. Virtuelizacija na nivou operativnog sistema

Virtuelizacija na nivou operativnog sistema je koncept koji za cilj ima efikasno iskorištenje hardverskih resursa, a zasniva se na ideji da kernel omogućava postojanje više izolovanih instanci grupa procesa (eng. *shared kernel virtualization*). U zavisnosti od toga koja

se konkretna tehnologija koristi (LXC [27], *Docker* [28], *Solaris* [29], *OpenVZ* [30], BSD *jail* [31] itd.) oni se nazivaju kontejneri (eng. *containers*), zone (eng. *zones*), virtuelni privatni serveri (eng. *virtual private servers*), virtuelna okruženja (eng. *virtual environments*) ili zatvori (eng. *jails*).

Kako je ovdje kernel koji se izvršava i na hostu i u virtuelnoj mašini isti i ujedno je i hipervizor, dobijaju se najbolje performanse i minimalni gubici. Ovaj pristup se često naziva i ekstremna paravirtuelizacija, jer je svaka funkcija kernela iz VM usklađena sa hipervizorom iz razloga što se koristi isti kernel. Samim tim se nameće ograničenje da operativni sistem u virtuelnoj mašini mora koristiti isti kernel iste verzije. Međutim, zahvaljujući načinu na koji je implementiran *Linux* kernel to ne znači neminovno i korištenje iste *Linux* distribucije, jer su svi slojevi iznad kernela, poput biblioteka i sistemskih servisa, nezavisni između hosta i virtuelne mašine. Upotreba istog kernela ne zahtijeva deduplikaciju operativnog sistema unutar gostujućeg OS-a, a iako koriste isti kernel svaka instanca je potpuno izolovana i nije svjesna postojanja ni hosta ni drugih instanci.

Ukoliko govorimo o *Linux* kontejnerima izolacija se zasniva na tri osnovne komponente: komandi *chroot*, kontrolnim grupama (eng. *cgroups)* i korisničkim prostorima (eng. *namespaces*). *Chroot* je komanda u *Linux*-u koja omogućava stvaranje (prividnog) fajl sistema za kontejner i promjenu korijenskog direktorijuma, odnosno za proces koji se izvršava i sve procese kojima je ovaj proces roditelj (eng. *parent*). Sa druge strane upotreba korisničkih prostora objedinjuje globalne sistemske resurse u sloj apstrakcije tako da svaki proces vidi samo resurse koji su mu dodijeljeni, čime se ostvaruje efikasna izolacija. Kontrolne grupe su još jedna od ugrađenih funkcionalnosti *Linux* kernela, koja omogućava kernelu da na efikasan i fleksibilan način upravlja i nadzire grupu procesa. Zahvaljujući navedenim tehnologijama moguće je vršiti dodjelu procesorskog vremena, memorije i slično, a moguće je i vršiti prioritizaciju resursa za različite procese.

Zahvaljujući svemu navedenom, moguće je na jednom serveru simultano pokrenuti zaista velik broj kontejnera, jer svaki kontejner praktično zauzima samo onoliko resursa koliko bi zauzimale aplikacije koje se u njemu izvršavaju. Dakle kontejner je u stvari ništa više nego grupa izolovanih procesa koji se izvršavaju direktno na operativnom sistemu domaćina (koji istovremeno upravlja njima) koristeći samo neophodne resurse za izvršavanje. Kako je navedena funkcionalnost i inače osnovni posao kernela operativnog sistema, očigledno je da se ovim pristupom ostvaruju minimalni gubici, a i količina koda koja je dodata na postojeći kôd u kernelu je minimalna. Iako manje koda obično znači da je i manja vjerovatnoća postojanja greške, naročito sigurnosnih propusta, važno je shvatiti da u ovom slučaju postojanje sigurnosnog propusta u kernelu kontejnera znači da isti sigurnosni propust postoji i u kernelu hosta, jer je riječ o istoj instanci kernela. Na osnovu svega navedenog kao logičan izbor tipa virtuelizacije u okviru mikroservisne arhitekture nameću se kontejnerske tehnologije.

2.2. Kontejnerske tehnologije

Umjesto da se za izolaciju okruženja za svaki mikroservis koriste klasične virtuelne mašine sa kompletnim operativnim sistemima i servisima, danas se, gotovo isključivo, koriste kontejnerske tehnologije. One omogućavaju da se više instanci efikasno izvršava na jednoj serverskoj mašini, pri čemu se koriste različita okruženja u potpunosti izolovana jedna od drugih, slično kao kod virtuelnih mašina, ali na mnogo jednostavniji način. U poređenju sa virtuelnim mašinama, kontejneri su mnogo jednostavniji i zahtijevaju značajno manje resursa, čime se omogućava efikasno izvršavanje velikog broja instanci istovremeno, dok sa virtuelnim

mašinama to nije slučaj. Kod virtuelnih mašina neophodan je dodatni skup procesa pa je samim tim potrebno i više resursa za njihovo izvršavanje. Osim toga, da bi se ostvarila virtuelizacija potreban je hipervizor odnosno softver koji omogućava virtuelizaciju odnosno upotrebu fizičkih resursa, kao i nadzor njihove upotrebe i raspodjelu između virtuelnih mašina.

Danas su kontejneri postali toliko popularni da u velikom broju okruženja predstavljaju podrazumijevani način izvršavanja, pa čak i isporuke, aplikacija ili aplikativnih komponenata, bez obzira na njihovu arhitekturu. Ukoliko posmatramo kontejnere u odnosu na način na koji se koriste odnosno izvršavaju, možemo da ih svrstamo u dvije grupe: kontejneri koji se izvršavaju na nivou operativnog sistema (OS) i aplikativni kontejneri [32].

Kada se govori o OS kontejnerima tu se obično misli na Linux kontejnere čiji način rada je već objašnjen u prethodnom tekstu. Oni su dizajnirani za pokretanje više procesa istovremeno i mogu se posmatrati kao jednostavniji oblik virtuelnih mašina. Iako dijele isti kernel sa domaćinom, moguće je instaliranje i konfigurisanje dodatnih aplikacija i servisa. Da bi se omogućilo ispravno upravljanje sa više procesa ovi kontejneri se izvršavaju kao odvojeni *init* procesi.

Danas se uglavnom u većini okruženja koriste i aplikativni kontejneri (eng. *application container)* koji se kreiraju iz aplikativne slike (eng. *application image*), a koja enkapsulira sve što jednoj aplikaciji treba za izvršavanje (fajlove, biblioteke itd.). Ovim se omogućava ponovljivost izvršavanja aplikacije, kao i prenosivost odnosno identično izvršavanje na različitim sistemima. Oni se zasnivaju na ideji da se unutar jednog kontejnera izvršava samo jedna aplikacija odnosno servis. Kako se izvršava samo jedna aplikacija, nije potrebno izvršavanje *init* procesa kao kod OS kontejnera, što ih čini još jednostavnijim i lakšim za skaliranje.

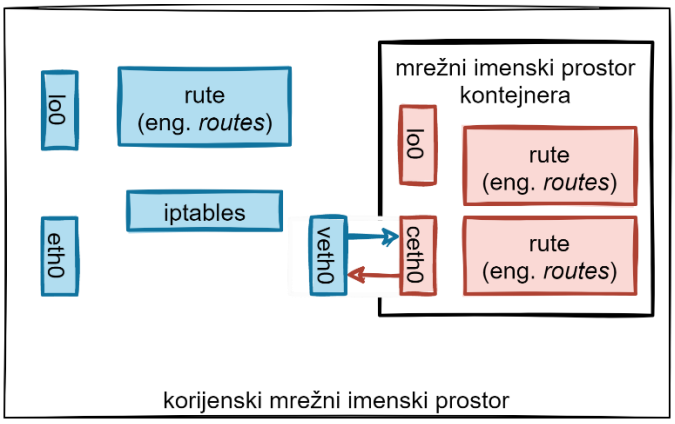
Aplikativni kontejneri se zasnivaju na principu nepromjenjivosti (eng. *immutability*) koji podrazumijeva da se jednom kreirana aplikativna slika tokom svog životnog ciklusa ne mijenja. Ukoliko je potrebno nešto promijeniti u tom slučaju se kreira nova slika. Ovakvi tipovi kontejnera su veoma pogodni za mikroservisne aplikacije.

2.2.1. Mrežna arhitektura

U prethodnom tekstu su opisane neke od karakteristika i funkcionalnosti *Linux* kernela, koje omogućavaju virtuelizaciju na nivou operativnog sistema, odnosno upotrebu kontejnera. Ukoliko posmatramo mrežnu arhitekturu kontejnera, ona je ništa više nego jednostavna kombinacija dobro poznatih *Linux* objekata:

- mrežni imenski prostori (eng. *network namespaces*) koji predstavljaju još jednu kopiju mrežnog steka sa svojim rutama, pravilima zaštitnog zida (eng. *firewall rules*) i mrežnim uređajima [33];
- virtuelni mrežni uređaji *veth* (eng. *virtual ethernet devices*) se mogu koristiti kao samostalni mrežni uređaji, ali prvenstveno im je uloga tunela između mrežnih imenskih prostora kako bi stvorili mrežni most od fizičkog mrežnog uređaja u drugom imenskom prostoru [34];
- virtuelni mrežni svičevi (eng. *bridge*) koji prosljeđuju datagrame između interfejsa koji su na njega povezani;
- rutiranje i translacija mrežnih adresa NAT (*Network Address Translation*).

U okviru korijenskog mrežnog prostora se za svaki mrežni interfejs, npr. *eth0*, kreira virtuelni mrežni interfejs *veth0* kao međusobno povezani par. Upotrebom mrežnih imenskih prostora se svakom kontejneru dodjeljuje po jedan interfejs u ovom slučaju *ceth0* koji komunicira direktno sa prethodno kreiranim *veth0* interfejsom.



Slika 2.4. Uopšteni prikaz osnovnih komponenata mrežne arhitekture kontejnera

2.2.2. *Docker*

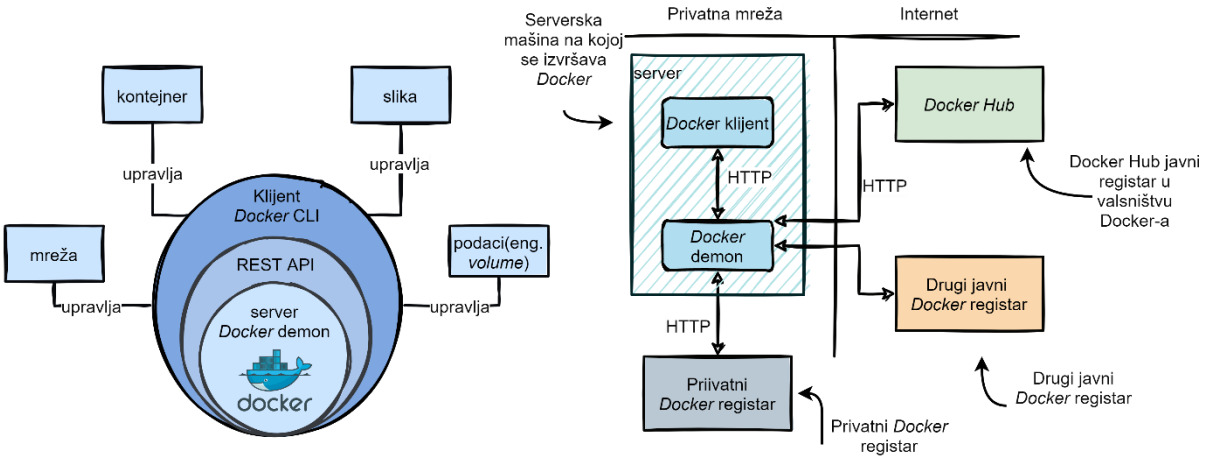
Iako se *Linux* kontejneri koriste već dugo, veliki porast popularnosti su doživjeli nakon što se pojavila *Docker* platforma [35]. Iako ona nije donijela ništa fundamentalno novo što već nije viđeno u svijetu *Linux* kontejnera, to je prva platforma koja je omogućila portabilnost kontejnera između različitih mašina na jednostavan i efikasan način. Osim toga, ona ne samo da pojednostavljuje proces kreiranja i pakovanja kontejnera, već je posebna pažnja posvećena dijelu koji se odnosi na različite biblioteke (može se posmatrati gotovo kao jedan vid statičkog kompajliranja) kao i cijeli fajl sistem operativnog sistema. Kao rezultat dobije se jedan jednostavan prenosiv paket koji se naziva *Docker* slika. Slika koju kreira *Docker* nije slika specifična samo za *Docker* platformu, već je to slika u OCI [36] formatu što znači da se iz ove slike može pokrenuti kontejner na bilo kojoj platformi koja podržava OCI format slika [37].

Docker je sastavni dio *Docker Engine* kontejnerske tehnologije odnosno platforme za kreiranje, razvoj, isporuku i izvršavanje aplikacija kao kontejnera. Koristi klijent-server arhitekturu i sastoji se od tri osnovne komponente:

- *Docker* demon – serverska komponenta koje se izvršava kao demon pod nazivom *Dockerd*;
- API – REST API interfejs koji aplikacije koriste za interakciju sa *Docker* demonom;
- CLI – klijentska komponenta koja se koristi za komunikaciju sa *Docker* demonom.

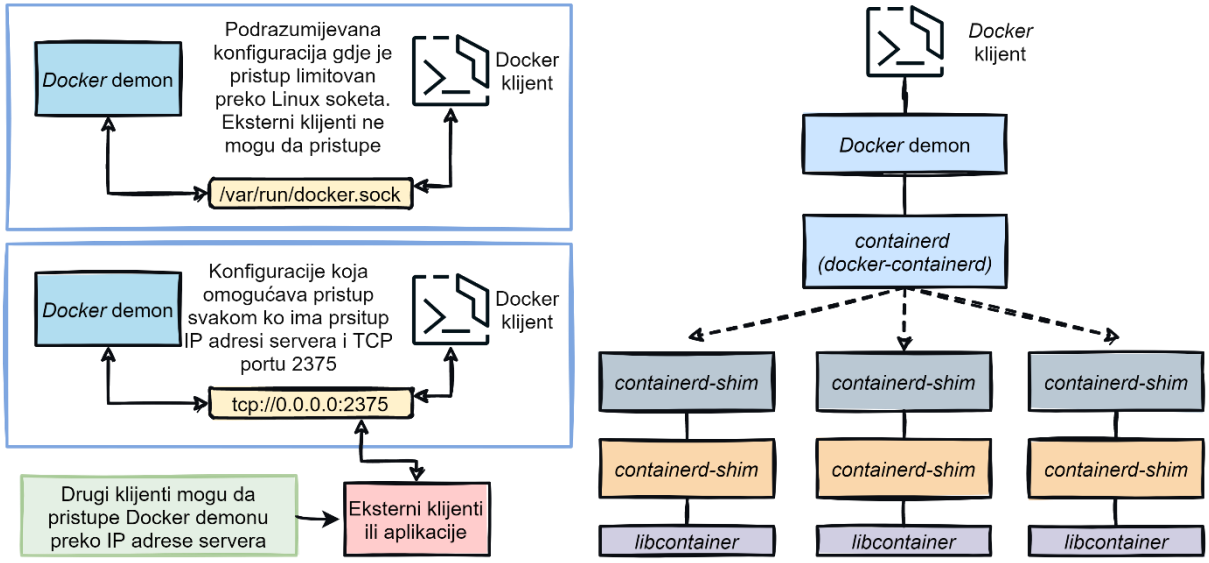
Docker klijent sadrži CLI interfejs koji koristi za komunikaciju sa *Docker* demonom i izdavanje komadi. Klijent može da se izvršava na istoj mašini kao i demon ili na da se poveže sa demonom na udaljenoj mašini, pri čemu može da komunicira i sa više demona. Prema podrazumijevanoj *Docker* konfiguraciji pristup demonu je ograničen na pristup samo sa lokalne mašine preko *Linux* soketa kao što je prikazano na slici 2.6. U ovom načinu pristupa korisnik koji izvršava *Docker* CLI aplikaciju treba da ima odgovarajuće permisije odnosno pristup fajlu na lokaciji */var/run/Docker.sock* u okviru fajl sistema. Ukoliko korisnik nije privilegovani (eng. *root)* korisnik, potrebno je da bude član korisničke grupe pod nazivom *Docker*.

Drugi način da bi se shvatio potencijalni rizik je da se pokretanje kontejnera posmatra kao mogućnost da se instalira bilo koji paket ili pokrene proizvoljan proces u okviru operativnog sistema. Postoji nekoliko načina da se *Docker* konfiguriše pravilno kako bi procesi koji se izvršavaju u privilegovanom modu mogli biti ograničeni. To se ostvaruje, na primjer, upotrebom mehanizma privilegija (eng. *capabilities*) [38] koje su sastavni dio kernela za upravljanje permisijama asociranih sa privilegovanom korisnikom (eng. *superuser*).



Slika 2.5. Arhitektura Docker engine platforme

Docker demon se oslanja na *containerd*, koji se izvršava takođe kao demon i koji koristi *runC* za upravljanje kontejnerima. Njemu se može pristupiti preko gRPC (*Google Remote Procedure Call*) protokola. Svi zadaci vezani za životni ciklus kontejnera kao što su kreiranje, pokretanje, zaustavljanje i brisanje kontejnera su delegirani *containerd* procesu preko *Docker* demona. Međutim, demon je odgovoran za upravljanje volumenima, mrežom i procesom kreiranja (eng. *build*) *Docker* slika.



Slika 2.6. Pristup Docker demonu preko različitih tipova soketa i način komunikacije Docker demona sa containerd odnosno runC

Kao što je prikazano na slici 2.6. *shim* pokreće *runC* nakon čega *runC* startuje kontejner, pri čemu *shim* postaje roditelj (eng. *parent*) novokreiranog kontejnera. Imamo samo jedan *containerd* proces pokrenut na sistemu, međutim postoji više *shim* procesa i to za svaki kontejner po jedan. Kao jednostavna CLI aplikacija, *runC* koristi *libcontainer* biblioteku čime

se izbjegava potreba da ovaj zahtjev izvršava *Docker* demon, što je bio slučaj u prvoj verziji *Docker*-a. Kako *containerd* upravlja kontejnerima sa ograničenim skupom instrukcija, a samim izvršavanjem kontejnera kao i svih drugih procesa upravlja kernel, moguće je izvršiti zaustavljanje ili ponovno pokretanje demon procesa bez ikakvog uticaja na kontejnere koji se izvršavaju.

Proces kreiranja i pokretanja jednog kontejnera se sastoji od nekoliko koraka:

- Za pokretanje kontejnera koristi se *Docker* CLI tako što se izvrši komanda *Docker run.*
- Nakon toga klijent pošalje POST zahtjev na API za kreiranje novog kontejnera.
- *Docker* demon nema načina da kreira kontejner. Kako je sva logika unutar *containerd* i *runC* on šalje zahtjev *containerd* procesu preko gRPC API-ja (ili *Linux* soketa) da startuje kontejner.
- Međutim, *containerd* takođe ne može da kreira kontejner već mora da se obrati *shim*-u.
- *Containerd* startuje *shim* proces za svaki kontejner (u slučaju da ih ima više) nakon čega *shim* startuje *runC* proces, koji potom startuje kontejner čime se izvršavanje *runC*-a okončava.

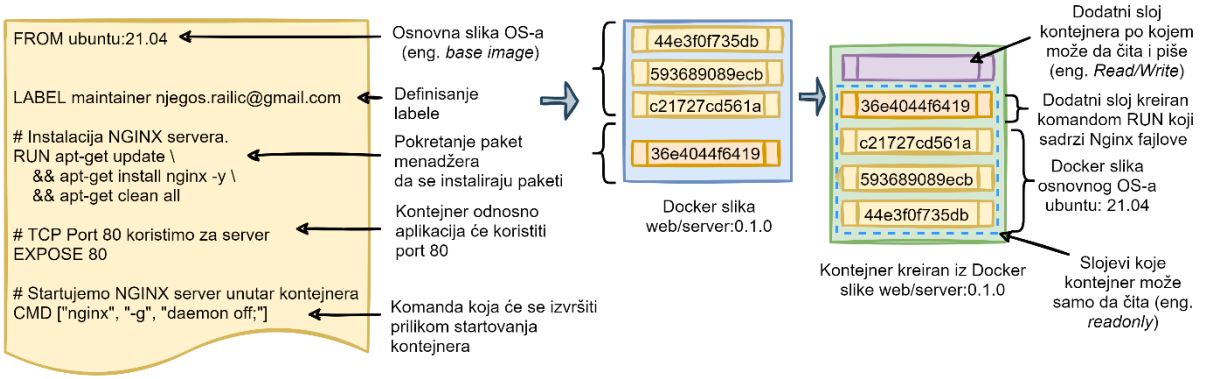
Da bi se jedan kontejner kreirao i startovao potrebna je odgovarajuća *Docker* slika. Servis za skladištenje i distribuciju verzionisanih *Docker* slika naziva se registar. Organizovan je u repozitorijume, pri čemu su u svakom repozitorijumu smještene sve postojeće verzije određenih *Docker* slika. Registar omogućava korisnicima da preuzimaju slike lokalno kao i da smještaju slike u registar pod uslovom da imaju odgovarajuće permisije za pristup. Podrazumijevano, *Docker* demon je podešen da komunicira sa *Docker Hub* [39] registrom koji je javno dostupan. Ovaj registar je u vlasništvu kompanije *Docker Inc.* i ima desetke hiljada slika spremnih za preuzimanje. Kao što se može vidjeti sa slike 2.5. jedan demon može da komunicira sa više registara istovremeno.

Registri mogu da budu javni i privatni u zavisnosti od toga da li se nalaze u privatnoj mreži ili su javno dostupni na internetu. Da bi se registri koristili sa *Docker*-om moraju da implementiraju odgovarajući API kao i da budu dodati u konfiguraciju demona. Postoji više implementacija ovog servisa, kako otvorenog koda, tako i komercijalnih, a najpoznatiji su *JFrog Artifactory* [40], *Nexus* [41], *Harbor* [42], pri čemu svi vodeći provajderi usluga u oblaku imaju i svoje implementacije kao što su GCR (*Google Container Registry*) [43], ACR (*Azure Container Registry*) [44] itd.

Da bi se automatizovao proces kreiranja *Docker* slika, a samim tim i obezbijedila ponovljivost procesa, te da bi on uvijek bio izvršen na isti način koristi se *Dockerfile* [45]. *Dockerfile* je tekstualni konfiguracioni fajl koji se sastoji od instrukcija za kreiranje slike. Upotrebom *Docker* klijenta moguće je izdati demonu *Docker build* naredbu da kreira sliku iz određenog *Dockerfile*-a. *Dockerfile* sadrži instrukcije u vidu komandi kao što su FROM, LABEL, RUN, COPY, CMD, ENTRYPOINT, itd. Prva komanda u fajlu je uvijek komanda FROM koja specifikuje koja se osnovna *Docker* slika koristi. To je obično slika sa minimalnim OS-om, u našem slučaju *Ubuntu linux* verzija 21.04 kao što je prikazano na slici 2.7.

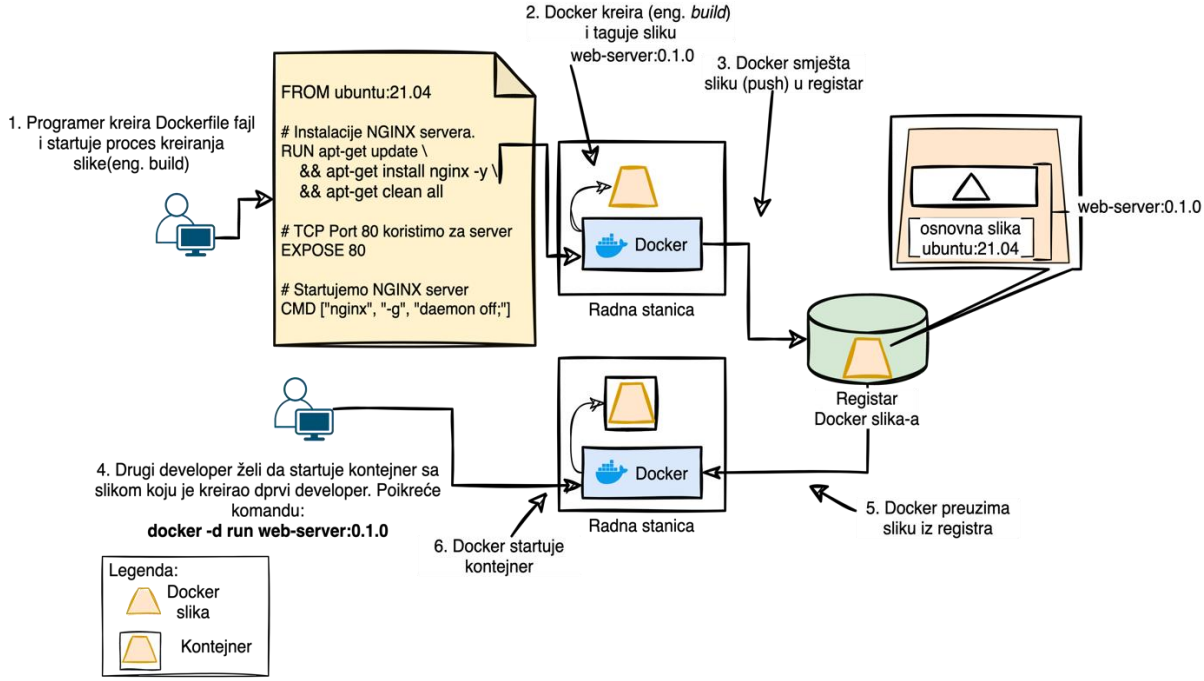
Sljedeća komanda u našem primjeru je komanda LABEL koja samo setuje odgovarajuću labelu, a nakon toga slijedi komanda RUN. Komanda RUN izvršava komande u okviru izabrane osnovne slike i instalira odgovarajuće pakete. Iz ovoga se zaključuje da je osim poznavanja sintakse *Dockerfile*-a, za kreiranje *Docker* slike neophodno poznavati i osnove

Linux operativnog sistema kao i paket menadžera. Nakon toga slijedi komanda EXPOSE koja definiše da će servis koji se bude izvršavao unutar kontejnera izložiti preko TCP porta 80, odnosno učiniti ga dostupnim za eksterne klijente. Posljednja komanda CMD označava komandu koja će se izvršiti prilikom startovanja kontejnera. Ova komanda je obično startovanje procesa koji će se izvršavati kao demon. U konkretnom primjeru to je startovanje *Nginx* aplikativnog servera.



Slika 2.7. Izgled Dockerfile-a za jednostavnu NodeJS aplikaciju

Docker koristi kompleksni (slojeviti) pristup za fajl sistem kao i za *Docker* slike. Time se rješavaju problemi koji nastaju kada se on koristi u velikim okruženjima. Na primjer, ukoliko bi se startovalo stotinu kontejnera, lokalni disk servera na kojem se izvršavaju kontejneri bi se vjerovatno brzo popunio. *Docker* koristi *copy-on-write* mehanizam kako bi se redukovala količina potrebnog prostora kao što je prikazano na slici 2.7.



Slika 2.8. Prikaz komponenata Docker platforme i njihove međusobne interakcije

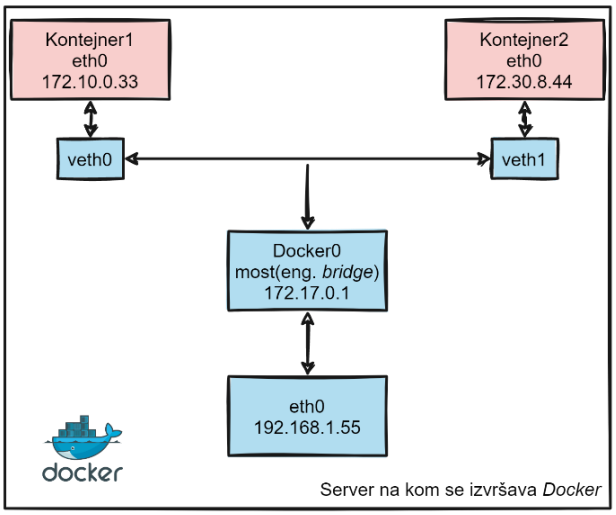
Prilikom kreiranja novog kontejnera iz *Docker* slike, umjesto kopiranja svih fajlova, kopiraju se samo izmjene koje se čuvaju u odvojenom sloju. Jedna osnovna slika može se koristiti za više kontejnera. Ovaj slojeviti pristup osim uštede prostora omogućava mnogo bolje performanse jer kontejneri mogu da se startuju veoma brzo direktno iz slike, pri čemu nije potrebno vršiti kopiranje fajlova. Svaki put kada se kreira novi kontejner iz *Docker* slike, kreira

se samo dodatni sloj za čuvanje izmjena na koji se može pisati (eng. *read-write*), dok su ostali slojevi iz slike samo za čitanje (eng. *read only*). Kao rezultat fajl sistem koji vidi kontejner je spoj osnovne slike i niza izmjena. *Docker* je veoma modularna platforma pa tako nudi više drajvera za pristup fajl sistemu, kao što su *overlay2* ili *device mapper*, koji omogućavaju upotrebu prethodno pomenutog slojevitog pristupa.

Kao što se može vidjeti sa slike 2.8. u procesu kreiranja polazi se od osnovne ili bazne slike (u konkretnom primjeru ubuntu:21.04), obično sa minimalnim operativnim sistemom, na koji se zatim dodaju potrebne komponente – aplikativni kod, konfiguracioni fajlovi, programske biblioteke i drugi fajlovi (u konkretnom primjeru instalira *Nginx* server). Budući da se od kontejnera očekuje da budu brzi i lagani (eng. *lightweight*), *Docker* slike su obično male. Kreirane slike se mogu publikovati u centralni registar po izboru, a uz ugrađenu integraciju sa javnim *Docker Hub* registrom, korisnike praktično samo jedna *Docker run* komanda dijeli od preuzimanja i pokretanja npr. zvanične distribucije izabranog veb servera ili servera baza podataka (u konkretnom primjeru web-server:0.1.0).

2.2.2.1. *Docker* mrežna arhitektura

Docker podržava više mrežnih arhitektura kao što su *overlay* i *macvlan*, međutim, njegovo podrazumijevano mrežno rješenje zasniva se na privatnom umrežavanju sa domaćinom (eng. *host-private networking*) implementirano pomoću drajvera mrežnog mosta [46]. To znači da je podrazumijevano umrežavanje zasnovano na privatnom adresnom prostoru, koji mora da se bude transliran pomoću NAT-a ili proksi servera ako mreža treba da se poveže sa drugom mrežom van servera, kao što je, na primjer, internet.



Slika 2.9. Primjer umrežavanja dva Docker kontejnera upotrebom drajvera mrežnog mosta

Slijedeći ovaj model (primjer na slici 2.9), *Docker* drajver implementira sljedeće funkcionalnosti:

- *Docker* kreira virtuelni mrežni most *Docker0* i dodjeljuje mu adresu 172.17.0.1 iz podmreže (opseg privatnih IP adresa) rezervisane za taj most. Mrežni most je uređaj koji spaja više mreža ili mrežnih segmenata u jednu mrežu kako bi omogućio komunikaciju između njih. To omogućava povezivanje više virtuelnih mreža i kontejnera na istom serveru u jednu mrežu.
- Da bi se kontejneri povezali sa virtuelnom mrežom svakom kontejneru se upotrebom *Linux* mrežnih imenskih prostora dodjeljuje po jedan virtuelni mrežni interfejs *veth*

povezan sa mostom. Ovaj interfejs se preslikava na odgovarajući stvarni mrežni interfejs zadužen za upravljanje mrežnim uređajem (mrežnom karticom) odnosno vezom između samog servera i mreže na koju je povezan. Svakom kontejneru se dodjeljuje stvarni interfejs kao i po jedna IP adresa iz mrežnog opsega dodijeljenog virtuelnom mostu.

Kako oba kontejnera imaju dodijeljen po jedan interfejs koji je na istoj logičkoj mreži oni mogu da komuniciraju međusobno. Iz prethodnog se zaključuje da kontejneri mogu da komuniciraju međusobno samo ukoliko su povezani na isti mrežni most u okviru istog servera. Kontejneri na različitim mašinama ne mogu da komuniciraju međusobno koristeći IP adrese koje su im dodijeljene, jer teoretski oni mogu dobiti iste mrežne segmente i eventualno iste IP adrese iako su na različitim serverima. Da bi kontejneri na različitim serverima međusobno komunicirali, moraju im biti dodijeljeni portovi asocirani sa IP adresom servera, koji se zatim prosljeđuju (proksiraju) ili transliraju do kontejnera. To znači da kontejneri moraju da vode računa koje portove koriste kako bi se izbjegla kolizija, ili da im se dodjeljivanje portova vrši dinamički. Kontejnerima se dodjeljuju različiti mrežni interfejsi, a kako nema dijeljenog mrežnog interfejsa između njih to stvara djelimičnu izolaciju.

2.3. Orkestracija kontejnera

Kao što je već rečeno, kontejnerske tehnologije omogućavaju da se aplikacija upakuje i pokrene na bilo kojem računaru obezbjeđujući pri tom isto okruženje na različitim sistemima odnosno ponovljivost izvršavanja. Sve što korisniku treba dolazi u jednom paketu. Kontejneri omogućavaju i jednostavan rad sa višestrukim verzijama okruženja. Često je u praksi teško izvodivo na istom računaru imati instalirane različite verzije servera baze podataka, aplikativnih servera ili programskih jezika. Iako je neke od navedenih problema moguće riješiti kompleksnim podešavanjem sistema, pitanje je koliko realno takvi pristupi oslikavaju stanje u eksploataciji za koje i pravimo aplikaciju. Jedno od rješenja je upotreba virtuelnih mašina, ali to vodi do nezanemarivih gubitaka performansi i potrebnih dodatnih resursa. Kako upotreba kontejnera tipično unosi minimalnu degradaciju performansi, a omogućava postojanje paralelnih verzija i kompleksnih okruženja, očigledno je da njihova upotreba predstavlja adekvatno rješenje pomenutih problema. Još jedna prednost upotrebe kontejnera je mogućnost jednostavnog skaliranja, odnosno pokretanja većeg broja instanci kontejnera koji sadrže određenu aplikaciju. Kao i pri svakom povećanju broja komponenata koje je moguće koristiti na veliki broj načina, dolazi do povećanja kompleksnosti sistema, naročito u slučajevima kada je neophodno riješiti probleme međuzavisnosti između određenih kontejnera. Ovaj problem je moguće riješiti upotrebom sistema za orkestraciju.

Sistemi za orkestraciju kontejnera rješavaju probleme kreiranja, upravljanja, skaliranja, isporuke, balansiranja opterećenja, visoke dostupnosti, kao i mrežne infrastrukture unutar jednog mikroservisnog okruženja u kojem se koriste kontejneri.

Najpopularniji sistemi za orkestraciju kontejnera su *Kubernetes* [47], *Docker Swarm* [48] i *Apache Mesos* [49]. Međutim, istraživanja [50] pokazuju da više od 90% kompanija koriste *Kubernetes* za orkestraciju i upravljanje mikroservisnim aplikacijama. Na to oko 50% otpada na organizacije koje same instaliraju i konfigurišu *Kubernetes*, dok preostalih 40% organizacija koristi neka od rješenja u oblaku zasnovana na *Kubernetes* platformi, kao što su *Google GKE* [51], *Amazon EKS* [52] i *Microsoft AKS* [53].

Docker Swarm je rješenje za orkestraciju kontejnera koje je proizvod kompanije *Docker*. Kao takav je čvrsto integrisan sa *Docker* API i veoma pogodan za orkestraciju kontejnera u okruženju gdje već postoji *Docker* infrastruktura. Osnovni koncepti vezani za *Docker* su ovdje

naslijeđeni, pa samim tim ukoliko se želi koristiti, nisu potrebne velike izmjene na infrastrukturi. *Docker Swarm* klaster se sastoji od više čvorova na kojima je instaliran *Docker*, od kojih svaki, u zavisnosti od konfiguracije, može da bude upravljački ili radni čvor, ali i da izvršava obje uloge istovremeno. Za kreiranje i konfigurisanje aplikacija koristi se *Docker compose* [54] CLI. Upotrebom odgovarajuće sintakse, u jednom YAML fajlu moguće je konfigurisati više aplikacija, koje se potom, izvršavanjem jedne komande kreiraju i startuju.

Iako je veoma jednostavan za upotrebu i koristi isti API kao i *Docker* klijent, ima dosta ograničenja u odnosu na *Kubernetes*. Omogućava vrlo jednostavno kreiranje klastera i većina mogućnosti je ograničena mogućnostima samog *Docker*-a. Osim toga, nema podršku za automatsko skaliranje mikroservisnih aplikacija, ali to se može riješiti upotrebom nekih eksternih rješenja koja se mogu lako integrisati u ovaj ekosistem. Ima implementirane funkcionalnosti kao što su visoka dostupnost upotrebom više upravljačkih čvorova, napredna mrežna infrastruktura sa DNS servisom, automatska detekcija i otklanjanje grešaka (ponovno pokretanje kontejnera), kao i visoka sigurnost koja se oslanja na integrisani CA servis. Zbog ovih funkcionalnosti, kao i jednostavnosti instalacije, očigledno je da je on često prvi izbor kao platforma za orkestraciju kontejnera, u organizacijama male i srednje veličine.

Apache Marathon [55] je orkestrator mikroservisne arhitekture zasnovan na *Apache Mesos* platformi. *Apache Mesos* je platforma otvorenog koda kreirana od strane kompanije *Apache*, kao distribuirani kernel za dinamičko upravljanje resursima. Dizajniran je za velike sisteme i maksimalnu redundansu. *Mesos* ima podršku za veliki broj okruženja razvijenih od strane *Apache* fondacije, kao što su *Hadoop* [56] i *Kafka* [57]. Određeni tipovi aplikacija mogu da ostvare bolje performanse koristeći ovu podršku kao i njegove osobine za dijeljenje resursa. Može da izvršava kontejnerizovane aplikacije u kombinaciji sa *Marathon*-om ali i druge tipove aplikacija. Često se kao takav u kombinaciji sa *Chronos* [58] platformom koristi za zakazivanje i izvršavanje zadataka na *Hadoop* platformi. *Marathon* je radni okvir (često se naziva meta-frejmvork) koji može da izvršava različite aplikacije i druge radne okvire. Nudi sve što nudi većina drugih orkestratora kao što je autoskaliranje, provjera ispravnosti mikroservisnih aplikacija kao i automatsku registraciju servisa, odnosno njihovih adresa. Može da izvršava *Docker* kontejnere kao i *Mesos* specifične kontejnere, međutim zbog veoma kompleksne implementacije i znanja potrebnog da bi se jedan ovakav sistem konfigurisao i održavao, uglavnom se koristi u veoma velikim organizacijama.

2.4. *Kubernetes*

Kubernetes je prenosiva, proširiva platforma otvorenog koda za upravljanje i orkestraciju mikroservisnih aplikacija. Platforma je nastala kao nasljednik *Google*-ove platforme pod nazivom *Borg* [59], implementirajući najbolje ideje i iskustva stečena u radu sa kontejnerizovanim aplikacijama tokom dugog niza godina [47], [59]. *Kubernetes* kao distribuirani sistem, osim orkestracije i upravljanja, nudi mnogo drugih funkcionalnosti, a najvažnije su:

- puštanje u isporuku i izvršavanje kontejnerizovanih aplikacija,
- dinamičko skaliranje u skladu sa zahtjevima,
- transparentno dinamičko puštanje u isporuku novih i vraćanje na prethodne verzije,
- distribucija različitih vrsta kredencijala,
- DNS servis,

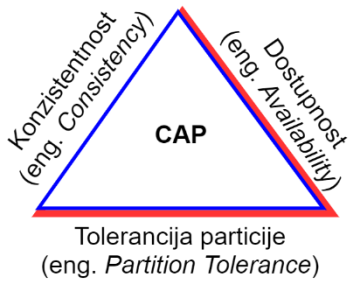
- automatska registracija i detekcija servisa, te
- balansiranje opterećenja.

Kubernetes se za upravljanje mikroservisnim aplikacijama oslanja na prethodno opisane karakteristike *Linux* kontejnera, što omogućava izvršavanje i upravljanje heterogenim aplikacijama bez potrebe poznavanja internih detalja same aplikacije. Sa aspekta *Kubernetes*-a svaka aplikacija je manje-više identična, jer je upakovana kao kontejnerska aplikacija i tako se i izvršava. Osim toga, upotreba kontejnera obezbjeđuje neophodnu izolaciju između aplikacija tokom njihovog životnog ciklusa. *Kubernetes* apstrahuje infrastrukturu odnosno resurse čime se pojednostavljuje razvoj, isporuka kao i upravljanje mikroservisnim aplikacijama. Samim tim broj servera na kojima se izvršavaju aplikacije nije bitan.

2.4.1. *Kubernetes* i CAP teorema

CAP teorema koju je definisao Eric Brewer, tvrdi da perzistentni distribuirani sistem može garantovati maksimalno dvije od sljedeće tri osobine [60]:

- konzistentnost – za svaki zahtjev za čitanje podataka garantuje se da će dobiti najnovije podatke ili neće uspjeti;
- dostupnost – mogućnost nastavka opsluživanja zahtjeva bez garancije da odgovor sadrži najnovije podatke;
- tolerancija particije – otpornost u slučaju pada ili pogoršanja mrežne komunikacije između čvorova.



Slika 2.10 CAP teorema

Prilikom projektovanja distribuiranog sistema nema tačnog odgovora na pitanje koje će dvije CAP osobine garantovati naš sistem. Na primjer, ukoliko projektujemo distribuiranu bazu podatka za jednu finansijsku instituciju uvijek ćemo uz toleranciju particije prednost dati konzistentnosti nego dostupnosti. U ovom slučaju cijena povrata netačnih podataka je jednostavno previsoka. Međutim, ukoliko želimo da projektujemo sistem za upravljanje visoko dostupnim aplikacijama, u tom slučaju ćemo izabrati uvijek dostupnost prije konzistentnosti. Razlog za ovu odluku se nameće sam po sebi. Ukoliko ovakav sistem zbog nekog razloga ne može da servisira korisničke zahtjeve izvršavajući posljednju verziju aplikacije, bolje je da je aplikacija dostupna i da opslužuje korisnike izvršavajući čak i stariju verziju aplikacije nego da je cijela aplikacija nedostupna.

Ukoliko to posmatramo sa aspekta *Kubernetes* platforme, koja je dizajnirana za orkestraciju visoko dostupnih aplikacija, ona garantuje dostupnost prije konzistentnosti. Kako nije moguće implementirati sve tri osobine iz CAP teoreme, umjesto konzistentnosti *Kubernetes* implementira model poznat kao konzistentnosti na kraju (eng. *eventual*

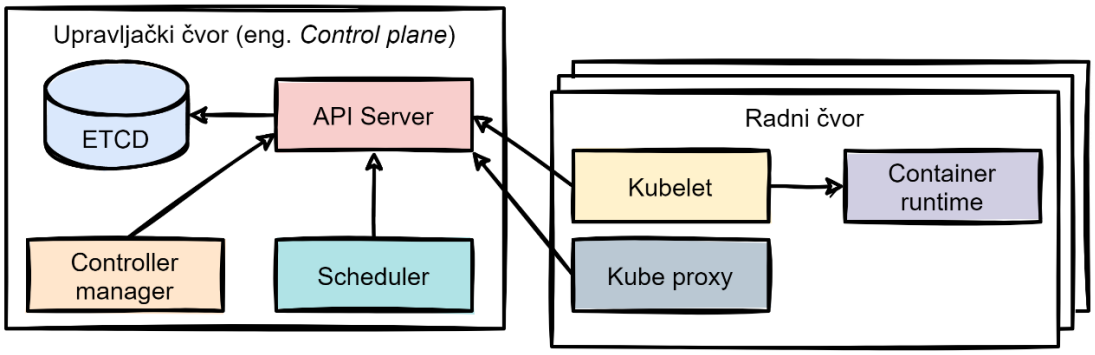
consistency). Ovaj model se koristi u distribuiranim računarskim sistemima za postizanje visoke dostupnosti. Prema definiciji, neformalno garantuje da ako se ne izvrši ažuriranje date stavke podatka, na kraju će svi pristupi toj stavci vratiti zadnju ažuriranu vrijednost [61]. Često se naziva i optimistička replikacija. Za sistem koji je postigao konzistentnost na kraju, kaže se da je konvergirao [62]. Iako ne postoji garancija koliko će to vremenski trajati, sve komponente sistema će na kraju odražavati zadnju ažuriranu vrijednost. Ono što je zagarantovano jeste da će *Kubernetes* platforma na kraju prevesti sistem iz trenutnog stanja (eng. *actual state*) u željeno stanje (eng. *desired state*).

Na primjer, pretpostavimo da se u okviru *Kubernetes* platforme izvršava jedna instanca mikroservisa. Ukoliko to promijenimo, odnosno deklarišemo da se izvršava pet kopija istovremeno, nakon nekog vremena sistem će iz trenutnog stanja konvergirati u željeno stanje. Vremenski interval da bi se ovo postiglo zavisi od više faktora, a ogleda se u aktivnostima koje komponente *Kubernetes* platforme treba da izvrše da bi se startovale četiri nove instance datog mikroservisa. Sa aspekta modela konzistentnosti na kraju, nije bitno koliko ovaj vremenski interval traje već sama garancija da će se konzistentnost postići. U nekim specifičnim situacijama, ukoliko se desi kvar na jednom fizičkom serveru ili nedostaje odgovarajućih resursa, ovaj vremenski interval može da traje i nekoliko dana.

2.4.2. Arhitektura *Kubernetes* sistema

Kubernetes klaster se sastoji od više čvorova, ali se oni mogu podijeliti u dvije osnovne grupe:

- upravljački čvorovi (eng. *master node*) na kojima su instalirane upravljačke komponente KCP (*Kubernetes Control Plane*) koje upravljaju cijelim *Kubernetes* sistemom;
- radni čvorovi (eng. *worker node*) na kojima se izvršavaju kontejneri, odnosno mikroservisne aplikacije.



Slika 2.11. Komponente Kubernetes sistema

Na slici 2.11. prikazan samo jedan upravljački čvor kako bi se na jednostavniji način prikazala arhitektura i objasnile osnovne komponente *Kubernetes* sistema. Međutim, u produkcionim okruženjima se gotovo uvijek koristi više instanci upravljačkih čvorova (obično tri ili pet) organizovanih u poseban HA klaster.

Kao što se može vidjeti sa slike na upravljačkom čvoru, odnosno čvorovima, se izvršavaju minimalno sljedeći servisi:

- ***API server*** je centralna tačka za komunikaciju. Sva komunikacija, kako između internih, tako i eksternih komponenata, može se ostvariti jedino preko ovog servisa. Pristup mu

se ostvaruje preko RESTful API servisa koristeći HTTPS protokol. On obrađuje i validira REST zahtjeve, te izvršava odgovarajuću poslovnu logiku.

- ***ETCD*** predstavlja osnovno skladište klastera i jedini je dio sistema koji je perzistentan i u kojem se čuva kompletna konfiguracija, trenutno i željeno stanje klastera. ETCD je definisan kao distribuirana, visoko dostupna ključ-vrijednost (eng. *key-value*) baza podataka za čuvanje najkritičnijih podataka distribuiranog sistema [63]. Zbog toga se u većim produkcionim sistemima ETCD servis izdvaja sa upravljačkih čvorova i konfiguriše kao zaseban HA klaster [64]. Komunikacija sa ETCD bazom podataka može da se ostvari samo preko API servera kao jedine komponente u okviru *Kubernetes* sistema koja komunicira sa ovom bazom podataka.
- ***Scheduler*** ima kao osnovnu ulogu da nadgleda API server, da detektuje pojavu novih zadataka za izvršavanje te da ih raspodijeli na neki od raspoloživih radnih čvorova. On radi na taj način što prvo filtrira čvorove nepodobne za izvršavanje zadataka, zatim koristeći kompleksan algoritam rangira pogodne čvorove. Algoritam uzima u obzir veliki broj parametara, a neki od njih su da li čvor ima dovoljno resursa, koliko se trenutno zadataka izvršava, da li postoje određene labele itd. *Scheduler* ne izvršava zadatke već je odgovoran samo za njihovo raspoređivanje i dodjeljivanje radnim čvorovima. Ukoliko *scheduler* ne može da pronađe pogodan čvor za izvršavanje zadatka, zadatak ne može biti raspoređen na izvršavanje i prelazi u stanje čekanja.
- ***Controller manager*** je zadužen za pokretanje svih kontrolnih servisa odgovornih za izvršavanje rutinskih zadataka unutar klastera. Često se kaže da je on "kontroler kontrolera" jer kreira nezavisne upravljačke petlje unutar kojih nadgleda rad ostalih kontrolera, među kojima su *Endpoint, Node* i *ReplicaSet*. Osnovna namjena kontroler servisa je da se izvršavaju u petlji (eng. *reconciliation loops*) i nadgledaju izmjene na API serveru i obezbijede konvergiranje trenutnog stanja klastera ka željenom stanju.

Na radnim čvorovima se izvršavaju svi kontejneri, odnosno korisničke mikroservisne aplikacije. Posmatrano sa višeg nivoa, njihova uloga se ogleda kroz izvršavanje sljedećih aktivnosti:

- ostvarivanje veze sa *Kubernetes* API serverom i čekanje na dodjelu zadataka;
- izvršavanje zadataka odnosno kontejnera;
- slanje izvještaja o trenutnom statusu kontejnera.

