



Univerzitet Union · Računarski Fakultet

Uporedna analiza rešenja: Docker Swarm i Kubernetes

MENTOR

prof. Mirjana Radivojević

KANDIDAT

Luka Jermić, 29M/2023

Sadržaj

1.0 Apstrakt	5
2.0 Uvod.....	6
3.0 Kontejneri	7
3.1 Šta su kontejneri?	7
3.2 Prednosti kontejnera.....	8
3.3 Problemi u radu sa kontejnerima	8
4.0 Docker.....	9
4.1 Arhitektura	9
4.1.1 Docker Daemon	9
4.1.2 Runc	9
4.1.3 Docker Client	10
4.1.4 Docker Desktop	10
4.1.5 Slika	10
4.1.6 Registar	10
4.1.7 Volumen	11
5.0 Docker swarm	12
5.1 Šta je Docker Swarm	12
5.2 Šta radi Docker Swarm	12
5.3 Kako Docker Swarm Radi	12
5.4 Arhitektura Docker Swarm-a	13
5.5 Usluge Docker Swarm-a	14
6.0 Kubernetes	15
6.1 Šta je Kubernetes	15
6.2 Prednosti korišćenja Kubernetes-a.....	15
6.4 Arhitektura Klastera Kubernetes	15
6.5 Komponente Kubernetes-a.....	16
6.6 Komponente Kontrolne Ravni	17
6.7 Mreža kubernetes-a	17
7.0 Docker Swarm i Kubernetes u Cloud-u	19
7.1 Prednosti Cloud Orkestracije	19
7.1.1 Skalabilnost.....	19
7.1.2 Visoka dostupnost	20
7.1.3 Optimizacija resursa.....	20
7.2 Interakcija IaaS, PaaS i SaaS sa Docker Swarm i Kubernetes.....	20

7.2.1 Infrastructure as a Service (IaaS)	21
7.2.2 Platform as a Service (PaaS)	22
7.2.3 Software as a Service (SaaS)	23
7.3 Kubernetes i Docker Swarm u trenutnim cloud okruženjima	24
7.3.1 Amazon Web Services (AWS)	24
7.3.2 Microsoft Azure	25
7.3.3 Google Cloud Platform (GCP).....	25
7.3.4 Ostali provajderi cloud usluga i hibridna cloud okruženja	26
7.4 Monitoring i load balancing praktican primer	27
7.4.1 Monitoring	27
7.4.2 Grafana.....	34
7.5 Automatsko skaliranje.....	35
7.5.1 Automatsko skaliranje u docker swarm-u	35
7.5.2 Automatsko skaliranje u kubernetes-u	39
8.0 Poređenje.....	40
8.1 Kategorijsko poređenje	40
8.1.1 Instalacija i Podešavanje	40
8.1.2 Skalabilnost.....	41
8.1.3 Balansiranje Opterećenja	41
8.1.4 Visoka Dostupnost	41
8.1.5 Mreža	42
8.1.6 Komanda-linijski Alati i GUI.....	42
8.2 Poređenje po metrikama	43
8.2.1 Poređenje u količini iskorišćene buffer memorije.....	43
8.2.2 Poređenje u menadžovanju memoriskog prostora	44
8.2.3 Poređenje u količini iskorišćenja procesora.....	45
8.2.4 Poređenje u GC (<i>garbage collection</i>)	46
8.2.5 Poređenje sistemskog opterećenja.....	47
9 Praktican Primer.....	48
9.1 Docker Swarm	48
9.1.1 Prvi Korak: Izgradnja Docker Image-a	48
9.1.2 Drugi Korak: Kreiranje Swarm Nodova	52
9.1.3 Treci Korak: Podešavanje nodova i swarm-a.....	53
9.2 Kubernetes	57
9.2.1 Prvi Korak: Izgradnja Docker Image-a	57

9.2.2 Drugi korak: Kreacija yaml fajla	57
9.2.3 Treci korak: podešavanje kubernetes klastera.....	59
9.2.4 Četvrti korak: otvaranje podova javnosti (Ingress)	61
10.0 Zaključak.....	63
11.0 Literatura	64

1.0 Apstrakt

U ovom radu vrši se temeljno poređenje Docker Swarm i Kubernetes tehnologija, koje su ključni alati u modernom okruženju razvoja softvera baziranom na kontejnerima. Docker, od svog nastanka 2013. godine, evoluirao je iz projekta jedne kompanije u standard za razvoj, testiranje i implementaciju aplikacija u kontejnerima i upotrebu u *cloud* okruženju. Docker Swarm i Kubernetes, kao dve dominante platforme za orkestraciju kontejnera, omogućavaju efikasno upravljanje i automatizaciju desetina, stotina, pa čak i hiljada kontejnera. Ovaj rad detaljno objašnjava kako Docker funkcioniše, opisuje arhitekture i ključne karakteristike Docker Swarm-a i Kubernetes-a, i pruža dubinsku analizu njihovih prednosti i ograničenja. Cilj je da se čitaocima pruži jasan uvid u situacije kada je preferiranje jedne tehnologije nad drugom opravdano, zavisno od specifičnih potreba projekta i tehničkog okruženja.

2.0 Uvod

U eri brzog razvoja softvera i kontinuirane integracije, kontejnerizacija je postala ključna tehnologija koja omogućava agilno i efikasno upravljanje aplikacijama u različitim okruženjima. U ovom kontekstu, alati poput Docker-a, koji je postao sinonim za kontejnerizaciju, i platforme za orkestraciju kao što su Docker Swarm i Kubernetes, postaju neizostavni za razvojne timove širom sveta. Ovaj uvodni deo rada pruža pregled ovih tehnologija, od njihovog porekla i osnovnih principa do praktične primene u razvoju modernih aplikacija. Kroz analizu funkcionalnosti, arhitekture i performansi, cilj je da se istaknu prednosti i nedostaci svake tehnologije, omogućavajući čitaocima da donesu informisane odluke pri odabiru alata za svoje projekte.

3.0 Kontejneri

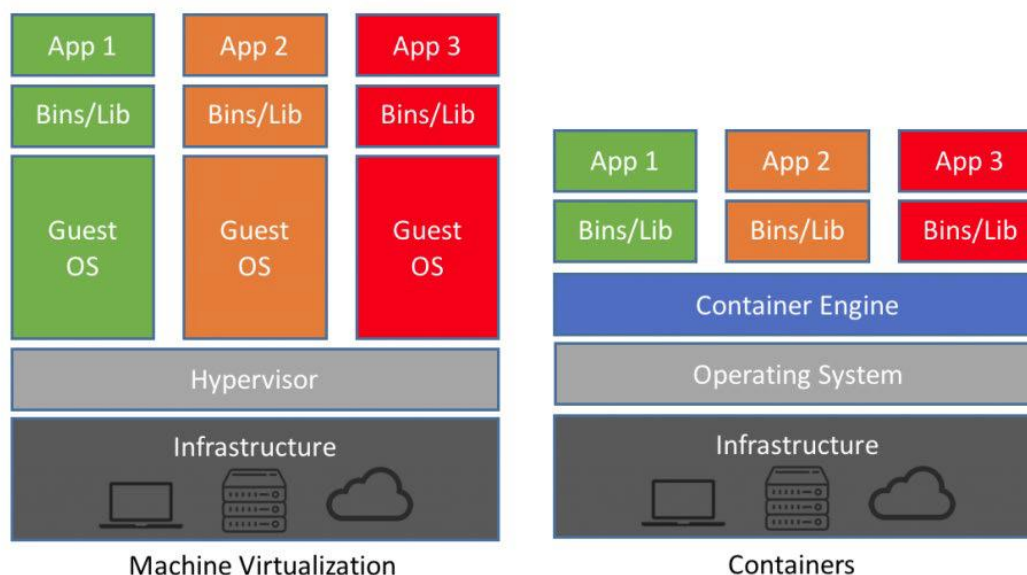
3.1 Šta su kontejneri?

Do nedavno, standard u industriji bio je da se za izvršavanje softverskih aplikacija koriste virtualne mašine (VMs). Ove mašine omogućavaju pokretanje aplikacija unutar zasebnog gostujućeg operativnog sistema, koji funkcioniše na virtualizovanom hardveru pod upravom operativnog sistema domaćina [18] [22].

Virtualne mašine su efikasne u osiguravanju temeljne izolacije za aplikacije, štiteći ih od problema koji mogu nastati u operativnom sistemu domaćina i obrnuto. Ipak, visok stepen izolacije koji pružaju dolazi uz značajan računarski trošak - značajna količina procesorske snage se troši na virtualizaciju hardvera za potrebe gostujućeg operativnog sistema.

Kontejneri zauzimaju drugačiji pristup. Koristeći funkcije niskog nivoa operativnog sistema domaćina, kontejneri postižu gotovo isti nivo izolacije kao virtualne mašine, ali koristeći znatno manje procesorske snage. Kontejneri omogućavaju pakovanje samo neophodnih komponenti aplikacije - same aplikacije, potrebnih biblioteka, zavisnosti i konfiguracijskih datoteka, što aplikaciju čini nezavisnom od operativnog sistema i infrastrukture na kojoj se pokreće.

Docker predstavlja sistem zasnovan na kontejnerima koji se koristi za aplikacije. Ako ste upoznati sa konceptom virtualnih servera, Docker nudi dodatne nivoe apstrakcije za vašu aplikaciju. Evo kako vizuelno možemo prikazati razlike između kontejnera i virtualnih mašina:



Slika 1 - Arhitektura kontejnera

3.2 Prednosti kontejnera

Primarne prednosti upotrebe kontejnera leže u njihovoj efikasnosti pri upotrebi resursa i visokoj fleksibilnosti. Nedostatak potrebe za kompletnim operativnim sistemom drastično smanjuje količinu prostora potrebnog za skladištenje, smanjujući ga sa nekoliko gigabajta na tek desetine ili stotine megabajta. Ovo nam omogućava da bez većih poteškoća pokrenemo veći broj kontejnera u odnosu na korišćenje virtuelizacije. Dodatno, značajno su umanjeni zahtevi za procesorskom snagom kao i vreme potrebno za aktiviranje kontejnera. Proces koji radi na operativnom sistemu domaćina ne repliciraju se unutar kontejnera, što je u suprotnosti sa virtuelnim mašinama koje zahtevaju celokupan operativni sistem za svaku pojedinačnu instancu.

Fleksibilnost kontejnera dolazi od činjenice da sve esencijalne datoteke potrebne za funkcionisanje aplikacije postoje unutar samog kontejnera. Kontejneri mogu imati nezavisna mrežna sučelja koja se razlikuju od onih na domaćinskom uređaju, izbegavajući time potencijalne konflikte prilikom upotrebe određenih mrežnih portova. Zahvaljujući ovoj prenosivosti, testiranje i praćenje grešaka postaju jednostavniji, jer se isti kontejneri koriste kako u testnoj tako i u produkcijskoj fazi.

3.3 Problemi u radu sa kontejnerima

Kontejneri se mogu opisati kao oskudne verzije virtuelnih mašina, jer uključuju samo esencijalne elemente neophodne za funkcionalnost aplikacija. Ovo bi moglo sugerisati da je upravljanje njihovom sigurnošću jednostavnije, ali to zapravo nije slučaj. Specifična struktura kontejnera onemogućava korišćenje uobičajenih sigurnosnih alata i protokola koji su standard u IT industriji. Dodatno, kontejneri dele osnovni *kernel* operativnog sistema na kojem operišu, što ih čini manje izolovanim u poređenju sa virtuelnim mašinama, te stoga bilo kakav sigurnosni propust u *kernelu* može kompromitovati sve kontejnere.

Jedan od ključnih izazova je i nemogućnost pokretanja kontejnera sa različitim operativnim sistemima na istoj hardverskoj platformi, što može biti značajna prepreka pri implementaciji složenih sistema. Takođe, uspostavljanje mrežne konekcije između izolovanih kontejnera nije trivialan zadatak. Problem se dodatno komplikuje time što kontejneri, po *default-u*, nisu dizajnirani da se međusobno "vide" i komuniciraju.

4.0 Docker

Docker, koji je dostupan kako za *Linux* tako i za *Windows* operativne sisteme, predstavlja ključni alat namenjen razvoju, implementaciji i upravljanju aplikacijama putem kontejnera. Ovaj softver omogućava programerima da biraju radnu platformu i razvijaju aplikacije nezavisno od operativnog sistema na kojem će one kasnije biti pokrenute. Iako su kontejneri korišćeni još nekoliko decenija, prava revolucija u njihovoj primeni započela je sa lansiranjem Docker-a, koji je ubrzo postao dominantna platforma za manipulaciju kontejnerima. Centralni element Docker-a je **Docker daemon**, servis koji upravlja kontejnerima, njihovim slikama i infrastrukturom. Ovaj *daemon* interaguje sa Docker client-om kroz *API*, formirajući klijent-server arhitekturu koja omogućava korisnicima da upravljaju procesima unutar Docker ekosistema [6] [8].

