SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Kontinuirana integracija i kontinuirana isporuka aplikacija visoke raspoloživosti i skalabilnosti postavljenih unutar Kubernetesa u oblaku

Mislav Jelušić

Zagreb, lipanj 2023.

Zahvaljujem svojoj obitelji i prijateljima na neizmjernoj podršci

Mentor: prof. dr. sc. Željko Ilić

Voditelj rada: prof. dr. sc. Željko Ilić

Sadržaj

1. Uvod

Posljednjih godina, brza evolucija tehnologija računalstva u oblaku revolucionirala je način na koji se aplikacije razvijaju, postavljaju i njima upravlja. Kako organizacije sve više prihvaćaju kontejnerizaciju i iskorištavaju snagu Kubernetesa za implementaciju svojih aplikacija, potreba za učinkovitim upravljanjem infrastrukturom i praksom kontinuirane isporuke postaje najvažnija. Platforme računalstva u oblaku, kao što je Amazon Web Services (AWS), pružaju robusnu osnovu za ulsuge upravljanog Kubernetes grozda i omogućavanje skalabilnih i otpornih implementacija aplikacija. Za učinkovito korištenje mogućnosti Kubernetesa i AWS-a, ključno je usvojiti suvremene prakse i metodologije koje usmjeravaju procese razvoja i implementacije. U ovom kontekstu, važno je uzeti u obzir koncepte kao što su infrastruktura kao kod, životni ciklus razvoja programa, agilna metodologija te kontinuirana integracija i kontinuirana isporuka. Organizacije prihvaćajući agilne metodologije, nastoje poboljšati suradnju, fleksibilnost i brzinu u svojim razvojnim procesima. Agilne prakse promoviraju iterativni i inkrementalni razvoj, omogućujući brže izlazak na tržište i bolju reakciju na potrebe kupaca. U kombinaciji s kontinuiranom integracijom i kontinuiranom isporukom, koje automatiziraju integraciju, testiranje i implementaciju softvera, organizacije mogu postići brzu i pouzdanu isporuku aplikacija. Uz to GitOps, operativni okvir koji je u skladu s načelima Gita, uvodi deklarativni pristup upravljanju cijelim životnim ciklusom aplikacije s kontrolom verzija. Argo CD, GitOps alat, omogućuje kontinuiranu isporuku aplikacija u Kubernetes grozdove automatskim sinkroniziranjem željenog stanja aplikacije sa stvarnim stanjem u grozdu. Korištenjem Argo CD-a, organizacije mogu postići veću stabilnost, reviziju i skalabilnost u svojim implementacijama. Ovaj diplomski rad opisat će cijeli proces i alate modernog razvoja aplikacija. U prvom poglavlju objašnjeno je korištenje Kubernetesa u oblaku i implementacija korištenjem alata Terraform. Drugo poglavlje se bavi životnim ciklusom programa, od kojih je objašnjena kontinuirana integracija i kontinuirana isporuka uz GitOps metodologiju, uz pripadajuće alate GitHub Actions i Argo CD. Zadnje poglavlje objašnjava implementaciju cijelog proces kontinuirane integracije i kontinuirane isporuke aplikacija unutar Kubernetesa na jednostavnom Python primjeru.

2. Kubernetes u oblaku

Kubernetes je moćan alat za orkestraciju kontejnera koji pojednostavljuje upravljanje kontejnerskim aplikacijama u oblaku. S porastom računalstva u oblaku, mnoge se organizacije okreću Kubernetesu za upravljanje svojim aplikacijama u oblaku. Kubernetes omogućuje programerima da implementiraju i skaliraju svoje aplikacije na visoko dostupan način otporan na pogreške. Iskorištavanjem elastičnosti oblaka i Kubernetesovih mogućnosti automatizacije, organizacije mogu optimizirati svoje resurse i smanjiti troškove. U tom kontekstu, upravljane Kubernetes usluge postaju sve popularnije jer pružaju prednosti Kubernetesa bez dodatnih troškova upravljanja temeljnom infrastrukturom. Ovo poglavlje će istražiti prednosti i izazove korištenja Kubernetesa u oblaku, s fokusom na upravljane Kubernetes usluge.

2.1. Mikroservisi

Popularizaciju kontejnera i Kubernetesa imala je mikroservisna arhitektura. Mikroservisna arhitektura je način razvoja programa kod kojeg se aplikacija sastoji od više malih, neovisnih usluga koje međusobno komuniciraju putem API-ja (*engl. Application Programming Interface*) ili komunikacija porukama (*engl. Message-queuing systems*). Nastala je na temeljima monolitne arhitekture u kojoj su usluge ili komponente bile čvrsto povezane. [2]

Prednosti ovakvog razvoja programa su poboljšana skalabilnost, fleksibilnost i tolerancija na pogreške, te mogućnost neovisne implementacije i ažuriranja usluga. Skalabilnost je moguća zbog same neovisnosti pojedine usluge, pa je tako moguće horizontalno ili vertikalno skalirati uslugu ovisno i njenom opterećenju unutar aplikacije. Aplikacijske usluge imaju različit prioritet i stadij razvoja. Odvojenost usluga unutar aplikacije omogućuje programerima neovisan razvoj svake pojedine usluge što dovodi do bržeg razvoja i izlaska na tržište čitave aplikacije. U slučajevima kvara jedne od usluga, cijela aplikacija može nastaviti raditi, a programeri se mogu fokusirati na popravak te usluge bez da je čitav sustav neaktivan. Neovisnost se također ogleda u pogledu korištenja tehnologija. Moguće je koristiti različite tehnologije unutar svake pojedine usluge zasebno. [2]

Ipak, postoje mane i izazovi ovakvih arhitektura. Jedna je složenost, iako je sve odvojeno veliki sustavi imaju jako puno usluga kojima treba na kraju upravljati i povezati što povećava složenost sustava. Također svaka mikrousluga zahtijeva vlastite resurse za rad što može dovesti do povećane potražnje za resursima u odnosu na monolitnu aplikaciju. To povećava troškove, posebno u slučaju povećanog opterećenja. Osim toga, može doći do dodatnih troškova zbog

komunikacije između usluga i potrebe za učinkovitim nadzorom i testiranjem aplikacije u cjelini. Razvoj i implementacija aplikacije temeljene na mikrouslugama može biti složena i zahtijeva specijalizirano znanje i infrastrukturu. Planiranje i dizajniranje sustava je kompliciranije zbog odluka o strategijama implementacije, upravljanju podacima, korištenju komunikacijskih protokola i drugih. [3]

Kako bi se olakšao početni razvoj te iskoristila fleksibilnost i skalabilnosti, mikroservisna arhitektura se postavlja na oblak, koji je opisan u kasnijim poglavljima. Osim toga, alati kao što su Kubernetes, Istio i Docker postali su popularni za upravljanje i implementaciju mikroservisnih aplikacija. [3]

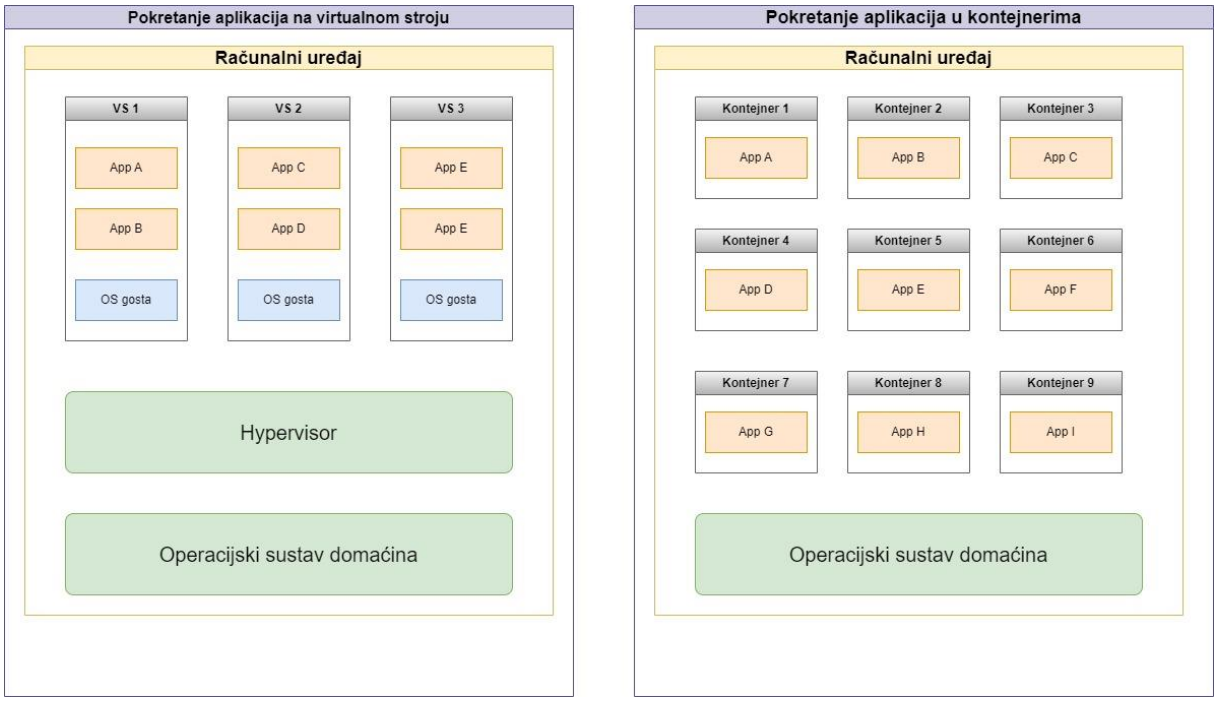
2.2. Kontejneri

Kontejneri su popularna tehnologija za implementaciju aplikacija temeljenih na mikroservisnoj arhitekturi. Pružajući lagan i prenosiv način za pakiranje i distribuciju aplikacija i njihovih ovisnosti. Mogućnost brzog pokretanja i uništavanja kontejnera, programerima olakšava testiranje i promjenu unutar koda na pojedinačne usluge bez utjecaja na cijelu aplikaciju. [5]

Kontejneri su oblik virtualizacije operacijskog sustava koji omogućuje aplikacijama da se izvode izolirano od osnovnog operativnog sustava. Oni pružaju lagano i prijenosno okruženje za rad koje se može lako premještati između različitih sustava. Kontejneri se sastoje od aplikacije i svih njezinih ovisnosti, pakiranih zajedno u jednu izvršnu sliku (*engl. image*). Ova se slika može jednostavno distribuirati i pokrenuti na bilo kojem sustavu koji podržava kontejnerizaciju putem registara (*engl. registries*). Registri su repozitoriji koji pohranjuju slike, koje se onda kasnije mogu preuzeti na bilo koje računalo povezano na Internet. [4]

Na slici (Slika 1) je prikazano pokretanje aplikacija unutar virtualnog stroja i unutar kontejnera. Kontejneri su izbacili potrebu za hipervizorom (*engl. Hypervisor*). Hipervizor virtualizira fizičke računalne resurse glavnog računala, kao što su CPU, memorija i pohrana, i dodjeljuje ih virtualnim strojevima koji mogu neovisno pokretati vlastite operativne sustave i aplikacije. [6] Dodatno, slika prikazuje učestalu praksu gdje se samo jedna aplikacija postavlja unutar kontejnera dok se kod virtualnih strojeva više aplikacija nalazi unutar jednog virtualnog stroja zbog nedostatka resursa koji se trebaju zauzeti za pojedini virtualni stroj. Što znači da su kontejneri mnogo „lakši“ od virtualnih strojeva. Odnosno, pomoću kontejnera je moguće pokretati puno više aplikacija na istom hardveru. Važno je napomenuti da svi kontejneri na

računalu koriste jednu jezgru domaćina, dok virtualni strojevi koriste vlastitu jezgru. To čini virtualni stroj više izoliranim u odnosu na kontejnere. [4]



Slika 1. Izolacija aplikacija na virtualnom stroju (lijevo), na kontejnerima (desno) [4]

Nekoliko je prednosti korištenja kontejnera u razvoju softvera. Kontejneri pružaju konzistenciju i mogućnost ponovnog stvaranja okruženja za pokretanje aplikacija, bez obzira na temeljni sustav. Lako prenošenje i pokretanje olakšava implementaciju i skaliranje aplikacija. Kontejneri također omogućuju lakše upravljanje aplikacijama, omogućujući programerima da lakše testiraju, ispravljaju pogreške i implementiraju svoje aplikacije. [5]

Docker je najpopularniji sustav za izvođenje kontejnera i upravljanje slikama. Omogućuje jednostavan i intuitivan način za izradu, pakiranje i distribuciju slika kontejnera. Docker slike mogu se jednostavno prenijeti u registar kontejnera, što olakšava dijeljenje i distribuciju slika u različitim sustavima. Docker je razvio svoj registar kontejnera koji se zove Docker Hub. Kako bi se izradile Docker slike potrebno je napisati Dockerfile odnosno tekstualnu datoteku koja sadrži niz instrukcija za izradu slike. Docker je moguće instalirati na svim popularnim operacijskim sustavima kao što su Linux, Microsoft Windows ili MacOS. [5]

2.3. Orkestracija kontejnera

Korištenjem mikroservisne arhitekture zajedno s kontejnerima, broj kontejnera raste i javlja potreba za lakšim upravljanjem i postavljanjem kontejnera odnosno aplikacijskih usluga unutar kontejnera. Rješenje se nalazi u orkestraciji kontejnera odnosno alatima za orkestraciju.

Orkestracija kontejnera je programsko rješenje koje pomaže u implementaciji, skaliranju i upravljanju kontejnerskom infrastrukturom. Omogućuje jednostavnu implementaciju aplikacija u više kontejnera rješavanjem izazova upravljanja pojedinačnim kontejnerima. Ova arhitektura omogućuje automatizaciju životnog ciklusa aplikacije pružajući jedinstveno sučelje za stvaranje i orkestriranje kontejnera. Pomoću ovih alata moguće je stvaranje, ažuriranje i uklanjanje aplikacije bez brige za infrastrukturu koja ih pokreće. Sposobnost jednostavnog skaliranje je još jedan od prednosti, a moguće je skalirati na dva načina. Horizontalno skaliranje dodaje više replika aplikacije. Dok se vertikalnim skaliranjem aplikaciji povećavaju resursi koje aplikacija može koristiti, kao što su procesorska snaga i memorija. Također, olakšano je upravljanje mrežnih dijelova infrastrukture kao i čuvanje od sigurnosnih rizika. [7]

2.3.1. Kubernetes

Kubernetes je najpopularniji alat za orkestraciju kontejnera i alat koji će se koristiti u ovom diplomskom radu. Početno je to bio projekt koji je razvijao Google, no kasnije je doniran organizaciji Cloud Native Computing Foundation (CNCF). CNCF razvija projekte otvorenog koda te je Kubernetes ujedno jedan od njih.

Kubernetes pruža platformu za implementaciju, upravljanje i skaliranje aplikacija u kontejneru. U svojoj srži, Kubernetes se sastoji od skupa glavnih čvorova i radnih čvorova. Glavni čvorovi odgovorni su za upravljanje radničkim čvorovima i raspoređivanje radnih opterećenja za njih. Radnički čvorovi odgovorni su za pokretanje kontejnerskih aplikacija. Kubernetes pruža skup API-ja i alata za implementaciju i upravljanje kontejnerskim aplikacijama, kao i skup kontrolera za upravljanje infrastrukturom potrebnom za pokretanje aplikacija. Kubernetes koristi deklarativne konfiguracijske datoteke, zvane manifesti, za definiranje željenog stanja infrastrukture. Ovi manifesti opisuju željeno stanje sustava, uključujući broj replika usluge, resurse potrebne za kapsulu i mrežna pravila. Kubernetes kontinuirano prati trenutno stanje sustava i radi na tome da trenutno stanje odgovara željenom stanju. [8]

Kubernetes je široko prihvaćen među pružateljima usluge oblaka, a većina pružatelja usluga sada nudi upravljane Kubernetes usluge. Upravljani Kubernetes omogućuje jednostavan i učinkovit način za upravljanje Kubernetes klasterima bez brige o temeljnoj infrastrukturi. Temeljna infrastruktura Kubernetesa se odnosi na komponente upravljačke ravnine koje se nalaze na glavnom čvoru i komponente radničkih čvorova. Što znači da se korisnik Kubernetesa ne mora brinuti oko instalacije Kubernetes komponenti na infrastrukturi oblaka, nego to pružatelj usluge radi automatski pri instalaciji upravljanog Kubernetesa od strane korisnika.