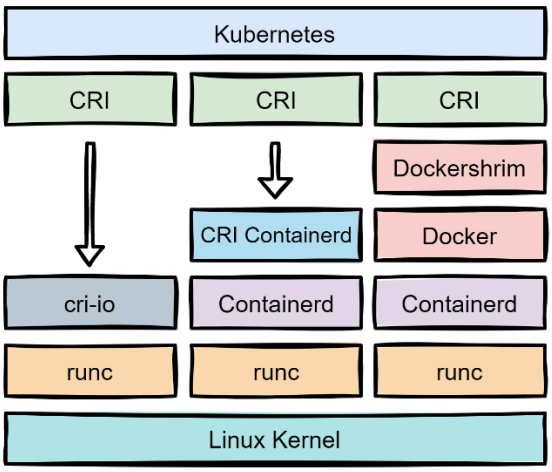
Komponente koje obezbjeđuju izvršavanje ovih aktivnosti se izvršavaju na svim radnim čvorovima:

- ***Kubelet*** je primarni agent odgovoran za komunikaciju sa API serverom i izvršavanje zadataka koji su dodijeljeni njegovom čvoru. Neke od osnovnih funkcionalnosti su da startuje kontejnere, provjerava ispravnost funkcionisanja (eng. *healthcheck*), *mount*-uje sekundarnu memoriju, upravlja kredencijalima i slično. Nakon što su zadaci završeni, *kubelet* šalje status nazad API serveru, koji na osnovu statusa odlučuje o sljedećim akcijama.
- ***Kube proxy*** je, kao što samo ime kaže, proksi servis koji se izvršava na radnim čvorovima, odgovoran za održavanje mrežnih pravila i rutiranje TCP i UDP paketa. Jedna od funkcionalnosti je da obezbjeđuje jedinstveno dodjeljivanje IP adresa servisima i implementira odgovarajuća mrežna pravila, tipično upotrebom *iptables* ili IPVS alata,

te da upravlja rutiranjem i balansiranjem saobraćaja ka servisima, odnosno kontejnerima koji se izvršavaju na tom radnom čvoru.

- ***Container runtime* (CR)** – *Kubelet* koristi *runtime* da bi izvršavao određene akcije vezane za kontejnere. To se prvenstveno odnosi na preuzimanje slika, pokretanje i zaustavljanje kontejnera itd. Na početku je *Kubernetes* podržavao samo *Docker*, dok danas ima podršku za više CR okruženja zahvaljujući modularnosti koja se ostvaruje preko CRI (*Container Runtime Interface*) interfejsa. Osim *Docker*-a, trenutno su najpopularniji *containerd* i *CRI-O* [65]. Kao što je već ranije objašnjeno, slika kreirana u OCI formatu može se koristiti za kreiranje kontejnera na bilo kojem od ovih *runtime*-a, što povećava fleksibilnost implementiranog rješenja i olakšava upotrebu u okviru drugih tehnologija u budućnosti.

Osim prethodno navedenih komponenata, svaki *Kubernetes* klaster ima i interni DNS servis kao jednu od vitalnih komponenata neophodnih za rad jednog ovakvog sistema. DNS servis je zasnovan na *CoreDNS* [66] serveru i koristi statičku IP adresu, što omogućava da svi kontejneri prilikom kreiranja automatski dobijaju odgovarajuću DNS konfiguraciju. Svaki novi servis koji se registruje, automatski se dodaje u internu DNS zonu i na taj način postaje dostupan drugim komponentama sistema.



Slika 2.12. Uporedni prikaz CR interfejsa i njihove međusobne komunikacije

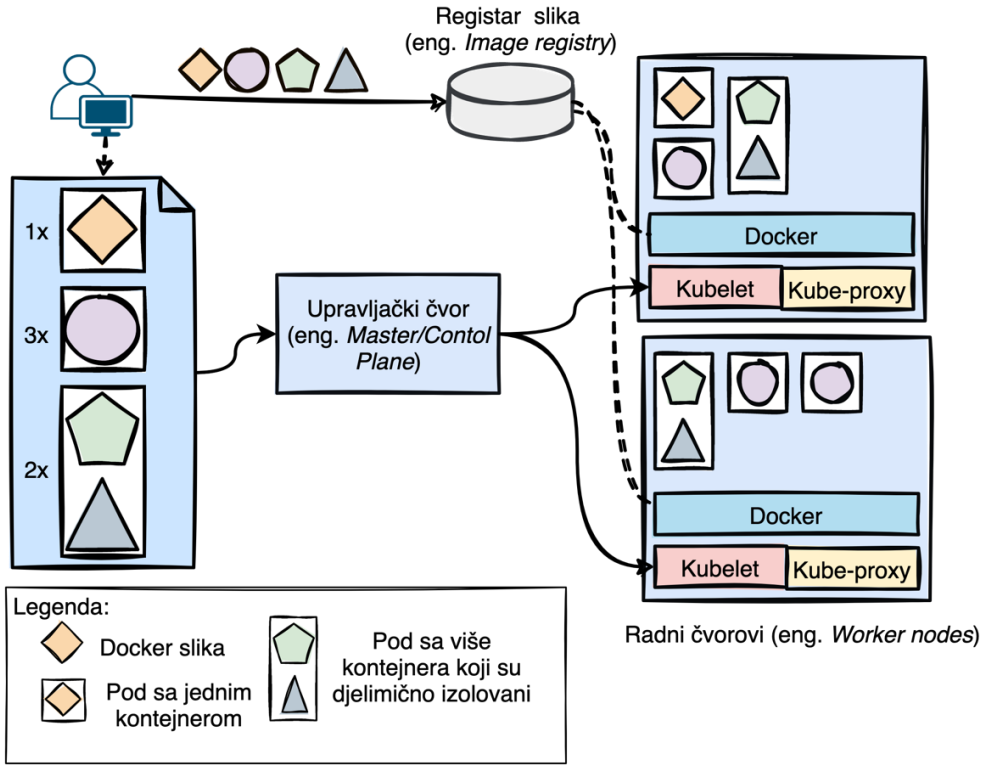
2.4.3. Osnovni koncepti

Kubernetes je izgrađen na nekoliko osnovnih koncepata koji se kombinuju i tako grade kompleksnije funkcionalnosti. U narednom tekstu je dat pregled osnovnih koncepata neophodnih da bi se razumjelo funkcionisanje platforme:

- ***Pod*** kao osnovni gradivni element predstavlja grupu od najmanje jednog kontejnera koji može da se isporuči odnosno kreira na *Kubernetes* sistemu. Umjesto da se kontejneri isporučuju, odnosno kreiraju individualno, uvijek se posmatra i operiše na nivou *Pod*-a. Kako su kontejneri u *Kubernetes* platformi dizajnirani da izvršavaju samo jedan proces, u slučaju da je potrebno izvršavanje više procesa, treba sami da implementiramo upravljanje procesima kao i upravljanje log fajlovima, što je kompleksno i ne predstavlja pogodan pristup [67]. Iz razloga što se ne preporučuje izvršavanje više nezavisnih procesa u jednom kontejneru kreiran je *Pod* kao konstrukcija višeg nivoa koja omogućava grupisanje kontejnera i upravljanje njima kao jednim smislenim objektom. Time se omogućava izvršavanje povezanih procesa u (gotovo) identičnom okruženju kao da se svi izvršavaju u istom kontejneru, ali uz izolaciju prema našim potrebama.

- ***Namespace*** ili imenski prostor je koncept koji omogućava izolaciju resursa odnosno kreiranje više virtuelnih klastera na jednom fizičkom klasteru, što može biti praktično ukoliko na jednom klasteru treba da postoji veći broj aplikacija i razvojnih timova. Ime virtuelnog klastera mora biti jedinstveno na nivou klastera. Njihova glavna namjena, osim izolacije, je ograničavanje dostupnih resursa i kreiranje mrežnih polisa prema potrebi. Prilikom kreiranja imenskog prostora treba imati na umu da nisu svi resursi definisani i dostupni na nivou imenskog prostora, već neki postoje i na nivou klastera, što znači da nije moguće implementirati potpunu izolaciju.
- ***Labels*** ili labele u formatu ključ/vrijednost koje se dodjeljuju objektima u klasteru, na primjer *Pod*-ovima. Osnovna namjena im je označavanje objekata koji su relevantni za korisnike, a mogu se koristiti za odabir i grupisanje objekata u kombinaciji sa imenom, koje je jedinstveno na nivou klastera za odgovarajući tip objekta. Labele je moguće kreirati uporedo sa objektom, ali isto tako ih je moguće i naknadno kreirati i dodijeliti objektu. Osim imena, svaki objekat u okviru klastera ima jedinstven UUID (*Universally unique identifier*) identifikator.
- ***Label Selector*** je mehanizam koji se koristi za identifikaciju skupova objekata na osnovu labela.
- ***Controller*** – Kao što je već rečeno, uloga kontrolera je da usklađuju trenutno stanje klastera sa željenim stanjem. Kontroleri se mogu uporediti sa kontrolnom petljom u robotici i automatizaciji, koja se neprestano izvršava i reguliše stanje sistema. U *Kubernetes*-u, kontroleri su kontrolne petlje koje nadgledaju stanje klastera, a zatim vrše ili zahtijevaju izmjene po potrebi. Svaki kontroler pokušava da trenutno stanje klastera približi željenom stanju. Kontroler prati stanje za najmanje jedan tip resursa u okviru *Kubernetes* platforme. Svaki tip resursa u okviru platforme ima polje koje sadrži specifikaciju (eng. *spec field*) koje predstavlja željeno stanje. Kontroler (kontroleri) za taj tip resursa su odgovorni za konvergenciju trenutnog stanja ka željenom stanju. Na primjer, *Replica Set* (RS) kontroler konstantno vrši nadzor na *Pod*-ovima koji se izvršavaju i obezbjeđuje da trenutni broj *Pod*-ova za odgovarajući tip resursa (određenu specifikaciju) odgovara željenom stanju. Specifikacija sadrži, na primjer, definiciju *Pod*-a uključujući definiciju kontejnera, resurse kao i broj replika. Ukoliko je specifikacijom definisano da se izvršavaju tri replike paralelno, RS kontroler će nastojati da sistem dovede u željeno stanje i osiguravati da se uvijek izvršavaju tri replike odnosno instance aplikacije. Kako bi znao koji *Pod*-ovi pripadaju kojoj specifikaciji, odnosno replici, koriste se labele i selektori labela. U okviru platforme postoji veliki broj kontrolera, a kako je *Kubernetes* proširiva platforma korisnici mogu kreirati dodatne kontrolere.
- ***Deployment*** resurs se koristi za kreiranje odnosno deklarativnu isporuku određenih verzija aplikacije, te između ostalog sadrži i specifikaciju odnosno šablon *Pod*-a sa listom kontejnera koji će se izvršavati unutar njega. Unutar samog resursa se deklariše željeno stanje *Pod*-a na osnovu čega DC (*Deployment Controller*) zajedno sa RS dovodi stanje klastera u željeno stanje. Takođe, upotreba ovog resursa omogućava isporuku novih verzija mikroservisne aplikacije bez prekida izvršavanja i bez povećanja rizika, tako što se nakon kreiranja nove verzije aplikacije korisnici preusmjeravaju na nju, nakon čega se stara verzija uklanja. Tokom ovog procesa *Kubernetes* provjerava ispravnost nove verzije (eng. *healthcheck*), tako da ukoliko nova verzija ne može pravilno biti pokrenuta ili generiše greške, pa ona neće proći validaciju ispravnosti, što znači da će korisnici i dalje koristiti staru verziju koja korektno funkcioniše.
- ***Volume*** predstavlja apstrakciju na nivou *Kubernetes*-a, koja omogućava kontejnerima da koriste različite vrste memorije, uključujući i perzistentnu memoriju. Tokom životnog

ciklusa kontejnera, fajl sistem je efemeran i vidljiv samo tom kontejneru. U slučaju da se kontejner restartuje, novi kontejner će biti kreiran iz iste *Docker* slike, ali u novom stanju, pri čemu će sve datoteke biti izgubljene. Takođe, problem nastaje i ukoliko kontejneri unutar jednog *Pod*-a treba da dijele podatke. Ovaj koncept rješava navedene probleme i omogućuje kontejnerima da veoma jednostavno koriste veliki broj memorija koje *Kubernetes* podržava. U zavisnosti kako je okruženje konfigurisano, kontejner može jednostavno da *mount*-uje volumen u okviru RAM memorije ili lokalnog diska radnog čvora na kom se izvršava, mrežnog diska i slično. *Volume* je dio *Pod*-a, tako da ne može da bude kreiran kao nezavisan objekt. Međutim, nakon što je kreiran, dostupan je svim kontejnerima unutar jednog *Pod*-a, ali svaki kontejner koji treba da mu pristupi mora ga *mount*-ovati zasebno, na bilo koju lokaciju u okviru svog fajl sistema (kontejnera).



Slika 2.13. Jednostavan pregled Kubernetes arhitekture i aplikacija koje se izvršavaju u okviru nje [35]

- ***Service*** ili servis je resurs koji se koristi za kreiranje jedne ulazne tačke za konekciju sa *Pod*-ovima unutar kojih se izvršava mikroservisna aplikacija. Svaki servis ima jedinstvenu IP adresu i port koji se ne mijenjaju za vrijeme životnog ciklusa servisa. Servis možemo predstaviti kao raspoređivač zahtjeva upućenih određenom mikroservisu na *Pod*-ove, unutar kojih se izvršavaju instance odnosno kopije mikroservisa. Kako je životni vijek *Pod*-a tipično kratak, a potrebno je ostvariti dugotrajniju dostupnost određenog servisa, naročito u slučaju horizontalnog skaliranja povećanjem broja instanci, upotreba servisa predstavlja rješenje za ovu vrstu problema. Često se u slučaju neke greške na samom *Pod*-u ili na čvoru na kom se izvršava taj *Pod*, automatski rekreira nova kopija istog *Pod*-a na drugom čvoru. Prilikom automatskog skaliranja tipično radimo sa više *Pod*-ova sa različitim IP adresama, a adrese *Pod*-ova se kreiraju tek nakon što se isti startuje, pa je samim tim jasno koliko bi teško bilo znati sve adrese, odnosno pravilno rutirati klijentske zahtjeve na odgovarajuća odredišta. Dodatno se za svaki servis automatski registruje po jedan *hostname* u okviru internog *Kubernetes* domena.

Da bismo objasnili kako se aplikacije isporučuju u okviru *Kubernetes* platforme koristićemo jednostavan primjer mikroservisne aplikacije koja se sastoji od tri mikroservisa prikazan na slici 2.13. Pretpostavimo da su unaprijed kreirane četiri *Docker* slike, od koje su tri mikroservisi koji implementiraju odgovarajuće funkcionalnosti, a četvrta kolektor log fajlova koji parsira logove aplikacije i šalje obrađene podatke na centralnu lokaciju.

Deskriptor aplikacije, kao što vidimo sa slike, sadrži tri *Pod*-a od kojih se dva sastoje od jednog kontejnera, dok treći sadrži dva. To znači da ova dva kontejnera unutar trećeg *Pod*-a nisu međusobno izolovani. Potrebno je naglasiti da se ovdje unutar svakog kontejnera i dalje izvršava samo jedna aplikacija odnosno mikroservis. Pomenuti deskriptor je fajl u YAML (*YAML Ain't Markup Language*) formatu koji sadrži deklarisane objekte neophodne da bi ova aplikacija bila uspješno instalirana u okviru *Kubernetes* platforme. Sa lijeve strane prikazan je broj replika odnosno koliko kopija svakog *Pod*-a će biti kreirano.

Nakon što programer koji ima odgovarajuća prava pristupa pošalje deskriptor *Kubernetes* master serveru preko API interfejsa, *Kubernetes* će kreirati odgovarajući broj *Pod*-ova na dostupnim radnim čvorovima. Nakon toga, *Kubelet* će poslati instrukcije *Docker*-u da startuje kontejnere, nakon čega će *Docker* preuzeti neophodne *Docker* slike i potom startovati kontejnere, čime će započeti proces izvršavanja mikroservisne aplikacije. Tokom cijelog životnog ciklusa aplikacije, *Kubernetes* će provjeravati deskriptor i osigurati da stvarno stanje objekata unutar klastera odgovara deskriptoru, odnosno željenom stanju. U slučaju da se desi problem sa nekim od kontejnera, *Kubernetes* će automatski pokušati ponovo pokrenuti kontejner i osigurati da se uvijek izvršava tačno određen broj kontejnera, odnosno replika.

2.4.4. Mrežna arhitektura

Kubernetes je dizajniran za izvršavanje distribuiranih sistema na klasteru serverskih mašina. Sama priroda distribuiranih sistema čini umrežavanje, odnosno način implementacije umrežavanja, najvažnijim dijelom. *Kubernetes* kao platforma forsira upotrebu odgovarajućih dodataka, odnosno priključaka (eng. *network plugin*) zasnovanih na CNI (*Container Network Interface*) [68] specifikaciji koja definiše način konfigurisanja mrežnih interfejsa na nivou kontejnera. Postoji više različitih implementacija, a najviše se koriste *Flannel* [69], *Project Calico* [70] i *Weave Net* [71].

Kubernetes nameće sljedeće fundamentalne zahtjeve za svaku implementaciju, pri čemu se zabranjuje bilo kakva politika segmentacije mreže:

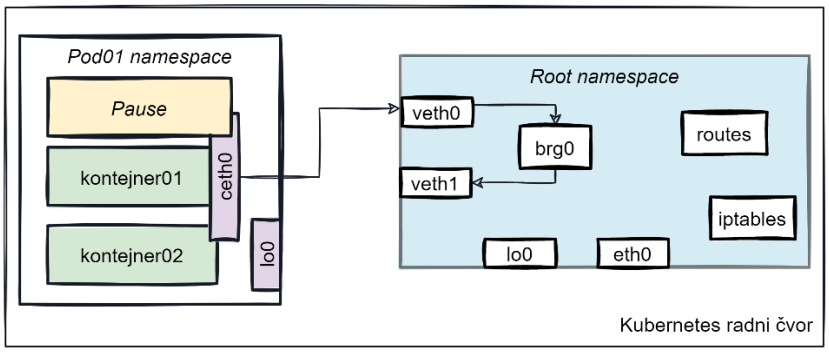
- svi *Pod*-ovi mogu da komuniciraju sa svim ostalim *Pod*-ovima bez upotrebe NAT-a (*Network Address Translation*);
- svi čvorovi mogu da komuniciraju sa svim *Pod*-ovima bez upotrebe NAT-a;
- IP adresa pod kojom jedan *Pod* vidi sebe je ista kao što ga drugi vide.

Kako bi se razumjela *Kubernetes* mrežna arhitektura, potrebno je razumjeti četiri osnovna mrežna scenarija:

- Kontejner-kontejner komunikacija (eng. *Container-to-Container networking*),
- *Pod-Pod* komunikacija (eng. *Pod-to-Pod networking*),
- Servis-*Pod* komunikacija (eng. *Pod-to-service networking*),
- Eksterna komunikacija.

2.4.4.1. Kontejner-Kontejner komunikacija

Za razliku od *Docker*-a, *Kubernetes* omogućava logičko grupisanje kontejnera u koncept pod nazivom *Pod*, omogućavajući kontejnerima da koriste jedan dijeljeni *veth* interfejs. Time se omogućava da oba kontejnera koriste istu IP adresu. Osim toga, kontejneri mogu da pristupaju jedni drugima preko dodijeljenih portova koristeći lokalni (eng. *localhost*) interfejs, što je isto kao pokretanje aplikacija na jednom serveru, sa dodatnim prednostima izolacije i dizajna usko povezane arhitekture kontejnera.



Slika 2.14. Komunikacija između dva kontejnera u okviru istog Pod-a

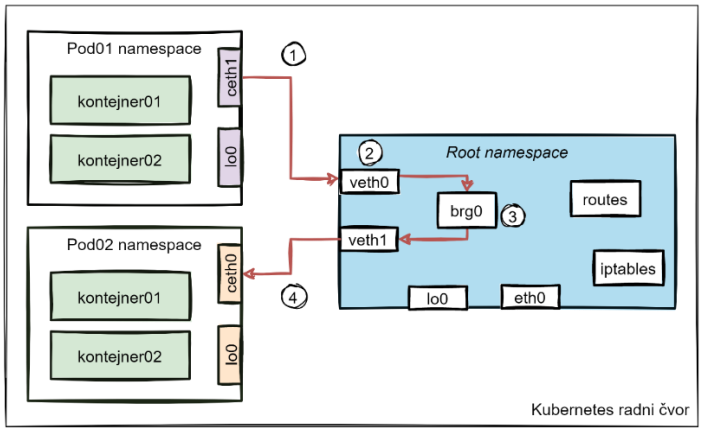
Da bi se primijenio ovaj koncept, *Kubernetes* kreira poseban kontejner za svaki *Pod*, koji obezbjeđuje mrežni interfejs za ostale kontejnere. Ovaj kontejner se naziva "*pause"* kontejner, jer se startuje komandom "*pause"* sa ciljem da obezbijedi mrežni imenski prostor za odgovarajući *Pod*. Glavna uloga ovog kontejnera je da obezbijedi IP adresu za *Pod* i konfiguriše mrežni prostor za sve kontejnere koji će biti kreirani u okviru datog *Pod*-a.

2.4.4.2. *Pod-Pod* komunikacija

Svakom *Pod* objektu prilikom kreiranja se dodjeljuje odgovarajuća IP adresa. Linux mrežni most (eng. *ethernet bridge*) je mrežna komponenta koja radi na drugom sloju OSI modela, sa primarnom ulogom da omogući transparentnu komunikaciju između dvije različite mreže.

Posmatrajmo sada kako se odvija komunikacija između dva *Pod*-a koja se nalaze na istom radnom čvoru prikazanu na slici 2.15:

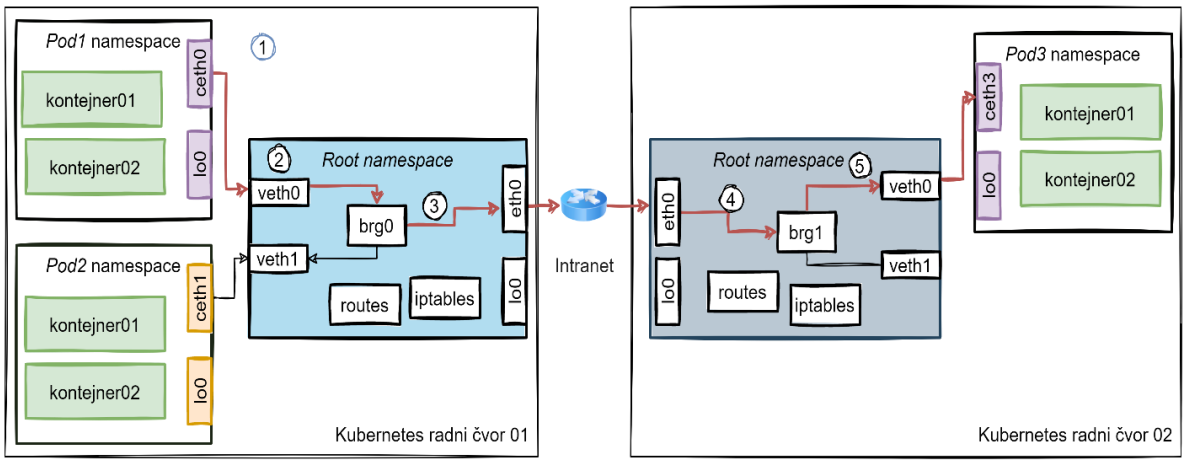
1. *Pod* sa nazivom *Pod1* šalje jedan paket na svoj interfejs *ceth0* u okviru svog mrežnog imenskog prostora.
2. Paket se dalje prosljeđuje na *veth0* interfejs u okviru korijenskog mrežnog imenskog prostora. Interfejs *ceth0* je povezan sa interfejsom *veth0*.
3. Mrežni most *brg01* koji ima *veth0* i *veth1* kao mrežne segmente, prosljeđuje paket na *veth1* interfejs. On koristi ARP protokol za razrješenje MAC adrese povezane sa IP adresom *Pod*-a sa nazivom *Pod2*, a kao rezultat dobije mrežni segment koji odgovara *veth1* interfejsu.
4. Paket se zatim prosljeđuje sa interfejsa *veth1* u okviru korijenskog mrežnog imenskog prostora, na *ceth1* interfejs u okviru *Pod02* mrežnog imenskog prostora.



Slika 2.15. Komunikacija između dva Pod-a koji se nalaze na istom radnom čvoru

Komunikacija između dva *Pod*-a koja se nalaze na različitim *Kubernetes* radnim čvorovima prikazana je na slici 2.16. a odvija se na sljedeći način:

1. *Pod* sa nazivom *Pod01* šalje jedan paket preko svog interfejsa *ceth1* koji se nalazi u okviru njegovog mrežnog imenskog prostora.
2. Paket se nakon toga prosljeđuje na interfejs *veth0* koji se nalazi u korijenskom mrežnom imenskom prostoru.
3. Mrežni most je odgovoran za održavanje tabele rutiranja između izvorne i odredišne destinacije. U ovom slučaju odredišna IP adresa nije član mrežnih segmenata u mostu pa razrješenje odredišta nije moguće upotrebom ARP protokola. Korijenski mrežni imenski prostor proslijediće paket na podrazumijevani mrežni prolaz putem svog *eth0* interfejsa. Kroz internu mrežnu infrastrukturu paket se prosljeđuje do interfejsa *eth0* na odredišnom radnom čvoru.
4. Paket se sa interfejsa *eth0* preusmjerava na mrežni most *brg0* u korijenskom mrežnom imenskom prostoru odredišnog radnog čvora.
5. Mrežni most *brg0* koristi ARP protokol da razrješenje MAC adrese koja je povezana sa adresom *Pod2* a kao rezultat dobija mrežni segment na *veth0* interfejsu, nakon čega prosljeđuje paket na taj interfejs.
6. Paket se prosljeđuje sa *veth0* interfejsa u okviru korijenskog na *ceth1* interfejs *Pod*-a *Pod03* imenskog prostora.

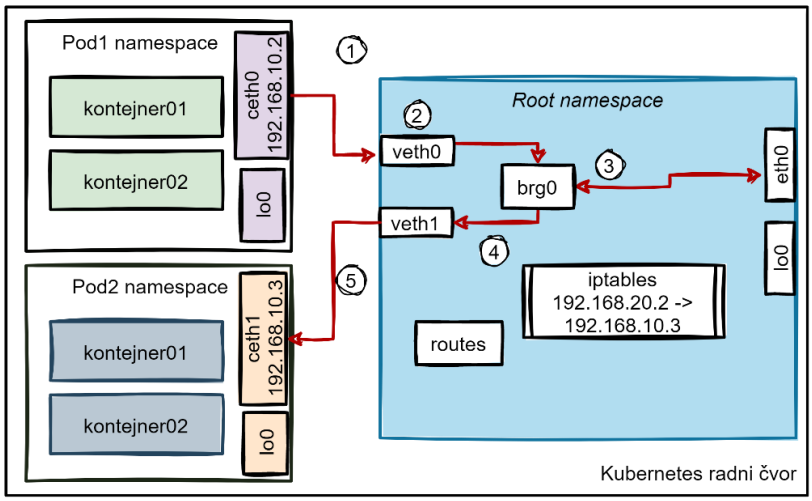


Slika 2.16. Komunikacija između dva Pod-a koji se nalaze na različitim radnim čvorovima

2.4.4.3. Komunikacija servis-*Pod*

Koncept servisa je već opisan u prethodnom tekstu i predstavlja apstraktan način da se mikroservis koji se izvršava na više *Pod*-ova učini dostupnim korisnicima preko mreže. *Pod*-ovi se mogu kreirati i obrisati u svakom trenutku kako bi se postiglo željeno stanje klastera pri čemu se njihove IP adrese stalno mijenjaju. Kako mikroservis implementira jednu jednostavnu funkcionalnost, on često može da bude pozivan od strane drugih mikroservisa. Praćenje ovih izmjena kao i rutiranje saobraćaja predstavlja jako kompleksne operacije u jednom ovakvom okruženju. *Kubernetes*, kao platforma za orkestraciju kontejnera, ima implementirane mehanizme za dodjeljivanje IP adresa servisima, registraciju DNS imena za servise kao i balansiranje zahtjeva koji dolaze preko odgovarajućeg porta servisa na grupu pozadinskih (eng. *backend*) *Pod*-ova.

Kubernetes vrši prosljeđivanje dolaznog saobraćaja na odgovarajuće pozadinske *Pod*-ove upotrebom selektora. Postoji i specifičan tip servisa koji ne koristi selektore, a primjena mu je na primjer, ako želimo saobraćaj sa jednog servisa proslijediti direktno na drugi servis. Svaki servis podrazumijevano koristi jedan port, a ukoliko jednom servisu želimo da dodijelimo više portova, onda svakom portu moramo da dodijelimo posebno ime kako bi bili jednoznačni.



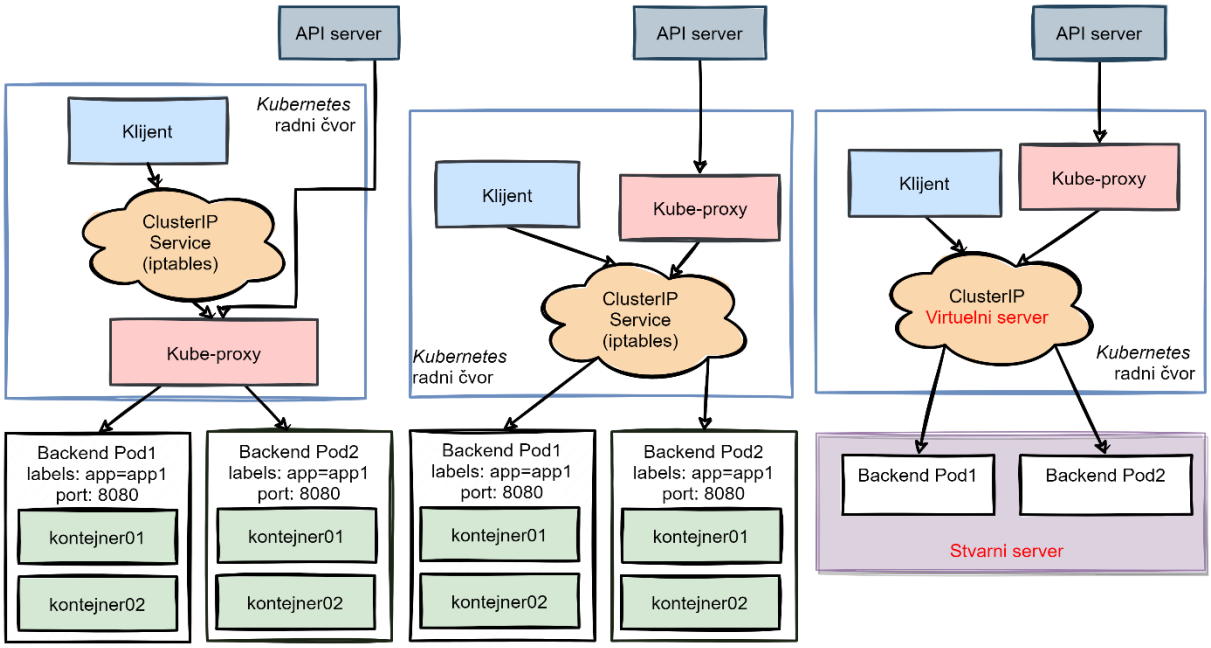
Slika 2.17. Komunikacija između Pod-a i servisa

Posmatrajmo sada dijagram na slici 2.17. i pretpostavimo da je servisu dodijeljena IP adresa 192.168.20.2. Servis koristi selektor i na osnovu labela prosljeđuje dolazne zahtjeve na *Pod2* kome je dodijeljena IP adresa 192.68.10.3. Za translaciju IP adrese servisa na adresu *Pod*-a zadužen je *kube-proxy* servis koji se, kao što je već objašnjeno, izvršava na svim *Kubernetes* čvorovima. U konkretnom primjeru na slici 2.17. to se ostvaruje upotrebom *iptables* programa. Ovaj proksi servis ima ulogu prosljeđivanja i raspoređivanja saobraćaja na odgovarajuće pozadinske (eng. *backend*) *Pod*-ove na kojima se izvršavaju mikroservisi.

Kao što je prikazano na slici 2.18. *kube-proxy* servis može da radi u jednom od tri proksi moda:

- *userspace* – u ovom načinu rada *kube-proxy* preko API servera prati promjene na KCP i u zavisnosti od zahtjeva ažurira servise i tačke konekcije (eng. *endpoints*). Kao što je objašnjeno svaki servis ima dodijeljen bar jedan port i jedinstvenu IP adresu. Za svaki port koji je dodijeljen servisu se rezerviše po jedan lokalni port na radnom čvoru.

Iptables vrši prosljeđivanje dolaznih konekcija sa IP adrese i porta servisa na odgovarajući lokalni port. *Kube-proxy* servis koji se izvršava u korisničkom prostoru (eng. *userspace*) sluša (eng. *listening*) na ovom portu i zadužen je za terminiranje dolaznih konekcija kao i prosljeđivanje zahtjeva na pozadinske *Pod*-ove. Prednost upotrebe ovog moda je što u slučaju da se desi terminacija pozadinskog *Pod*-a i prosljeđivanje zahtjeva ne uspije, *kube-proxy* može ponovo da pokuša i proslijedi konekciju na drugi *Pod*. Ovaj način rada se ne preporučuje za upotrebu, jer je zastario (eng. *deprecated*).



Slika 2.18. Način rada kube proxy servisa: userspace, iptables, IPVS [72]

- *iptables* – ovaj način rada je veoma sličan prethodnom, ali su *iptables* pravila konfigurisana na način da automatski prosljeđuju pakete direktno na pozadinske *Pod*-ove. Ovo je mnogo učinkovitiji način nego kada se zahtjevi rutiraju od kernela do *kube-proxy* servisa, a potom nazad do kernela što negativno utiče na performanse i rezultuje u dodatnom kašnjenju mrežnih paketa. Međutim, upotreba *iptables* takođe dodaje latenciju kako prilikom pristupa servisu tako i prilikom ažuriranja mrežnih pravila. Problem se ogleda u načinu na koji je *iptables* integrisan sa *netfilter* radnim okvirom unutar samog kernela. Vrijeme procesiranja paketa je u direktnoj vezi sa brojem mrežnih pravila, pa sa većim brojem pravila *iptables* treba više vremena za procesiranje.
- IPVS (*IP Virtual Server*) – u ovom načinu rada *kube-proxy* nadgleda servise i krajnje tačke, komunicira sa *netlink* interfejsom i konfiguriše odgovarajuća IPVS pravila. Ova kontrolna petlja obezbjeđuje željeno stanje održavajući sinhronizaciju između IPVS pravila i KCP-a. IPVS je zadužen za prosljeđivanje dolaznih konekcija na pozadinske krajnje tačke odnosno *Pod*-ove. IPVS proksi način rada se zasniva na integraciji sa *netfilter* radnim okvirom koji se izvršava u okviru kernela. IPVS radi na četvrtom sloju OSI modela pri čemu je prosljeđivanje dolaznih zahtjeva “<ServisIP>:<Port>” na odgovarajuće pozadinske *Pod*-ove potpuno transparentno. Za razliku od *iptables*-a, IPVS koristiti heš tabelu kao pozadinsku strukturu podataka za smještanje konfiguracije odnosno pravila rutiranja, pri čemu se ostvaruje mnogo brže rutiranje i ažuriranje pravila u odnosu na *iptables* [73].

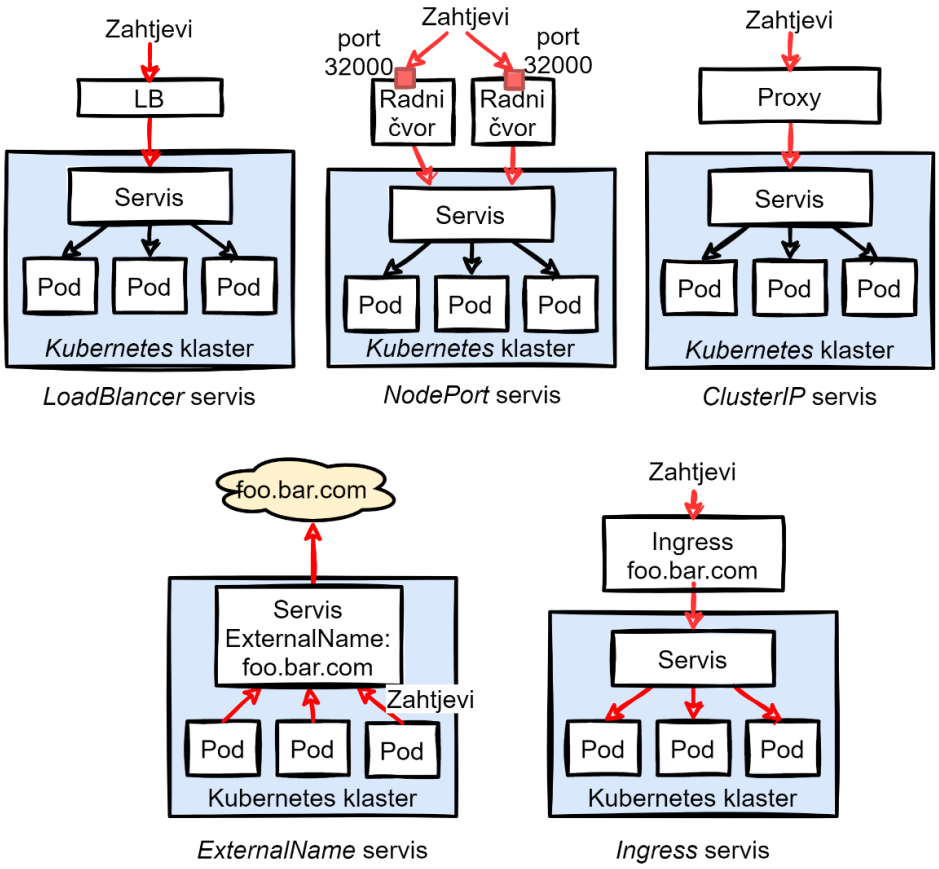
U odnosu na prethodno opisane načine rada IPVS obezbjeđuje mnogo veću mrežnu propusnost. Prednosti IPVS načina rada u odnosu na *iptables* su sljedeće:

- pruža bolju skalabilnost i performanse za velike klastere;
- podržava sofisticiranije algoritme za balansiranje opterećenja od *iptables*-a (eng. *least load, least connections, locality, weighted* itd.);
- podržava provjeru ispravnosti servera (eng. *server health*) i rekonektovanja (eng. *connection retry*) itd.

2.4.4.4. Eksterna komunikacija

Kako bi se ostvarila komunikacija sa drugim entitetima izvan *Kubernetes* plaforme, u okviru same platforme postoje četiri tipa servisa:

- ***ClusterIP*** – Ovaj tip servisa izlaže mikroservisnu aplikaciju preko interne IP adrese unutar klastera. Ovakav servis je dostupan samo unutar klastera preko *kube-proxy* servisa upotrebom jednog od prethodno opisanih načina rada.



Slika 2.19. Tipovi servisa u okviru Kubernetes platforme

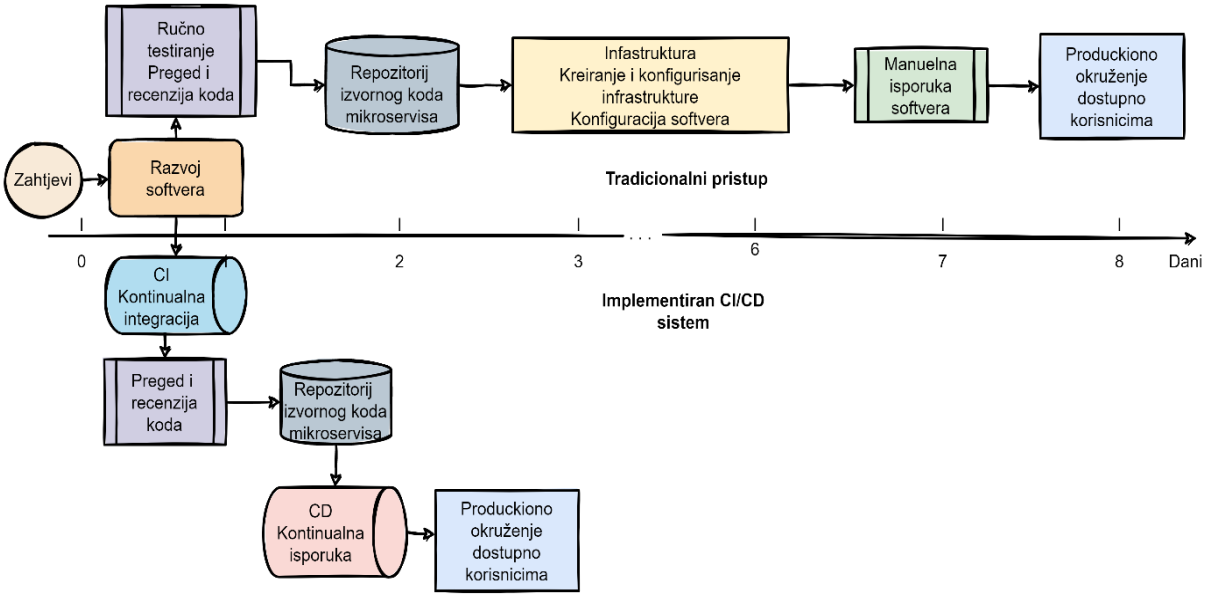
- ***NodePort*** – Ovaj tip servisa je najprimitivniji način za usmjeravanje vanjskog dolaznog saobraćaja na mikroservisnu aplikaciju. *NodePort* servis, kao što naziv govori, otvara određeni port na svim čvorovima (upravljačkim i radnim) i svaki zahtjev poslan na taj port se prosljeđuje na odgovarajuće pozadinske *Pod*-ove. Klijenti mogu da mu pristupe u formatu "<adresa čvora>:<*NodePort* port>". Dozvoljena je upotreba portova samo iz određenog opsega i obično se za ovaj tip servisa koristi balansiranje opterećenja zasnovano na DNS-u.

- ***LoadBalancer*** – Ovaj tip servisa omogućava da se mikroservisna aplikacija učini dostupnom eksterno koristeći infrastrukturu nekog od provajdera usluga u oblaku. Kada se kreira ovaj tip servisa dobija se jedna eksterna IP adresa preko koje se saobraćaj rutira na interni servis i okviru *Kubernetes* platforme.
- ***ExternalName*** – Koristi se za referenciranje eksternih servisa kao i servisa koji se nalaze u drugom virtuelnom klasteru. Za razliku od prethodno definisanih servisa, koji se koriste za usmjeravanje dolaznog saobraćaja u jedan *Kubernetes* klaster, ovaj tip servisa sa koristi za eksterni saobraćaj. Oslanja se na DNS infrastrukturu i omogućava da se jednom internom servisu u okviru *Kubernetes* platforme, pored internog dodijeli i eksterni DNS *hostname*. Način rada sličan je ostalim tipovima servisa s jedinom razlikom što se preusmjeravanje izvršava na nivou DNS servisa pa nije potrebna upotreba proksija.

Osim ova četiri osnovna tipa servisa, u praksi se često koristi pristup koji nije tip servisa, a naziva se *Ingress*. On se koristi kao pametni ruter i ulazna tačka za sve eksterne zahtjeve obično putem HTTP i HTTPS protokola. Sastoji se od dvije osnovne komponente: *Kubernetes* API objekta i samog *Ingress* servisa koji se izvršava kao zaseban *Pod* i ne dolazi instaliran sa ostalim *Kubernetes* komponentama. Postoji više implementacija *Ingress* kontrolera, a najpoznatija je ona bazirana na *Nginx* veb serveru. Osnovna uloga mu je da pojednostavi rutiranje kako bi se izbjeglo korištenje većeg broja drugih servisa. To se ostvaruje *Ingress* funkcionalnostima kao što su rutiranje bazirano na virtuelnim hostovima, rutiranje bazirano na URI putanjama (*Uniform Resource Identifier Path*), balansiranje saobraćaja kao i SSL terminacija na samom *Ingress* serveru.

3. LINIJA ZA UVOĐENJE U EKSPLOATACIJU

U tipičnom korporativnom okruženju razvojni tim se obično sastoji od više programera koji istovremeno rade na razvoju softvera. U zavisnosti od veličine organizacije, često imamo situacije da više razvojnih timova istovremeno radi na razvoju jednog softverskog proizvoda. Kako bi se omogućio istovremeni rad i integracija izmjena koje naprave, uglavnom se koristi neki od sistema za kontrolu verzija. Ovi sistemi omogućavaju praćenje istorije izmjena i upravljanje izvornim kodom, koji je smješten na nekoj centralnoj lokaciji, obično u vidu repozitorijuma izvornog koda. U takvom sistemu često se koristi neka od strategija grananja, gdje postoji jedna grana (eng. *branch*) koja je glavna (eng. *master*). Za svaku izmjenu, bez obzira na to da li je riječ o novoj funkcionalnosti ili ispravci neke uočene greške, obično se kreira dodatna grana (eng. *feature branch*) koja se nakon recenzije i odobrenja integriše u glavnu granu. Ukoliko su ciklusi isporuke softvera duži od nekoliko dana, integracija odnosno spajanje ovih izmjena u glavnu granu više nije tako trivijalan zadatak. Bez obzira na to što sistemi za kontrolu verzije omogućavaju lako praćenje istorije izmjena i otklanjanje eventualnih konflikata, sam proces integracije obično zahtijeva dosta vremena jer se on radi ručno.



Slika 3.1. Uporedni prikaz isporuke mikroservisne aplikacije sa i bez CI/CD procesa

Često postoji interferencija između različitih grana, što ponekad može rezultovati u nemogućnosti predviđanja vremena potrebnog za integraciju. U najgorim slučajevima dolazi do veoma kompleksnih konflikata koji se mogu riješiti samo prepisivanjem dijela koda. Pored dugog razvojnog ciklusa, ovo može da uzrokuje kašnjenje u fazi testiranja kao i isporuke, jer se neke greške mogu otkriti tek na samom kraju. Kako svaka aplikacija prije same isporuke u produkciono okruženje treba da bude isporučena u nekoliko drugih okruženja, kao što se može vidjeti na gornjem dijelu slike 3.1, za dodatne aktivnosti kao što su instaliranje i konfigurisanje infrastrukture troši se mnogo resursa. To je posebno specifično za monolitne aplikacije gdje je kompletan izvorni kôd za sve aplikacije unutar jednog repozitorijuma. Ovi problemi se mogu djelimično ublažiti ukoliko je kôd dobro strukturiran u module, ali kako aplikacija postaje sve složenija i dodaje se više funkcionalnosti, postupak isporuke aplikacije postaje sve sporiji i nefunkcionalan. To se može ispoljiti u nemogućnosti brze isporuke kritičnih ispravki ili u posebnim situacijama isporuke djelimično ili potpuno neispravne aplikacije korisnicima.

Kao što se može zaključiti, postoji više faktora odnosno izazova, koji utiču na sam proces isporuke softvera i o kojima treba voditi računa ukoliko se proces želi automatizovati. Kao odgovor na ove izazove, organizacije su se počele okretati mikroservisnoj arhitekturi. Razvoj malih odvojenih servisa, koji nisu čvrsto povezani, ima mnogo prednosti. Na primjer, oni mogu da se skaliraju nezavisno jedni od drugih, timovi mogu da rade nezavisno i koriste različite tehnologije i programske jezike. Međutim, istraživanja su pokazala da upotreba mikroservisa donosi nove izazove povezane sa komunikacijom između servisa, tehnološkom raznolikošću kao i mnogo kompleksnijim testiranjem [8]. Mnoge organizacije danas žele da automatizuju ove procese i implementiraju CI/CD sistem, ali iako postoji više metodologija i šablona za razvoj softvera, nema ih mnogo o tome kako ga automatizovati odnosno vršiti isporuku efektivno i pouzdano.

CI je praksa razvoja softvera koja za cilj ima da riješi neke od navedenih problema tako što se softver kontinualno integriše tokom razvoja. Pojam kontinualne integracije nastao je razvojem procesa ekstremnog programiranja Kenta Becka [74], kao jedne od metodologija u okviru njegove metodologije dvanaest faktora (eng. *12-Factor Application Methodology*). Primarni cilj korištenja CI je osigurati da se promjena može izvršiti na jednom mikroservisu nezavisno od ostalih mikroservisa u okviru jedne mikroservisne aplikacije. Dakle, cilj je održati sve učesnike u ovakvom sistemu sinhronizovane i osigurati da se novi kôd pravilno integriše sa postojećim kodom. Svaki put kad se dodaje novi kod, startuje se CI linija koja se sastoji od nekoliko koraka, većinom testova, koji osiguravaju da se softver može pravilno kreirati i isporučiti u različita okruženja. Ovo omogućava da se problemi identifikuju kako bi se razvojnim programerima pružile pravovremene povratne informacije. To obezbjeđuje da je kôd u svakom trenutku funkcionalan, jer se sa svakom promjenom problem odmah detektuje i rješava [1].