4.1 Arhitektura

Docker koristi arhitekturu klijent-server. Docker klijent je alat komandne linije koji omogućava korisnicima da interaguju sa Docker *daemon-om*. Docker klijent i *daemon* mogu funkcionisati na istom sistemu, ali je takođe moguće da se Docker klijent poveže sa udaljenim Docker *daemon-om*, pri čemu komuniciraju putem *REST API*-ja. Još jedan Docker klijent je Docker Compose, koji omogućava upravljanje aplikacijama sastavljenim od više kontejnera.

Docker *daemon* je pozadinski servis koji operiše na vašem operativnom sistemu i upravlja kreiranjem, pokretanjem i distribuiranjem Docker objekata poput slika, kontejnera, mreža i volumena. Docker *daemon* čeka na zahteve iz *REST API*-ja i na osnovu njih izvršava različite operacije. Daemon takođe može komunicirati sa drugim *daemon-ima* za upravljanje Docker uslugama.

4.1.1 Docker Daemon

Docker Daemon funkcioniše kao glavna servisna komponenta koja sluša *API* zahteve i upravlja image-ima, kontejnerima, mrežama i *volume-ima*. Osim što upravlja lokalnim resursima, Docker Daemon može da komunicira sa drugim docker daemonima kako bi koordinisao delovanje više sistema. Poznat je i pod imenom *containerd*, što je deo koji je izdvojen iz Docker-a i koji može biti korišćen nezavisno, čak i od strane sistema kao što je Kubernetes. *Containerd* omogućava upravljanje kontejnerima na niskom nivou, pružajući osnovne funkcionalnosti potrebne za pokretanje i upravljanje kontejnerima, kao što su kreiranje, pokretanje, pauziranje i zaustavljanje kontejnera.

4.1.2 Runc

Runc predstavlja *low-level container runtime* koji je odgovoran za stvaranje i pokretanje kontejnera. Ova komponenta uključuje *libcontainer* biblioteku, koja je napisana u *Go* programskom jeziku i predstavlja osnovu za izvršavanje kontejnera na niskom nivou. *Runc* omogućava pokretanje kontejnera sa *Linux kernel* izolacijskim funkcijama kao što su *namespaces* i *cgroups*, pružajući izolaciju resursa i sigurnost.

4.1.3 Docker Client

Docker Client je osnovni način interakcije sa Docker sistemom, gde korisnici pretežno unose komande putem konzole. Ovaj klijent omogućava korisnicima da šalju komande Docker Daemon-u, čime se upravlja celokupnim Docker okruženjem. *Docker Client* koristi jednostavnu i intuitivnu sintaksu, omogućavajući lako kreiranje, pokretanje i upravljanje kontejnerima, mrežama, i volumenima.

4.1.4 Docker Desktop

Docker Desktop je aplikacija namenjena za operativne sisteme *Windows* i *Mac*, koja korisnicima omogućava lakše korišćenje Docker-a na desktop računarima. Ova aplikacija uključuje Docker Daemon, Docker Client, Docker Compose, kao i niz drugih alata koji olakšavaju razvoj, testiranje i implementaciju aplikacija unutar Docker kontejnera. Docker Desktop dolazi sa grafičkim korisničkim interfejsom koji olakšava upravljanje kontejnerima i slikama, omogućavajući korisnicima da vizuelno pregledaju i upravljaju svojim Docker resursima.

4.1.5 Slika

Docker slika je lagani, samostalni, izvršni paket koji sadrži sve što je potrebno za pokretanje softvera, uključujući kod, *runtime*, biblioteke i konfiguracione datoteke. Slika takođe sadrži druge konfiguracije za kontejner, kao što su varijable okruženja, zadana naredba za pokretanje i drugi metapodaci. Dakle, slika je šablon za čitanje koji sadrži skup uputstava za kreiranje kontejnera, a kontejner je tada pokrenuta instanca slike. Docker slike se prave od serije slojeva, pri čemu svaki sloj predstavlja instrukciju u Dockerfile datoteci slike. Ovaj dizajn omogućava Docker slikama da dele slojeve, što ih čini efikasnijim, lakšim za skladištenje i bržim za prenos. Docker slike se koriste za kreiranje Docker kontejnera, koji su pokretne instance slika, obezbeđujući prenosiv i dosledan okruženje u različitim razvojnim, testnim i proizvodnim postavkama. Ova enkapsulacija i prenosivost čine Docker neophodnim alatom za savremeni razvoj i implementaciju softvera.

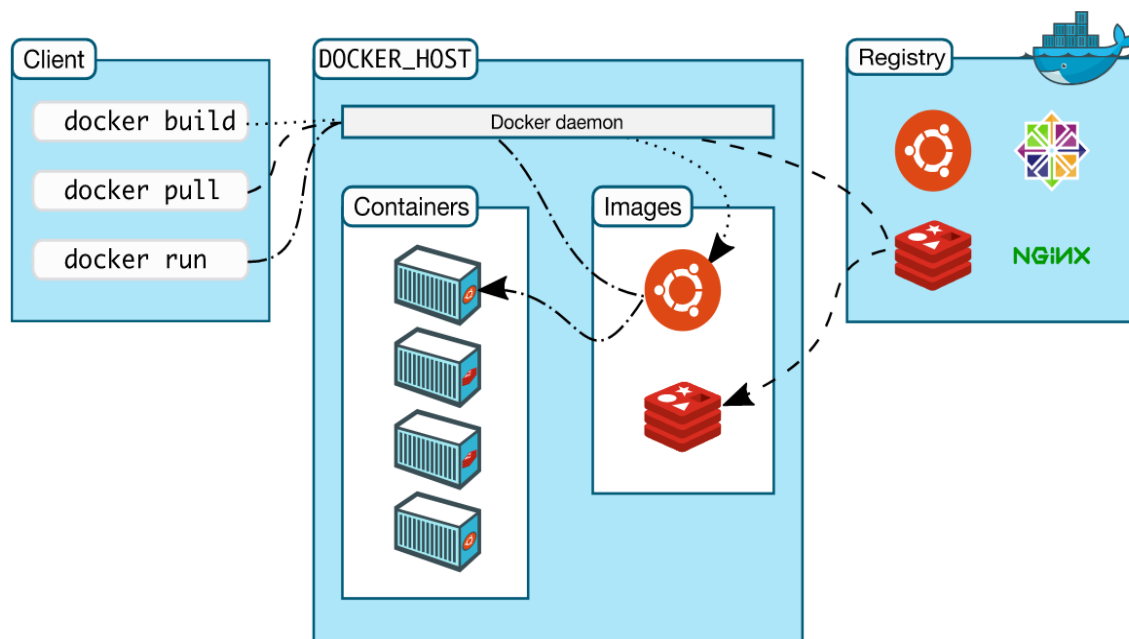
4.1.6 Registar

Docker registar je skladište za Docker slike. Docker klijenti se povezuju s registrima da bi preuzeli slike za upotrebu ili učitali slike koje su kreirali. Registri mogu biti javni ili privatni. Glavni javni registar je *Docker Hub*, web stranica na kojoj se dele slike kontejnera. Na njemu je moguće pronaći brojne gotove slike, što je jedan od razloga popularnosti Dockera.

Posebna vrsta slika na *Docker Hubu* su oficijalne slike, čiji razvoj i održavanje finansira sam Docker. Osim slike *hello-world* kojom se testira ispravnost instalacije Dockera, među službenim slikama su *httpd*, *python*, *php*, *node*, *haproxy*, *mariadb*, *mongo*, *postgres*, *redis*, *nextcloud*, *memcached*, *docker* (da, Docker može pokrenuti drugi Docker unutar kontejnera koji zatim može pokretati Docker kontejnere) i mnoge druge. Na stranici svake od službenih slika date su detaljne upute za njihovo korišćenje.

4.1.7 Volumen

Docker volumeni su široko korišćen i koristan alat za osiguranje postojanosti podataka tokom rada unutar kontejnera. Volumeni su fajl sistemi montirani na kontejnere za očuvanje podataka koje generiše radni kontejner. Volumeni se čuvaju na domaćinu i omogućavaju jednostavno pravljenje sigurnosnih kopija i deljenje fajl sistema između kontejnera. Ovo omogućava korisnicima da zadrže podatke čak i kada se kontejneri brišu ili premeste, osiguravajući da kritični podaci ostanu dostupni i zaštićeni.



Slika 2 - Docker arhitektura

5.0 Docker swarm

5.1 Šta je Docker Swarm

Docker Swarm je alat za orkestraciju kontejnera, omogućavajući grupisanje i raspoređivanje Docker kontejnera. IT administratori i programeri mogu uspostaviti i upravljati klasterom Docker čvorova kao jednim virtuelnim sistemom. Docker Swarm omogućava programerima da spoje više fizičkih ili virtuelnih mašina u klaster, poznate kao *nodovi* ili *demoni*. Administratori i programeri mogu pokretati kontejnere, povezivati ih na više hostova, upravljati resursima čvorova i poboljšati dostupnost aplikacija.

Docker Engine koristi Swarm mode kao sloj između operativnog sistema i kontejner slika, integrirajući orkestracione mogućnosti Docker Swarm-a. Swarm mode koristi standardni Docker *API* za interakciju sa drugim Docker alatima, kao što je Docker Machine. Ova integracija omogućava lako postavljanje i upravljanje kontejnerima na različitim platformama i okruženjima. [5]

5.2 Šta radi Docker Swarm

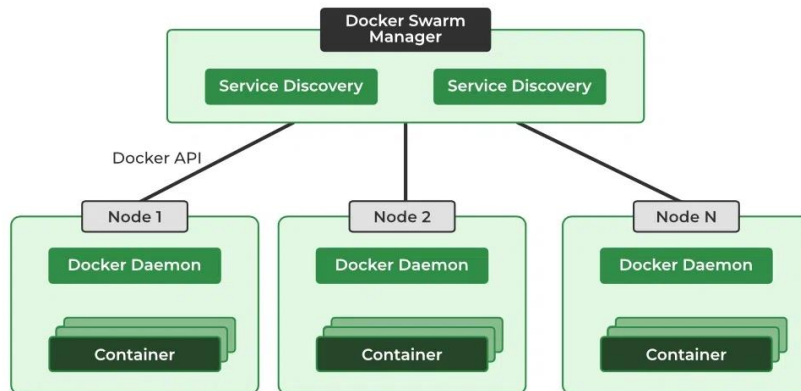
Docker platforma uključuje razne alate, usluge, sadržaj i automatizacije koje pomažu programerima da kreiraju, distribuiraju i pokreću aplikacije bez potrebe za konfiguiranjem osnovnog razvojnog okruženja. Sa Docker-om, programeri mogu pakovati i pokretati aplikacije u laganim kontejnerima - izolovanim okruženjima koja omogućavaju efikasno i besprekorno izvršavanje aplikacija u različitim uslovima.

Docker Swarm omogućava automatizaciju različitih aspekata orkestracije, uključujući skaliranje aplikacija, upravljanje dostupnošću i balansiranje opterećenja. Ovo omogućava administratorima da efikasno upravljaju resursima i osiguraju visoku dostupnost aplikacija bez potrebe za složenim manuelnim intervencijama.

5.3 Kako Docker Swarm Radi

Docker Swarm pretvara resurse hosta u zajednički bazen za Docker kontejnere. Kontejneri mogu biti povezani sa više hostova unutar klastera. Svaki čvor može rasporediti i pristupiti bilo kojem kontejneru unutar swarm-a. Swarm uključuje više *worker nodova* i najmanje jedan menadžer *node* za kontrolu aktivnosti klastera.

Swarm menadžeri koriste *Raft* konsenzusni algoritam za održavanje konzistentnog stanja klastera. Ovaj algoritam omogućava replikaciju stanja između menadžera, osiguravajući visok nivo dostupnosti i otpornosti na kvarove. *Worker nodovi* izvršavaju zadatke koje im dodeljuju menadžeri, omogućavajući efikasno raspoređivanje resursa i skaliranje aplikacija.



Slika 3 - Docker Swarm arhitektura

5.4 Arhitektura Docker Swarm-a

Docker Swarm koristi distribuiranu arhitekturu za upravljanje klasterom kontejnera. Sastoji se od dva glavna tipa *nodova*: menadžer *nodovi* (*manager nodes*) i radni *nodovi* (*worker nodes*). Ova arhitektura omogućava efikasno skaliranje, visoku dostupnost i robustan mehanizam za orkestraciju kontejnera.