Ovime se zaobilazi dugi put učenja o arhitekturi Kubernetesa i instalacije komponenti Kubernetesa. Također je olakšana nadogradnja na novije verzije, pa tako većina pružatelja usluge ima jednostavni način podizanja verzije Kubernetesa u par koraka.

2.3.2. Glavni koncepti Kubernetesa

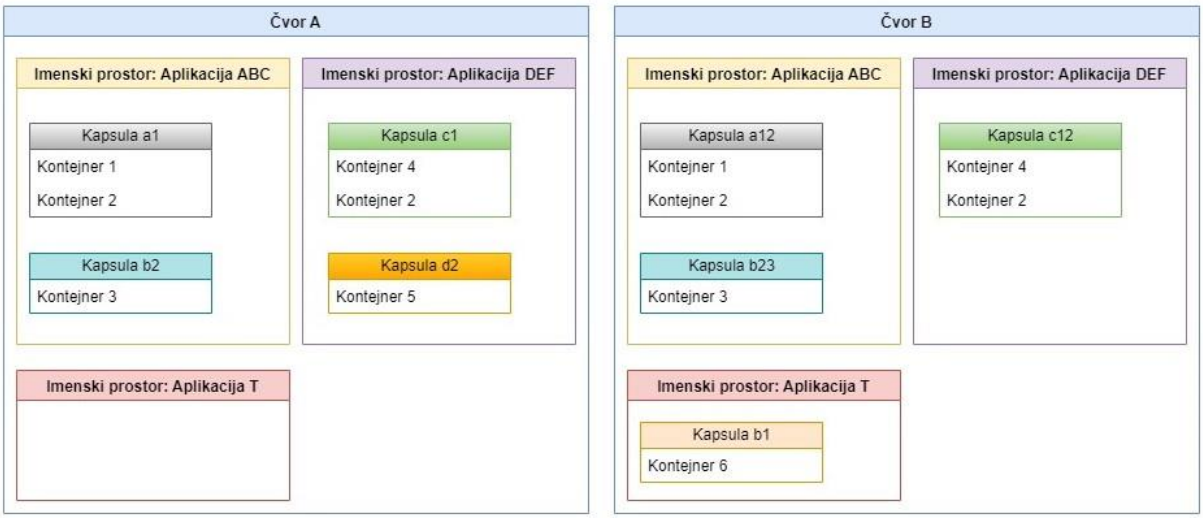
Razumijevanje Kubernetesa počinje od njegovih koncepata, iako ih ima mnogo, u ovom radu će biti opisani i definirani samo glavni za osnovno razumijevanje rada u Kubernetesu.

Radni čvorovi mogu biti fizički ili virtualni strojevi. Svaki od čvorova sadrži usluge potrebne za pokretanje kapsula. Kubernetes grozd je skup glavnih i radnih čvorova. Svaki grozd se sastoji od minimalno jednog glavnog i jednog radnog čvora. Komunikacija korisnika s grozdom se obavlja preko API-ija na glavnom čvoru.

Kako bi se odijelili resursi unutar Kubernetes grozda koriste se imenski prostori (*engl. Namespaces*). Imenski prostor je način stvaranja više virtualnih grozdova unutar jednog fizičkog grozda. Oni pružaju način za podjelu i izolaciju resursa u grozdu te za njihovo organiziranje prema različitim okruženjima, timovima, aplikacijama ili svrhama.

Kapsule (*engl. Pods*) su najmanje jedinice unutar Kubernetesa koje je moguće stvoriti. Jedan ili više kontejnera je moguće staviti unutar jedne kapsule, ali najčešće je to samo jedan kontejner. Kontejneri se u kapsule postavljaju skidanjem kontejnera s nekog od registara odnosno repozitorija.

Na slici (Slika 2) je prikaz odnosa čvorova, imenskih prostora, kapsula i kontejnera. Imenski prostor je postojan na svakom čvoru kada ga izradimo. Nakon toga je moguće unutar imenskog prostora postavljati kapsule koje se mogu postaviti na željeni čvor ili nasumično. Na prikazu je jasno vidljivo kako kapsule mogu imati jedan ili više kontejnera.

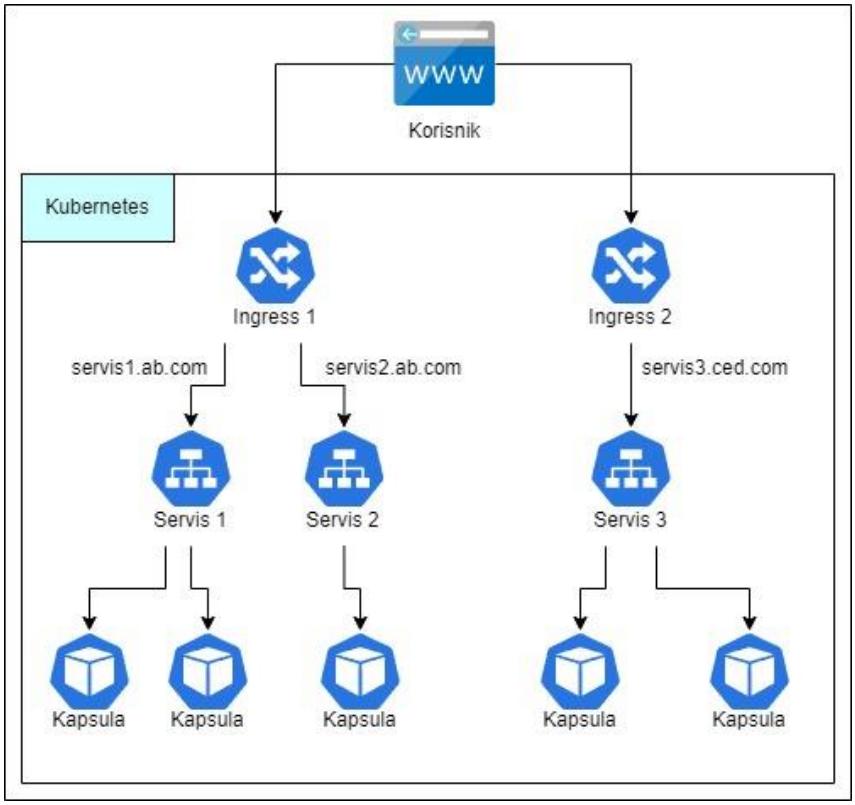


Slika 2. Prikaz odnosa čvorova, imenskih prostora, kapsula i kontejnera

Kapsule se mogu postavljati zasebno ili ih može automatizirano postavljati resurs ReplicaSet koji ima točan broj i konfiguraciju kapsule. Resursi Deployments stvara resurs ReplicaSet, a sami korisnik postavlja Deployments na Kubernetes. Resurs Deployments se koristi kada želimo postaviti aplikacije „bez stanja“. Također, resursi koji postavljaju kapsule na grozd su StatefulSet i DaemonSet. StatefulSet se koristi za aplikacije „sa stanjem“, većinom su to neke baze podataka i slične aplikacije koje zahtijevaju trajnu pohranu. DaemonSet osigurava da svaki čvor na grozdu pokreće kopiju neke kapsule odnosno aplikacije, obično se koristi za aplikacije koje zapisuju događaje (*engl. Logging*) ili vrše nadzor sustava (*engl. Monitoring*). Zadnji resurs od resursa koji su zaduženi za radna opterećenja su poslovi odnosno cron poslovi. Poslovi stvaraju kapsule koje se pokreću i završavaju. Koriste se u svrhu jedinstvenih periodičkih zadataka, migracije podataka ili izrade sigurnosnih kopija.

Bitni resursi za čuvanje podataka su Persistent Volume Claim (PVC) i Persistent Volume (PV). PV je dio umreženog resursa za pohranu koji osigurava administrator ili ga dinamički osigurava Kubernetes. PVC je zahtjev za određenom količinom resursa za pohranu koji je definirao programer. Nakon što se PVC stvori, Kubernetes preslikava PVC u PV, koji pruža stvarni resurs za pohranu. PV može unaprijed osigurati administrator ili ga Kubernetes može dinamički osigurati na temelju StorageClass resursa navedenog u PVC-u. PV se zatim spaja na kapsulu, koja onda može čitati i pisati na resurs za pohranu. Resurs Secret rješava pitanje sigurnosti osjetljivih podataka koji se koriste u aplikacijama. Povezivanjem varijabli okruženja s resursom Secret unutar resursa Deployments moguće je zaštiti osjetljive podatke. Secret čuva podatke u zaštićenom obliku te se podaci kodiraju Base64 standardom.

Za razumijevanje mrežnog dijela Kubernetesa osnovni su resursi Ingress i servis (*engl. Service*). Ingress je Kubernetes API objekt koji pruža način upravljanja vanjskim pristupom servisima u Kubernetes klasteru. Omogućuje usmjeravanje prometa na različite servise na temelju URL putanje ili naziva hosta, što olakšava izlaganje više servisa pod jednom IP adresom. Ingress također podržava TLS za sigurne veze i pruža mogućnosti balansiranja opterećenja. Servisi omogućuju komunikaciju aplikacija unutar Kubernetesa. Oni pružaju stabilnu IP adresu i DNS naziv skupu kapsula i mogu uravnotežiti promet među njima. Servisi se mogu klasificirati u četiri vrste: ClusterIP, NodePort, LoadBalancer i ExternalName. Sve četiri vrste imaju neku namjenu, a najkorišteniji su ClusterIP i NodePort. ClusterIP je vrsta servisa koja izlaže servis na unutarnjoj IP adresi klastera. NodePort izlaže servis na IP adresi svakog čvora na statičkom priključku. Na slici (Slika 3) se može vidjeti u kojoj su relaciji Ingress i servis. Ingress prima zahtjeve „izvan“ Kubernetes grozda te ga prosljeđuje zadanom servisu ovisno o konfiguraciji Ingressa. Servis prosljeđuje zahtjeve replikama kapsula. [8]



Slika 3. Ingress i Servis

2.3.3. **Alat Helm**

Helm je upravitelj paketa za Kubernetes koji pojednostavljuje implementaciju i upravljanje aplikacijama i uslugama na Kubernetes grozdovima. Helm chart su paketi koji sadrže sve potrebne resurse i konfiguracije za implementaciju aplikacije ili usluge na Kubernetes grozdu.

Helm koristi YAML konfiguracijske datoteke za definiranje resursa koji bi trebali biti postavljeni kao dio aplikacije unutar Kubernetesa. Konfiguracija tih datoteka je deklarativna. Ovo omogućuje korisnicima da specificiraju željeno stanje svojih resursa, a Helm će osigurati da su resursi kreirani i konfigurirani kako je navedeno.

Snaga Helma je u ponovnom korištenju. Helm chartovi se mogu koristiti za implementaciju višestrukih instanci aplikacije ili usluge i mogu se lako dijeliti s drugima. To olakšava implementaciju i upravljanje aplikacijama na dosljedan i ponovljiv način te pomaže u smanjenju vremena i truda potrebnog za implementaciju i upravljanje aplikacijama na Kubernetes grozdovima.

Helm chartovi koriste semantičko određivanje verzija za praćenje promjena resursa i konfiguracija koje definiraju. To olakšava praćenje i vraćanje promjena te nadogradnju na novije verzije aplikacije ili usluge. Svakim novim postavljanjem ili ažuriranjem aplikacije, povećava se broj koji se naziva revizija (engl. *Revision*). U svakom trenutku moguće je vraćanje na stare revizije. Uz reviziju, istovremeno se prate verzija charta i verzija aplikacije. [9]

2.3.4. Tok rada alata Helm

Alat Helm potrebno je instalirati lokalno na računalu s kojeg će se izrađivati datoteke. To je najlakše učiniti posjećivanjem službene stranice (https://helm.sh/) na kojoj se mogu pronaći instalacijske upute. Instalacija se odvija u terminalu, kao i korištenje Helma nakon toga.

Nakon instalacije, postoje dvije opcije stvaranje vlastitog charta ili pronalazak postojećeg charta na internetu te njegovo skidanje s repozitorija.

Za stvaranje vlastitog charta prvo je potrebno stvoriti novi direktorij u koji se pohranjuje Helm chart. Ovaj direktorij trebao bi sadržavati datoteku pod nazivom „Chart.yaml" koja definira metapodatke za spomenuti chart, datoteku „values.yaml“ koja će definirati vrijednosti za konfiguracijske datoteke resursa, kao i direktorij „templates" koji sadrži yaml konfiguracijske datoteke za resurse. Nakon toga je potrebno definirati resurse, u direktoriju „templates" potrebno je izraditi yaml konfiguracijske datoteke za resurse koji će se implementirati kao dio Helm charta. Konfiguracijske datoteke koriste sintaksu i definiraju željeno stanje Kubernetes resursa. U slučaju korištenja drugih Helm chartova u vlastitom chartu nužno ih je dodati kao ovisnosti u datoteci „Chart.yaml". Ovisnosti se specificiraju pomoću polja „dependencies" u metapodacima charta. Nakon što se definiraju svi željeni resursi i ovisnosti, moguće je zapakirati svoj Helm chart u tar arhivu pomoću naredbe "helm package". Ovo će stvoriti

datoteku s ekstenzijom „.tgz" koja sadrži Helm chart i sve njegove ovisnosti. Instaliranje vlastitog Helm charta na Kubernetes grozd obavlja se uz pomoć naredbe „helm install". Ovo će stvoriti resurse definirane u spomenutom chartu i konfigurirati ih kako je navedeno.

Instaliranje postojećeg charta od treće strane odvija se u nekoliko koraka. Prije instaliranja Helm charta iz repozitorija, nužno je dodati repozitorij u vlastiti Helm CLI. To je moguće učiniti pomoću naredbe „helm repo add", iza koje slijedi naziv i URL spremišta. Nakon dodavanja repozitorija, potrebno je ažurirati indeks repozitorija radi sigurnosti preuzimanja najnovijeg charta. Ažuriranje se radi uz pomoć naredbe „helm repo update". Nakon što se indeks repozitorija ažurira, moguće je tražiti chart pomoću naredbe „helm search". Ova naredba će vratiti popis chartova koji su dostupni u repozitoriju, zajedno s njihovim verzijama i opisima. Nakon pronalaska željenog charta, naredba „helm install" uz dodatak naziva charta i željene konfiguracije, instalirat će taj chart na Kubernetes grozd. Na primjer „helm install postgres-exporter prometheus-community/prometheus-postgres-exporter -f values.yaml“, preuzeti će chart „prometheus-postgres-exporter“ s repozitorija „prometheus-community“ te mu dodijeliti vrijednosti koje su definirane u datoteci „values.yaml“, naposljetku ga instalirati na Kubernetes grozd pod nazivom postgres-exporter. [9]

2.4. Računalstvo u oblaku

Računalstvo u oblaku (engl. *Cloud Computing*) je pojam koji se koristi za opisivanje isporuke računalnih usluga kao što su pohrana, obrada, umrežavanje, softver, analitika i ostalih usluga putem interneta. Ove se usluge pružaju na zahtjev, obično na osnovi tekućeg plaćanja.

Računalstvo u oblaku revolucioniralo je način na koji tvrtke i pojedinci pristupaju i koriste tehnologiju. Eliminirana je potreba za skupocjenim hardverskim i softverskim instalacijama, omogućavajući korisnicima pristup njihovim podacima i aplikacijama s bilo kojeg mjesta, u bilo koje vrijeme i na bilo kojem uređaju.

Postoji nekoliko vrsta modela računalstva u oblaku, podjela je moguća na javne oblake, privatne oblake i hibridne oblake. Javni oblaci su u vlasništvu tvrtki koje nude svoje usluge i dostupni su široj javnosti. Tvrtke održavaju fizičku infrastrukturu te klijentima nude računalne usluge putem interneta. Privatnim oblacima upravlja jedna organizacija te ih koristi isključivo ta organizacija. Javnosti nije dostupan pristup privatnim oblacima te nije moguće koristiti njihove usluge. Hibridni oblaci su kombinacija javnih i privatnih oblaka, omogućujući organizacijama da iskoriste prednosti obaju modela.