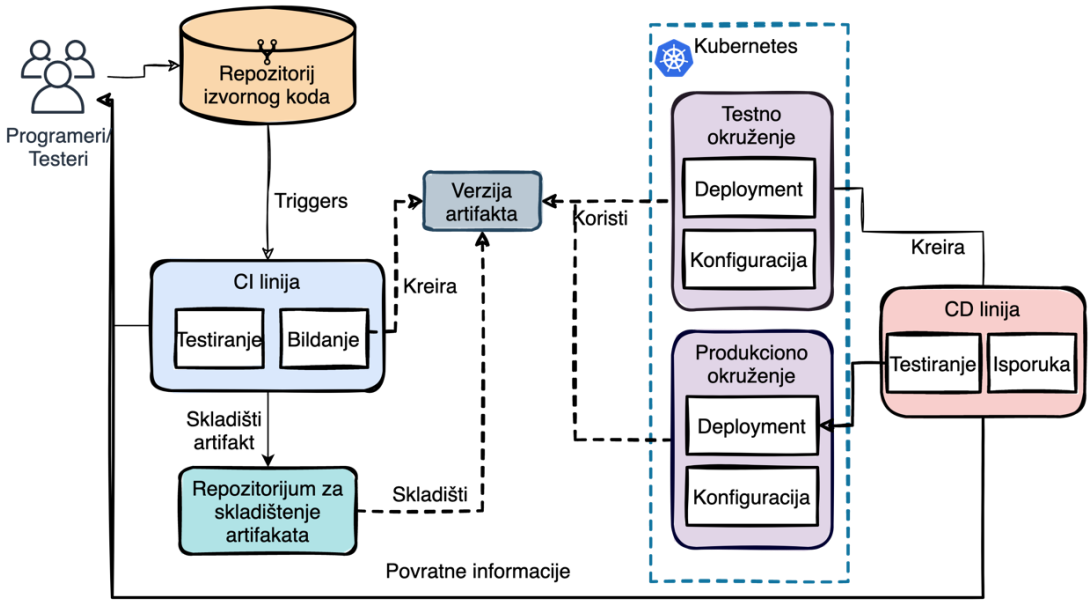
Nadovezujući se na kontinualnu integraciju, kontinualna isporuka (CDE) proširuje ovaj pristup na isporuku softvera. Slično fazi integracije, stavljanje isporuke na sam kraj razvoja softvera sadrži neke značajne nedostatke. Sa jedne strane korisnici moraju da čekaju neko vrijeme da bi dobili nove funkcionalnosti koje donose određene pogodnosti, iako su te funkcionalnosti već neko vrijeme spremne. Sa druge strane, razvojni programeri propuštaju priliku da dobiju vrijedne povratne informacije od korisnika. Baš kao što povratne informacije koje pružaju automatizovani testovi u procesu kontinualne integracije, pravovremene povratne informacije od korisnika omogućuju programerima da uoče greške čim se pojave i da ih isprave. Mogućnost redovne isporuke na pouzdan način, naravno, opet zahtijeva automatizaciju procesa čineći ga ponovljivim i daleko manje sklonim greškama.

CDE je softverski pristup u kom timovi kontinualno proizvode softver u kratkim ciklusima, osiguravajući da se softver može pouzdano isporučiti u bilo kom trenutku [3]. CDE zahtijeva CI, jer se oslanja na isti osnovni princip podjele procesa na male korake. Sve promjene koda se automatski kreiraju, testiraju i pripremaju za produkciju. Kad se CDE pravilno implementira, programeri uvijek imaju verziju artifakta koja je prošla kroz standardizovan testni proces i koji je spreman za isporuku u produkciono okruženje.

Korak dalje ide ideja o kontinualnom uvođenju u produkciju (CD). Ako se verzije automatski isporučuju u produkciono okruženje bez eksplicitnog odobrenja razvojnog programera ili testera, čineći cijeli postupak isporuke softvera automatizovanim, pristup se naziva kontinualno uvođenje u produkciju. To znači da se linija koja počinje s dodavanjem koda, kroz potpuno automatizovan proces, završava sa isporučenim softverskim proizvodom u produkciji, koji je potpuno funkcionalan i spreman za upotrebu odnosno dostupan korisnicima. Da bi se to postiglo u razvojnom okruženju se moraju kreirati odgovarajući testovi i

automatizovati svi koraci kako bi se izbjegla eventualna iznenađenja prilikom uvođenja u produkciju.

CI/CD sistem se može predstaviti kao linija (slika 3.2) gdje se novi kôd dodaje na jednoj strani, prolazi kroz nekoliko odvojenih faza ili koraka i na kraju se automatski isporučuje u produkciono okruženje. CI/CD linije su obično iterativne, a kako se kôd kreće kroz liniju, kvalitet koda postaje veći, jer prolazi kroz više faza testova i biva verifikovan na više načina. Problemi detektovani u ranim fazama zaustavljaju odnosno prekidaju izvršenje linije i istovremeno šalju rezultate testova razvojnim inženjerima, pri čemu se svi sljedeći koraci takođe otkazuju. Glavni cilj uspješne CI/CD linije i pravog CD sistema je automatizacija isporuke softvera u produkciono okruženje, pri čemu kôd prolazi kroz više iteracija pokrenutih automatski bez interakcije sa razvojnim programerima. To omogućuje brzu integraciju i isporuku malih izmjena u produkciono okruženje, svodeći rizik od eventualnih grešaka na minimum. U kontinualnom procesu uvođenja u produkciju uobičajeno je da razvojni timovi imaju nekoliko različitih pretprodukcijskih okruženja. Ta okruženja se koriste da bi se aplikacija testirala i detektovale potencijalne softverske anomalije.



Slika 3.2. Uopšteni prikaz CI/CD sistema

Neke od odlika i karakteristika robusnog CI/CD sistema [75] za mikroservisnu arhitekturu su:

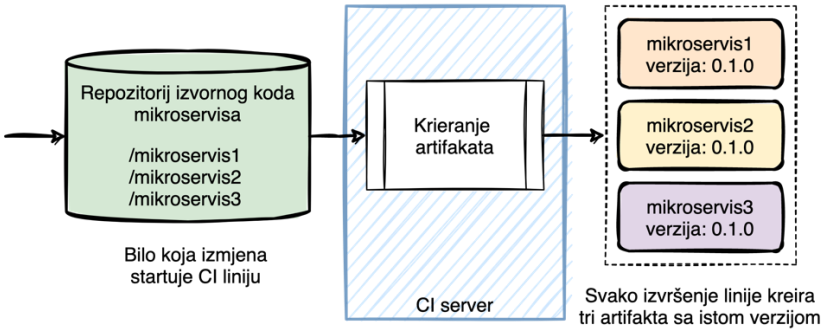
- svaki tim može da kreira i isporučuje nove verzije nezavisno, bez uticaja i ometanja drugih timova;
- prije nego što se nova verzija mikroservisne aplikacije isporuči u produkciju, ona se isporučuje u razvojno/testno/QA okruženje radi testiranja;
- nova verzija mikroservisne aplikacije može se instalirati uporedo sa postojećom;
- implementirane su odgovarajuće politike za kontrolu pristupa.

3.1. Repozitorijum izvornog koda

Prije nego što se počne sa uspostavljanjem CI/CD procesa unutar jedne organizacije, bez obzira na veličinu, odluka koja se mora donijeti u određenom trenutku je kako organizovati

strukturu repozitorijuma izvornog koda. Kako ta odluka djelimično utiče na sam dizajn CI/CD sistema, bitno je odlučiti da li se za sve mikroservisne aplikacije želi koristiti jedan repozitorijum izvornog koda ili koristiti zaseban repozitorijum za svaku mikroservisnu aplikaciju.

Prvi i najjednostavniji pristup je da se koristi jedan veliki repozitorijum (eng. *monorepo*) tako da kod ovog pristupa imamo samo jednu CI liniju za kreiranje artifakata kao što je to prikazano na slici 3.3. Izmjena samo jedne linije koda će startovati CI liniju koja će pokrenuti testove i proces kreiranja artifakata za sve mikroservise u okviru repozitorijuma, a kao rezultat generisaće se više artifakata (u našem slučaju *Docker* slika), iste verzije koje su direktno vezane za ovo izvršenje CI linije.



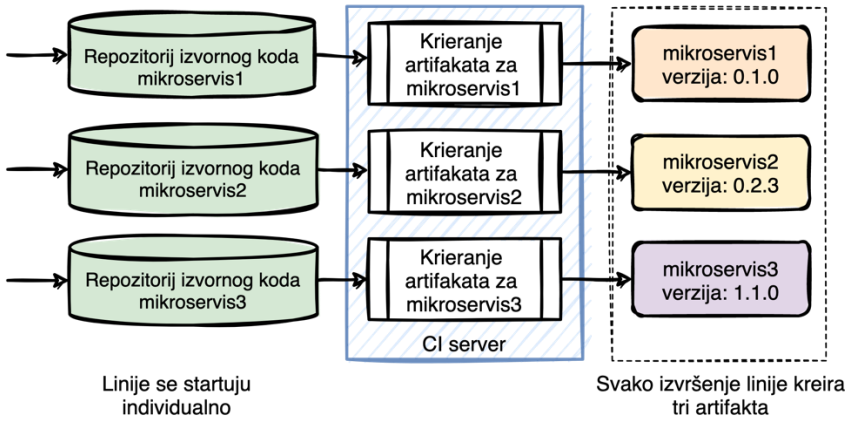
Slika 3.3. Upotreba jednog repozitorijuma i jednog CI procesa za sve mikroservise

Ovaj pristup izgleda jednostavno jer se manipuliše samo sa jednim repozitorijumom, a samim tim se dizajn i implementacija CI/CD sistema pojednostavljuje. Ukoliko to posmatramo sa tačke gledišta razvojnih programera, stvari takođe izgledaju jednostavnije i ovaj pristup im omogućava istovremeni rad na više mikroservisa. Iako ovaj pristup savršeno funkcioniše u nekim slučajevima to je pristup koji treba izbjegavati [1], jer ima nekoliko nedostataka. Na primjer ukoliko se kreira veoma jednostavna izmjena koja mijenja samo jednu liniju koda u okviru jednog mikroservisa, onda će se pokrenuti testovi i kreiranje artifakata za sve mikroservise, uključujući i one koji nisu zahvaćeni izmjenom. Time se troše dodatni hardverski resursi kao i vrijeme programera, a kompletan proces usporava, jer se nepotrebno izvršavaju testovi nad mikroservisima koji nisu zahvaćeni izmjenom. Ovakav proces često podrazumijeva da se isporuka novih verzija radi istovremeno za sve mikroservise. U slučaju da ova izmjena generiše grešku za jedan mikroservis, isporuka novih verzija svih ostalih mikroservisa je blokirana dok se greška ne ispravi.

Ukoliko se koristi pristup gdje se za svaki mikroservis kreira zaseban repozitorijum kao što je prikazano na slici 3.4, u tom slučaju imamo odvojenu CI liniju za svaki mikroservis. Ovdje svaki mikroservis ima svoj repozitorijum za izvorni kod, koji je odvojen od ostalih i mapiran na zasebnu CI/CD liniju. U slučaju da se vrši izmjena, kreira se samo jedan artifakt pa se samim tim izmjene na produkciju mogu isporučivati nezavisno od drugih mikroservisa. Ovaj pristup nema nedostatke kao prethodni, ali opet treba imati na umu da vršenje izmjena na više mikroservisa kod ovakvog pristupa može biti veoma komplikovano. Takođe, jasno se vidi koji tim je vlasnik datog mikroservisa, što pojednostavljuje upravljanje kontrolom pristupa.

Testovi za mikroservise treba da budu unutar istog repozitorijuma, jer će kao što je već rečeno, svaki mikroservis će imati zasebnu CI/CD liniju. Takođe, važno je napomenuti da bi i konfiguracioni parametri trebali da budu u ovom repozitorijumu, ali s obzirom na to da je kompleksna i bitna komponenta jednog ovakvog sistema, posebno je obrađena kasnije u ovom poglavlju.

U tabeli 3.1. dat je uporedni prikaz prednosti i nedostataka za oba prethodno objašnjena pristupa. Zbog kompleksnosti ovakvih sistema, kao i varijacija u implementaciji procesa od kojih ovih sistemi zavise, praksa pokazuje da ne postoji jedinstveno rješenje koje bi zadovoljilo sve zahtjeve i odgovaralo svima, bez obzira na veličinu tima i metodologije razvoja softvera koje se primjenjuju. Iako su ovdje prikazana dva osnovna pristupa, često u praksi imamo organizaciju repozitorijuma izvornog koda zasnovanu na pristupima koji su modifikovane odnosno prilagođene verzije prethodna dva. Na primjer, često imamo jedan repozitorijum u kojem se nalazi izvorni kôd svih servisa, ali sa druge strane imamo odvojene CI/CD linije. Ovaj pristup takođe ima svoje prednosti i nedostatke.



Slika 3.4. Upotreba zasebnog repozitorijuma izvornog koda i CI linije za svaki mikroservis

Često se preporučuje da se prilikom započinjanja novog projekta koristi jedan repozitorijum u koji se tokom inicijalne faze smještaju svi mikroservisi. Glavni razlog za to je što obično prilikom inicijalne faze projekta, nije jasno koje funkcionalnosti treba izdvojiti zasebno i koliko mikroservisa je potrebno. Osim toga, izbjegava se nepotrebno trošenje vremena na konfigurisanje zasebnih CI/CD linija. Nakon što se inicijalna faza projekta završi i mikroservisi se uspješno isporuče u razvojno okruženje, u sljedećoj fazi može se planirati reorganizacija repozitorijuma izvornog koda, odnosno izdvajanje mikroservisa u zasebne repozitorijume.

Tabela 3.1. Uporedni pregled prednosti i nedostataka između dva načina organizacije repozitorijuma izvornog koda

	Jedan repozitorijum izvornog koda	**Zaseban repozitorijum za svaki mikroservis**
Prednosti	Dijeljenje koda Jednostavnija standardizacija koda i alata Lakše refaktorisanje koda Bolji i lakši pregled – sav kôd je u jednom repozitorijumu	Jasno definisano ko je vlasnik repozitorijuma Potencijalno manje konflikata prilikom integracije Pruža podršku za migraciju iz monolita na mikroservisnu arhitekturu
Nedostaci	Izmjene u dijeljenom kodu mogu da zahvate više mikroservisa Velika mogućnost konflikata prilikom integracije Alati se moraju skalirati da bi podržali veliki repozitorijum Kontrola pristupa Kompleksniji proces za isporuku	Komplikovan proces za dijeljenje koda Teže je forsirati standarde vezano za izgled koda i alata koji se koriste Upravljanje zavisnostima (eng. *dependency management*) Teško razumijevanje koda i načina rada sistema Nedostatak zajedničke infrastrukture

3.1.1. Sistemi za kontrolu verzija

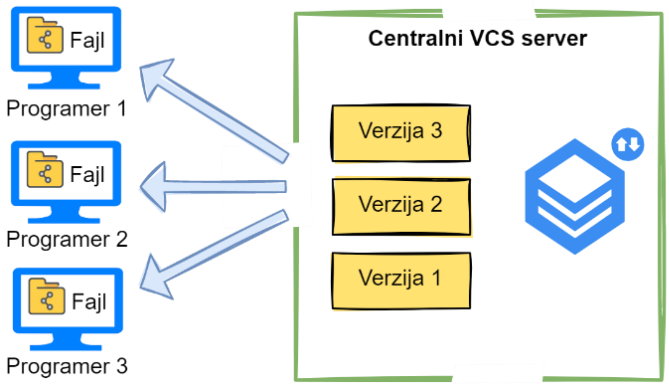
Kontrola verzija (eng. *version control*) je opšti naziv za aktivnost praćenja promjena i spajanja konkurentnih izmjena nad artifaktima od interesa za posmatranu aktivnost, uz očuvanje istorije izmjena. U softverskom inženjerstvu, kontrola verzija poznatija kao upravljanje izvornim kodom (eng. *Source Code Management*) je klasa sistema odgovornih za upravljanje promjenama u računarskim programima, dokumentima, velikim veb sajtovima i drugim kolekcijama informacija [76]. Osim primarne namjene za praćenje modifikacija i njihove istorije u repozitorijumu izvornog koda, pomaže i prilikom rješavanja konflikata koji se mogu pojaviti u trenutku integracije (spajanja) izmjena iz (više) radnih grana u glavnu granu. Predstavlja dio šire discipline pod nazivom upravljanje konfiguracijom softvera (eng. *Software Configuration Management* – SCM), koja omogućava kontrolisano praćenje i evoluciju softverskog proizvoda.

Kako razvoj softvera postaje sve složeniji i više programera radi na razvoju istog mikroservisa, proces koordinacije i upravljanja izmjenama postaje kompleksniji. Osim toga, jako teško je obezbijediti da svi programeri imaju pristup posljednjoj verziji, odnosno najnovijim izmjenama. Kao odgovor na ove izazove nastali su sistemi za kontrolu verzije (eng. *Version Control System*), koji su danas jedan od osnovnih alata u procesu razvoja softvera. U zavisnosti od arhitekture oni se oni se dijele u dvije osnovne grupe:

- centralizovana ili klijent-server arhitektura,
- distribuirana ili *peer-to-peer* arhitektura.

3.1.1.1. Centralizovani sistemi za kontrolu verzija

Kod centralizovanih sistema za kontrolu verzije zasnovanih na klijent-server arhitekturi, repozitorijum izvornog koda, koji sadrži sve verzionisane fajlove, nalazi se na centralnom serveru, dok programeri odnosno klijenti imaju pristup samo trenutnoj verziji. U ovakvoj arhitekturi, koja se dugo godina koristila kao standardna, server čuva kompletnu istoriju izmjena, odakle svaki klijent preuzima posljednju verziju, nakon čega smješta izmjene na centralni server. Najpoznatiji predstavnici centralizovanih sistema su *Subversion* [77], *CVS* [78] i *Perforce* [78].



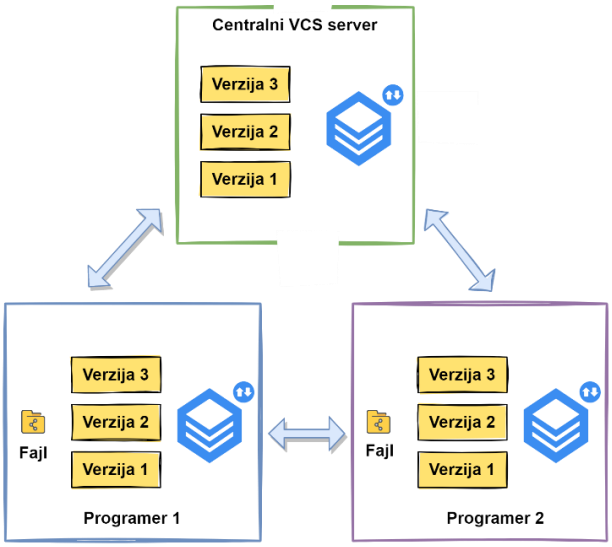
Slika 3.5. Centralizovani sistemi za kontrolu verzije

Ovakva arhitektura omogućava administratorima sistema bolju kontrolu i jednostavnije upravljanje sistemom, jer je repozitorijum izvornog koda centralizovan, dok programeri imaju bolji uvid u izmjene i znaju šta ostali programeri rade na projektu. Međutim, ovakva arhitektura ima nekoliko nedostataka, među kojima najveći predstavlja ovaj centralizovani server kao

jedna tačka otkaza (eng. *single point of failure*). Ukoliko je ovaj server nedostupan upotreba i rad sa sistemom za kontrolu verzija praktično je onemogućen. Dodatno, potrebno je pravilno vršiti bekap i arhiviranje centralnog repozitorijuma jer u slučaju hardverskih ili softverskih problema na samom serveru svi podaci bi mogli biti nepovratno izgubljeni. Ovakvi sistemi imaju lošiju skalabilnost u odnosu na distribuirane sisteme za kontrolu verzija.

3.1.1.2. Distribuirani sistemi za kontrolu verzija

Sistemi koji su implementirani u skladu sa distribuiranom arhitekturom (eng. *peer-to-peer*) nazivaju se distribuiranim sistemima za kontrolu verzija DVCS (*Distributed Version Control Systems*). Kod ovih sistema klijenti ne preuzimaju samo trenutno stanje, već svi imaju lokalnu kopiju cjelokupnog repozitorijuma, čime se omogućava rad bez pristupa centralnom serveru (eng. *offline)*. Štaviše, mnogi od ovih sistema se prilično dobro nose sa više repozitorijuma na daljinu sa kojima mogu da rade, tako da se kolaboracija između različitih timova može ostvariti veoma jednostavno. Najpoznatiji predstavnici distribuiranih sistema za kontrolu verzija su *Git* [79], *Mercurial* [80] i *Bazaar* [81].



Slika 3.6. Distribuirani sistemi za kontrolu verzije

3.1.1.3. *Git* distribuirani sistem za kontrolu verzije

Linus Torvalds, tvorac *Linux* operativnog sistema, je 2005. godine kreirao *Git* distribuirani sistem za kontrolu verzija, a implementacija je zasnovana na programskom jeziku C u kombinaciji sa *shell* skriptama. Sam projekat je nastao sa ciljem da zadovolji potrebe tadašnje *Linux* zajednice koja je radila na razvoju *Linux* kernela, a koji je po zahtjevima bio veoma neobičan za vrijeme u kojem je nastajao. Na razvoju *Linux* kernela, prilično velikog softverskog projekta otvorenog koda, radio je veliki broj programera, a promjene u softveru su se slale kao zakrpe i arhivirani fajlovi. *BitKeeper* [83], alat koji su do tada koristili, je postao komercijalan tako da je Linus odlučio da osmisle sopstveni alat, a neki od glavnih ciljeva koje je taj alat trebao da zadovolji su:

- brzina,
- jednostavan dizajn,
- snažna podrška za nelinearni razvoj (na hiljade paralelnih grana),
- potpuno distribuiran,

- mogućnost da efikasno rukuje velikim projektima kao što je *Linux* jezgro (brzina i veličina podataka).

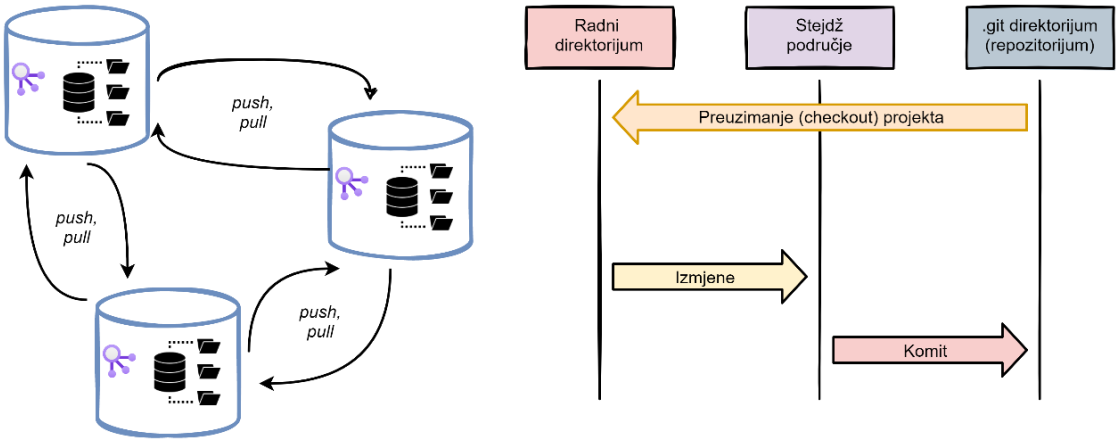
Osnovni koncepti na kojima se zasniva *Git* alat su:

- Repozitorijum (eng. *repository*) predstavlja kolekciju komita, od kojih je svaki od njih arhiva odnosno slika radnog stabla u određenom vremenskom trenutku. Takođe, definiše glavu (eng. HEAD), na osnovu koje se identifikuje grana ili komit iz kojeg je trenutno radno stablo kreirano.
- Indeks (eng. *index*) je jedna velika binarna datoteka na lokaciji *<korijenski direktorijum repozitorijuma>/.git/index*, koja sadrži listu imena svih datoteka u trenutnoj grani, SHA1 heš vrijednosti njihovih sadržaja i vremenske oznake.
- Radno stablo (eng. *working tree*) je bilo koji direktorijum na sistemu datoteka koji ima *Git* repozitorijum povezan sa njim, označen prisustvom direktorijuma pod nazivom *.git*.
- Komit (eng. *commit*) je slika radnog stabla u određenom vremenskom trenutku. Stanje glave u trenutku kada je komit kreiran postaje roditelj tog komita i to je ono što stvara istoriju revizija.
- Grana (eng. *branch*) je pokretni pokazivač na jedan od komita koji se često naziva i referenca.
- Oznaka (eng. *tag*) je referenca koja upućuje na određeni komit. Tag je slično što i grana koja se ne mijenja. Koristi se obično za označavanje određenog komita u istoriji, kako bi se označila odgovarajuća verzija (npr. v1.0.0).
- Master grana (eng. *master branch*) je podrazumijevana grana u okviru repozitorijuma i predstavlja glavnu granu za razvoj u većini repozitorijuma. Sve izmjene koje se vrše na drugim granama se po završetku implementacije integrišu u ovu granu, a nakon toga isporučuju u produkciono okruženje. Iako je podrazumijevana, ova grana se ne razlikuje od drugih grana.
- Glava je pokazivač na granu ili komit koji je posljednji preuzet, i koji će postati roditelj odnosno nadređeni novom komitu ukoliko se isti kreira. Na primjer, ukoliko smo na master grani, tada će glava pokazivati na master, a ako se kreira novi komit on će biti potomak revizije na koju je master ukazivao, a master će se ažurirati kako bi pokazivao na novi komit.

Git ima tri glavna stanja u kojima se fajlovi u okviru jednog repozitorijuma mogu naći, a to su: komitovano, modifikovano i stejdžovano. Komitovano znači da su podaci smješteni u lokalnoj bazi podataka na sigurnom. Modifikovano znači da je fajl promijenjen ali da izmjena još uvijek nije komitovana odnosno sačuvana u bazi podataka. Stejdžovano znači da je modifikovani fajl označen da se u svom trenutnom stanju uključi u sljedeći komit koji se bude kreirao [82].

Kako je *Git* distribuirani sistem to znači da nijedna kopija repozitorijuma ne može da bude označena kao centralna i da su sve jednake. To omogućava da programeri mogu dijeliti izmjene međusobno prije spajanja izmjena u produkciju. Dalje, programeri mogu da vrše promjene na svojoj lokalnoj kopiji repozitorijuma bez pristupa internetu odnosno bez potrebe za komunikacijom sa ostalim kopijama repozitorijuma. Nakon što su određene izmjene završene na lokalnoj kopiji repozitorijuma, one se dalje mogu proslijediti na drugu kopiju radi eventualnih testiranja.

Kada se datoteka doda za praćenje u okviru *Git* repozitorijuma, ona se kompresuje pomoću *zlib* algoritma [83], a zatim se kreira jedinstvena heš vrijednost pomoću SHA-1 heš funkcije, koja odgovara sadržaju datoteke. *Git* potom skladišti ove podatke u internoj bazi podataka unutar direktorijuma *.git-objects* u okviru datog repozitorijuma. Ime datoteke je generisana heš vrijednost, a datoteka sadrži kompresovani sadržaj originalne datoteke. Ove datoteke se nazivaju *blob*-ovima, a kreiraju se svaki put kada se u repozitorijum doda nova datoteka ili izmijeni jedna od postojećih datoteka. Upotreba *Git* alata prilikom implementacije CI/CD sistema, donosi mnogo pogodnosti a jedna od njih je mogućnost da se vrlo lako prate izmjene i da se u slučaju detektovanih izmjena startuje CI/CD linija, a potom na osnovu izmijenjenih fajlova izvršavaju ili preskoče određeni koraci.



Slika 3.7. Arhitektura i stanja Git distribuiranog sistema za kontrolu verzije

Git implementira *staging* indeks koji djeluje kao međuprostor za izmjene koje su spremljene za komitovanje. Kada se izmjene dodaju na stejdžing kao spremne za komitovanje, njihov kompresovani sadržaj referencira se u posebnoj indeks datoteci čija je struktura podataka u obliku stabla. Stablo je *Git* objekat koji povezuje *blob* objekte sa stvarnim imenima datoteka, permisijama i vezama do drugih stabala i na taj način reprezentuje stanje određenog skupa datoteka i direktorijuma. Nakon što su sve izmjene dodate u indeks, on može biti dodat u repozitorijum, čime se kreira komit objekat u *Git* bazi objekata. Nakon toga, ovaj komit je glava stabla (HEAD) za određenu reviziju, koja sadrži ime autora, njegovu adresu elektronske pošte, datum i deskriptivni tekst kao opis izmjene kreirane datim komitom. Svaki komit sadrži referencu na roditeljski komit (eng. *parent commit*) i tako se vremenom uspostavlja istorija razvoja projekta.

Svi *Git* objekti – *blob*-ovi, stabla i komiti su kompresovani, heširani i čuvaju se u bazi podataka objekata, na osnovu njihove heš vrijednosti. Oni se nazivaju labavi objekti (eng. *loose objects*). Ukoliko se izmijeni jedna datoteka, ne čuvaju se razlike fajlova (eng. *diff*) već cijela datoteka, što *Git* čini veoma brzim, jer je cijeli sadržaj datoteke za svaku reviziju dostupan kao slobodan objekt. Međutim, određene operacije, kao što su smještanje (eng. *push*) komita u udaljeni repozitorijum, skladištenje velikog broja objekata ili ručnog pokretanja naredbe za recikliranje (eng. *garbage collection*) mogu inicirati proces prepakivanja objekata u odvojene fajlove koji se nazivaju paketi (eng. *pack files*). U procesu pakovanja čuva se samo posljednja verzija datoteke netaknuta, dok se od originalne verzije čuva samo kompresovana razlika, kako bi se eliminisao suvišan sadržaj i smanjila veličina. Ovdje se polazi od pretpostavke da će biti potreban brži pristup posljednjoj, odnosno najnovijoj verziji datoteke. Kao rezultat nastaju

paketi u obliku datoteka sa *.pack* ekstenzijom, u kojima je smješten sadržaj datoteke, pri čemu svaki ima odgovarajući indeks fajl sa *.idx* ekstenzijom, koji sadrži referencu upakovanih objekata i njihovih lokacija unutar *.pack* fajla. Ovi paketi se prenose preko mreže kada se grane smještaju ili preuzimaju iz udaljenih repozitorijuma, nakon čeka se paketi raspakuju da bi se stvorili labavi objekti u repozitorijumu objekata.

3.2. Strategije grananja

Za razvojni tim od vitalnog značaja je izvorni kôd softvera na kojem rade. Uporedo sa razvojem softvera, raste baza koda (eng. *codebase*) i eventualno broj programera koji rade na razvoju, odnosno implementaciji novih funkcionalnosti. Iako je vremenom razvijen niz alata, prije svega VCS alata za upravljanje izvornim kodom i kontrolu verzija, još uvijek postoje određeni izazovi pri izboru efikasnog načina za upotrebu funkcionalnosti koje oni nude [84]. Izazovi se prvenstveno ogledaju u načinu na koji voditi odvojene grane u okviru repozitorijuma izvornog koda tokom razvoja softvera, kao i integraciji izmjena u glavnu granu. To dodatno može da zavisi i od konteksta, tipa softvera koji se razvija, strukturi tima kao i praksama koje tim slijedi. Vremenom je definisan određen broj strategija koju su evoluirale iz praksi upotrebljavanih u timovima i organizacijama različitih veličina. Kao i većina šablona koji se koriste u razvoju softvera, ove strategije nisu zlatna pravila, već smjernice kako integraciju izmjena i isporuku softvera pojednostaviti i učiniti efikasnijim. Kako je pravilan izbor strategije grananja takođe bitan prilikom implementacije CI/CD sistema, ukratko će biti dat kratak pregled osnovnih strategija koje se koriste.

Prema Martinu Fowler-u, strategije grananja se mogu svrstati u dvije glavne kategorije [84], koje će u narednom tekstu biti ukratko obrađene.

3.2.1. Osnovni modeli

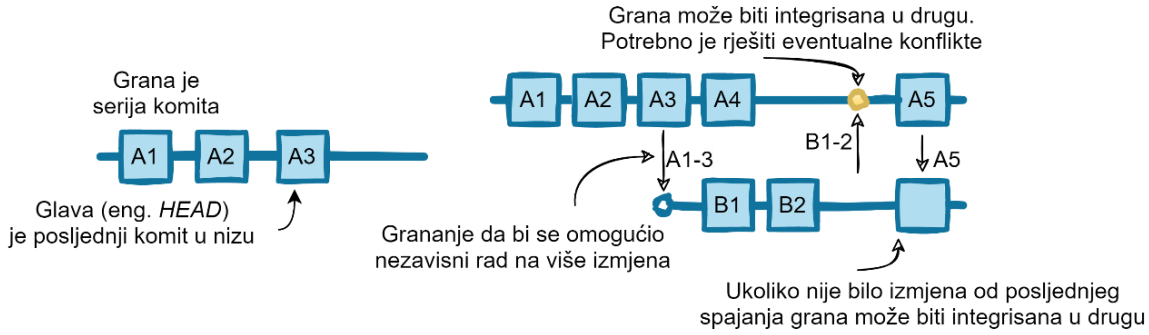
Strategije grananje koje se odnose na praćenje grananja pripadaju kategoriji osnovnih modela. One su orijentisane na to kako grananje pomaže u upravljanju razvojem softvera od repozitorijuma izvornog koda, do finalnog softvera koji se izvršava u produkciji.

3.2.1.1. Izvorno grananje

Sistemi za kontrolu verzija omogućavaju da istovremeno više programera radi na istom repozitorijumu izvornog koda, uključujući mogućnosti da mijenjaju iste fajlove. Kako svaki od njih radi na lokalnoj kopiji repozitorijuma, po završetku izmjena one moraju da se integrišu i isporuče u produkciju. Sistemi za kontrolu verzija olakšavaju ovaj proces tako što se svaka izmjena sastoji od niza komita određene sekvence, što zapravo predstavlja granu, dok je posljednji komit u tom nizu glava grane. Grane u okviru repozitorijuma izvornog koda se kreiraju procesom grananja. Pod tim se podrazumijeva kreiranje nove grane odnosno dijeljenje originalne grane u dvije grane. Analogno, spajanje grana (eng. *merge*) nastaje tako što se izmjene iz jedne grane integrišu u drugu granu.

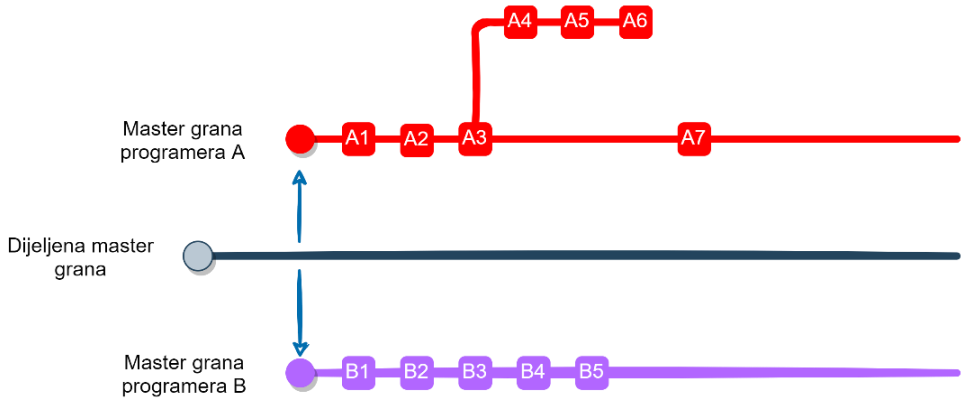
Radi boljeg razumijevanja posmatrajmo sada jednu tipičnu situaciju na slici 3.9, specifičnu za moderne razvojne timove gdje dva programera rade na razvoju mikroservisa, pri čemu je izvorni kôd smješten u okviru *Git* repozitorijuma. Programer A kreira lokalnu kopiju repozitorijuma izvornog koda i kreira nekoliko izmjena odnosno komita A1, A2,…, A6 nakon čega odlučuje da te izmjene nisu potrebne i kreira novi komit A7 koji poništava sve prethodne komite odnosno izmjene koje je napravio. Istovremeno programer B kreira lokalnu kopiju repozitorijuma izvornog koda i kreira nekoliko izmjena B1, B2,…, B5. Oba programera rade

na lokalnoj kopiji iste (master) grane. Da bi se ove izmjene sačuvale u dijeljeni repozitorijum svi ovi komiti moraju biti integrisani u istu master granu. Teoretski programer A ne mora da svoje izmjene isporuči u dijeljeni repozitorijum, ali izmjene koje je kreirao sastoje se od niza komita i predstavljaju jednu granu. Ukoliko on ne želi da te izmjene sačuva tada mora da izvrši resetovanje lokalne master grane na stanje koje je bilo prije nego što je počeo da kreira izmjene, pri čemu će glava master grane ponovo biti resetovana na posljednji komit prije A1.



Slika 3.8. Izvorno grananje

Iako praksa pokazuje da sistemi za kontrolu verzija olakšavaju praćenje istorije i integraciju izmjena, oni ga ne čine trivijalnim. Na primjer, ako oba programera izmijene naziv varijable doći će do konflikta koji se mora ručno riješiti. Pokazuje se da su sistemi za kontrolu verzija jako dobri kada se radi o tekstualnim razlikama između fajlova. Međutim, veoma česte su situacije da su izmjene integrisane na master granu ali da mikroservis i dalje ne radi. Ovdje se obično radi o semantičkim konfliktima, koji nastaju na primjer ako jedan programer promijeni naziv funkcije koju poziva drugi dio koda. Kada se pojave ovakvi konflikti odgovornost se prebacuje na CI liniju odnosno testove pa se softverski artifakt neće uspješno kreirati ili ukoliko se uspješno kreira pojaviće se problemi tokom izvršavanja.



Slika 3.9. Primjer upotrebe izvornog grananja

Iz prethodnog se jasno zaključuje zašto je izbor strategije grananja jako bitan za implementaciju CI/CD sistema. Ovaj problem je poznat svima koji su radili sa distribuiranim sistemima, gdje imamo jedno dijeljeno stanje (centralni repozitorijum izvornog koda) koje programeri paralelno ažuriraju. Implementacija sistema koji će obezbijediti neku vrstu konsenzusa odnosno veoma složenu validaciju, implementirajući neki vid serijalizovane integracije, osiguravajući pri tom ispravnost artifakta i pravilno izvršavanje mikroservisa je veoma kompleksno. Ne postoji način za stvaranje determinističkog algoritma za pronalaženje konsenzusa već programeri moraju poznavati sam proces isporuke softvera. Često je pri rješavanju konflikata potrebna njihova intervencija, što obično uključuje kreiranje novog komita a ponekad i zasebne grane koja sadrži izmjene kombinovane iz više drugih grana.

3.2.1.2. Glavna linija

Glavna linija (eng. *mainline*) je posebna grana koju smatramo trenutnim stanjem koda za odgovarajući repozitorijum izvornog koda. Razvojni timovi koriste različita imena za ovu posebnu granu, često u zavisnosti od tipa sistema za kontrolu verzija koji se koristi. Ustaljeni nazivi za ovu granu su *master* i *main*. Međutim, ukoliko se koristi naziv master za glavnu liniju zajedno sa *Git* sistemom za kontrolu verzija, u tom slučaju taj naziv može da ima više značenja. Svaka lokalna kopija repozitorijuma izvornog koda ima svoju lokalnu master granu. Razvojni timovi obično imaju jedan centralni repozitorijum kao ishodište svih lokalnih kopija. U tom slučaju glavna linija je master grana u okviru centralnog repozitorijuma, dok stanje lokalnih master grana može da odstupa od glavne linije.

Ukoliko se želi započeti sa razvojem nove funkcionalnosti, obično se izvrši ažuriranje lokalne master grane (ili cijelog repozitorijuma), kako bi se broj eventualnih konflikata prilikom spajanja izmjena na kraju, sveo na minimum. Nakon toga kreira se zasebna grana u koju se dodaju željene izmjene u vidu jednog ili više komita. Ukoliko su izmjene spremne za isporuku ili postoji potreba da se one podijele sa ostatkom tima, ta grana se može smjestiti u centralni repozitorijum, nakon čega svako može da je preuzme odnosno sinhronizuje sa svojom lokalnom kopijom. Upotrebom ove strategije svako može veoma jednostavno i brzo da započne razvoj softvera od posljednje stabilne (ispravne verzije) izmjene. Dodatno, glavna linija ne samo da olakšava uvid u stanje baze koda, već je temelj većine dugih obrazaca koji će biti opisani u narednom tekstu.

3.2.1.3. Zdrava grana

Strategija zdrave grane (eng. *healthy branch*) se zasniva na ideji da se prilikom svakog komita izvrše svi automatizovani testovi kako bi se osiguralo da nema grešaka na glavnoj grani odnosno da je grana zdrava. To podrazumijeva pisanje softvera bez ili sa veoma malo grešaka. Ova razvojna praksa podrazumijeva da se prilikom pisanja softvera, piše i sveobuhvatan paket automatizovanih testova. Automatizacija testova, kao i njihovo izvršavanje prilikom svake izmjene, implementira se upotrebom CI linije, dok teoretski osigurava da glavna grana ostane zdrava.

U slučaju da dođe do greške prilikom izvršavanja CI linije, to obično znači da testovi nisu izvršeni uspješno ili da se desila greška prilikom kompajliranja pa samim tim kôd nije moguće integrisati u glavnu granu. Tada se glavna grana obično zaključava i nisu dozvoljene nikakve izmjene, osim izmjena neophodnih da bi se grana dovela u zdravo stanje.

Kada se govori o testovima neophodnim kako bi se osiguralo da je grana zdrava, treba voditi računa o stepenu testiranja, jer izvršavanje temeljnijih testova obično traje duže vremena čime se odlaže i povratna informacija o tome da li se izmjene mogu integrisati. Ovaj problem se obično rješava razdvajanjem testova u okviru CI linije u više faza. Prva faza ovih testova bi trebala da se izvršava veoma brzo i ne duže od deset minuta, ali i dalje razumno sveobuhvatna. Idealno bi bilo da se prilikom svakog komita izvrše svi testovi, ali na primjer izvršavanje testova performansi može da traje i do nekoliko sati, pa samim tim nije praktično.

Međutim, ako kôd prođe uspješno sve testove i izvršava se bez greške to ne mora uvijek značiti da je dobar odnosno kvalitetan. Da bi se održao konstantan tempo isporuke neophodno je održavati kvalitet koda. To se obično radi na način da kôd pregledaju odnosno recenziraju drugi programeri prije integracije sa glavnom granom. Nezavisno od veličine organizacije i

metodologije razvoja softvera, svaki tim bi trebao da ima jasno definisane standarde kako održavati grane zdravima uz očuvanje visokog nivoa kvaliteta koda.

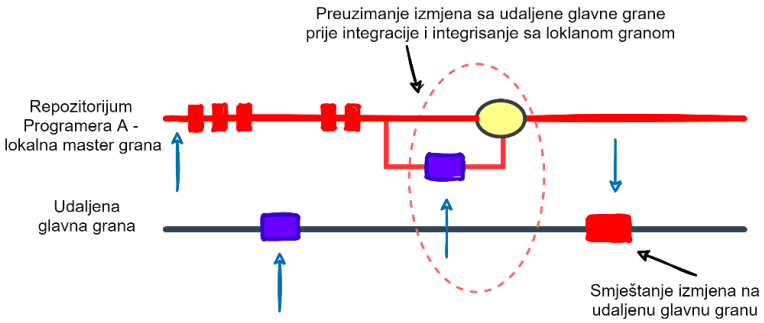
Da bi se glavna grana održala u zdravom stanju najvažnije je imati odgovarajuće testove, a posebno prethodno pomenute vrste testova koji se izvršavaju u prvoj fazi za svega nekoliko minuta. Sama implementacija ovih testova obično zahtijeva znatne resurse ali nakon što se implementira, znatno pojednostavljuje i ubrzava proces isporuke u produkciju. Sam proces razvoja softvera se kompletno mijenja, a praksa pokazuje da nakon što se ovaj proces uspostavi i stabilizuje, izmjene se mogu puštati sa glavne grane na produkciju. Ukoliko je glavna grana u zdravom stanju, sa jedne strane to omogućava programerima da kreiranjem nove grane započnu rad na određenim izmjenama, dok sa druge strane skraćuje put isporuke softvera u produkciono okruženje.

3.2.2. Integracioni modeli grananja

Integracioni modeli grananja se odnose na integraciju izmjena, odnosno kako više programera kombinuju rad u koherentnu cjelinu. Kao što se može zaključiti iz prethodnog teksta, grananje se odnosi na međusobnu izolaciju i integraciju koda. Moderni sistemi za kontrolu verzija olakšavaju grananje i nadgledanje izmjena nad granama, čime se omogućava i određen vid izolacije. U jednom trenutku, nakon što su izmjene završene, one se moraju integrisati sa glavnom granom. Strategije grananja zapravo daju smjernice o načinu i vremenu (trenutku) integracije.

3.2.2.1. Integracija glavne grane

Strategija integracije glavne grane (eng. *mainline integration*) se zasniva na ideji da programeri integrišu svoje izmjene na način da prvo preuzmu posljednje izmjene sa glavne grane, integrišu svoje izmjene u lokalnu glavnu granu i nakon što se izmjene uspješno integrišu, smještaju ih nazad na glavnu granu u okviru centralnog repozitorijuma. Kao što je već pomenuto, glavna grana daje jasnu definiciju kako izgleda trenutno stanje mikroservisa. Jedna od najvećih prednosti korištenja strategije glavne grane je što pojednostavljuje integraciju i omogućava svakom programeru da integriše izmjene nezavisno.



Slika 3.10. Grafički prikaz koraka kod integracije glavne grane

Posmatrajmo to sada na konkretnom primjeru prikazanom na slici 3.10. Recimo da programer A želi da započne razvoj nove funkcionalnosti za određeni mikroservis i da se *Git* koristi kao distribuirani sistem za kontrolu verzija. Ukoliko programer nema lokalnu kopiju repozitorijuma, on će je napraviti kloniranjem udaljenog repozitorijuma, a ukoliko ima lokalnu kopiju on će da se prebaci na lokalnu master granu i preuzme posljednju verziju glavne (master) grane iz udaljenog repozitorijuma. Dok on radi na razvoju nove funkcionalnosti, drugi programer B smjesti nove izmjene na glavnu granu. U jednom trenutku, nakon što su izmjene

završene i testirane lokalno, programer A želi da ih integriše u glavnu granu. Prvi korak je da se preuzme posljednja verzija glavne grane koja sadrži izmjene programera B. Kako programer B radi na lokalnoj master grani on će moći da vidi u lokalnoj istoriji da je njegova lokalna master grana jedan komit iza glavne grane pa je potrebno integrisati taj jedan komit koji sadrži izmjene koje je kreirao programer B.

Integracija izmjena se može uraditi na dva različita načina: spajanjem (eng. *merge*) ili rebaziranjem (eng. *rebase*). Uglavnom je ustaljen naziv spajanje kada se govori o grananju bez obzira na to koji način se koristi. Jedina je razlika između spajanja i rebaziranja je što će rezultujuća struktura stabla *Git* istorije biti drugačija (jedno će imati grane, drugo ne), pri čemu će spajanje stvoriti jedan dodatni komit.

Uz zavisnosti od toga koje fajlove su programeri izmijenili, integracija izmjena koje je napravio programer B će se uspješno integrisati sa lokalnim. Međutim, vrlo je vjerovatno da će se desiti neki tekstualni ili semantički konflikti. Tekstualni konflikti obično nastaju ukoliko su oba programera izmijenila isti fajl i većina sistema za kontrolu verzija ima već ugrađene alate za njihovo rješavanje. Veći problem predstavljaju eventualni semantički konflikti i bez obzira što su izmjene uspješno integrisane, programer A mora da izvrši određen broj lokalnih testova kako bi potvrdio da integrisani kôd zadovoljava standarde glavne grane (kako bi se očuvala zdravom). Iako je integracija završena lokalno, važno je napomenuti da izmjene još uvijek nisu integrisane na udaljenu glavnu granu, i tek nakon što se smjeste na glavnu granu u okviru centralnog repozitorijuma izvornog koda, a potom CI/CD linija uspješno izvrši, proces integracije je završen, a izmjene će biti dostupne ostalim programerima. Često se desi da u međuvremenu neko smjesti nove izmjene na glavnu granu pa će to rezultovati u nemogućnosti smještanja komita na glavnu granu. U tom slučaju programer A mora da ponovi kompletan proces odnosno da preuzme posljednju verziju glavne grane.

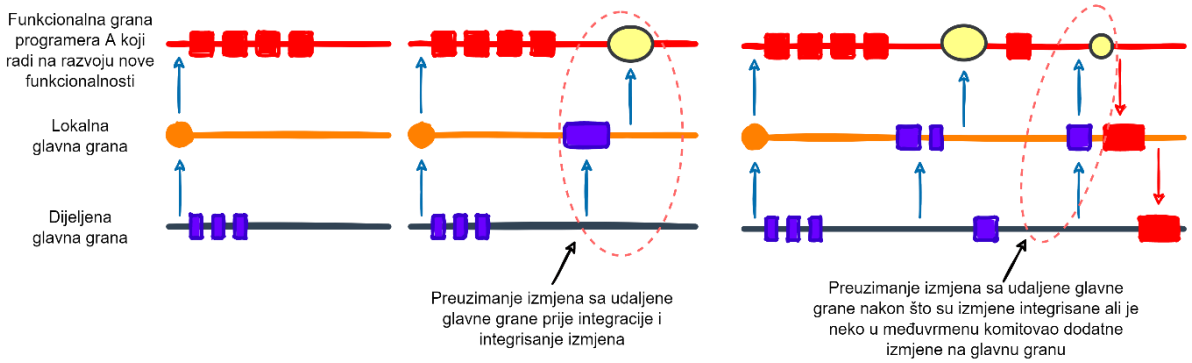
Ovu strategiju integracije moguće je jedino koristiti ukoliko se koristi strategija glavne grane kao strategija grananja. Često se sam koncept može koristiti ukoliko programeri rade na zasebnoj grani i periodično preuzimanje izmjena sa master grane može olakšati i smanjiti broj eventualnih konflikata kod kasnije integracije na master granu.