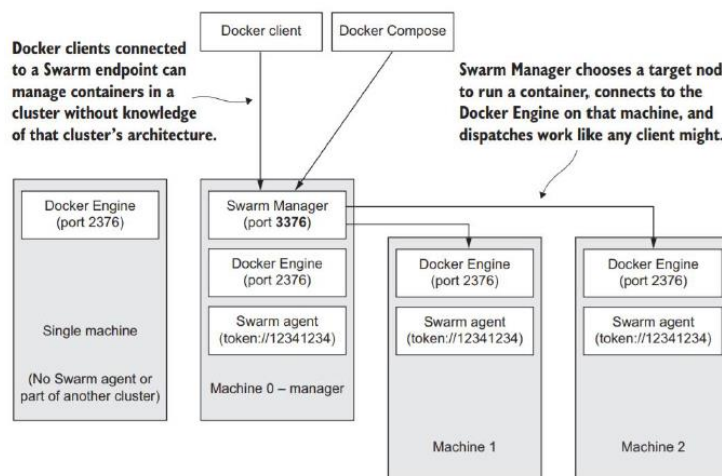
Menadžer Nodovi: Menadžer *nodovi* su srce Docker Swarm klastera. Oni su odgovorni za održavanje stanja klastera, planiranje zadataka (*tasks*), i pružanje *API endpoint-a* za interakciju sa klasterom. Menadžeri koriste Raft konsenzusni algoritam za postizanje distribuisanog konsenzusa, što omogućava visok nivo dostupnosti i otpornosti na kvarove. Raft algoritam omogućava replikaciju stanja između više menadžera, čime se obezbeđuje konzistentnost i otporno liderstvo (*leader election*) u slučaju kvara trenutnog lidera.

Komponente Menadžera: Menadžer *nod* se sastoji od nekoliko ključnih komponenti:

- **API:** Prihvata komande od klijenta i upravlja interakcijama sa klasterom.
- **Orchestrator:** Evaluira stanje klastera i osigurava da se željeno stanje poklapa sa trenutnim stanjem, stvarajući zadatke koji treba da se izvrše.
- **Allocator:** Dodeljuje resurse novim zadacima, kao što su *IP* adrese i mrežni resursi.
- **Scheduler:** Dodeljuje zadatke radnim nodovima na osnovu dostupnih resursa i opterećenja nodova.
- **Dispatcher:** Prosleđuje zadatke radnim nodovima i prati njihovo izvršenje putem heartbeat mehanizma koji održava komunikaciju sa radnim nodovima.

Docker Swarm API: *Endpoint-i* Docker Swarm Manager-a izlažu Swarm API. Klijenti mogu koristiti ovaj API za upravljanje ili inspekciju klastera. Štaviše, Swarm API je ekstenzija Docker *Remote API*-ja. Zapravo, bilo koji Docker klijent može se direktno povezati sa Swarm *endpoint-om*, kao da je u pitanju pojedinačna mašina.

Slika ispod pokazuje kako je moguće koristiti isti Docker klijent za interakciju sa Swarm klasterom. Međutim, implementacija Docker Remote API-ja razlikuje se od Docker Engine-a. Naime, u zavisnosti od specifične operacije, jedan zahtev klijenta može uticati na jedan ili više Swarm čvorova.



Slika 4 - Swarm klaster sa jednostavnim Docker klijentom

Radni Nodovi: Radni *nodovi* izvršavaju zadatke koje im dodeljuju menadžer *nodovi*. Oni redovno šalju informacije o svom stanju menadžerima, uključujući dostupne resurse i status zadataka. Radni *nodovi* su skalabilni i mogu se dodavati ili uklanjati iz klastera bez ometanja celokupnog sistema. Ovo omogućava horizontalno skaliranje i otpornost na kvarove pojedinačnih *nodova*.

Visoka Dostupnost i Oporavak: Docker Swarm implementira mehanizme za visoku dostupnost i oporavak sistema. Korišćenjem multiple menadžer *nodova* i *Raft* algoritma, klaster može da preživi kvar jednog ili više menadžera bez gubitka stanja ili funkcionalnosti. Radni *nodovi* mogu preuzeti zadatke sa drugih *nodova* u slučaju kvara, čime se obezbeđuje kontinuitet usluga.

Interoperabilnost i Integracija: Docker Swarm je dizajniran da bude potpuno *interoperabilan* sa postojećim Docker alatima i ekosistemom. Korisnici mogu koristiti poznati Docker *CLI* za upravljanje Swarm klasterom, što olakšava prelazak sa jednostavnih Docker *deploymenata* na složenije orkestrirane sisteme. Integracija sa Docker Compose omogućava definisanje multi-kontejnerskih aplikacija koristeći jednostavne *YAML* datoteke, što dodatno pojednostavljuje upravljanje aplikacijama.

5.5 Usluge Docker Swarm-a

U Docker Swarm-u, usluge se mogu rasporediti na dva načina: replikovane i globalne. Replikovane usluge zahtevaju od administratora da specificira broj identičnih "replika" zadataka, dok globalne usluge prate sve kontejnere na čvoru, raspoređujući po jedan zadatak na svaki dostupni čvor.

Replikovane usluge omogućavaju skaliranje aplikacija tako što raspoređuju više instanci iste usluge na različite *nodove*. Ovo osigurava da je aplikacija dostupna čak i ako neki *nodovi* zakažu. Globalne usluge, s druge strane, garantuju da se svaka instanca usluge pokreće na svakom *nodu* u klasteru, što je korisno za aplikacije koje zahtevaju uniformno prisustvo na svim *nodovima*, kao što su usluge *monitoringa* ili *logovanja*.

6.0 Kubernetes

6.1 Šta je Kubernetes

Kubernetes je sistem za upravljanje aplikacijama u kontejnerima preko klastera mašina, koji je razvio Google na osnovu iskustava sa svojim internim sistemima Borg i Omega. Ovaj *open-source* sistem omogućava efikasno upravljanje i skaliranje velikog broja aplikacija, tako da korisnici vide sve resurse kao jedinstvenu celinu. Kubernetes abstrahuje infrastrukturu, omogućavajući jednostavno postavljanje aplikacija bez obzira na veličinu klastera, što ga čini sličnim Docker Swarm-u po master/slave arhitekturi. Platforma pruža usluge kao što su otkrivanje servisa, skaliranje, balansiranje opterećenja, samoizlečenje i izbor *lidera*, omogućavajući programerima da se fokusiraju na razvoj poslovne logike umesto na infrastrukturne detalje. Zbog ovih funkcionalnosti, Kubernetes je postao najprihvaćenije rešenje na tržištu za orkestraciju kontejnera. [7]

6.2 Prednosti korišćenja Kubernetes-a

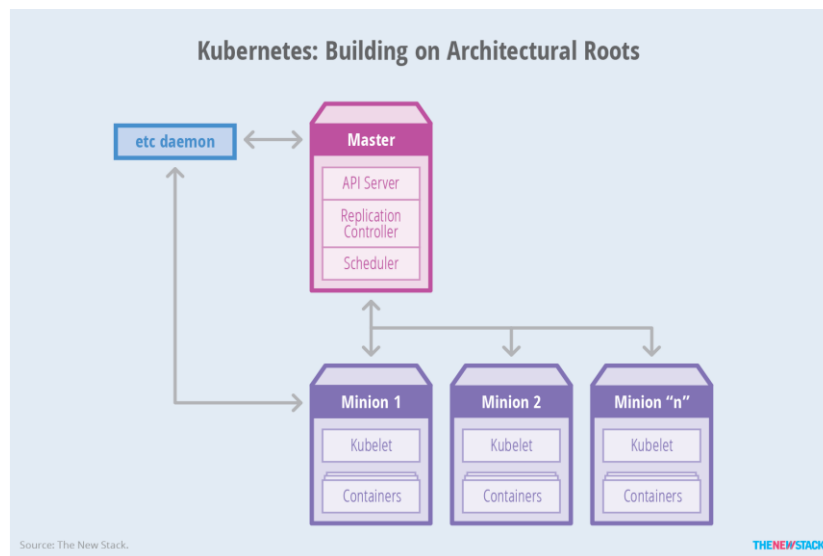
- **Automatizovano postavljanje i upravljanje:** Kubernetes automatizuje postavljanje, skaliranje i operacije nad kontejnerima aplikacija. Ovo eliminiše potrebu za ručnim upravljanjem procesima, smanjujući mogućnost ljudskih grešaka i povećavajući efikasnost postavljanja.
- **Skalabilnost:** Kubernetes automatski skalira aplikacije na osnovu zahteva koji se realno posmatra. Ovo se postiže horizontalnim skaliranjem *pod-ova*, koje prilagođava broj aktivnih kontejnera prema potrebi.
- **Visoka dostupnost:** Kubernetes poboljšava dostupnost aplikacija i smanjuje kašnjenje, osiguravajući da su usluge dostupne kad god su potrebne i da uvek rade optimalno.
- **Smanjenje troškova:** Kubernetes optimizuje korišćenje infrastrukture, minimizirajući troškove sprečavanjem prekomernog provizionisanja resursa i upravljajući efikasnom upotrebom računarske snage.
- **Povećana produktivnost programera:** Prebacivanjem velikog dela operativnog tereta na Kubernetes, programeri mogu više da se fokusiraju na pisanje koda nego na upravljanje postavljanjem i skaliranjem svojih aplikacija.

6.4 Arhitektura Klastera Kubernetes

Kubernetes klaster se sastoji od master čvora, jednog ili više *minion* čvorova i sistema za trajno skladištenje podataka. U klasteru postoji najmanje jedan radni čvor. Master čvor je kontrolni centar klastera, odgovoran za orkestraciju i implementaciju aplikacija preko radnih čvorova. Svi zahtevi korisnika prema Kubernetes-u prolaze kroz REST API sloj, omogućavajući kreiranje, čitanje, ažuriranje i brisanje (CRUD) objekata unutar klastera. Master čvor takođe koristi *etcd* za čuvanje podataka o stanju klastera i sadrži Controller-Manager koji upravlja životnim ciklusom aplikacija.

Na *minion* čvorovima, *Kubelet* je ključna komponenta koja izvršava specifične zadatke platforme i upravlja pod-ovima, koji su osnovne jedinice implementacije u Kubernetes-u. *Kubelet* sarađuje sa *cAdvisor-om* za prikupljanje podataka o performansama i stanju sistema, što je ključno za nadzor i optimizaciju rasporeda pod-ova. Pored *Kubelet-a*, minion čvorovi uključuju i *runtime* za kontejnere (kao što su Docker, *rkt*, *cri-o*), koji upravlja životnim ciklusom kontejnera, i Kube Proxy, koji upravlja mrežnim saobraćajem i balansiranjem opterećenja.

Dodatne funkcionalnosti Kubernetes-a uključuju DNS za rezoluciju imena servisa, Ingress Controller za rutiranje eksternih zahteva, Heapster za monitoring klastera i analizu performansi, i grafički interfejs Dashboard za pregled i upravljanje resursima klastera. Ova arhitektura omogućava Kubernetes-u da bude fleksibilan, otvoren i prilagodljiv platformi za orkestraciju kontejnera.



Slika 5 - Arhitektura Klastera Kubernetes

6.5 Komponente Kubernetes-a

Kubernetes se sastoji od brojnih komponenti, od kojih svaka igra specifičnu ulogu u celokupnom sistemu. Ove komponente mogu se podeliti u dve kategorije:

- **Čvorovi:** Svaki klaster Kubernetes-a zahteva najmanje jedan radni čvor, koji je kolekcija radnih mašina koje čine čvorove na kojima će naš kontejner biti raspoređen. Radni čvorovi uključuju komponente poput *kubelet*, koji komunicira sa kontrolnom ravni i upravlja životnim ciklusom *pod-ova* na čvoru, i *kube-proxy*, koji održava mrežna pravila za saobraćaj između *pod-ova*.
- **Kontrolna Ravan:** Radni čvorovi i bilo koji podovi unutar njih biće pod kontrolom Kontrolne Ravn. Kontrolna ravan se sastoji od nekoliko ključnih komponenti koje zajedno omogućavaju upravljanje klasterom i raspoređivanje aplikacija.

6.6 Komponente Kontrolne Ravni

Kontrolna ravan je kolekcija različitih komponenti koje nam pomažu u upravljanju opštim zdravljem klastera. Na primer, ako želite da postavite nove podove, uništite podove, skalirate podove itd. U suštini, četiri glavne komponente rade na Kontrolnoj Ravni:

- **Kube-API server:** API server je komponenta Kontrolne Ravni Kubernetes-a koja izlaže Kubernetes API. To je početna kapija za klaster koja sluša ažuriranja ili upite preko *CLI*-a poput *kubectl*. *Kubectl* komunicira sa *API* Serverom kako bi obavestio šta treba da se uradi, kao što su kreiranje *pod-ova* ili brisanje *pod-ova* itd. *API* server validira primljene zahteve, a zatim ih prosleđuje drugim komponentama u sistemu.
- **Kube-Scheduler:** Kada *API* Server primi zahtev za raspoređivanje *pod-ova*, zahtev se prosleđuje Planeru (*Scheduler-u*). On inteligentno odlučuje na kojem čvoru rasporediti pod za bolju efikasnost klastera, uzimajući u obzir faktore kao što su dostupnost resursa i trenutna opterećenja čvorova.
- **Kube-Controller-Manager:** *Kube-controller-manager* je odgovoran za pokretanje kontrolera koji upravljaju različitim aspektima kontrolne petlje klastera. Ovi kontroleri uključuju kontroler replikacije, koji osigurava da željeni broj replika određene aplikacije radi, i kontroler čvora, koji osigurava da su čvorovi pravilno označeni kao "spremni" ili "nespremni" na osnovu njihovog trenutnog stanja.
- **Etc:** To je ključ-vrednost skladište Klastera. Promene stanja Klastera se čuvaju u *etcd*. Deluje kao mozak Klastera jer govori Planeru i drugim procesima o raspoloživim resursima i o promenama stanja klastera. *etcd* je distribuirana baza podataka visokih performansi koja osigurava doslednost podataka u klasteru.