Podjela prema modelu usluga koji neki oblak nudi su infrastruktura kao usluga (engl. *Infrastructure as a Service - IaaS)*, platforma kao usluga (engl. *Platform as a Service – PaaS*) i softver kao usluga (engl. *Softver as a Service – SaaS*). Infrastruktura kao usluga označava skup usluga gdje pružatelj upravlja dijelom infrastrukture zadužene za pohranu podataka, servere i mrežni dio te korisniku isporučuje pristup virtualnim strojevima ili jednostavnom skladištenju podataka. Platforma kao usluga je model u kojem se pruža platforma za razvoj, pokretanje i upravljanje aplikacijama i uslugama. Uključuje niz alata i usluga koje programeri mogu koristiti za izradu, testiranje i implementaciju aplikacija, bez brige o temeljnoj infrastrukturi ili upravljanju infrastrukturom. Primjer takve usluge bi bila Amazonova implementacija Kubernetesa na njihovom oblaku - EKS (engl. *Elastic Kubernetes Service*). Softver kao usluga, model je računalstva u oblaku u kojem se softverske aplikacije pružaju kao usluga putem interneta, umjesto da se instaliraju i izvode na pojedinačnim računalima ili poslužiteljima. Uz SaaS, korisnici mogu pristupiti softveru i koristiti ga putem web-preglednika ili druge klijentske aplikacije, a davatelj usluge odgovoran je za hosting, održavanje i podršku. SaaS omogućuje korisnicima korištenje softverskih aplikacija na temelju pretplate, umjesto da ih moraju kupiti i instalirati na vlastite sustave.

Jedna od glavnih prednosti računalstva u oblaku je njegova skalabilnost. Uz mogućnost brzog i jednostavnog dodavanja ili uklanjanja resursa, organizacije mogu povećati ili smanjiti svoje računalne potrebe prema potrebi, bez potrebe za skupim ulaganjima u hardver. To im omogućuje da budu agilniji i osjetljiviji na promjenjive poslovne potrebe.

Još jedna prednost računalstva u oblaku je njegova isplativost. Plaćajući samo resurse koje koriste, organizacije mogu uštedjeti novac na hardveru, održavanju i troškovima energije. Osim toga, računalstvo u oblaku može smanjiti potrebu za IT osobljem, budući da mnoge zadatke povezane s upravljanjem lokalnom infrastrukturom obavlja pružatelj usluga u oblaku.

Sigurnost fizičke infrastrukture i ponekih softverskih stavki isto tako preuzima pružatelj. Zbog toga, postoje i neki potencijalni nedostaci računalstva u oblaku. Jedna od briga je sigurnost jer se osjetljivi podaci pohranjuju i obrađuju izvan mjesta organizacije. Kako bi se to riješilo, organizacije moraju odabrati uglednog i sigurnog pružatelja usluga oblaka i implementirati odgovarajuće sigurnosne mjere za zaštitu svojih podataka. [10][11]

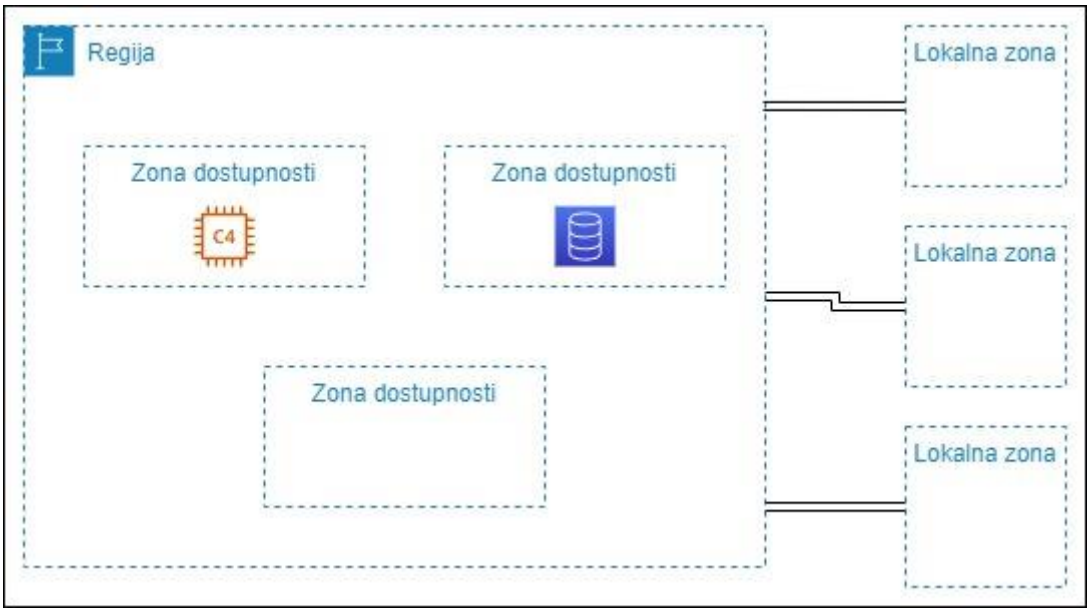
2.4.1. **Karakteristike platforme Amazon Web Services**

Amazon Web Services (AWS) je platforma za računalstvo u oblaku koja pruža širok raspon usluga i alata za tvrtke svih veličina. Ove su usluge osmišljene kako bi omogućile tvrtkama izgradnju, implementaciju i upravljanje aplikacijama odnosno radnim opterećenjima u oblaku, kao i pohranjivanje i upravljanje podacima.

AWS nudi preko 200 različitih proizvoda i usluga, koji su grupirani u nekoliko kategorija: računalstvo, pohrana i isporuka sadržaja, baza podataka, umrežavanje, sigurnost, analitika, strojno učenje, umjetna inteligencija, integracija aplikacija, razvojni alati i Internet stvari (engl. *Internet of Things* - IoT). Neke od najpopularnijih i najčešće korištenih usluga uključuju Amazon Elastic Compute Cloud (EC2) za računalstvo, Amazon Simple Storage Service (S3) za pohranu i Amazon Relational Database Service (RDS) za baze podataka.

Jedna od ključnih prednosti korištenja AWS-a je njegova skalabilnost. Tvrtke mogu jednostavno i brzo povećati ili smanjiti svoje resurse kako bi se zadovoljile promjenjive potrebe aplikacija, odnosno radno opterećenje sustava. To im omogućuje uštedu novca plaćajući samo resurse koje koriste, umjesto održavanja fiksne infrastrukture. AWS također nudi model određivanja cijena po principu "pay-as-you-go", što znači da tvrtke plaćaju samo resurse koje potroše.

AWS također ima globalnu infrastrukturu koja je dizajnirana za visoku dostupnost i pouzdanost. Ima više podatkovnih centara smještenih diljem svijeta, što pomaže osigurati da aplikacije i radna opterećenja ostanu dostupni čak i ako postoje problemi s određenim podatkovnim centrom. Infrastruktura odnosno podatkovni centri su raspoređeni po lokacijama. Ove lokacije sastoje se od AWS regija, zona dostupnosti i lokalnih zona. Svaka AWS regija je zasebno geografsko područje. Svaka AWS regija ima više izoliranih lokacija poznatih kao zone dostupnosti. Korištenjem lokalnih zona, moguće je smjestiti resurse na više lokacija bliže svojim korisnicima. Amazon RDS omogućuje da se resursi, kao što su instance baze podataka, i sami podatci smjeste na više lokacija. Resursi se ne repliciraju automatski u AWS regijama, nego je potrebno to samostalno učiniti. Svaka regija označena je svojim identifikacijskim imenom, primjerice podatkovni centar koji se nalazi u Frankfurtu, za svoj identifikacijski kod koristi „eu-central-1“. Identifikacijski kodovi su bitni prilikom podizanja nekih resursa. Na slici (Slika 4) se vidi u kojem su odnosu regija, zone dostupnosti i lokalne zone. Zona dostupnosti se nalazi unutar regija i svaka je neovisna jedna od druge te se resursi mogu stvarati unutar njih.



Slika 4. Lokacijska raspodjela

AWS također nudi razne alate i usluge za nadzor i upravljanje aplikacijama, kao i za rješavanje problema i otklanjanje grešaka.

Uz svoje tehničke mogućnosti, AWS također nudi niz opcija podrške kako bi pomogao tvrtkama da izvuku najviše iz njegovih usluga. To uključuje dokumentaciju, forume i resurse zajednice, kao i sveobuhvatnije planove podrške koji omogućuju pristup tehničkim stručnjacima i drugim resursima.

Međutim, postoje i neki izazovi u korištenju AWS-a. Jedan od izazova je trošak korištenja AWS-a. Iako model određivanja cijena po principu "pay-as-you-go" može biti isplativ za tvrtke koje koriste samo malu količinu resursa, može postati skuplji za tvrtke koje koriste veliku količinu resursa. To može biti posebno izazovno za tvrtke koje imaju fluktuirajuće potrebe za resursima, jer će možda trebati platiti za resurse koje ne koriste.

Drugi izazov je složenost korištenja AWS-a. AWS nudi širok raspon proizvoda i usluga, što može biti neodoljivo za tvrtke koje tek počinju koristiti oblak. Zbog toga poduzećima može biti teško odrediti koji su im proizvodi i usluge potrebni i kako ih učinkovito koristiti. Problem se može javiti i kod tvrtki koje nemaju obučene ljude za korištenje AWS-a što može dodatno koštati organizaciju u vidu vremena i novca. [11][12]

2.4.2. **AWS virtualni privatni oblak**

Virtualni privatni oblak (engl. *Virtual Private Cloud* – VPC) je logički izolirani dio AWS oblaka koji korisnicima omogućuje pokretanje resursa u virtualnoj mreži koju sami definiraju.

VPC omogućuje korisnicima stvaranje virtualne mreže koja je izolirana od ostatka AWS oblaka i kontrolu mrežnih konfiguracija svojih resursa.

VPC je podijeljen u jednu ili više podmreža, koje su grupe instanci Amazon Elastic Compute Cloud (EC2) koje se nalaze u određenoj zoni dostupnosti (AZ). Podmreže mogu biti javne ili privatne, ovisno o tome imaju li ili nemaju direktni put do interneta.

Svaki VPC ima sigurnosni dio koji štiti cijelu ili dijelove te mreže. Stoga postoje usluge kao što su sigurnosna grupa (engl. *Security groups*) i liste kontrole mrežnog pristupa (engl. *Network Access Control Lists* – NACLs). Sigurnosna grupa je virtualni vatrozid koji kontrolira dolazni i odlazni promet za grupu instanci Amazon EC2. Sigurnosne grupe mogu se koristiti za dopuštanje ili odbijanje prometa na temelju protokola, pristupa (engl. *Port*) i izvornih/odredišnih IP adresa. NACL je virtualni vatrozid koji kontrolira dolazni i odlazni promet za VPC podmrežu. NACL se mogu koristiti za dopuštanje ili odbijanje prometa na temelju protokola, priključaka i izvornih/odredišnih IP adresa.

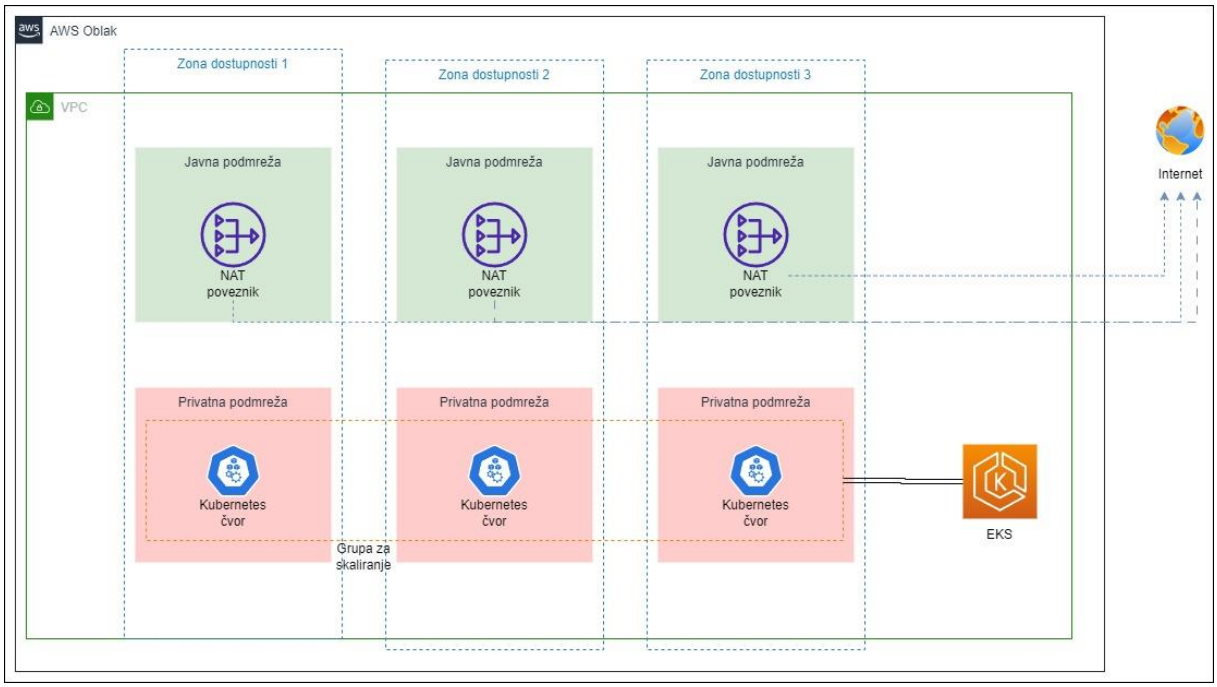
Ostale komponente koje se mogu koristiti unutar VPC-a su internetski poveznik (engl. *Internet Gateway – IGW*) i tablice usmjeravanja (engl. *Route Tables*). Internetski poveznik je VPC komponenta koja omogućuje resursima u VPC-u da se povežu na internet. Djeluje kao most između VPC-a i interneta i omogućuje protok odlaznog prometa s VPC-a na Internet. Tablica usmjeravanja je VPC komponenta koja kontrolira protok prometa između podmreža u VPC-u, kao i između VPC-a i interneta. Tablica usmjeravanja sastoji se od skupa pravila, koja određuju kamo promet treba biti usmjeren na temelju odredišta prometa. [13]

2.4.3. **Usluga AWS Elastic Kubernetes Service**

AWS EKS (engl. *Elastic Kubernetes Service* – EKS) potpuno je upravljana Kubernetes usluga koja olakšava implementaciju, pokretanje i upravljanje kontejnerskim aplikacijama na AWS oblaku. Uz AWS EKS, tvrtke mogu brzo i jednostavno kreirati, konfigurirati i upravljati Kubernetes grozdovima, bez brige o temeljnoj infrastrukturi ili upravljanju infrastrukturom. Korisnik se brine o radnim čvorovima te resursima koji se nalaze na njima kao što su kapsule, servisi, imenski prostori i ostalo. Dok su glavni čvorovi skriveni i održavani od strane pružatelja usluge odnosno AWS-a.

AWS EKS integrira se s drugim AWS uslugama kao što su Amazon Elastic Container Service (ECS) i Amazon Elastic Container Registry (ECR), što olakšava izgradnju, implementaciju i upravljanje kontejnerskim aplikacijama na AWS platformi. AWS EKS je visoko skalabilan i

visoko dostupan, s više zona dostupnosti i više glavnih čvorova kako bi se osigurala dostupnost aplikacija i radnih opterećenja. AWS EKS kompatibilan je s Kubernetes zajednicom otvorenog koda i podržava sve najnovije značajke i ažuriranja Kubernetesa. Na slici (Slika 5) je prikazan primjer AWS EKS koji je smješten unutar VPC-a. Kubernetes čvorovi su smješteni unutar privatnih podmreža (engl. *Subnet*), a svaka privatna podmreža je smještena u zasebnu zonu dostupnosti. To omogućuje zalihost i naposljetku visoku raspoloživost sustava. Unutar javnih podmreža smještena je još jedna grupa čvorova koja je direktno izložena internetu. Pomoću nje je moguće izlagati neke pristupe internetu ili može poslužiti kao bastion prema privatnim podmrežama. [14]



Slika 5. Primjer mrežne infrastrukture EKS-a

2.4.4. **AWS komandna linija**

Komunikacija s oblakom AWS moguća je putem naredbenog redka (engl. *Command Line Interface* – CLI), što može automatizirati i olakšati procese, a ponekad je nužna kod komunikacije s pojedinim resursima. Postoji nekoliko koraka prije nego je moguće ispravno koristiti AWS CLI. Na AWS stranicama potrebno je pronaći instalacijske upute za AWS CLI te ih slijediti. Provjera instalacije je moguća pomoću naredbe „aws –version“.

Nakon instalacije nužno je konfigurirati svoje AWS vjerodajnice. To se može učiniti pomoću naredbe "aws configure", koja će zatražiti ID ključ za pristup AWS-u (engl. *AWS access key*

ID) i tajni ključ za pristup (engl. *secret access key*). Ključevi se generiraju prilikom stvaranja AWS korisnika. [15]

Sada je moguća komunikacija s AWS-om putem komandne linije.