3.2.2.2. Funkcionalno grananje

Funkcionalno grananje (eng. *feature branching*) se zasniva na ideji da se sav razvoj radi na zasebnoj grani, koju programeri kreiraju kad počnu da rade na novoj funkcionalnosti ili ispravci greške, a nakon što se implementacija završi radi se integracija sa glavnom granom. Posmatrajmo sada jedan jednostavan primjer, gdje programer A radi na razvoju nove funkcionalnosti. On započinje rad sa posljednjom stabilnom verzijom mikroservisa, tako što preuzme posljednju verziju glavne grane u svoj lokalni repozitorijum, a zatim kreira novu granu koja počinje od trenutnog vrha odnosno glave glavne grane. On radi na implementaciji nove funkcionalnosti u ovoj zasebnoj grani dok je ne završi.

On može povremeno da preuzme posljednju verziju glavne grane i integriše je u svoju, kako bi mogao da utvrdi da li će bilo koje druge izmjene izazvati konflikt sa njegovim izmjenama. Ovdje se radi o integraciji glavne grane u lokalnu granu tako da se izmjene još uvijek neće pojaviti na glavnoj grani. Dodatno, neki timovi žele da se sav kôd integrisan ili ne čuva u centralnom repozitorijumu. Ukoliko je potrebno, on može da smjesti ovu granu u centralni repozitorijum tako da drugi programeri mogu da vide njegove izmjene, čime se osigurava i bekap lokalne grane. U slučaju da se radi na više funkcionalnosti istovremeno onda se za svaku od njih kreira zasebna grana.

Kada se govori o funkcionalnom grananju, veoma važan faktor prilikom implementacije je frekvencija. Česta integracija povećava učestalost spajanja izmjena, ali smanjuje njihovu složenost i rizik. Česta integracija takođe mnogo brže detektuje konflikte i pravovremeno upozorava razvojne timove na konflikte. Veoma kompleksni konflikti prilikom spajanja obično su rezultat konflikta koji je bio latentan tokom razvoja, a pojavio se tek tokom integracije. Ukoliko je kôd dobro pokriven testovima trebao bi da detektuje semantičke greške i onemogući njihovo probijanje do produkcije. Međutim, čak i kada je kôd jako dobro pokriven testovima veoma teško je integrisati velike izmjene na glavnu granu. Osim toga, često se mogu pojaviti različite greške koje je izuzetno teško razumjeti.



Slika 3.11. Grafički prikaz integracije izmjena upotrebom strategije funkcionalnog grananja

3.2.2.3. Kontinualna integracija

Kontinualna integracija (eng. *continuous integration*) se zasniva na ideji da programeri rade integraciju izmjena na glavnu granu čim imaju spreman komit odnosno izmjenu za podijeliti, a to obično podrazumijeva izmjene na kojima se ne radi više od jednog dana. Jednom kada tim uspostavi visokofrekventnu integraciju i isporuku u produkciju, postavlja se pitanje koliko često isporučivati izmjene, odnosno kako naći optimalnu frekvenciju koja će proces razvoja i isporuke softvera učiniti efikasnim i manje stresnim, svodeći broj konflikata prilikom integracije na minimum.

Kod ove strategije se izmjene integrišu na glavnu granu čim se ostvari neki napredak u kreiranju izmjena, a lokalna grana je i dalje zdrava. Važno je napomenuti da to ne podrazumijeva da je izmjena kompletna (funkcionalnost na kojoj se radi je implementirana) već da su kreirane izmjene koje ima smisla komitovati odnosno isporučiti na glavnu granu. Osnovno pravilo je da svi programeri isporučuju izmjene na glavnu granu na dnevnom nivou, odnosno da ne bi trebalo imati novog koda u lokalnom repozitorijumu starijeg od jednog dana. Praksa pokazuje da većina timova koja koristi strategiju kontinualne integracije, integriše izmjene na glavnu granu više puta na dnevnom nivou.

Timovi koji koriste ovu strategiju moraju se navići na ideju integracije djelimično implementiranih funkcionalnosti kako bi se ostvarila visokofrekventna integracija. Dodatno, oni moraju da razmotre kako da se djelimično završene funkcionalnosti ne izlažu korisnicima. Skrivanje pozadinskih funkcija obično je lako jer ih niko ne koristi, međutim problem može biti sa korisničkim interfejsom. Za to se obično koriste konfiguracioni parametri (eng. *feature flags*) koje, pored skrivanja određenih funkcionalnosti, omogućavaju i selektivnu aktivaciju funkcionalnosti samo određenom skupu korisnika. Ovdje je jako važno imati dobre testove koji bi potvrdili da djelimično implementirane funkcionalnosti ne unose nove greške. To obično podrazumijeva istovremeno pisanje testova za ovakve nezavršene funkcionalnosti koji se

takođe isporučuju na glavnu granu. Ukoliko se govori o lokalnom repozitorijumu i strategiji grananja većina programera obično radi na lokalnoj master grani i vrše integraciju na udaljenu glavnu granu, dok određen broj i dalje koristi zasebnu granu uz povremeno rebaziranje sa glavnom granom.

3.3. Kreiranje i verzionisanje artifakata

Pod pojmom artifakt u CI/CD sistemu se podrazumijeva rezultat procesa kreiranja (eng. *build*) aplikacije odnosno binarni fajl koji predstavlja upakovanu aplikaciju ili softverski modul spreman za isporuku u aplikativno okruženje. Iako postoji više formata artifakata uzimajući u obzir da je fokus rada na mikroservisnoj arhitekturi, u našem slučaju će to biti *Docker* slika u OCI formatu, koja se kao što je već objašnjeno nakon kreiranja ne može mijenjati.

Svako uspješno izvršenje CI linije kao dijela CI/CD sistema, će kao rezultat generisati jednu sliku, što će nakon određenog vremenskog perioda rezultovati u velikom broju aplikativnih slika. Međutim, samo neke od njih su kandidati za isporuku na produkciju i samo neke od njih će biti promovisane odnosno isporučene (eng. *deployed*) u produkciono okruženje. Zbog toga, bez obzira na broj mikroservisa, preporučuje se definisanje politike na nivou organizacije za upravljanje i verzionisanje slika, odnosno brisanje nepotrebnih slika koje nisu isporučene u produkciono okruženje.

Bez obzira na to koji se *container engine* odnosno format slika koristi na nivou organizacije, potrebno je koristiti jedinstvenu konvenciju koja će se koristiti za verzionisanje artifakata kao i davanje naziva resursima koji se kreiraju unutar mikroservisnog okruženja. Ovakav pristup ima nekoliko prednosti, a između ostalih olakšaće monitoring i omogućiti jednostavnije dijagnostifikovanje problema prilikom isporuke nove verzije mikroservisa.

Prilikom kreiranja aplikativnih slika treba voditi računa o permisijama, pri čemu se prvenstveno misli na obezbjeđivanje da se kontejneri ne izvršavaju kao privilegovani. Pravilno konfigurisan sistem za orkestraciju kontejnera trebao bi da ima konfigurisane uloge koje ovo zabranjuju, osim u slučaju nekih sistemskih servisa koji su sastavni dio *Kubernetes* platforme. Međutim, kako se ovo odnosi na samu konfiguraciju okruženja, koja nije sastavni dio izvornog koda, a samim tim ni artifakata odnosno slika, biće obrađena zasebno kasnije u ovom poglavlju.

3.3.1. Verzionisanje *Docker* slika upotrebom semantičkih verzija

Ukoliko je unutar organizacije implementiran CD proces, to znači da se nove verzije mikroservisa isporučuju učestalo i njih je potrebno pratiti. Posmatrajući arhitekturu mikroservisne aplikacije jasno je da mikroservisi komuniciraju međusobno. Da bi se obezbijedila pouzdana isporuka novih verzija bez rizika da izmjene koje nisu kompatibilne prouzrokuju greške i nedostupnost sistema, potrebno je usvojiti odgovarajuću proceduru i način označavanja verzija. Šema verzionisanja, koja se veoma često koristi i koja omogućava jednostavan i razumljiv način označavanja verzija, je upotreba semantičkih verzija (eng. *semantic versioning specification*) [85] koja se sastoji od tri broja razdvojena tačkama u formatu X.Y.Z, gdje je:

- X je glavna verzija (eng. *major*) i onda se uvećava samo u slučaju da se uvode izmjene koje nisu kompatibilne sa prethodnom verzijom;
- Y je sporedna verzija (eng. *minor*) koja se uvećava u slučaju da se dodaju neke nove funkcionalnosti već postojećem mikroservisu;

- Z je verzija ispravke (eng. *patch*) koja donosi samo ispravke za već postojeće funkcionalnosti.

Svaka izmjena sporedne verzije ili verzije ispravke mora da bude kompatibilna sa prethodnim verzijama. Slijedeći ovu šemu za označavanje verzija, svi razvojni timovi i programeri znaju koliko velike promjene donosi nova verzija. Shodno tome oni imaju fleksibilnost da izaberu koju verziju žele da koriste, a pored toga, ukoliko odaberu verziju u formatu X.Y.Z, oni znaju da se ona nikad neće promijeniti. Takođe, mogu da konfigurišu svoj mikroservis na način da automatski preuzimaju ažuriranja (verzije) vezana za ispravke.

3.3.2. Verzionisanje *Docker* slika upotrebom heš vrijednosti *Git* komita

Ukoliko se koristi napredan sistem za kontinuiranu isporuku i puštanje u produkciju softvera, u većini slučajeva se pokazuje da upotreba semantičkih verzija nije praktična. Osnovni razlog za to je što se u tom slučaju mora unaprijed usaglasiti gdje će se čuvati verzija mikroservisa odnosno slike, a onda to implementirati u okviru CI linije. Obično se to rješava upotrebom labele u okviru *Dockerfile* ili jednog fajla (pod nazivom TAG ili *version*) u okviru repozitorijuma izvornog koda. To znači da se svaki put prilikom promjene verzije mora napraviti ručna izmjena (inkrement) verzije unutar samog repozitorijuma mikroservisa. Zbog ovih nedostataka, obično se pribjegava upotrebi SHA-1 heš vrijednosti komita koji se kreira kada se ova izmjena integriše sa glavnom granom repozitorijuma. Po dizajnu *Git* sistema, ova vrijednost je jedinstvena i nepromjenjiva, pri čemu se uvijek odnosi se na tačno određenu verziju mikroservisne aplikacije.

Upotrebom ovog načina označavanja često se uz samu heš vrijednost dodaje i naziv grane prilikom generisanja izmjene. To znači da verzija slike kada se vrši izmjena na glavnoj grani izgleda na primjer main-2334965 (eng. *main* ili *master* je glavna grana) ili pr-3498567 ukoliko se radi o sporednoj grani na kojoj se vrši određena izmjena pri čemu *PR* (eng. *pull request*) označava da se radi o zahtjevu za integraciju na glavnu granu (eng. *merge request*). Ovo su samo neki od primjera kako se označavanje slika može vršiti i svaka organizacija ima slobodu da izabere i koristi onaj način koji joj najbolje odgovara.

Upotreba ovog načina označavanja omogućava da se veoma lako nađe odgovarajući komit unutar repozitorijuma izvornog koda mikroservisne aplikacije. Na primjer, naziv slike main-2334965 jasno nam govori da se radi o izmjeni na glavnoj (*main*) grani, a da je broj komita 2334965. Na osnovu ovih podataka, u okviru repozitorijuma izvornog koda može se lako pronaći izmjena odnosno komit i vidjeti šta je zapravo izmijenjeno u okviru te izmjene. Važno je još jednom napomenuti da se i semantičko verzionisanje može koristiti ukoliko postoji način da ih CI/CD sistem generiše automatski ili upotrebom nekog eksternog sistema. To najviše zavisi od alata koji se koriste kao i načina na koji je proces isporuke softvera organizovan.

3.4. Digitalno potpisivanje i potvrda porijekla

Potpisivanje koda (eng. *code signing*) je proces digitalnog potpisivanja artifakata odnosno izvršnih fajlova ili skripti sa ciljem potvrde autora softvera i garancije da kôd nije izmijenjen ili oštećen od trenutka kada je potpisan. Proces obično koristi upotrebu kriptografskog heša za potvrdu autentičnosti i integriteta [86]. Jedan primjer je RPM (*RedHat Package Manager*) paket koji podrazumijevano validira potpise *.rpm* paketa prilikom njihove instalacije.

Na isti način, organizacije koje koriste mikroservisnu arhitekturu, žele da budu sigurne da slike koje koriste potiču iz pouzdanog izvora koji često može da bude i neki od njihovih razvojnih timova. Implementacija mehanizma za bezbijednu tranziciju slika od razvoja do produkcije, koji će obezbijediti autentičnost odnosno validaciju potpisa, poboljšati sigurnost kontejnera i smanjiti eventualnu količinu zahtjeva za provjerom valjanosti u produkcionom okruženju [89]. Potpisivanje slika bez obzira na to koji se format slika koristi, uglavnom je sastavni dio sigurnosnih politika u većini organizacija koje koriste kontejnerske tehnologije. Sa sigurnosnog aspekta, kao takvog, trebalo bi ga integrisati u CI/CD proces ili u bilo koji drugi proces koji kreira slike kontejnera [88].

Implementacija jednog ovakvog procesa uglavnom se sastoji od dva koraka: potpisivanja slike i validacije potpisa prilikom kreiranja kontejnera. Sistem za potpisivanje slika trebao bi, između ostalog, da obezbijedi:

- provjeru porijekla (odakle slika dolazi),
- potvrdu autentičnosti i integriteta, odnosno da slika nije promijenjena i
- eventualno upotrebu polisa kako bi se filtriralo koje validirane slike se mogu koristiti.

Neki od postojećih alata koji se koriste su skopeo [90] koji je razvijen od strane kompanije RedHat i cosign [91].

3.5. Aplikativno okruženje i konfiguracija okruženja

Nezavisno od toga koliko se individualnih komponenata razvija i isporučuje u produkciju, jedan od najvećih problema sa kojima se susreću razvojni inženjeri i sistem administratori je razlika između okruženja u koja se aplikacija isporučuje odnosno izvršava. Pod razlikama se ne podrazumijeva samo razlika između razvojnog i produkcionog okruženja, već i razlike između istih produkcionih okruženja. Osim toga, nezaobilazna činjenica je da će se okruženje odnosno konfiguracija jedne produkcione mašine mijenjati sa vremenom.

Razlike koje ovdje pominjemo mogu varirati od hardverskih razlika između serverskih mašina, operativnog sistema, pa sve do biblioteka koje su instalirane na svakoj od njih. Upravljanje konfiguracijom vrše uglavnom sistem administratori koji su dio operativnih timova, dok većinom u razvojnom okruženju razvojni inženjeri vrše izmjene. Razlika je u tome što ove dvije grupe inženjera imaju obično različite poglede na same procese oko upravljanja konfiguracijom, ažuriranjem sigurnosnih ispravki i slično, pa se često dešava da ova dva okruženja budu veoma različita. Takođe, na produkcionom okruženju se izvršavaju aplikacije od više razvojnih timova i samim tim održavanje konzistentne konfiguracije u svim tim okruženjima, u zavisnosti od veličine organizacije i broj razvojnih timova, često predstavlja izazov [73]. Produkciono okruženje treba da osigura okruženje za sve aplikacije, kako bi se one mogle uspješno izvršavati bez obzira na različite zahtjeve koji mogu biti čak i različite verzije biblioteka koje su u konfliktu. Na primjer, ukoliko imamo dvije aplikacije pisane u PHP programskom jeziku od kojih je jedna kompatibilna sa PHP verzijom 5.5, a druga je nova aplikacija razvijena za PHP verziju 7.0.

ITIL definiše proces upravljanja konfiguracijom (eng. Configuration Management) [90] kao proces koji omogućava kontrolu konfiguracije aplikacija i infrastrukture, ali obezbjeđuje i uvid u istorijske izmjene, deklarisano (željeno) i trenutno stanje konfiguracije. Osim toga, sam proces je usko povezan sa procesima kao što su upravljanje izmjenama (eng. Change Management) i puštanje u produkciju nove verzije aplikacije.

Efikasno upravljanje konfiguracijom pruža sljedeće prednosti:

- omogućava izmjene i podrazumijeva upotrebu standarda i najboljih praksi;
- značajno smanjuje vrijeme rješavanja incidenata korištenjem centralnog skladišta za konfiguraciju o infrastrukturi kojima mogu pristupiti drugi programi;
- uključuje grupisanje konfiguracija;
- omogućuje i potpomaže ispunjavanje poslovnih ciljeva i zahtijeva korisnika;
- pruža tačne informacije o stanju sistema kako bi se omogućilo ljudima da donose pravovremene odluke. Na primjer, omogućava brže rješavanje incidenata i problema;
- minimizuje broj problema koji mogu nastati zbog nepravilne konfiguracije infrastrukture ili samih aplikacija;
- optimizuje upotrebu softverskih i hardverskih resursa.

Da bi se smanjio broj problema koji se mogu pojaviti u produkcionom okruženju, bilo bi idealno ukoliko bi se aplikacija mogla izvršavati uvijek u istom okruženju ili okruženjima koja imaju konzistentnu konfiguraciju. Iako postoji mnogo alata za upravljanje konfiguracijom, pri čemu se podrazumijeva da se konfiguracija sistema čuva na centralnoj lokaciji, ponekad je jako teško obezbijediti željenu konfiguraciju sistema na velikom broju servera. Problemi mogu nastati jer različiti serveri koriste različite operativne sisteme i hardverske komponente, kao na primjer mrežne kartice, različite biblioteke i slično. Osim toga svi ovi serveri mijenjaju konfiguraciju vremenom, na primjer ažuriranje biblioteka i instalacija različitih softverskih paketa povlači sa sobom odgovarajuće izmjene na disku.

Ukoliko želimo da imamo skalabilno okruženje, trebalo bi da arhitektura sistema podržava dodavanje nove aplikacije u bilo kom trenutku, bez mogućnosti da ta nova aplikacija proizvede konflikt sa bilo kojom drugom. Dakle kontejneri, za razliku od virtuelnih mašina ili fizičkih servera, ne bi trebalo da mijenjaju svoje stanje. Nepromjenjiva infrastruktura (eng. *immutable infrastructure*) podrazumijeva da bilo koji artifakt koji se isporuči jednom (u ovom slučaju aplikativna slika) se ne mijenja više tokom svog životnog ciklusa. Ukoliko se on promijeni iz nekog razloga, on se markira za brisanje i zamjenjuje se novim. Ovdje se prvenstveno misli na artifakt odnosno *Docker* sliku, jer jasno je da će kontejner prilikom izvršavanja vršiti izmjene u okviru okruženja u kojem se izvršava, na primjer generisati log fajlove.

Jedan dio procesa upravljanja konfiguracijom je i proces upravljanja softverskom konfiguracijom, a uloga mu je praćenje i nadgledanje izmjena meta-podataka konfiguracije softverskog sistema. U procesu razvoja softvera, upravljanje konfiguracijom se koristi zajedno sa CI/CD sistemima za kontrolu verzija. Kako se prilikom isporuke softvera koristi isti slika za sva okruženja, konfiguracija softvera je neizostavni dio neophodan da bi se softver mogao isporučiti, inicijalizovati i pustiti u izvršenje odnosno učiniti dostupnim.

Konfiguracija se obično razlikuje od okruženja do okruženja, a kada se govori o mikroservisnoj arhitekturi može da obuhvata:

- specifikaciju resursa neophodnih za izvršenje aplikacije kao što su memorija i procesor;
- tokene ili druge tajne parametre za pristup drugim servisima ili na primjer bazama podataka;
- druge konfiguracione parametre koji određuju vanjske veze sa drugim servisima.

Konfiguracioni parametri bi trebali da se čuvaju u okviru sistema za kontrolu verzija, kako bi se izmjene mogle lakše pratiti i kontrolisati [3]. Međutim, u zavisnosti od samog softvera, za čuvanje dijela konfiguracije se mogu koristiti i eksterni sistemi. Ukoliko govorimo o jednom korporativnom okruženju, u zavisnosti od sigurnosnih politika, čuvanje seritifikata ili drugih tajnih konfiguracionih parametara u okviru sistema za kontrolu verzija, osim što ne zadovoljava sigurnosne standarde, je često nepraktično. Jedan od popularnih sistema za čuvanje lozinki i drugih tajnih parametara je *Hashi Corp Vault*, a upotreba nije ograničena na mikroservisnu arhitekturu. Ukoliko su neki od parametara dinamički, ili se koriste za aktivaciju određenih funkcionalnosti softvera (eng. *feature toggle*), obično se koristi *Hashi Corp Consul*. *Consul* osim što nudi distribuirano KV skladište, se takođe koristi i kao alat za detekciju (eng. *autodiscovery*) i registraciju servisa.

U praksi je uobičajeno da se prilikom razvoja softvera koristi nekoliko aplikativnih okruženja kroz koje se kôd testira kako bi se detektovale potencijalne anomalije u softveru prije nego što se isporuče u produkciju. Iako je, u zavisnosti od načina na koji je razvojni i test proces organizovan unutar jedne organizacije, moguće kreirati više okruženja, to dodatno komplikuje same procese. Generalno, praksa pokazuje da su četiri okruženja dovoljna u većini slučajeva i ukoliko nema nekih posebnih zahtjeva, najbolje se zadržati na tom broju:

- razvojno okruženje (eng. *development*) gdje programeri mogu da testiraju izmjene prije nego što su one odobrene za integraciju i isporuku;
- pretprodukcijsko okruženje (eng. *staging*) koje je obično kopija produkcionog okruženja, sa namjenom da potvrdi da sve izmjene iz prethodnih okruženja rade kako je predviđeno prije nego se učine dostupnim korisnicima;
- produkcijsko okruženje (eng. *production*) okruženje gdje se aplikacija izvršava odnosno pušta u upotrebu i gdje je direktno dostupna krajnjim korisnicima.

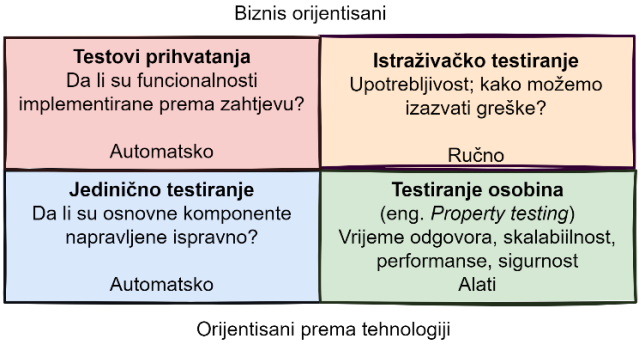
Osim ova tri okruženja, neke kompanije koriste još jedno dodatno QA testno okruženje (eng. *quality assurance environment*). Način organizacije aplikacionih okruženja prvenstveno zavisi od veličine organizacije kao i samog procesa isporuke softvera. Tu se prvenstveno misli na to da li ogranizacija ima poseban tim odgovoran za upravljanje isporukom softvera (eng. *release engineering team*) ili je ta odgovornost dodijeljena razvojnim timovima.

3.6. Testiranje

Testiranje softvera je zasebna disciplina i uzimajući u obzir obim rada, biće ukratko obrađena, kako bi se prezentovao osnovni koncept i uloga u okviru CI/CD sistema. Iako postoje mnogi alati, kao i biblioteke za testiranje, još uvijek je teško pronaći odgovore na pitanje kako pravilno i efektivno testirati mikroservisne aplikacije. Mikroservisna arhitektura dodaje novi sloj kompleksnosti i, da bi se testiranje izvršilo pravilno, efektivno i pouzdano, potrebno je da se razumije koje sve vrste testova postoje i koja je njihova uloga.

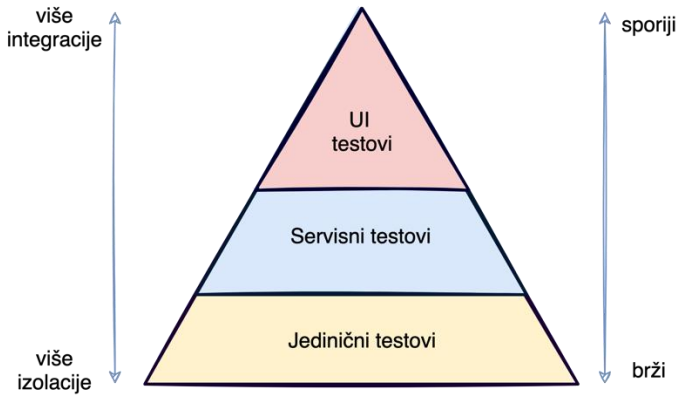
Kao što je već rečeno u prethodnom tekstu, testiranje je sastavni dio jednog CI/CD sistema i obično dio koji je najteže implementirati ispravno. Testovi se po pravilu pokreću prilikom svake izmjene i njihova ispravnost nam daje potvrdu da li je mikroservisna aplikacija spremna za isporuku u produkciju. Uzimajući u obzir da postoji veliki broj različitih vrsta testova, logično pitanje koje se postavlja je, koje od njih implementirati i kako ih postaviti odnosno integrisati unutar same linije za isporuku. Ako se pozovemo na preporuke [91], da su brze povratne informacije veoma bitne, tada se kao razuman izbor nameće da se testovi koji se

brzo izvršavaju prvi i stave na početak CI linije. Testove koji su kompleksniji, i čije izvršavanje traje duže, treba staviti na kraj linije. Ponekad, da bi se poslale brze povratne informacije, moguće je neke vrste testova izvršavati i paralelno, pri čemu se skraćuje vrijeme potrebno za izvršenje linije.



Slika 3.12. Uporedni prikaz vrsta testiranja softvera [94]

Na slici 3.12. prikazana je kategorizacija testova u dvije osnovne grupe [93]. Na donjoj strani slike su testovi koji su više orijentisani ka tehnologiji i koji razvojnim programerima omogućavaju da testiraju izmjene. U ove testove spadaju jedinični testovi (eng. *unit test*) kao i testovi performansi (eng. *performance test*). Ovi testovi se uglavnom brzo izvršavaju i obično su automatizovani. Na gornjoj strani su testovi koji su više orijentisani ka biznis strani odnosno licima bez tehničkog znanja koji bi trebali da potvrde ispravnost funkcionalnosti. Neki od njih, kao na primjer UAT (*User Acceptance Test*), se obično izvršava ručno. Kako bi se omogućila njihova upotreba u okviru CI/CD sistema neophodno je da se svi testovi automatizuju i prilagode tako da se mogu veoma brzo izvršavati.



Slika 3.13. Testna piramida

Međutim, kada se govori o automatizaciji ovog procesa odnosno automatskom testiranju softvera, tu se obično pominje jedan dodatni koncept pod nazivom testna piramida [93] kao što je to prikazano na slici 3.13. Ovaj koncept se često koristi kako bi se razumjelo koje automatizovane testove treba implementirati, odnosno o opsegu koji bi oni trebali obuhvatati.

Testna piramida u izvornom obliku sastoji se od tri sloja od kojih bi trebao da se sastoji jedan testni modul, a to su:

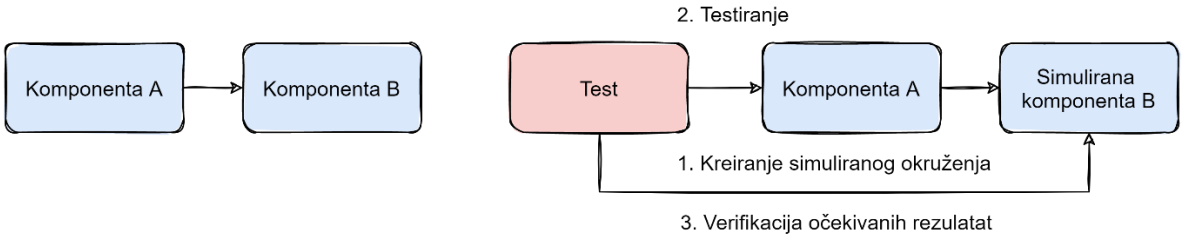
- jedinični testovi,
- servisni testovi (eng. *service test*),
- testovi korisničkog interfejsa (eng. *user interface test*).

Pokazuje se da ova piramida previše pojednostavljuje sam proces testiranja i često može zavarati. Međutim, može da bude dobra polazna osnova ukoliko se kreira novi modul za testiranje mikroservisne aplikacije. Posebno je važno zapamtiti da je potrebno implementirati testove različite granularnosti i namjene, a što se ide više ka gornjem nivou, broj testova bi trebao da bude manji.

3.6.1. Jedinično testiranje

Jedinično testiranje predstavlja testiranje izolovanih komponenata unutar jednog sistema. Komponenta od interesa trebalo bi da posmatramo kao izdvojenu cjelinu koja se može izvući iz konteksta sistema i testirati izolovano od ostalih komponenata [94]. Jedinični testovi osiguravaju da odgovarajuća komponenta unutar repozitorijuma izvornog koda radi kako treba.

Prilikom jediničnog testiranja, komponenta koja se testira se izvlači iz stvarnog sistema u kome komunicira sa ostalim komponentama i stavlja se u test kontekst simuliranog okruženja gdje se nalaze komponente koje simuliraju rad stvarnih komponenata. Na slici 3.14. prikazana je komponenta A koja komunicira sa eksternom komponentom B, što može biti, na primjer, drugi mikroservis ili server base podataka. Prilikom testiranja kreira se simulirana verzija komponente B (eng. *mocking* i *stubbing*) tako da se podese očekivani rezultati i onda se komunicira sa njom umjesto sa eksternim servisom. Jednostavnim rječnikom to podrazumijeva da se klase, moduli, funkcije, a nekad i cijeli dijelovi jednog sistema zamjenjuju lažnim (eng. *fake*) komponentama čije se ponašanje može kontrolisati. U ovom slučaju jedinični test osigurava da je komponenta A potpuno funkcionalna i za takav test nije potrebna dostupnost eksternog mikroservisa ili konfigurisan server baze podataka. Od jediničnih testova se očekuje da se brzo izvršavaju, a upotrebom simuliranog okruženja se izbjegava potreba za pozivanje eksternih servisa ili komunikacija sa serverom baze podataka čime se skraćuje vrijeme potrebno za izvršavanje.



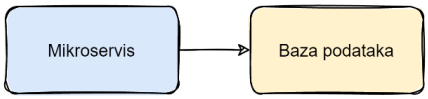
Slika 3.14. Jedinično testiranje izolovane komponente mikroservisne aplikacije

Prednosti ovakvog načina testiranja su što se testiranje može obaviti na nivou jedne komponente sistema, pri čemu se osigurava ispravnost komponente koja se integriše u postojeći sistem. U slučaju da se detektuje problem u mikroservisnoj aplikaciji koja se sastoji od više komponenata, gdje se ne koristi jedinično testiranje, pronalazak problematične komponente može biti komplikovano.

3.6.2. Integraciono testiranje

Svaka malo kompleksnija mikroservisna aplikacija će se integrisati sa nekim eksternim sistemima, pri čemu to mogu biti baze podataka, drugi mikroservisi i slično. Da bi se jedinični testovi brzo izvršavali, kao što je već rečeno, rad ovakvih sistema se simulira. Ipak, kako će jedna mikroservisna aplikacija komunicirati sa eksternim komponentama, potrebno ih je testirati i to sa realnim komponentama. Zbog toga se integraciono testiranje vrši kao logičan nastavak jediničnog testiranja. Kako bi se izbjegla zabuna u vezi sa prethodno navedenom

piramidom testiranja važno je napomenuti da se integraciono testiranje često naziva testiranje komponenata kao i servisno testiranje. Na slici 3.15. je prikazan primjer testiranja integracije sa realnom bazom podataka.

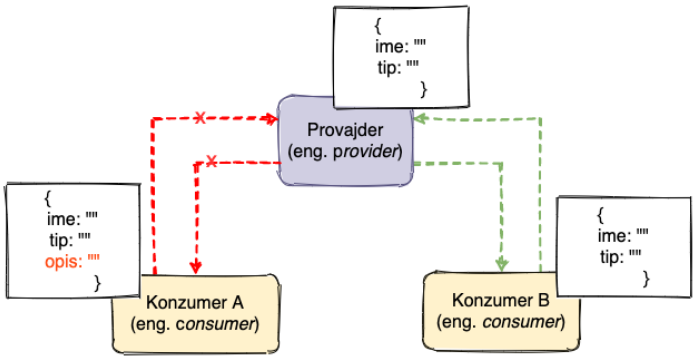


Slika 3.15. Integracioni test sa bazom podataka testira rad izvornog koda mikroservisne aplikacije sa realnom bazom podataka

Da bi se pravilno izvršilo integraciono testiranje, mikroservisna aplikacija se mora zapravo pokrenuti, a umjesto simuliranih komponenata koristiti prave. To znači da umjesto simulacije baze podataka, moramo kreirati bazu podataka i pokrenuti server baze podataka, povezati se i izvršiti odgovarajuće akcije. Ukoliko se radi o čitanju sa fajl sistema to znači da moramo kreirati pravi fajl na fajl sistemu i onda ga pročitati da bi smo verifikovali integraciju. Dobra praksa je da se integraciono testiranje vrši tako da se testira integracija sa jednom po jednom komponentom zasebno, u manjim cjelinama pa da se zatim dodaju ostale komponente.

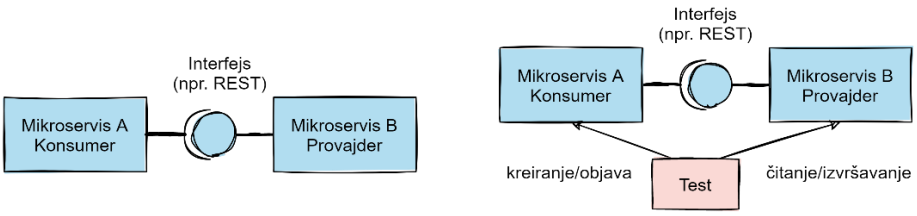
3.6.3. Testiranje interfejsa

Kao što je već rečeno, upotreba mikroservisne arhitekture omogućava timovima da rade nezavisno na mikroservisima koji nisu čvrsto povezani (eng. *loosely coupled*). Kako su ovi mikroservisi obično samo komponente jedne mikroservisne aplikacije oni trebaju da komuniciraju preko unaprijed definisanih interfejsa.



Slika 3.16. Testiranje interfejsa između provajdera i konzumera

Interfejsi mogu da budu implementirani upotrebom različitih tehnologija. Implementacija se često zasniva na upotrebi REST-a u kombinaciji sa JSON-om (*JavaScript Object Notation*) preko HTTPS protokola ili RPC (eng. *Remote Procedure Call*) u kombinaciji sa gRPC ili JSON RPC. Ovaj test obično pišu svi timovi, a svrha mu je da utvrdi da li implementirani interfejsi odgovaraju specifikacijama. Osim toga, koristi se kako bi utvrdio uticaj izmjena koje vrše timovi odgovorni za određene mikroservise odnosno provajderi (eng. *providers*) na konzumere odnosno korisnike koji koriste ove mikroservise (eng. *consumers*).

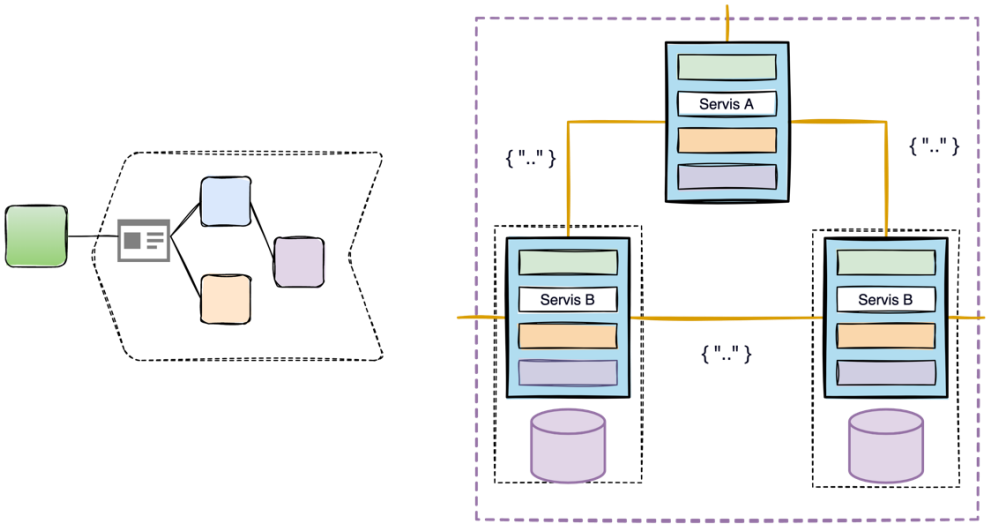


Slika 3.17. Testiranje interfejsa

3.6.4. Testiranje sa kraja na kraj

Testiranje sa kraja na kraj (eng. *end-to-end testing*) treba da pokrije testiranje kompletne aplikacije. Ovi testovi su najpouzdaniji za potvrdu da li aplikacija radi kako se očekuje ili ne. Oni često podrazumijevaju testiranje korisničkog interfejsa, ukoliko ga aplikacija posjeduje, i simulaciju stvarnog korisnika. U teoriji bi trebali simulirati stvarnog korisnika aplikacije ili bar izvršiti akcije stvarnog korisnika. U praksi je ove testove obično najteže napisati i potrebno je potrošiti najviše vremena za njihov razvoj.

Ukoliko se radi o mikroservisnoj arhitekturi, u tom slučaju je potrebno testirati komunikaciju sa svim eksternim mikroservisima. Važno je napomenuti da ovi testovi ne trebaju da pokrivaju testiranje interne biznis logike drugih mikroservisa, već samo komunikaciju sa njima. Kako se jedna mikroservisna aplikacija sastoji od više mikroservisa, ovi testovi obezbjeđuju potvrdu ispravnosti komunikacije između više mikroservisa.



Slika 3.18. Testiranje sa kraja na kraj

Ovi testovi utiču na razvoj mikroservisne arhitekture pa pisanje ovakvih testova zahtijeva poznavanje domena problema koje konkretna aplikacija rješava. Tako, na primjer, ukoliko dođe do promjene arhitekture mikroservisne aplikacije, na primjer prilikom spajanja dva mikroservisa u jedan ili razdvajanja postojećeg u više, ovi testovi mogu da potvrde da je poslovna logika i dalje funkcionalna odnosno ostala nepromijenjena.

Testovi sa kraja na kraj imaju veoma veliki broj nedostataka koji znatno rastu sa brojem sistema koji se testiraju. Praksa pokazuje da su kompanije, koje su koristile ove testove za mikroservisne aplikacije, uglavnom napustile ovaj vid testiranja i okrenule se drugim oblicima testiranja koji djelimično pokrivaju ili u potpunosti zamjenjuju ovaj vid testiranja [1].

3.6.5. Testiranje konfiguracije

Testiranje konfiguracije je takođe jedan bitan korak CI/CD linije. Ovi testovi se obično pišu kako bi se validirali konfiguracioni parametri na osnovu strukturiranih podataka o konfiguraciji. Osnovna namjena je da se programerima obezbijede pravovremene povratne informacije i stopira izvršenje CI/CD linije u ranoj fazi sa ciljem uštede resursa. Pretpostavimo, na primjer, da su uokviru *Kubernetes* platforme konfigurisane sigurnosne polise tako da je onemogućeno izvršavanje privilegovanih kontejnera. Ukoliko programer

konfiguriše svoj mikroservis (kontejner) tako da se izvršava u privilegovanom modu, samo kreiranje *Pod*-a neće biti moguće. Nakon što se ova aplikacija isporuči u određeno okruženje, ona neće proći test ispravnosti. Programer će morati da provede neko vrijeme analizirajući logove *Kubernetes* sistema da bi detektovao problem, a sama CI/CD linija će se nepotrebno izvršiti do kraja pri čemu će vrijeme biti nepovratno izgubljeno.

Ovakvi testovi unutar linije mogu se koristiti da se zabranila upotreba posljednje verzije (*latest*) *Docker* slike i osiguralo da se uvijek koristi specifična verzija. Ovim će se spriječiti automatska promjena verzije mikroservisa prilikom kreiranja novog *Pod*-a u slučaju da se u međuvremenu u repozitorijum smjesti nova verzija *Docker* slike označena kao *latest*. Samim tim se sprečavaju eventualne greške i problemi koji mogu nastati u ovakvim situacijama.

3.7. Progresivna isporuka

Težnja organizacija da unaprijede svoje poslovanje i budu konkurente na tržištu, bez obzira na njihovu veličinu, u direktnoj je vezi za brzinom isporuke novih izmjena odnosno funkcionalnosti. Kao rezultat često se opredjeljuju za implementaciju CI/CD sistema. Kod tradicionalnog načina isporuke aplikacija, u okruženjima koja nemaju implementiran CI/CD sistem, nakon što se nova verzija aplikacije isporuči u okruženje, ona je automatski dostupna svim korisnicima. Međutim, to sa sobom nosi određeni rizik jer se aplikacija istovremeno isporučuje svim korisnicima. Bez obzira na to koliko je testova implementirano u okviru CI linije, često se može desiti greška prilikom isporuke aplikacije u produkciono okruženje. Ova greška ne mora direktno da bude povezana sa greškom u kodu.

Praksa pokazuje da organizacije koje imaju implementiran CDE proces, uviđaju da kontinualna isporuka softvera i dalje predstavlja izazov bez obzira na implementiran CI/CD sistem. Ovdje se naročito misli na sam proces aktivacije nove verzije mikroservisne aplikacije u produkcionom okruženju. One često idu korak dalje i uvode jedan dodatni proces koji se naziva progresivna isporuka (eng. *progressive delivery*). Progresivna isporuka unapređuje i poboljšava ovaj korak, izdvajajući ga od same isporuke, što omogućava da se potencijalna šteta, prouzrokovana lošom verzijom aplikacije ili greškom u konfiguraciji, ograniči na mnogo manju grupu umjesto na cijelu bazu korisnika [97]. Timovima se daje veća fleksibilnost, jer mogu da odluče kolika velika ta inicijalna grupa korisnika treba da bude. Uglavnom su svi koncepti slični i kreće se od manjih grupa, a onda se nova verzija progresivno isporučuje (učini dostupnom) većim grupama, čime se sam uticaj promjene može procijeniti.

Kako su osnova za implementaciju progresivne isporuke strategije za testiranje i uvođenje u eksploataciju, u narednoj glavi dat je pregled najčešće korištenih strategija.

4. STRATEGIJE ZA TESTIRANJE I UVOĐENJE U EKSPLOATACIJU

Prilikom uvođenja nove verzije mikroservisa u eksploataciju unutar jednog okruženja, nova verzija nije istovremeno dostupna svim korisnicima. U zavisnosti od načina na koji je implementiran proces, tek nakon što se uspješno završe sve akcije predviđene strategijom isporuke i mikroservis postane dostupan svim korisnicima, smatra se da je verzija realizovana odnosno aktivirana. ITIL-ova definicija opisuje isporuku i realizaciju kao akcije koje imaju za cilj planiranje, zakazivanje i kontrolu kretanja verzije softvera kroz različita softverska okruženja [95]. Primarni cilj ovih aktivnosti je osigurati i zaštiti integritet softverskog okruženja tako što će obezbijediti da se tačno odgovarajuće verzije komponenata aktiviraju odnosno učine dostupnim.

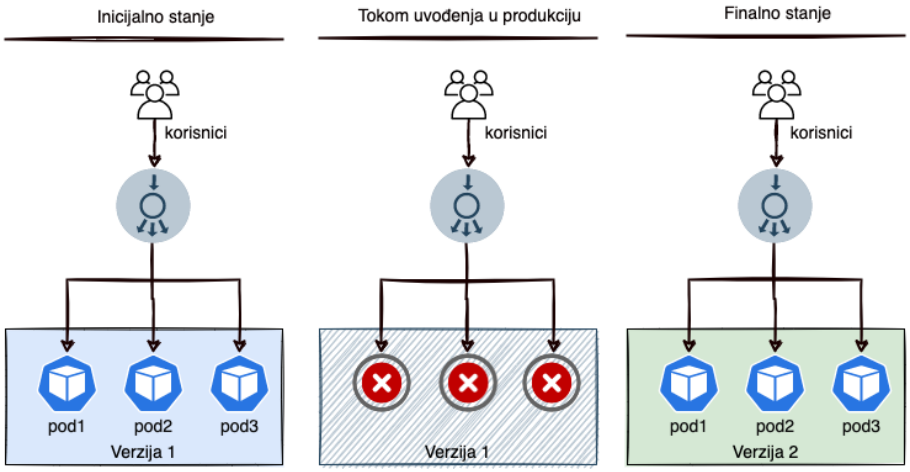
Da bi se ovaj cilj uspješno realizovao, postoje različite strategije za testiranje i uvođenje u eksploataciju, od kojih se većina zasniva na kreiranju više različitih verzija istog mikroservisa paralelno. U zavisnosti od toga koja strategija je izabrana, prilikom isporuke nove vezije mikroservisa, moguće je na određen način kontrolisati koja će verzija servisa (nova ili postojeća) da bude aktivna, odnosno servisira korisnički zahtjev. U narednom tekstu dat je kratak pregled nekoliko najčešće upotrebljavanih strategija koje se koriste prilikom automatizacije procesa za uvođenje u eksploataciju. Pored osnovnog pregleda, dat je i kratak pregled njihovih prednosti i nedostataka, a koji pristup je najbolji zavisi od konkretnog slučaja.

4.1. Strategije za uvođenje u eksploataciju

Strategije za uvođenje u eksploataciju, koje su biti obrađene u ovom poglavlju, omogućavaju fleksibilnost i automatizaciju procesa prilikom uvođenja nove verzije u eksploataciju. Postoji nekoliko pristupa, a koji od njih upotrijebiti zavisi od zahtjeva i konkretnog cilja.

4.1.1. Strategija ponovnog kreiranja

Prilikom upotrebe strategije ponovnog kreiranja (eng. *recreate deployment pattern*) vrši se kompletno rekreiranje mikroservisa, tako što se trenutna verzija mikroservisa ukloni i kreira mikroservis nove verzije. Na sljedećoj slici prikazan je dijagram koji prikazuje primjenu ove strategije na jednom mikroservisu.



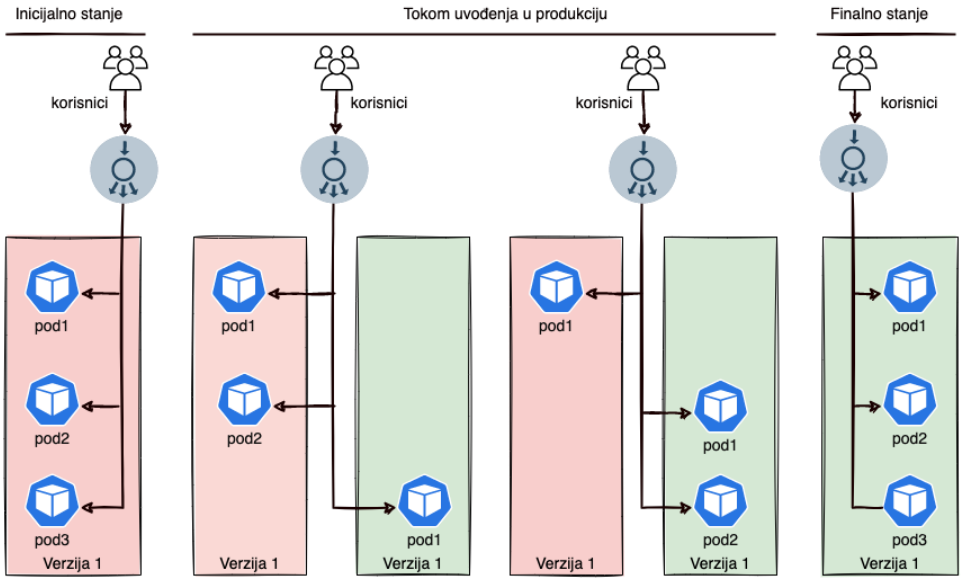
Slika 4.1. Prikaz faza isporuke softvera upotrebom strategije ponovnog kreiranja [99]

Osnovna prednost ovog pristupa je njegova jednostavnost i nisu potrebni dodatni resursi da bi se sam proces implementirao. Pored toga nije potrebno da postoji više verzija istog mikroservisa paralelno a samim tim se izbjegavaju izazovi kompatibilnosti sa prethodnim verzijama kako za bazu podataka tako i za sam mikroservis.

Ukoliko se koristi ovaj pristup treba voditi računa da on podrazumijeva nedostupnost aplikacije tokom samog procesa uvođenja u eksploataciju. To naravno nije problem sa mikroservisima čije se verzije mogu isporučivati u prethodno najavljenim i odobrenim vremenskim intervalima. Međutim, ukoliko se radi o kritičnim mikroservisima sa visokim nivoom dostupnosti, u tom slučaju neka druga strategija se nameće kao logičan izbor.

4.1.2. Postepena zamjena

Ova strategija podrazumijeva da nova verzija aplikacije postepeno zamjenjuje (eng. *rolling updates*) prethodnu u određenom vremenskom periodu. Tokom trajanja tog vremenskog intervala obe verzije egzistiraju nezavisno jedna od druge, bez uticaja na korisničko iskustvo. Ovaj proces omogućava da se veoma brzo i lako odradi povrat na prethodnu verziju bilo koje komponente koja nije kompatibilna sa prethodnom verzijom mikroservisa.