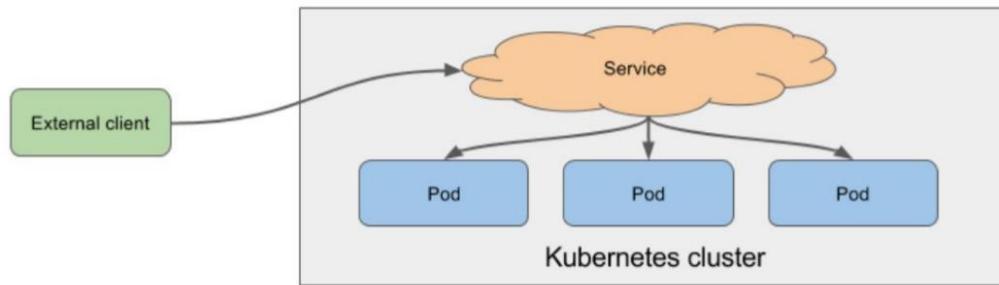
6.7 Mreža Kubernetes-a

Mreža u Kubernetes-u se razlikuje od podrazumevane mreže u Docker-u. Kubernetes omogućava komunikaciju između podova bez obzira na host na kojem se nalaze, dodeljujući svakom podu sopstvenu IP adresu. Ovo eliminiše potrebu za eksplicitnim mapiranjem portova i stvara model sličan virtuelnim mašinama ili fizičkim serverima, što olakšava alokaciju portova, otkrivanje servisa, balansiranje opterećenja i migraciju aplikacija.

Kubernetes definiše mrežni model sa osnovnim zahtevima: svi kontejneri mogu međusobno komunicirati bez korišćenja *NAT-a*; svi čvorovi mogu komunicirati sa svim kontejnerima i obratno, bez *NAT-a*; IP adresa koju kontejner vidi je ista kao ona koju drugi vide. Ovo omogućava fleksibilnost u izboru implementacija mrežnih rešenja, ali zahteva ispunjenje osnovnih zahteva za pravilno funkcionisanje.

Da bi se olakšala interakcija sa podovima, Kubernetes uvodi koncept "Servisa". Servis identifikuje određenu komponentu koja može biti implementirana od strane više podova. Svaki servis dobija IP adresu i port koji ostaju nepromenjeni, omogućavajući klijentima da komuniciraju sa podovima bez obzira na njihovu fizičku lokaciju. Servis takođe omogućava balansiranje opterećenja između više podova.

Kubernetes koristi *Kube Proxy* za upravljanje saobraćajem i preusmeravanje zahteva ka odgovarajućim podovima. Dodatno, sistem može koristiti DNS za rešavanje imena servisa i Ingress Controller za rutiranje spoljašnjih zahteva. Ovaj mrežni model je omogućio Kubernetes-u da se istakne u odnosu na druge platforme, fokusirajući se na jednostavnost i fleksibilnost u upravljanju aplikacijama.



Slika 6 - Kubernetes servis prikazan aplikativnim klijentima

7.0 Docker Swarm i Kubernetes u Cloud-u

U današnjem digitalnom dobu, aplikacije postaju sve složenije, a potrebe za skalabilnošću i fleksibilnošću rastu. Cloud orkestracija kontejnera predstavlja ključnu tehnologiju koja omogućava efikasno upravljanje aplikacijama i njihovim komponentama u distribuiranim okruženjima. Orkestracija kontejnera u cloud-u odnosi se na proces automatizacije implementacije, upravljanja, skaliranja i koordinacije aplikacija smeštenih u kontejnere, raspoređenih na više servera ili klastera.

Kubernetes i Docker Swarm su ključni u ekosistemu *cloud-native* tehnologija, predstavljajući vodeće primere elastičnih platformi. Ove platforme funkcionišu kao međusloj, olakšavajući implementaciju i upravljanje prilagođenim, standardizovanim jedinicama za implementaciju. Oni poboljšavaju iskorišćenost resursa omogućavajući deljenje resursa preko računarskih, mrežnih i skladišnih resursa. Oni su ključni za implementaciju principa *cloud-native* aplikacija (CNA), nudeći horizontalnu skalabilnost, elastičnost i otpornost. Oba sistema podržavaju *inkapsulaciju* heterogenih CNA komponenti, omogućavajući standardizovano pakovanje i jednostavnu implementaciju preko različitih cloud okruženja. Ovaj pristup smanjuje zavisnost od dobavljača, iako migracija CNA između različitih platformi, poput Kubernetes-a i Docker Swarm-a, može zahtevati dodatne inženjerske napore [2].

7.1 Prednosti Cloud Orkestracije

7.1.1 Skalabilnost

Skalabilnost je jedan od ključnih aspekata cloud orkestracije kontejnera, omogućavajući aplikacijama da se dinamički prilagođavaju promenama u opterećenju. Orkestracioni alati poput Kubernetes-a pružaju automatsko skaliranje, koje se odnosi na sposobnost sistema da automatski povećava ili smanjuje broj instanci aplikacija u zavisnosti od trenutnih potreba.

Ovo se postiže putem horizontalnog skaliranja, gde se dodaju nove instance aplikacija, i vertikalnog skaliranja, gde se povećavaju resursi postojećih instanci. Na primer, *Kubernetes Horizontal Pod Autoscaler* (HPA) omogućava automatsko povećanje broja podova (pods) kada se detektuje povećano opterećenje, i smanjenje broja podova kada opterećenje opadne [2].

Docker Swarm takođe može imati automatsko skaliranje, ali je implementacija ovakvih rešenja znatno kompleksnija u poređenju sa Kubernetes-om. Ova fleksibilnost omogućava organizacijama da efikasno koriste resurse, smanjujući nepotrebne troškove i osiguravajući optimalne performanse aplikacija. Ova fleksibilnost omogućava organizacijama da efikasno koriste resurse, smanjujući nepotrebne troškove i osiguravajući optimalne performanse aplikacija [3].

7.1.2 Visoka dostupnost

Visoka dostupnost u cloud aplikacijama je ključni aspekt koji obezbeđuju alati za orkestraciju poput Kubernetesa i Docker Swarma. Kubernetes osigurava kontinuiranu dostupnost putem server *klasteringa*, gde više servera zajednički upravlja radnim opterećenjima. U slučaju kvara servera, Kubernetes pre raspodeljuje radno opterećenje na druge servere u klasteru, sprečavajući zastoje i održavajući kontinuitet usluga. Ovakav pristup pruža otpornost na razne scenarije kvara, uključujući padove aplikacija, hardverske kvarove, pa čak i potpune prekide rada na lokaciji. Slično tome, Docker Swarm, takođe podržava visoku dostupnost putem svog mehanizma *klasteringa*, omogućavajući uslugama da ostanu dostupne čak i ako pojedinačni čvorovi zakažu. I Kubernetes i Docker Swarm koriste tehnologiju kontejnerizacije, pružajući efikasnu izolaciju resursa i dosledan učinak aplikacija u različitim okruženjima. Ove karakteristike omogućavaju Kubernetes-u i Docker Swarm-u da obezbede visoku dostupnost, osiguravajući da cloud aplikacije ostanu pouzdane i dostupne u različitim uslovima [1][2].

7.1.3 Optimizacija resursa

Efikasno korišćenje resursa je još jedan ključni aspekt cloud orkestracije kontejnera, koji omogućava organizacijama da optimizuju svoje operativne troškove. Orkestracioni alati omogućavaju dinamičko prilagođavanje resursa trenutnim potrebama aplikacija, čime se izbegava prekomerna potrošnja resursa i smanjuju troškovi. Na primer, Kubernetes omogućava korisnicima da definišu resursne zahteve i ograničenja za svaki pod, obezbeđujući da aplikacije koriste samo onoliko resursa koliko im je zaista potrebno. Ova funkcionalnost omogućava optimizaciju kapaciteta i smanjuje potrebu za prekomernim resursima, što rezultira značajnim uštedama.

Kontekstualno upravljanje resursima je još jedan aspekt optimizacije resursa, gde orkestracioni alati omogućavaju dinamičku alokaciju resursa na osnovu trenutnih poslovnih prioriteta i potreba. Ovo omogućava organizacijama da efikasno upravljaju svojim resursima i obezbede da kritične aplikacije dobijaju prioritet u korišćenju resursa. Ovakva fleksibilnost i prilagodljivost čine cloud orkestraciju kontejnera ključnim elementom u modernim IT okruženjima, omogućavajući organizacijama da optimizuju svoje operacije i postignu bolje performanse svojih aplikacija.

7.2 Interakcija IaaS, PaaS i SaaS sa Docker Swarm i Kubernetes

Kada pričamo o cloud-u, ne možemo izostaviti IaaS, PaaS i SaaS, kao ni način na koji oni interaguju sa alatima za orkestraciju kontejnera. Ove tri ključne kategorije cloud usluga pružaju različite nivoe upravljanja resursima i aplikacijama, omogućavajući korisnicima da biraju između potpune kontrole nad infrastrukturom i gotovih rešenja za razvoj i implementaciju aplikacija. U ovom detaljnom pregledu istražujemo upotrebu, implementaciju, prednosti i izazove Docker Swarm i Kubernetes platformi unutar IaaS, PaaS i SaaS modela, uz primere iz stvarnog sveta [1][2][3].

7.2.1 Infrastructure as a Service (IaaS)

Pregled IaaS-a: IaaS pruža virtualizovane računalne resurse preko interneta, uključujući virtualne mašine (VM-ove), skladište i mrežu. Korisnici su odgovorni za upravljanje operativnim sistemima, middleware-om i aplikacijama. IaaS platforme uključuju *AWS EC2*, *Google Compute Engine*, *Microsoft Azure VM-ove* i druge.

Implementacija sa Docker Swarm:

- **Postavljanje Klastera:** Docker Swarm može biti implementiran na *VM-ovima* koje pružaju IaaS platforme. Tipična postavka uključuje Swarm *manager* čvor i više *worker* čvorova. Manager čvor upravlja zadacima orkestracije, dok *worker* čvorovi pokreću aplikacije u kontejnerima.
- **Mreža:** Swarm pruža ugrađenu *overlay* mrežu za otkrivanje servisa i komunikaciju između kontejnera preko čvorova. Podržava i ulazno i interno balansiranje opterećenja, distribuirajući saobraćaj preko replika servisa.
- **Primer Upotrebe:** Kompanija za e-trgovinu može koristiti Docker Swarm na *AWS EC2* za implementaciju skalabilne arhitekture mikroservisa. Kompanija može upravljati implementacijom kontejnera, skaliranjem i ažuriranjima uz minimalan infrastrukturni trošak.

Implementacija sa Kubernetes:

- **Postavljanje Klastera:** Kubernetes može biti postavljen korišćenjem alata kao što su *kubeadm*, *kops*, ili upravljane usluge kao što je *Google Kubernetes Engine (GKE)*.
- **Upravljanje Resursima:** Kubernetes koristi apstrakcije kao što su Podovi, *Deployment-i* i Servisi za upravljanje aplikacijama u kontejnerima. Podržava napredno planiranje, samoizlečenje i funkcije auto-skaliranja.
- **Mreža i Skladište:** Kubernetes pruža *Container Network Interface (CNI)* za upravljanje mrežom i *Persistent Volumes (PVs)* za skladište. Integrira se sa IaaS rešenjima za blok skladište kao što su *Amazon EBS* ili *Google Persistent Disks*.
- **Primer Upotrebe:** *Fintech startup* može koristiti Kubernetes na *Google Cloud's Compute Engine* za implementaciju platforme za trgovanje zasnovane na mikroservisima. Kubernetes-ovo auto-skaliranje i balansiranje opterećenja osiguravaju visoku dostupnost i nisko kašnjenje.

Prednosti:

- **Skalabilnost:** I Swarm i Kubernetes mogu dinamički skalirati usluge na osnovu potražnje, optimizujući korišćenje resursa.
- **Fleksibilnost:** Korisnici imaju potpunu kontrolu nad infrastrukturom, omogućavajući prilagođavanje i optimizaciju.
- **Efikasnost Troškova:** *Pay-as-you-go* modeli pomažu u upravljanju troškovima, posebno sa funkcijama kao što je *Kubernetes Cluster Autoscaler*.