2.5. Infrastruktura kao kod

Infrastruktura kao kod (engl. *Infrastructure as Code* - IaC) je metodologija za upravljanje i postavljanje infrastrukture i povezanih resursa na oblak koristeći standardizirani i automatizirani način. Uključuje korištenje konfiguracijskih datoteka i skripti za definiranje i postavljanje infrastrukture, umjesto ručnog konfiguriranja resursa putem korisničkog sučelja (engl. *User Interface* – UI) ili sučelja naredbenog retka (engl. *Command Line Interface* – CLI). Infrastruktura kao kod moćan je i učinkovit pristup upravljanju i postavljanju infrastrukture i povezanih resursa.

Jedna od ključnih prednosti korištenja IAC-a jest to što organizacijama omogućuje upravljanje svojom infrastrukturom na učinkovitiji i konzistentniji način. Korištenjem konfiguracijskih datoteka i skripti, organizacije mogu osigurati da je njihova infrastruktura dosljedno konfigurirana i postavljena u više okruženja. To pomaže smanjiti rizik od pogrešaka i povećava pouzdanost njihovih sustava.

IaC također olakšava praćenje i promjene verzija infrastrukture. Korištenjem konfiguracijskih datoteka, organizacije mogu koristiti sustave kontrole verzija (git) za praćenje i vraćanje promjena, što pomaže smanjiti rizik od neželjenih posljedica i olakšava rješavanje problema.

Uz svoje prednosti za upravljanje infrastrukturom, IaC se također može koristiti za automatizaciju postavljanja aplikacija i usluga. Korištenjem konfiguracijskih datoteka i skripti za definiranje i postavljanje aplikacijskih okruženja, organizacije mogu pojednostaviti svoj proces postavljanja i smanjiti vrijeme i trud potreban za postavljanje novih aplikacija i usluga.

Jedan od glavnih alata koji se koristi za implementaciju IaC-a je alat za upravljanje konfiguracijom kao što su Puppet, Chef ili Ansible. Ovi alati omogućuju organizacijama da definiraju svoju infrastrukturu pomoću konfiguracijskih datoteka i skripti, a zatim koriste te datoteke za postavljanje i upravljanje svojom infrastrukturom na dosljedan i automatiziran način. Korištenjem konfiguracijskih datoteka i skripti za definiranje i implementaciju infrastrukture, organizacije mogu povećati učinkovitost, pouzdanost i agilnost, dok također

smanjuju rizik od pogrešaka te vrijeme i trud koji su potrebni za upravljanje i implementaciju sustava.

Postoje dva načina kojim možemo pisati kod, deklarativni i imperativni način. Deklarativni pristup definira željeno stanje sustava, uključujući potrebne resurse i sva svojstva koja bi trebala imati, a IaC alat će to konfigurirati kako je deklarirano kodom. Deklarativni pristup također čuva popis trenutnog stanja vaših sistemskih objekata, što uklanjanje infrastrukture čini jednostavnijim za izvršiti. Imperativni pristup definira specifične naredbe potrebne za postizanje željene konfiguracije, a te se naredbe zatim moraju izvršiti ispravnim redoslijedom. Mnogi IaC alati koriste deklarativni pristup i automatski će osigurati željenu infrastrukturu. Ako napravite promjene u željenom stanju, deklarativni IaC alat će primijeniti te promjene umjesto vas. Imperativnim pristupom potrebno je unaprijed smisliti kako treba primijeniti promjene. IaC alati često mogu raditi oba pristupa, ali imaju tendenciju da daju prednost jednom pristupu nad drugim. [16]

2.5.1. **Alat Terraform**

Terraform je alat otvorenog koda za infrastrukturu kao kod (IaC) koji korisnicima omogućuje definiranje i pružanje infrastrukture i povezanih resursa na već poznati i automatizirani način. Razvio ga je i održava HashiCorp i naširoko ga koriste organizacije svih veličina za upravljanje svojom infrastrukturom i povezanim resursima u oblaku ili lokalno.

Jedna od ključnih značajki Terraforma je njegova sposobnost podrške za više pružatelja usluga oblaka i lokalnih okruženja. Podržava preko 100 različitih pružatelja usluga, uključujući glavne pružatelje usluga oblaka kao što su Amazon Web Services (AWS), Microsoft Azure i Google Cloud Platform, kao i lokalna okruženja kao što je VMware vSphere. To korisnicima omogućuje upravljanje i dodjelu resursa u širokom rasponu okruženja pomoću jednog alata.

Terraform koristi konfiguracijske datoteke napisane u HashiCorp Configuration Language (HCL) za definiranje infrastrukture i povezanih resursa. Ove konfiguracijske datoteke nazivaju se "Terraform moduli" i mogu se koristiti za definiranje resursa kao što su računalne instance, spremnici za pohranu i mrežne konfiguracije.

Jedna od prednosti korištenja Terraforma je njegova sposobnost upravljanja resursima na deklarativan način. To znači da korisnici definiraju željeno stanje svojih resursa u konfiguracijskim datotekama, a Terraform se brine o osiguravanju i konfiguriranju resursa kako

bi odgovarali tom stanju. To pomaže osigurati da su resursi dosljedno i predvidljivo konfigurirani te olakšava praćenje i promjene verzija infrastrukture.

Terraform se također integrira s brojnim drugim alatima i uslugama, uključujući sustave kontrole verzija, cjevovode kontinuirane integracije/kontinuirane isporuke (CI/CD) i alate za praćenje i upozoravanje. Ovo omogućuje korisnicima da iskoriste Terraform u svoje postojeće procese te pomaže u pojednostavljenju upravljanja i implementacije infrastrukture i povezanih resursa. [17]

2.5.2. **Tok rada Terraforma**

Kako bi se koristio Terraform prvo ga je potrebno instalirati lokalno na računalu. To se može učiniti tako da se slijede upute za instalaciju na službenim stranicama Hashicorp Terraforma (https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli).

Nakon što se Terraform instalira i provjeri valjanost putem komandnog retka, potrebno je izraditi konfiguracijske datoteke koje će definirati željene resurse. Takve datoteke pišu se u jeziku HCL posebno dizajniranom za Terraform. Svaka takva datoteka koja postoji u nekom direktoriju čini skup koji se naziva modul. Terraform modul omogućuje stvaranje logičke apstrakcije na nekom skupu resursa. Moduli omogućuju grupiranje resursa i ponovno korištenje grupe resursa zajedno u nekom budućem projektu više puta. Postoje korijenski moduli koji je obično i glavni modul iz kojega povlačimo ostale module odnosno module djecu, čija lokacija može biti lokalna, iz službenih Terraform registara, Git repozitorija ili direktno iz HTTP URL-ova.

Nakon izrade modula moguće je dalje koristiti Terraform CLI za inicijalizaciju projekta. Uz pomoć komande „terraform init“ postiže se inicijalizacija konfiguracijskih datoteka te se preuzimaju sve potrebne datoteke i ovisnosti.

Nakon što se konfiguracija inicijalizira, moguće je dalje koristiti Terraform CLI za stvaranje i upravljanje resursima. Za stvaranje resursa koristi se naredba "terraform apply" koja će stvoriti resurse definirane u Terraform modulu. Kako bi se upravljalo i imalo više kontrole nad resursima. Koriste se naredbe kao što su „terraform validate“ i „terraform plan". Naredba „terraform validate“ služi kako bismo provjerili konfiguracijske datoteke, provjerava se je li sintaksa valjana i interno dosljedna. Dok se naredba „terraform plan“ koristi za pregled promjena koje bi se eventualno izvršile naredbom „terraform apply“. Ako je u nekom trenutku

potrebno obrisati sve resurse stvorene preko modula dovoljno je pokrenuti naredbu „terraform destroy" za brisanje resursa.

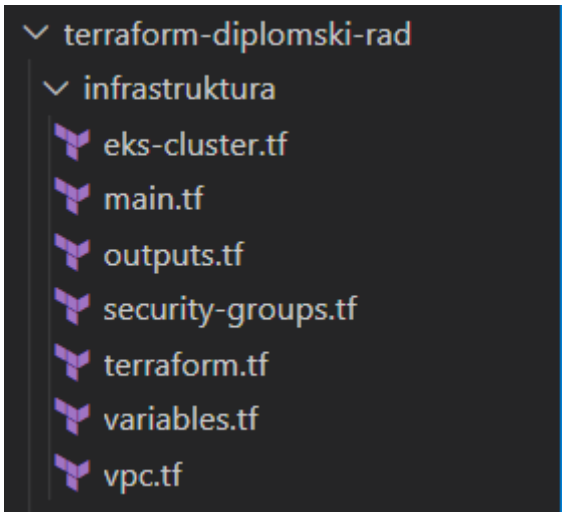
Terraform sva stanja „pamti“ i zapisuje u datoteku stanja koju kreira. Datoteka je u json formatu te ju je moguće spremati lokalno ili udaljeno. Udaljeno spremanje omogućava da više korisnika istovremeno mogu nadograđivati i koristiti Terraform skripte nad istim resursima. To je moguće zato što Terraform u slučaju pokretanja komandi od strane jednog korisnika, zaključava datoteku stanja i drugi korisnici ne mogu u tom trenutku izvršavati naredbe nad tim resursima. Datoteka stanja sadrži informacije o resursima koji su kreirani, uključujući njihov tip, ID i svojstva. Prilikom pokretanja Terraform komandi, Terraform uspoređuje stanje resursa u datoteci stanja sa željenim stanjem definiranim u konfiguraciji. Ako postoje bilo kakve razlike, Terraform će stvoriti, ažurirati ili izbrisati resurse prema potrebi kako bi odgovarali željenom stanju. Moguće je koristiti Terraform CLI za izravni pregled i manipuliranje datotekom stanja. Na primjer, postoji mogućnost koristiti naredbu "terraform state show" za pregled trenutnog stanja resursa ili naredbu "terraform state rm" za brisanje resursa iz datoteke stanja. [18]

2.6. Postavljanje infrastrukture

U ovom potpoglavlju bit će prikazano stvaranje svih AWS resursa potrebnih za normalan rad unutar Kubernetesa uz pomoć alata Terraform. Nakon toga uz pomoć alata Helm na Kubernetes grozd postaviti alat Argo CD koji će se koristiti i biti opisan kasnije u ovom diplomskom radu.

2.6.1. Izgradnja infrastrukture uz pomoć Terraforma

Cilj je uz pomoć Terraforma izgraditi infrastrukturu na oblaku pružatelja naziva AWS, potrebnu za razvoj aplikacija na Kubernetes grozdu. Za to su potrebni sljedeći resursi VPC s pripadajućim podmrežama i sigurnosnim grupama u kojima će se naći EKS grozd. Na slici (Slika 6) se vide datoteke čiji je nastavak „.tf“. To su konfiguracijske datoteke u kojima ćemo podesiti željene parametre.



Slika 6. Datoteke početnog Terraform direktorija

Unutar datoteke „eks-cluster.tf“ nalaze se vrijednosti koje grade Kubernetes grozd. Bitne vrijednosti koje se mogu podesiti su tip instanci koje grade grozd, minimalni i maksimalni broj instanci, sigurnosne grupe instanci, VPC i podmreže u kojima će biti stvorene, verzije Kubernetes grozda i ime grozda.

Datoteka „main.tf“ sadržava vrijednosti bitne za pružatelje modula i resurs koji generira nasumični niz. Svi pružatelji modula i njihove verzije su definirane unutar „terraform.tf“ datoteke.

U datoteci „outputs.tf“ definiraju se vrijednosti koje želimo iskoristiti kasnije. To je bitno kako bi se kasnije mogli povezati na Kubernetes grozd.

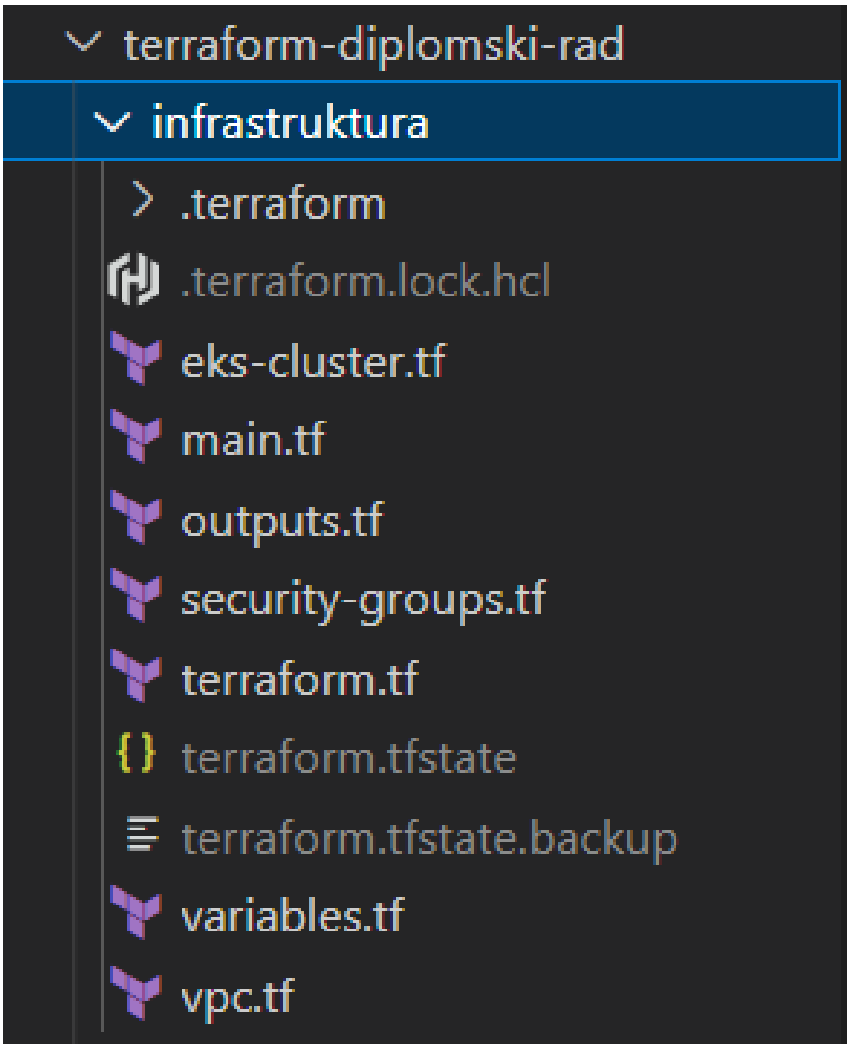
Datoteka „security-groups.tf“ je zadužena za izgradnju sigurnosnih grupa i tu se nalaze sve vrijednosti za ograničavanje i dopuštanje pristupa.

Varijable i njihove vrijednosti dodajemo u datoteku „variables.tf“, tu se dodavaju varijable koje se često izmjenjuju.

Na kraju, u datoteci „vpc.tf“ moguće je izmijeniti vrijednosti podmreža ili samog VPC-a, to jest njihove IP adrese.

