SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2124

SKALABILNA ARHITEKTURA MIKROUSLUGA ZASNOVANA NA PLATFORMI DOCKER S PRIMJENOM NA PROBLEME STROJNOG UČENJA

Lucia Penić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2124

SKALABILNA ARHITEKTURA MIKROUSLUGA ZASNOVANA NA PLATFORMI DOCKER S PRIMJENOM NA PROBLEME STROJNOG UČENJA

Lucia Penić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

DIPLOMSKI ZADATAK br. 2124

Pristupnica: **Lucia Penić (0036483011)**

Studij: Računarstvo

Profil: Računarska znanost

Mentor: izv. prof. dr. sc. Alan Jović

Zadatak: **Skalabilna arhitektura mikrousluga zasnovana na platformi Docker s primjenom na probleme strojnog učenja**

Opis zadatka:

Platforma Docker koristi se za olakšani razvoj, raspoređivanje i pokretanje aplikacija u različitim radnim okolinama. Upravljanje Dockerom nije intuitivno i potrebno je podrobno razumijevanje arhitekture kako bi se uspješno uspostavilo funkcionalnu raspodijeljenu aplikaciju. U ovom diplomskom radu razmatrat će se primjena takve raspodijeljene arhitekture na probleme strojnog učenja te prednosti orkestracije Dockerovih procesa. U tu svrhu, potrebno je razviti web aplikaciju temeljenu na mikrouslugama koja će se sastojati od sučelja za unos podataka modela, međuspremnika kao reda za obradu podataka te od orkestriranih Dockerovih procesa koji uzimaju podatke kao poruke iz reda za obradu te ih predobrađuju. Skalabilni algoritam strojnog učenja treba preuzimati predobrađene podatke iz procesa i graditi model podataka u paraleli. Za tu potrebu u radu se treba iskoristiti algoritam strojnog učenja koji omogućuje jednostavnu paralelizaciju (npr. slučajna šuma). Rezultat će biti proces koji ispisuje rezultate modeliranja u odgovarajućem obliku i prikazuje ih korisniku kroz sučelje web aplikacije. Za orkestraciju Dockerovih procesa koristit će se jedno besplatno dostupno rješenje po izboru (npr. Swarm, Kubernetes). Funkcioniranje sustava vrednovat će se na jednom većem slobodno dostupnom skupu podataka po izboru, uz prikaz usporedbe poboljšanja koje ovakav sustav ima u odnosu na neorkestriranu monolitnu izgradnju modela strojnog učenja.

Rok za predaju rada: 30. lipnja 2020.

SADRŽAJ

1. Uvod **1**

2. Srodni radovi **2**

3. Primjena kontejnerske tehnologije na probleme strojnog učenja **3**

3.1. Mikrouslužna arhitektura 3
3.2. Kontejnerske tehnoglogije 4
3.3. Pristupi u ostvarenju cjevovoda strojnog učenja 4
3.3.1. Faza pripreme podataka 6
3.3.2. Faza učenja modela 6
3.3.3. Faza isporuke modela 7
3.4. Prednosti i mane 7

4. Docker **10**

4.1. Arhitektura sustava Docker 10
4.2. Docker Engine 11
4.3. Dockerovi objekti 12
4.3.1. Dockerova slika 12
4.3.2. Dockerov kontejner 13
4.4. Dockerov registar 14
4.5. Dockerfile 14

5. Orkestracija sustava **19**

5.1. Glavne značajke platforme Kubernetes 19
5.2. Arhitektura sustava Kubernetes 20
5.2.1. Master čvor 20
5.2.2. Ostali čvorovi 21
5.3. Osnovni objekti 22
5.4. Nadzor metrika u sustavu Kubernetes 23

6. Implementacija i vrednovanje programskog rješenja **25**
6.1. Analiza sentimenta 25
6.2. Arhitektura sustava 26
6.3. Kontejnerizacija mikrousluga 26
6.4. Postavljanje Kubernetesovog grozda računala 27
6.4.1. Postavljanje čvora master 28
6.4.2. Postavljanje radnih čvorova 29
6.4.3. Postavljanje mreže 29
6.4.4. Postavljanje privatnog Dockerovog registra 30
6.5. Isporuka mikrousluga na Kubernetes 31
6.6. Vrednovanje programskog rješenja 34
6.6.1. Priprema modela 34
6.6.2. Horizontal Pod Autoscaler 35
6.6.3. Test opterećenja 36

7. Zaključak **40**

Literatura **41**

1. Uvod

Cilj ovog rada je istražiti i pokazati prednosti i mane korištenja raspodijeljene arhitekture u razvoju i isporuci modela strojnog učenja. Nadalje, u ovom radu razmotrit će se mogućnosti skaliranja takvog raspodijeljenog cjevovoda i posljedice koje takav pristup uzrokuje.

Nakon uvoda, slijedi drugo poglavlje rada u kojem je navedena nekolicina srodnih radova te kratki osvrt na njih.

Treće poglavlje je kratak pregled teme razvoja cjevovoda za modele strojnog učenja. Razmatraju se različiti pristupi u ostvarenju cjevovoda tijekom razvoja samog modela te se razmatra isporuka gotovog proizvoda korisniku.

U četvrtom poglavlju dan je osvrt na temu mikrouslužne arhitekture te primjena kontejnerske tehnologije u takvom sustavu.

Peto je poglavlje nastavak na četvrto, a detaljnije opisuje arhitekturu i način korištenja trenutno najprihvaćenije tehnologije za kontejnerizaciju, platforme Docker.

Šesto poglavlje govori o važnosti orkestracije u mikrouslužnim sustavima te prednostima korištenja platforme Kubernetes u tu svrhu. Osim toga, u ovom poglavlju dotaknut ću se teme nadzora sustava i skaliranja pojedinih dijelova sustava.

U sedmom poglavlju opisano je programsko rješenje ostvareno u sklopu ovog rada.

Konačno, posljednje poglavlje je zaključak na temu.

2. Srodni radovi

U programskom inženjerstvu mikrouslužni način organizacije sustava, kontejnerizacija mikrousluga i orkestracija već su dobro istraženi i široko zastupljeni pojmovi. Međutim, primjena takvih sustava na strojno učenje još je uvijek u začecima i postoji tek nekolicina radova koji razmatraju ovakav način organizacije sustava.

Jedan takav rad s detaljnim uputama za izradu i isporuku modela strojnog učenja koristeći sustav Kubernetes dostupan je na osobnim web stranicama autora Caspera Hansena [12]. Na primjeru algoritma slučajne šume i jednostavnog skupa podataka izgrađen je model za predviđanje potrošnje goriva automobila. Za isporuku modela korisniku korištena je web usluga Flask unutar Dockerovog kontejnera i sustav Kubernetes. Tako isporučen model postaje dostupan za korištenje putem zahtjeva HTTP POST.

Na stranicama tvrtke Rubik's Code može se pronaći serija blogova autora Nikole M. Živkovića koja se bavi kontejnerizacijom neuronske mreže u Dockerove kontejnere [22]. U tri članka, autor na primjeru skupa podataka *iris* detaljno prolazi kroz korake učenja modela, njegovog spremanja, potom i učitavanje u Flask web uslugu te njeno pakiranje u Dockerove kontejnere i isporuku u kombinaciji s tehnologijama TenserFlow i gRPC .

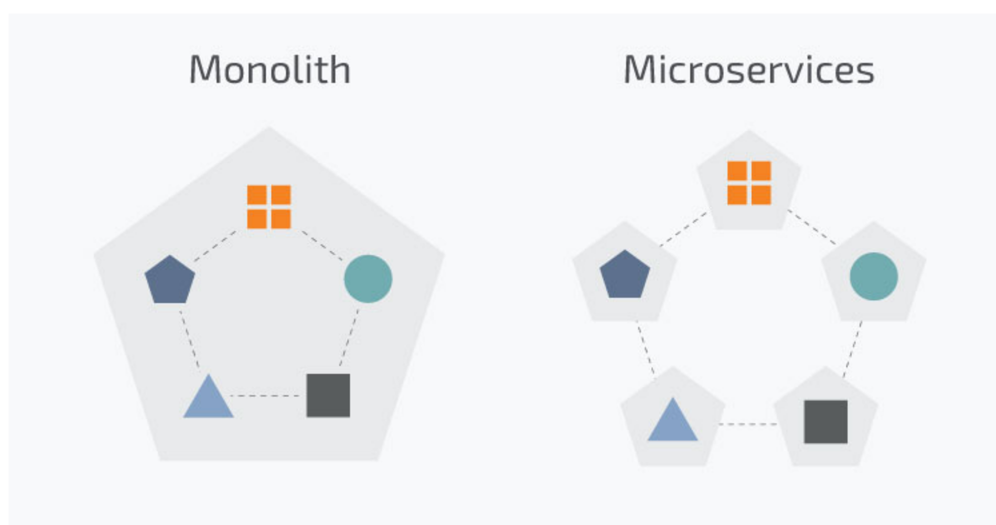
Sličan članak objavljen je i na web stranicama *Towards Data Science*. Autor članka objašnjava kako na kako na platformi Google Cloud koristeći alat Terraform podignuti Kubernetes grozd računala i na njemu isporučiti model za prepoznavanje i klasifikaciju slika, a koji koristi duboku konvolucijsku neuronska mrežu [19].

Iako se ne bavi strojnim učenjem, vrijedi spomenuti i diplomski rad kolegice Paule Kokić [16] u kojem se prednosti raspodijeljene arhitekture, kontejnerizacije i orkestraciji ispituju na primjeru NodeJS aplikacije. Rad prikazuje koncepte skalabilnosti i raspoloživosti koristeći Kubernetes.

3. Primjena kontejnerske tehnologije na probleme strojnog učenja

3.1. Mikrouslužna arhitektura

Prilikom organizacije nekog sustava, monolitna arhitektura (engl. *monolithic architecture*) dugo je vremena bila prvi izbor za računalne inženjere. Međutim, razvojem internetskih tehnologija, takvi sustavi postajali su sve složeniji i teži za održavanje. Tada se javlja ideja da se logika aplikacije odvoji na više manjih, nezavisnih usluga od kojih svaka oblikuje dio poslovne logike sustava [14]. Ovakav način organizacije programskog koda nazivamo mikrouslugama (engl. *microservices*). Nastali su kao podvrsta uslužno orijentiranog arhitekturalnog stla (engl. *service-oriented architecture, SOA*).



Slika 3.1: Razlika između monolitne i mikrouslužne arhitekture [10]

Glavna ideja mikrouslužne arhitekture jest oblikovati decentralizirani sustav koji se sastoji od niza komponenti koje prate princip jedne odgovornosti (engl. *single respon-*

sibility principle). Svaka komponenta je samostalna i neovisna, a s ostalim komponentama u sustavu komunicira putem poruka. Nadalje, mikrousluge se oblikuju tako da skrivaju unutarnju logiku dok prema van oblikuju REST (engl. *Representational state transfer*) sučelje. Ovo inženjerima pruža mogućnost podjele odgovornosti na način da je svaki tim odgovoran za svoje mikrousluge te ih može razvijati, isporučivati i održavati neovisno o ostalima [21]. Osim toga, dolazi do eliminacije jedne točke otkaza, lakše se otkriva kvar ili žarišna točka uskog grla. S druge strane, orkestracija i nadzor postaju znatno teži, mora se osigurati uobičajen način komunikacije između mikrousluga i samo testiranje cijelog sustava je zahtjevnije [16].