Izazovi:

- **Kompleksnost:** Upravljanje infrastrukturom i slojem orkestracije zahteva značajnu stručnost.
- **Sigurnost:** Korisnici su odgovorni za obezbeđivanje *VM-ova*, mreža i aplikacija, što može biti izazovno.

7.2.2 Platform as a Service (PaaS)

Pregled PaaS-a: PaaS pruža platformu koja omogućava programerima da razvijaju, implementiraju i upravljaju aplikacijama bez brige o osnovnoj infrastrukturi. Nudi *runtime* okruženja, razvojne alate, baze podataka i više. Primeri uključuju *Google App Engine*, *Microsoft Azure App Service* i *Heroku*.

Implementacija sa Docker Swarm:

- **Usluge Kontejnera:** Docker Swarm može biti integrisan u PaaS okruženje kako bi pružio orkestraciju kontejnera kao uslugu. PaaS provajderi mogu nuditi Docker Swarm klastere kao upravljanoj uslugu, apstrahujući osnovnu infrastrukturu od korisnika.
- **Tok Posla Programera:** Programeri mogu implementirati aplikacije koristeći Docker slike, koristeći Swarm-ove funkcije otkrivanja servisa i balansiranja opterećenja. PaaS platforme mogu automatizovati skaliranje i ažuriranja, pojednostavljujući razvojni ciklus.
- **Primer Upotrebe:** Pružalac SaaS-a može koristiti Docker Swarm na PaaS platformi kao što je *Heroku* za isporuku aplikacija u kontejnerima korisnicima. Platforma upravlja skaliranjem i održavanjem, omogućavajući pružaocu da se fokusira na razvoj aplikacija.

Implementacija sa Kubernetes:

- **Upravljanje Kubernetes Usluge:** PaaS ponude često uključuju upravljane Kubernetes usluge, kao što su *Google Kubernetes Engine (GKE)*, *Azure Kubernetes Service (AKS)* i *Amazon Elastic Kubernetes Service (EKS)*. Ove usluge apstrahuju složenost upravljanja klasterima, pružajući potpuno upravljano okruženje.
- **Upravljanje Aplikacijama:** Kubernetes u PaaS okruženjima podržava *CI/CD pipeline*, monitoring i logging. Korisnici mogu implementirati aplikacije koristeći *Helm charts* ili *Kubernetes manifeste*, koristeći funkcije kao što su *Ingress* kontroleri za napredno rutiranje i *TLS* terminaciju.
- **Primer Upotrebe:** Kompanija za *streaming* medija može koristiti *GKE* za upravljanje mikroservisnom arhitekturom svoje platforme za *streaming*. Kubernetesova nativna podrška za servisne mreže kao što je *Istio* može poboljšati vidljivost, sigurnost i upravljanje saobraćajem.

Prednosti:

- **Produktivnost Programera:** PaaS platforme pojednostavljuju proces implementacije, omogućavajući programerima da se fokusiraju na kod umesto na infrastrukturu.
- **Upravljanje Usluge:** Funkcije kao što su auto-skaliranje, monitoring i pravljenje kopija često su dostupne u osnovnoj ponudi.
- **Integracija:** PaaS platforme često se integrišu sa drugim cloud uslugama, kao što su baze podataka i skladište, pojednostavljujući razvojni proces.

Izazovi:

- **Vendor Lock-in:** Korisnici mogu biti ograničeni na usluge i konfiguracije koje nudi PaaS provajder.
- **Ograničena Kontrola:** Iako PaaS apstrahuje upravljanje infrastrukturom, takođe ograničava mogućnosti prilagođavanja i optimizacije.

7.2.3 Software as a Service (SaaS)

Pregled SaaS-a: SaaS pruža potpuno funkcionalne softverske aplikacije preko interneta. Provajder upravlja infrastrukturom, platformom i aplikacijom, nudeći korisnicima pristup putem web pretraživača ili *API-ja*. Primeri uključuju *Google Workspace*, *Salesforce* i *Dropbox*.

Implementacija sa Docker Swarm:

- **Arhitektura Mikroservisa:** Docker Swarm može se koristiti za implementaciju SaaS aplikacija kao zbir mikroservisa. Svaki mikroservis može biti nezavisno skaliran, ažuriran i upravljan, nudeći fleksibilnost i modularnost.
- **Otkriće Servisa i Balansiranje Opterećenja:** Swarm-ovo ugrađeno otkrivanje servisa i balansiranje opterećenja osiguravaju da se zahtevi efikasno usmeravaju ka odgovarajućim kontejnerima. Ovo je ključno za održavanje performansi i pouzdanosti u *multi-tenant* SaaS aplikacijama.
- **Primer Upotrebe:** Pružalac SaaS-a koji nudi *CRM* softver može koristiti Docker Swarm za upravljanje *multi-tenant* arhitekturom, izolirajući podatke korisnika i osiguravajući skalabilnost.

Implementacija sa Kubernetes:

- **Multi-tenantnost:** Kubernetes pruža funkcije kao što su *Namespaces* i *Network Policies*, koje se mogu koristiti za implementaciju *multi-tenantnosti* u SaaS aplikacijama. Ovo omogućava izolaciju i sigurnost između različitih korisnika.
- **CI/CD Pipeline:** Kubernetes se integriše sa CI/CD alatima kao što su *Jenkins*, *GitLab CI* i drugi, omogućavajući automatizovane implementacije i povratke. Ovo je kritično za SaaS provajdere kako bi brzo implementirali nove funkcije i ažuriranja.
- **Serverless Computing:** Kubernetes može takođe ugostiti *serverless* funkcije koristeći okvire kao što je Knative, omogućavajući SaaS provajderima da ponude *Function as a Service (FaaS)* mogućnosti. Ovo omogućava događajima vođeno, troškovno efikasno izvršenje zadataka.
- **Primer Upotrebe:** Platforma za e-učenje može koristiti Kubernetes za upravljanje svojom mikroservisnom arhitekturom, sa funkcijama kao što su *HPA* i *VPA* koje osiguravaju efikasno korišćenje resursa i minimalno vreme prekida.

Prednosti:

- **Skalabilnost i Performanse:** Kubernetes-ove napredne mogućnosti planiranja i upravljanja resursima osiguravaju optimalne performanse i skalabilnost, što je ključno za SaaS aplikacije sa promenljivim obrascima korišćenja.
- **Sigurnost i Usklađenost:** Kubernetes pruža funkcije kao što su *Role-Based Access Control (RBAC)* i *Pod Security Policies*, koje su ključne za obezbeđivanje *multi-tenant* okruženja.
- **Kontinuirana Implementacija:** Kubernetes podržava *blue-green* implementacije, *canary releases* i druge strategije implementacije, omogućavajući kontinuiranu isporuku funkcija uz minimalan rizik.

Izazovi:

- **Operativna Kompleksnost:** Upravljanje SaaS aplikacijom na Kubernetes-u zahteva stručnost u upravljanju aplikacijama i infrastrukturom, kao i razumevanje Kubernetes ekosistema.
- **Upravljanje Troškovima:** Dinamična priroda alokacije resursa u Kubernetes-u može dovesti do nepredvidivih troškova, što zahteva pažljivo praćenje i optimizaciju.

7.3 Kubernetes i Docker Swarm u trenutnim cloud okruženjima

U brzo evoluirajućem pejzažu *cloud computinga*, veliki provajderi cloud usluga poput *Amazon Web Services (AWS)*, *Microsoft Azure* i *Google Cloud* integrisali su platforme za orkestraciju kontejnera u svoje ponude. Ove platforme omogućavaju efikasno upravljanje aplikacijama u kontejnerima, obezbeđujući skalabilnost, pouzdanost i jednostavnost implementacije.

7.3.1 Amazon Web Services (AWS)

Kubernetes na AWS-u: Amazon Elastic Kubernetes Service (EKS)

Amazon EKS je AWS-ova upravljana Kubernetes usluga, koja nudi potpuno upravljanu Kubernetes kontrolnu ravan. EKS pojednostavljuje implementaciju, skaliranje i upravljanje Kubernetes klasterima, omogućavajući korisnicima da se fokusiraju na svoje aplikacije, a ne na osnovnu infrastrukturu [10][11].

- **Integracija sa AWS uslugama:** *EKS* se besprekorno integriše sa drugim AWS uslugama, kao što su *Amazon EC2* za računске resurse, *Amazon RDS* za baze podataka i *Amazon S3* za skladištenje objekata. Takođe podržava *AWS Identity and Access Management (IAM)* za sigurnu autentifikaciju i autorizaciju.
- **Mreža i balansiranje opterećenja:** *EKS* koristi *AWS VPC CNI* dodatak za upravljanje mrežom, omogućavajući Kubernetes Podovima da imaju istu IP adresu unutar *VPC* kao i *EC2 instance*. Takođe se integriše sa *Elastic Load Balancing (ELB)* za distribuciju saobraćaja, podržavajući i *Classic Load Balancers* i *Application Load Balancers (ALB)*.
- **Primer upotrebe:** Servis za *stream-ovanje* medija može koristiti *EKS* za upravljanje svojom mikroservisnom arhitekturom, osiguravajući skalabilnost i visoku dostupnost tokom perioda visokog saobraćaja.

Docker Swarm na AWS-u

Iako Docker Swarm nije upravljana usluga na AWS-u, može se implementirati na *EC2* instancama. Korisnici mogu kreirati Swarm klaster *provisioniranjem EC2* instanci i instaliranjem Docker-a.

- **Skalabilnost:** Swarm može skalirati usluge povećanjem broja replika, koje mogu biti raspoređene na više *EC2* instanci.
- **Balansiranje opterećenja:** Docker Swarm pruža nativno balansiranje opterećenja, distribuirajući dolazne zahteve među dostupnim replikama servisa. Takođe podržava upotrebu eksternih *load balancer-a* kao što je *AWS ALB* za napredno upravljanje saobraćajem.

7.3.2 Microsoft Azure

Kubernetes na Azure-u: Azure Kubernetes Service (AKS)

Azure Kubernetes Service (AKS) je Azure-ova upravljana Kubernetes ponuda, koja pruža potpuno upravljano Kubernetes kontrolnu ravan. AKS pojednostavljuje implementaciju, upravljanje i operacije Kubernetes-a, integrišući se sa Azure-ovim paketom usluga [12][13].

- **Azure integracija:** AKS se integriše sa *Azure Active Directory* za kontrolu pristupa zasnovanu na ulogama (RBAC), *Azure Monitorom* za *logging* i metriku, i *Azure DevOps-om* za *CI/CD pipeline*. Takođe podržava *Azure Kubernetes Service Virtual Nodes*, koji koriste *Azure Container Instances* za *serverless* Kubernetes podove.
- **Mreža i skladište:** AKS podržava osnovno i napredno umrežavanje, uključujući *Azure CNI* i *Kubenet*. Za skladište, integriše se sa *Azure Disks* i *Azure Files*, pružajući opcije za trajno skladištenje aplikacija.
- **Primer upotrebe:** Firma za finansijske usluge može koristiti AKS za upravljanje aplikacijama u kontejnerima, koristeći *Azure-ove* sertifikate za usklađenost i sigurnosne funkcije za rukovanje osetljivim podacima.

Docker Swarm na Azure-u

Docker Swarm može biti implementiran na Azure Virtualne Mašine (VMs). Korisnici mogu *provisionirati VM-ove* i postaviti Docker Swarm *klaster*, upravljajući aplikacijama u kontejnerima.

- **Upravljanje uslugama:** Swarm-ove usluge i mogućnosti skaliranja mogu se upravljati putem Docker komandi, sa *Azure-ovim load balancer-ima* i mrežnim sigurnosnim grupama koje pružaju dodatnu kontrolu nad saobraćajem i sigurnošću.
- **Integracija sa Azure uslugama:** Iako nije toliko integrisan kao AKS, Docker Swarm i dalje može koristiti *Azure-ove* usluge, kao što su *Azure Blob Storage* za skladištenje podataka i *Azure Load Balancer* za distribuciju saobraćaja.

7.3.3 Google Cloud Platform (GCP)

Kubernetes na GCP-u: Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) je potpuno upravljana Kubernetes usluga na *GCP-u*. GKE nudi pojednostavljeno iskustvo za implementaciju, upravljanje i skaliranje Kubernetes klastera.

- **Integracija sa GCP uslugama:** GKE se integriše sa *Google Cloud-ovim* uslugama kao što su *Cloud Storage*, *BigQuery* i *Stackdriver* za monitoring i *logging*. Podržava *Identity and Access Management (IAM)* za sigurnu kontrolu pristupa i integriše se sa *Google Cloud-ovim marketplace-om* za jednostavnu implementaciju rešenja treće strane.
- **Mreža i balansiranje opterećenja:** GKE pruža nativnu podršku za globalni *load balancer Google Clouda*, omogućavajući besprekornu distribuciju saobraćaja preko više regiona. Takođe nudi napredne mrežne funkcije kao što su *VPC-native* klasteri i *Network Policy* podrška.
- **Primer upotrebe:** Platforma za e-trgovinu može koristiti GKE za upravljanje svojim *backend* servisima, osiguravajući pouzdanost i skalabilnost tokom događaja sa velikim saobraćajem, poput Crnog petka.