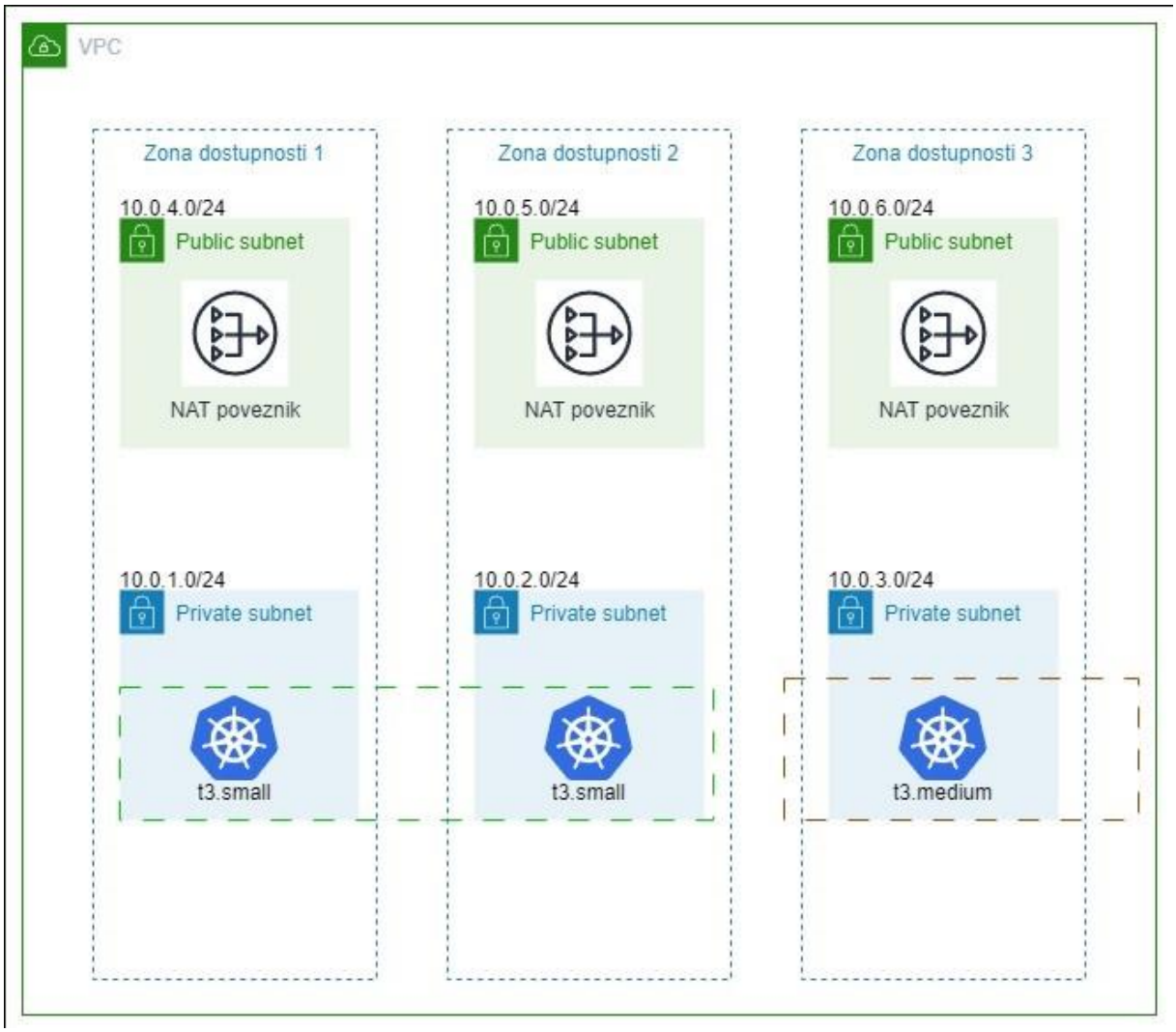
Kada se podese svi željeni parametri. Dovoljno je prvo inicijalizirati projekt tako što se pozicionira unutar direktorija infrastruktura i izvrši naredba „terraform init“. Nakon toga je proizvoljno izvršiti komandu „terraform validate“ i „terraform plan“. Nužna naredba nakon inicijalizacije je „terraform apply“ koja će stvoriti resurse na oblaku nakon samo nekoliko minuta. Slika (Slika 7) prikazuje direktorij „infrastruktura“ nakon izvršenja naredbi „terraform

init“ i „terraform apply“. Terraform je dodao nekoliko novih datoteka i jedan direktorij. Datoteka „terraform.tfstate“ je datoteka stanja. Ostale datoteke su datoteke pružatelja modula.



Slika 7. Direktorij nakon izvršenja naredbi

Slika (Slika 8) prikazuje skicu arhitekture infrastrukture koja se podigla na AWS oblaku. Unutar VPC-a stvorile su se privatne i javne podmreže svaka u svojoj zoni dostupnosti. Unutar privatnih podmreža je EKS grozd koji ima tri EC2 instance. Dvije slabije instance su tipa „t3.small“ i imaju svoju grupu čvorova dok je treća jača instanca u zasebnoj grupi čvorova i tipa je „t3.medium“. [19]



Slika 8. Skica infrastrukture

2.6.2. **Spajanje na Kubernetes grozd**

Kada se sve uspješno instaliralo, potrebno je konfigurirati kubectl koji vrši komunikaciju s glavnim Kubernetes čvorovima. U tome pomažu vrijednosti koje se definiraju unutar datoteke „outputs.tf“, parametar regija „region“ i parametar imena grozda „cluster_name“. Naredbom „aws eks --region $(terraform output -raw region) update-kubeconfig --name $(terraform output -raw cluster_name)“ dodajemo konfiguraciju unutar „kubeconfig“ datoteke iz koje kubectl izvlači vrijednosti.

Sada je dovoljno s naredbama „kubectl cluster-info“ i „kubectl get nodes“ provjeriti ispravnost spajanja na grozd. U slučaju postojanja više konfiguracija grozdova unutar datoteke „kubeconfig“ potrebno je promijeniti kontekst. To se radi uz pomoć naredbe „kubectl config use-context CONTEXT_NAME“.

2.6.3. **Postavljanje pomoćnih alata na Kubernetes uz pomoć Helm Charta**

Nakon izgradnje infrastrukture i spajanja na Kubernetes grozd. Cilj je postavljanje pomoćnog alata za razvoj aplikacija Argo CD pomoću Helma.

Prvi korak je pronalazak Argo CD Helm chart projekta. Na github profilu argoproj moguće je pronaći repozitorij argo-helm, koji sadrži svu dokumentaciju za instalaciju alata Argo CD na Kubernetes grozd pomoću Helma.

Drugi korak je spremanje datoteke „values.yaml“ lokalno u direktorij pod nazivom alati-helm. Dobra je praksa izmijeniti podrazumijevani naziv „values.yaml“ u „values-argo.yaml“. Slijedi podešavanje željenih parametara unutar same datoteke. Bitno je izmijeniti dio „argocdServerAdminPassword:“, jer je to lozinka koja je potrebna za prijavu u aplikaciju Argo CD.

Idući korak je dodati prostor imena naziva „argocd“ u koji ćemo smjestiti aplikaciju. To se radi uz pomoć naredbe „kubectl create namespace argocd“.

Posljednji korak je izvršenje naredbe „helm -n argocd upgrade --install argocd argo/argo-cd -f values-argo.yaml“. Zastavica „-n“ označava u koji prostor imena će se instalirati Helm chart. Dio naredbe „upgrade –install“ zadužen je za prvu instalaciju ili ažuriranje postojeće instalacije. Nakon toga ide ime Helm charta koji će se prikazivati unutar Kubernetesa te poslije toga ime repozitorija i ime aplikacije. Zadnja stavka koja se dodaje u naredbu označava iz koje datoteke se izvlače parametri za instalaciju.

3. Životni ciklus razvoja programa

Životni ciklus razvoja programa (*engl. Software Development Life Cycle* – SDLC) je proces koji se koristi u programskom inženjerstvu za dizajn, razvoj, testiranje i implementaciju visokokvalitetnog programa. SDLC pruža strukturiran i sustavan pristup izradi softvera, osiguravajući da zadovoljava potrebe korisnika i dionika, da je visoke kvalitete i da se isporučuje na vrijeme i unutar proračuna. Cilj SDLC-a je minimizirati rizike projekta kroz planiranje unaprijed tako da softver ispuni očekivanja kupaca tijekom proizvodnje i nakon toga. Ova metodologija ocrtava niz koraka koji dijele proces razvoja softvera na zadatke koje možete dodijeliti, izvršiti i mjeriti. [20]

Svaka od višestrukih faza ima jedinstven skup procesa i rezultata. Ispunjavanjem tih procesa i rezultata, pojedinci i timovi su bliže uspjehu. Također, SLDC im pomaže isporučiti najbolji mogući proizvod, stvarajući u konačnici sretne i lojalne kupce. SLDC obično ima od 5 do 7 faza. To su planiranje, prikupljanje i definiranje zahtjeva, dizajn, razvoj, testiranje, implementacija i održavanje. Timovi i voditelji biraju faze ovisno i njihovim potrebama.

U fazi planiranja se određuju raspodjela resursa, vremenski raspored, trošak projekta te se planira kapacitet projekta prije nego što se dogovore detalji. Tijekom ove faze definira se opseg projekta i identificiraju ciljevi. Faza prikupljanja i definiranja zahtjeva je nastavak na planiranje. Povratne informacije dionika sintetiziraju se i oblikuju u funkcionalne zahtjeve koje inženjeri i dizajneri mogu koristiti za izradu potrebnog posla. U fazi dizajniranja određuju se programski jezici koji će se koristiti, dogovara se arhitektura, sučelje sustava, dizajn korisničkog sučelja, slažu se dijagrami i svi ostali tehnički detalji koji pomažu u dizajniranju programa. Faza razvoja služi kako bi se sve isplanirano i dizajnirano provelo u djelo. U fazi testiranja softver se testira kako bi se osiguralo da ispunjava navedene zahtjeve. Ova faza uključuje funkcionalno testiranje, testiranje performansi i sigurnosno testiranje. U fazi postavljanja softver se pušta u proizvodnju. Ova faza uključuje instalaciju programa, obuku krajnjih korisnika i programsku podršku. U fazi održavanja program se održava i ažurira. Ova faza uključuje ispravke grešaka, poboljšanja značajki i ažuriranja programa kako bi bio ažuran s najnovijim tehnologijama. [21]

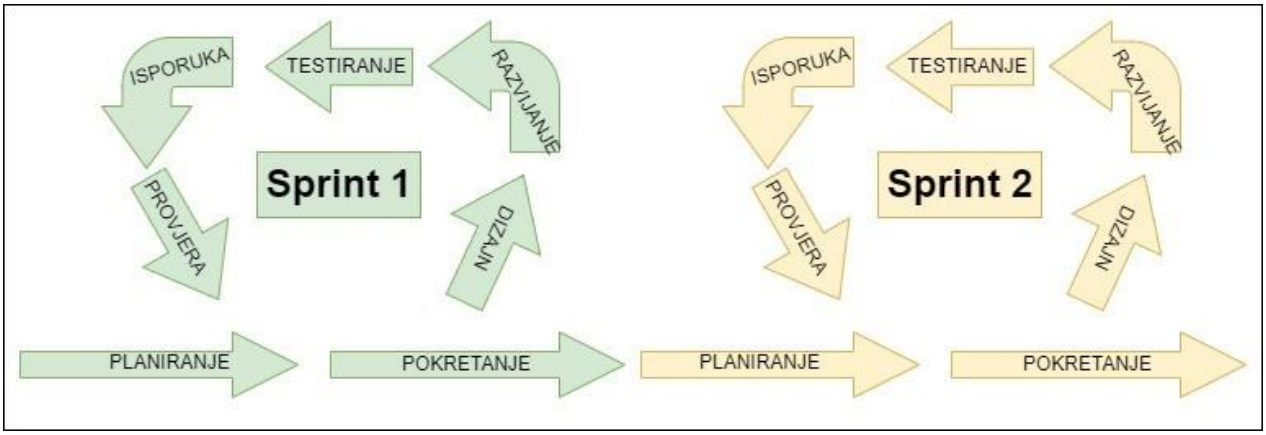
Različiti modeli i metodologije raspoređuju SDLC faze različitim kronološkim redoslijedom kako bi optimizirali razvojni ciklus. Neke od poznatijih metodologija su model vodopada, iterativni model, spirala te agilna metodologija. Model vodopada raspoređuje sve faze sekvencijalno tako da svaka nova faza ovisi o ishodu prethodne faze. Konceptualno, dizajn teče

iz jedne faze u drugu, poput vodopada. Iterativni proces sugerira da timovi započnu razvoj softvera s malim podskupom zahtjeva. Zatim iterativno poboljšavaju verzije tijekom vremena dok kompletan softver ne bude spreman za proizvodnju. Tim proizvodi novu verziju softvera na kraju svake iteracije. Spiralni model kombinira male ponovljene cikluse iterativnog modela s linearnim sekvencijalnim tokom modela vodopada kako bi se analizi rizika odredio prioritet. Možete koristiti spiralni model kako biste osigurali postupno izdavanje i poboljšanje softvera izgradnjom prototipova u svakoj fazi. Agilni model će se obraditi u sljedećem potpoglavlju. [20]

3.1. Agilna metodologija

Agilna metodologija je iterativni i inkrementalni pristup razvoju softvera koji naglašava zadovoljstvo korisnika, kontinuiranu isporuku i timsku suradnju. Prvi put ga je u Agile Manifestu 2001. predstavila skupina programera koji su željeli poboljšati tradicionalni model vodopada, koji je imao ograničenja u ispunjavanju promjenjivih zahtjeva kupaca i odgođeno vrijeme isporuke. Od tada je Agilna metoda postala jedna od najpopularnijih za razvoj programa, s mnogim dostupnim varijacijama i okvirima. [22]

Na slici (Slika 9) su prikazana dva sprinta. U Agilnoj metodologiji, sprint je vremenski ograničeno razdoblje tijekom kojeg se određena količina posla mora dovršiti i pripremiti za pregled. Obično je to razdoblje od jedan do četiri tjedna, najčešće su to dva tjedna. Sprint započinje planiranjem te dizajniranjem isplaniranog, slijedi razvijanje temeljeno na dizajnu. Nakon toga slijede testiranje i isporuka, te se napravljeno pregledava, ako je sve uredu slijedi pokretanje aplikacije odnosno puštanje u rad. Završetkom dogovorenog razdoblja kreće se u drugi sprint. [24]



Slika 9. Prikaz dva sprinta

Postoje 4 glavna stupa Agilnog pristupa koji se spominju u manifestu: [23]

1. Više cijenite ljude i njihove međusobne odnose nego procese i oruđa
2. Više cijenite upotrebljiv program nego iscrpnu dokumentaciju
3. Više cijenite suradnju s naručiteljem nego pregovaranje
4. Više cijenite reagiranje na promjenu nego ustrajanje na planu

Također, postoji i 12 principa Agilne metodologije:

1. Neka klijenti budu zadovoljni ranom i dosljednom isporukom vrijednosti
2. Podijeliti velike zadatke na manje zadatke koji se mogu brzo dovršiti
3. Shvatiti da timovi koji se sami organiziraju mogu dati najbolje rezultate
4. Pružiti pojedincima okruženje i podršku koja im je potrebna da obave posao
5. Stvoriti procese koji promiču održive napore tijekom vremena
6. Održavati stabilan ritam isporuke za završeni posao
7. Promjenjivi zahtjevi su dobrodošli u bilo kojem trenutku tijekom projektnog vremenskog okvira
8. Okupljanje projektnog tima i vlasnike tvrtki svakodnevno tijekom cijelog projekta
9. Redovito razmišljanje o tome kako poboljšati učinkovitost i prilagoditi razvojne prakse
10. Mjerenje napredaka količinom dovršenog posla vrijednog za kupca
11. Neprekidna težnja izvrsnosti
12. Iskorištavanje promjena kako bi se postigla konkurentska prednost

Sva ova načela ne određuju kako i koju tehnologiju koristiti u razvojnim procesima. No, razvitkom Agilne metodologije i raznih tehnologija javila se DevOps filozofija, gdje se pokušava spojiti programere i timove za infrastrukturu da rade isprepleteno. Cilj DevOps-a je pojednostaviti životni ciklus razvoja programa, većom suradnjom spomenutih timova i automatizacijom procesa gdje god je to moguće. DevOps filozofija potiče kulturu kontinuiranog poboljšanja, eksperimentiranja i učenja iz neuspjeha, kako bi se brže i pouzdanije isporučio softver visoke kvalitete. Iz Agilne metodologije, a kasnije DevOps filozofije se javlja potreba za automatizacijom i standardizacijom nekih procesa unutar ciklusa razvoja programa. Automatizacijom se razvijaju prvi cjevovodi za kontinuiranu integraciju i kontinuiranu isporuku.