3.2. Kontejnerske tehnoglogije

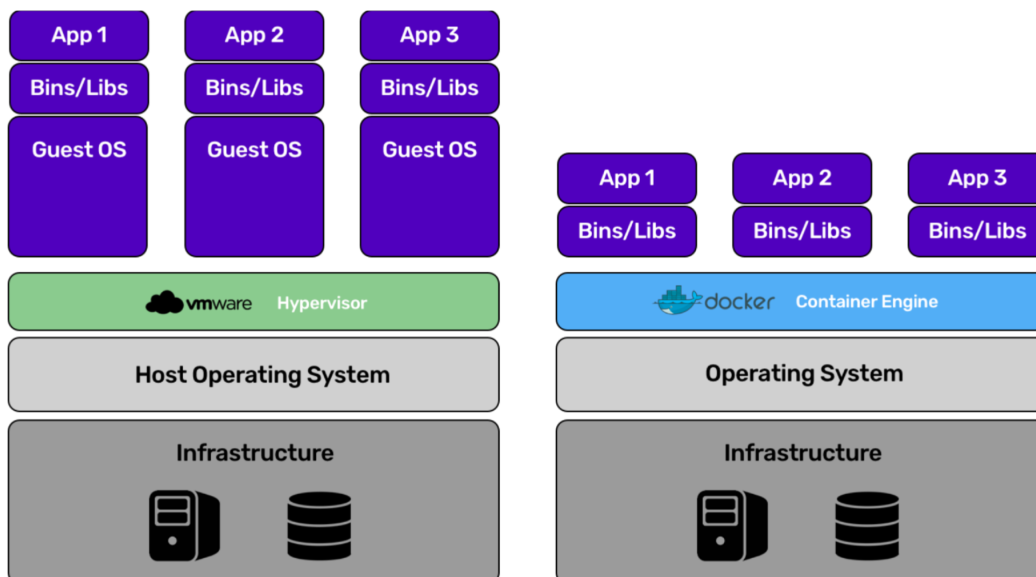
Pojam koji se usko povezuje s mikrouslugama je kontejnerska tehnologija (engl. *container technology*). Kao nova alternativa virtualnim strojevima, kontejneri pružaju izoliranu, lako prenosivu okolinu za razvoj i isporuku programske podrške.

Slika 3.2 prikazuje osnovnu razliku kontejnera u odnosu na virtualne strojeve. Dok virtualni stroj unutar jednog domaćina omogućava podizanje višestrukih operacijskih sustava, kontejneri se oslanjaju na postojeći operacijski sustav i unutar njega pružaju izoliranu okolinu [2]. Zbog toga puno bolje iskorištavaju resurse domaćina na kojem se nalaze što im daje glavnu prednost nad virtualnim strojevima: lagani su u smislu zauzeća kako fizičke tako i radne memorije (engl. *lightweight*).

Konceptualna ideja za korištenje kontejnera u svrhu izolacije javila se još 1979. u sklopu UNIX operacijskog sustava kao mogućnost dodjeljivanja prava pristupa samo jednom procesu [11]. Ideja je brzo prihvaćena i danas postoje mnoge implementacije različitih proizvođača. U praksi je ipak jedna od njih, platforma Docker, prihvaćenija i raširenija od drugih. Glavni mehanizmi rada sustava Docker i način korištenja s ciljem ostvarenja kontejnerizacije mikrousluga opisani su u 4. poglavlju ovog rada.

3.3. Pristupi u ostvarenju cjevovoda strojnog učenja

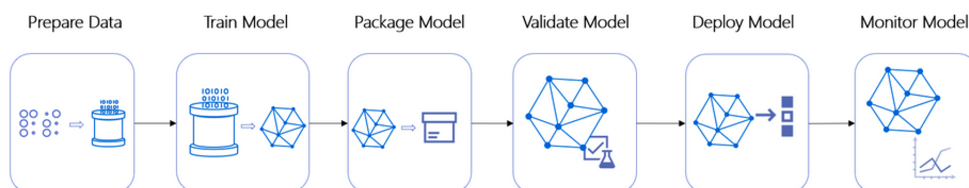
Zahvaljujući napretku znanosti, optimizaciji samih algoritama, olakšanom prikupljanju velikih količina podataka, poboljšanju hardvera te novim pristupima u ostvarenju



Slika 3.2: Razlika između virtualnih strojeva i kontejnera [1]

arhitekture, učenje i isporuka modela strojnog učenja postaje znatno pristupačnija. Danas su takvi modeli neizostavni dio brojnih proizvoda, od web aplikacija do kućanskih aparata. S rastom popularnosti i novih načina primjene, rastu i zahtjevi za isporukom takvih sustava.

Klasične faze u razvoju modela strojnog učenja još se nazivaju i cjevovodom (engl. *machine leaning pipeline*). Razlikujemo faze pripreme podataka, učenja modela, njegovog spremanja i vrednovanja te naposljetku isporuke i nadzora (slika 3.3). One su tijekom godina znatno olakšane koristeći virtualna okruženja dostupna kroz web platforme kao što su Hadoop ili Apache Spark. Razlog popularnosti ovih rješenja je obrada unutar grozda računala (engl. *cluster*) i korištenje paralelizacije kako bi se ubrzali pojedini dijelovi cjevovoda [7].



Slika 3.3: Tipična struktura cjevovoda za razvoj modela strojnog učenja [1]

S porastom popularnosti programske podrške u oblaku, javlja se ideja da se prednosti takvih rješenja kao što su skalabilnost, fleksibilnost u izboru tehnologija i mogućnost jednostavnog repliciranja iskoriste u razvoju i isporuci modela strojnog učenja. U

kombinaciji s tim, raste popularnost i korištenja kontejnerske tehnologije u svrhu brze automatizacije i enkapsulacije okruženja.

Možemo izdvojiti tri faze koje posebno profitiraju kod primjene kontejnerske tehnologije. To su faze: pripreme podataka, učenja modela i isporuke [8].

3.3.1. Faza pripreme podataka

Tijekom pripreme podataka, znanstvenici pretvaraju skup podataka u značajke iz kojih će model učiti. U tom procesu često mijenjaju skup podataka, pročišćavaju ga od stršećih podataka i nedostajućih vrijednosti, isprobavaju utjecaj i međuovisnost pojedinih značajki i slično. U tom procesu često trebaju mogućnost repliciranja radnog okruženja u izoliranu okolinu koju mogu pokretati neograničen broj puta s različitim ulaznim podacima. Osim toga vrlo je lako ubrzati proces pokretanjem više kontejnera u paraleli. Za tu namjenu prirodno se nameće korištenje kontejnera.

3.3.2. Faza učenja modela

Faza učenja modela mnogo je zahtjevnija i zanimljivija za razmatrati u kontekstu mikrousluga i orkestracije. Postoji nekoliko pristupa za primjenu kontejnera.

Prvi se odnosi na ugađanje hiperparametara modela. Dok se neki parametri mogu izračunati matematičkim funkcijama i tako pronaći oni koji najbolje odgovaraju danom problemu, hiperparametri se mogu pronaći samo iscrpnim pretraživanjem različitih vrijednosti (engl. *grid search*). U tom kontekstu, kontejneri se mogu koristiti za pakiranje algoritma strojnog učenja sa svim zavisnostima (engl. *dependencies*). Ovako spremljen model možemo promatrati kao crnu kutiju, koja na ulazu uzima odabrane parametre i na izlazu daje statistiku modela npr. točnost, preciznost, odziv, mjeru F1, i tako dalje.

Nadalje, kontejneri olakšavaju brzo skaliranje opterećenja prema gore ili dolje kroz više čvorova. U tom načinu veliku ulogu igra orkestracija. Moguće je nadziranje metrika sustava te prilagodba stanju sustava tako da se na temelju dostupnih resursa odabire računalo s optimalnim karakteristikama. Osim toga, ovo pojednostavljuje isprobavanje različitog hardvera, odnosno potragu za zadovoljavajućim omjerom između

cijene sklopovlja i kvalitete modela dobivene temeljem evaluacijskih mjera.

Samo učenje modela odnosi se na unos podataka za učenje u funkciju modela i rješavanje optimizacijskog problema. Veći broj primjera za učenje rezultira i preciznijim modelom. Međutim, često je obrada nad velikim skupom izuzetno sporo i zahtjeva mnogo računalnih resursa. To osjetno usporava razvoj, testiranje i isporuku modela, a i uzrokuje povećanje cijene razvoja. Izgradnja modela u paraleli mogla bi riješiti navedene probleme, ali to podrazumijeva i algoritam prilagođen takvom načinu rada.

3.3.3. Faza isporuke modela

Isporuka modela korisniku jedna je od najvažnijih stavki u procesu. Kontejneri omogućavaju da se model isporuči korisniku kao mikrousluga u sklopu većeg sustava. Osim toga, osiguravaju jednostavno prenošenje između testnog i produkcijskog okruženja bez straha da će se dogoditi različite verzije paketa zbog kojih često nastaju kolizije.

Pri izgradnji produkcijskog sustava poželjno je ostvariti sljedeće ciljeve [15]:

- smanjiti latenciju
- ostvariti integriran sustav, sa slabo povezanim dijelovima sustava
- moguće horizontalno i vertikalno skaliranje
- omogućiti asinkronu komunikaciju preko poruka
- podržati odradu u stvarnom vremenu (engl. *real time processing*) i grupnu obradu (engl. *batch processing*)
- postoji upravljanje greškama, sustav je otporan na greške

Gornji ciljevi lako se ostvaruju korištenjem platforme Docker za kontejnerizaciju i sustava Kubernetes za orkestraciju. U praktičnom dijelu ovog rada dan je primjer primjene tih tehnologija u fazi isporuke modela.

3.4. Prednosti i mane

U ovom poglavlju nastojat ću sumirati već spomenute prednosti i poteškoće na koje nailazimo pri kontejnerizaciji i orkestraciji modela strojnog učenja. Neke od prednosti

su:

- izolacija – modeli se mogu zapakirati sa svim zavisnostima i koristiti kao crna kutija
- prenosivost – jednostavno prenošenje između testnog i produkcijskog okruženja
- replikacija – konfiguracija kontejnera piše se u obliku predložaka na temelju kojih se može pokrenuti neograničen broj primjeraka procesa što olakšava dijeljenje s ostalim inženjerima putem registara
- brzina – u usporedbi s virtualnim strojevima, podizanje kontejnera je mnogo brže jer nema potrebe za cijelim operacijskim sustavom
- bolje iskorištenje resursa – kontejneri u izolacijsku okolinu pakiraju sve i samo ono što je potrebno kako bi se program mogao izvršavati čime se postiže optimizacija korištenja računalnih resursa
- dostupnost resursa za učenje – zahvaljujući registrima, postoji mnoštvo dostupnih predložaka koje korisnici mogu koristiti za brz početak
- ponovno korištenje – jedna od temeljnih smjernica za razvoj ovakvog sustava je modularnost i apstrakcija komponenti, koje se onda mogu lako zamijeniti unutar sustava ili ponovo iskoristiti u nekom drugom
- paralelizacija – koristeći horizontalno skaliranje, lako se ubrzavaju procesi potrebni za izgradnju i korištenje modela
- fleksibilnost u izboru tehnologija – u jednom sustavu korisnik može uvijek izabrati alat koji je najprikladniji za određeni problem, bez straha da će doći do konflikta s ostalim mikrouslugama

S druge strane, uvođenje mikrouslužne arhitekture dovodi do povećanja složenosti sustava. Umjesto jedne masivne aplikacije, sada imamo veliki broj dinamičnih mikrousluga pisanih u različitim tehnologijama. To može izazvati neke od poteškoća kao što su [16]:

- složena orkestracija – svaka mikrousluga piše se neovisno o ostalima i ima vlastiti životni ciklus zbog čega se često mijenja stanje sustava
- otežan nadzor mikrousluga – dosadašnji alati korišteni za nadzor i uzbunu (engl. *alerting*) više nisu dovoljni i moramo pronaći nove načine za praćenje cijelog sustava

- prilagodba algoritama – u slučaju da želimo koristiti navedene tehnologije u fazi izgradnje modela, algoritmi se moraju prilagoditi da podržavaju paralelizaciju
- prilagodba komunikacije – mora se osigurati dogovoren način komunikacije unutar sustava zbog različitosti tehnologija
- otkrivanje pogreške – u slučaju kvara, znatno je teže pronaći u kojoj mikrousluzi ili postavkama sustava je nastao problem
- održavanje visoke raspoloživosti – sustav mora implementirati mogućnost brzog otkrivanja i oporavka od pogreške

4. Docker

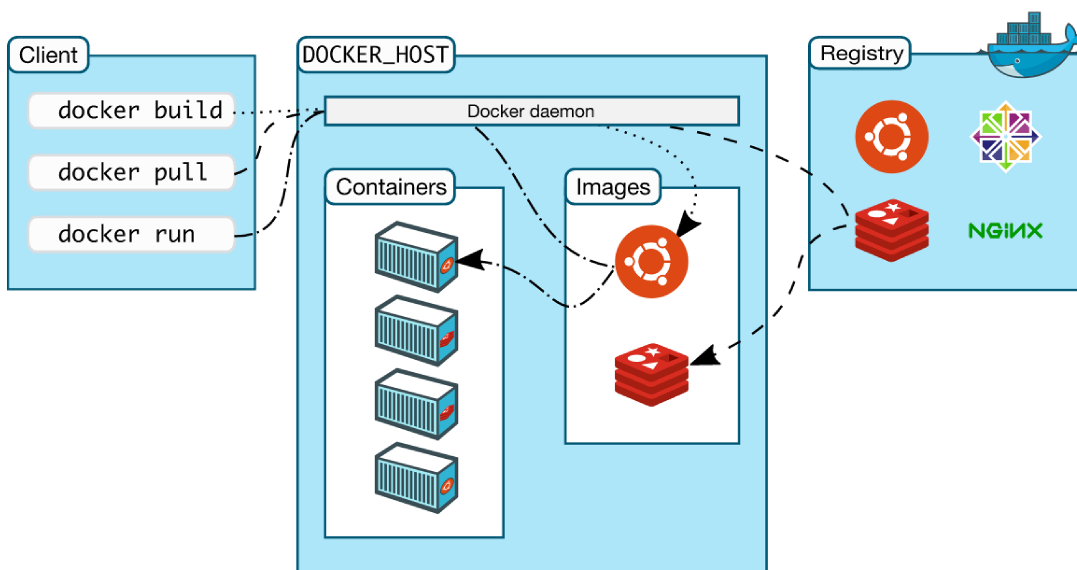
Platforma Docker je sustav otvorenog koda koji pruža niz alata za razvoj, isporuku i pokretanje korisničkih programa [13]. Ovim alatom korištenje kontejnera postalo je znatno lakše jer omogućava razvoj, implementaciju i pokretanje aplikacija unutar izoliranih paketa. Jednom konfiguriran paket, odnosno Dockerova slika (engl. *Docker image*), sadrži sve što je potrebno za pokretanje željene mikrousluge. To uključuje sve programske knjižice (engl. *libraries*), zavisnosti (engl. *dependencies*), konfiguraciju okoline i slično. Ovo omogućava laku skalabilnost, prijenos na druge poslužitelje i nezavisnost od domaćina na kojem se mikrousluga izvodi.

4.1. Arhitektura sustava Docker

Arhitekturu sustava Docker čine četiri glavne komponente:

- Dockerov klijent
- Dockerov domaćin (engl. *Docker host*)
- Dockerov registar (engl. *Docker registry*)
- Dockerovi objekti

Slika 4.1. ilustrativni je prikaz načina na koji su ove komponente povezane u sustav Docker. Dockerov klijent komunicira s Dockerovim *deamonom* koji pak upravlja Dockerovim objektima. Klijent i *deamon* mogu biti na istom ili na različitim domaćinima, te jedan klijent može potencijalno komunicirati s više *daemona*. *Daemon* još mora moći komunicirati s nekim od Dockerovih registara, kako bi mogao preuzeti osnovne slike (engl. *base images*). Pojedine komponente detaljnije su objašnjene niže u tekstu.

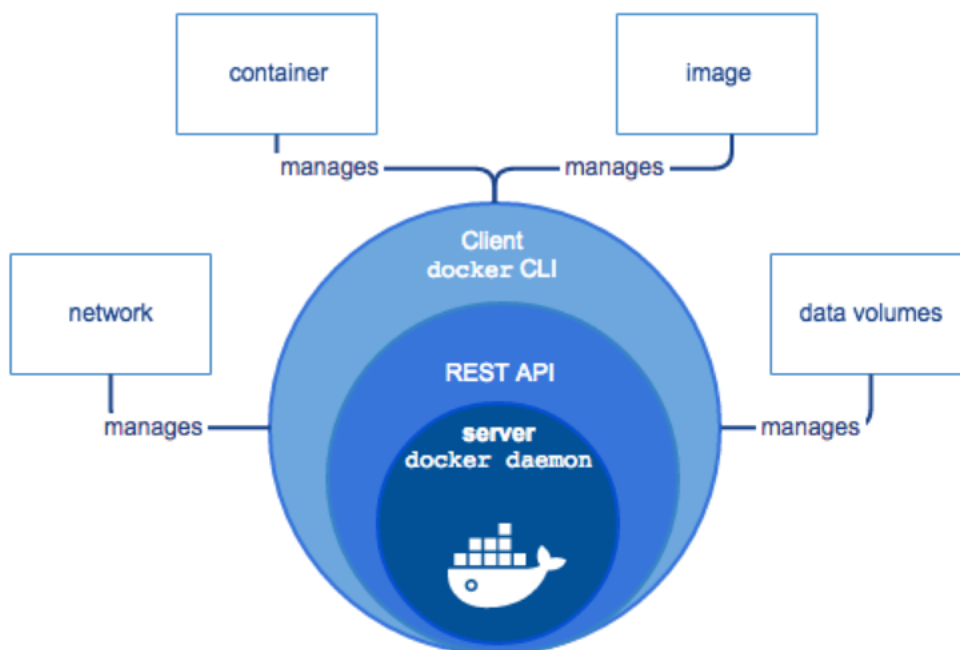


Slika 4.1: Povezanost komponenti sustava Docker [13]

4.2. Docker Engine

Osnovicu sustava čini Docker *Engine*, aplikacija koja povezuje klijenta i poslužitelj, odnosno klijentsko komandno sučelje i domaćina na kojem se izvodi Dockerov *deamon*. Između njih nalazi se aplikacijsko sučelje. Grafički prikaz komponenti dan je na slici 4.2.

- Dockerov poslužitelj je tip dugotrajnog procesa, odnosno *daemon process*. Vrti se u pozadini, a njegova je uloga upravljati Dockerovim objektima na zahtjev klijenta.
- aplikacijsko sučelje (engl. *Representational State Transfer Application Programming Interface,* REST API) stoji između korisnika, odnosno komandnog sučelja i *deamona*. Omogućava komunikaciju između klijenta i *deamona* preko UNIX utičnica (engl. *sockets*) ili mrežnog sučelja (engl. *network interface*).
- Dockerovo komandno sučelje (engl. *Command Line Interface*, CLI) u obliku *docker* naredbe omogućava korisniku da upravlja *deamonom*. Korisnik koristi neku od Dockerovih naredbi preko komandne ljuske ili unutar skripte, a upute za izvršavanje prenose se preko REST API-ja do *deamona* koji naposljetku izvrši iste.



Slika 4.2: Vizualizacija komponenti Docker Engina [13]

4.3. Dockerovi objekti

Kao što je spomenuto u poglavlju 4.1.1 Dockerov *deamon* upravlja Dockerovim objektima. U Dockerovom okruženju razlikujemo nekoliko vrsta entiteta, odnosno Dockerovih objekata. Neki od njih su: slike, kontejneri, volumeni, mreže, usluge, i tako dalje.

4.3.1. Dockerova slika

Dockerova slika je predložak namijenjen isključivo za čitanje (engl. *read only*), a koji sadrži naredbe za stvaranje Dockerovih kontejnera [3]. Svaki korisnik može koristiti neku od postojećih slika ili stvoriti vlastitu sliku što postiže pisanjem posebne konfiguracijske datoteke zvane Dockerfile. Korisnik sliku može napisati iz temelja (engl. *from scratch*, no češće se koristi neka od osnovnih slika (engl. *base images*) koja se dalje nadograđuje kako bi se dobila slika prilagođena korisniku. Osnovne slike, ali i slike drugih korisnika, mogu se pronaći i povući (engl. *pull*) s Docker Hub registra slika. Detaljnije o registrima objašnjeno je u poglavlju 4.4. Naredbe se pišu u sljedećem

obliku:

```
docker image <naredba>
```

U ovom radu, popisane su i kratko objašnjene samo neke naredbe, a potpuni popis može se pronaći u službenoj dokumentaciji [4].

Tablica 4.1: Često korištene naredbe za Dockerove slike

NAREDBA	OPIS
build	izgradnja slike iz Dockerfile datoteke
inspect	ispis svih slika na domaćinu
ls	zaustavljanje postojećeg kontejnera
pull	povlačenje slike s Docker Hub repozitorija
rm	uklanjanje slike

Dockerov *daemon* gradi sliku liniju po liniju, prateći strukturu Dockerfilea te iz svake od njih stvara jedan sloj (engl. *layer*) slike, što omogućava da, jednom kada želimo izmijeniti sliku, iznova gradimo samo one slojeve u kojima je došlo do promjene. Upravo ova optimizacija čini Dockerove slike lakšim, manjim i bržim u usporedbi s ostalim virtualizacijskim tehnologijama [13].

4.3.2. Dockerov kontejner

Dockerov kontejner je pokrenuta instanca slike. Prilikom pokretanja slike, predaju se argumenti kojima opisujemo uvjete izvođenja i bitne konfiguracijske opcije. Kontejneri se grade iz slike, ali sve naknadne izmjene unutar kontejnera i opcije korištene prilikom stvaranja gube se ukoliko se ne koristi trajna pohrana ili ako se iz trenutne verzije kontejnera ne spremi slika.

Korištenjem naredbi komandnog sučelja moguće je upravljanje kontejnerima: stvaranje, pokretanje, zaustavljanje, brisanje, spajanje u virtualnu mrežu, povezivanje s pohranom i slično. Kontejner pruža potpunu izolaciju od datotečnog sustava domaćina na kojem se izvodi, a s drugim kontejnerima može komunicirati preko IP adrese ili izričitim povezivanjem u mrežu.

Naredbe za upravljanje kontejnerima pišu se u sljedećem obliku:

```
docker container <command>
```

U ovom radu, popisane su i kratko objašnjene samo neke naredbe. Puni popis može se pronaći u službenoj dokumentaciji [4].