Docker Swarm na GCP-u

Docker Swarm može biti implementiran na *Google Compute Engine (GCE)* instancama. Iako *GCP* ne nudi upravljanu Docker Swarm uslugu, korisnici mogu ručno postaviti Swarm klaster.

- **Otkriće usluga i balansiranje opterećenja:** Swarm-ove native funkcije otkrivanja usluga i balansiranja opterećenja mogu biti pojačane globalnim *load balancerom GCP-a*, nudeći robusne mogućnosti upravljanja saobraćajem.
- **Skaliranje i upravljanje:** Korisnici mogu ručno skalirati Swarm usluge ili putem skripti za automatizaciju, koristeći *GCP-ove* funkcije *autoskaliranja VM-ova*.

7.3.4 Ostali provajderi cloud usluga i hibridna cloud okruženja

IBM Cloud Kubernetes Service

IBM Cloud Kubernetes Service nudi upravljano Kubernetes rešenje sa integracijom u *IBM-ove* cloud usluge, kao što su *Watson* za *AI* mogućnosti i *IBM Cloud Databases*. Podržava *multi-zone* klastere za visoku dostupnost i oporavak od katastrofe.

Red Hat OpenShift

OpenShift, baziran na Kubernetes-u, nudi platformu za *enterprise-grade* Kubernetes. Integrira se sa raznim cloud provajderima, uključujući *AWS*, *Azure* i *GCP*, i podržava hibridne cloud implementacije. *OpenShift* dodaje funkcije prijateljske za developere, kao što su *source-to-image builds* i *CI/CD pipeline*.

Hibridni Cloud i Multi-Cloud

Kubernetes i Docker Swarm su takođe ključni u hibridnim i *multi-cloud* strategijama. Alati kao što su *Rancher* i *OpenShift* olakšavaju upravljanje višestrukim klasterima preko različitih cloud provajdera i *on-premises* okruženja. *Kubernetesove* federacijske mogućnosti omogućavaju ujedinjeno upravljanje i oporavak od katastrofe preko klastera.

7.4 Monitoring i load balancing praktican primer

7.4.1 Monitoring

Monitoring je ključna komponenta u razvoju i održavanju softverskih sistema, naročito u kontekstu skalabilnosti, gde omogućava precizno praćenje performansi, stabilnosti i zdravlja aplikacija i infrastrukture. Kao što je pomenuto u paragrafu o skalabilnosti, mogućnost dinamičkog prilagođavanja broja instanci aplikacija zahteva stalno praćenje ključnih metrika kako bi se pravovremeno reagovalo na promene u opterećenju. Monitoring se koristi za prikupljanje podataka o radu aplikacija, uključujući brzinu odziva, upotrebu resursa, greške i druge bitne metrike. Ovi podaci omogućavaju programerima da identifikuju i rešavaju probleme pre nego što postanu kritični, što je posebno važno za održavanje visoke dostupnosti i optimalnih performansi aplikacija.

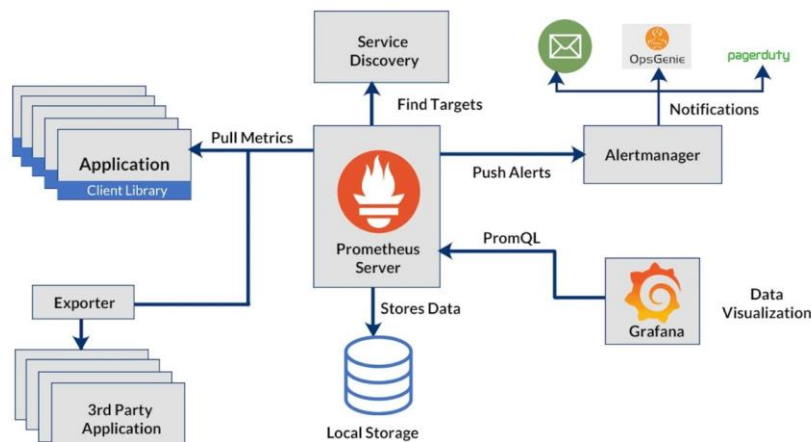
Kroz korišćenje različitih alata za monitoring, kao što su Prometheus, Grafana ili drugi, programeri mogu postaviti alarme i notifikacije. To im omogućava da brzo reaguju na nepredviđene situacije, minimiziraju zastoje u radu sistema i obezbede kontinuiranu dostupnost usluga. Monitoring takođe pomaže u detaljnom podešavanju sistema i optimizaciji performansi, čime se doprinosi sveukupnoj efikasnosti i stabilnosti softverskih rešenja.

7.4.1.1 Prometheus

Prometheus je moćan *open-source* sistem za nadgledanje i upozoravanje, dizajniran za prikupljanje i obradu metrika iz različitih izvora u distribuiranim sistemima. Razvijen od strane *SoundCloud-a*, Prometheus se odlikuje visokom skalabilnošću i mogućnošću prikupljanja vremenski označenih podataka iz različitih izvora, što ga čini idealnim za praćenje performansi aplikacija, infrastrukture i drugih sistema. Osnovne funkcionalnosti Prometheus-a uključuju prikupljanje metrika putem HTTP *pull* modela, fleksibilan upitni jezik *PromQL* za obradu i analizu podataka, kao i mogućnost definisanja pravila za alerting, što omogućava automatsko obaveštavanje korisnika o potencijalnim problemima.

Prometheus koristi vremenski označene podatke, koji se sastoje od naziva metrike, vrednosti i opcionalnih labela, omogućavajući detaljno filtriranje i agregaciju podataka. Metrike se obično prikupljaju od strane *exporter-a*, koji predstavljaju klijente dizajnirane za prikupljanje specifičnih metrika iz aplikacija ili sistema. Ovi *exporter-i* su lako dostupni za mnoge popularne tehnologije, uključujući Docker, Kubernetes, baze podataka i mnoge druge sisteme.

Zahvaljujući svojoj modularnosti i bogatoj ekosistemu dodataka, Prometheus je postao standardni alat za nadgledanje u modernim *DevOps* i *SRE* timovima. Integracija sa drugim alatima, poput Grafane za vizualizaciju podataka, dodatno poboljšava korisničko iskustvo i omogućava jednostavnije praćenje i analizu performansi sistema.



Slika 7 - Arhitektura Prometheus-a

7.4.1.2 Podešavanje Prometheus na Docker Wwarm-u

Podešavanje Prometheus-a na Docker Swarm-u uključuje nekoliko ključnih koraka, koji zajedno omogućavaju efikasno prikupljanje i analizu metrika iz različitih servisa u vašem okruženju. Ovi koraci uključuju pripremu konfiguracione datoteke, definisanje Docker servisa i pokretanje sistema na Swarm-u. U nastavku ćemo detaljno objasniti svaki od ovih koraka, pružajući dublje razumevanje potrebnih koraka i njihovih funkcionalnosti.

1. Priprema konfiguracione datoteke za Prometheus

Konfiguraciona datoteka za Prometheus, obično nazvana *prometheus.yml*, je srce sistema za prikupljanje metrika. Ova datoteka definiše kako i odakle će Prometheus prikupljati podatke. U njoj se postavljaju globalne postavke poput intervala prikupljanja (*scrape_interval*), što određuje koliko često će Prometheus posećivati definisane izvore metrika.

Takođe, konfiguraciona datoteka sadrži sekciju *scrape_configs*, gde se definišu izvori podataka koje Prometheus treba da nadgleda. Svaki izvor je opisan putem *job_name* i *static_configs* parametara. *Job_name* je naziv zadatka koji olakšava identifikaciju izvora, dok *static_configs* sadrži listu ciljeva (*targets*) - obično URL adresa servisa koji izlaže metrike. Na ovaj način, možete pratiti različite komponente vašeg sistema, kao što su aplikacije, baze podataka, ili Docker servisi.

2. Definisanje Docker Swarm servisa

Nakon pripreme konfiguracione datoteke, sledeći korak je definisanje Docker servisa za Prometheus u Docker Swarm okruženju. Ovaj korak zahteva kreiranje *docker-compose.yml* datoteke, koja opisuje servise koji će se pokretati unutar klastera.

U ovoj datoteci, potrebno je specificirati Docker sliku koju će Prometheus koristiti, kao i konfiguraciju zapremina za perzistentno skladištenje podataka i mrežnu povezanost. Montiranje konfiguracione datoteke u kontejner je ključno, jer omogućava Prometheus-u pristup postavkama definisanim u *prometheus.yml*. Takođe, potrebno je mapirati *port-eve*, kako bi Prometheus-ov web interfejs bio dostupan van kontejnera, obično na portu 9090.

3. Pokretanje servisa na Docker Swarm-u

Kada su sve datoteke spremne, sledeći korak je pokretanje Prometheus-a u Docker Swarm okruženju. Ovo se postiže korišćenjem komande *docker stack deploy*, koja omogućava postavljanje i pokretanje servisa unutar definisanog *stack-a*. *Stack* predstavlja grupu povezanih servisa koji se upravljaju kao celina, olakšavajući orkestrasiju i upravljanje.

Tokom ovog procesa, Docker Swarm će automatski rasporediti servise na dostupne čvorove unutar klastera, osiguravajući optimalno korišćenje resursa. Nakon pokretanja, Prometheus web interfejs će biti dostupan na definisanom portu, omogućavajući pregled metrika i konfiguraciju *alerting* pravila.

4. Povezivanje sa dodatnim servisima

Jedna od ključnih prednosti Prometheus-a je mogućnost proširivanja sistema putem različitih *exporter-a* tj. klijenata koji izlažu metrike iz specifičnih aplikacija ili sistema. Na primer, *node-exporter* prikuplja metrike o performansama host mašine, dok *cadvisor* prikuplja podatke o Docker kontejnerima. Ove eksportere je potrebno dodati u konfiguraciju Prometheus-a i pokrenuti kao dodatne servise unutar Docker Swarm-a, omogućavajući centralizovano prikupljanje metrika iz različitih izvora.

7.4.1.3 Podešavanje u našem primeru swarm-a

Prvi korak je kreacija konfiguraciona datoteka za Prometheus.

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: 'myapp'
    dns_sd_configs:
      - names:
        - 'tasks.myapp'
        type: 'A'
        port: 8080
    metrics_path: '/actuator/prometheus'
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
  - job_name: 'swarm-listener'
    static_configs:
      - targets: ['swarm-listener:8080']
  - job_name: 'alert-receiver'
    static_configs:
      - targets: ['alert-receiver:5001']
rule_files:
  - '/etc/prometheus/alert.rules.yml'
```

Globalna podešavanja:

- **scrape_interval: 10s:** Ovo postavlja interval prikupljanja metrika na svakih 10 sekundi, što znači da će Prometheus posećivati sve definisane ciljeve na svakih 10 sekundi da bi prikupio najnovije metrike.

scrape_configs:

- **myapp:**
 - Korišćenje *dns_sd_configs* sa *tasks.myapp* omogućava Prometheus-u da automatski otkriva servise po DNS imenu unutar Swarm-a, ciljani port je 8080. Ova konfiguracija cilja na */actuator/prometheus* putanju, koja je tipično korišćena za Spring Boot aplikacije da izlože metrike.
- **cadvisor:**
 - *static_configs* sa ciljem *cadvisor:8080* omogućava prikupljanje metrika o Docker kontejnerima i hostovima. *CAdvisor* izlaže ove metrike na portu 8080.
- **swarm-listener:**
 - Ovaj posao prikuplja metrike od *swarm-listener* servisa na portu 8080, omogućavajući monitoring promena unutar Docker Swarm-a.
- **alert-receiver:**
 - Definiše ciljeve na *alert-receiver:5001* za prikupljanje podataka o obaveštenjima i alerting sistemu.

rule_files:

- **rule_files:** - */etc/prometheus/alert.rules.yml*: Ova konfiguracija specificira putanju do datoteke sa pravilima za alarme, koja omogućava definisanje pravila za aktiviranje upozorenja na osnovu prikupljenih metrika.

Docker compose

Zatim ćemo servis za prometheus u našem docker compose fajlu. Servis koristi zvaničnu Prometheus Docker sliku. Konfiguracija uključuje montiranje lokalne *prometheus.yml* i *alert.rules.yml* datoteke u kontejner, omogućavajući Prometheus-u pristup potrebnim konfiguracijama. Servis je izložen na portu 9090.