Slika 4.2. Prikaz faza isporuke softvera upotrebom strategije postepene zamjene [99]

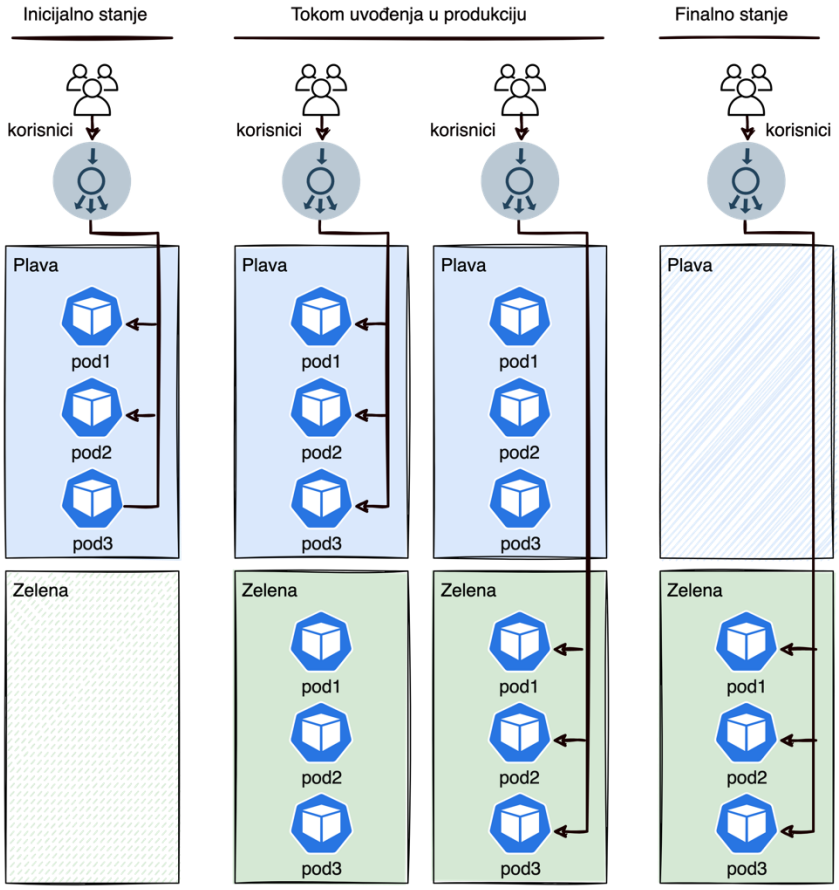
Kubernetes platforma ima ugrađen ovaj mehanizam kao primarni mehanizam u slučaju da se izvrši izmjena *Deployment* objekta, što inicira kreiranje novog *ReplicaSet* objekta za novu verziju mikroservisa. Nakon toga, odgovarajući kontroler u okviru platforme, izvršava određene operacije kako bi sistem doveo u željeno stanje. Kontroler ažurira postojeći i novi *ReplicaSet* objekt, čime se vrši postepreno kreiranje novih *Pod*-ova, dok se stari *Pod*-ovi skaliraju prema nuli odnosno brišu. Kontroler ovdje ima primarnu ulogu, jer obezbjeđuje da se stari *Pod*-ovi ne brišu dok novi nisu spremni za serviranje zahtijeva. Pored toga, *Kubernetes* ima kompletnu istoriju izmjena, tako da se vrlo lako može izvršiti povrat na prethodnu verziju.

Jedan od izazova, koje ova strategija donosi sa sobom, je da su tokom trajanja procesa puštanja u eksploataciju nove verzije obe verzije mikroservisa su dostupne i opslužuju zahtjeve. Tokom tog perioda zahtjev može biti proslijeđen na bilo koju verziju mikroservisa. U slučaju

da se rade veće izmjene koje zahtijevaju izmjenu na klijentskoj strani, treba voditi računa da su obje verzije podržane, dok se prethodna verzija ne deaktivira.

4.1.3. Plavo-zelena isporuka

Plavo-zelena strategija (eng. *blue-green*) isporuke je način isporuke pri kom se nova verzija softvera isporučuje paralelno sa prethodnom verzijom. Nakon što se izvrše sva potrebna testiranja i validacija nove verzije, saobraćaj se preusmjerava sa trenutne na novu verziju. Nakon preusmjerenja saobraćaja, vrši se monitoring nove verzije sa ciljem detekcije eventualnih problema. Ukoliko se utvrdi da nova verzija funkcioniše bez problema, stara verzija se može obrisati.



Slika 4.3. Prikaz faza isporuke softvera upotrebom plavo-zelene strategije [99]

Upotrebom ove strategije dobija se mogućnost vrlo lakog povrata na prethodnu verziju preusmjeravanjem odnosno prerutiranjem zahtjeva na prethodnu (plavu verziju). Ukoliko želimo upotrijebiti ovu strategiju za isporuku softvera u tradicionalnom monolitnom okruženju, to bi teoretski značilo kreiranje dva identična okruženja. Nova verzija bi se isporučivala u jedno okruženje i testirala te nakon toga bi se na nivou usmjerivača opterećenja sav saobraćaj preusmjeravao sa prethodne na novu verziju.

Jedan od koristi koju donosi mikroservisna arhitektura je da se isporuka nove verzije obično vrši na nivou mikroservisa. To podrazumijeva da se izmjena vrši u istom okruženju, a aktivacija i preusmjeravanje saobraćaja na novu verziju vrši se isključivo upotrebom već ugrađenih mehanizama unutar same platforme za orkestraciju. To znači da ukoliko želimo imati više okruženja kako bismo koristili neku od strategija, nije potrebno kreiranje novih serverskih klastera. Dakle, potrebno je da se kreira novi *Deployment* objekat koji u specifikaciji

sadrži novu verziju aplikacije kao i novi set labela. Nakon što se kreira novi *Deployment* objekat bez brisanja prethodnog, i kontejneri budu spremni, manipulacijom selektorima na samom servisu svi dolazni zahtjevi se mogu preusmjeriti na novu verziju.

Upravljanje izmjenama na bazi podataka sa ovom tehnikom često može biti izazov, posebno ako nova verzija mikroservisa zahtijeva i izmjenu na bazi podataka da bi ispravno funkcionisala. Jedan od pristupa je da se razdvoji isporuka nove verzije aplikacije od izmjena na bazi. Nakon toga se izmjene dizajniraju i vrše tako da baza podržava obe verzije mikroservisa.

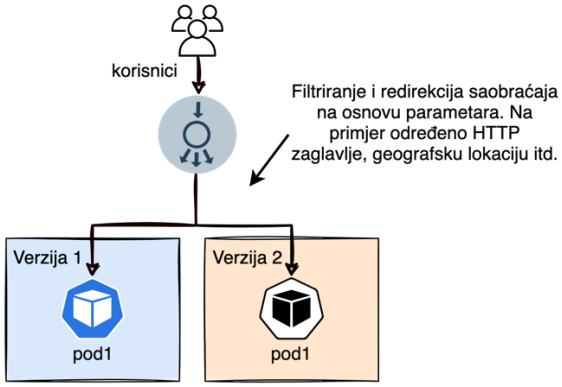
Osim toga, treba imati na umu da će prilikom trajanja isporuke biti startovano duplo više *Pod*-ova odnosno kontejnera. U slučaju da mikroservis zahtijeva mnogo resursa može postojati potreba za privremenim skaliranjem klastera kako bi se obezbijedili neophodni resursi.

4.2. Strategije za testiranje prilikom uvođenja u eksploataciju

Strategije za testiranje prilikom uvođenja u eksploataciju imaju istu namjenu kao i strategije za puštanje u eksploataciju. Međutim, njihova primarna namjena je testiranje novih verzija softvera koje prvenstveno donose nove funkcionalnosti. Dodatno, one omogućavaju testiranje i analizu u produkcionom okruženju i sa realnim zahtjevima za razliku od standardnog testiranja koje podrazumijeva testiranje sa unaprijed definisanim skupom podataka.

4.2.1. A/B testiranje

A/B testiranje je strategija koja se bazira na odlukama donesenim na osnovu statistika odnosno na osnovu podataka. Osnovna ideja je da se zahtjevi distribuiraju na osnovu prethodno definisanih pravila samo određenim klijentima, kao što je to prikazano na slici 4.4.



Slika 4.4. Prikaz faza testiranja upotrebom A/B testiranja [99]

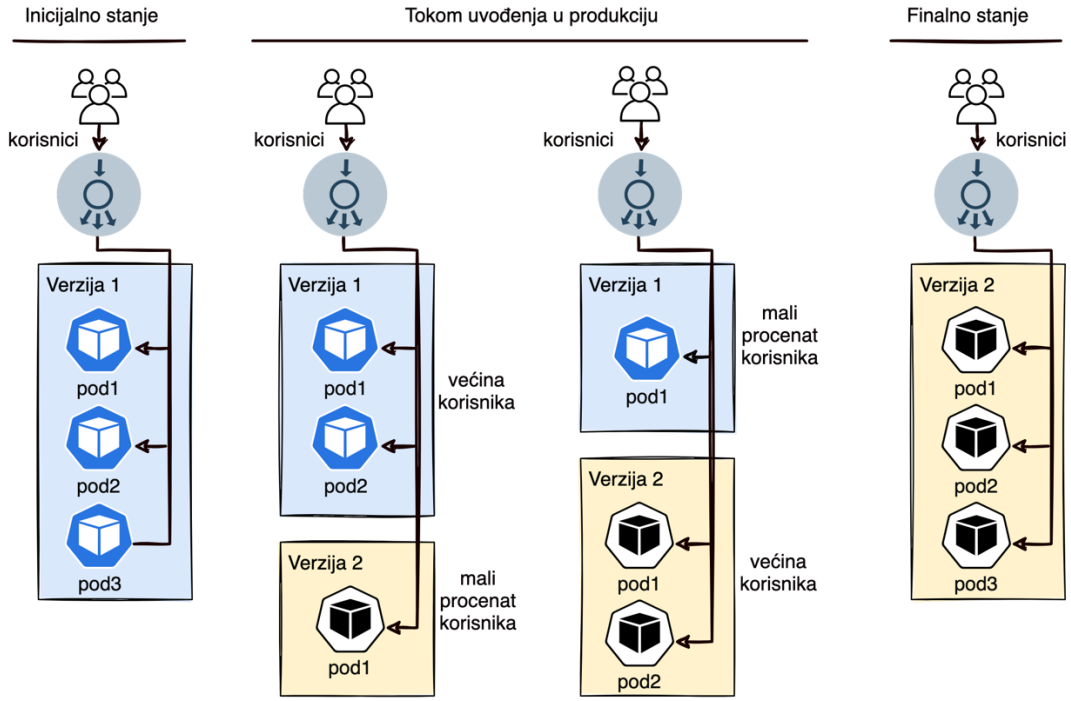
Ova strategija je najbolja za mjerenje performansi kao i funkcionalnosti mikroservisa. Dok se prethodno opisane strategije uglavnom fokusiraju na sigurno uvođenje u eksploataciju nove verzije kao i brzog povratka na prethodnu verziju, primarna namjena ove strategije je monitoring i provjera ispravnosti novih funkcionalnosti u odnosu na prethodnu verziju. Osim toga, ova strategija omogućava komparaciju performansi i njihovu analizu, jer nove verzije softvera mogu znatno da utiču na performanse.

Implementacija ove strategije je veoma kompleksna, jer je potreban odgovarajući uzorak podataka koji sa sigurnošću može da potvrdi da je jedna verzija bolja od druge. Potrebno je unaprijed odrediti neophodan broj zahtjeva, zatim vršiti testiranje određen vremenski period

da bi se mogla odraditi statistička analiza, nakon čega se vrši analiza rezultata. Ukoliko se vrši više A/B testiranja za isti mikroservis istovremeno, sam proces monitoringa i analize može biti veoma komplikovan.

4.2.2. Kanarinac strategija

Kanarinac strategija (eng. *Canary strategy)* je veoma slična prethodno opisanoj plavo-zelenoj strategiji. Ona smanjuje rizik prilikom uvođenja nove verzije u eksploataciju tako što se nova verzija isporučuje veoma malom broju korisnika prije nego što se učini dostupnom svim korisnicima. Nakon toga vrši se monitoring i prati ponašanje nove verzije. Ova tehnika omogućava da se puštanje u eksploataciju obavi u kontrolisanim koracima, upotrebom realnih podataka kao i detekciju problema prije nego oni zahvate sve korisnike. Ova tehnika je mnogo kompleksnija jer zahtijeva dinamičko rutiranje prema različitim verzijama mikroservisa, a pored toga neophodan je izuzetno kvalitetan monitoring sistem.



Slika 4.5. Prikaz faza testiranja upotrebom kanarinac strategije [99]

Jedna od prednosti ove strategije u odnosu na druge je što se testiranje vrši u produkcionom okruženju, umjesto da se testiranje vrši nad simuliranim podacima u testnom okruženju. Prilikom implementacije ove strategije treba voditi računa i jasno definisati korake odnosno plan na koji način i kada će se vršiti inkrementalno povećanje redirekcije korisničkih zahtjeva na novu verziju.

Veoma važan aspekt je implementirati robusan monitoring sistem kako bi se jasno mogli detektovati eventualni problemi, što često iziskuje mnogo resursa, kako ljudskih tako i infrastrukturnih. Prednosti ove strategije su što se veoma lako može aktivirati prethodna verzija mikroservisa jednostavnom redirekcijom saobraćaja. Takođe, ova strategija omogućava testiranje različitih verzija bez prekida u serviranju zahtjeva.

Glavni nedostatak kod upotrebe strategije kanarinca je da se mora upravljati sa više verzija mikroservisa istovremeno. U zavisnosti od internih procedura za realizaciju nove verzije, taj broj može biti veći od dva, ali praksa pokazuje da je broj različitih verzija istog

mikroservisa istovremeno u produkciji najbolje svesti na minimum. Sam proces aktivacije nekad može biti veoma spor i potrajati nekoliko sati u slučaju smanjenog saobraćaja, jer je potrebno prikupiti podatke unutar monitoring sistema da bi se potvrdila ispravnost verzije.

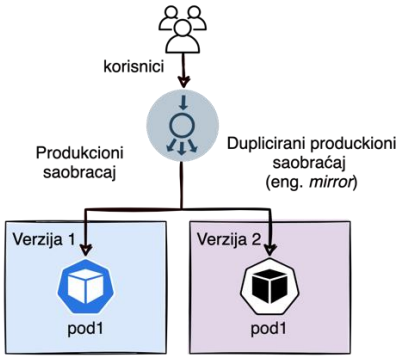
Još jedna stvar kod koje treba biti veoma oprezan je upravljanje verzijama baze podataka kao i omogućavanjem novih funkcionalnosti prilikom isporuke nove verzije. Međutim, upotrebom različitih metodologija i šablona ti problemi se mogu vješto izbjeći. Ovdje se prvenstveno misli na kompatibilnost između verzija (eng. *backward and forward compatibility*) čime se obezbjeđuje da obje verzije prilikom isporuke koriste istu bazu podataka.

Ova strategija može se koristiti za implementaciju A/B testiranja, jer je njihova tehnička implementacija veoma slična. Sa druge strane, treba postaviti jasnu razliku između ove dvije strategije. Dok je upotreba strategije kanarinca dobar način za detekciju problema i regresiju, A/B testiranje se više koristi za analizu različitih verzija. Praktično to bi značilo da za A/B prikupljanje podataka za statističku analizu može potrajati danima, dok se isporuka i aktivacija nove verzije mikroservisa upotrebom strategije kanarinca obično završi za par minuta.

4.2.3. Strategija sjenke

Upotrebom različitih strategija, kao na primjer strategije kanarinca, moguće je izložiti korisnike inferiornoj verziji aplikacije u toku ranih faza testiranja. Ovaj rizik se može djelimično kontrolisati upotrebom oflajn tehnika i testiranjem unaprijed predefinisanim podacima. Međutim, ove tehnike ne omogućavaju validaciju performansi i unapređenja jer nema interakcije korisnika sa novom verzijom mikroservisa.

Strategija sjenke (eng. *shadow test pattern*) podrazumijeva da se nova verzija servisa izvršava u drugom okruženju, pri čemu je nova verzija sakrivena od korisnika kao što je to prikazano na slici 4.6.



Slika 4.6. Prikaz faza testiranja upotrebom strategije sjenke [99]

Svaki dolazni korisnički zahtjev se replicira i na novu verziju. Ovo se može obavljati u realnom vremenu, ali je moguće snimiti produkcioni saobraćaj i kasnije ga primijeniti prilikom testiranja nove verzije mikroservisa.

Osnovna prednost ove strategije je da teoretski nema uticaja na produkciono okruženje. Sav saobraćaj je dupliciran i testiranje se izvodi na izolovanom okruženju. Upotrebom produkcionog saobraćaja i podataka omogućava se testiranje funkcionalnosti, eventualnih grešaka, performansi i rezultati se mogu uporediti sa produkcionim. Ova strategija se obično koristi u kombinaciji sa drugim strategijama, često sa strategijom kanarinca. Prvo se vrši testiranje nove funkcionalnosti upotrebom strategije sjenke, a potom se upotrebom strategije

kanarinca testira korisničko iskustvo tako što se nova verzija granuralno aktivira za određen broj korisnika. Time se obezbjeđuje da se nova verzija ne aktivira u produkcionom okruženju dok nisu zadovoljeni svi uslovi koji obezbjeđuju stabilnost i performanse nove verzije.

Prilikom testiranja treba voditi računa da testiranje ne izvrši izmjene na produkcionom okruženju odnosno izbjeći bilo kakve izmjene korisničkih podataka unutar samog produkcionog okruženja. Na primjer, ako se radi testiranje mikroservisa za procesiranje plaćanja, treba voditi računa da se te korisničke transakcije ne dupliraju. Iz ovoga se jasno vidi da je ova strategija veoma kompleksna za implementaciju i donosi sa sobom mnogo operativnih troškova usljed održavanja dva identična okruženja paralelno.

4.3. Preporuke i smjernice

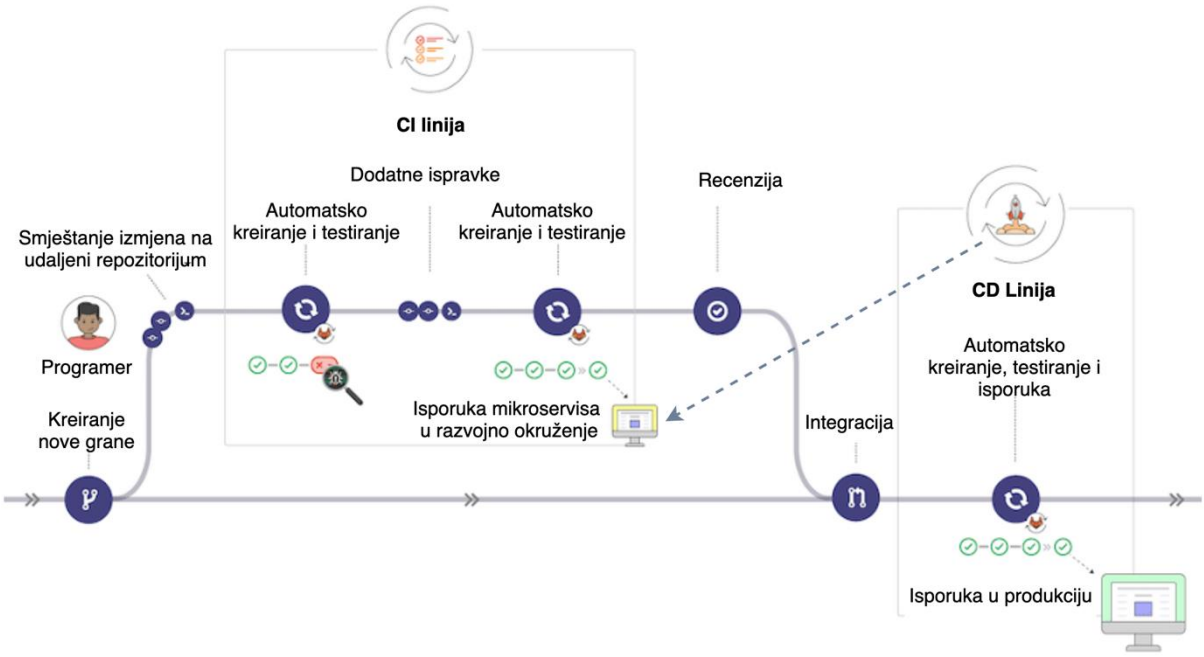
Postoji više strategija i svaka od njih ima svoje prednosti i nedostatke, a prilikom njihovih implementacija najvažniji faktori o kojima treba voditi računa su resursi kao i vrijeme potrebno da bi se neka od ovih strategija implementirala. Međutim, platforme za orkestraciju mikroservisnih arhitektura, od kojih je jedna i *Kubernetes,* nastale su sa ciljem da olakšaju i omoguće svakoj kompaniji da implementira neku od ovih strategija integrišući je u sam proces puštanja nove verzije u produkciju ili da koristi istovremeno više strategija u zavisnosti od konkretnog slučaja i specifičnih zahtjeva. Tako, na primjer, minimalne izmjene mogu da se isporuče upotrebom strategije postepene zamjene, dok se za velike izmjene obično koristi plavo-zelena isporuka. Nove funkcionalnosti se obično realizuju upotrebom A/B testiranja dok se za testiranje integracije sa novim provajderima, odnosno drugim mikroservisima, obično koristi strategija sjenke.

Postoji više načina da se nova verzija mikroservisa isporuči u određeno okruženje i, ukoliko se radi o razvojnom ili testnom okruženju, strategija ponovnog kreiranja je obično dobar izbor, jer ova okruženja ne servisiraju korisničke zahtjeve i privremena nedostupnost neće prouzrokovati probleme. Kada se radi o produkcionom okruženju, plavo-zelena isporuka se često koristi, ali treba imati na umu da je odgovarajuće testiranje veoma bitno. Ukoliko nova verzija može da izazove nestabilnosti i nije sigurno kakav uticaj na okruženje može imati, onda se može koristiti strategija kanarinca, čime se omogućava korisnicima da testiraju aplikaciju uz manje potencijalne negativne efekte. U slučaju da je potrebno testirati novu funkcionalnost na određenoj grupi korisnika (na primjer, svi korisnički zahtjevi koji dolaze preko mobilnog telefona treba da budu proslijeđeni na verziju A, dok ostali treba da budu proslijeđeni na verziju B) kao logičan izbor nameće se A/B strategija testiranja.

5. PRIJEDLOG RJEŠENJA ZA CI/CD

U ovom poglavlju biće detaljno opisan prijedlog rješenja za implementaciju CI/CD sistema, dok će detaljan opis implementacije biti dat u narednoj glavi.

CI/CD linija se sastoji od različitih komponenata, prije svega alata i servisa, koji funkcionišu kao jedna cjelina, obezbjeđujući efikasnu i pouzdanu isporuku softvera uz visok nivo frekventnosti. Kako se CI/CD linija sastoji od dva logička dijela, tako se i sam proces implementacije obično sastoji od dva dijela. Prvi dio predstavlja CI linija čija je uloga da izvrši sve neophodne testove, kreira neophodne artifakte osiguravajući očuvanje kvaliteta koda i glavnu granu u zdravom stanju. Sa druge strane, CD linija je zadužena za isporuku i aktivaciju određene verzije mikroservisa u određeno okruženje. U konkretnom slučaju, kako je fokus rada na mikroservisnoj arhitekturi, CD linija je zadužena za komunikaciju sa mikroservisnom platformom i orkestraciju isporuke.



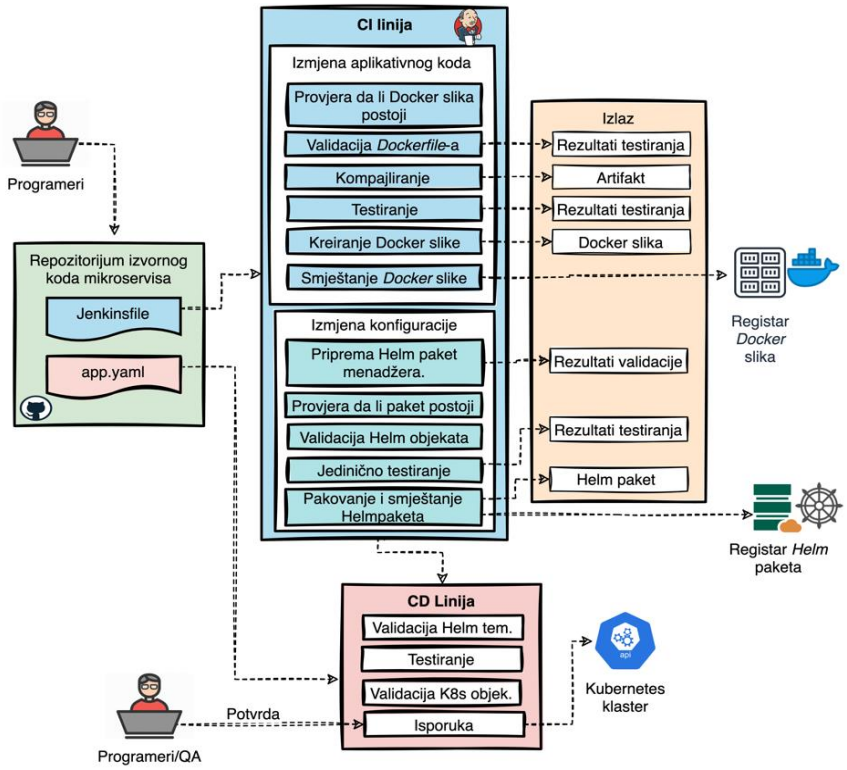
Slika 5.1. Uopšteni prikaz CI/CD sistema

Postoji više alata za kontinualnu integraciju koji uglavnom dolaze sa velikim brojem dodataka za lakšu integraciju sa eksternim sistemima, koji nisu obavezno sastavni dio CI sistema, ali su neophodni da bi se CI proces uspostavio. Tu se prije svega misli na podršku za autentikaciju i autorizaciju, integraciju sa sistemima za kontrolu verzije, registrima za smještanje artifakata kao i orkestratorima za mikroservisne aplikacije. Međutim, bez obzira na to koji alat se odabere, sam proces implementacije je veoma sličan i zasniva se na preslikavanju unaprijed dizajniranih koraka u odgovarajuću liniju. U ovom radu za implementaciju CI linije je izabran *Jenkins* CI alat kao jedan od najpopularnijih alata otvorenog koda za ovu namjenu [98][100][101]. *Docker* je izabran kao mehanizam odnosno alat za kontejnerizaciju, a sam proces kreiranja *Docker* slika je zasnovan na upotrebi *Dockerfile*-a.

Kao alat za instalaciju i konfigurisanje mikroservisnih aplikacija i jedna od glavnih komponenata CD linije izabran je *Helm* paket menadžer [102]. Osim ovog alata, kao dio praktičnog rada implementirana je CLI aplikacija čija je primarna uloga orkestracija isporuke. *Docker Hub* je izabran za registar *Docker* slika, dok je *ChartMuseum* [103] konfigurisan i

korišten kao registar za skladištenje *Helm* paketa. *Git* je izabran kao alat za kontrolu verzije odnosno upravljanje izvornim kodom. Centralni repozitorijumi izvornog koda su smješteni na *Github* [104] serveru. Osim prethodno nabrojanih, nekoliko drugih alata se koristi za validaciju sintakse, testiranje konfiguracije okruženja, validaciju *Kubernetes* manifest fajlova i oni će biti ukratko opisani kasnije.

Teoretski bilo bi idealno kad bi se sve izmjene u izvornom kodu i konfiguraciji mikroservisne aplikacije mogle tretirati na isti način, i da se koristi ista linija za njihovu isporuku [73]. U praksi to nije moguće jer artifakti i konfiguracioni parametri imaju različit životni ciklus. Sa druge strane, CI/CD linije za različite mikroservisne aplikacije su veoma slične i bilo bi dobro ako bi se taj dio koda izdvojio u eksterni repozitorijum izvornog koda koji bi se mogao referencirati i po potrebi koristiti kao eksterna biblioteka. To bi omogućilo da se implementacija CI/CD linija centralizuje, umjesto da svaka mikroservisna aplikacija odnosno timovi za njihovo održavanje kreiraju zasebne CI/CD linije. Za implementaciju je izabran sličan koncept zasnovan na upotrebi *Jenkins* dijeljene biblioteke (JSL – *Jenkins Shared Library*) i omogućava da više linija koristi klase kao i globalne varijable definisane u jednoj takvoj biblioteci.



Slika 5.2. Arhitektura CI/CD linije

Šabloni za dizajniranje CD linija zasnovani na praksi, preporučuju da se dijeljena logika izdvoji od same linije pa je slijedeći tu preporuku ona izdvojena u CLI aplikaciju [105]. CLI aplikacija je napisana u *Go* programskom jeziku i zadužena je za upravljanje isporukom aplikacija u *Kubernetes* okruženje. Osim toga, koristeći direktnu komunikaciju sa *Kubernetes* platformom osigurava jasan prikaz statusa isporuke mikroservisa.

Jenkins server je konfigurisan tako da izvršava *Jenkins* agente odnosno zadatke, dinamički na *Kubernetes* platformi. Zadaci se izvršavaju u okviru *Pod*-a koji se sastoji od nekoliko *Docker* kontejnera. Ovdje je važno napomenuti da se za izvršenje svih koraka u okviru CI/CD linije koriste kontejneri kao dio nepromjenjive infrastrukture (eng. *immutable*

infrastructure). Ovaj koncept se zasniva na ideji da postoji samo jedan način da se naprave izmjene, a to je da se dio infrastrukture potpuno uništi i ponovo kreira. To znači da se za svako izvršenje određenog zadatka dinamički kreira zaseban *Kubernetes Pod* iz istih *Docker* slika, čiji je životni ciklus povezan sa vremenom izvršenja linije. Nakon što se izvršenje linije završi, *Pod* se automatski terminira.

Poređenje pristupa za organizaciju repozitorijuma izvornog koda, dato u trećoj glavi, pokazuje da svaka od metoda ima svoje prednosti i nedostatke. Iz toga proizlazi da nema jedinstvenog pristupa koji odgovara svima, a odluka se donosi na osnovu više faktora, između ostalog način organizacije timova kao i samog procesa isporuke softvera. Često može da zavisi od toga u kojoj je fazi proces implementacije mikroservisnih aplikacija, broja mikroservisa, veličine razvojnih timova i same organizacije, a između ostalog i da li se radi implementacija nove ili redizajn postojeće CI/CD linije. Za implementaciju praktičnog rada odabran je pristup gdje je izvorni kôd mikroservisa organizovan u odvojene repozitorijume. Testovi za određeni mikroservis nalaze se u istom repozitorijumu zajedno sa izvornim kodom. CI linija obezbjeđuje da određena izmjena napravljena u repozitorijumu izvornog koda mikroservisne aplikacije, startuje proces izvršavanja skupa unaprijed predefinisanih testova, kao koraka u okviru linije, a rezultuje u kreiranju artifakta (u našem slučaju *Docker* slika) koji je spreman za isporuku u određeno okruženje. CI linija može da se pokrene na tri načina:

- kreiran je zahtjev za integraciju izmjena na glavnu granu;
- zahtjev za integraciju izmjena na glavnu granu je pregledan i odobren, nakon čega je kreiran komit na glavnoj grani;
- CI linija je startovanja ručno od strane programera ili testera.

Nakon što je linija startovana, sljedeći koraci će uvijek biti izvršeni:

- Kreiranje lokalne kopije JSL biblioteka deklarisane u *Jenkinsfile* datoteci sa odgovarajućeg centralnog repozitorijuma izvornog koda, a potom kloniranje samog repozitorijuma mikroservisa koji je pokrenuo liniju.
- Sljedeći korak sastoji se od detekcije izmjena upotrebom *Git* alata i konfigurisanje određenih varijabli, na osnovu kojih se određuje dalji tok izvršavanja CI i CD linija. Ovdje se detektuje koja izmjena je startovala samu liniju, da li postoji *app.yaml* fajl kao i da li je izmjena generisana kao komit na glavnu granu ili iz zahtjeva za integraciju na glavnu granu. Na osnovu ovih parametara se zna da li se isporuka vrši u razvojno ili produkciono okruženje.
- Validacija sadržaja *app.yaml* datoteke sa ciljem detekcije sintaksnih grešaka.

5.1. CI linija

Izvršavanje CI linije odnosno testova za cilj ima da osigura funkcionalnost aplikacije i istovremeno održi visok kvalitet izvornog koda. Detaljni koraci su prikazani na slici 5.3. Ukoliko se, na primjer, kreira zahtjev za dodavanje izmjena u repozitorijum (eng. *merge request*), to će rezultovati slanjem određenog događaja na CI server, što će biti okidač za pokretanje CI/CD linije.

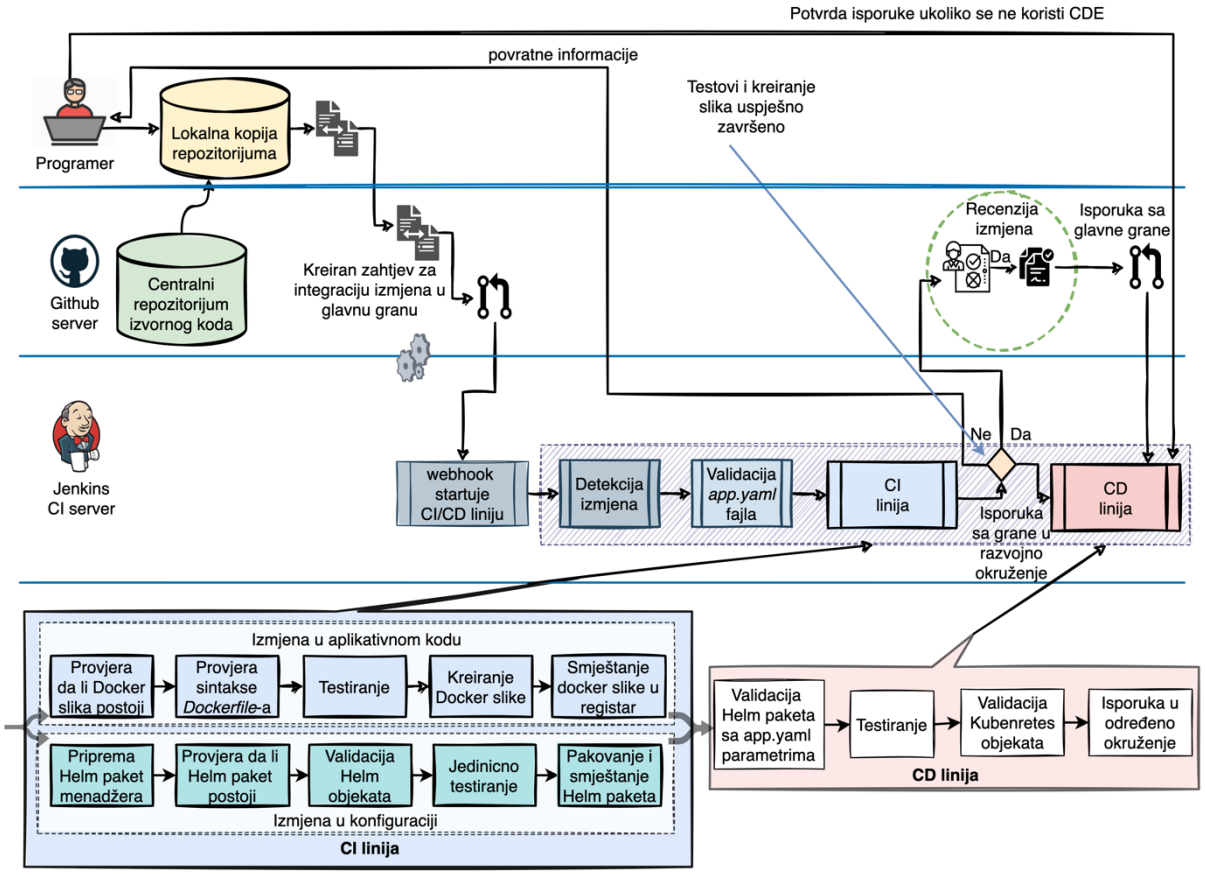
Da bi se CI/CD linija startovala potrebno je da se u okviru korijenskog direktorijuma izvornog koda nalazi datoteka pod nazivom *Jenkinsfile*. Ova datoteka omogućava CI/CD liniji da koristi *Jenkins* dijeljenu biblioteku, deklasiranjem naziva biblioteke koristeći *@Library*

anotaciju unutar *Jenkinsfile* datoteke kao što je prikazano na slici 5.2. Sadržaj ovog fajla se može posmatrati kao skup instrukcija koje se prosljeđuju *Jenkins* CI serveru. Ako posmatramo sliku 5.3, to bi značilo da je potrebno startovati liniju pod nazivom *pipelineCICD* koja je implementirana u okviru JSL biblioteke pod nazivom *Jenkins-pipeline-lib*. Liniji će se proslijediti parametar u formatu ključ-vrijednost. Prosljeđivanje određenih parametara definiše kako će se kreirati linija odnosno koji koraci će se izvršiti.

```
@Library('Jenkins-pipeline-lib')_

pipelineCICD {
  //skip Docker build step
  skipDockerBuildPipeline = true
}
```

Slika 5.3. Primjer Jekinsfile datoteke kao i opciono prosljeđivanje skipDockerBuildPipeline parametra

Kao što se vidi sa slike 5.2, CI dio linije je moguće u potpunosti preskočiti. Logično pitanje koje se nameće je da li je to i zašto potrebno i kakav je onda uopšte smisao CI linije u tom slučaju. Recimo, ukoliko se CI/CD linija želi koristiti za isporuku neke eksterne aplikacije, u tom slučaju *Docker* slika aplikacije već postoji pa je moguće kompletno preskočiti korake zadužene za testiranje i kreiranje slike. Logično, u tom slučaju možemo da koristimo samo CD dio linije kako bismo instalirali aplikaciju u željeno okruženje.



Slika 5.3. Detaljni prikaz koraka CI/CD sistema

Iako su CI i CD dijelovi jedne linije, slijedeći preporuke za implementaciju CI/CD linija, one su logički odvojene, što znači da se mogu startovati automatski ili ručno i izvršavati nezavisno jedna od druge [105]. Prilikom startovanja CI/CD linije, zahvaljujući osobinama *Git* fajl sistema, vrši se provjera koji su fajlovi izmijenjeni i na osnovu detektovanih izmjena

određuje daljnji tok izvršavanja linije. Ukoliko, na primjer, korak za detekciju izmjena detektuje samo izmjene unutar *app.yaml* datoteke, izvršenje CI linije će biti preskočeno. To se može desiti, na primjer, ukoliko želimo da instaliramo neku stariju verziju mikroservisne aplikacije i u tom slučaju se mijenja samo verzija *Docker* slike unutar *app.yaml* datoteke.

Praksa pokazuje da je najbolje izbjegavati kreiranje artifakata odnosno *Docker* slika za specifično okruženje [106]. Ideja koja stoji iza ovog stava je da se artifakt koji se kreira jednom koristi za sva okruženja, dok se konfiguracija specifična za dato okruženje postavlja tokom isporuke aplikacije [3]. Svaki mikroservis koji implementira određenu funkcionalnost i izvršava se u produkcionom okruženju ima niz konfiguracionih parametra kao što su adrese određenih servera, konfiguraciju za pristup serveru baze podataka, zatim veći broj zavisnosti (eng. *dependency*) kao što su određene lozinke ili sertifikati, konfiguracioni fajlovi ili parametri koji mogu biti smješteni u eksternim servisima kao što su *Consul* [107] ili ETCD [63]. Ukoliko je sam proces isporuke softvera pravilno dizajniran slijedeći preporuke, izmjena bilo kojeg od prethodno navedenih parametara trebala bi biti moguća bez potrebe da se kreira novi artifakt.

Kao što je prikazano na slici 5.3. predloženo rješenje za CI liniju se sastoji od dva dijela. Osnovne dvije funkcionalnosti CI linije su da se na osnovu detektovanih izmjena u okviru repozitorijuma izvornog koda izvrše sljedeće dvije akcije:

- izvršavanje testova i kreiranje *Docker* slike za određeni mikroservis;
- izvršavanje testova za validaciju konfiguracije aplikacije i kreiranje *Helm* paketa koji će se koristiti za instalaciju i konfigurisanje aplikacije prilikom isporuke u određeno okruženje.

5.1.1. Izmjena u aplikativnom kodu aplikacije

Kreiranje sigurne, robusne i brze linije za kreiranje *Docker* slika nije obično tako jednostavno kao što zvuči. Često je najveći izazov automatizacija rješavanja zavisnosti odnosno upotreba više faza i osnovnih *Docker* slika (eng. *base image*) kako bi se sam proces ubrzao. Osim toga, organizacija repozitorijuma u okviru registra *Docker* slika takođe često predstavlja izazov.

Pod pretpostavkom da se u okviru korijenskog direktorijuma repozitorijuma izvornog koda nalazi *Jenkinsfile* datoteka određenog sadržaja, a u slučaju detekcije izmjena u aplikativnom kodu mikroservisa, sljedeći koraci će biti izvršeni:

- Prvi korak je provjera strategije verzionisanja, parsiranje verzije *Docker* slike, a potom provjera u repozitorijumu za skladištenje *Docker* slika da li slika sa tom verzijom već postoji. Ukoliko postoji, izvršavanje će ovdje biti zaustavljeno i programeru će biti poslata odgovarajuća poruka. Ovaj korak je neophodan kako bi se izbjeglo kreiranje slike sa već postojećom verzijom, koje obično traje nekoliko minuta. Time se izbjegava nepotrebno izvršenje procesa kreiranja slike, koju kasnije ne bi bilo moguće smjestiti u repozitorijum zbog kolizije sa postojećom verzijom. Sa druge strane, loša praksa je prepisivanje već postojećih slika koje su isporučene u registar *Docker* slika pa se samim tim ova metoda ne koristi.
- Kako bi se programerima pružile brze i jasne povratne informacije, što je jedna od dobrih karakteristika ovakvog sistema, drugi korak je validacija sintakse *Docker* fajla upotrebom alata *Hadolint* [108].

- Sljedeći korak je izvršavanje testova i kreiranje *Docker* slike. Uspješno izvršenje ove faze za rezultat će imati novu verziju *Docker* slike smještenu u odgovarajući repozitorijum *Docker* slika. Ovakva slika može da se isporuči u bilo koje okruženje i izvrši na bilo kojem serveru sa instaliranom podrškom za *Docker* pa čak i na lokalnim mašinama programera. U tom slučaju preduslov za uspješno startovanje mikroservisa je da se obezbijede svi neophodni resursi i konfiguracioni parametri. U slučaju da je došlo do neke greške vezano za testove ili sam proces kreiranja, programeru će biti poslata odgovarajuća povratna informacija.

Tokom procesa implementacije ustanovljeno je da iako semantička šema za verzionisanje *Docker* slika ima nekoliko prednosti, često je lakše koristiti strategiju baziranu na heš vrijednosti *Git* komita, pa je u samoj CI liniji implementirana podrška za obje strategije. Upotreba semantičke šeme za verzionisanje omogućava da se korisnici mogu pretplatiti na odgovarajuću verziju donosno redoslijed i raspored izdavanja koji im najbolje odgovara. Osim toga, ova šema verzionisanja je čitljivija i veoma intuitivna. Samim tim može se lako utvrditi da li nova verzija sadrži samo zakrpu ili ipak donosi izmjene koje nisu kompatibilne sa prethodnim verzijama mikroservisa, kao što je to opisano detaljno u prethodnom poglavlju. Međutim, upotreba ove strategije zahtijeva postojanje datoteke pod nazivom TAG u okviru korijenskog direktorijuma repozitorijuma izvornog koda, unutar koje se nalazi sama verzija za označavanje *Docker* slike. Svaki put prilikom izmjene aplikativnog koga potrebno je izvršiti inkrementovanje verzije u okviru ovog fajla. Sa druge strane, strategija verzionisanja upotrebom heš vrijednosti daje mnogo više fleksibilnosti, jer se verzija slike generiše svaki put automatski. Izbor same šeme verzionisanja moguće je odabrati pomoću konfiguracionog parametra *DockerImageTaggingStrategy*, pri čemu je podrazumijevana strategija bazirana na upotrebi heš vrijednosti *Git* komita.

Osim toga, važno je napomenuti da je kompletna CI linija dinamički generisana tako da je moguće vrlo lako uvesti dodatne korake ili poslati određene komande koje će biti izvršene, na primjer, tokom izvršavanja jediničnih testova. Ovim se omogućava velika fleksibilnost, jer različiti mikroservisi mogu imati različite zahtjeve u pogledu okruženja neophodnog za izvršenje različitih grupa testova kao i programskih jezika u kojima su napisani.

5.1.2. Izmjene konfiguracije okruženja

Za instaliranje i konfigurisanje mikroservisa izabran je *Helm* paket menadžer za *Kubernetes*, koji omogućava jednostavno instaliranje i konfigurisanje aplikacija i servisa u *Kubernetes* okruženju. Konfiguracioni parametri uključuju, ali nisu ograničeni na konfiguraciju servisa, kredencijala i parametara za pristup drugim podacima, imena servera kao i kredencijale za autentikaciju i pristup drugim servisima.

Slijedeći metodologiju aplikacije dvanaest faktora (eng. *The Twelve-Factor App*) [109] za izradu softvera, svi konfiguracioni parametri su odvojeni od aplikativnog koda. Time se omogućuje da se jednom kreirana *Docker* slika za odgovarajući mikroservis, može instalirati odnosno isporučiti u bilo koje aplikativno okruženje. Osim toga, praksa pokazuje da je proces za upravljanje konfiguracijom aplikacije jedan od osnovnih preduslova da bi se kreirao robusan i skalabilan CI/CD.

Da bi se jedna aplikacija uspješno instalirala u okviru *Kubernetes* platforme potrebno je kreirati određen broj objekata. Međutim, ti objekti moraju da slijede unaprijed definisane šeme i specifikacije, a zbog njihove kompleksnosti to često može da bude jako komplikovan proces. Obično se pod tim podrazumijeva kreiranje nekoliko datoteka u YAML formatu, koji se nakon

toga prosljeđuju *Kubernetes* API servisu koji omogućava manipulaciju stanjem objekata unutar *Kubernetes* klastera. Ukoliko se ovaj broj multiplicira sa više okruženja odnosno većim brojem klastera, može se naslutiti kompleksnost problema. U zavisnosti on načina na koji je organizovan proces razvoja i isporuke softvera, to često znači da se od programera ili sistem administratora očekuje kreiranje i ažuriranje ovih datoteka.

Da bi se pojednostavio ovaj proces koristi se *Helm,* paket menadžer *za Kubernetes,* koji pojednostavljuje proces instalacije i konfiguracije mikroservisnih aplikacija. On je *Kubernetes* ekvivalent paket menadžerima *apt* [110] i *yum* [111], koji se koriste na *Linux* operativnom sistemu. Njegova osnovna namjena je da pojednostavi proces instalacije i isporuke mikroservisa, omogućujući instalaciju većeg broja broj komponenata odnosno resursa kao jedne cjeline. Na sljedećem primjeru prikazana je osnovna struktura jednog *Helm* paketa.

```
naziv-Helm-paketa/
 Chart.yaml      # Datoteka u YAML formatu koji sadrži podatke o Helm paketu
 LICENSE         # Opciono: Tekstualna datoteka koja sadrži licencu za Helm paket
 README.md       # Opciono: Tekstualna datoteka koja obično sadrži korisne informacije
 values.yaml     # Datoteka u YAML formatu koja sadrži podrazumijevane konfiguracione vrijednosti
za Helm paket
 values.schema.json # Opciono: JSON šema koja opisuje strukturu vrijednosti u values.yaml fajla.
Može se koristiti za validaciju konfiguracionih parametara.
 charts/         # Opciono: Direktorijum koji sadrži bilo koji Helm paket koji ovaj paket koristi.
 templates/      # Direktorijum koji sadrđi templejte, koji u kombinaciji sa parametrima iz
vaules.yaml datoteke generišu valiked Kubernetes manifest
 templates/NOTES.txt # Opciono: Tekstualna datoteka koja sadrži kratke napomene o korištenju
```

Slika 5.4. Opis formata i sadržaja Helm paketa

Kao što se vidi sa slike 5.4. i opisa, ideja je da se unutar samog paketa nalaze šabloni za kreiranje *Kubernetes* objekata kojim se može manipulisati prosljeđujući odgovarajuće konfiguracione parametre, a kao rezultat dobijaju se odgovarajući *Kubernetes* manifesti u YAML formatu. *Helm* posjeduje klijentsku aplikaciju čija je osnovna namjena komunikacija sa *Kubenetes* API serverom i manipulacija objektima, sa ciljem da se oni dovedu u odgovarajuće stanje. Drugim riječima, *Helm* klijent ima odgovornost da kreira sve objekte definisane u paketu tokom prvog instaliranja, kao i da kod svake naredne isporuke obezbijedi njihovo ažuriranje, koje često osim ažuriranja može podrazumijevati i brisanje nekih objekata. Na primjer, kod prve isporuke kreira se *Deployment* objekat u okviru *Kubernetes* klastera koji, između ostalog, sadrži naziv i verziju *Docker* slike. Kod svake sljedeće promjene, na primjer promjena verzije slike, on neće kreirati novi *Deployment* objekat već samo ažurirati postojeći.

Verzionisanje *Helm* paketa omogućuje da se izmjene vrše veoma jednostavno i brzo mijenjajući samo verziju paketa. Međutim, kako bi se sam proces učinio stabilnim i osigurala visoka frekventnost isporuke, neophodno je unutar same CI/CD linije uspostaviti proces za testiranje, kreiranje i smještanje ovih paketa. Dodatno, važno je napomenuti da sam paket ne treba da sadrži konfiguracione parametre, jer se on koristi za instaliranje aplikacije u bilo kojem okruženju. Sam paket obično sadrži podrazumijevane konfiguracione parametre neophodne da bi se izvršili neophodni testovi sa ciljem da potvrde validnost šablona prije samog pakovanja i smještanja paketa u *Helm* registar. *Helm* registar je veoma jednostavan HTTP server gdje se smještaju *Helm* paketi u vidu *tag.gz* arhivskih fajlova. Dodatno, ovaj server sadrži jedan indeks fajl u kojem se nalazi list svih *Helm* paketa, koji klijent parsira i koristi prilikom određenih operacija.