3.2. Kontinuirana integracija i kontinuirana isporuka

Kontinuirana integracija i kontinuirana isporuka (*engl. Continuous Integration and Continuous Delivery/Deployment* - CI/CD) skup je praksi i načela kojima je cilj pojednostaviti i automatizirati proces razvoja i isporuke softvera. Engleski pojam CI/CD nije zapravo konstantan u svim izvorima. Drugi dio skraćenice CD može označavati dvije stvari, kontinuiranu isporuku (*engl. Continuous Delivery*) i kontinuirano raspoređivanje (*engl. Continuous Deployment*). U ovom diplomskom radu oba pojma će biti unificirana u jedan s pojmom kontinuirana isporuka. CI/CD uključuje integraciju promjena koda, automatiziranu izgradnju, sveobuhvatno testiranje i automatiziranu implementaciju kako bi se osigurala glatka i učinkovita isporuka softvera. Ovaj skup praksi zapravo automatizira velik dio ili sve ručne ljudske intervencije koje su potrebne da bi se novi kod prebacio u proizvodnju, to obuhvaća fazu izgradnje, testiranja (uključujući integracijske testove, jedinične testove i regresijske testove) i fazu postavljanja. [25]

Kontinuirana integracija je praksa u kojoj je cilj svakog razvojnog tima što više manjih promjena unutar koda, koji se zatim učitava u zajednički sustav git-a odnosno repozitorija za kontrolu verzija. Kada je izvorni kod učitan, slijedi automatska izgradnja, validacija i testiranje aplikacije. Prva faza je često validacija statičkom analizom koda. Nakon toga slijedi pakiranje i kompajliranje koda odnosno izgradnja. Dok je zadnja faza uobičajeno za testiranje. [25]

Nakon uspješne kontinuirane integracije slijedi kontinuirana isporuka. Kontinuirana isporuka nam omogućava postavljanje aplikacije u radna okruženja. Radna okruženja mogu biti razvojna koja se u engleskom obično nazivaju dev, testna okruženja su staging, te proizvodna okruženja production. Kako bi se izvela automatizirana isporuka bitno je postaviti kriterije koji se moraju zadovoljiti prije nego li aplikacija može stupiti u proizvodnju. Stoga kontinuirana isporuka ne postoji bez kvalitetne kontinuirane integracije. Timovi također mogu imati odluke da automatiziraju potpunu isporuku samo na neproizvodnim okruženjima. Takav pristup možda usporava isporuku aplikacija klijentima, ali štiti od neželjenih i slučajnih grešaka članova tima. [25]

Postoje osnove koje bi se svaka primjena CI/CD praksi trebala pridržavati. Izvorni kod se čuva na jednoj od platformi za git, zajedno sa svim potrebnim datotekama potrebnim za rad aplikacije. Postojanje jedinstvenih i stabilnih grana na git-u te je bitno biti u konstanti s tim granama. Također je bitno imati automatizirane izgradnje i testiranje, kao i česte promjene koda, bile one implementacija novih značajki ili ispravke grešaka. Još jedna bitna značajka je

imati stabilna radna okruženja i ne testirati u proizvodnom radnom okruženju. Za testiranje proizvodnog radnog okruženja bitno je imati identično radno okruženje koje će biti namijenjeno za testiranje. Svaki programer mora moći vidjeti sve promjene i najnovije podatke koda kako bi se mogao pratiti napredak i potencijalne greške u kodu. Uz to programeri moraju biti naviknuti i komforni na promjene u bilo koje vrijeme. [25]

3.2.1. Koristi kontinuirane integracije i kontinuirane isporuke

Prakse kontinuirane integracije i kontinuirane isporuke nude brojne prednosti za organizacije koje žele pojednostaviti procese razvoja i isporuke softvera. CI/CD omogućuje brzu isporuku softvera automatiziranjem različitih faza procesa razvoja i postavljanja. Automatizacijom integracije koda, testiranja i implementacije, organizacije mogu značajno smanjiti vrijeme potrebno za uvođenje novih značajki i ispravke grešaka. Ovo ubrzano vrijeme izlaska na tržište osigurava konkurentsku prednost dopuštajući organizacijama da brzo odgovore na potrebe kupaca. [25]

Ranim i čestim testiranjem tijekom životnih ciklusa razvoja, održava se visoka kvaliteta programa što rezultira zadovoljnijim klijentima. Također osigurava manje potrebe za nenadanim intervencijama što olakšava programerima održavanje aplikacija. [25]

Automatizacijom se ubrzava rad programera, tako što vrijeme koje bi programeri provodili na ručnim procesima, mogu iskoristiti na pametniji način. Ubrzavanjem programera, ubrzava se i proces razvoja cijelog sustava na kojem rade, što omogućava skraćenje rokova. Također se automatizacijom dolazi do trenutačnog dobivanja povratnih informacija koje su od ključnog značaja za poboljšavanje programa. Organizacije ovim pristupom njeguju način razmišljanja stalnog poboljšanja te to dovodi do sve boljih programskih rješenja. Bržim rješavanjem grešaka, timovi su u mogućnosti brže odgovoriti na incidente u proizvodnji. Ovo produžava vrijeme rada sustava što rezultira podizanjem prihoda i reputacije poslovanja. [25]

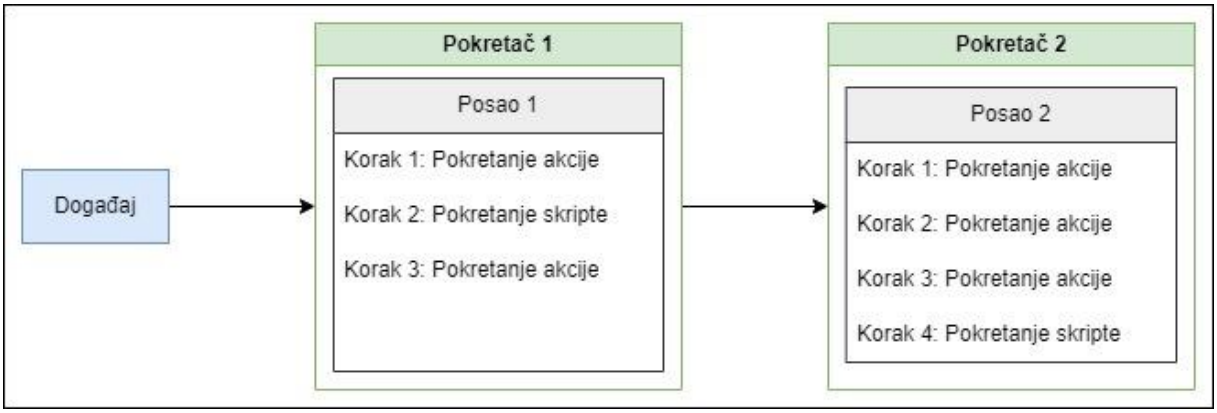
3.2.2. Alat GitHub Actions

Skup CI/CD praksi koje su implementirane u nekom alatu naziva se cjevovod (*engl. Pipeline*). To je tijek rada (*engl. workflow*) koji definira faze (*engl. stages*), radnje (*engl. actions*) i ovisnosti (*engl. dependencies*) potrebne za izgradnju, testiranje i implementaciju promjena programa. Alat GitHub Actions je platforma dizajnirana za implementaciju CI/CD cjevovoda. Platforma je nastala uz već postojeći sustav kontrole verzija GitHub. Pomoću alata GitHub Actions programeri mogu definirati tijekove rada, automatizirati zadatke, izgraditi, testirati i

implementirati svoje aplikacije. GitHub Actions pruža visok stupanj prilagodbe, što omogućuje razvojnim programerima da definiraju vlastite tokove rada i prilagode ih specifičnim zahtjevima. Ti tokovi rada mogu sadržavati uvjete, paralelne ili sekvencijalne korake, varijable okoline (*engl. Environment variables*) i tajne (*engl. Secrets*) za sigurno pohranjivanje podataka. Ova fleksibilnost omogućuje programerima prilagođavanje radnih procesa njihovim specifičnim razvojnim metodama i potrebama projekta. GitHub Actions potiče suradnju i angažiranost zajednice. Razvojni programeri mogu dijeliti svoje tokove rada kao predloške za višekratnu upotrebu, omogućavajući drugima da iskoriste i doprinesu njihovim naporima u automatizaciji. Zajednica GitHub Actions je aktivna, s ogromnom kolekcijom otvorenog koda za radne procese i akcije. [26]

3.2.3. Glavne značajke alata GitHub Actions

Komponente GitHub Actions-a bitne za razumijevanje rada alata su tijek rada (*engl. Workflow*), događaj (*engl. Event*), poslovi (*engl. Jobs*), korak (*engl. Step*), akcije (*engl. Actions*) i pokretač (*engl. Runner*). Na slici (Slika 10) je skiciran jedan tijek rada GitHub Actionsa i prikazane su njegove komponente. [26]



Slika 10. Primjer tijeka rada

Tijek rada je konfigurabilni automatizirani proces koji pokreće jedan ili više poslova. Za definiranje se koriste YAML datoteke koje se postavljaju unutar repozitorija nad kojim se želi pokrenuti tijek rada. Unutar repozitorija se postavi direktorij „.github/workflows“ te se unutar tog direktorija dodaju tijekovi rada s YAML sintaksom. Mogu se pokretati ručno, prema rasporedu ili okidanjem događaja kao što su slanje koda u repozitorij ili zahtjevom za spajanje grana. Događaj je specifična aktivnost u repozitoriju koja pokreće tijek rada. [26]

Poslovi su skupovi koraka unutar tijeka rada koji se pokreću na istom pokretaču. Svaki korak može biti shell skripta ili akcija. Koraci se izvršavaju redom i ovise jedan o drugom, stoga nije

moguće preskakati korake unutar jednog posla. Koraci unutar istog posla koji se izvršavaju na istom pokretaču mogu koristiti iste podatke. Poslovi se obično pokreću paralelno, ali je moguće dodati ovisnosti. Što znači da će posao koji ovisi o drugom poslu, čekati njegov završetak. [26]

Akcije su prilagođene aplikacije za neku namjenu. Obavljaju složene zadatke koji se često ponavljaju. Koriste se kako bi se smanjio ponavljajući kod u datotekama. Mogućnosti s akcijama su beskonačne, a najčešće se koristi akcija za povlačenje koda s git repozitorija na pokretač. Moguće je pisati vlastite akcije ili koristiti postojeće koje se mogu pronaći na GitHub Marketplaceu. Pokretač je poslužitelj koji pokreće tijekove rada. Svaki pokretač može izvoditi jedan posao odjednom. GitHub omogućuje pokretače sa sljedećim operacijskim sustavima: Ubuntu Linux, Microsoft Windows i macOS. Pokretači su besplatni za javne repozitorije dok se privatni repozitoriji naplaćuju ovisno o minutama rada na mjesečnoj bazi. Svaki tijek rada se pokreće na novom i tek kreiranom virtualnom stroju koji možda nema sve željene programe instalirane na njemu, pa je ponekad potrebno prvo instalirati potrebne programe. Također je moguće izraditi vlastite pokretače koji će biti neovisni o GitHub platformi i tada su oni besplatni. No, onda je potrebno brinuti se o ažuriranjima i održavanju tih pokretača na vlastitim resursima. [26]

3.3. GitOps

GitOps je metoda u programskom razvoju kod koje je Git jedini izvor točnih podataka za automatizaciju i baratanje implementacijama u Kubernetesu ili infrastrukturi oblaka. Ali u ovom diplomskom radu fokus je na GitOps praksama koje se koriste unutar Kubernetesa. Metoda kombinira načela kontrole verzija, deklarativne konfiguracije i kontinuirane isporuke kako bi se osiguralo da se željeno stanje sustava uvijek odražava u stvarnom stanju. Obično se deklarativna konfiguracija piše u YAML datotekama. GitOps tijek rada uključuje proces koji kontinuirano nadzire Git repozitorij i primjenjuje sve promjene. Ovaj proces često olakšava GitOps alat ili platforma koja čita željeno stanje iz Git repozitorija i automatski ga primjenjuje na radno okruženje. Ovo osigurava da je sustav uvijek u željenom stanju kako je definirano u Git repozitoriju. GitOps ubrzava i poboljšava rad programera jer programer jedino što treba je napraviti svoj dio u izvornom kodu i predati kod na Git repozitorij. Onda se GitOps alatima kontinuirano isporučuje aplikacija na sva radna okruženja. Obzirom da je sve unutar Git repozitorija, postoji jedini izvor istine (*engl. Single source of truth*) koji također pomaže kod incidenata i vraćanje u neka prethodna stabilna stanja. Samo korištenje Gita daje veću sigurnost zbog toga što Git koristi kriptografiju. Isto tako zna se tko je i kada stavio kod na Git te je

moguće postaviti provjere koda prije spajanja s glavnim granama. Postoje dva načina kod kojih GitOps alati prate promjene na repozitoriju. Događaj guranja (*engl. push*) koda je način u kojem programer gurne kod na repozitorij i tada aktivira CI/CD cjevovod. Drugi način je povlačenje (*engl. pull*) kod kojeg alat kontinuirano prati repozitorij i povlači promjene s njega. [27]

3.3.1. **Alat Argo CD**

Službena dokumentacija dijeli kratak opis u jednoj rečenici. Argo CD je deklarativni GitOps alat za kontinuiranu isporuku za Kubernetes. Svi pojmovi ove rečenice su opisani u ovom radu, ali ovo poglavlje će se baviti konceptima i razlozima za korištenje baš ovog alata. Argo CD je moćan i popularan alat koji donosi nekoliko prednosti procesima implementacije. Slijedeći GitOps metodologiju, Argo CD promiče korištenje Gita kao jedinog izvora istine, pružajući kontrolu verzija, mogućnost revizije i ponovljivost konfiguracija aplikacija. Argo CD omogućava definiranje aplikacije deklarativno koristeći YAML manifeste, osiguravajući da je željeno stanje aplikacija jasno definirano i lako verzionirano. Ovaj alat ima mogućnost automatske sinkronizacije. Što znači da kontinuirano prati promjene željenog Git repozitorij i automatski sinkronizira stvarno stanje aplikacija sa željenim stanjem definiranim u Gitu. To eliminira potrebu za ručnom intervencijom i osigurava uvijek ažurnu implementaciju aplikacija. Kao alat koji ide ruku pod ruku s Kubernetesom, lako je integriran s Kubernetes API-jima te koristi Kubernetes resurse kako bi upravljao postavljanjem i ažuriranjem aplikacija. Isto tako omogućuje skalabilnost i otpornost. Popularnost alata doprinosi tome da su verzije ažurne te se otkrivene greške brzo rješavaju. [28]

Bitno je naglasiti glavne koncepte alata. Aplikacija (*engl Application*) je grupa Kubernetes resursa kako je definirano nekim manifestom. Aplikacija je CRD (*engl. Custom Resource Definition*), to je produženi Kubernetes API s kojim se može definirati vlastite tipove resursa. Vrsta izvora aplikacije (*engl. Application source type*) označava koji se alat koristi za izradu aplikacije, mogućnosti su direktno git repozitorij, Helm chart, Kustomize direktoriji ili drugo. Ciljano stanje (*engl. Target state*) je željeno stanje aplikacije. Stanje uživo (*engl. Live state*) je stanje uživo te aplikacije. Status sinkronizacije (*engl. Sync status*) daje odgovor na pitanje je li ciljano stanje i stanje uživo jednako. Sinkronizacija (*engl. Sync*) je proces pokretanja aplikacije u ciljano stanje, odnosno primjena promjena na Kubernetesu. Status operacije sinkronizacije (*engl. Sync operation status*) govori je li sinkronizacija uspjela ili ne. Osvježi (*engl. Refresh*) uspoređuje najnoviji kod u gitu sa stanjem uživo i gleda što je drugačije. Zdravlje (*engl. Health*) gleda zdravlje aplikacije, odgovara na pitanje je li aplikacija spremna za rad. [28]

4. Razvoj cjevovoda za kontinuiranu integraciju i kontinuiranu isporuku

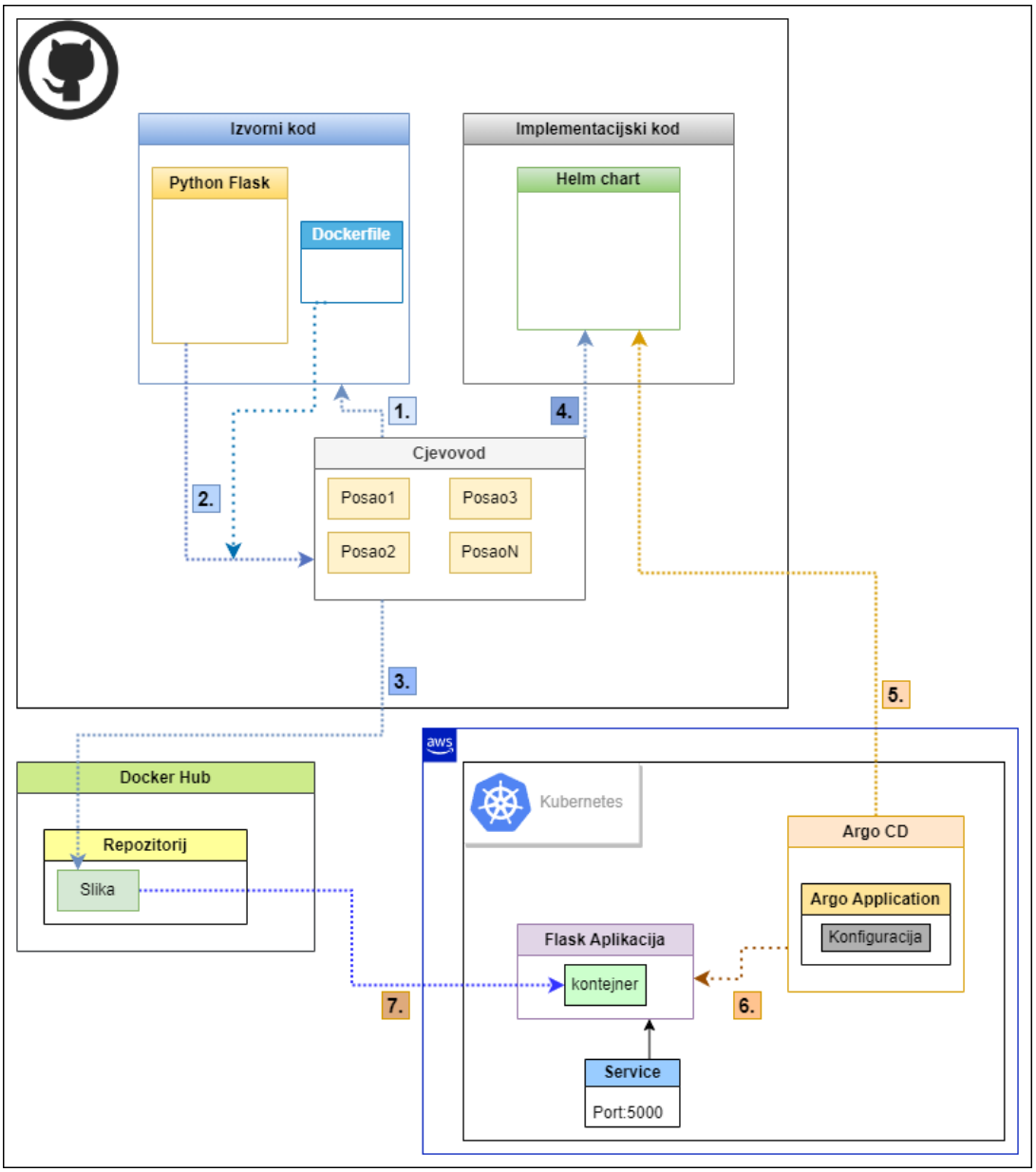
U ovom poglavlju prikazat će se izgradnja cjevovoda za kontinuiranu integraciju i kontinuiranu isporuku jednostavne Python Flask aplikacije. Cilj cjevovoda je testiranje, izgradnja i slanje Docker slike na javni repozitorij, verzioniranje i automatska implementacija aplikacije na Kubernetes grozd. U prethodnim poglavljima opisani su i implementirani dijelovi arhitekture koji će biti korišteni pri izgradnji cjevovoda.