NAREDBA	OPIS
create	stvaranje novog kontejnera
start	pokretanje postojećeg kontejnera
stop	zaustavljanje postojećeg kontejnera
run	pokretanje naredbe unutar novog kontejnera
exec	pokretanje naredbe unutar postojećeg, pokrenutog kontejnera
rm	uklanjanje jednog ili više kontejnera

4.4. Dockerov registar

Dockerov registar je zapravo spremište Dockerovih slika. Docker Hub službeni je registar tvrtke Docker i javan je za sve korisnike. U njemu se nalazi mnoštvo osnovnih slika (engl. *base images*) i slika drugih korisnika koje se mogu preuzeti korištenjem *pull* naredbe ili se pak može dijeliti vlastite slike s drugim korisnicima naredbom *push* [13].

Docker Engine prema standardnim postavkama slike preuzima upravo iz Docker Huba, ali te postavke moguće je nadjačati i koristiti svoj privatni registar slika. Privatni registar stvara se povlačenjem Docker slike zvane *registry*, stvaranjem kontejnera iz nje te postavljanjem proizvoljnih sigurnosnih postavki.

4.5. Dockerfile

Dockerfile možemo zamisliti kao recept, odnosno to je datoteka koja sadrži niz instrukcija koje govore Dockerovom *deamonu* kako da stvori sliku. Piše se u formatu specifičnom za Docker te ima niz ključnih riječi od kojih su neke obavezne, a neke ne.

Dockerfile je potrebno postaviti u vršni direktorij projekta, iz kojeg on može izvući

kontekst za slanje Dockerovom *daemonu*. Preporučuje se izbjegavanje pisanja apsolutnih putova jer to znatno otežava prenošenje na druge poslužitelje.

Format Dockerfile datoteke:

```
# Comment
# directive=value
INSTRUCTION arguments
```

Razlikujemo dva tipa linija u Dockerfile datoteci, one koje počinju znakom # te one koje počinju s nekom od ključnih riječi, odnosno instrukcijom.

Linije koje počinju sa znakom # Docker tretira kao komentare i ignorira ih tijekom prevođenja, osim ako se nakon znaka ne nalazi ključna riječ *directive*. U tom slučaju, radi se o parserskoj direktivi koja je zapravo specijalni slučaj komentara.

Drugi tip linije sastoji se od same instrukcije i njenih argumenata. Instrukcije nisu osjetljive na razliku između velikih i malih slova (engl. *case-insensitive*), ali među korisnicima postoji uvriježeno mišljenje da se one pišu velikim slovima kako bi datoteka bila čitljivija.

Naredbom *docker image build*, gradimo novu sliku iz Dockerfile datoteke. Već je spomenuto da se slike unutar Docker sustava grade slojevito. Svaka linija datoteke predstavlja jedan sloj slike, koji ima svoj ID dobiven kriptiranjem sha256 algoritmom. Slika se gradi slijedno, prateći instrukcije od početka datoteke prema kraju. Ovo je bitno znati jer prilikom izgradnje (engl. *build*) Docker daemon koristi slojeve iz svoje privremene memorije kako bi ubrzao izgradnju i optimirao spremište. To znači da redoslijed pisanja linija u Dockerfileu mora biti pomno odabran kako bismo smanjili vrijeme izgradnje prilikom izmjena u npr. kodu aplikacije. Zato se na vrhu Dockerfile datoteke pišu stvari koje se rijetko mijenjaju (npr. instalacije paketa), a stvari koje imaju česte izmjene, kao što je kod aplikacije, pišu se što je niže moguće [13]. Slika 4.3 prikazuje ispis naredbe *build* u kojem je vidljivo korištenje privremene memorije ((engl. *using cache*)) i slojevita izgradnja slike liniju po liniju.

Od mnoštva instrukcija u nastavku su navedene i ukratko objašnjene samo neke.

```
Sending build context to Docker daemon  3.005MB
Step 1/7 : FROM python:3.6
 ---> 0db2e2bbf438
Step 2/7 : WORKDIR /model
 ---> Using cache
 ---> ad7f9dd50c1a
Step 3/7 : COPY requirements.txt /model/requirements.txt
 ---> Using cache
 ---> 844b3318a840
Step 4/7 : RUN pip install -r requirements.txt
 ---> Using cache
 ---> 91fbd8606f6f
Step 5/7 : ADD . /model
 ---> cfa68a9b4ed0
Step 6/7 : ENTRYPOINT [ "python" ]
 ---> Running in c1f72384cb28
Removing intermediate container c1f72384cb28
 ---> 15c11fc3806a
Step 7/7 : CMD [ "api.py" ]
 ---> Running in e24e3e807868
Removing intermediate container e24e3e807868
 ---> bd8603d6676c
Successfully built bd8603d6676c
Successfully tagged simple_flask_app:1.2
```

Slika 4.3: Ispis naredbe *docker image build* prikazuje slojevitu izgradnju slike

FROM

Mora postojati u svakom valjanom Dockerfileu te mora biti prva od instrukcija. Služi za specificiranje baznog sloja (engl. *base layer*) nad kojim će se graditi ostatak slike. Na Docker repozitoriju može se pronaći niz već pripremljenih i gotovih baznih slika, a moguće je koristiti i neku od vlastitih slika. Također, moguće je početi “od nule” navodeći kao argument FROM scratch.

RUN

Omogućava pokretanje naredbi ljuske unutar kontejnera. Ovo je korisno na primjer za instaliranje paketa koji bazna slika nema ili za raspakiravanje arhive. Povezivanje više naredbi u lanac moguće je znakovima "&&", a preporučuje se kako bi smanjili broj slojeva koje Docker stvori pri stvaranju slike.

LABEL

Omogućava postavljanje raznih metapodataka, npr. email adresu održavatelja sustava, opis sustava i slično.

EXPOSE

Služi isključivo kao obavijest za osobu koja koristi Dockerfile kako bi ta osoba znala koji port mora otvoriti na kontejneru prilikom pokretanja.

ENV

Koristi se za postavljanje varijabli okoline kao što su razni ključevi, šifre i slično. Velika prednost je ta što su varijable okoline podržane na svakom operacijskom sustavu.

COPY

Naredba služi za kopiranje datoteka ili direktorija s lokalnog računala (argument <src>) u datotečni sustav kontejnera (argument <dest>).

WORKDIR

Ovom naredbom moguća je promjena radnog direktorija unutar samog kontejnera. Kontejner ima datotečni sustav u kojem smo, po pretpostavljenoj vrijednosti, pozicionirani u korijenu (engl. *root*). *Workdir* naredba daje nam mogućnost da se pomaknemo u željeni direktorij.

ENTRYPOINT

Omogućava promjenu načina ponašanja kontejnera u kojem će se on ponašati kao izvršna datoteka. Svi argumenti predani prilikom pokretanja kontejnera naredbom *docker container run* dodat će se na naredbu zapisanu u instrukciji *entrypoint*. Unutar jednog Dockerfilea, samo zadnja napisana instrukcija ima učinak na izvođenje.

CMD

Unutar Dockerfilea u pravilu može postojati jedna ili više *cmd* naredbi, ali samo zadnja će imati utjecaja na kontejner. Služi za postavljanje zadanog načina pokretanja kontejnera. Drugim riječima, jednom kada se kontejner pokrene iz stvorene slike, pokrenut će se s naredbom danom *cmd* instrukcijom. Ovo ponašanje moguće je nadjačati koristeći *exec* tip naredbe prilikom pokretanja Dockerovog kontejnera. Moguće je koristiti *cmd* i kako bi se predalo argumente naredbi *entrypoint*.

5. Orkestracija sustava

Kao što je napisano u poglavlju 3, uvođenje mikrouslužne arhitekture ima brojne prednosti, ali više izoliranih jedinica znači i više posla oko isporuke, konfiguracije produkcijskog okruženja, nadzora cjelokupnog životnog ciklusa i koordinacije mikrousluga. U modernim sustavima, korisnici očekuju da je usluga uvijek dostupna, a uzmemo li u obzir i sve veću potrebu za neprekidnim razvojem i isporukom (engl. *continuous delivery*), inženjeri moraju na dnevnoj bazi voditi računa o desecima ili stotinama mikrousluga [3]. Iz tog razloga javlja se potreba za centraliziranim načinom upravljanja mikrouslugama. Na tržištu postoji više dostupnih orkestracijskih alata namjenjenih tome, a neki od njih su: Docker Swarm, Google Kubernetes, Mesosphere Mesos, Amazon ECS i Hashicorp Nomad.

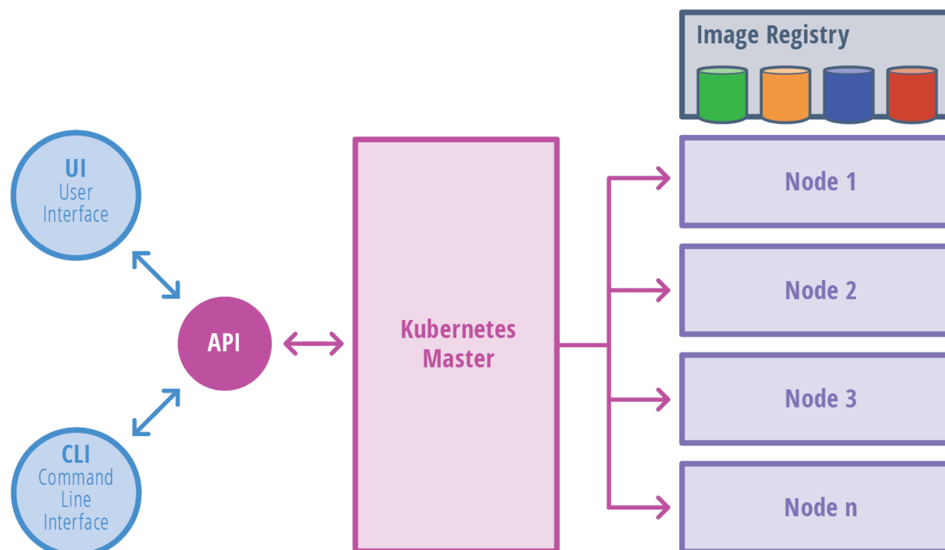
5.1. Glavne značajke platforme Kubernetes

Kubernetes, koji se ponekad naziva i K8s, sustav je otvorenog koda koji omogućava korisnicima automatiziranu isporuku, skaliranje i upravljanje kontejniziranih aplikacija [6]. Između ostalog, platforma omogućа korisnicima grupiranje mikrousluga u logičke jedinice, pruža mogućnost izbora okruženja lokalno (engl. *on-premises*), u oblaku (engl. *cloud*) ili pak hibridni te praktičan način prebacivanja između odabranih. Korisnik može vrlo jednostavno skalirati pojedine dijelove sustava horizontalno ili vertikalno, prebacivati promet između verzija sustava, raditi ažuriranje sustava [9] i još mnogo toga.

5.2. Arhitektura sustava Kubernetes

Suistav Kubernetes primarno je namijenjen za koordinaciju u velikim, složenim sustavima koji koriste više od jednog virtualnog ili fizičkog računala. Računala su organizirana u grozd (engl. *cluster*) povezan mrežom. Po ulogama u sustavu razlikujemo jedan ili više glavnih čvorova (engl. *master nodes*) od ostalih, jednostavno nazvanih čvorovi (engl. *nodes*). Slikoviti prikaz organizacije grozda računala vidljiv je na slici 5.1.

Korisnik s grozdom komunicira koristeći Kubernetes API, a okruženje konfigurira pišući deklarativni plan u obliku JSON ili YAML datoteke [9]. U datotekama korisnik opisuje željeno ponašanje i stanje sustava, ali ne i način na koji će se promjene odviti. Za dovođenje sustava u željeno stanje zaslužan je niz unutarnjih mehanizama jednim imenom nazvan kontrolna ravnina (engl. *control plane*). Kada korisnik zatraži izmjenu u sustavu, na primjer stvaranje novog objekta, kontrolna ravnina zapiše promjenu u dnevnik izmjena i počne izvoditi zadane naredbe pokrećući potrebne mehanizme i dodjeljujući ih čvorovima koji ih u konačnici izvršavaju.