Kao što je prethodno rečeno, da bi se koristila i aktivirala linija za kreiranje *Helm* paketa neophodno je da postoji direktorijum pod nazivom *chart* unutar korijenskog direktorijuma repozitorijuma izvornog koda mikroservisa. Sama arhitektura rješenja podržava da se koriste

eksterni *Helm* paketi kao i proizvoljni *Helm* registri. Ukoliko ovaj direktorijum ne postoji, ovaj dio CI linije neće biti startovan, u suprotnom sljedeći koraci će biti izvršeni:

- Parsiranje verzije *Helm* paketa i provjera da li paket sa datom verzijom već postoji u datom *Helm* registru. Kako nije moguće smjestiti istu verziju u registar, ovaj korak je napravljen kako bi se spriječio eventualni konflikt i pravovremeno poslale povratne informacije programerima. Time se izbjegava nepotrebno trošenje vremena na testiranje i kreiranje paketa koji ne može da se smjesti u registar.
- U sljedećem koraku biće izvršena validacija (eng. *linting*) objekata unutar samog direktorijuma kao i validacija konfiguracionih parametara na osnovu JSON šeme koja se nalazi unutar *values.schema.json* i sastavni je dio *Helm* paketa**.**
- Kako bi se ostvarila kontrola koje parametre programeri mogu da proslijede odnosno koje objekte da kreiraju unutar samog *Kubernetes* klastera, implementirano je jedinično testiranje *Helm* šablona. Ovi testovi omogućavaju granularnu kontrolu, koje objekte je moguće kreirati iz *Helm* paketa, prilikom isporuke mikroservisnih aplikacija. Na primjer, moguće je blokirati kreiranje šablona za izmjene permisija u okviru *Kubernetes* klastera ili upotreba *latest* verzije *Docker* slike unutar *Deployment* objekta. Naravno ove restrikcije se mogu obezbijediti i na strani samog *Kubernetes* klastera, ali uvijek treba imati na umu da programeri treba da dobiju pravovremeno obavještenje. Često se prilikom implementacije ovakvih ograničenja, postavlja pitanje da li kontrolu vršiti u okviru same CI/CD linije ili upotrebom određenih alata u okviru same *Kubernetes* platforme. Mnogo je jednostavnije i pravilnije blokirati ovakve izmjene unutar samog CI/CD sistema, nego dozvoliti da se *Helm* paket kreira i smjesti u registar, da bi se potom ustanovilo da on može da se instalira, što rezultuje nepotrebnim trošenjem vremena i resursa. Za jedinično testiranje prilikom implementacije izabran je *Conftest* [112] alat, gdje je dat samo jedan primjer kako se upotrebom *rego* [113] programskog jezika može vršiti validacija *Helm* šablona. Ovaj alat je izabran jer olakšava validaciju samih šablona, koji kao rezultat generišu određen broj YAML fajlova, pri čemu neki od njih mogu da sadrže i po nekoliko stotina redova. Takođe, važno je napomenuti da se ovdje ne radi validacija specifikacije *Kubernetes* objekata u odnosu na njihovu šemu, već se u skladu sa sigurnosnom politikom organizacije definiše koje vrste objekata i sa kojim vrijednostima je moguće kreirati u okviru samog klastera. Ovim se izbjegava kreiranje *Helm* paketa koji se kasnije ne bi mogao instalirati zbog sigurnosnih politika na strani *Kubernetes* klastera. Ukoliko se radi validacija na strani samog *Kubernetes* klastera, često je izbor je *Kubernetes* operator pod nazivom *OPA Gatekeeper* [114], koji omogućava kreiranje polisa upotrebom *rego* programskog jezika.
- Nakon što su testovi uspješno izvršeni, sljedeći korak je kreiranje *Helm* paketa odnosno odgovarajuće arhive koristeći *Helm* klijent, kao i njeno smještanje u odgovarajući registar, što je ujedno i posljednji korak ovog dijela CI linije.

5.2. CD linija

Srce sistema je CLI aplikacija pod nazivom *piggle*, odgovorna za orkestraciju isporuke i komunikaciju sa *Kubernetes* API. Upotreba zasebne aplikacije za ovu namjenu, umjesto da se *Kubernetes* API serverom komunicira direktno iz *JSL groovy* koda ili eventualno *shell* skripte, ima nekoliko prednosti. Na primjer, ukoliko se korak *Helm* instalacije izvrši uspješno, to ne mora da znači da je kompletna isporuka uspješna. Dodatni korak validacije je ovdje neophodan kako bi se validirao status mikroservisa i da li je on uspješno isporučen u aplikativno okruženje.

Osim toga u slučaju da se organizacija odluči za promjenu CI servera postojeća CLI aplikacija kao i poslovna logika sadržana unutar nje može da se nastavi koristiti.

Klijentska aplikacija napisana je u *Go* programskom jeziku i koristi već postojeće biblioteke za interakciju sa *Helm* klijentom i *Kubernetes* API servisom. CI/CD sistem je dizajniran tako da se samo nominalno stanje čuva unutar samog repozitorijuma mikroservisa u datoteci pod nazivom *app.yaml*. Primjer takvog fajla prikazan je na slici 5.5, a implementacija je opisana u šestoj glavi. Klijentska aplikacija parsira ovaj fajl, prosljeđuje konfiguracione vrijednosti *Helm* alatu, a na osnovu konfiguracionih vrijednosti i *Helm* paketa generiše *Kubernetes* manifest datoteke. Na osnovu ovih datoteka, klijentska aplikacija upotrebom *Helm* biblioteke vrši kreiranje *Kubernetes* objekata, što je sastavni dio instalacije aplikacije na jednom ili više *Kubernetes* klastera. Ovaj koncept omogućuje veoma jednostavnu izmjenu bilo kojeg konfiguracionog parametra bez potrebe da se kreira nova verzija *Helm* paketa. Sve vrijednosti unutar fajla, osim vrijednosti *common*, su globalne i primjenjuju se na sve klastere.

```yaml
version: '1.0'

common:
  appName: backend
  image:
    repository: njegosrailic/backend
    tag: 0.1.0
  namespace: backend

chart:
  name: backend
  version: 0.1.0

clusters:
  k8s-dev-c1:
    image:
      pullSecret: regsecret
      tag: 0.1.0
    replicaCount: 1
    service:
      externalPort: 90
      internalPort: 8080
      name: backend-service
      type: NodePort
```

Slika 5.5. Primjer app.yaml datoteke za backend mikroservis

Klijentska aplikacija parsira *app.yaml* fajl i na osnovu konfiguracionih vrijednosti izvršava odgovarajuće akcije. CD linija će se startovati svaki put ukoliko postoji validan fajl *app.yaml* unutar repozitorijuma izvornog koda mikroservisne aplikacije. CD dio linije će se startovati čak i ako nema direktne izmjene na ovom fajlu. Prvi korak će biti provjera stanja aplikacije, čime će se utvrditi da li verzija i konfiguracija mikroservisa odgovaraju deklarativnoj specifikaciji unutar *app.yaml* fajla. Osim toga, ovim se omogućava da se CI/CD linija u svakom trenutku može startovati bez rizika da se desi nešto što bi uticalo na funkcionisanje same mikroservisne aplikacije.

Ukoliko je CD linija pokrenuta, sljedeći koraci će biti izvršeni:

- Prvi korak je validacija *Helm* šablona koristeći konfiguracione parametre koji su proslijeđeni u *app.yaml* fajlu. Ovo je neophodno jer, kao što je već rečeno u prethodnom tekstu, *Helm* paket sadrži podrazumijevane konfiguracione parametre, međutim, programer može da proslijedi bilo koju vrijednost konfiguracionih parametara.
- Sljedeći korak je generisanje *Kubernetes* manifest fajlova za dati klaster, a kao rezultat se dobijaju specifikacije objekata u YAML formatu. Nakon toga se upotrebom *Kubeval* [115] alata vrši validacija ovih fajlova u odnosu na šemu za određenu verziju *Kubernetes* API verzije.

- Sljedeći korak je ponovo testiranje *Helm* paketa, ali u ovom slučaju sa konfiguracionim parametrima koji su proslijeđeni u okviru *app.yaml* fajla za dati klaster.
- Ukoliko su svi prethodni koraci uspješno završeni, u posljednjem koraku se izvršenje delegira klijentskoj aplikaciji koja izvršava nekoliko koraka. Aplikacija je zadužena za orkestraciju isporuke aplikacije, što podrazumijeva provjeru statusa, kreiranje virtuelnog klastera kao i instalaciju odnosno isporuku same mikroservisne aplikacije. Detaljan opis koraka klijentske aplikacije dat je u narednoj glavi.

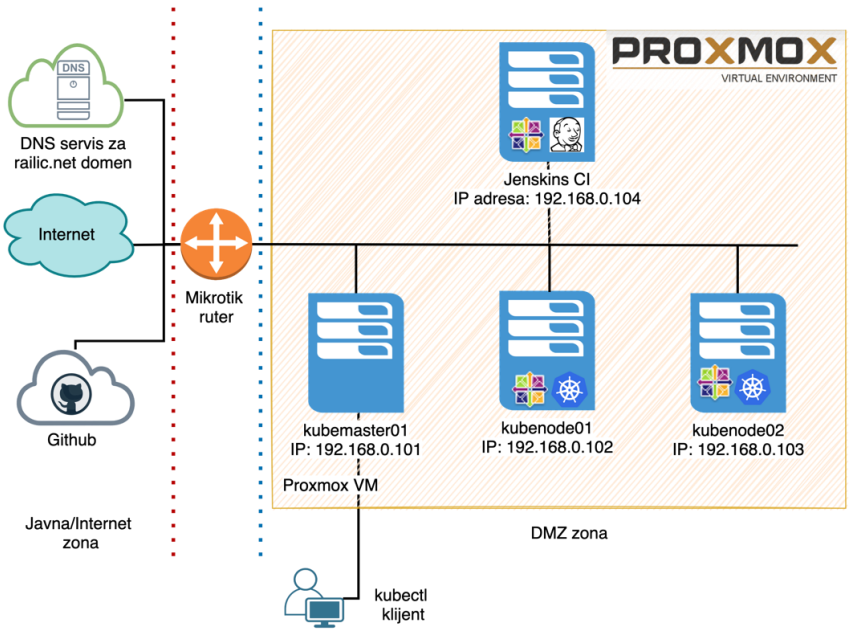
Važno je napomenuti da se kombinacija naziva aplikacije, imena virtuelnog klastera, naziva *Helm* paketa i naziva repozitorijuma izvornog koda koristi da bi se spriječio konflikt tokom životnog ciklusa aplikacije. Ukoliko se bilo koji od ovih parametara promijeni, isporuka se neće izvršiti uspješno. Razlog za to je sprečavanje isporuke mikroservisa u virtuelni klaster gdje je već isporučen drugi mikroservis. Isporuke više mikroservisa u isti virtuelni klaster nije podržana kako bi se izbjegla eventualna kolizija i slijedila osnovna uloga ovog koncepta, a to je izolacija između različitih mikroservisa.

6. OPIS IMPLEMENTIRANOG RJEŠENJA I REZULTATI PRIMJENE

U ovoj glavi opisan je praktični dio rada. U prvom dijelu dat je kratak opis infrastrukture korištene za implementaciju rada kao i alata upotrijebljenih za kreiranje i konfigurisanje okruženja. Drugi dio daje opis implementirane biblioteke za CI/CD kao i klijentske aplikacije za orkestraciju isporuke mikroservisa u aplikativna okruženja, kao i opis realizacije progresivne isporuke za mikroservisne aplikacije.

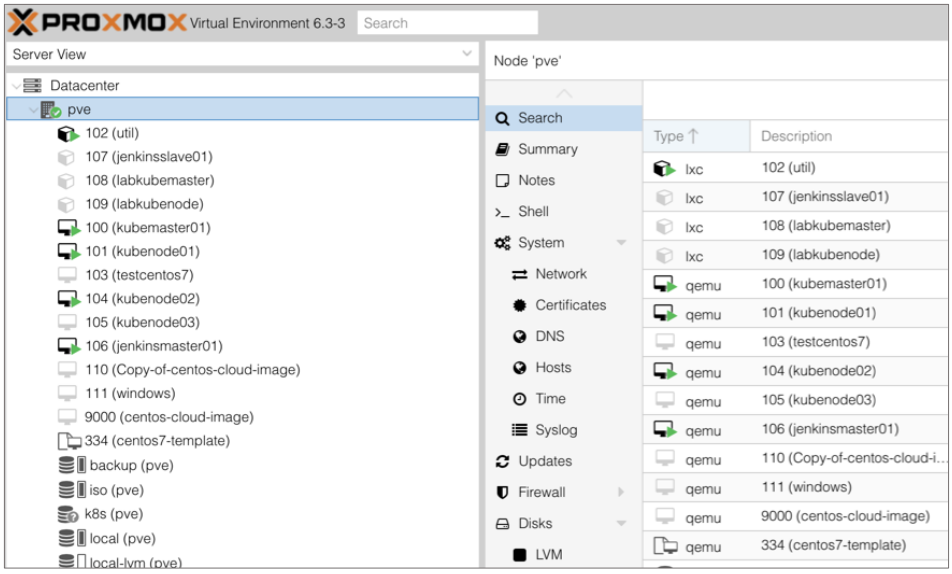
6.1. Infrastruktura

Za implementaciju praktičnog dijela rada, kao što je to objašnjeno u uvodnom dijelu rada, izabran je *Kubernetes* sistem za orkestraciju kontejnera. Kompletna instalacija je bazirana na upotrebi infrastrukture kao koda (IAC - *Infrastrucutre as Code*), gdje je umjesto ručnog konfigurisanja komponenata i alata kompletno okruženje definisano u konfiguracionim datotekama. Ovaj pristup je izabran sa jedne strane da bi se pojednostavio sam proces konfiguracije okruženja i izbjegla potreba za ručnim operacijama, koja može biti komplikovana u mikroservisnom okruženju. Sa druge strane, upotreba ovih tehnologija omogućava veoma jednostavno ponovno kreiranje kompletnog okruženja korištenog prilikom implementacije praktičnog rada. Ovo naravno treba posmatrati sa rezervom, jer uzimajući u obzir temu i kompleksnost rada, neki dijelovi procesa nisu potpuno automatizovani. Infrastruktura koja je korištena obuhvata fizičku opremu, jedan fizički server i ruter, virtuelne mašine, kao i eksterni *Github* servis, kao centralizovanu lokaciju za skladištenje svih repozitorijuma izvornog koda. Kako bi se internim mikroservisima omogućio pristup sa interneta, korišten je DNS domen *railic.net* koji je hostovan na *Netlify* [116] servisu.



Slika 6.1. Logička šema infrastrukture korištene za implementaciju praktičnog rada

Kao virtuelizacijska platforma korišten je *ProxMox Virtual environment* [117], gdje je za potrebe implementacije kreirano i konfigurisano nekoliko virtuelnih mašina, kao što je to prikazano na slici 6.1. Korisnički interfejs *ProxMox* virtualizacijske platforme sa listom virtuelnih mašina je prikazan na slici 6.2.



Slika 6.2. Prikaz konfigurisanih virtuelnih mašina korištenih za implementaciju praktičnog dijela rada

Lista virtuelnih mašina sa pripadajućim IP adresama i kratkim opisom prikazana je u tabeli 6.1.

Tabela 6.1. Tabelarni pregled virtuelnih mašina

Naziv virtuelne mašine	**IP adresa**	**Vrsta**	**Operativni sistem**	**Opis**
pve	192.168.0.86	Fizički server	Debian/ Proxmox v6.3	Virtuelizacijski server
kubemaster01	192.168.0.101	Virtuelna mašina	CentOS 7	*Kubernetes* upravljački čvor
kubenode01	192.168.0.102	Virtuelna mašina	CentOS 7	*Kubernetes* radni čvor
kubenode02	192.168.0.103	Virtuelna mašina	CentOS 7	*Kubernetes* radni čvor
jenkinsmaster01	192.168.0.104	Virtuelna mašina	CentOS 7	*Jenkins* master

6.1.1. *Packer* alat

Za kreiranje instalacione slike korišten je alat pod nazivom *packer* [118]. Njegova uloga je da pojednostavi kreiranje slika odnosno šablona koji će se koristiti za kreiranje virtuelnih mašina. Glavna svrha upotrebe je da se izbjegne instaliranje operativnog sistema i konfigurisanje svake virtuelne mašine ponaosob.

```
packer
├── centos7.json
├── http
│   └── ks.cfg
├── ks.cfg
└── packer_cache
    └── port
```

Slika 6.3. Pregled stabla packer direktorijuma

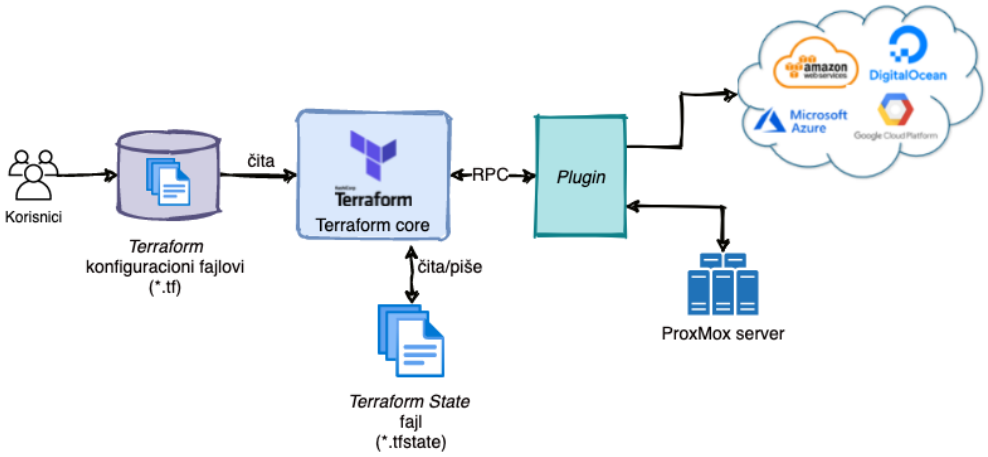
Kompletna konfiguracija ovog alata se sastoji od svega dva fajla i to se može provjeriti u priloženom kodu. Fajl na lokaciji */prakticni-rad/packer/centos7.json* sadrži konfiguraciju za alat *packer*, koje on koristi kao instrukcije prilikom kreiranja slike virtuelne mašine. Osim toga tu su i kredencijali za pristup *ProxMox* serveru. Dodatni fajl na lokaciji */prakticni-rad packer/http/ks.cfg* je standardni *Linux Kickstart* [119] fajl koji se koristi za automatizaciju instalacije *Linux* operativnog sistema. Nakon što se pomoću *packer* alata kreira virtuelna mašina i instalira osnovni *Linux* operativni sistem, ona se konvertuje u sliku virtuelne mašine

koja se sačuva u okviru *ProxMox* platforme. Iz ove slike se kasnije upotrebom *terraform* alata kreira infrastruktura odnosno odgovarajuće virtuelne mašine.

6.1.2. *Terraform* alat

Terraform je alat otvorenog koda za upravljanje IaC-om. Koristi se za definisanje i upravljanje infrastrukturom upotrebom deklarativnog jezika. *Terraform* je napisan u programskom jeziku *Go* oslanjajući se na najpoznatije karakteristike ovog programskog jezika, kao što su brzina i postizanje visokog nivoa konkurentnosti [120]. Interna arhitektura ovog alata može se podijeliti u dvije cjeline:

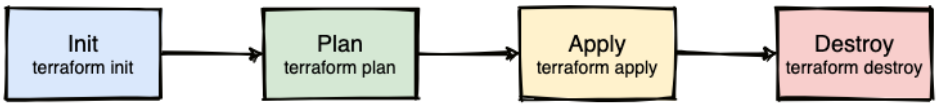
- *Terraform* jezgro (eng. *terraform core*);
- *Terraform plugin* koji se sastoji iz dva dijela, a to su provajderi i provizori (eng. *provisioners*).



Slika 6.4. Prikaz arhitekture terraform alata

Jezgro alata distribuira se kao mala statički kompajlirana binarna datoteka. Ova binarna izvršna datoteka sa stanovišta korisnika je u stvari CLI alat, preko kojeg se izdaju naredbe i kontrolišu različite operacije. Osim što se koristi kao ulazna tačka za komunikaciju sa korisnicima, jezgro je zaduženo za učitavanje i validaciju konfiguracionih fajlova. U *terraform*-u se svaki objekat naziva resurs. Nakon što učita i validira konfiguracione fajlove, jezgro generiše graf koji se sastoji od svih resursa i veza između njih. Podaci o svim postojećim resursima čuvaju se u okviru fajla koji se naziva fajl stanja (eng. *state file*). Osnovne operacije koje korisnik može da izvrši su:

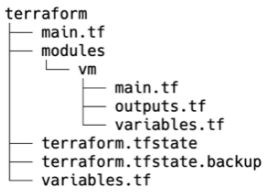
- *plan* – operacija koja ne generiše izmjene, već samo prikazuje izmjene koje će se izvršiti;
- *apply* – izvršavanje izmjena;
- *destroy* – briše odnosno uništava sve resurse koji se nalaze u fajlu stanja.



Slika 6.5. Prikaz toka izvršenja operacija (eng. provisioning workflow) terraform alata

Jezgro komunicira sa dodacima preko RPC protokola i delegira komande za izvršavanje. Komanda *apply* će se izvršavati dok se ne završe sve izmjene da bi se infrastruktura deklarisana u konfiguracionim fajlovima dovela u željeno stanje ili dok se ne desi greška. U skladu sa

izmjenama *terraform* će da ažurira fajl stanja. Jedini podatak koji *terraform* ima o stanju infrastrukture nalazi se u ovom fajlu pa je jasno da je uz konfiguracione fajlove najbitnija komponenta. Zbog toga *terraform* ima opciju da se ovaj fajl čuva na udaljenoj lokaciji (eng. *remote state file*) u okviru različitih servisa u oblaku, kao što su *Amazon S3* [121], *Google Cloud Storage* [122] i *Azure Blob Storage* [123].



Slika 6.6. Prikaz stabla terraform direktorijuma

Infrastruktura za implementaciju praktičnog rada je kreirana koristeći *terraform* alat. Da bi *terraform* komunicirao sa *Proxmox* serverom potreban je odgovarajući provajder. Korišteni provajder je dostupan na adresi: https://github.com/Telmate/terraform-provider-proxmox. Instalacija i konfiguracija samog provajdera opisana je u okviru prethodno pometnutog repozitorijuma izvornog koda. Konfiguracioni fajlovi su priloženi uz rad i nalaze se u direktorijuma na lokaciji */praktični-rad/terraform*, a struktura direktorijuma prikazana je na slici 6.6.

```
terraform {
 required_version = ">= 0.12.0"
}

provider "proxmox" {
 pm_tls_insecure = true
 pm_api_url      =
"https://192.168.0.86:8006/api2/json"
 pm_password     = "XXXX"
 pm_user         = "terraform-proxmox-user@pam"
}

module "kubemaster01" {
 source      = "./modules/vm"
 disk_config   = local.disk_config
 network_config = local.kubemaster01_network_config
 vm_quemu_config = local.kubemaster01_quemu_config
}

module "kubenode01" {
 source      = "./modules/vm"
 disk_config   = local.disk_config
 network_config = local.kubenode01_network_config
 vm_quemu_config = local.kubenode01_quemu_config
}

module "kubenode02" {
 source      = "./modules/vm"
 disk_config   = local.disk_config
 network_config = local.kubenode02_network_config
 vm_quemu_config = local.kubenode02_quemu_config
}

module "Jenkinsmaster01" {
 source      = "./modules/vm"
 disk_config   = local.disk_config
 network_config =
local.Jenkinsmaster01_network_config
 vm_quemu_config =
local.Jenkinsmaster01_quemu_config
}
```

```
variable "ip_address" {
 type    = map
 description = "Datastore cluster onto which VMs
will be deployed"

 default = {
  kubemaster01    = "192.168.0.101"
  kubenode01      = "192.168.0.102"
  kubenode02      = "192.168.0.103"
  Jenkinsmaster01 = "192.168.0.103"
 }
}

##########################################
#  Server configuration           #
##########################################
locals {
 kubemaster01_quemu_config = {
  name = "kubemaster01"
  desc = "kubemaster01"
  boot       = "cdn"
  bootdisk    = "virtio0"
  template_name  = "centos-cloud-image"
  cores      = 1
  ipconfig0    =
"ip=${var.ip_address["kubemaster01"]}/24,gw=192.1
68.0.1"
  memory     = 4096
  os_type     = "cloud-init"
  pool       = ""
  sockets     = 2
  ssh_user    = "root"
  target_node   = "pve"
  ssh_private_key = var.ssh_private_key
  sshkeys     = var.sshkeys
  ciuser     = null
  cipassword    = null
  scsihw     = "virtio-scsi-pci"
 }
```

Slika 6.7. Prikaz glavnog main.tf terrafom fajla i dijela variables.tf fajla koji sadrži osnovnu konfiguraciju za kreiranje virtuelnih mašina

U okviru konfiguracije koristi se jedan modul koji ustvari predstavlja sve neophodne fajlove za kreiranje jedne virtuelne mašine (resursa). Umjesto da se kôd svaki put kopira za

svaku novu virtuelnu mašinu ponaosob, ovi fajlovi se grupišu u cjelinu koja se naziva modul pa se resurs (u ovom slučaju virtuelna mašina) kreira iz modula. Kao što možemo da vidimo sa slike 6.7. iz sadržaja *main.tf* fajla, minimalna verzija *terraform*-a sa kojom je kôd kompatibilan je 0.12.0. Takođe, vidimo da se koristi *Proxmox* provajder, koji koristeći odgovarajuće kredencijale, komunicira sa *Proxmox* API serverom na adresi *https://192.168.0.86:8006/api2/json*. Iz priloženog se vidi da su deklarisane četiri virtuelne mašine od kojih tri za *Kubernetes* i jedna za *Jenkins* CI server. Osnovi konfiguracioni parametri, kao što su naziv virtuelne mašine, IP adresa, tip i veličina diska, neophodni da bi se kreirala jedna virtuelna mašina, proslijeđuju se modulu koristeći datoteku pod nazivom *variables.tf*.

6.1.3. *Kubernetes* klaster

Nakon što su kreirane virtuelne mašine upotrebom *terraform* alata, kreiran je *Kubernetes* klaster koji se sastoji od jednog upravljačkog i dva radna čvora.

6.1.3.1. *Kubespray* alat

Nakon što su virtuelne mašine kreirane, izvršena je instalacija i inicijalizacija *Kubernetes* klastera upotrebom *kubespray* [124] alata. Ovaj alat je zasnovan na *Ansible* alatu za upravljanje konfiguracijom i nudi veliku fleksibilnost. Fleksibilnost se ogleda u mogućnosti da se veoma brzo i lako odradi izmjena konfiguracije koja je definisana kao IaC, a u dodatne skripte i alate koji su sastavni dio *kubespray*-a su integrisani znanje i praktično iskustvo inženjera koji rade svakodnevno sa mikroservisnim arhitekturama.

```
cat prakticni-
rad/kubespray/inventory/mycluster/hosts.yml
all:
  hosts:
    kubemaster01:
      ansible_host: 192.168.0.101
      ip: 192.168.0.101
      access_ip: 192.168.0.101
    kubenode01:
      ansible_host: 192.168.0.102
      ip: 192.168.0.102
      access_ip: 192.168.0.102
    kubenode02:
      ansible_host: 192.168.0.103
      ip: 192.168.0.103
      access_ip: 192.168.0.103
  children:
    kube-master:
      hosts:
        kubemaster01:
    kube-node:
      hosts:
        kubenode01:
        kubenode02:
    etcd:
      hosts:
        kubemaster01:
    k8s-cluster:
      children:
        kube-master:
        kube-node:
    calico-rr:
      hosts: {}
```

Slika 6.8. Prikaz konfiguracionih parametara korištenih za instaliranje klastera upotrebom kubespray alata

To omogućava da se praktično izvršenjem jedne komande na unaprijed kreiranim virtuelnim mašinama, instalira potpuno funkcionalan *Kubernetes* klaster. Instalacija klastera ne mora da se obavi na unaprijed kreiranim virtuelnim mašinama, jer alat podržava kreiranje klastera na infrastrukturi većine poznatih pružaoca usluga u oblaku kao što su AWS [125], GCP [126] i *Microsoft Azure* [127]. Dodatno, alat ima podršku za veliki broj operativnih sistema kao i za deset mrežnih komponenata (eng. *network plugins*).

Tabela 6.2. Prikaz osnovnih konfiguracionih komponenata sistema

Naziv komponente/parametra	Vrijednost	Opis
Mrežni segment za servise	10.233.0.0/18	Servisima u okviru platforme može da bude dodijeljena adresa samo iz ovog segmenta.
Mrežni segment za *Pod*-ove	10.233.64.0/18	*Pod*-ovima u okviru klastera može da bude dodijeljena adresa samo iz ovog segmenta.
***Kubernetes* klaster verzija**	v1.15.3	
Container runtime	*Docker*	Kao što je već rečeno kao CR je izabran *Docker* kako najpopularniji CR u toku pisanja rada.
Mrežni dodatak	*Calico*	Mrežni dodatak koji je zadužen za rutiranje saobraćaja.

6.1.3.2. *Helmfile* alat

Nakon instalacije svih komponenata *Kubernetes* platforme, dobija se potpuno funkcionalan sistem za orkestraciju mikroservisnih aplikacija. Međutim, u zavisnosti od konkretne namjene klastera, često je potrebno instalirati nekoliko dodatnih komponenata ili izvršiti dodatnu konfiguraciju i okviru same platforme. Infrastruktura za implementaciju praktičnog rada kreirana je upotrebom *terraform* alata, a *Kubernetes* klaster upotrebom *kubespray* alata. Ukoliko slijedimo prethodno pomenute smjernice koje preporučuje ITIL, sva konfiguracija bi trebala da se čuva u sistemu za kontrolu verzija. Ovim se ostvaruje bolja kontrola konfiguracije samog sistema omogućujući veoma brzo kreiranje novih klastera ili reinstalaciju postojećih.

Iako je *Helm* paket menadžer praktično standard za instaliranje i upravljanje aplikacijama, često direktna interakcija sa *Helm* klijentskom aplikacijom nije praktična ukoliko govorimo o instalaciji sistemskih komponenata. Osim toga, ukoliko se ista CI/CD linija koristi za isporuku sistemskih i korisničkih aplikacija, to često može dovesti do veoma kritičnih situacija. Ukoliko je ta linija neispravna, praktično je nemoguće isporučiti bilo kakve izmjene konfiguracije u okviru *Kubernetes* klastera osim ručnih izmjena u samoj konfiguraciji. Osim što su ručne izmjene nepraktične i podložne greškama, dodatni problem predstavlja kompleksnost konfiguracije. Ona se obično sastoji od manifesta u obliku YAML fajlova koji mogu da sadrže nekoliko desetina pa do nekoliko stotina linija teksta.

```
├── charts                - Direktorijum koji sadrži lokalne Helm pakete.
│   ├── chartmuseum
│   ├── flagger
│   ├── grafana
│   ├── metrics-server
│   ├── nginx-ingress
├── Helmfile.d            - Direktorijum koji sadrži Helmfiles odnosno deklaraciju infrastrukture
│   ├── dev
│   ├── prod
└── values                - Konfiguracioni parametri razdvojeni po okruženju
    ├── dev
    └── prod
```

Slika 6.9. Prikaz organizacije Helmfile repozitorijuma

U okruženjima u kojima su procesi za menadžment infrastrukture automatizovani i koriste IaC, često se koristi neki od alata kao što je *Helmfile* [128] pa je isti korišten i za ovaj rad. *Helmfile* predstavlja dodatni sloj deklarativne specifikacije, koji omogućava kompoziciju više *Helm* paketa da bi se kreirao sveobuhvatan artifakt za isporuku bilo jedne aplikacije ili kompletne infrastrukture. Osim upotrebe šablona i paketa za generisanje *Kubernetes* manifesta, koje obezbjeđuje sam *Helm*, *Helmfile* omogućava lako manipulisanje konfiguracionim parametrima i njihovu separaciju između različitih okruženja.

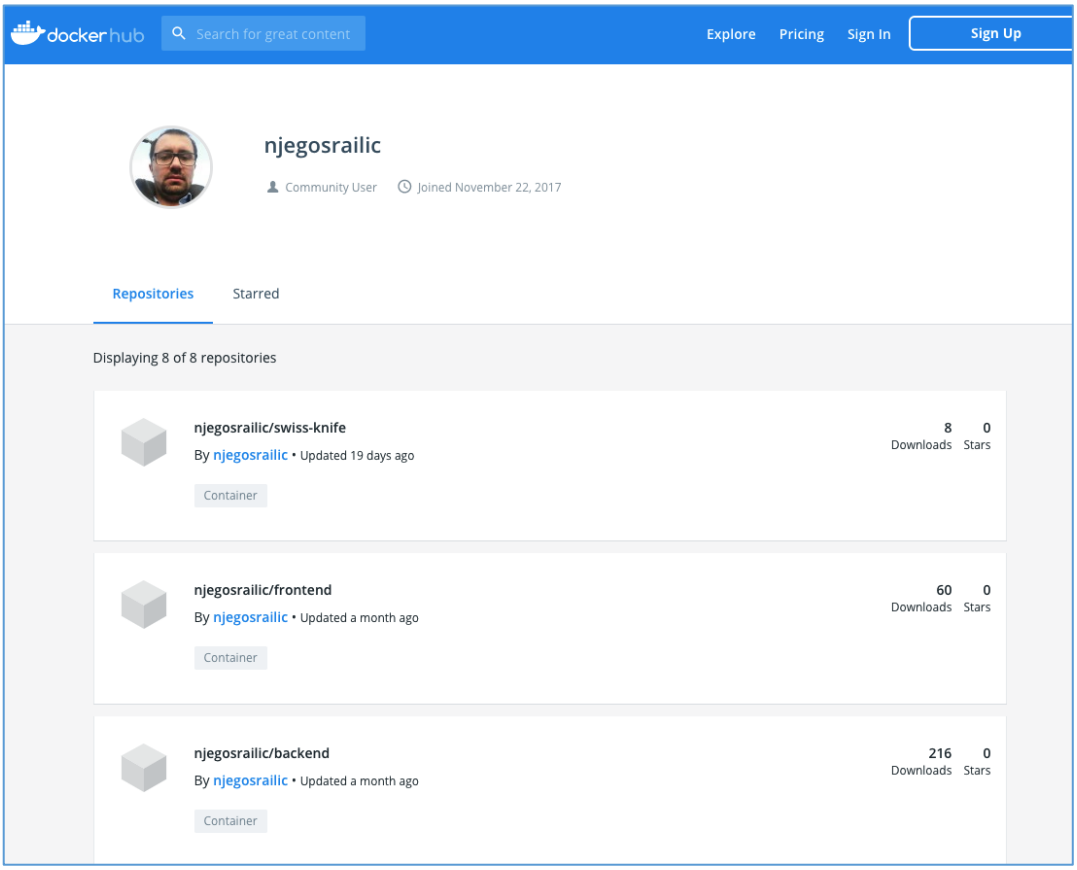
Lista instaliranih komponenata odnosno *Helm* paketa prikazana je na slici 6.10. Kao što se može vidjeti, instalirano je nekoliko paketa. Jedna od komponenata je *Nginx Ingress* [129] server koji se koristi kao pametni ruter u okviru platforme. Dodatno, instalirana je i komponenta *Flagger* [130] za progresivnu isporuku, koja je kasnije objašnjena. Kako je isporuka aplikacija praktično nemoguća bez monitoringa, instalirane su komponente *Prometheus* [131], *Grafana* [132] i *Kubernetes* server za metrike [133]. Kao što se može vidjeti sa slike, *Helm* paketi mogu biti smješteni lokalno unutar *charts* direktorijuma (*nginx-ingres*, *metrics-server*, *chartmuseum*) ili unutar određenog registra *Helm* paketa (*prometheus*, *grafana*, *Flagger*).

```
NAME            NAMESPACE       LABELS                  CHART                           VERSION
nginx-ingress   ingress         env:prod,job:nginx-ingress../../charts/nginx-ingress
prometheus      monitoring      env:prod,job:prometheus prometheus-community/prometheus 14.4.1
grafana         monitoring      env:prod,job:grafana    grafana/grafana                 6.14.1
flagger         flagger         job:flagger             flagger/flagger                 1.4.0
metrics-server  metrics-server  job:metrics-server      ../../charts/metrics-server
chartmuseum     chartmuseum     job:chartmuseum         ../../charts/chartmuseum
```

Slika 6.10. Lista instaliranih Helm paketa (eng. release)

6.1.4. Registar *Docker* slika

Za implementaciju ovog rješenja kao registar *Docker* slika izabran je javni *DockerHub* registar. Kredencijali za pristup ovom registru su konfigurisani unutar *Jenkins* servera, koristeći ugrađeni menadžer za upravljanje kredencijalima, a pristup im se vrši direktno iz CI/CD linije koristeći odgovarajući identifikator. Sve *Docker* slike koje su korištene za implementaciju praktičnog dijela rada su dostupne javno na adresi u okviru *DockerHub* registra: https://hub.*docker*.com/u/njegosrailic.



Slika 6.11. Javni registar Docker slika hostovan u okviru DockerHub platforme

6.1.5. Registar za *Helm* pakete

Instalaciju mikroservisne aplikacije moguće je izvršiti direktno iz direktorijuma koji sadrži odgovarajuću strukturu *Helm* paketa opisanu u poglavlju broj 5. Međutim, u praksi se često ovaj direktorijum arhivira i smješta kao standardna arhiva u *tar.gz* formatu. Ovim se omogućava brza izmjena konfiguracija u slučaju da se izmjene trebaju vratiti na neku prethodnu verziju. Za povrat na prethodnu verziju konfiguracije dovoljno je samo izmijeniti verziju paketa umjesto da se izmjena radi ponovo na samom repozitorijumu izvornog koda, koji mora proći kompletnu CI liniju i sve testove.

Za smještanje *Helm* paketa koriste se registri koji su veoma jednostavni HTTP serveri, implementirajući odgovarajući API za manipulaciju paketima. *ChartMuseum* registar otvorenog koda je izabran kao registar za *Helm* pakete. Ukoliko se na primjer, izvrši provjera servisa u okviru virtuelnog klastera pod nazivom *production*, upotrebom komande na slici 6.12, vidi se da je servis dostupan na IP adresi 10.233.7.244.

```
kubectl get svc -n production | grep chartmuseum
chartmuseum                     NodePort    10.233.7.244   <none>        8080:30303/TCP   460d
```

Slika 6.12. Prikaz ChartMuseum servisa u okviru Kubernetes platforme

6.1.6. CI infrastruktura

Za implementaciju praktičnog dijela rada kao CI alat izabran je *Jenkins*. *Jenkins* je alat otvorenog koda napisan u programskom jeziku JAVA sa velikim brojem dodataka kreiran za kontinualnu integraciju. Veliki broj dodataka obezbijeđuje integraciju sa različitim sistemima, čime se organizacijama omogućava da ubrzaju proces razvoja softvera automatizujući procese. *Jenkins* omogućava integraciju procesa životnog ciklusa razvoja softvera kao što su između ostalih: kreiranje artifakata, testiranje i statička analiza.

Jenkins koristi *master-slave* arhitekturu za upravljanje izvršavanjem zadataka. Server je zadužen za zakazivanje i raspoređivanje zadataka. Iako se na njemu mogu izvršavati i zadaci, uzimajući u obzir da je on veoma kritična tačka u kompletnoj arhitekturi, u praksi se zadaci izvršavaju na agentima koji se sa serverom mogu povezati preko SSH (*Secure Shell*) ili JNLP (*Java Network Launch Protocol*). U zavisnosti on načina na koji se kreiraju agenti, mogu biti statički odnosno unaprijed kreirane i konfigurisane virtuelne mašine, ili dinamički gdje se agenti kreiraju automatski u zavisnosti od potrebe, obično kao kontejneri.

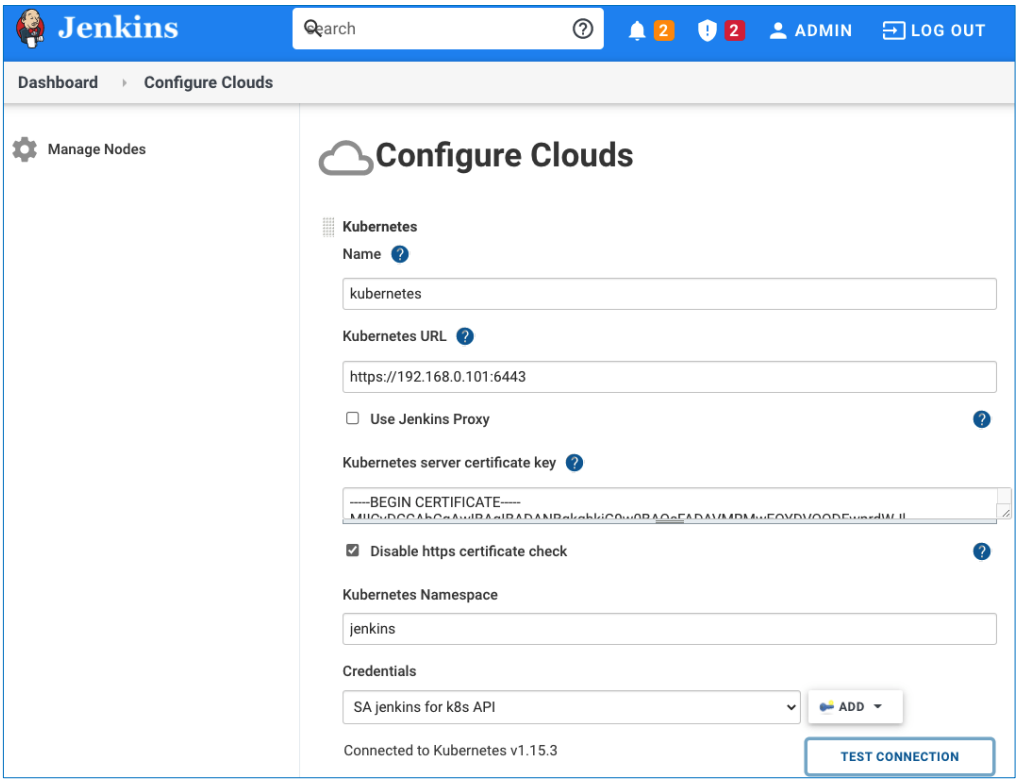
Jenkins dodaci obezbjeđuju integraciju sa tehnologijama poput *Docker*-a i *Kubernetes*-a, koje omogućuju kreiranje dinamičkih agenata i izvršavanje zadataka kako na pojedinačnim serverima na kojima se izvršava *Docker*, tako i u okviru *Kubernetes* platforme. Zadaci koji se izvršavaju tokom svog životnog ciklusa obično obrađuju određene podatke i mijenjaju okruženje u kojem se izvršavaju. Kako bi se osigurala ponovljivost izvršenja, obezbjeđujući da se svaki zadatak izvršava u zasebnom ali identičnom okruženju, u radu je korišten pristup sa dinamičkim agentima. Uzimajući u obzir već pomenute osobine kontejnera, oni se čine kao logičan izbor za upotrebu.

6.1.6.1. Integracija sa *Kubernetes* platformom

Jenkins master server je instaliran na zasebnoj virtuelnoj mašini i konfigurisan ručno. *Jenkins* je povezan na *Kubernetes* platformom upotrebom *Kubernetes* [134] dodatka. Nakon što se pokrene izvršenje određenog zadatka, *Jenkins* server kreira *Pod* koji predstavlja

dinamički agent u okviru *Kubernetes* platforme, koji će se nakon inicijalizacije povezati sa serverom. Potom će započeti izvršavanje zadataka, a agent će izvještavati server o statusu. Nakon izvršenja zadataka, *Pod* se kompletno briše.

Da bi se *Jenkins* serveru omogućilo kreiranje agenata u okviru *Kubernetes* platforme potrebno je iskonfigurisati okruženje i odgovarajuće permisije. Nakon što se kreira virtuelni klaster pod nazivom *jenkins*, neophodno je da se kreiraju RBAC (*Role-based access control*) uloge sa odgovarajućim permisijama. Pod tim se podrazumijeva kreiranje odgovarajućih *Kubernetes* objekata. Njihova specifikacija data je u fajlu *prakticni-rad/jenkins/rbac.yaml*, koji se nalazi na priloženom disku. Nakon što su objekti kreirani, neophodno je da se sa *Kubernetes* platforme eksportuje autentikacijski token, a zatim doda u skladište kredencijala u okviru *Jenkins* servera, kreirano upotrebom određenog (*Jenkins credentials plugin)* dodatka. Posljednji korak je dodavanje adrese *Kubernetes* API servisa i validacija konekcije i permisija.



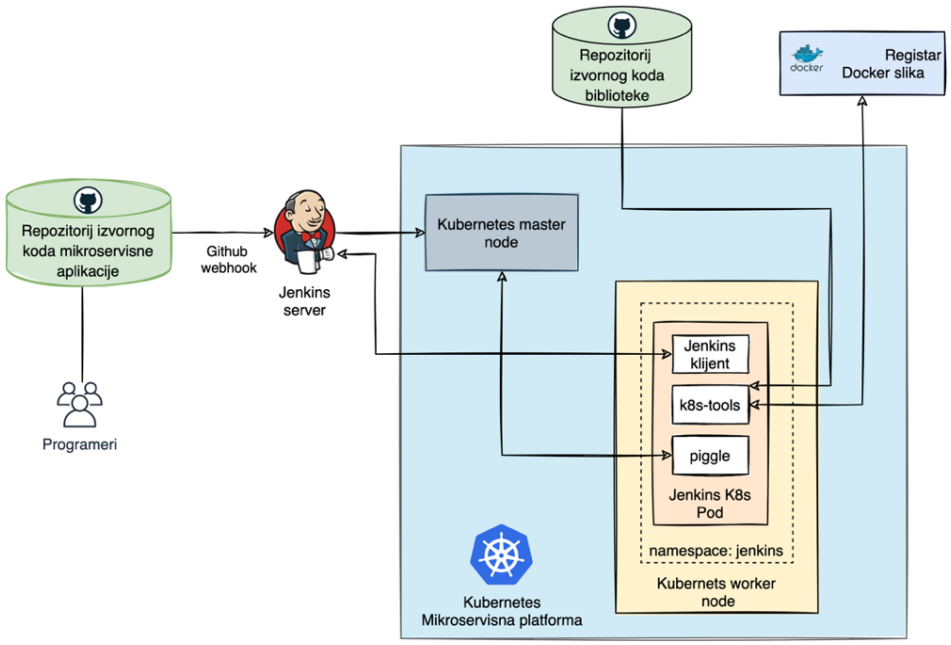
Slika 6.13. Administrativni panel za konfiguraciju Kubernetes dodatka

Na slici 6.14. prikazana je uopštena arhitektura komunikacije *Jenkins* CI servera sa *Kubernetes* platformom. Možemo vidjeti da se dinamički *Jenkins* agent odnosno *Pod* sastoji od tri kontejnera:

- *Jenkins* klijent – kreiran iz odgovarajuće *Docker* slike koja sadrži *Jenkins* klijent izvršni fajl kao i određene biblioteke neophodne za komunikaciju sa serverom;
- *k8s-tools* – *Docker* slika koja sadrži sve neophodne alate koji se koriste u okviru CI linije;
- *piggle* – CLI alat za orkestraciju isporuke aplikacija, koji je kasnije opisan u zasebnom poglavlju.

U okviru administrativnog panela *Kubernetes* dodatka dostupnog preko *Jenkins* UI interfejsa moguće je kreirati šablon za agenta. Međutim, u konkretnoj implementaciji odlučeno

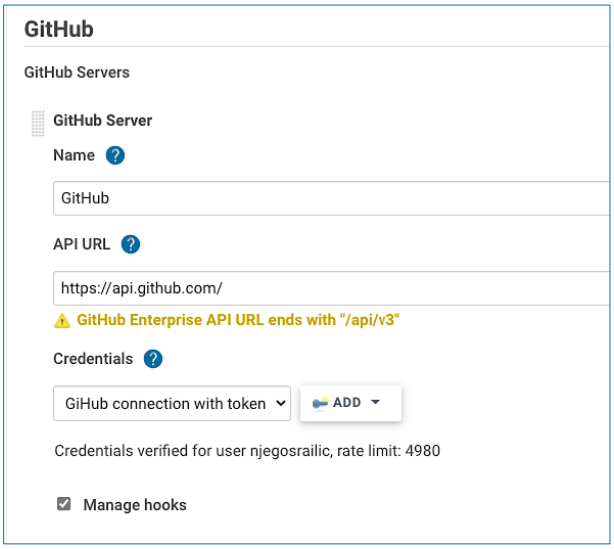
je da se šablon doda u okviru biblioteke (JSL implementirana u okviru praktičnog rada) kako bi se kompletna konfiguracija čuvala u okviru repozitorijuma i lakše kontrolisala.