4.1. Arhitektura cjevovoda

Za implementaciju cjevovoda kontinuirane integracije i kontinuirane isporuke na standardiziran, sistematiziran i skalabilan način potrebno je koristiti više alata. Platforma GitHub koristit će se za pohranu izvornog i implementacijskog koda. GitHub Actions koji pokreće cjevovod smješten je unutar repozitorija izvornog koda. Unutar Kubernetes grozda koristiti će se Argo CD za automatsku implementaciju aplikacije. Korištenjem alata Docker izgraditi će se Dockerfile, Docker slika, te će se također iskoristiti mogućnost platforme Docker Hub kao repozitorij za pohranu Docker slika. Kubernetes manifesti zaduženi za normalan rad aplikacije biti će zapakirani u Helm chart koji će se pohraniti u vlastiti repozitorij na platformi GitHub. Kubernetes je instaliran na platformi AWS pomoću Terraforma.

Na slici (Slika 11) su prikazani koraci koje cjevovod kontinuirane integracije i kontinuirane isporuke izvodi:

1. Cjevovod prati promjene u repozitoriju izvornog koda
2. Nakon željenog okidača cjevovoda, primjerice izmjene koda, aktivira se cjevovod
    - Slijedi testiranje aplikacije i izgradnja Docker slike
3. Slanje Docker slike na Docker Hub repozitorij
4. Verzioniranje i izmjena vrijednosti verzija unutar Helm chart
5. Argo CD kontinuirano prati izmjene unutar repozitorija odnosno Helm charta
6. Nakon što primijeti promjenu, slijedi automatska sinkronizacija aplikacije na Kubernetesu
7. Posljednji korak je preuzimanje nove Docker slike s repozitorija to jest ažuriranje aplikacije



Slika 11. Arhitektura cjevovoda

4.2. Python Flask web aplikacija

Za prikaz rada cjevovoda nad nekim programskim kodom, izabrana je jednostavna Python Flask web aplikacija. U nastavku je prikazan kod aplikacije. Aplikacija će na http zahtjev http://localhost:5000/ vraćati niz „Pozdrav, diplomskom radu!“.

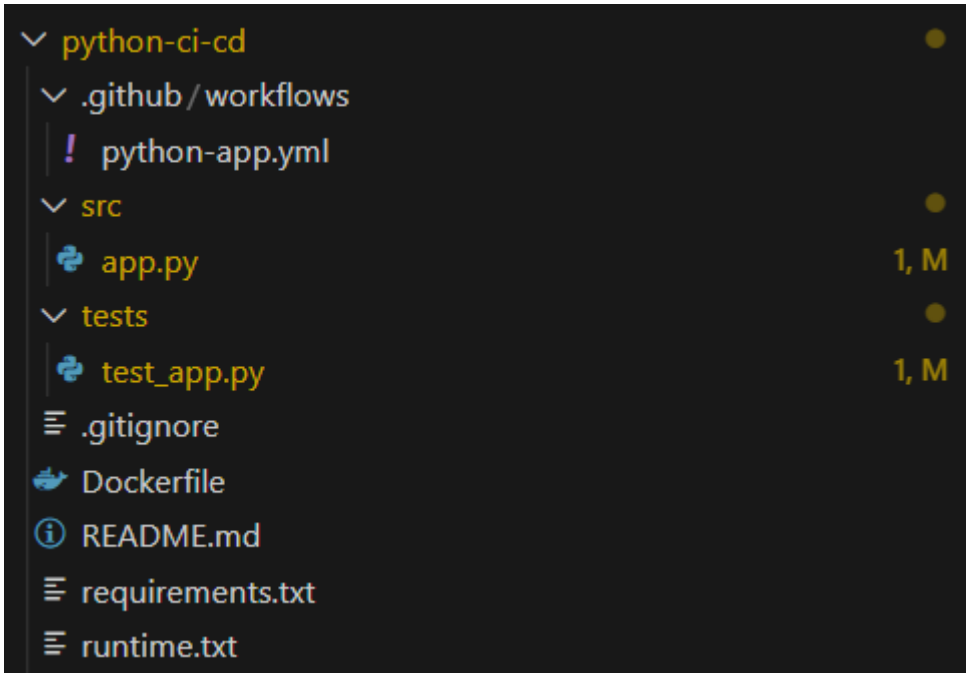
```
from flask import Flask

app = Flask(__name__)

@app.route("/")
```

```
def index():
    return "Pozdrav, diplomskom radu!"

if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Kako bi se aplikacija testirala koristit će se modul pytest. Testirat će se ispravnost vraćenog odgovora na http zahtjev.

```
from app import index

def test_index():
    assert index() == "Pozdrav, diplomskom radu!"
```

Na slici (Slika 12) je prikazan izgled repozitorija izvornog koda. Već spomenuti kod nalazi se u direktorijima „src“ i „tests“. Osim toga u repozitoriju se nalaze „.gitignore“ koji će ignorirati sve izgenerirane datoteke koje ne trebaju u git repozitoriju. Requirments.txt sadrži module bez kojih aplikacija ne može, te runtime.txt prikazuje verziju Pythona koji se koristi. U direktorij „.github/workflows“ postavlja se cjevovod aplikacije. Naziv direktorija je bitan jer GitHub na taj način prepoznaje gdje se nalaze instrukcije cjevovoda za njegovo pokretanje.



Slika 12. Repozitorij izvornog koda

Aplikacija će se zapakirati u Docker sliku uz pomoć Dockerfilea koji je dan u nastavku. Za baznu sliku koristi se python:3.9-slim, za radni direktorij postavlja se „/app“, svi potrebni moduli instaliraju se naredbom „pip3.9“ iz datoteke „requirements.txt“. Nakon toga se kopira sadržaj izvornog koda i izlaže se aplikaciju na port 5000. U zadnjem koraku se pokreće aplikacija Python verzijom 3.9. Naredba „docker build -t helloflask:latest“ izrađuje Docker sliku po imenu „helloflask“ s verzijom latest.

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip3.9 install --no-cache-dir -r requirements.txt

COPY src/app.py .

EXPOSE 5000

CMD ["python3.9", "app.py"]
```

4.3. Poslovi kontinuirane integracije

Cjevovod se konfigurira u datoteci python-app.yaml. Na početku je potrebno definirati ime cjevovoda. Zatim okidače koji će aktivirati cjevovod, to će biti „push“ i „pull_request“ na main granu.

```
name: Python application

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
```

Nakon toga se definiraju poslovi. Poslovi kontinuirane integracije su testiranje, izgradnja Docker slike i njezino slanje na repozitorij.

4.3.1. Testiranje Flask aplikacije

Prvi posao koji će cjevovod obaviti je testiranje. Za pokretača je izabrana klasična instanca ubuntu operacijskog sustava. Slijede koraci

1. Kloniranje repozitorija izvornog koda
2. Instalacija Python verzije 3.9 na instancu pokretača
3. Instalacija svih potrebnih modula na instancu
4. Provođenje lintera, testiranja programerske konvencije odnosno statička analiza koda
5. Posljednji korak je testiranje s pytestom, potrebno je definirati varijablu okruženja PYTHONPATH kako bi pytest znao gdje se nalazi izvorni kod koji treba testirati

```
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Clone repo in runner
      uses: actions/checkout@v3
    - name: Set up Python 3.9
      uses: actions/setup-python@v3
      with:
        python-version: "3.9"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127
--statistics
    - name: Test with pytest
      run: |
        export PYTHONPATH=src
        pytest
```

4.3.2. Izgradnja i slanje Docker slike na repozitorij

Prije nego krene ovaj posao, prethodni posao mora biti uspješan. Stoga je na početku dodan dio „needs“ s vrijednosti test, odnosno trenutni posao treba uspješan test posao kako bi se pokrenio. Za izgradnju i slanje Docker slike na Docker Hub repozitorij poslužit će već postojeći GitHub Actionsi „Login to Docker Hub“ pomoću kojega se prijavljuje na platformu te „Build and push Docker images“ koji izrađuje Docker sliku i šalje ju na repozitorij.

Repozitorij imena mislavhuddle je definiran unutar ključa „tags“, a unikatni Docker tag je postignut „github.sha“ varijablom. Kako bi senzitivne podatke sakrili od javnosti korištena je značajka „secrets“ u koju je definirano korisničko ime i token za prijavu na Docker Hub.

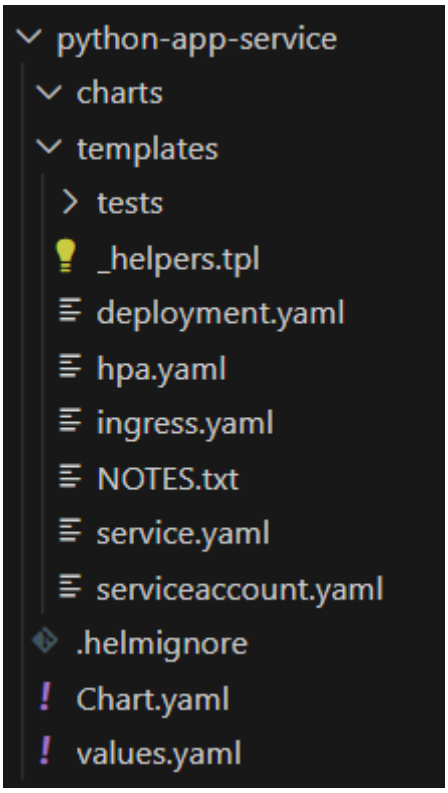
```
  build-and-push-docker-image:
    needs: test
    runs-on: ubuntu-latest
    steps:

    - name: Clone repo in runner
      uses: actions/checkout@v3

    - name: Login to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKERHUB_USERNAME }}
        password: ${{ secrets.DOCKERHUB_TOKEN }}

    - name: Build and push Docker images
      uses: docker/build-push-action@v4.0.0
      with:
        push: true
        tags: mislavhuddle/helloflask:${{ github.sha }}
```

4.4. Posao kontinuirane isporuke

Prije dodavanja posljednjeg posla u cjevovod, potrebno je izraditi repozitorij u kojem će se nalaziti Helm chart i testirati ga. Nakon validacije i provjere rada instalacije aplikacije na Kubernetes pomoću Helm charta, slijedi dodavanje Argo aplikacije. Argo aplikacijom se definira koji git repozitorij će alat Argo CD pratiti i povlačiti s njega promjene. Povlačenjem promjena i automatskim sinkroniziranjem stanja s gita u stanje s Kubernetesa, postiže se kontinuirana isporuka aplikacije. Kako bi se promjene ispravno prikazivale i okidale Argo CD sinkroniziranje, pomoću cjevovoda će se izmjenjivati verzija Helm charta i Docker taga u repozitoriju Helm charta odnosno implementacijskog koda.

4.4.1. Implementacija aplikacije pomoću Helm charta

Aplikacija će se implementirati uz pomoć Helm charta koji će biti smješten u vlastiti git repozitorij odvojen od izvornog koda. Izrada Helm charta lako se može izvesti pomoću naredbe „helm create python-app-service“. Na slici (Slika 13) je prikaz datoteka kreiranih komandom. Nužne izmjene vrijednosti koje će se napraviti, nalaze se unutar datoteka „values.yaml“ i „deployment.yaml“.



Slika 13. Helm chart

Unutar datoteke „values.yaml“, ime slike i tag treba izmijeniti u vlastite vrijednosti, kako slijedi. Dodatno, da bi se dobila visoko raspoloživa aplikacija sa strane Kubernetes arhitekture dobro je dodati više replika kapsula. Stoga za ključ replicaCount potrebno je postaviti vrijednost 3. Na taj način jednostavno se može izvesti horizontalno skaliranje. Vertikalno skaliranje je moguće napraviti izmjenama „cpu“ i „memory“ vrijednosti ispod ključa „resources“. Na kraju za ime resursa postavit će se „hello-flask-app“.

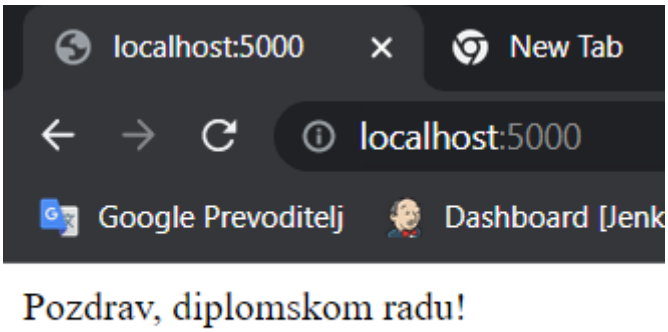
```
replicaCount: 3

image:
  repository: mislavhuddle/helloflask
  pullPolicy: IfNotPresent
  tag: latest

nameOverride: "hello-flask-app"
fullnameOverride: "hello-flask-app"
```

U „deployment.yaml“, mijenja se containerPort u port 5000 jer se aplikacija izlaže na taj port.

Također, kako bi za aplikaciju mogli koristiti prosljeđivanje porta (*engl.* Port forward) na lokalno računalo, za tip servisa treba postaviti NodePort.

Validaciju Helm charta i generiranje svih Kubernetes manifesta u jednu datoteku pod imenom „__debug-tpl-0“, moguće je postići naredbom „helm template --debug python-app-service python-app-service/ -f python-app-service/values.yaml > __debug-tpl-0“. Ako je sve prošlo uredu, unutar datoteke će se pojaviti svi Kubernetes manifesti koji će se instalirati na grozd ukoliko se pokrene naredba „helm upgrade --install python-app-service python-app-service/ -f python-app-service/values.yaml“. Validaciju instaliranih resursa moguće je napraviti u nekoliko koraka:

1. Dohvaćanjem svih resursa u imenskom prostoru „default“ naredbom „kubectl get all -n default“
2. Provjera spremnosti resursa – „Ready“ 1/1
3. Provjera kapsule „hello-flask-app-test-connection“, koja bi trebala imati vrijednost „STATUS Completed“
4. Posljednje provjera prosljeđivanjem porta na lokalno računalo naredbom „kubectl -n default port-forward service/hello-flask-app 5000:5000“
5. Nakon toga je dovoljno otvoriti u pregledniku http://localhost:5000/
6. Na slici (Slika 14) se može vidjeti stanje ispravne aplikacije



Slika 14. Prikaz ispravnosti aplikacije

Nakon validacije ispravnosti instaliranih resursa, instalirani chart je moguće obrisati naredbom „helm -n default delete python-app-service“. Ispravni Helm chart potrebno je učitati na git repozitorij pod nazivom „python-app-deployment“.