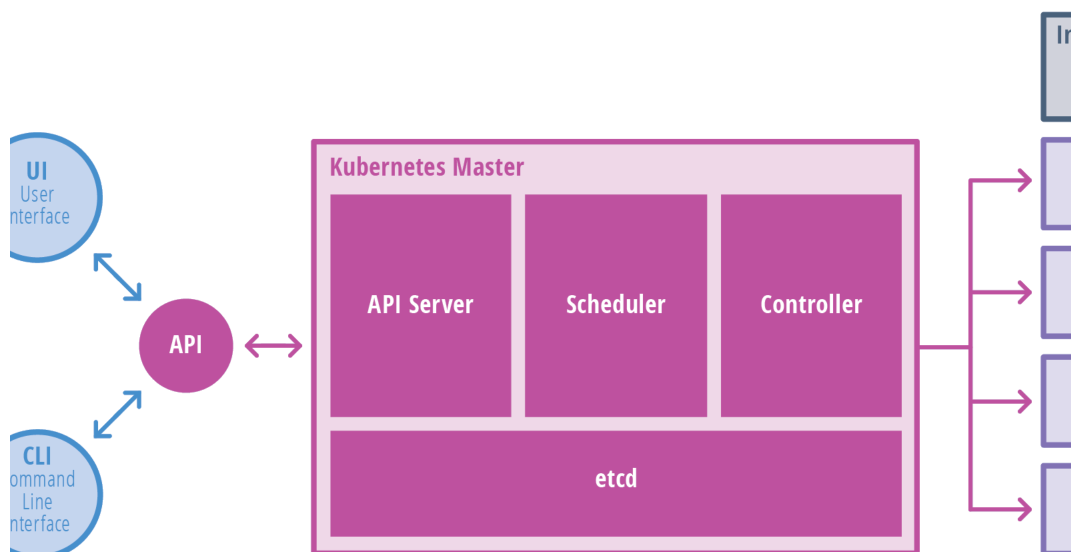


Slika 5.1: Pogled na organizaciju grozda računala u sustavu Kubernetes [17]

5.2.1. Master čvor

Kubernetes master naziv je za grupu procesa koji su zaslužni za upravljanje grozdom (slika 5.2). Prema službenoj dokumentaciji [4] to su:

- etcd – sinkronizirana distribuirana baza podataka na principu ključ-vrijednost (engl. *key-value*) koja sadrži sve konfiguracijske podatke te podatke o objektima i stanju sustava.
- kube-apiserver – glavna ulazna točka za korisnika. On je posrednik između nekog od korisničkih alata (na primjer kubectl) i upravljanja sustavom na grozdu.
- kube-control-manager – usluga koja je odgovorna za svu logiku sustava, održavanje stanja, praćenje životnog ciklusa, osiguravanje izvođenja potrebnih akcija.
- kube-scheduler – proces koji je zadužen za dodjeljivanje zadataka konkretnim čvorovima. Njegova je uloga pratiti kapacitet pojedinih čvorova i odabir najboljeg kandidata za provođenje zadatka.



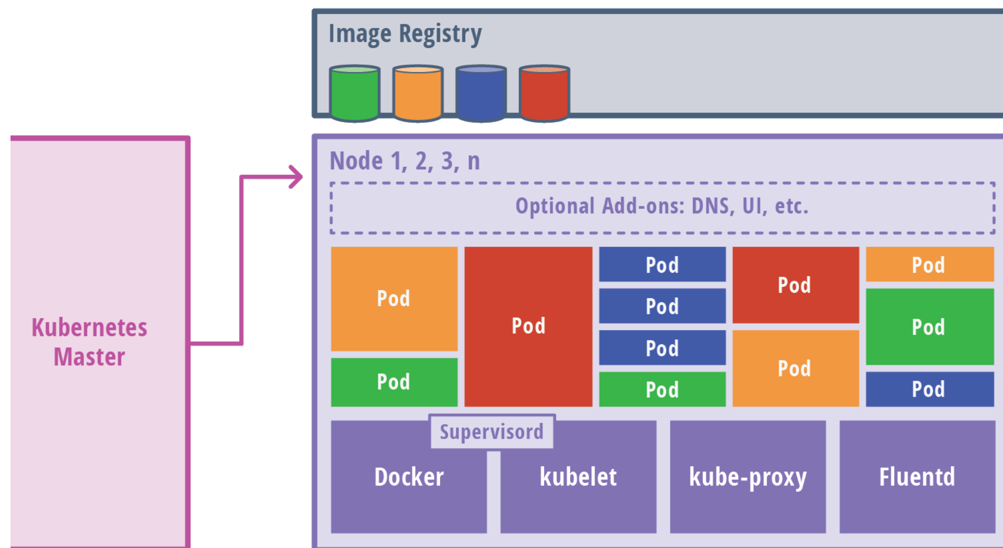
Slika 5.2: Procesi unutar čvora master u sustavu Kubernetes [17]

U praksi se često koristi repliciranje mastera kako bi se osigurala raspoloživost i redundancija [4], a tipično se koristi tri računala.

5.2.2. Ostali čvorovi

Svi čvorovi koji nisu dio master grupe zapravo su samo radnici koji obavljaju zadatke. Na njima se vrti neka od kontejnerskih tehnologija (na primjer Docker) i dva procesa (slika 5.3):

- kubelet – usluga preko koje kontrolna ravnina komunicira s čvorom. Također je u komunickaciji s etcd kako bi prikupio informacije potrebne za provođenje potrebnih akcija te djeluje kao posrednik između sustava Kubernetes i Dockera.
- kube-proxy – upravlja mrežnim prometom. Ova usluga omogućava komunikaciju između mikrousluga, proslijeđuje zahtjeve te dodaje pravila u vatrozid



Slika 5.3: Procesi unutar radnog čvora u sustavu Kubernetes [17]

5.3. Osnovni objekti

Unatoč tome što su kontejneri osnovni mehanizam, Kubernetes pruža veći broj različitih apstrakcija kojima se korisnicima olakšava opis željenog stanja sustava [4]. Te apstrakcije su zapravo Kubernetes objekti. U nastavku su objašnjeni oni osnovni:

- ljuska (engl. *pod*) – najmanja građevna jedinica. Već je spomenuto da se u sklopu sustava Kubernetes ne koriste kontejneri direktno. Naime, jedan kontejner ili više njih se pakira u jedinicu nazvanu ljuska. Sustav ljusku tretira kao jednu nedjeljivu aplikaciju čiji su kontejneri usko povezani i dijele čitav životni ciklus i dostupne resurse. Nad kontejnerima postoje još i druge apstrakcije koje se koriste puno češće, ali se u pozadini oslanjaju na ljuske.
- usluga (engl. *service*) – pruža način za prikazivanje ljuske kao mrežne usluge. Budući da su životni ciklusi ljuski dinamični, što znači da se stvaraju i uništavaju po potrebi, mora se osigurati način da ih se jednoznačno identificira.

Kubernetes svakoj ljusci pri stvaranju dinamički dodjeljuje IP adresu, ali zato, pomoću objekta usluge svaka grupa ljuski ima uvijek isto DNS ime. Upravo se zato komunikacija između mikrosuluga u sustavu Kubernetes uvijek odvija preko usluge, a ne preko ljuski. Postoji nekoliko vrsta usluga kao što su: clusterIP, NodePort, LoadBalancer i ExternalName.

- nositelj podataka (engl. *volume*) – tip objekta koji služi kao memorijski prostor. Omogućava spremanje podataka nakon uništavanja objekta (engl. *persistency*) i dijeljenje podataka između kontejnera unutar jedne ljuske. Nositelj podataka u suštini je direktorij koji može i ne mora u sebi imati neke podatke, a dostupan je kontejnerima u ljusci. Svaki kontejner zasebno mora povezati (engl. *mount*) putanju do direktorija s vlastitim datotečnim sustavom. Nositelj podataka vezan je uz ljusku koja ga koristi i s njom dijeli svoj životni ciklus. Unutar sustava Kubernetes postoje različite vrste nositelja podataka i jedna ljuska može koristiti više njih istovremeno.
- imenski prostor (engl. *namespace*) – pruža podršku za stvaranje virtualnih grozdova unutar fizičkog Kubernetes grozda. Drugim riječima, omogućava još jedan način izolacije, ali ovaj put na razini projekata ili korisnika. Imenski prostori dodjeljuju jedinstveno ime za grupu resursa, ljuski i ostalih objekata. Bitno je napomenuti da svaki objekt pripada isključivo jednom imenskom prostoru. Pri podizanju grozda računala stvaraju se tri osnovna imenska prostora: *default* za sve objekte bez naznačenog imenskog prostora, *kube-system* za unutarnje objekte sustava Kubernetes i *kube-public* za objekte koji moraju biti dostupni svim korisnicima.

Osim navedenih, Kubernetes ima još niz apstrakcija kojima je moguće uređivati sustav. Neki od njih su: *Deployment*, *Replication Controllers*, *Replication Sets* i drugi. Oni su objekti više razine i pružaju dodatne, napredne funkcionalnosti, ali u sklopu ovog rada neće se detaljno obrađivati.

5.4. Nadzor metrika u sustavu Kubernetes

Praćenje i nadzor mikrousluga jedan je od bitnih aspekata u životnom ciklusu sustava. Pravilno postavljeni alati olakšavaju održavanje sustava zdravim i brzim. Osim toga, nadzor je bitan za razumijevanje ponašanja sustava kako bismo mogli što bolje prilago-

diti performanse, skalirati broj ljuski ili povećati, odnosno smanjiti, resurse koje neki objekt koristi. Nadalje, ako se u sustavu pojavi greška ili se on neočekivano uspori, potrebno je pravovremeno primijetiti novo stanje kako bismo mogli brzo reagirati.

Platforma Kubernetes omogućava nam dohvat raznih informacija o trenutnom stanju sustava, odnosno objekata u sustavu, koristeći *kubectl* i naredbe kao što su na primjer:

```
kubectl get <tip_objekta>
kubectl describe <tip_objekta> <ime_ljuske>
kubectl exec -it <ime_ljuske> -- /bin/bash
kubectl logs <ime_ljuske>
```

Time možemo dobiti jednostavne meta podatke o objektu, detaljan opis ili se pak spojiti interaktivno u *bash* ljusku nekog kontejnera. Međutim, te informacije, iako izuzetno korisne za pronalazak i otklanjanje grešaka (engl. *debugging*) unutar aplikacije, nisu dovoljne jer nam daju samo informacije na razini pojedinih objekata.

Podatke na razini sustava možemo dobiti koristeći *metrics server*. Ovaj API preko kubeleta na pojedinim čvorovima sakuplja informacije o zauzeću memorije i opterećenosti procesora. Iako pruža ograničen skup metrika sustava, koriste ga neki od osnovnih dijelova sustava Kubernetes kao što su Horizontal Pod Autoscaler i Kubernetes Dashboard. Također, prikupljeni podaci ne spremaju se dugotrajno, zbog čega se često koriste alati s dugotrajnijom pohranom kao što je Prometheus [20, 5].