```
prometheus:
  image: prom/prometheus:latest
  volumes:
    - ./prometheus.yml:/etc/prometheus/prometheus.yml
    - ./alert.rules.yml:/etc/prometheus/alert.rules.yml
  ports:
    - "9090:9090"
  networks:
    - monitoring
  deploy:
    restart_policy:
      condition: none
```

7.4.1.4 Podešavanje Prometheus na Kubernetes-u

Podešavanje Prometheus-a na Kubernetes platformi zahteva kreiranje nekoliko ključnih Kubernetes skripti koje omogućavaju pravilno pokretanje i funkcionisanje Prometheus servisa unutar klastera. Ove skripte uključuju *Deployment*, *Service*, *ConfigMap*, i odgovarajuće uloge i dozvole za pristup.

Deployment

Deployment skripta je osnova za pokretanje Prometheus aplikacije u Kubernetes klasteru. *Deployment* definiše broj replika aplikacije, što omogućava skaliranje i visoku dostupnost. On takođe specificira Docker sliku koja će biti korišćena za kreiranje podova, zajedno sa konfiguracijom resursa kao što su CPU i memorija. *Volume mount*-ovi su takođe ključni deo konfiguracije, jer omogućavaju pristup potrebnim konfiguracionim datotekama i *perzistentnom* skladištu podataka. Kroz *deployment*, možemo lako upravljati verzijama aplikacije, ažuriranjima i održavanjem.

Service

Service skripta se koristi za izlaganje Prometheus aplikacije unutar i izvan klastera. Definiše stabilnu DNS adresu i *load balancing* za podove koji pokreću Prometheus, omogućavajući korisnicima i drugim aplikacijama pristup metrikama koje Prometheus prikuplja. Ova skripta obično koristi *ClusterIP*, *NodePort*, ili *LoadBalancer* tipove servisa, u zavisnosti od potreba za dostupnošću.

ConfigMap

ConfigMap je skripta koja se koristi za skladištenje konfiguracionih podataka koje podovi koriste. U slučaju Prometheus-a, *ConfigMap* se koristi za skladištenje konfiguracione datoteke (*prometheus.yml*) koja definiše pravila za prikupljanje metrika, izvore metrika, i pravila za obaveštavanje. Ova konfiguracija se može lako ažurirati bez potrebe za promenom aplikacionog koda, što čini *ConfigMap* fleksibilnim alatom za menadžment konfiguracija.

Dodatne skripte

Osim glavnih skripti, podešavanje Prometheus-a na Kubernetes-u često uključuje dodatne skripte kao što su:

- **ServiceAccount:** Omogućava Prometheus-u da se autentifikuje u klasteru i pristupi potrebnim resursima.
- **ClusterRole i ClusterRoleBinding:** Definišu dozvole i pristupne politike za Prometheus, uključujući mogućnost čitanja metrika iz različitih izvora unutar klastera.
- **PersistentVolume i PersistentVolumeClaim:** Koriste se za obezbeđivanje perzistentnog skladištenja podataka, što je važno za očuvanje istorijskih podataka i stabilnost sistema.

Ove skripte zajedno omogućavaju stabilno i efikasno funkcionisanje Prometheus-a unutar Kubernetes klastera, omogućavajući prikupljanje, skladištenje i analizu metrika iz različitih izvora. Ova infrastruktura podržava sveobuhvatno nadgledanje i analitiku, što je ključno za održavanje performansi i stabilnosti aplikacija u proizvodnom okruženju.

7.4.1.5 Podešavanje u našem primeru Kubernetes-a

Prometheus-Deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  labels:
    app: prometheus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus
      containers:
        - name: prometheus
          image: prom/prometheus
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: config-volume
              mountPath: /etc/prometheus/
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
      volumes:
        - name: config-volume
          configMap:
            name: prometheus-config
```

Prometheus-Service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
spec:
  type: NodePort
  ports:
    - port: 9090
      targetPort: 9090
  selector:
    app: prometheus
```

Cluster-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - pods
      - services
      - endpoints
      - namespaces
    verbs:
      - get
      - list
      - watch
```

Cluster-role-binding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: default
```

Service-account.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
  namespace: default
```


Configuration-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
    scrape_configs:
      - job_name: 'kubernetes'
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
            action: keep
            regex: true
          - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
            action: replace
            target_label: __metrics_path__
            regex: (.+)
          - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
            action: replace
            target_label: __address__
            regex: ([^:]+)(?::\d+)?(\d+)?
            replacement: $1:$2
          - source_labels: [__meta_kubernetes_namespace]
            action: replace
            target_label: kubernetes_namespace
          - source_labels: [__meta_kubernetes_pod_name]
            action: replace
            target_label: kubernetes_pod_name
```

Opis funkcionalnosti

Prometheus Deployment YAML Skripta

- **Replicas:** Definisan je jedan pod za Prometheus, što je tipično za okruženja koja ne zahtevaju visoku dostupnost ili skaliranje.
- **Containers:** Specifikacija za Prometheus kontejner uključuje Docker sliku *prom/prometheus*, izlaganje porta 9090, i *mount-ovanje* volumena sa konfiguracionim datotekama. Argument `--config.file=/etc/prometheus/prometheus.yml` definiše putanju do glavne konfiguracione datoteke.
- **Volumes:** Koristi *ConfigMap* za skladištenje konfiguracione datoteke, omogućavajući Prometheus-u da pristupi potrebnim podešavanjima.

Prometheus Service YAML Skripta

- **Type:** *NodePort* tip servisa omogućava izlaganje Prometheus-a na fiksnoj portu na svim čvorovima klastera, što olakšava pristup sa spoljne strane.
- **Ports:** Port 9090 je izložen kako na servisu, tako i na kontejneru, omogućavajući pristup web interfejsu Prometheus-a.

Prometheus ConfigMap YAML Skripta

- **Data:** *ConfigMap* sadrži konfiguracionu datoteku *prometheus.yml* koja definiše globalne postavke i scrape konfiguracije. U ovom slučaju, *scrape_interval* je postavljen na 15 sekundi, što određuje frekvenciju prikupljanja metrika.
- **Kubernetes SD Configs:** Korišćenje *kubernetes_sd_configs* omogućava automatsko otkrivanje podova unutar klastera. *relabel_configs* sekcija koristi se za prilagođavanje metrika, uključujući filtriranje po specifičnim anotacijama i dodavanje labela kao što su *kubernetes_namespace* i *kubernetes_pod_name*.

7.4.2 Grafana

Podaci koje prikuplja Prometheus često nisu pregledni sami po sebi, posebno kada se radi o velikom broju metrika koje dolaze iz različitih izvora. Takođe, može se desiti da se koriste različiti alati za prikupljanje metrika, što može dodatno zakomplikovati proces nadgledanja i analize. U takvim situacijama, Grafana dolazi kao rešenje.

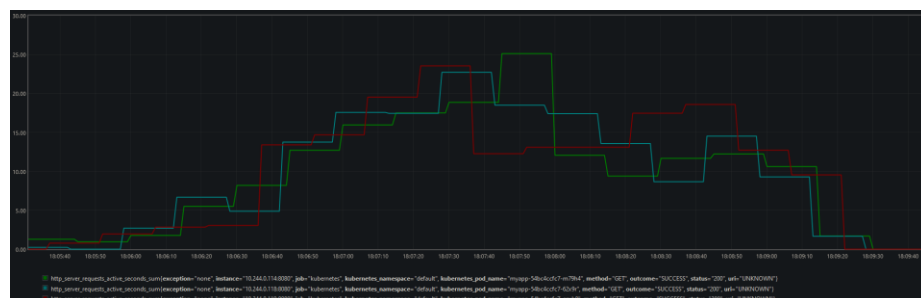
Grafana je moćan *open-source* alat za analizu i vizualizaciju podataka, koji omogućava korisnicima da stvaraju interaktivne i prilagodljive *dashboard-e* za praćenje performansi sistema i aplikacija. Pomoću Grafane, korisnici mogu lako kreirati grafikone, tabele i druge vizualizacije koje omogućavaju dublje razumevanje podataka i brže donošenje odluka.

Jedna od glavnih prednosti Grafane je njena sposobnost integracije sa različitim izvorima podataka, uključujući baze podataka, cloud servise i sisteme za nadgledanje. Takođe, Grafana podržava napredne funkcije poput upozorenja (*alerting*), koje korisnicima omogućavaju da postave pragove za obaveštenja i automatske reakcije na promene u sistemu. Iako se u ovom delu nećemo previše zadržavati na Grafani, ona predstavlja ključnu komponentu u ekosistemu za monitoring i vizualizaciju, omogućavajući lakše praćenje i analizu podataka koje prikuplja Prometheus.

Takođe je iznenađujuće laka za postavljanje, jer se jednostavno integriše kao zasebna komponenta koja zatim prikuplja sve informacije putem svog interfejsa. Na primer, kada koristimo Prometheus kao izvor metrika, dovoljno je da u Grafani definišemo novi izvor podataka i postavimo ga da prikuplja podatke sa Prometheus-ovog porta.



Slika 8 - swarm_http_server_requests_active_seconds_sum



Slika 9 - kubernetes_http_server_requests_active_seconds_sum

Slike iznad (slika 8,9) su vizuelni prikaz broja poziva ka aplikaciji tokom testiranja sa *k6* alatom, prikazano pomoću Prometheus-a.

7.5 Automatsko skaliranje

Automatsko skaliranje je ključna funkcionalnost u modernim distribuiranim sistemima koja omogućava dinamičko prilagođavanje broja instanci aplikacije u zavisnosti od trenutnog opterećenja. Ovaj proces se temelji na metrikama prikupljenim od **monitoring alata**, kao što su Prometheus. Automatsko skaliranje omogućava optimalno korišćenje resursa, smanjenje troškova i povećanje dostupnosti aplikacija, automatskim povećanjem broja instanci kada je opterećenje visoko i smanjenjem kada je opterećenje nisko.

U Kubernetes-u, implementacija automatskog skaliranja je znatno olakšana zahvaljujući ugrađenim alatima kao što su *Horizontal Pod Autoscaler (HPA)*, koji automatski prilagođava broj replika podova na osnovu definisanih metrika. Kubernetes pruža fleksibilan i moćan sistem za upravljanje skaliranjem, koji je dobro integrisan sa monitoring alatima, omogućavajući brzu reakciju na promene u opterećenju.

Nasuprot tome, u Docker Swarm-u, implementacija automatskog skaliranja može biti nešto složenija, jer zahteva dodatne alate ili skripte za prilagođavanje broja servisa. Iako Docker Swarm pruža osnovne mogućnosti za skaliranje, on se oslanja na eksterni monitoring i skripte za automatsko prilagođavanje, što može zahtevati više manualnog rada i konfiguracije.

U sledećim delovima, detaljno ćemo objasniti našu praktičnu implementaciju automatskog skaliranja, koristeći metrike prikupljene od monitoring alata, kako bismo osigurali optimalne performanse i efikasnost naših aplikacija u oba okruženja.

7.5.1 Automatsko skaliranje u Docker Swarm-u

U Docker Swarm-u, budući da nema ugrađenu podršku za automatsko skaliranje, koristimo spoljašnje alate i prilagođene skripte za monitoring i reagovanje na promene u opterećenju. U našem projektu, implementirali smo rešenje koje uključuje korišćenje *Alertmanager-a* i dodatnih prilagođenih skripti.

Alertmanager prati metrike prikupljene od strane *Prometheus-a* i drugih monitoring alata. Kada metrika pređe definisane pragove, *Alertmanager* generiše upozorenje koje se prosleđuje na željenu lokaciju. U našem slučaju, ta upozorenja se šalju na spoljašnji alat, uključujući Prometheus i dodatni prilagođeni program za nadgledanje.

Kada *Python* skripta, koju smo napisali specifično za ovu svrhu, detektuje primljeno upozorenje, ona preduzima odgovarajuće akcije za automatsko skaliranje. Skripta može povećati ili smanjiti broj replika servisa unutar Docker Swarm-a, u zavisnosti od trenutnog opterećenja i potreba sistema. Ova implementacija omogućava efikasno korišćenje resursa i održavanje performansi aplikacija, prilagođavajući se dinamičkim promenama opterećenja.

7.5.1.1 Postavljanje menadžera upozorenja

Alertmanager.yml

```
global:
  resolve_timeout: 5m
route:
  receiver: 'webhook'
receivers:
- name: 'webhook'
  webhook_configs:
  - url: 'http://alert-receiver:5001/alerts'
    send_resolved: true
```

Alertmanager-service.yml

```
services:
  alertmanager:
    image: prom/alertmanager:latest
    volumes:
      - ./alertmanager.yml:/etc/alertmanager/alertmanager.yml
    ports:
      - "9093:9093"
    networks:
      - monitoring
    deploy:
      restart_policy:
        condition: none
```

Alert.rules.yml

```
groups:
- name: auto-scaling
  rules:
  - alert: HighRequestRate
    expr: sum(rate(http_server_requests_seconds_count{job="myapp"}[1m])) > 5
    for: 20s
    labels:
      severity: warning
      service_name: myapp
    annotations:
      summary: "High request rate detected"
      description: "The request rate for the application has exceeded 5 requests per second for more than 1 minute."
- name: example
  rules:
  - alert: AppStarted
    expr: vector(1)
    for: 10s
    labels:
      severity: info
    annotations:
      summary: "App Start Alert"
      description: "This alert is triggered every 10 seconds to indicate the app has started."
```

Alert-receiver u docker-compose kao eksterni alat za hvatanje upozorenja