4.4.2. Dodavanje aplikacije na Argo CD

Argo CD kao GitOps alat koji prati promjene git repozitorija, na neki način mora imati konfiguraciju u kojoj će biti definirano koju granu i git repozitorij će pratiti. Za to služi CRD Application. Argo Application se konfigurira yaml sintaksom i implementira naredbom

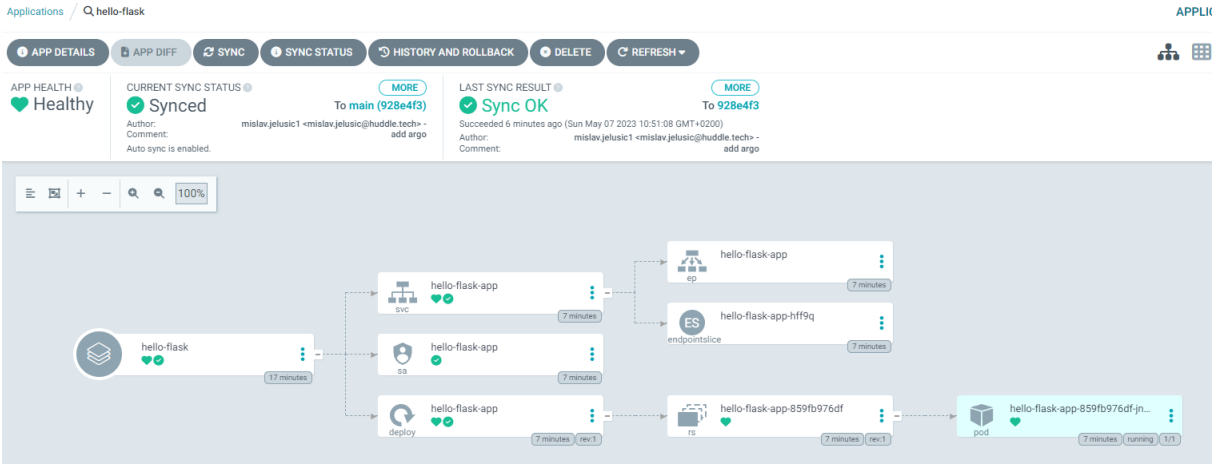
„kubectl -n argocd apply -f argo-application.yaml“. U nastavku je dana konfiguracija datoteke „argo-application.yaml“:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: hello-flask
  namespace: argocd
spec:
  syncPolicy:
    automated:
      prune: true
  project: default
  source:
    repoURL: https://github.com/meslav-ye/python-app-deployment.git
    targetRevision: main
    path: python-app-service
  destination:
    server: https://kubernetes.default.svc
    namespace: default
```

Aplikacija prati prethodno kreirani git repozitorij, main granu te relativni put ka Helm chartu. Također je definirano da će se manifesti implementirati na trenutni Kubernetes grozd u kojem je instaliran alat Argo CD. Podešeno je automatsko sinkroniziranje „syncPolicy“, te opcija „prune“ koja će pri ažuriranju obrisati resurse koji ne postoje u manifestu, ali se nalaze na grozdu. Ova opcija je dobra ako se želi postići GitOps održavanje stanja.

4.4.3. Automatska implementacija aplikacije

Nakon dodavanja Argo aplikacije, provjera se može napraviti na Argo CD grafičkom sučelju. Prvo se treba proslijediti port Argo CD servisa, to se može napraviti naredbom „kubectl -n argocd port-forward svc/argocd-server 8080:443“. Prijava u Argo CD grafičko sučelje zahtjeva korisničko ime i lozinku. Lozinku je moguće dohvatiti naredbom „kubectl get secret argocd-initial-admin-secret -n argocd -o 'go-template={{.data.password | base64decode}}'“. Dok je korisničko ime „admin“. Nakon prijave, odabirom „hello-flask“ Argo aplikacije otvoriti će se prozor kao na slici (Slika 15). Tu je moguće provjeriti ispravnost cijele implementacije Python aplikacije. Također, moguće je napraviti istu validaciju kao u prethodnom primjeru.



Slika 15. Prikaz Argo aplikacije

4.4.4. Automatsko verzioniranje

Sada je sve spremno da se u cjevovod doda posljednji posao. Posao će raditi

1. Kloniranje implementacijskog git repozitorija
2. Dohvaćanje trenutne verzije iz Helm charta
3. Povećavanje verzije Helm charta i izmjena Docker taga u datoteci values.yaml
4. Slanje izmjena na implementacijski git repozitorij

Ovaj posao je bitan za kontinuiranu isporuku, kao što je već navedeno alat Argo CD će sada pokupiti sve promjene. Kako je u cjevovod uvedeno automatsko verzioniranje, verzionira se implementacijski git repozitorij. Argo CD prati taj repozitorij te radi na automatskoj isporuci aplikacije pri svakoj novoj promjeni verzije taga i Helm charta.

Posljednji posao odnosno dio cijelog cjevovoda je dan u nastavku:

```
5.   Deploy:
6.     needs: build-and-push-docker-image
7.     runs-on: ubuntu-latest
8.     steps:
9.
10.      - name: Checkout repo
11.        uses: actions/checkout@v2
12.        with:
13.          repository: meslav-ye/python-app-deployment.git
14.          ref: main
15.          token: ${{ secrets.CHART_REPO_TOKEN }}
16.
17.      - name: Get current version
```

```
18.         run: echo "Current version is $(cat python-app-
    service/Chart.yaml | grep version | cut -d ' ' -f 2)"
19.
20.       - name: Increment version
21.         run: |
22.           VERSION=$(cat python-app-service/Chart.yaml | grep version |
    cut -d ' ' -f 2)
23.           NewVERSION=$(echo $VERSION | awk -F '.' '{print
    $1"."$2"."$3+1}')
24.           sed -i "s/version: $VERSION/version: $NewVERSION/g" python-
    app-service/Chart.yaml
25.           currTag=$(cat python-app-service/values.yaml | grep -w tag |
    awk {'print $2'})
26.           sed -i "s/tag: $currTag/tag: ${{ github.sha }}/g" "python-app-
    service/values.yaml"
27.
28.       - name: Commit and push changes
29.         run: |
30.           git config --global user.email "misso998@gmail.com"
31.           git config --global user.name "Mislav Jelusic"
32.           git add .
33.           git commit -m "Update version to $(cat python-app-
    service/Chart.yaml | grep version | cut -d ' ' -f 2)"
34.           git push origin main
35.
```

„CHART_REPO_TOKEN“ je token za pristup git repozitoriju iz instance pokretača. Za izmjenu verzija koristi se bash naredba „sed“. Posljednji korak su git naredbe.

Uz dodavanje posljednjeg posla, zaokružena je cijela priča kontinuirane integracije i kontinuirane isporuke.

Zaključak

Ovaj diplomski rad prikazuje primjer čitavog razvoja programa od njegovog početka do postavljanja čvrstih temelja za nastavak razvijanja visoko raspoloživih i skalabilnih aplikacija. Prvo je prikazan alat za orkestraciju kontejnera Kubernetes zajedno s upraviteljem paketa Helm. Alatom Helm organizacije mogu zapakirati sve ovisnosti i konfiguracije za Kubernetes aplikacije, omogućujući dosljednost i jednostavnu implementaciju u različitim okruženjima. Kako bi organizacije jednostavnije i brže prešle na Kubernetes, prikazana je implementacija Kubernetesa u oblaku AWS. Implementacija upravljanog Kubernetesa na AWS-u napravljena je uz pomoć alata Terraform, koji kao i Helm omogućava dosljednost i jednostavnost implementacije resursa. Izgrađen je Kubernetes s tri radna čvora. Svaki od njih u svojoj zoni dostupnosti radi bolje raspoloživosti. Na Kubernetes je implementiran alat Argo CD uz pomoć Helma. Nadalje opisan je životni ciklus razvoja programa, uz metodologije kontinuirane integracije i kontinuirane isporuke, GitOpsa i agilne metodologije gdje su opisane prednosti i mane. Na kraju na konkretnoj Python aplikaciji provedena je kontinuirana integracija i kontinuirana isporuka i opisani su svi koraci. Aplikacija je isporučena u tri kapsule kako bi se prikazalo jednostavno horizontalno skaliranje. Nastavak ovog diplomskog rada bi bilo omogućavanje autoskaliranja. Uz to, postavljanje kapsula na sve radne čvorove podjednako. Obije opcije moguće je konfigurirati samo unutar Kubernetesa. Također, implementacija dostupnosti aplikacije na Internet putem URL-a uz pomoć ingressa i load balancera. Za DNS konfiguraciju moguće je koristiti AWS-ov servis „Route 53“. Cjevovod je moguće poboljšati integracijskim testovima koji bi se pokretali na zasebnom okruženju. Također, izradom više od jednog okruženja, dobiva se prostor za kvalitetnije testiranje i smanjuje mogućnost pogreške na produkcijskoj aplikaciji. Više okruženja bi dovelo i do izmjene na cjevovodu zbog implementacije procedure objavljivanja aplikacije.

Literatura

[1] A. Faizi, When to use Kubernetes?, Web link: https://medium.com/@asad_5112/when-to-use-kubernetes-fea0dc677a8c , Datum pristupa: 1.4.2023.

[2] "*Microservices: a definition of this new architectural term*", Web link: https://martinfowler.com/articles/microservices.html , Datum pristupa: 2.4.2023.

[3] *Microservices Architecture*, Web link: https://microservices.io/ , Datum pristupa: 2.4.2023.

[4] M. Lukša, "Kubernetes in Action", Manning Publications, 2019

[5] "*What are containers?*", Web link: https://www.netapp.com/devops-solutions/what-are-containers/ , Datum pristupa: 3.4.2023.

[6] "*What is hypervisor*", Web link: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor , Datum pristupa: 3.4.2023.

[7] "*What is Container orchestration*" by Middleware Team, Web link: https://middleware.io/blog/what-is-container-orchestration/ , Datum pristupa: 5.4.2023.

[8] "*Kubernetes dokumentacija*", Web link: https://kubernetes.io/docs , Datum pristupa: 11.4.2023.

[9] "*Helm dokumentacija*", Web link: https://helm.sh/docs/ , Datum pristupa: 18.3.2023.

[10] M. Boisvert, S. J. Bigelow, W. Chain, Infrastructure as a Service, Web link: https://www.techtarget.com/searchcloudcomputing/definition/Infrastructure-as-a-Service-IaaS Datum pristupa: 10.3.2023.

[11] "*What is cloud computing?*", Web link: https://aws.amazon.com/what-is-cloud-computing/ , Datum pristupa: 10.3.2023.

[12] "*Regions, Availability Zones, and Local Zones*", Web link: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html , Datum pristupa: 1.4.2023.

[13] "*What is Amazon VPC*", Web link: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html , Datum pristupa: 2.4.2023.

[14] "*AWS Elastic Kubernetes Service Documentation*", Web link: https://docs.aws.amazon.com/eks/index.html , Datum pristupa: 13.3.2023.

[15] "*AWS Command Line Interface Documentation*", Web link: https://docs.aws.amazon.com/cli/index.html , Datum pristupa: 14.3.2023.

[16] "*What is Infrastructure as Code (IaC)?*", Web link: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac , Datum pristupa: 16.3.2023.

[17] "*Terraform dokumentacija*", Web link: https://developer.hashicorp.com/terraform/docs , Datum pristupa: 17.3.2023.

[18] "*The Core Terraform Workflow*", Web link: https://developer.hashicorp.com/terraform/intro/core-workflow , Datum pristupa: 17.3.2023.

[19] "*Provision an EKS Cluster (AWS)*", Web link: https://developer.hashicorp.com/terraform/tutorials/kubernetes/eks , Datum pristupa: 20.3.2023.

[20] "What is SDLC (Software Development Lifecycle)?", Web link: https://aws.amazon.com/what-is/sdlc/#:~:text=The%20software%20development%20lifecycle%20(SDLC,expectations%20during%20production%20and%20beyond. , Datum pristupa: 18.4.2023.

[21] J. Bartlett, An Overview of the Software Development Life Cycle (SDLC), Web link: https://blog.testlodge.com/software-development-life-cycle/ , 9.1.2023.

[22] "*CI/CD and Agile: Why CI/CD Promotes True Agile Development*", Web link: https://codefresh.io/learn/ci-cd-pipelines/ci-cd-and-agile-why-ci-cd-promotes-true-agile-development/#:~:text=Enabling%20Agile%20with%20CI%2FCD,deployment%20and%20software%20release%20processes. , Datum pristupa: 21.4.2023.

[23] "*Manifesto for Agile Software Development*", Web link: https://agilemanifesto.org/ , Datum pristupa: 21.4.2023.

[24] J. Adam, What is Agile software development?, Web link: https://kruschecompany.com/agile-software-development/ , Datum pristupa: 21.4.2023.

[25] "*What is CI/CD?*" by GitLab, Web link: https://about.gitlab.com/topics/ci-cd/ , Datum pristupa: 28.4.2023.

[26] "*GitHub Actions dokumentacija*", Web link: https://docs.github.com/en/actions , Datum pristupa: 29.4.2023.

[27] "*Guide to GitOps*" by Weaveworks, Web link: https://www.weave.works/technologies/gitops/ , Datum pristupa: 30.4.2023.

[28] "*Argo CD dokumentacija*", Web link: https://argo-cd.readthedocs.io/en/stable/ , Datum pristupa: 2.5.2023.

Skraćenice

API	Application Programming Interface
CNCF	Cloud Native Computing Foundation
PVC	Persistent Volume Claim
PV	Persistent Volume
URL	Uniform Resource Locator
IP	Internet Protocol
DNS	Domain Name System
TLS	Transport Layer Security
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
EKS	Elastic Kubernetes Service
IT	Information Technology
AWS	Amazon Web Services
EC2	Amazon Elastic Compute
S3	Simple Storage Service
RDS	Relational Database Service
VPC	Virtual Private Cloud
AZ	Availability zone
NACL	Network Access Control List
IGW	Internet Gateway
ECS	Elastic Container Service
ECR	Elastic Container Registry
CLI	Command Line
UI	User Interface
IaC	Infrastructure as a Code
HCL	HashiCorp Configuration Language
HTTP	The Hypertext Transfer Protocol
SDLC	Software Development Life Cycle
CI	Continuous Integration
CD	Continuous Delivery
CRD	Custom Resource Definition

Sažetak

Upravljani Kubernetes u oblaku ubrzava instalaciju te olakšava održavanje, skalabilnost i raspoloživost aplikacija. Infrastruktura kao kod daje dosljednost i jednostavnost postavljanja resursa na oblak. Helm omogućava upravljanje paketima Kubernetes manifesta te također omogućava dosljednost i jednostavnost implementacije. Životni ciklus razvoja programa je proces dizajniranja, razvoja, testiranja i implementacije programa. Kontinuiranom integracijom i kontinuiranom isporukom ubrzavaju se i osiguravaju procesi razvoja programa. U ovom radu implementiran je cjevovod kontinuirane integracije i kontinuirane isporuke u alatima GitHub Actions i Argo CD, korištenjem platforme GitHub za pohranu koda. Jednostavna Flask aplikacija implementirana je na upravljanom AWS Kubernetesu uz pomoć Helm charta. Resursi na oblaku su postavljeni s alatom Terraform koji služi za infrastrukturu kao kod.

Ključne riječi: Kubernetes, oblak, cjevovod, infrastruktura kao kod, automatizacija, kontinuirana integracija, kontinuirana isporuka, skalabilnost, raspoloživost

Summary

Managed Kubernetes in the cloud speeds up installation and simplifies the maintenance, scalability, and availability of applications. Infrastructure as code provides consistency and ease of deploying resources to the cloud. Helm enables management of Kubernetes manifest packages, providing consistency and simplicity in implementation. The software development lifecycle involves the processes of design, development, testing, and implementation of programs. Continuous integration and continuous delivery accelerate and ensure efficient program development processes. In this master thesis, a pipeline for continuous integration and continuous delivery is implemented using GitHub Actions and Argo CD tools, utilizing the GitHub platform for code storage. A simple Flask application is deployed on managed AWS Kubernetes using a Helm chart. Cloud resources are provisioned using the Terraform tool, which serves as infrastructure as code.

Key words: Kubernetes, cloud, pipeline, infrastructure as a code, automatization, continuous integration, continuous delivery, scalability, availability