Prometheus je sustav otvorenog koda za monitoriranje sustava. Korisnik može odabrati mjere koje želi pratiti kroz neko vremensko razdoblje, a Prometheus tada sakuplja podatke od pokrenutih mikrousluga i pohranjuje ih u vlastitu bazu podataka. Korisnik može u bilo kojem trenu pristupiti sakupljenim podacima u obliku tablice ili grafova ako se koristi neki od vizualizacijskih alata. Ima vlastiti upitni jezik i dostupan je vanjskim alatima kao API. U kombinaciji s vizualizacijskim alatom Grafana, moguće je vrlo jednostavno dobiti različite podatke o sustavu [20].

6. Implementacija i vrednovanje programskog rješenja

U sklopu ovog rada napravljena je i demonstracija koncepata i tehnologija objašnjenih u prijašnjim poglavljima s primjenom na problem strojnog učenja. Ostvaren je sustav koji omogućava korisniku izgradnju modela strojnog učenja s odabranim parametrima te korištenje tog modela za klasifikaciju nad vlastitim podacima na velikoj skali s prilagodbom s obzirom na broj zahtjeva.

U sljedećim potpoglavljima dan je opis izabranog sustava, njegovo pakiranje u Dockerove kontejnere te u konačnici isporuka sustava na grozd računala s alatom Kubernetes.

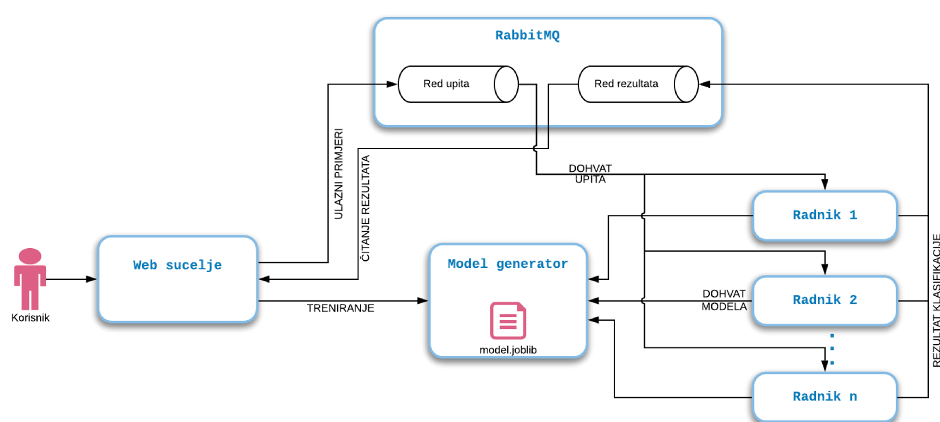
6.1. Analiza sentimenta

Kao pokazni primjer strojnog učenja izabran je često korišten primjer iz područja računalne obrade prirodnoga jezika (engl. *Natural-language processing*, NLP): analiza sentimenta. Analiza sentimenta odnosi se na prepoznavanje emocija u tekstu, odnosno klasifikaciju teksta na one s pozitivnim ili negativnim konotacijama.

Skup podataka pronađen je na platformi Kaggle pod nazivom "Sentiment140 dataset with 1.6 million tweets". Programski kod za izgradnju modela i njegovu primjenu preuzet je, uz manje izmjene, od korisnika Nikit Periwal [18].

6.2. Arhitektura sustava

Programska podrška sastoji se od četiri mikrousluge od kojih je jedan RabbitMQ, sustav za komunikaciju putem reda poruka. Ostali dijelovi sustava su korisničko sučelje napravljeno kao web usluga Flask, model generator API koji na korisnički upit pokreće učenje modela i služi kao spremište modela te konačno, model radnik. Implementiran je kao Pythonova skripta koja sluša poruke s RabbitMQ reda poruka, dohvaća podatke o modelu od model generatora i zatim radi predviđanje koristeći model i rezultat vraća u red poruka. Grafički prikaz implementiranog sustava vidljiv je na slici 6.1.



Slika 6.1: Grafički prikaz implementiranog sustava

6.3. Kontejnerizacija mikrousluga

Kontejnerizacija u praktičnom dijelu rada ostvarena je koristeći već spomenuti alat, Docker. Format pisanja Dockerfile datoteka i značenje ključnih riječi dani su u poglavlju 4.5.1.

Za mikrouslugu RabbitMQ nije potrebno pisati Dockerfile jer postoji već dostupna slika putem Docker Huba iz koje se može stvoriti kontejner sa željenom konfiguracijom.

Ostale su mikrousluge pisane u programskom jeziku Python i Dockerfile datoteka se razlikuje u manjim detaljima. Zato je u nastavku dan primjer za jednu od njih.

```
FROM python:3.6
```

```
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
ADD . /app
ENTRYPOINT [ "python" ]
CMD [ "app.py" ]
```

Kao temeljna slika izabrana je službena slika Dockerovog tima za programski jezik Python i verziju 3.6. Ova slika u sebi sadrži Pythonov interpreter, instalacijski paket *pip* i ostale programske knjižice potrebne za osnovno služenje programskim jezikom Python. Sljedeće dvije naredbe unutar datotečnog sustava slike stvaraju direktorij s imenom "app" i u njega kopiraju datoteku "requirements.txt". Ta datoteka sadrži popis dodatnih Pythonovih alata koje ćemo koristiti u kodu programske podrške. Primjeri korištenih alata su: flask, pika, joblib, pandas, numpy, sklearn i slični. Naredbom RUN pokrećemo instalaciju tih alata.

Slijedi dodavanje svih datoteka u kojima se nalazi programski kod. Bitno je naglasiti da se ovdje koristi točka kao relativni put do izvorišne adrese. Drugim riječima, Dockerfile datoteka mora biti smještena u vršni direktorij projekta. Posljednje dvije naredbe označavaju da će se prilikom pokretanja kontejnera iz ove slike pokrenuti datoteka "app.py" koristeći Pythonov interpreter.

6.4. Postavljanje Kubernetesovog grozda računala

Kako bi isporučili opisanu programsku podršku koristeći sustav Kubernetes, najprije je potrebno instalirati i povezati komponente opisane u poglavlju 6.2.

Izabran je grozd od tri virtualna računala na platformi Digital Ocean. Sva računala imaju iste karakteristike:

- Operacijski susav: Debian GNU/Linux 9
- Procesor: 2 jezgre
- Radna memorija: 4 GB
- Fizička memorija: 80 GB SSD

Jedno od računala služit će nam kao master čvor, dok će ostala dva biti radni. Instalacija će se provoditi koristeći kubeadm, alat namijenjen za jednostavno podizanje Kubernetesovog grozda u skladu s najboljom praksom.

6.4.1. Postavljanje čvora master

Kako bismo započeli instalaciju moramo se spojiti s odabranim računalom putem SSH protokola. Nakon toga, prvi korak jest dohvatiti i instalirati sustav Docker. To se ostvaruje nizom sljedećih naredbi:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
    | sudo apt-key add -
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable"
sudo apt-get update
sudo apt-get install -y docker-ce
sudo apt-mark hold docker-ce
```

Sljedeći korak je dohvat i instalacija alata kubeadm, kubectl i kubelet.

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
    | sudo apt-key add -
cat << EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
    deb https://apt.kubernetes.io/ kubernetes-xenial main
    EOF
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

Konačno, moguće je inicijalizirati grozd koristeći kubeadm. Nakon izvođenja ove naredbe, vrlo je bitno spremiti token i hash jer su oni privremeni, a služe za povezivanje ostalih čvorova u grozd.

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

6.4.2. Postavljanje radnih čvorova

Postavljanje radnih čvorova mnogo je jednostavnije nego postavljanje čvora master. Najprije je potrebno instalirati Docker, te Kubernetesove komponente kubectl, kubeadm i kubelet na isti način kao što je to izvedeno na čvoru master (objašnjeno u poglavlju 6.2.1). Nakon toga potrebno je jednostavno zalijepiti niže navedenu naredbu s ranije dobivenim tokenom i hashom. Provedbom ove naredbe će se čvorovi pridružiti grozdu.

```
sudo kubeadm join $ip_adresa:6443
       --token $token
       --discovery-token-ca-cert-hash $hash
```

6.4.3. Postavljanje mreže

Postavljanje mreže najbitniji je dio postavljanja grozda. Bez toga, koordinacija mikrousluga unutar Kubernetesovog grozda ne bi bila moguća. Postojali bi problemi komunikacije između ljuski, dinamičke alokacije ljuski i dodjeljivanje IP adresa te mogućnost da više ljuski koristi isti port. Kubernetes iz tog razloga ima drugačiji pristup: ljuske tretira kao virtualne strojeve i svaki dobije vlastitu virtualnu IP adresu preko koje je dostupan unutar grozda računala.

U ovom radu se za konfiguriranje mrežnih postavki koristi paket Flannel.

```
echo "net.bridge.bridge-nf-call-iptables=1" \
    | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
kubectl apply -f https://raw.githubusercontent.com
/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/ \
Documentation/kube-flannel.yml
```

Naposljetku, možemo se vratiti na čvor master i provjeriti da su se svi čvorovi unutar grozda uspješno povezali međusobno i u mrežu (slika 6.2).

```
kubectl get nodes
```

```
NAME            STATUS   ROLES    AGE   VERSION
ip-10-0-1-101   Ready    master   85s   v1.12.2
ip-10-0-1-102   Ready    <none>   63s   v1.12.2
ip-10-0-1-103   Ready    <none>   60s   v1.12.2
```

Slika 6.2: Prikaz uspješnog dohvata čvorova označava uspješno konfiguriran sustav Kubernetes

6.4.4. Postavljanje privatnog Dockerovog registra

Već je spomenuto kako je kontejnerizacija ključni element sustava Kubernetes te da je potrebno u tu svrhu povezati grozd računala s nekim od registara kako bi čvorovi mogli dohvatiti potrebne slike. U tu svrhu u ovom radu koristi se privatni Dockerov registar podignut na master čvoru.

Privatni, lokalni Dockerov registar stvara se iz već dostupne Dockerove slike nazvane *registry* donjom naredbom. Naredba stvara kontejner imena *registry* dostupan na portu 5000.

```
docker run -d
           -p 5000:5000
           --restart=always
           --name registry registry:2
```

Za produkcijska okruženja potrebno je osigurati još neke sigurnosne postavke kao što je postavljanje SSL certifikata i ostvarivanje kontrole pristupa, više o tome može se pronaći u službenoj dokumentaciji [13].

Kada gradimo vlastite slike, vrlo je bitno označiti te slike potrebnim zastavicama kako bi Docker mogao sliku spremiti na odgovarajuću lokaciju. Struktura oznake je sljedeća:

```
domaćin:port/ime_slike:verzija_slike
```

Primjer naredbi za dodavanje oznake i spremanje u lokalni registrar su:

```
docker tag moja_slika:1.0 localhost:5000/web_api:1.0
docker push localhost:5000/web_api:1.0
```

Nakon izgradnje pojedinih mikrousluga i njihove kontejnerizacije, sve se slike moraju na opisani način pohraniti u registar kako bi Kubernetes čvorovi mogli pristupiti slikama i iz njih stvoriti odgovarajuće kontejnere, odnosno ljuske.

6.5. Isporuka mikrousluga na Kubernetes

Nakon što su sve mikrousluge zapakirane u kontejnere, spremljene u Dockerov registar i grozd računala je spreman za korištenje, potrebno je napisati konfiguracijske datoteke za svaku od mikrousluga. Konfiguracijske se datoteke pišu u obliku YAML datoteka. Svaka od njih ima parametar tip (engl. *kind*) kojim se označava željeni tip objekta. U ovom radu korišteni su samo neki objekti, konkretno: *Namespace*, *Deployment* i *Service*.

Prvo stvaramo imenski prostor u kojem će živjeti svi objekti korišteni u radu.