Slika 6.14. Integracija Jenkins CI servera sa Kubernetes platformom i upotreba dinamičkih agenata

6.1.6.2. Integracija *Jenkins* CI servera sa repozitorijumima izvornog koda

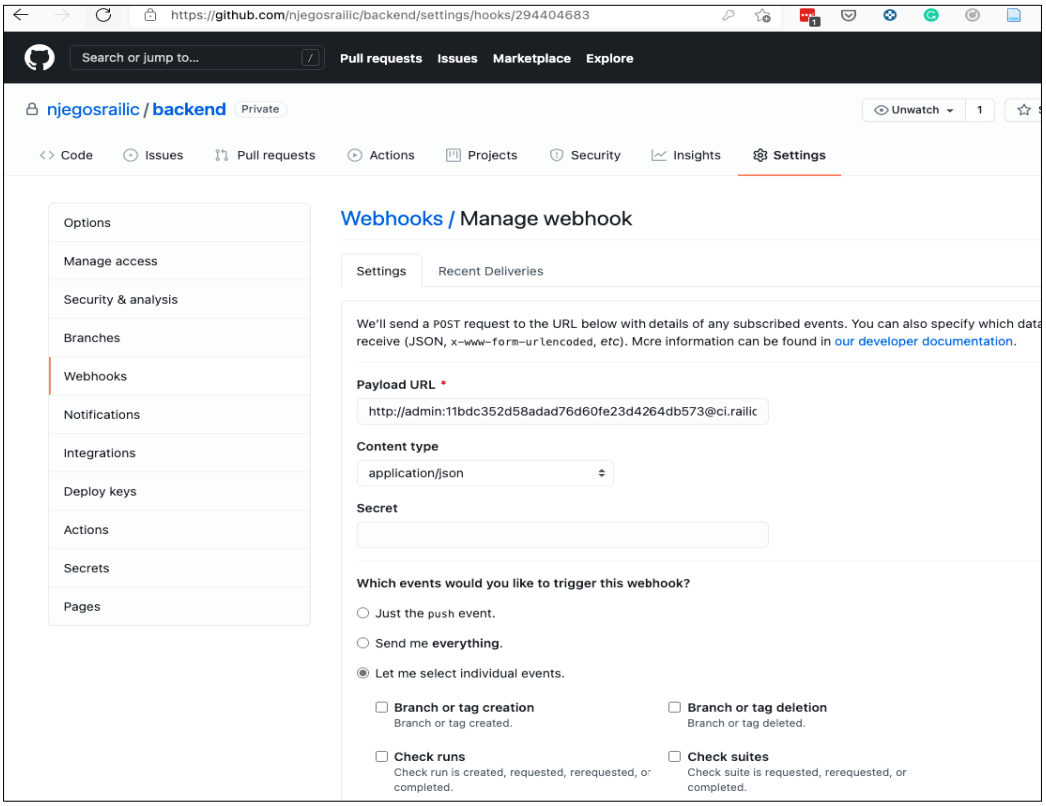
Komunikacija *Jenkins* CI servera sa *Github* platformom, koja omogućava jednostavniju administraciju i upravljanje repozitorijima izvornog koda koji koriste *Git* VCS, ostvarena je preko *Github* [135] dodatka. Ovdje je važno napomenuti da je korišten besplatan nalog koji, iako nema limite za kreiranje broja repozitorijuma, ima određene limite vezano za funkcionalnosti. Na slici 6.15. se može vidjeti da je dovoljno da se unese API adresa *Github* servera i izaberu određeni kredencijali. Kredencijali su kreirani u okviru *Github* naloga i dodati u *Jenkins* skladište za kredencijale. Ova konfiguracija omogućuje *Jenkins* CI serverima i agentima da pristupe bilo kojem repozitorijumu u okviru naloga sa korisničkim imenom „*njegosrailic*“.



Slika 6.15. Konfiguracija Github dodatka za integraciju sa Github servisom

6.1.6.3. *Webhook*

Gihub platforma omogućava da se bilo koja aplikacija pretplati za primanje događaja nastalih kao posljedica akcija koje se dešavaju u okviru određenog repozitorijuma izvornog koda. To se ostvaruje pomoću mehanizma pod nazivom *webhook* (eng. *web callback* ili *HTTP Push API*) pomoću kojeg jedna aplikacija pruža informacije drugim aplikacijama u realnom vremenu [136]. Za razliku od tipičnih API-ja gdje bi aplikacija morala da prati API server i sama preuzima podatke prilikom određenih događaja, *webhook* mehanizam omogućava automatsku isporuku informacija drugoj aplikaciji u realnom vremenu, slanjem (eng. *push*) istih preko API interfejsa. Nedostatak ovog koncepta je što aplikacija koja želi da primi informacije mora da implementira odgovarajući API. Međutim, *Jenkins* server ima implementiranu integraciju za *Github webhook* servis koju je neophodno konfigurisati.



Slika 6.16. Prikaz konfiguracije Github webhook servisa za repozitorijum izvornog koda pod nazivom backend

Da bi se ostvarila integracija *Jenkins* CI servera i *Github*-a, koja omogućuje da se kod određenih događaja šalju informacije *Jenkins* CI serveru, prvo je potrebno generisati odgovarajući token na strani *Jenkins* servere. Potom se taj token doda u konfiguraciju određenog repozitorijuma zajedno sa adresom *Jenkins* API servera, kao što je prikazano na slici 6.16. Ovaj token će *Github webhook* servis koristiti za autentikaciju i autorizaciju prilikom isporuke događaja *Jenkins* API serveru.

6.1.6.4. Repozitorijumi izvornog koda

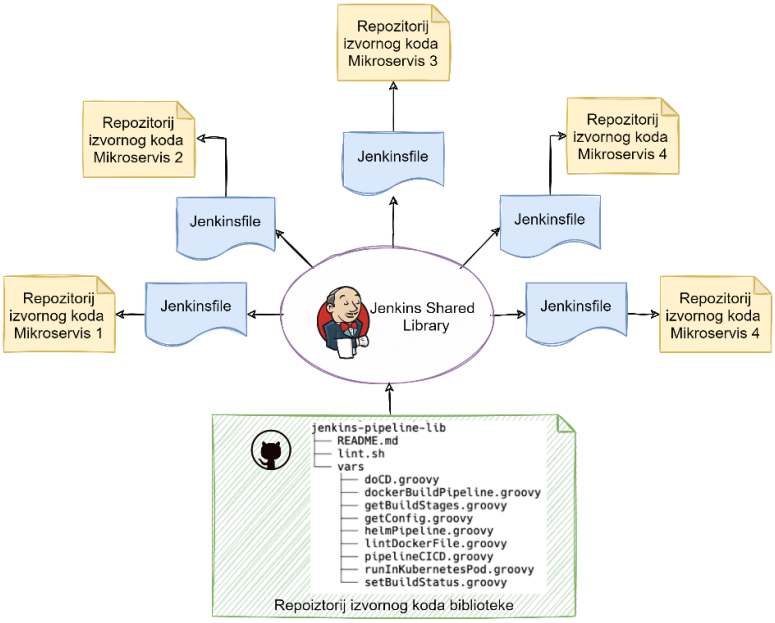
Za implementaciju praktičnog rada korišteno je nekoliko nekoliko mikroservisnih aplikacija, a kako bi se validiralo implementirano rješenje one su odvojene u zasebne repozitorijume izvornog koda. Lista korištenih repozitorijuma izvornog koda i njihov kratak opis prikazan je u tabeli 6.3.

Tabela 6.3. Prikaz korištenih repozitorijuma izvornog koda korištenih za implementaciju praktičnog rada sa kratkim opisom

Naziv repozitorijuma	Lokacija	Opis
frontend	Lokalno: prakticni-rad/frontend	Jednostavna UI aplikacija za validaciju praktičnog dijela rada napisana u React JS-u.
backend	Lokalno: prakticni-rad/backend	Jednostavna aplikacija koja implementira REST API interfejs za validaciju praktičnog dijela rada napisana u *Go* programskom jeziku.
jenkins-pipeline-lib	Lokalno: prakticni-rad/jenkins-pipeline-lib	JSL biblioteka za CI/CD
k8s-tools	https://*github*.com/njegosrailic/k8s-tools Lokalno: prakticni-rad/k8s-tools	Repozitorijum za kreiranje *Docker* slike koja sadrži alate neophodne za implementaciju CI/CD linije.
piggle	Lokalno: prakticni-rad/piggle	CLI alat za orkestraciju isporuke mikroservisnih aplikacija.

6.2. *Jenkins* dijeljena biblioteka za CI/CD

Pretpostavimo da svaka mikroservisna aplikacija ima svoj zaseban repozitorijum za izvorni kôd. Većina njih unutar linije za isporuku ima slične ili gotovo iste korake za kreiranje, testiranje i isporuku. Umjesto da se ti koraci ponavljaju za svaku aplikaciju, bilo bi poželjno da se zajednički kôd izdvoji u eksternu biblioteku koja bi se kasnije mogla referencirati i koristiti u drugim linijama. To je upravo način na koji je implementirano ovo rješenje upotrebom koncepta pod nazivom JSL (*Jenkins Shared Library*).



Slika 6.17. Prikaz organizacije repozitorijuma izvornog kôda JSL biblioteke

Izvorni biblioteke nalazi se u direktorijumu *prakticni-rad/jenkins-pipeline-lib*. Repozitorijum izvornog koda je dostupan na adresi: *https://github.com/njegosrailic/jenkins-pipeline-lib*, a nakon što se ova adresa doda u konfiguraciju *Jenkins* servera, druge linije mogu da koriste sve klase ili globalne varijable koje su definisane u JSL biblioteci. Konfiguracija i integracija biblioteke je detaljno objašnjena u drugom dijelu ovog poglavlja. Da bi se njima

pristupilo potrebno je da se unutar *Jenkins* fajla koristi *@Library* anotacija čime će se obezbijediti da se navedena biblioteka importuje prije nego linija započne sa izvršavanjem.

Upotreba ove biblioteke ima nekoliko pogodnosti. Prije svega ona može da se koristi za bilo koju mikroservisnu aplikaciju, bez obzira u kom je programskom jeziku napisana. Osim toga, ona omogućava timovima da jednostavno, u svega nekoliko linija koje je potrebno dodati unutar *Jenkinsfile* datoteke, dobiju potpuno funkcionalne linije za isporuku. Kako se ona nalazi u odvojenom repozitorijumu izvornog koda, sve izmjene se mogu jasno pratiti. Kako bi se jednostavno omogućilo dodavanje novih izmjena bez rizika da se izazovu problemi i nekompatibilnost sa prethodnim verzijama, koristi se semantičko verzionisanje. Time se omogućuje svakom timu da specifikuje tačno određenu verziju biblioteke koju želi da koristi unutar *Jenkinsfile* [137] fajla.

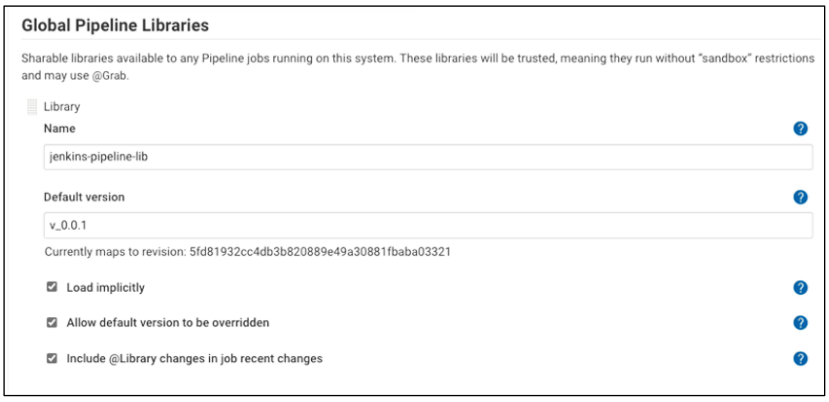
Dodatna pogodnost ove biblioteke je što ona sakriva svu kompleksnost linija za kreiranje i isporuku i programeri se mogu fokusirati na pisanje koda umjesto da rade na implementaciji dijelova CI/CD sistema. Samim tim unutar *Jenkinsfile* fajla oni specifikuju šta treba da se uradi dok biblioteka sadrži svu logiku o tome kako da se to uradi. Kao što se može vidjeti sa slike 6.18. biblioteka se sastoji od nekoliko fajlova od kojih su neki linije a drugi dijelovi linija koji sadrže kompleksnije funkcije ili funkcionalnosti koje se mogu koristiti u bilo kojoj drugoj liniji. Na primjer, *lintDockerfile.groovy* je u stvari jedna faza koja se može koristiti za validaciju sintakse *Dockefile* fajla u bilo kojoj drugoj liniji za isporuku. Nazivi fajlova su deskriptivni sami za sebe i opisuju ukratko namjenu svakog od njih.

```
jenkins-pipeline-lib
├── README.md
├── lint.sh
└── vars
    ├── doCD.groovy
    ├── dockerBuildPipeline.groovy
    ├── getBuildStages.groovy
    ├── getConfig.groovy
    ├── helmPipeline.groovy
    ├── lintDockerFile.groovy
    ├── pipelineCICD.groovy
    ├── runInKubernetesPod.groovy
    └── setBuildStatus.groovy
```

Slika 6.18. Prikaz stabla repozitorijuma izvornog koda Jenkins dijeljene bibiloteke

6.2.1. Konfiguracija biblioteke na strani *Jenkins* servera

Da bi se biblioteka mogla koristiti neophodno je da se konfiguriše na administrativnom panelu *Jenkins* servera. Konfiguracija je veoma jednostavna i sastoji se od dodavanja adrese izvornog koda. Osim toga, neophodno je da se doda naziv biblioteke, koji se kasnije koristi unutar *Jenkinsfile* fajla, kako bi se ova biblioteka referencirala.



Slika 6.19. Prikaz konfiguracije JSL biblioteke u okviru administrativnog panela Jenkins CI servera

Slika 6.20. Prikaz integracije JSL biblioteke sa Github serverom u okviru administrativnog panela Jenkins CI servera

6.2.2. Konfiguracija dinamičkih *Jenkins* agenata

Na slici 6.21. prikazana je konfiguracija dinamičkog agenta upotrebom direktive *PodTemplate* koja je sastavni dio *Kubernetes* dodatka za *Jenkins* server.

```
def call(Closure body) {
    def label = "k8s-worker-${UUID.randomUUID()}"
    def podServiceAccount = 'jenkins'

    podTemplate(
        label: label,
        containers: [
            containerTemplate(
                name: 'jnlp',
                image: 'jenkinsci/jnlp-slave:3.10-1-alpine',
                args: '${computer.jnlpmac} ${computer.name}'
            ),
            containerTemplate(
                name: 'docker',
                image: 'docker:latest',
                command: 'cat',
                ttyEnabled: true
            ),
            containerTemplate(
                name: 'k8s-tools',
                image: 'njegosrailic/k8s-tools:v1.0.1',
                command: 'cat',
                ttyEnabled: true
            ),
            containerTemplate(
                alwaysPullImage: true,
                name: 'piggle',
                image: 'njegosrailic/piggle:latest',
                command: 'cat',
                ttyEnabled: true
            ),
        ],
        volumes: [
            emptyDirVolume(mountPath: '/home/jenkins/agent/workspace', memory: true),
            hostPathVolume(mountPath: '/var/run/docker.sock', hostPath: '/var/run/docker.sock'),
            secretVolume(secretName: 'kube-config', mountPath: '/home/jenkins/.kube'),
            configMapVolume(configMapName: 'config', mountPath: '/etc/piggle/config')
        ],
    ) {
        node(label) {
            body()
        }
    }
}
```

Slika 6.21. Prikaz konfiguracije Pod šablona za dinamičke Jenkins agente

Na slici 6.14. dat je vizuelni prikaz komunikacije *Jenkins* agenta koji se izvršava u okviru *Kubernetes* platforme.

Kao što se može vidjeti iz priloženog koda, *Kubernetes Pod* se sastoji od četiri kontejnera:

- *jnlp* kontejner – *Jenkins* agent koji se preko JNLP protokola povezuje sa *Jenkins* master serverom. U okviru ovog kontejnera se kreira radni prostor (eng. *workspace*) što znači da će se svaki put u okviru ovog kontejnera kreirati lokalna kopija repozitorijuma izvornog koda mikroservisa koji je startovao liniju;
- *docker* – Kreiran iz standardne *Docker* slike koja sadrži binarni *Docker* fajl. Ovaj kontejner komunicira sa *Docker* demonom *Kubernetes* čvora na kojem se izvršava i koristi za kreiranje slika u okviru CI linije;
- *k8s-tools* – Kontejner koji sadrži binarne fajlove svih alata koji se koriste u okviru CI/CD linije, kao što su *Helm, Kubeval, Hadolint, Conftest* i drugi;
- *piggle* – Klijentska aplikacija za isporuku mikroservisnih aplikacija.

6.2.3. Konfiguracioni parametri

Kreiranje robusne, upravljive i višenamjenske linije za isporuku mikroservisnih aplikacija gotovo je nezamislivo bez upotrebe konfiguracionih parametara. Konfiguracioni parametri predstavljaju vrijednosti koje biblioteka prepoznaje, a omogućavaju centralno upravljanje funkcionalnostima obezbjeđujući veću fleksibilnost. Svaki ozbiljniji softverski proizvod ima mogućnost konfigurisanja, gdje se konfiguracija obično čuva u okviru tekstualnog fajla određene strukture i sintakse. Ovim se obezbjeđuje jedan vid dinamičke konfiguracije, gdje korisnici (ne obavezno programeri) mogu da unesu dinamiku u izvršavanje same linije što omogućava da se, na primjer, preskoče određeni koraci.

Ukoliko bi se za konfiguraciju linije koristio još jedan konfiguracioni fajl to bi značilo da se on mora čuvati u okviru samog repozitorijuma izvornog koda. U okviru linije bi bilo potrebno implementirati parsiranje ovog fajla i validaciju sintakse. Kako je *Jenkinsfile* fajl neophodan da bi se startovala sama linija, odlučeno je da se isti koristi za prosljeđivanje konfiguracionih parametara. Sama biblioteka već ima podrazumijevanu konfiguraciju, a u okviru fajla *getConfig.groovy* implementirana je logika koja vrši spajanje proslijeđenih sa podrazumijevanim parametrima. Svaka aplikacija, ukoliko je to potrebno, može da redefiniše konfiguraciju tako što će proslijediti konfiguracioni parametar u vidu para koji se sastoji od ključa i vrijednosti kao što je prikazano na slici 6.22.

```
def call(givenConfig = [:]) {
    def defaultConfig = [
    /**
      * The DockerHub profile URL, will be used for putting Docker images during this build.
      */
    'DockerHubURL': 'njegosrailic',
    /**
      * The DockerHub credetial ID used that will be used to fetch the credetial from Jenkins
credential store.
      */
    'DockerHubCredential': 'Dockerhub-credential',
    /**
      * The Docker image tagging startegy that will be used. Supported values are semver and
commitHash.
      */
    'DockerImageTaggingStrategy': 'commitHash',
    /**
      * The Github credential ID for accessing to Github.
      */
    'GithubCredentialId': 'ca7f1fce-d414-4e02-b90b-a7639b287b07',
    /**
      * The Helm registry for pusblishing Helm charts. This is a local chartmuseum instance
runnina on Kubernetes.
```

Slika 6.22. Prikaz dijela getConfig.groovy fajla

Da bi smo objasnili fleksibilnost ove dinamičke konfiguracije, pretpostavimo da se jedna ovakva CI/CD linija koristi u mikroservisnom okruženju jedne organizacije, čije okruženje broji preko hiljadu mikroservisnih aplikacija. U slučaju da je potrebno promijeniti adresu registra za smještanje *Helm* paketa, dovoljno je izvršiti promjenu jedne linije u okviru same biblioteke. Već prilikom sljedećeg izvršavanja *Jenkins* server će preuzeti novu verziju, a sama izmjena neće imati uticaj na linije koje ne koriste ovu podrazumijevanu vrijednost. U slučaju da linija ne nudi ovu mogućnost, to bi vjerovatno značilo izmjenu u svakom repozitorijumu izvornog koda ponaosob, što bi rezultovalo u hiljadu zahtjeva za integraciju, izvršavanje testova, recenziju izmjena i na kraju integraciju. Iz ovoga se može naslutiti koliko je resursa potrebno da bi se ove izmjene napravile u slučaju nepostojanja date funkcionalnosti.

6.3. CLI aplikacija

Aplikacija za orkestraciju isporuke i aktivacije mikroservisa napisana je u programskom jeziku *Go*. Na slici 6.23. prikazana je ogranizacija repozitorijuma izvornog koda kao i interfejs same aplikacije.

```
.
├── LICENSE
├── Makefile
├── README.md
├── cmd
│   ├── configurens.go
│   ├── deploy.go
│   ├── lint.go
│   ├── root.go
│   ├── clusters.go
│   └── template.go
├── config
│   └── config-map-config.yaml
├── Docker-compose.yaml
├── go.mod
├── go.sum
├── main.go
├── piggle.Dockerfile
├── pkg
│   ├── app
│   ├── client
│   ├── config
│   ├── constants
│   ├── Helm
│   ├── Kubernetes
│   └── util
└── test
    └── fixtures
```

```
--| piggle -h
Piggle CLI tool for deploying Kubernetes applications.
It uses Helm as a package manager. The deployments are being
configured
inside the app.yaml file supposed to live inside the root directory.

Usage:
 piggle [flags]
 piggle [command]

Available Commands:
 configurens Configure a namespace in K8s cluster.
 deploy    Performs deploy of a Kubernetes application
 help      Help about any command
 lint      This commands performs validation of app.yaml file
 template  Generate Helm templates from a chart and values
 clusters  Returns list of declared clusters in app.yaml

Flags:
 -h, --help      help for piggle
Use "piggle [command] --help" for more information about a command.
```

Slika 6.23. Prikaz organizacije repozitorijuma izvornog koda piggle aplikacije kao i CLI interfejsa

6.3.1. Isporuka i konfiguracija *piggle* aplikacije

Podrazumijevana isporuka aplikacije je u vidu *Docker* slike, a ukoliko je potrebno, moguće je kreirati artifakt za specifičnu arhitekturu upotrebom *make* alata i predefinisane *Makefile* datoteke, koja se nalazi u okviru repozitorijuma izvornog koda aplikacije. Kao što je prikazano na slici 6.14. *piggle* se izvršava u okviru *Jenkins* agent *Pod*-a kreiranog dinamički na *Kubernetes* platformi.

Za ispravno funkcionisanje aplikacije, neophodan je odgovarajući konfiguracioni fajl i kredencijali za pristup *Kubernetes* API serveru. Predviđeno je da se konfiguracioni fajl u formatu prikazanom na slici 6.24. smjesti u *Kubernetes ConfigMap* objekat a potom se kao volume mountuje na lokaciju /etc/piggle/config unutar *piggle* kontejnera. Kredencijali za pristup *Kubernetes* klasterima se smještaju u *Kubernetes Secret* objekat pod nazivom *kube-*

config, u formatu standardnog *kubectl* konfiguracionog fajla, a potom se postavi (eng. *mount*) na lokaciju /home/jenkins/.kube.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: config
 namespace: Jenkins
data:
 config.yaml: |
  clusters:
   - name: k8s-dev-c1
     context: Kubernetes-
admin@cluster.local
     environment: dev
     cloudProvider: aws
```

```
containerTemplate(
    alwaysPullImage: true,
    name: 'piggle',
    image: 'njegosrailic/piggle:latest',
    command: 'cat',
    ttyEnabled: true
  ),
],
volumes: [
  emptyDirVolume(mountPath: '/home/Jenkins/agent/workspace', memory: true),
  secretVolume(secretName: 'kube-config', mountPath: '/home/Jenkins/.kube'),
  configMapVolume(configMapName: 'config', mountPath: '/etc/piggle/config')
],
```

Slika 6.24. Prikaz konfiguracije piggle applikacije u okviru Kubernetes Pod-a

6.3.2. Specifikacija *app.yaml* fajla

Prilikom razvoja ovog rješenja prvenstveno se vodilo računa o tome kako ga učiniti što jednostavnijim za upotrebu, omogućiti pouzdanu i brzu isporuku uz očuvanje visokog stepena fleksibilnosti i proširivosti. Sa jedne strane imamo programere koji po uspješnom završetku izvršenja CI linije imaju *Docker* sliku i *Helm* paket spremne za isporuku u određeno aplikativno okruženje. Bez obzira na to kako je organizovan način isporuke softvera u okviru organizacije organizovan i da li organizacija ima centralizovan ili distribuiran tim za upravljanje isporukom (eng. *release engineering*), ideja kojom se vodilo prilikom dizajniranja sistema je omogućiti jednostavno upravljanje CD linijom. Tu se primarno misli na to da se od programera ne očekuje poznavanje *groovy* programskog jezika i pisanje linija za isporuku, već da se konfiguracijom specifikuje šta da se uradi, a da sama biznis logika unutar linije odluči kako da se to uradi. Kao rezultat, odlučeno je da se kreira konfiguraciona datoteka pod nazivom *app.yaml*, čija je specifikacija data u tabeli 6.4. Ovaj koncept nije nov i koristi se, na primjer, za *Google App Engine* [138], ali je sam dizajn i implementacija autentična.

```
version: '1.0'

common:
  appName: hello
  namespace: hello-app
  image:
    repository: njegosrailic/hello
    tag: 0.1.0

chart:
  name: hello
  version: 0.1.1

clusters:
  k8s-dev-c1:
    continuousDelivery: true
    replicaCount: 1
```

Slika 6.25. Primjer app.yaml fajla za aplikaciju pod nazivom hello

Svaka verzija mikroservisa definiše se u *app.yaml* datoteci koja predstavlja deskriptor mikroservisa i sadrži podatke neophodne za isporuku aplikacije u odgovarajuće okruženje. Osim za konfiguraciju aplikacije, ova datoteka se koristi kao konfiguracioni fajl za CD liniju. Primjer ove datoteke za aplikaciju pod nazivom *hello* prikazan je na slici 6.25. JSL biblioteka ne koristi direktno podatke iz ove datoteke, već sve operacije vezano za CD liniju prvenstveno delegira klijentskoj aplikaciji. Sama biblioteka je zadužena za izvršavanje koraka linije, a sva logika CD linije, je enkapsulirana u okviru *piggle* aplikacije. Prilikom izvršavanja CD linije sami koraci se generišu dinamički, u zavisnosti od broja klastera deklarisanih u *app.yaml* fajlu.

Isporuka aplikacije upotrebom *piggle* aplikacije se sastoji od sljedećih koraka:

- Prvi korak predstavlja inicijalizacija isporuke koji podrazumijeva osnovnu validaciju, kako bi se osiguralo da se isporuka može izvršiti. To obično podrazumijeva provjeru konekcije sa *Kubernetes* API serverom i validaciju kredencijala. Dodatno vrši se provjera da li određeni klaster postoji u konfiguraciji, kao i provjera *App* CRD objekta. Ovaj korak završava prikazom svih resursa koji će biti kreirani u okviru određenog klastera.
- Sljedeći korak je provjera statusa aplikacije i prikaz postojećih resursa.
- Instaliranje i konfigurisanje aplikacije.
- Prikaz rezultata isporuke.

Tabela 6.4. Opis specifikacije app.yaml fajla

Parametar	Opis parametra
`version`	Obavezan. Predstavlja verziju *app.yaml* fajla i ovaj parametar se koristi prvenstveno od strane *piggle* aplikacije za parsiranje i validaciju samog fajla.
`common`	Obavezan. Predstavlja parametre koji su zajednički za sve klastere specifikovane u okviru parametra `clusters`.
`common.appName`	Obavezan. Predstavlja naziv same aplikacije koje će *piggle* aplikacija koristiti prilikom instaliranja same aplikacije (eng. *release name*) upotrebom *Helm* menadžera za pakete.
`common.namespace`	Obavezan. Naziv virtuelnog klastera u koji će se aplikacija isporučiti.
`chart`	Obavezan. U okviru ovog parametra deklarišu se parametri vezani za naziv i verziju *Helm* paketa koji će se koristi za isporuku aplikacije.
`chart.name`	Obavezan. Naziv *Helm* paketa koji će se koristiti za instalaciju i konfiguraciju aplikacije.
`chart.version`	Obavezan. Verzija *Helm* paketa koji će se koristiti za instalaciju i konfiguraciju aplikacije.
`clusters`	Obavezan. Predstavlja listu klastera za isporuku aplikacije. Virtuelni klaster specifikovan `common.namespace` parametrom će biti kreiran u okviru *Kubernetes* klastera specifikovanih u ovoj listi. Da bi se određeni klaster mogao specifikovati u okviru ovog fajla neophodno je da prethodno bude dodat u konfiguraciju *piggle* aplikacije, kao što je to objašnjeno u prethodnom tekstu.
`continuousDelivery`	Opcion. Podrazumijevana vrijednost za razvojna okruženja je *false* dok je za produkciona *true*.

6.3.3. CRD objekat i problem kolizije

Resurs u okviru *Kubernetes* platforme je krajnja tačka (eng. *endpoint*) API-ja, koji sadrži kolekciju objekata određene vrste. Jedan primjer resursa je ugrađeni resurs *pods* koji sadži kolekciju *Pod* objekata.

Tabela 6.5. Opis specifikacije App CRD objekat

Element	Opis
`name`	Obavezan. Predstavlja verziju *app.yaml* fajla i ovaj parametar se koristi prvenstveno od strane *piggle* aplikacije za parsiranje i validaciju samog fajla.
`maintainer`	Obavezan. Predstavlja parametre koji su zajednički za sve klastere specifikovane u okviru parametra `clusters`.
`chartName`	Obavezan. Predstavlja naziv same aplikacije koje će *piggle* aplikacija koristiti prilikom instaliranja same aplikacije (eng. *release name*) upotrebom *Helm* menadžera za pakete.
`namespace`	Obavezan. Naziv virtuelnog klastera u koji će se aplikacija isporučiti.
`gitRepository`	Obavezan. U okviru ovog parametra deklarišu se parametri vezani za neziv i verziju *Helm* paketa koji će se korisiti za isporuku aplikacije.

Kako je *Kubernetes* proširiva platforma, ona omogućava proširenje sopstvenog API servera dodavanjem novih tipova (*Custom Resource Definition* – CRD) objekata. Da bi se kreirao objekat, potrebno je definisati ime (validno DNS ime), verziju kao i određenu specifikaciju objekta, nakon čega *Kubernetes* kreira odgovarajuću RESTful putanju za svaku verziju. U tabeli prikazana je specifikacija *App* CRD objekta kreiranog kao dio ovog sistema za čuvanje osnovnih podataka o aplikacijama isporučenih upotrebom CI/CD linije.

Da bi se ovaj CRD resurs mogao koristiti, potrebno je da se *App* CRD objekat kreira u okviru *Kubernetes* klastera specifikovanih u *piggle* konfiguraciji. To se obezbjeđuje instalacijom *Helm* paketa pod nazivom *app* verzije 0.1.0, koji sadrži šablon odnosno specifikaciju za *App* CRD objekat.

```
# Source: app/templates/app-crd.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: app.railic.net
  labels:
    app: release-name-app
    heritage: Tiller
    project: app
    release: release-name
spec:
  group: app.railic.net
  names:
    kind: App
    listKind: AppList
    plural: app
    singular: apps
    shortNames:
      - app
      - apps
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this representation
            of an object. Servers should convert recognized schemas to the latest
            internal value, and may reject unrecognized values. More info:
https://git.k8s.io/community/contributors/devel/api-conventions.md#resources'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase. More info:
https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds'
          type: string
        metadata:
          type: object
        status:
          type: object
        spec:
          description: 'Application description object'
          type: object
          properties:
            name:
              description: 'Application name'
              type: string
            maintainer:
              description: 'Maintainer'
              type: string
            chartName:
              description: 'Helm chart used for deployment'
              type: string
            namespace:
              description: 'Namespace name'
              type: string
            gitRepository:
              description: 'Application repository path'
              type: string
```

Slika 6.26. Prikaz specifikacije App CRD objekta

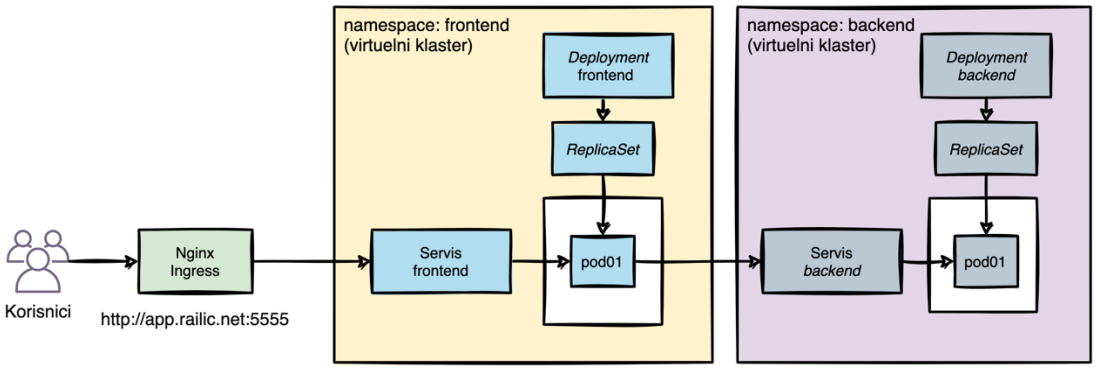
U slučaju da se pokuša isporuka mikroservisa pod istim imenom u isti virtuelni klaster, unutar kojeg je prethodno isporučen neki drugi mikroservis, *piggle* aplikacija će provjerom *App* objekta i parametara iz *app.yaml* fajla to detektovati i generisati grešku. Greška će biti generisana i u slučaju da se pokuša izmjena bilo kojeg od prethodno specifikovanih elemenata u okviru *app.yaml* fajla. Za svaki mikroservis koji je isporučen upotrebom ovog CD alata, kreira se po jedan *App* objekat u okviru *application* virtuelnog klastera, koji sadrži osnovne parametre vezane za mikroservis. Klijentska aplikacija *piggle* je zadužena za kreiranje i

ažuriranje ovog objekta prilikom svake isporuke, čime se izbjegava eventualna kolizija kao i mogućnost da se, na primjer, nehotice prebriše aplikacija koja se izvršava u produkcionom okruženju. Isto važi i ukoliko se želi promijeniti naziv *Helm* paketa koji se koristi za instalaciju aplikacije, posebno jer je potrebno prethodnu verziju deinstalirati i potom izvršiti instalaciju nove.

Ukoliko se bilo koji od ovih parametara želi promijeniti za već postojeću aplikaciju, neophodna je intervencija administratora *Kubernetes* klastera i brisanje tog parametra iz već postojećeg objekta. U zavisnosti od sigurnosne politike firme, programerima se mogu dati odgovarajuće permisije i omogućiti ručno ažuriranje ovog objekta upotrebom *kubectl* alata. Preduslov za ovo je postojanje odgovarajućeg modela permisija (eng. *ownership*) koje bi osigurale da programer može izvršiti izmjenu *App* objekta samo za određenu aplikaciju.

6.4. Mikroservisno okruženje za validaciju praktičnog dijela rada

Za validaciju praktičnog dijela rada kreirano je testno okruženje u okviru *Kubernetes* klastera koje se sastoji od dva mikroservisa. Pojednostavljen grafički prikaz okruženja dat je na slici 6.27. Mikroservisna aplikacija se sastoji od dva mikroservisa pod nazivima *frontend* i *backend*. *Frontend* predstavlja korisnički interfejs napisan upotrebom okvira *React* i dostupan je na adresi http://app.railic.net:5555. Interfejs se sastoji od jednostavnog tabelarnog prikaza logova, u formatu IP adresa i datum pristupa API servisu. Podaci se preuzimaju sa *backend* servisa, koji implementira jednostavan REST API interfejs napisan u programskom jeziku *Go*. Ova dva mikroservisa će biti korištena za testiranje funkcionalnosti implementiranog rješenja i detaljan prikaz načina rada isporuke i aktivacije novih verzija mikroservisa.



6.27. Pojednostavljena šema mikroservisnog okruženja za validaciju praktičnog dijela rada

6.4.1. Progresivna isporuka i uvođenje u eksploataciju nove verzije mikroservisa

Postoje dva načina za efektivnu aktivaciju nove verzije, a to su: upotreba zastavica koje omogućavaju da se određena funkcionalnost aktivira za specifičnu grupu korisnika i strategija kanarinca. Obično se ove dvije tehnike nadopunjuju, jer je praktično nemoguće izvršiti istovremenu aktivaciju više različitih verzija u istom okruženju upotrebom strategije kanarinca. Upotreba zastavica daje mnogo veću kontrolu, jer ukoliko se neka nova funkcionalnost aktivira zastavicom vrlo lako može i da se isključi. Kako se u potpuno automatizovanom procesu CD i progresivna isporuka često koriste zajedno, u praktičnom radu je takođe implementirana upotrebom već postojećeg alata pod nazivom *Flagger* [130] koji je integrisan u sam sistem.

Flagger je alat za progresivnu isporuku koji automatizuje proces aktivacije aplikacija u okviru *Kubernetes* platforme. Njegova glavna funkcionalnost ogleda se u omogućavanju kontrolisane aktivacije nove verzije mikroservisne aplikacije, zasnovanu na upotrebi

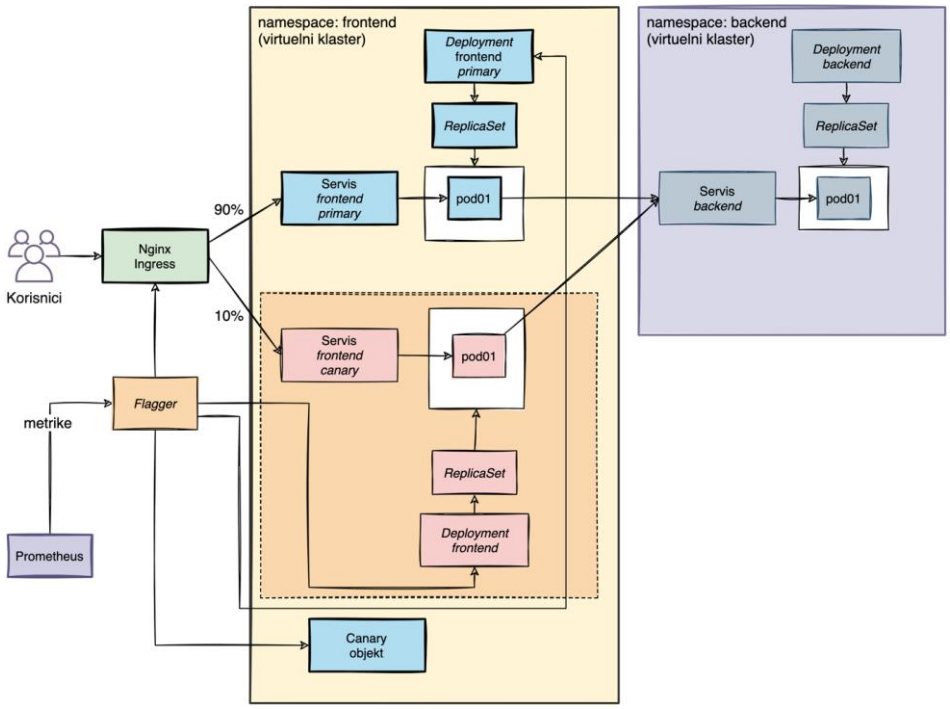
odgovarajućih metrika i automatsko izvršavanje određenih testova. *Flagger* implementira nekoliko strategija (strategija kanarinca, A/B testiranje, plavo-zelena isporuka) koristeći napredne platforme za rutiranje (*App Mesh* [139], *Istio* [140], *Linkerd* [141]) i usmjeravanje saobraćaja (NGINX [129], *Traefik* [142]). Osim toga, podržava integraciju sa većinom poznatih sistema za monitoring i sistema za slanje obavještenja.

Flagger se izvršava kao demon u okviru *Kubernetes* platforme, a implementiran je upotrebom šablona operatera (eng. *operator pattern*). Kako je *Kubernetes* proširiva platforma, operatori su softverske komponente (ekstenzije) koje omogućavaju da se određene operacije automatizuju upotrebom CRD objekata. Operatorski servis na osnovu CRD specifikacije izvršava određene operacije kako bi potrebne objekte doveo u željeno stanje.

```
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  creationTimestamp: "2021-08-01T11:52:25Z"
  generation: 6
  name: frontend-canary
  namespace: frontend
  resourceVersion: "30899218"
  selfLink: /apis/flagger.app/v1beta1/namespaces/frontend/canaries/frontend-canary
  uid: 389ffc6b-243a-4233-aa00-852ee874662e
spec:
  analysis:
    interval: 10s
    maxWeight: 50
    metrics:
    - interval: 1m
      name: error-rate
      thresholdRange:
        max: 5
    stepWeight: 5
    threshold: 10
    webhooks:
    - metadata:
        cmd: hey -z 1m -q 10 -c 2 http://app.railic.net:5555/
      name: load-test
      timeout: 5s
      url: http://flagger-loadtester.test/
  autoscalerRef:
    apiVersion: autoscaling/v2beta2
    kind: HorizontalPodAutoscaler
    name: frontend
  ingressRef:
    apiVersion: networking.k8s.io/v1beta1
    kind: Ingress
    name: frontend-micro-frontend
  progressDeadlineSeconds: 60
  provider: nginx
  service:
    port: 80
    targetPort: 80
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend-micro-frontend
status:
  canaryWeight: 0
  conditions:
  - lastTransitionTime: "2021-08-01T13:27:22Z"
    lastUpdateTime: "2021-08-01T13:27:22Z"
    message: Canary analysis completed successfully, promotion finished.
    reason: Succeeded
    status: "True"
    type: Promoted
  failedChecks: 0
  iterations: 0
  lastAppliedSpec: 76684c8b88
  lastPromotedSpec: 76684c8b88
  lastTransitionTime: "2021-08-01T13:27:22Z"
  phase: Succeeded
  trackedConfigs: {}
```

Slika 6.28. Primjer Flagger Canary objekta

Princip rada *Flagger* alata se zasniva na upotrebi *Canary* CRD objekta implemetiranog u okviru API kolekcije. Da bi se *Flagger* koristio za progresivnu isporuku nove verzije određenog mikroservisa, neophodno je kreiranje *Canary* objekta u okviru istog virtuelnog klastera. Primjer *Canary* objekta za *frontend* mikroservis dat je na slici 6.28. *Flagger* se izvršava u petlji i prati sve izmjene vezano za *Canary* objekte u klasteru na kom se izvršava. Nakon što *flagger* detektuje *Canary* objekat u okviru određenog virtuelnog klastera, on kreira dodatne objekte, između ostalog *Ingress* objekat koji će se koristiti za usmjeravanje saobraćaja na novu verziju, kao i replike servisa, Deployment, HPA (*Horizontal Pod Autoscaler*) itd, kao što je prikazano na slikama 6.28 i 6.29.



Slika 6.29. Prikaz arhitekture testnog okruženja sa Flagger alatom

```
➜ kubectl get ingress -n frontend
NAME                           HOSTS            ADDRESS         PORTS   AGE
frontend-micro-frontend        app.railic.net   192.168.0.103   80      31d
frontend-micro-frontend-canary app.railic.net   192.168.0.103   80      30d

➜ kubectl get all -n frontend
NAME                                                READY   STATUS    RESTARTS   AGE
pod/frontend-micro-frontend-primary-5c669dcddc-8crcd   1/1     Running   0          4h18m

NAME                                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/frontend-micro-frontend         NodePort    10.233.49.134   <none>        80:30333/TCP     31d
service/frontend-micro-frontend-canary  ClusterIP   10.233.18.209   <none>        80/TCP           30d
service/frontend-micro-frontend-primary ClusterIP   10.233.34.216   <none>        80/TCP           30d

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/frontend-micro-frontend         0/0     0            0           31d
deployment.apps/frontend-micro-frontend-primary 1/1     1            1           30d

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/frontend-micro-frontend-c77445445      0         0         0       4h48m
replicaset.apps/frontend-micro-frontend-primary-5456c68f5b 0     0         0       30d
replicaset.apps/frontend-micro-frontend-primary-79898c9dff 0     0         0       4h59m

NAME                                                  REFERENCE                                     TARGETS
MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/frontend          Deployment/frontend-micro-frontend            <unknown>/66%   1
4         0         31d
horizontalpodautoscaler.autoscaling/frontend-primary  Deployment/frontend-micro-frontend-primary    <unknown>/66%   1
4         1         30d

NAME                                STATUS      WEIGHT   LASTTRANSITIONTIME
canary.flagger.app/frontend-canary  Succeeded   0        2021-08-31T21:19:12Z
```

Slika 6.30. – Prikaz Kubernetes objekata u okviru frontend virtuelnog klastera

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    meta.helm.sh/release-name: frontend
    meta.helm.sh/release-namespace: frontend
    nginx.ingress.kubernetes.io/use-regex: "true"
  generation: 1
  labels:
    app.kubernetes.io/instance: frontend
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: micro-frontend
    helm.sh/chart: micro-frontend-0.1.0
  name: frontend-micro-frontend
  namespace: frontend
spec:
  rules:
  - host: app.railic.net
    http:
      paths:
      - backend:
          serviceName: frontend-micro-frontend
          servicePort: http
        path: /
```

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    meta.helm.sh/release-name: frontend
    meta.helm.sh/release-namespace: frontend
    nginx.ingress.kubernetes.io/canary: "false"
    nginx.ingress.kubernetes.io/use-regex: "true"
  labels:
    app.kubernetes.io/instance: frontend
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: micro-frontend
    helm.sh/chart: micro-frontend-0.1.0
  name: frontend-micro-frontend-canary
  namespace: frontend
  ownerReferences:
  - apiVersion: flagger.app/v1beta1
    blockOwnerDeletion: true
    controller: true
    kind: Canary
    name: frontend-canary
spec:
  rules:
  - host: app.railic.net
    http:
      paths:
      - backend:
          serviceName: frontend-micro-frontend-canary
          servicePort: http
        path: /
```

Slika 6.31. Uporedni prikaz Nginx Ingress objekta kreiranog koristeći Helm paket i odgovarajuće konfiguracione parametre sa objektom kreiranog automatski od strane Flagger alata

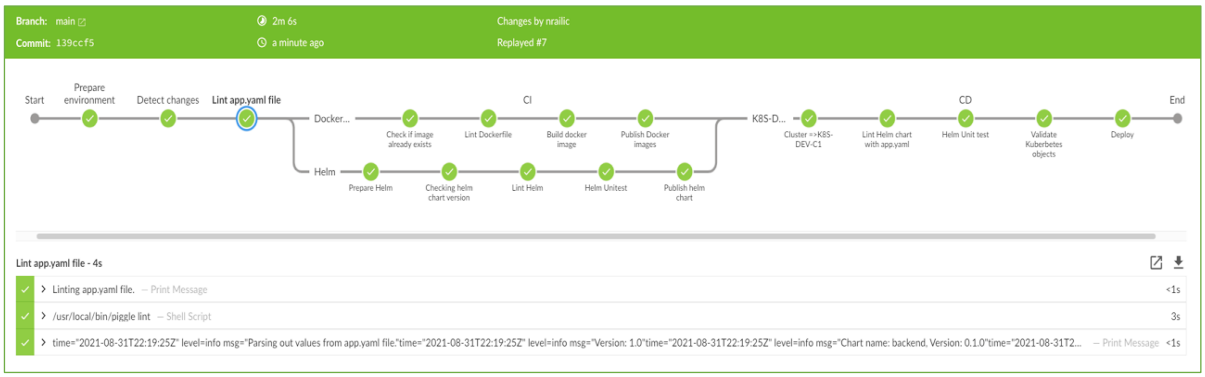
Nakon što *Flagger* kreira sve potrebne objekte, izvršavajući se kao demon, on posmatra *Deployment* objekat sa ciljem detekcije izmjena. Nakon što detektuje određenu promjenu, kreiraju se *Pod*-ovi sa novom verzijom aplikacije i startuje preusmeravanje određenog broja zahtjeva na novu verziju. *Flagger* se oslanja na metrike koje dobija od sistema za monitoring (u našem slučaju *Prometheus*), a dodatno može da izvršava i test performansi. Ukoliko je procenat grešaka u okviru zadatog, nakon što se zadovolje određeni preduslovi, stara verzija aplikacije će biti deaktivirana, u suprotnom aktivacija nove verzije se zaustavlja.