```
---
kind: Namespace
apiVersion: v1
metadata:
  name: diplomski-ns
  labels:
    name: diplomski-ns
```

Nakon toga, stvaramo objekt tipa *Deployment* za svaku grupu kontejnera ili u ovom slučaju za sve četiri mikrousluge. Primjer je dan za konfiguraciju web sučelja:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: diplomski-ns
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: localhost:5000/frontend:1.0
        ports:
        - name: flaskport
          containerPort: 5000
      imagePullSecrets:
        - name: local_registry_secret
```

Objasnit ćemo samo neke od važnijih elemenata datoteke. Tip objekta postavljen je na *Deployment*. *Metadata* dio odnosi se na sve meta podatke objekta, tu navodimo željeno ime objekta i imenski prostor u kojem će se pokrenuti. Sa *spec* označavamo dio datoteke u kojem opisujemo konkretna svojstva ljuski unutar *deploymenta* te predložak (engl. *template*) za svaku od njih. U predlošku opisujemo kontejnere odnosno sliku iz koje će se izgraditi te potreban port.

Kako bi se ljuske mogle međusobno pronaći i raspoznati moramo definirati objekte tipa *Service*. U ovom radu razlikujemo dva tipa tih objekata: *targetPort* i *NodePort*. *TargetPort* koristi se za komunikaciju između ljuski, specificiranjem ovog tipa otvara se port na ljusci preko kojeg ona može komunicirati s ostalim ljuskama u tom imenskom prostoru. NodePort tip usluge koristi se za one ljuske kojima korisnik želi pristupiti izvan imenskog prostora, odnosno iz vanjske mreže.

Stvaranjem sljedeće usluge, ljuska u kojoj se nalazi mikrousluga s nazivom model-generator postaje dostupna na portu 5000. Ostale je mikrousluge mogu pronaći koristeći simboličko ime model-generator-local i port 5000.

```
apiVersion: v1
kind: Service
metadata:
  name: model-generator-local
```

```
  namespace: diplomski-ns
  labels:
    app: model-generator
spec:
  ports:
  - port: 5000
  selector:
    app: model-generator
```