```yaml
version: '1.0'

common:
  appName: frontend
  namespace: frontend
  image:
    repository: njegosrailic/frontend
    tag: 0.3.1
chart:
  name: frontend
  version: 0.1.0

clusters:
  k8s-dev-c1:
    replicaCount: 1
    canary-deploy:
      flagger:
        analysis:
          interval: 10s
          maxWeight: 50
          metrics:
          - interval: 1m
            name: error-rate
            thresholdRange:
              max: 5
          stepWeight: 5
          threshold: 10
          webhooks:
          - metadata:
              cmd: hey -z 1m -q 10 -c 2 http://app.railic.net:5555/
            name: load-test
            timeout: 5s
            url: http://flagger-loadtester.test/
```

Slika 6.32. Prikaz app.yaml fajla sa konfiguracijom za progresivnu isporuku

Postavlja se pitanje kako integrisati *Flagger* sa CI/CD linijom i omogućiti programerima jednostavnu aktivaciju progresivne isporuke upotrebom *Flagger*-a i strategije kanarinca. Ukoliko jedna mikroservisna aplikacija želi da koristi ovaj alat, neophodno je da kreira CRD *Canary* objekat u okviru *Kubernetes* sistema sa određenim parametrima. Prvo logično rješenje koje se nameće je da se objekat doda unutar *app.yaml* fajla. Međutim, koristeći slično rješenje u praksi se pokazalo da ovo uvodi dodatnu kompleksnost jer je manipulacija jako velikim YAML fajlovima jako komplikovana i podložna greškama bez obzira na veliki broj alata za validaciju sintakse. Zbog toga je kreiran odvojen *Helm* paket koji sadrži šablon za CRD objekt. Svaka aplikacija koja želi da koristi ovaj objekat mora da uključi *canary-deploy Helm* paket. Nakon toga moguće je konfiguracione parametre za *Canary* objekat prosljeđivati direktno iz *app.yaml* fajla kao što je prikazano na slici 6.32.

Pretpostavimo sada da želimo isporučiti novu verziju *frontend* mikroservisa. Izmjena se sastoji od promjene boje pozadine na glavnoj strani korisničkog interfejsa. Programer je napravio zasebnu granu sa izmjenom i kreirao zahtjev za integraciju u glavnu granu. Nakon što je izmjena prošla sve testove isporučena je u testno okruženje. Nakon toga, zatražena je recenzija datog zahtjeva za integraciju od ostalih članova tima. Potom je izmjena odobrena i integrisana u glavnu granu. Kao što možemo vidjeti na slici 6.32, CI/CD linija je uspješno završena kao i isporuka aplikacije što je ujedno i posljednji korak linije.



Slika 6.33. Prikaz CI/CD linije za kontinualnu isporuku mikroservisnih aplikacija kroz Jenkins korisnički interfejs

Nakon što je izvršavanje CI/CD linije uspješno završeno, to znači da su svi objekti ažurirani u skladu sa konfiguracionim parametrima iz *app.yaml* fajla. U konkretnom slučaju izvršena je promjena verzije *frontend* servisa odnosno izmjena *tag* vrijednosti u okviru *Deployment* objekta za dati mikroservis. Na ovom primjeru možemo jasno da vidimo razliku između isporuke (eng. *deploy*) mikroservisa i aktivacije (eng. *release*). U trenutku neposredno poslije same isporuke, bez obzira na to što je nova verzije isporučena, ona još uvijek nije u upotrebi, tačnije ne servisira korisničke zahtjeve. Ukoliko bi se za isporuku koristile strategije koje nudi sama *Kubernetes* platforma, na primjer *rolling updates* strategija, u zavisnosti od same konfiguracije, kreirao bi se određen procenat *Pod*-ova sa novom verzijom, a nakon što oni prođu validaciju ispravnosti, postepeno preusmjeravao saobraćaj na njih uz minimalan rizik i bez prekida. Na kraju bi se *Pod*-ovi koji serviraju staru verziju (0.3.0) aplikacije, postepeno brisali. Međutim, kako se za implementaciju praktičnog rada koristio *Flagger* jer nudi mogućnost progresivne isporuke, on je zadužen za upravljanje saobraćajem kao i aktivaciju nove verzije.

Kao što je rečeno, *Flagger* u petlji koristeći *Kubernetes* API prati izmjene na *Deployment* objektu *frontend* mikroservisa, jer je *Canary* objekat kreiran u okviru *frontend* virtuelnog

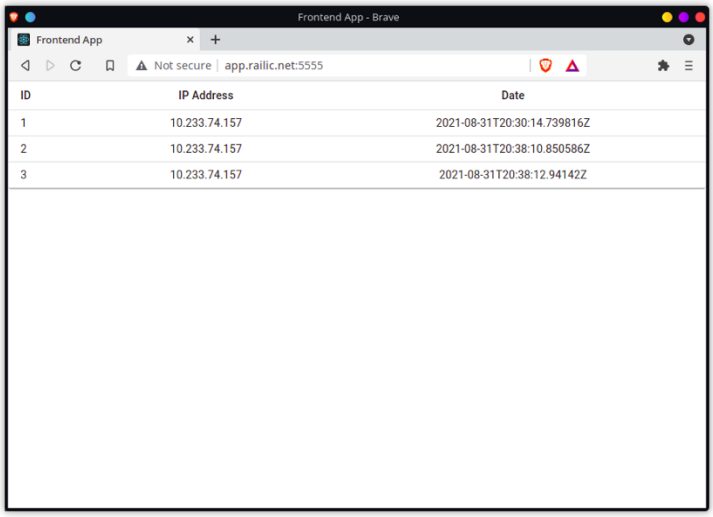
klastera. Nakon što *Flagger* detektuje izmjenu verzije *Docker* slike u *Deployment* objektu, on kreira novi *Pod* sa novom verzijom slike 0.3.1. Nakon što novi *Pod* prođe validaciju, započinje se progresivna isporuka koristeći strategiju kanarinca. Na osnovu prethodno prikazanog *Canary* objekta, on će započeti sa preusmjeravanjem 5% zahtjeva na novu verziju i taj procenat će se uvećavati za 5% svakih deset sekundi. Takođe, možemo da vidimo da je prag tolerancije na greške podešen na 5% u vremenskom intervalu od jedne minute.

```
NAME              STATUS      WEIGHT   LASTTRANSITIONTIME
frontend-canary   Succeeded   0        2021-08-31T20:50:52Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   5        2021-08-31T21:17:02Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   10       2021-08-31T21:17:12Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   15       2021-08-31T21:17:22Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   20       2021-08-31T21:17:32Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   25       2021-08-31T21:17:42Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   30       2021-08-31T21:17:52Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   35       2021-08-31T21:18:02Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   40       2021-08-31T21:18:12Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   40       2021-08-31T21:18:12Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   40       2021-08-31T21:18:12Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   45       2021-08-31T21:18:22Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Progressing   50       2021-08-31T21:18:34Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Promoting     0        2021-08-31T21:18:42Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Finalising    0        2021-08-31T21:19:02Z
NAME              STATUS        WEIGHT   LASTTRANSITIONTIME
frontend-canary   Succeeded     0        2021-08-31T21:19:12Z
```

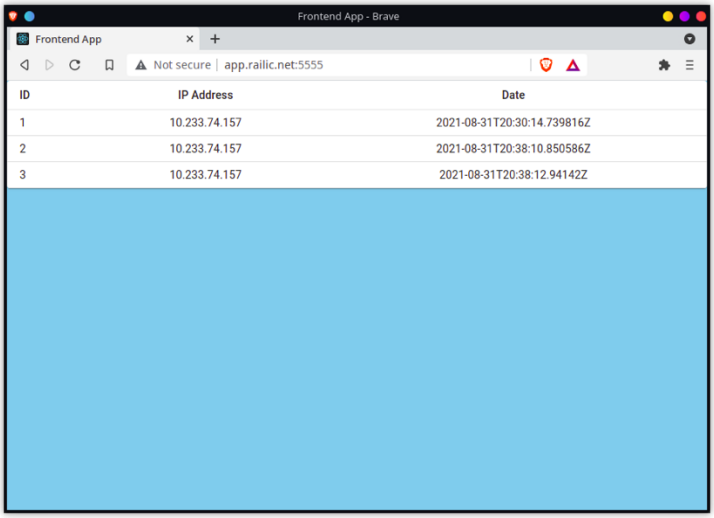
Slika 6.34. – Prikaz progresivne isporuke nove verzije backend mikroservisa

Navedene tvrdnje se mogu dokazati posmatranjem logova *Canary* objekta, kao što je prikazano na slici 6.34. Isporuka će da traje dok se ne završi uspješna tranzicija na novu verziju ili dok nivo grešaka ne pređe prag tolerancije. Ukoliko procent grešaka pređe prag tolerancije, progresivna isporuka će biti prekinuta, a stara verzija aplikacije će nastaviti sa izvršavanjem. Ukoliko bi korisnik pristupio *frontend* web aplikaciji i povremeno osvježavao stranicu, on bi naizmjenično dobijao prikaz stare i nove (verzija sa plavom pozadinom) aplikacije sve dok traje isporuka kao što je to prikazano na slikama 6.35 i 6.36. Nakon što je isporuka uspješno završena stari pod se briše i rekreira sa novom verzijom mikroservisa. Time se obezbjeđuje da objekti kao što su servis, *Deployment* i *Pod*-ovi koji su označeni kao primarni izvršavaju trenutnu verziju aplikacije, primarno zaduženu za serviranje korisničkih zahtjeva.

Flagger alat se oslanja na metrike koje dobija od *Prometheus* monitoring alata koji periodično prikuplja metrike kako o samom *nginx ingress* kontroleru tako i o aplikativnim *Pod*-ovima a te metrike se mogu prikazati na primjer u okviru samog *Prometheus* alata, a za napredne vizuelizacije se može koristiti na primjer *Grafana* alat. *Flagger* je praktično nemoguće koristiti bez odgovarajućih metrika. Dodatno, važno je napomenuti da *Flagger* ima dodatne funkcionalnosti kao, na primjer, simuliranje korisničkih zahtjeva upotrebom određenog *weebhook*-a, što omogućava progresivnu isporuku i aktivaciju čak i ako nema trenutno aktivnih korisnika.

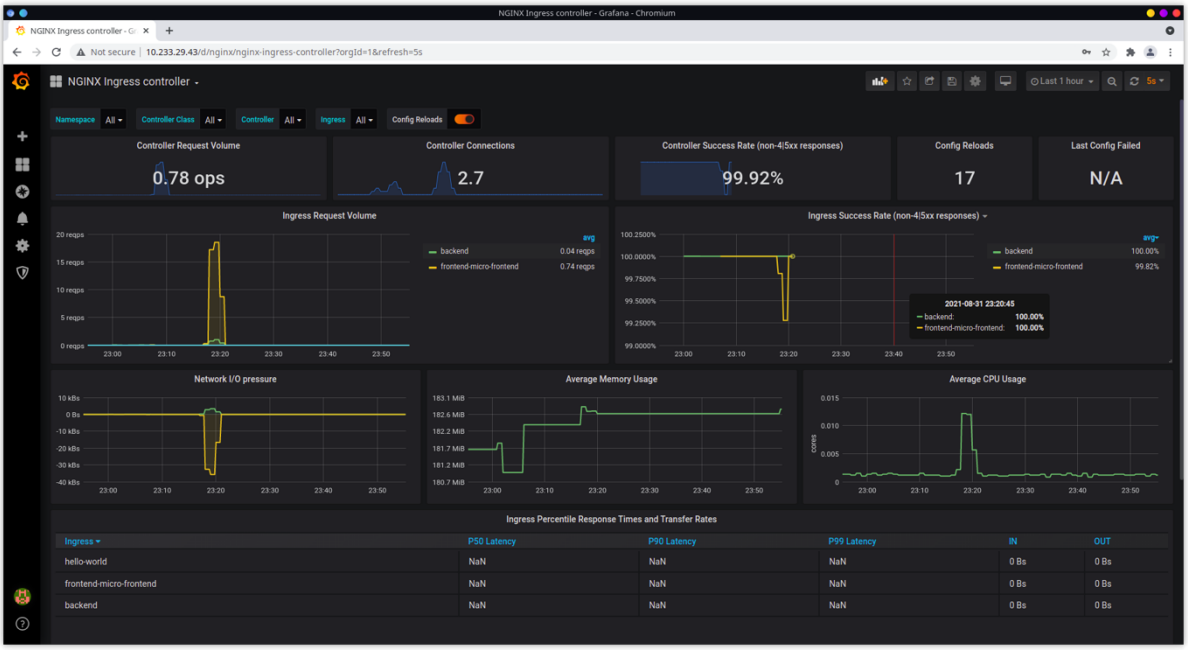


Slika 6.35. Prikaz korisničkog interfejsa frontend aplikacije verzija 0.3.0



Slika 6.36. Prikaz korisničkog interfejsa frontend aplikacije verzija 0.3.1

Slika 6.37. Grafički prikaz metrika prilikom progresivne isporuke nove verzije frontend mikroservisa i preusmjeravanja saobraćaja upotrebom strategije kanarinca



Slika 6.38. Grafana alat za monitoring – prikaz aktivnosti na strani Nginx Ingress servera prilikom progresivne isporuke

7. ZAKLJUČAK

U ovom radu opisan je pristup rješenju problema implementacije kontinualne isporuke za mikroservisne aplikacije, odnosno CI/CD procesa u korporativnom okruženju. Da bi se shvatio kontekst problema, u radu je dat i kratak pregled za svaki od problema ponaosob. Kako rad obrađuje probleme povezane sa efikasnom isporukom i aktivacijom aplikacija u mikroservisnom okruženju, dat je kratak pregled oblasti virtuelizacije i kontejnerskih tehnologija, kao i najpopularnijih orkestratora za mikroservisnu arhitekturu sa posebnim osvrtom na *Kubernetes* platformu.

Rješenje za isporuku mikroservisnih aplikacija, koje je razvijeno u ovom radu, se sastoji od dva dijela. Prvi dio predstavlja dijeljena biblioteka za *Jenkins* server koja implementira CI/CD liniju i funkcionalnosti koje se prvenstveno odnose na kontrolu izvršenja toka same linije, kao i dio logike primarno vezan za CI liniju. Drugi dio predstavlja klijentska aplikacija za orkestraciju isporuke mikroservisnih aplikacija.

Upotrebom tehnologija otvorenog koda i industrijskih standarda implementiran je modularan CI/CD sistem koji je olakšao dizajn, implementaciju kao i buduća proširenja i prilagođavanja sistema. Na primjer, kreiranje novih linija ili zamjena CD alata bio bi trivijalan korak, jer bi CI linija ostala nepromijenjena. Zbog svog dizajna alat se može koristiti u organizacijama različitih veličina uz eventualna minimalna prilagođavanja. Implementirano rješenje je poboljšana verzija rješenja koje se koristi u okruženju koje broji preko četiri hiljade inženjera i nekoliko hiljada mikroservisnih aplikacija, što dodatno potvrđuje prethodne navode.

Rad daje pregled trenutnog stanja na polju isporuke mikroservisnih aplikacija i kao takav može se iskoristiti kao polazna osnova za buduća istraživanja. Arhitektura i modularnost implementiranog rješenja omogućavaju da se uz izmjenu postojećih koraka i dodavanje novih, implementira sistem za kontinualnu isporuku u organizaciji bilo koje veličine. Automatizujući korake kreiranja artifakata, testiranja i isporuke obezbjeđuje se visok kvalitet koda bez unošenja dodatnih grešaka, pri čemu je kreirana verzija artifakta uvijek spremna za isporuku u produkciono okruženje. Time se omogućava visokofrekventna, efikasna i pouzdana isporuka mikroservisnih aplikacija na dnevnoj bazi.

LITERATURA

[1] N. Sam, *Building Microservices: Designing Fine-Grained systems*. 2015.

[2] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "An empirical study of architecting for continuous delivery and deployment," *Empir. Softw. Eng.*, vol. 24, no. 3, pp. 1061–1108, Jun. 2019, doi: 10.1007/s10664-018-9651-4.

[3] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2010.

[4] E. Fadda, P. Plebani, and M. Vitali, "Monitoring-aware Optimal Deployment for Applications based on Microservices," *IEEE Trans. Serv. Comput.*, pp. 1–1, 2019, doi: 10.1109/TSC.2019.2910069.

[5] "Continuous Integration," *Martin Fowler*, Maj 01, 2006. https://martinfowler.com/articles/continuousIntegration.html (posljednja posjeta Avg. 20, 2020).

[6] S. Danilo, "Canary Release," *Martin Fowler*, Jun. 25, 2014. https://martinfowler.com/bliki/CanaryRelease.html (posljednja posjeta Jan. 01, 2020).

[7] D. Farley and J. Humble, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[8] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, Apr. 2018, pp. 39–397. doi: 10.1109/ICSA.2018.00013.

[9] M. Flower and J. Lewis, "Microservices," *martinfowler.com*. https://martinfowler.com/articles/microservices.html (posljednja posjeta Apr. 09, 2021).

[10] C. D. Graziano, *Performance analysis of xen and kvm hypervisors for hosting the xen worlds.* Place of publication not identified: Proquest, Umi Dissertatio, 2012.

[11] A. Hat Red, "Ansible is Simple IT Automation." http://www.ansible.com (posljednja posjeta Maj 02, 2021).

[12] P. Webteam, "Powerful infrastructure automation and delivery | Puppet." https://puppet.com/ (posljednja posjeta Maj 02, 2021).

[13] "Terraform by HashiCorp," *Terraform by HashiCorp*. https://www.terraform.io/ (posljednja posjeta Maj 08, 2021).

[14] "Infrastructure as Code: Chef, Ansible, Puppet, or Terraform?" https://www.ibm.com/cloud/blog/chef-ansible-puppet-terraform (posljednja posjeta Maj 02, 2021).

[15] "What is virtualization?," *Opensource.com*. https://opensource.com/resources/virtualization (posljednja posjeta Maj 13, 2021).

[16] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," in *Proceedings of the fourth symposium on Operating system principles - SOSP '73*, Not Known, 1973, p. 121. doi: 10.1145/800009.808061.

[17] "What is a hypervisor?" https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor (posljednja posjeta Maj 13, 2021).

[18] "VMware vSphere Documentation." https://docs.vmware.com/en/VMware-vSphere/index.html (posljednja posjeta Maj 14, 2021).

[19] "Hyper-V Technology Overview." https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview (posljednja posjeta Maj 14, 2021).

[20] “What is Oracle VM Server?” https://docs.oracle.com/cd/E50245_01/E50249/html/vmcon-ovm-server-overview.html (posljednja posjeta Maj 14, 2021).
[21] “Oracle VM VirtualBox.” https://www.virtualbox.org/ (posljednja posjeta Maj 14, 2021).
[22] “VMware Workstation Player.” https://www.vmware.com/products/workstation-player.html (posljednja posjeta Maj 08, 2021).
[23] “QEMU.” https://www.qemu.org/ (posljednja posjeta Maj 17, 2021).
[24] “Paravirtualization,” *SearchServerVirtualization*. https://searchservervirtualization.techtarget.com/definition/paravirtualization (posljednja posjeta Maj 13, 2021).
[25] “KVM,” *KVM Org*. https://www.linux-kvm.org/page/KVM_Features (posljednja posjeta Maj 17, 2021).
[26] “Xen Project,” *Xen Project*. https://xenproject.org/ (posljednja posjeta Maj 17, 2021).
[27] “Linux Containers.” https://linuxcontainers.org/ (posljednja posjeta Maj 17, 2021).
[28] “Empowering App Development for Developers | Docker.” https://www.docker.com/ (posljednja posjeta Maj 17, 2021).
[29] “Introduction to Solaris Zones.” https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html (posljednja posjeta Maj 17, 2021).
[30] “Open source container-based virtualization for Linux.,” *OpenVz*. https://openvz.org/ (posljednja posjeta Maj 17, 2021).
[31] “Jails,” *The FreeBSD Project*. https://docs.freebsd.org/en/books/handbook/jails/ (posljednja posjeta Maj 17, 2021).
[32] “Operating System Containers vs. Application Containers | @RisingStack,” *RisingStack Engineering - Node.js Tutorials & Resources*, Maj 19, 2015. https://blog.risingstack.com/operating-system-containers-vs-application-containers/ (posljednja posjeta Apr. 17, 2021).
[33] “ip-netns(8) - Linux manual page.” https://man7.org/linux/man-pages/man8/ip-netns.8.html (posljednja posjeta Maj 13, 2021).
[34] “veth(4) - Linux manual page.” https://man7.org/linux/man-pages/man4/veth.4.html (posljednja posjeta Maj 13, 2021).
[35] M. Lukša, *Kubenretes in Action*. Shelter Island, NY: Manning Publications Co, 2018.
[36] “OCI Image Format,” *GitHub*. https://github.com/opencontainers/image-spec (posljednja posjeta Maj 17, 2021).
[37] D. J. Walsh, “A history of low-level Linux container runtimes,” *Opensource.com*. https://opensource.com/article/18/1/history-low-level-container-runtimes (posljednja posjeta Apr. 04, 2021).
[38] “capabilities(7) - Linux manual page.” https://man7.org/linux/man-pages/man7/capabilities.7.html (posljednja posjeta Maj 17, 2021).
[39] “Docker Hub.” https://hub.docker.com/ (posljednja posjeta Maj 02, 2021).
[40] “Docker Registry - JFrog - JFrog Documentation.” https://www.jfrog.com/confluence/display/JFROG/Docker+Registry (posljednja posjeta Maj 18, 2021).
[41] “Nexus Repository OSS - Software Component Management | Sonatype.” https://www.sonatype.com/products/repository-oss (posljednja posjeta Avg. 29, 2021).
[42] “Harbor.” https://goharbor.io/ (posljednja posjeta Maj 18, 2021).
[43] “Container Registry,” *Google Cloud*. https://cloud.google.com/container-registry (posljednja posjeta Maj 18, 2021).
[44] “Azure Container Registry | Microsoft Azure.” https://azure.microsoft.com/en-us/services/container-registry/ (posljednja posjeta Maj 18, 2021).

[45] “Dockerfile reference,” *Docker Documentation*, Maj 17, 2021. https://docs.docker.com/engine/reference/builder/ (posljednja posjeta Maj 18, 2021).
[46] “Networking overview,” *Docker Documentation*, Maj 10, 2021. https://docs.docker.com/network/ (posljednja posjeta Maj 13, 2021).
[47] “Kubernetes Documentation.” https://kubernetes.io/docs/ (posljednja posjeta Jan. 02, 2021).
[48] “Docker Swarm, Orchestration,” *Docker Documentation*, Apr. 08, 2021. https://docs.docker.com/get-started/orchestration/ (posljednja posjeta Apr. 08, 2021).
[49] “Apache Mesos,” *Apache Mesos*. http://mesos.apache.org/ (posljednja posjeta Apr. 09, 2021).
[50] “Kubernetes adoption, security, and market share trends report,” *StackRox: Kubernetes and container security solution*. https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/ (posljednja posjeta Apr. 03, 2021).
[51] “Kubernetes - Google Kubernetes Engine (GKE),” *Google Cloud*. https://cloud.google.com/kubernetes-engine (posljednja posjeta Maj 18, 2021).
[52] “Amazon EKS | Managed Kubernetes Service | Amazon Web Services,” *Amazon Web Services, Inc.* https://aws.amazon.com/eks/ (posljednja posjeta Maj 18, 2021).
[53] “Azure Kubernetes Service (AKS) | Microsoft Azure.” https://azure.microsoft.com/en-us/services/kubernetes-service/ (posljednja posjeta Maj 18, 2021).
[54] “Overview of Docker Compose,” *Docker Documentation*, Maj 06, 2021. https://docs.docker.com/compose/ (posljednja posjeta Maj 08, 2021).
[55] “Marathon: A container orchestration platform for Mesos and DC/OS.” https://mesosphere.github.io/marathon/ (posljednja posjeta Apr. 09, 2021).
[56] “Apache Hadoop.” https://hadoop.apache.org/ (posljednja posjeta Maj 08, 2021).
[57] “Apache Kafka,” *Apache Kafka*. https://kafka.apache.org/ (posljednja posjeta Maj 08, 2021).
[58] “Chronos: Fault tolerant job scheduler for Mesos.” https://mesos.github.io/chronos/ (posljednja posjeta Maj 08, 2021).
[59] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, Bordeaux France, Apr. 2015, pp. 1–17. doi: 10.1145/2741948.2741964.
[60] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.
[61] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009, doi: 10.1145/1435417.1435432.
[62] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, “Flexible update propagation for weakly consistent replication,” in *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP ’97*, Saint Malo, France, 1997, pp. 288–301. doi: 10.1145/268998.266711.
[63] “ETCD,” *ETCD*. https://etcd.io/ (posljednja posjeta Maj 21, 2021).
[64] “Options for Highly Available topology,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/ (posljednja posjeta Apr. 04, 2021).
[65] “Container runtimes,” *Kubernetes*. https://kubernetes.io/docs/setup/production-environment/container-runtimes/ (posljednja posjeta Apr. 04, 2021).
[66] “CoreDNS: DNS and Service Discovery.” https://coredns.io/ (posljednja posjeta Apr. 04, 2021).
[67] M. Lukša, *Kubernetes in action*. Shelter Island, NY: Manning Publications Co, 2018.

[68] “CNI Specification,” *GitHub*. https://github.com/containernetworking/cni (posljednja posjeta Maj 19, 2021).
[69] *flannel-io/flannel*. flannel-io, 2021. Posljednja posjeta: Apr. 04, 2021. [Online]. Available: https://github.com/flannel-io/flannel
[70] “Project Calico – Secure Networking for the Cloud Native Era.” https://www.projectcalico.org/ (posljednja posjeta Apr. 04, 2021).
[71] “Weave Net: Network Containers Across Environments | Weaveworks.” https://www.weave.works (posljednja posjeta Apr. 04, 2021).
[72] “Service,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/ (posljednja posjeta Maj 10, 2021).
[73] “Comparing kube-proxy modes: iptables or IPVS? – Project Calico.” https://www.projectcalico.org/comparing-kube-proxy-modes-iptables-or-ipvs/ (posljednja posjeta Maj 13, 2021).
[74] K. Beck and C. Andres, *Extreme programming explained: embrace change*, 2nd ed. Boston, MA: Addison-Wesley, 2005.
[75] “Building a CI/CD pipeline for microservices on Kubernetes - Azure Architecture Center.” https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd-kubernetes (posljednja posjeta Apr. 10, 2021).
[76] B. O’Sullivan, *Mercurial: the definitive guide*, 1st ed. Sebastopol, CA: O’Reilly Media, Inc, 2009.
[77] “Apache Subversion.” http://subversion.apache.org/ (posljednja posjeta Jun. 11, 2021).
[78] “Concurrent Versions System.” http://savannah.nongnu.org/projects/cvs (posljednja posjeta Jun. 11, 2021).
[79] “Perforce Software | Development Tools For Innovation at Scale.” https://www.perforce.com/ (posljednja posjeta Jun. 11, 2021).
[80] “Git.” http://git-scm.com/ (posljednja posjeta Jun. 11, 2021).
[81] “Mercurial SCM.” https://www.mercurial-scm.org/ (posljednja posjeta Jun. 11, 2021).
[82] “Bazaar.” http://bazaar.canonical.com/en/ (posljednja posjeta Jun. 11, 2021).
[83] N. McAllister, “Linus Torvalds’ BitKeeper blunder,” *InfoWorld*, Maj 02, 2005. https://www.infoworld.com/article/2670360/linus-torvalds--bitkeeper-blunder.html (posljednja posjeta Jul. 21, 2021).
[84] “Git - Book.” https://git-scm.com/book/sr/v2 (posljednja posjeta Jun. 12, 2021).
[85] “zlib Home Site.” https://zlib.net/ (posljednja posjeta Jun. 13, 2021).
[86] “Patterns for Managing Source Code Branches,” *martinfowler.com*. https://martinfowler.com/articles/branching-patterns.html (posljednja posjeta Jun. 16, 2021).
[87] “Semantic Versioning 2.0.0.” https://semver.org (posljednja posjeta Apr. 02, 2021).
[88] “Introduction to Code Signing.” https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms537361(v=vs.85) (posljednja posjeta Jun. 13, 2021).
[89] M. Roberts, “Signing and Verifying Container Images.” https://www.openshift.com/blog/signing-and-verifying-container-images (posljednja posjeta Jun. 13, 2021).
[90] “Skopeo - Small, secure, daemon-less container tool for inspecting, moving, signing, and verifying Open Container Initiative (OCI) compliant container images.” https://catalog.redhat.com/software/containers/detail/5dca40fdbed8bd164a06aaf3 (posljednja posjeta Jun. 13, 2021).
[91] *Container Signing, Verification and Storage in an OCI registry*. sigstore, 2021. Posljednja posjeta: Jun. 13, 2021. [Online]. Available: https://github.com/sigstore/cosign

[92] “Change Management | IT Process Wiki,” *IT Process Wiki - the ITIL® Wiki*. https://wiki.en.it-processmaps.com/index.php/Change_Management (posljednja posjeta Apr. 18, 2021).
[93] “Extreme Programming Values.” http://www.extremeprogramming.org/values.html (posljednja posjeta Apr. 20, 2021).
[94] L. Crispin and J. Gregory, *Agile testing: a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley, 2009.
[95] H. Vocke, “The Practical Test Pyramid,” *martinfowler.com*. https://martinfowler.com/articles/practical-test-pyramid.html (posljednja posjeta Apr. 02, 2021).
[96] J. Popović, *Testiranje softvera u praksi*. CET, 2012.
[97] “Introducing Progressive Delivery,” *SD Times*, Maj 11, 2021. https://sdtimes.com/devops/introducing-progressive-delivery/ (posljednja posjeta Avg. 13, 2021).
[98] “Freshservice ITSM System | ITIL-aligned service desk software.” freshservice.com/itsm/release-management (posljednja posjeta Maj 22, 2021).
[99] “Application deployment and testing strategies,” *Google Cloud*. Posljednja posjeta: Okt. 31, 2021. [Online]. Available: https://cloud.google.com/architecture/application-deployment-and-testing-strategies
[100] “Jenkins,” *Jenkins*. https://www.jenkins.io/https:/www.jenkins.io/ (posljednja posjeta Jul. 14, 2021).
[101] J. F. Smart, *Jenkins: the definitive guide*. Beijing: O’Reilly Media, 2011.
[102] “Helm.” https://helm.sh/ (posljednja posjeta Jul. 14, 2021).
[103] “ChartMuseum - Helm Chart Repository.” https://chartmuseum.com/ (posljednja posjeta Avg. 29, 2021).
[104] “GitHub: Where the world builds software,” *GitHub*. https://github.com/ (posljednja posjeta Jul. 14, 2021).
[105] “7 Pipeline Design Patterns for Continuous Delivery,” *SingleStone*, Jul. 01, 2020. https://www.singlestoneconsulting.com/blog/7-pipeline-design-patterns-for-continuous-delivery/ (posljednja posjeta Jul. 15, 2021).
[106] “8 CI/CD Best Practices for Your DevOps Journey,” *CloudBees*. https://www.cloudbees.com/blog/8-cicd-best-practices-your-devops-journey (posljednja posjeta Jul. 15, 2021).
[107] “Consul by HashiCorp,” *Consul by HashiCorp*. https://www.consul.io/ (posljednja posjeta Avg. 29, 2021).
[108] *Hadolint, Dockerfile linter, validate inline bash, written in Haskell*. Haskell Dockerfile Linter, 2021. Posljednja posjeta: Jul. 14, 2021. [Online]. Available: https://github.com/hadolint/hadolint
[109] “The Twelve-Factor App.” https://12factor.net/ (posljednja posjeta Jul. 14, 2021).
[110] “Package Management with Advanced Packaging Tool (APT),” *Ubuntu*. https://ubuntu.com/server/docs/package-management (posljednja posjeta Jul. 14, 2021).
[111] K. Chinthaguntla, “Linux package management with YUM and RPM,” *Enable Sysadmin*. https://www.redhat.com/sysadmin/how-manage-packages (posljednja posjeta Jul. 14, 2021).
[112] “Conftest, Write tests against structured configuration data.” https://www.conftest.dev/ (posljednja posjeta Avg. 29, 2021).
[113] “Rego, Policy Language,” *Open Policy Agent*. https://openpolicyagent.org/docs/latest/policy-language/ (posljednja posjeta Avg. 29, 2021).

[114] “OPA Gatekeeper: Policy and Governance for Kubernetes,” *Kubernetes*, Avg. 06, 2019. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/ (posljednja posjeta Avg. 29, 2021).
[115] “Kubeval, validate one or more Kubernetes configuration files.” https://kubeval.instrumenta.dev/ (posljednja posjeta Avg. 29, 2021).
[116] “Netlify: Develop & deploy the best web experiences in record time,” *Netlify*. https://www.netlify.com/ (posljednja posjeta Jul. 24, 2021).
[117] “Proxmox - Powerful open-source server solutions.” https://www.proxmox.com/en/ (posljednja posjeta Avg. 29, 2021).
[118] “Packer by HashiCorp,” *Packer by HashiCorp*. https://www.packer.io/ (posljednja posjeta Avg. 29, 2021).
[119] “Complete Kickstart Guide (How to Save Time Installing Linux),” *Linux Today*, Maj 09, 2019. https://www.linuxtoday.com/blog/install-kickstart-linux/ (posljednja posjeta Avg. 29, 2021).
[120] “How Goroutines Work.” https://blog.nindalf.com/posts/how-goroutines-work/ (posljednja posjeta Maj 23, 2021).
[121] “Cloud Object Storage S3 | Store & Retrieve Data Anywhere | Amazon Simple Storage Service (S3),” *Amazon Web Services, Inc.* https://aws.amazon.com/s3/ (posljednja posjeta Jul. 24, 2021).
[122] “Google Cloud Storage,” *Google Cloud*. https://cloud.google.com/storage (posljednja posjeta Jul. 24, 2021).
[123] “Azure Blob Storage | Microsoft Azure.” https://azure.microsoft.com/en-us/services/storage/blobs/ (posljednja posjeta Jul. 24, 2021).
[124] “Kubespray - Deploy a Production Ready Kubernetes Cluster.” https://kubespray.io/ (posljednja posjeta Avg. 29, 2021).
[125] “Cloud Services - Amazon Web Services (AWS),” *Amazon Web Services, Inc.* https://aws.amazon.com/ (posljednja posjeta Avg. 29, 2021).
[126] “Cloud Computing Services,” *Google Cloud*. https://cloud.google.com/ (posljednja posjeta Avg. 29, 2021).
[127] “Cloud Computing Services | Microsoft Azure.” https://azure.microsoft.com/en-us/ (posljednja posjeta Avg. 29, 2021).
[128] “Helmfile, Deploy Kubernetes Helm Charts,” Avg. 29, 2021. https://github.com/roboll/helmfile (posljednja posjeta Avg. 29, 2021).
[129] “NGINX Ingress Controller.” https://kubernetes.github.io/ingress-nginx/ (posljednja posjeta Avg. 29, 2021).
[130] “Flagger.” https://flagger.app/ (posljednja posjeta Avg. 13, 2021).
[131] “Prometheus - Monitoring system & time series database.” https://prometheus.io/ (posljednja posjeta Avg. 29, 2021).
[132] “Grafana: The open observability platform,” *Grafana Labs*. https://grafana.com/ (posljednja posjeta Avg. 29, 2021).
[133] “Kubernetes Metrics Server.” https://github.com/kubernetes-sigs/metrics-server (posljednja posjeta Avg. 29, 2021).
[134] “Kubernetes plugin for Jenkins.” https://plugins.jenkins.io/kubernetes (posljednja posjeta Avg. 05, 2021).
[135] “GitHub Jenkins Plugin,” *GitHub*. https://plugins.jenkins.io/github (posljednja posjeta Avg. 05, 2021).
[136] “What’s a Webhook?,” *SendGrid*, Jun. 24, 2014. https://sendgrid.com/blog/whats-webhook/ (posljednja posjeta Avg. 08, 2021).

[137] “Using a Jenkinsfile,” *Using a Jenkinsfile*. https://www.jenkins.io/doc/book/pipeline/jenkinsfile/https:/www.jenkins.io/doc/book/pipeline/jenkinsfile/ (posljednja posjeta Avg. 08, 2021).
[138] “app.yaml Reference | App Engine standard environment for Python 2,” *Google Cloud*. https://cloud.google.com/appengine/docs/standard/python/config/appref (posljednja posjeta Avg. 31, 2021).
[139] “App Mesh - Application-level networking for all your services.” https://aws.amazon.com/app-mesh/ (posljednja posjeta Avg. 30, 2021).
[140] “Istio,” *Istio*. https://istio.io/latest/ (posljednja posjeta Avg. 30, 2021).
[141] “Linkerd, The world’s lightest, fastest service mesh.” https://linkerd.io/ (posljednja posjeta Avg. 30, 2021).
[142] “Traefik.” https://doc.traefik.io/traefik/ (posljednja posjeta Sep. 01, 2021).

Biografija

Njegoš Railić, dipl. inž. elektrotehnike, rođen je 29.01.1986. godine u Sanskom Mostu. U Prnjavoru je završio osnovnu školu 2000. godine, a zatim i srednju Elektrotehničku školu 2004. godine. Osnovne studije, na studijskom programu Računarstvo i informatika, upisao je 2004. godine na Elektrotehničkom fakultetu Univerziteta u Banjoj Luci, a diplomirao u novembru 2019. godine (ocjenom 10) na temu "*Failover Active Directory Domain Controller* servisa na *Windows* 2008 server operativnom sistemu" sa prosječnom ocjenom 8.72. Studije II ciklusa studija, na studijskom programu Računarstvo i informatika na Elektrotehničkom fakultetu Univerziteta u Banjoj Luci, upisao je 2009. godine, na kojem je položio sve ispite sa prosječnom ocjenom 9,2.

Od jula 2006. do novembra 2009. godine bio je zaposlen na Elektrotehničkom fakultetu, Banja Luka, gdje je kao student bio angažovan na poslovima sistemske administracije i razvoju aplikacija za podršku rada studentske službe. Od januara 2010. do novembra 2010. godine bio je zaposlen u Republičkom hidrometeorološkom zavodu, Banja Luka, gdje je na poziciji sistem inženjera bio angažovan na poslovima sistem i mrežne administracije. Zatim je od decembra 2010. do avgusta 2016. godine bio zaposlen u preduzeću *Asseco-SEE* BiH, filijala Banja Luka, gdje je kao sistem inženjer bio zadužen za rješenja iz oblasti IT infrastrukture kao i implementaciji različitih sistema za bankarsko poslovanje. Od oktobra 2016. godine je zaposlen u preduzeću *Wayfair GMBH*, u Berlinu, gdje je na poziciji vodećeg SRE (eng. Site Reliability Engineer) angažovan na razvoju IT infrastrukture sačinjene od brojnih servisa uključujući sisteme za upravljanje konfiguracijom, dinamičko skaliranje i orkestraciju servisa. Na ovoj poziciji primarno je fokusiran na projektovanje i razvoj mirkoservisne arhitekture, kao i autoamtizaciju procesa.

Прилог 2.

УНИВЕРЗИТЕТ У БАЊОЈ ЛУЦИ
ПОДАЦИ О АУТОРУ ОДБРАЊЕНОГ МАСТЕР/МАГИСТАРСКОГ РАДА

Име и презиме аутора мастер/магистарског рада: **Његош Раилић**

Датум, мјесто и држава рођења аутора: **29.01.1986, Сански Мост, БиХ**

Назив завршеног факултета/Академије аутора и година дипломирања:

Електротехнички факултет Универзитета у Бањој Луци, 2009. године

Датум одбране завршног/дипломског рада аутора: **12.11.2009.**

Наслов завршног/дипломског рада аутора: ***Failover Active Directory Domain Controller*** **servisa na *Windows* 2008 server operativnom sistemu**

Академско звање коју је аутор стекао одбраном завршног/дипломског рада:

Дипломирани инжењер електротехнике – 240 ECTS

Академско звање које је аутор стекао одбраном мастер/магистарског рада:

Магистар рачунарства и информатике

Назив факултета/Академије на коме је мастер/магистарски рад одбрањен:

Електротехнички факултет Универзитета у Бањој Луци

Наслов мастер/магистарског рада и датум одбране:

Имплементација процеса за континуирано увођење у експлоатацију микросервиса, 21.12.2021.

Научна област мастер/магистарског рада према CERIF шифрарнику: **T120**

Имена ментора и чланова комисије за одбрану мастер/магистарског рада:

Проф. др Зоран Ђурић, предсједник
Доц. др Михајло Савић, ментор
Проф. др Дражен Брђанин, члан

У Бањој Луци, дана 7.12.2021.

Декан

_____

Прилог 3.

Изјава 1

ИЗЈАВА О АУТОРСТВУ

Изјављујем да је
мастер/магистарски рад

Наслов рада: Имплементација процеса за континуирано увођење у експлоатацију микросервиса

Наслов рада на енглеском језику: The Implementation of Continuous Deployment Process for Microservices

- ☒ резултат сопственог истраживачког рада,
- ☒ да мастер/магистарски рад, у цјелини или у дијеловима, није био предложен за добијање било које дипломе према студијским програмима других високошколских установа,
- ☒ да су резултати коректно наведени и
- ☒ да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

У Бањој Луци 7.12.2021

Потпис кандидата

Изјава 2

Изјава којом се овлашћује Електротехнички факултет
Универзитета у Бањој Луци да мастер/магистарски рад учини јавно доступним

Овлашћујем Електротехнички факултет Универзитета у Бањој Луци да мој мастер/магистарски рад, под насловом

Имплементација процеса за континуирано увођење у експлоатацију микросервиса

који је моје ауторско дјело, учини јавно доступним.

Мастер/магистарски рад са свим прилозима предао/ла сам у електронском формату, погодном за трајно архивирање.

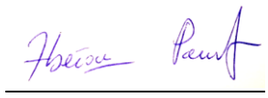
Мој мастер/магистарски рад, похрањен у дигитални репозиторијум Универзитета у Бањој Луци, могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице *(Creative Commons)*, за коју сам се одлучио/ла.

1. Ауторство
2. Ауторство - некомерцијално
3. **Ауторство - некомерцијално - без прераде**
4. Ауторство - некомерцијално - дијелити под истим условима
5. Ауторство - без прераде
6. Ауторство - дијелити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

У Бањој Луци 7.12.2021

Потпис кандидата



Изјава 3

Изјава о идентичности штампане и електронске верзије мастер/магистарског рада

Име и презиме аутора: Његош Раилић

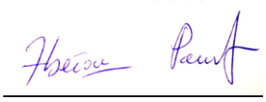
Наслов рада: Имплементација процеса за континуирано увођење у експлоатацију микросервиса

Ментор: Доц. Др Михајло Савић

Изјављујем да је штампана верзија мог мастер/магистарског рада идентична електронској верзији коју сам предао/ла за дигитални репозиторијум Универзитета у Бањој Луци.

У Бањој Луци 7.12.2021

Потпис кандидата



Др Зоран Ђурић, редовни професор
Електротехнички факултет Универзитета у Бањој Луци

Др Михајло Савић, доцент
Електротехнички факултет Универзитета у Бањој Луци

Др Дражен Брђанин, ванредни професор
Електротехнички факултет Универзитета у Бањој Луци

НАУЧНО-НАСТАВНОМ ВИЈЕЋУ
ЕЛЕКТРОТЕХНИЧКОГ ФАКУЛТЕТА
УНИВЕРЗИТЕТА У БАЊОЈ ЛУЦИ

Одлукама Научно-наставног вијећа Електротехничког факултета Универзитета у Бањој Луци број 20/3.947-1061/19 од 12.09.2019. године и број 20/3.5-10/21 од 13.01.2021. године, именовани смо за чланове Комисије за завршни рад II циклуса, под називом „**Имплементација процеса за континуирано увођење у експлоатацију микросервиса**“, кандидата **Његоша Раилића**. Након прегледа приложеног рада подносимо сљедећи

ИЗВЈЕШТАЈ

1. БИОГРАФСКИ ПОДАЦИ КАНДИДАТА

Његош Раилић, дипл. инж. електротехнике, рођен је 29.01.1986. године у Санском Мосту. У Прњавору је завршио основну школу 2000. године, а затим и средњу Електротехничку школу 2004. године. Основне студије, на студијском програму Рачунарство и информатика, уписао је 2004. године на Електротехничком факултету Универзитета у Бањој Луци, а дипломирао у новембру 2009. године (оцјеном 10) на тему “*Failover Active Directory Domain Controller* сервиса на *Windows* 2008 сервер оперативном систему“ са просјечном оцјеном 8.72. Студије II циклуса студија, на студијском програму Рачунарство и информатика на Електротехничком факултету Универзитета у Бањој Луци, уписао је 2009 године, на којем је положио све испите са просјечном оцјеном 9,2.

Од јула 2006. до новембра 2009. године био је запослен на Електротехничком факултету, Бања Лука, гдје је као студент био ангажован на пословима системске администрације и развоју апликација за подршку рада студентске службе. Од јануара 2010. до новембра 2010. године био је запослен у Републичком хидрометеоролошком заводу, Бања Лука, гдје је на позицији систем инжењера био ангажован на пословима системске и мрежне администрације. Затим је од децембра 2010. до августа 2016. године био запослен у предузећу *Asseco-SEE* БиХ, филијала Бања Лука, гдје је као систем инжењер био задужен за рјешења из области ИТ инфраструктуре, као и имплементацији различитих система за банкарско пословање. Од октобра 2016. године је запослен у предузећу *Wayfair GMBH*, у Берлину, гдје је на позицији водећег SRE (eng. *Site Reliability Engineer*) ангажован на развоју ИТ инфраструктуре сачињене од бројних сервиса, укључујући системе за управљање конфигурацијом, динамичко

скалирање и оркестрацију сервиса. На овој позицији примарно је фокусиран на пројектовање и развој миркосервисне архитектуре, као и аутоамтизацију процеса.

Кандидат је до сада објавио три рада на научним и стручним конференцијама националног и међународног значаја:

- **Његош Раилић**, Михајло Савић: „Architecting Continuous Integration and Continuous Deployment for Microservice Architecture“, ISBN: 978-1-7281-8230-8, INFOTEH, Јахорина 2021.
- Михајло Савић, Милорад Божић, **Његош Раилић**, „Систем за прикупљање, складиштење, приказивање и обраду података метеоролошких станица – METSTARS“, INFOTEH 2009, Јахорина, 2009.
- **Његош Раилић**, Горан Обрадовић, Алексеј Аврамовић: „Систем за 3Д симулацију модела сунчевог система“, ISBN 978-99938-806-3-9, Студенти у сусрет науци, Бања Лука 2009.

2. ОСНОВНИ ПОДАЦИ О РАДУ

Завршни рад II циклуса студија кандидата Његоша Раилића, под називом "**Имплементација процеса за континуирано увођење у експлоатацију микросервиса**", садржи 110 нумерисаних страница са укупно 85 слика и шест табела, а организован је у седам глава:

1. Увод,
2. Микросервисна архитектура,
3. Линија за увођење у експлоатацију,
4. Стратегије за тестирање и увођење у експлоатацију,
5. Приједлог рјешења за ЦД,
6. Опис имплементираног рјешења и резултати примјене,
7. Закључак.

Списак коришћене литературе садржи 142 цитирана извора.

3. АНАЛИЗА РАДА

У уводном дијелу рада прво су описани мотивација за израду рада и предмет истраживања, а затим су наведени циљеви, методологија и остварени резултати истраживања. Потом је укратко приказан садржај рада по главама и наведен научни рад у којем су објављени резултати истраживања. Основни мотив за истраживање и израду рада, представља непостојање рјешења које обједињује и имлементира све неопходне функционалности система за континуирано увођење микросервиса у експлоатацију.

Након уводне главе, друга глава обухвата теоријски увод у микросервисну архитектуру, њено поређење са другим архитектурама информационих система, као и кратак преглед савремених технологија неопходних да би се разумјела микросервисна архитектура. Затим је дат детаљан опис архитектуре *Kubernetes* платформе за оркестрацију контејнера, са освртом на дистрибуиране системе и појам евентуалне конзистентности. Посебан осврт дат је на умрежавање контејнера и сам начин рада *Kubernetes* платформе.

У трећој глави дат је детаљан преглед компонената линија за увођење у експлоатацију софтвера. Прво је дат уопштен преглед саставних компонената једног система за увођење у експлоатацију микросервиса, а затим је свака од њих детаљно

описана. Након тога, објашњена је важност употребе система за контролу верзија и дат кратак преглед начина организације репозиторијума изворног кода. Осим тога, описани су системи за контролу верзија, након чега слиједе стратегије гранања. Иако је тестирање софтвера засебна дисциплина, узимајући у обзир њену важност, дат је кратак преглед тестирања и објашњена је улога тестирања приликом аутоматизације испоруке микросервисних апликација.

Четврта глава даје преглед стратегија за тестирање и увођење у експлоатацију са посебним освртом на разлике између њих. Потом су описане стратегије засноване на поновном креирању, постепеној замјени и плаво-зеленој испоруци, као и њихове предности и недостаци. Иако стратегије за тестирање приликом увођења у експлоатацију имају сличну намјену, оне се користе првенствено за тестирање нових функционалности софтверских производа, па је дат и кратак преглед А/Б тестирања и стратегије канаринца, као и стратегије сјенке.

У петој глави дат је приједлог рјешења за имплементацију система за континуирано увођење микросервиса у продукцију. Овдје је приказана архитектура рјешења са описом основних дијелова. Како се систем састоји од двије логичке цјелине, прије свега ЦИ линије за креирање артефаката уз очување квалитета и исправности кода, а затим и ЦД линије чија је основна улога поуздана испорука софтвера, дат је детаљан опис предложених корака. Потом је дат детаљан опис развијених компонената.

Шеста глава описује имплементацију предложеног рјешења, прије свега инфраструктуре која је кориштена приликом имплементације и валидацију практичног дијела рада. Овдје је дат кратак преглед кориштених алата и начин на који је конфигурисана платформа за оркестрацију микросервисних апликација. Потом је дат опис имплементације библиотеке, као и клијентске апликације за оркестрацију увођења у експлоатацију микросервисних апликација. На крају је извршена валидација имплементираног рјешења на конкретном примјеру.

У седмој глави дата су закључна разматрања као и препоруке и смјернице за будућа истраживања.

4. НАЈВАЖНИЈИ ДОПРИНОСИ

Комисија сматра да је кандидат, кроз спроведено истраживање, реализовао завршни рад II циклуса студија који садржи више значајних доприноса, од којих су најважнији сљедећи:

1. У овом раду предложен је један приступ за рјешавање проблема континуалне испоруке за микросервисне апликације у корпоративном окружењу. У складу са предложеним приступом, употребом технологија отвореног кода и индустријских стандарда, у потпуности је имплементиран модуларан систем који олакшава пројектовање, имплементацију и одржавање микросервисних апликација.
2. Приступ је експериметално верификован на једном примјеру. Резулати показују да имплементирано рјешење омогућава веома лако увођење у експлоатацију микросервисних апликација. Аутоматизујући кораке креирања артефаката, тестирања и испоруке, обезбјеђује се висок квалитет кода без уношења додатних грешака, при чему је креирана верзија артефакта увијек спремна за испоруку у продукционо окружење. Уз употребу прогресивне испоруке и различитих стратегија, омогућава се високофреквентна, ефикасна и поуздана испорука микросервисних апликација на дневној бази.

5. ЗАКЉУЧАК И ПРИЈЕДЛОГ

Комисија сматра да завршни рад II циклуса под називом "**Имплементација процеса за континуирано увођење у експлоатацију микросервиса**", кандидата Његоша Раилића, садржи све потребне елементе и резултате којима су остварени постављени циљеви истраживања, па са задовољством предлаже Научно-наставном вијећу Електротехничког факултета Универзитета у Бањој Луци да усвоји извјештај Комисије и одобри заказивање усмене јавне одбране.

Бања Лука, 04.11.2021. године.

1. Проф. др Зоран Ђурић, предсједник
2. Доц. др Михајло Савић, ментор
3. Проф. др Дражен Брђанин, члан