NodePort tip objekta u ovom radu koristio se u svrhu otvaranja web sučelja korisnicima. Donji primjer pokazuje kako se pretpostavljeni port za Flask uslugu 5000 mapira na port 30000 grozda računala. Korisnik iz vanjske mreže nakon stvaranja usluge bit će u mogućnosti putem IP adrese grozda računala na portu 3000 pronaći sučelje za komunikaciju s ostatkom sustava.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-external
  namespace: diplomski-ns
spec:
  type: NodePort
  ports:
    - nodePort: 30000
      port: 5000
      targetPort: 5000
  selector:
    app: frontend
```

Preostaje još samo stvaranje objekata iz konfiguracijskih datoteka naredbom:

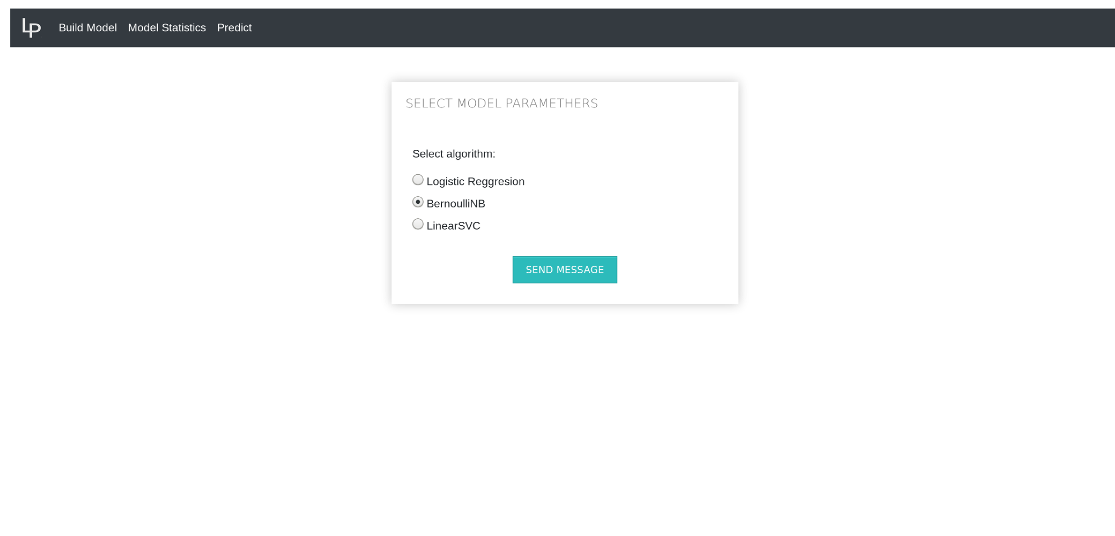
```
kubectl apply -f ime_datoteke
```

6.6. Vrednovanje programskog rješenja

Kako bi usporedili monolitnu arhitekturu s mikrouslužnom, promatrat ćemo sustav iz dvije različite perspektive. Usporedit ćemo onaj koji ima samo jednu mikrouslugu koja radi klasifikaciju (mikrousluga model-radnik) sa sustavom u kojem je omogućeno skaliranje broja radnika kako bi se prilagodio trenutnom zauzeću resursa. Pretpostavit ćemo da se sustav s jednim radnikom ponaša približno jednako kao i monolitna arhitektura.

6.6.1. Priprema modela

Prije samog vrednovanja programskog rješenja, potrebno je izgraditi željeni model strojnog učenja. Korisnik putem web sučelja može izabrati željeni algoritam strojnog učenja i podesiti neke od parametara. Odabir se vrši putem forme u 2 koraka, gdje je prvo potrebno odabrati željeni algoritam (slika 6.3) i zatim parametre za izgradnju modela.



Slika 6.3: Prikaz web sučelja za odabir algoritma strojnog učenja

Slika 6.4 prikazuje kako izgleda izbor parametara za algoritam naivnog Bayesa. Algoritam je dostupan putem programske knjižice *sklearn* pod nazivom *BernoulliNB*. Model se uči na temelju već spomenutog skupa podataka iz kojeg je uzorkovano 100 tisuća od 1.6 milijuna dostupnih Twitterovih objava. Od tih 100 tisuća, korisnik može odabrati koji točno omjer podataka za učenje i vrednovanje želi (engl. *train-test ratio*).

Slika 6.4: Prikaz web sučelja za odabir parametara za algoritam strojnog učenja

Odaberemo li omjer 80:20 i parametar Laplaceovog zaglađivanja postavimo na 1, tada se klikom na gumb "*Send message*" šalje zahtjev HTTP mikrousluzi model-generator. Rezultat je naučen model sa sljedećim karakteristikama:

- točnost = 75.22
- preciznost = 75.73
- odziv = 74.10
- mjera F1 = 75.54

Model je pohranjen unutar kontejnera model-generator i dostupan je za korištenje radnicima koji vrše klasifikaciju putem zahtjeva HTTP.

6.6.2. Horizontal Pod Autoscaler

U kontekstu sustava Kubernetes automatsko skaliranje moguće je namjestiti koristeći Horizontal Pod Autoscaler (HPA). Implementiran je kao Kubernetes API i kontroler, a zadužen je za automatsko prilagođavanje broja pojedinih ljuski uvjetima u sustavu. U trenutnoj verziji API-ja dostupna je podrška isključivo za skaliranje prema iskorištenosti procesora, ali moguće je koristiti i neke druge, prilagođene metrike koristeći beta verziju API-ja. Njome bi na primjer mogli upravljati temeljem zauzeća RAM memorije, broja poruka u redu poruka, broja zahtjeva u sekundi i slično. HPA implementiran je kao petlja što znači da periodično provjerava stanje sustava i skalira broj podova u ovisnosti o uočenom. Standardna vrijednost periodičnog provjeravanja je 15

sekundi, ali moguće ju je i podesiti prema želji. Osim toga, postoji i periodično hlađenje, odnosno smanjenje broja ljuski (engl. *downscale*) kada se za to pokažu uvjeti. Standardna vrijednost za period hlađenja je pet minuta.

Slika 6.5: Prikaz skaliranja sustava koristeći HPA

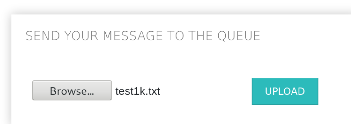
Graf na slici 6.5 prikazuje kako radi HPA od trenutka kada u sustav dođe veliki broj zahtjeva do trenutka kada se sustav vraća u početno stanje. HPA neprestano dohvaća podatke od metrics API-ja te izračunava optimalan broj ljuski. Na slici taj trenutak prikazan je zelenom linijom i oznakom *desired*. Zatim slijedi prilagodba sustava i povećanje broja ljuski što je označeno žutom linijom. U ovom slučaju broj ljuski povećan je na broj pet. Nakon što je obrada podataka gotova, ponovno dolazi do izračuna optimalnog broja ljuski i prilagodbe. Drugim riječima, sustav se vraća u početno stanje. Iz grafa vidimo da je za dani test optimalan broj ljuski pet.

Gore opisanim načinom ostvarujemo elastičan sustav koji se prilagođava dostupnim resursima. Također je bitno naglasiti da svaki radnik ima ograničene resurse kako bi jednostavnije prikazali svojstva i ponašanje nekog stvarnog produkcijskog sustava.

6.6.3. Test opterećenja

Test opterećenja napravit ćemo tako da se sustavu pošalje 1000 zahtjeva za klasifikaciju, odnosno da se u red poruka pošalje 1000 poruka. Zahtjev šaljemo postavljanjem datoteke (slika 6.6) u kojoj je svaka linija jedna Twitterova objava nasumično uzorkovana iz skupa podataka. Datoteka se potom obrađuje tako da se svaka linija zapakira u jednu poruku namjenjenu za RabbitMQ red poruka.

Nakon što se datoteka pošalje, slijedi obrada i slanje poruka. Budući da su radnici

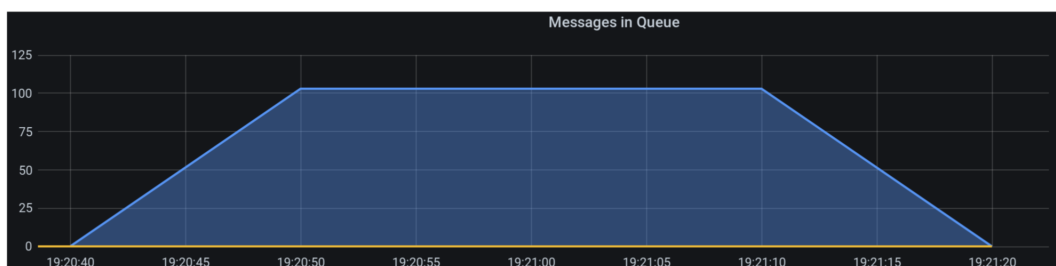


Slika 6.6: Prikaz web sučelja za postavljanje datoteke s porukama

implementirani kao potrošači (engl. *consumer*), oni već na pojavu prve poruke kreću s radom i pripremom rezultata. Nakon što se poruka obradi, radnik ju stavlja na red poruka "rezultati" kao što je prikazano na grafu sustava na slici 6.1.

Grafovi na slikama 6.7 i 6.8 prikazuju obradu 1000 poruka kroz vrijeme. Y-os prikazuje trenutni broj poruka spremnih za obradu, dok se na x-osi nalazi vrijeme.

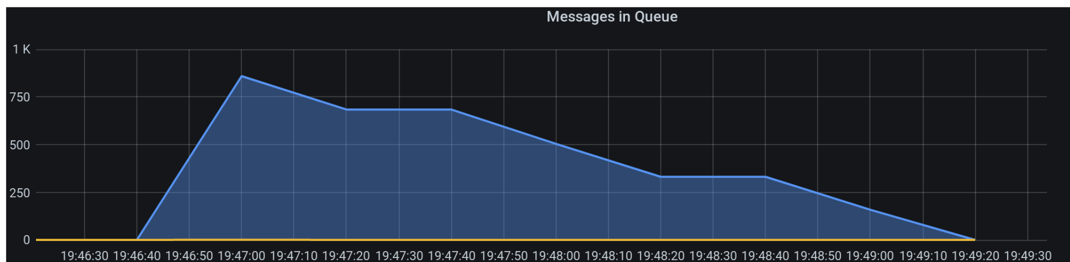
Slika 6.7 test je za jednog radnika ograničenog na maksimalan iznos od 500m (*milicorea*) što u kontekstu sustava Kubernetes znači da može koristiti 1/2 procesorske snage jezgre kojoj je ljuska pridružena. Još je važno naglasiti, da jedna točka na grafu prikazuje trenutni broj poruka dostupnih za obradu. Budući da obrada kreće već od pojave prve poruke u redu poruka, graf nikada neće doseći broj 1000. U ovom slučaju u redu poruka se ne zadržava više od 100 poruka u nekom trenu jer radnik dovoljno brzo preuzima i obrađuje poruke. Od trenutka 19:21:10, poruke uopće više ne dolaze u sustav te broj poruka opada sve do nule.



Slika 6.7: Graf prikazuje broj poruka u redu u danom trenutku koristeći jednog radnika

Na slici 6.8 prikazan je graf dobiven testiranjem sustava koji koristi HPA. U početnom stanju sustav ima jednog radnika ograničenog na 100m, odnosno 1/10 procesorske snage jezgre kojoj je ljuska pridružena. Broj radnika će se u ovom slučaju skalirati na

pet, što znači da svi zajedno imaju istu količinu resursa (500m) kao i jedan radnik. Graf za pet radnika poprilično se razlikuje od onog s jednim radnikom. Vidimo da se najveći broj poruka koji se nalazi u redu penje do 800 poruka u trenutku 19:47:00. Razlog tome je već objašnjeno ponašanje HPA, kojem treba 15 sekundi da uopće primijeti pojačano opterećenje sustava i tek nakon toga prilagođava broj ljuski radnika. Nakon te kritične točke, broj poruka u sustavu počinje padati.



Slika 6.8: Graf prikazuje broj poruka u redu u danom trenutku koristeći više radnika skaliranih s HPA

Zanimljivo je primijetiti da grafovi idu u prilog monolitnoj arhitekturi. Za obradu tisuću zahtjeva, jednom radniku treba otprilike 30 sekundi, dok sustav koji skalira pet radnika tu istu količinu obradi za 160 sekundi. Intuitivno gledajući, to ima smisla jer više razdvojenih usluga, znači i više međuprocesne komunikacije, više otvorenih opisnika datoteka (engl. *file descriptors*) i slično. Treba uzeti u obzir i vrijeme da HPA primijeti novo stanje u sustavu i prilagodi broj ljuski. Osim toga, svaki radnik prilikom pokretanja mora postaviti potrebne konfiguracijske postavke i učitati podatke o modelu kojeg će koristiti. Kod jednog radnika toga nema, što također utječe na smanjeno vrijeme obrade.

Ravne linije na grafu mogu se objasniti svojstvom alata za nadzor kojima je granulacija postavljena na pet sekundi. Prometheus svakih pet sekundi dohvaća podatke od sustava RabbitMQ i sprema ih u vlasitu bazu podataka. Rezultat toga je manji broj točaka u na grafu, dok se sve točke između aproksimiraju. S druge strane, moguće je i to da je broj poruka koji se u danoj točki stavio u red točno jednak broju trenutno obrađenih poruka, čime bi broj dostupnih poruka ostao isti kroz neko vrijeme.

Unatoč produljenom vremenu obrade pri korištenju mikrousluga, moramo uzeti u obzir da u stvarnom svijetu naš sustav nije konstantno pod maksimalnim opterećenjem, već broj zahtjeva varira tijekom dana. Ako bismo promatrali prosjek zauzeća resursa i cijene koju ćemo za iste platiti, onda mikrouslužna arhitektura dolazi do izražaja. Naravno, ovo je jako teško za dokazati i testirati jer ovisi o konkretnom problemu i

sustavu. Za to je potrebna detaljna analiza tržišta kojem se sustav isporučuje, njihovih navika i potreba.

7. Zaključak

S porastom popularnosti mikrouslužne arhitekture, javlja se i ideja da se takav način organizacije primijeni na probleme strojnog učenja. Promatramo li faze razvoja u klasičnom cjevovodu strojnog učenja, primjenu možemo pronaći u gotovo svakoj od njih. U ovom radu dano je teorijsko razmatranje primjene mikrouslužne arhitekture u kombinaciji s kontejnerskim tehnologijama na pojedine faze, te praktična primjena u fazi isporuke modela korisniku.

Prema jednostavnom testu sustava provedenom u poglavlju 6.6, dolazimo do rezultata koji daje naslutiti da je monolitna arhitektura ipak brža u obradi većeg broja zahtjeva. Kada bi imali sustav koji je pod stalnim opterećenjem, bez velikih oscilacija, to bi bilo istina. Međutim, bitno je uzeti u obzir da se veliki volumeni podataka u većini sustava pojavljuju samo povremeno i da veliki postotak vremena sustav ne koristi sve dostupne resurse. U takvim uvjetima mikrouslužna arhitektura, s pravilno postavljenim nadzorom i upravljanjem te mogućnošću skaliranja pojedinih dijelova sustava, znatno prevladava nad monolitnom.

Mogućnosti izolacije i jednostavne reprodukcije okruženja, jednostavne izmjene pojedinih komponenti bez narušavanja raspoloživosti sustava te mogućnost skaliranja pojedinih dijelova nezanemarive su prednosti u odnosu na samu brzinu izvođenja u jednom vremenskom periodu. U konačnici, prije razvoja svakog sustava, iznimno je važno napraviti analizu zahtjeva te nastojati predvidjeti ponašanje sustava kako bismo odabrali najprikladniju arhitekturu za konkretan problem.

LITERATURA

[1] What are azure machine learning pipelines?, 2020. URL https://cloudwwh.com/kubernetes-architecture/.

[2] Algoritmia. Productionizing Machine Learning Models with Containers. 2018. URL http://justinsgage.com/assets/data/portfolio/wp2.pdf.

[3] The Kubernetes Authors. Learn kubernetes basics, 2020. URL https://kubernetes.io/docs/tutorials/kubernetes-basics/.

[4] The Kubernetes Authors. Concepts, 2020. URL https://kubernetes.io/docs/concepts/.

[5] The Kubernetes Authors. Resource metrics pipeline, 2020. URL https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/.

[6] The Kubernetes Authors. Official page, 2020. URL https://kubernetes.io/.

[7] Mary Branscombe. How Kubernetes Could Orchestrate Machine Learning Pipelines. 2019. URL https://thenewstack.io.

[8] Mary Branscombe. Why Kubernetes and containers are the perfect fit for machine learning. 2019. URL https://jaxenter.com/containers-machine-learning-165203.html.

[9] Justin Ellingwood. An introduction to kubernetes, 2018. URL https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[10] Romana Gnatyk. Microservices vs monolith: which architecture is the best choice for your business?, 2018. URL https://www.n-ix.com/.

[11] Imesh Gunaratne. The evolution of linux containers and their future, July 2018. URL https://dzone.com/articles/evolution-of-linux-containers-future.

[12] Casper Hansen. Introduction To Machine Learning Deployment Using Docker and Kubernetes. 2020. URL https://mlfromscratch.com/deployment-introduction/.

[13] Docker Inc. Docker overview. URL https://docs.docker.com/get-started/overview/.

[14] Anton Kharenko. Monolithic vs. Microservices Architecture. *Medium*, 2015. URL https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59.

[15] Semi Koen. Architecting a Machine Learning Pipeline. *Medium*, 2019. URL https://towardsdatascience.com/architecting-a-machine-learning-pipeline-a847f094d1c7.

[16] Paula Kokić. Unaprijeđenje sigurnosti pomoću mikroservisa (Diplomski rad). *Sveučilište u Zagrebu, Fakultet organizacije i informatike*, 2018. URL https://urn.nsk.hr/urn:nbn:hr:211:697587.

[17] Janakiram MSV. Kubernetes: An overview, 2016. URL https://thenewstack.io/kubernetes-an-overview/.

[18] Nikit Periwal. Twitter sentiment analysis for beginners, 2020. URL https://www.kaggle.com/stoicstatic/twitter-sentiment-analysis-for-beginners.

[19] Dimitris Poulopoulos. How to Deploy your Machine Learning Models on Kubernetes. 2020. URL https://towardsdatascience.com/.

[20] Ran Ribenzaft. The Top 5 Kubernetes Metrics You Need to Monitor. 2020. URL https://epsagon.com/observability/the-top-5-kubernetes-metrics-you-need-to-monitor/.

[21] Jetinder Singh. The What, Why, and How of a Microservices Architecture. 2018. URL https://medium.com/.

[22] Nikola M. Zivkovic. Deploying Machine Learning Models. 2020. URL https://rubikscode.net/2020/02/10/deploying-machine-learning-models-pt-1-flask-and-rest-api/.

Sažetak

Mikrouslužna arhitektura je poznati i prihvaćeni način organizacije programske potpore. Korištenjem kontejnerizacije, mikrouslužna arhitektura je poboljšana ponajviše lakom prenosivosti usluge i osiguravanjem izoliranosti okoline izvođenja. U usporedbi s monolitnom arhitekturom, mikrouslužna arhitektura nudi brojne prednosti, ali i nedostatke. Jedan od većih nedostataka je upravo nadzor i orkestracija. Ovaj rad bavi se primjenom navedenih arhitekturnih koncepata na probleme strojnog učenja, konkretno na fazu isporuke modela korisniku. Na primjeru modela za analizu sentimenta izgrađen je sustav s kojim korisnik komunicira putem web usluge i šalje mu zahtjeve za klasifikacijom. Ostvareni sustav zapakiran je u kontejnere koristeći Docker, trenutno najpopularniju kontejnersku tehnologiju, i orkestriran je koristeći platformu Kubernetes.

Ključne riječi: mikrouslužna arhitektura, Docker, Kubernetes, orkestracija, strojno učenje

Scalable Microservice Architecture Based on Docker Platform Applied to Machine Learning Problems

Abstract

In the world of software development, microservices are a well known and already accepted software organization method. Microservices may be improved using container technologies, which ensure portability and isolation of execution environment. Compared to monolithic architecture, microservice architecture offers numerous advantages, but there are also some disadvantages. Some of the major shortcomings are orchestration and monitoring. This paper deals with the application of the mentioned concepts to machine learning with practical application to the model deployment phase. The system consists of microservices used for sentiment analysis, as a machine learning problem example. The user can communicate with the model via a web service and can send classification requests. The system is then packaged in containers using Docker, currently the most popular container technology, and is orchestrated using the Kubernetes platform.

Keywords: microservices, Docker, Kubernetes, orchestration, machine learning