```
alert-receiver:
  build: ./alert-receiver
  image: shus300/alert-receiver
  ports:
    - "5001:5001"
  networks:
    - monitoring
```

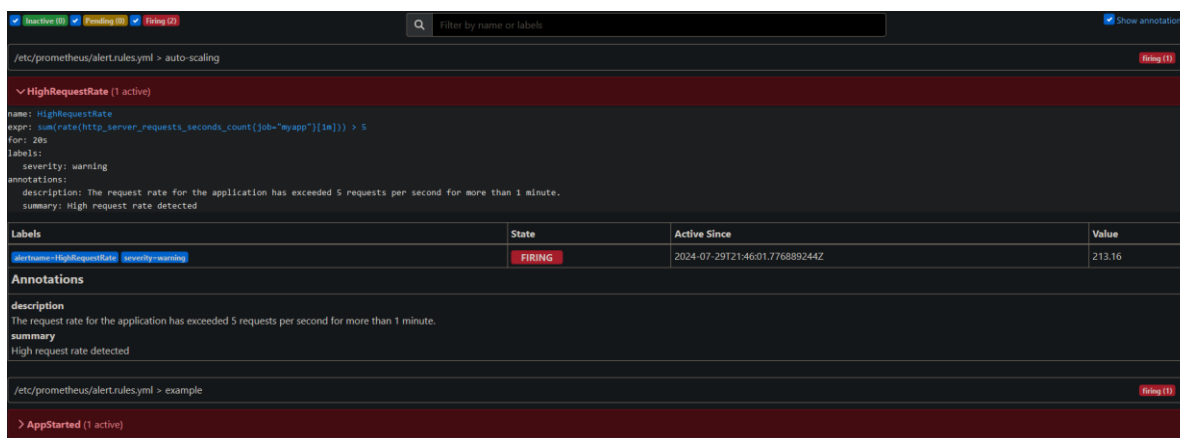
Opis funkcionalnosti

- **Globalne postavke:** U sekciji *global* definisano je koliko dugo *Alertmanager* treba da čeka pre nego što označi upozorenje kao rešeno (*resolve_timeout: 5m*).
- **Route i Receivers:** Glavna ruta (*route*) definiše osnovnu konfiguraciju za prosleđivanje upozorenja. U ovom slučaju, sva upozorenja se šalju na *receiver* pod imenom *webhook*, koji je konfigurisan da šalje podatke na URL *http://alert-receiver:5001/alerts*. Ovaj URL predstavlja prilagođeni *endpoint* koji se koristi za prijem upozorenja i dodatne obrade.
- **Service:** Definiše Docker servis za *Alertmanager*. Koristi sliku *prom/alertmanager:latest* i mapira lokalnu konfiguracionu datoteku *alertmanager.yml* unutar kontejnera. Servis je izložen na portu 9093 i povezan na mrežu *monitoring* kako bi imao pristup metrikama.

Pravila i grupacije upozorenja

Alert Groups sekcija definiše grupe pravila za upozorenja. U vašem primeru, postoje dve grupe:

1. **auto-scaling:**
 - **HighRequestRate:** Ova pravila se aktiviraju kada je brzina zahteva prema aplikaciji (*http_server_requests_seconds_count*) veća od 5 zahteva po sekundi u poslednjem minutu. Upozorenje traje 20 sekundi i ima labelu *severity: warning*. Opisuje da je otkrivena visoka brzina zahteva i pruža dodatne informacije putem *summary* i *description* polja.
2. **example:**
 - **AppStarted:** Ovo je primer upozorenja koje se aktivira svakih 10 sekundi (*expr: vector(1)*), jednostavno kao indikator da je aplikacija pokrenuta. Ima labelu *severity: info* i služi kao demonstracija ili testiranje.



Slika 10 - Upozorenje prikazano na prometheus-u

```

WAITING FOR ALERTS:
Received alert:
Alert Name: HighRequestLatency
Severity: warning
Method: GET
URI: /api/instant
Summary: High request latency detected
Description: Request latency for endpoint /api/instant is above 0.5 seconds for more than 1 minute.

Received alert:
Alert Name: HighRequestLatency
Severity: warning
Method: GET
URI: /api/delayed
Summary: High request latency detected
Description: Request latency for endpoint /api/instant is above 0.5 seconds for more than 1 minute.

Received alert:
Alert Name: HighRequestLatency
Severity: warning
Method: GET
URI: /api/superDelayed
Summary: High request latency detected
Description: Request latency for endpoint /api/instant is above 0.5 seconds for more than 1 minute.

```

Slika 11 - Upozorenje prikazano na eksternoj aplikaciji za hvatanje upozorenja

7.5.1.2 Monitoring upozorenja i reagovanje

Nakon što je *Alertmanager* pravilno podešen, moguće je dodatno proširiti sistem za monitoring upozorenja uključivanjem drugih alata za hvatanje i obradu upozorenja, pored *Prometheus-a*.

U našem slučaju, kada *Alertmanager* detektuje da su zadati uslovi ispunjeni, kao što je visoka količina zahteva prema aplikaciji, generiše upozorenje koje se prosleđuje na naš prilagođeni *endpoint*. Nakon što upozorenje stigne do sistema, pokreće se *Python* skripta specijalno dizajnirana za automatsko skaliranje. Ova skripta analizira upozorenje i na osnovu definisanih pravila automatski prilagođava broj replika servisa unutar Docker Swarm-a.

```

import requests
from flask import Flask, request, jsonify

app = Flask(__name__)
@app.route('/', methods=['POST'])
def scale_service():
    alert = request.json
    for alert in alert['alerts']:
        service_name = alert['labels']['service_name']
        if alert['status'] == 'firing':
            scale_up(service_name)
        elif alert['status'] == 'resolved':
            scale_down(service_name)
    return jsonify({"status": "success"}), 200

def scale_up(service_name):
    print(f"Scaling up service: {service_name}")
    scale(service_name, 1) # Adjust scale factor as needed

def scale_down(service_name):
    print(f"Scaling down service: {service_name}")
    scale(service_name, -1) # Adjust scale factor as needed

def scale(service_name, scale_factor):
    url =
    f"http://localhost:2375/services/{service_name}/update"
    headers = {'Content-Type': 'application/json'}
    current_replicas = get_current_replicas(service_name)
    data = {
        "Name": service_name,
        "TaskTemplate": {
            "ContainerSpec": {
                "Image": "shus300/myapp:latest"
            }
        },
        "Mode": {
            "Replicated": {
                "Replicas": current_replicas + scale_factor
            }
        }
    }
    response = requests.post(url, headers=headers,
    json=data)
    print(response.json())

```

Ova Python skripta implementira automatsko skaliranje Docker Swarm servisa na osnovu upozorenja koje prima od *Alertmanager-a*. Skripta koristi *Flask framework* za kreiranje web servera koji prima HTTP POST. Kada stigne upozorenje, skripta analizira njegovo stanje (*firing* ili *resolved*) i odgovarajuće poziva funkcije za skaliranje servisa *scale_up* ili *scale_down*. Ove funkcije pozivaju glavnu funkciju *scale*, koja koristi *Docker Remote API* da prilagodi broj replika specifičnog servisa u Docker Swarm-u, šaljući POST zahtev sa ažuriranim brojem replika na URL *http://localhost:2375/services/{service_name}/update*. Ova konfiguracija omogućava dinamičko prilagođavanje resursa aplikacije u zavisnosti od trenutnog opterećenja, optimizujući performanse i efikasnost sistema

7.5.2 Automatsko skaliranje u Kubernetes-u

Automatsko skaliranje u Kubernetes-u je značajno jednostavnije zahvaljujući ugrađenom alatu *Horizontal Pod Autoscaler (HPA)*. *HPA* automatski prilagođava broj replika podova u zavisnosti od trenutnog opterećenja i potreba aplikacije, koristeći metrike prikupljenih iz Prometheus-a ili drugih izvora. Ovaj mehanizam omogućava optimalno korišćenje resursa, smanjenje troškova i održavanje stabilnosti sistema, bez potrebe za složenim spoljnim skriptama ili alatima.

Horizontal Pod Autoscaler (HPA) je Kubernetes kontroler koji dinamički skalira broj replika podova tokom *deployment-a* na osnovu prikupljenih metrika. *HPA* kontinuirano prati definisane metrike i koristi ih za donošenje odluka o povećanju ili smanjenju broja replika. Na primer, *HPA* može biti konfigurisan da povećava broj replika kada prosečno korišćenje CPU-a pređe određeni prag, ili da smanjuje broj replika kada opterećenje padne.

```
C:\Users\Shus\Programming\IdeaProjects\Master\SwarmKubernetesCompare\kubernetes>kubectl get hpa
NAME          REFERENCE          TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
myapp-hpa     Deployment/myapp    cpu: 6%/20%   1         10        1         4m
```

Slika 12 - HPA low cpu usage

```
C:\Users\Shus\Programming\IdeaProjects\Master\SwarmKubernetesCompare\kubernetes>kubectl get hpa
NAME          REFERENCE          TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
myapp-hpa     Deployment/myapp    cpu: 98%/20%   1         10        10        7m49s
```

Slika 13 - HPA high cpu usage



Slika 14 - Auto skaliranje od HPA

Prve dve slike prikazuju rezultate komande *kubectl get hpa*, gde se vidi kako *HPA* automatski prilagođava broj replika podova na osnovu trenutnog opterećenja CPU-a. Prva slika pokazuje nizak opterećenje CPU-a (6%) gde se održavalo na 1 replici, dok druga slika pokazuje visoko korišćenje CPU-a (98%) što je dovelo do skaliranja na 10 replika.

Treća slika je grafikon iz Grafane, koji vizualizuje broj aktivnih podova tokom vremena. Ovi prikazi jasno demonstriraju kako *HPA* reaguje na promene u opterećenju, omogućavajući optimalno korišćenje resursa i stabilnost aplikacije.

8.0 Poređenje

Kubernetes i Swarm nisu konkurentski alati. Oni omogućavaju isto krajnje rešenje, ali u zavisnosti od raznih faktora korisnik bi trebao da zna koji alat da koristi. Kubernetes je bolje koristiti u sledećim slučajevima:

- Starije i stabilnije rešenje sa boljim monitoring opcijama
- Za razvoj kompleksnijih aplikacija
- Veliki broj sistema u klasteru

Sa druge strane, Swarm je bolji za:

- Korisnike koji imaju samo osnovni nivo znanja o Docker-u i ne žele previše vremena da potroše na instalaciju i konfiguraciju klastera
- Razvoj manje kompleksnih aplikacija

Swarm pruža jednostavno rešenje koje omogućava korisniku da brzo kreira i pokrene svoj klaster, dok Kubernetes podržava veće i kompleksnije zahteve. Swarm je popularniji među programerima koji preferiraju brzo startovanje aplikacije na klasteru, bez puno konfiguracije, dok je Kubernetes korišćeniji u produkcionim rešenjima od strane visoko profilisanih internet kompanija koje su napravile popularne servise koje koriste veliki broj ljudi.

8.1 Kategorijsko poređenje

8.1.1 Instalacija i Podešavanje

Instalacija Kubernetes-a je složenija u odnosu na Docker Swarm. Kubernetes zahteva preuzimanje i instalaciju *kubectl*, što može varirati u zavisnosti od operativnog sistema. Kubernetes može biti postavljen na različitim platformama, ali instalacija uključuje konfiguraciju IP adresa klastera, definisanje uloga nodova, i podešavanje dodatnih paketa poput *Minikube* ili *MicroK8s*. Takođe, upravljanje autentifikacijom i autorizacijom, kao i podešavanje *role-based access control (RBAC)*, su kritični aspekti koje treba konfigurirati.

Docker Swarm, s druge strane, ima jednostavniji proces instalacije i konfiguracije. Nakon instalacije *Docker Engine*-a, dovoljno je dodeliti IP adrese hostovima, otvoriti potrebne portove i postaviti menadžer i radne *nodeove*. Swarm režim se lako aktivira pomoću jednostavne komande *docker swarm init*. Swarm koristi isti Docker *CLI* koji je već poznat korisnicima, što dodatno olakšava upravljanje i konfiguraciju klastera. [6]

8.1.2 Skalabilnost

Kubernetes nudi napredne opcije za automatsko skaliranje koje mogu automatski prilagoditi broj instanci u zavisnosti od trenutnog opterećenja. Horizontalno autoskaliranje u Kubernetes-u omogućava dodavanje ili uklanjanje *pod*-ova na osnovu definisanih metrika poput CPU ili memorijskog opterećenja. Ovaj sistem omogućava fino podešavanje resursa u realnom vremenu, što je ključno za aplikacije sa visokim zahtevima za performanse. Kubernetes takođe podržava *Cluster Autoscaler* koji automatski podešava broj *nodova* u klasteru kako bi se prilagodio promenama u resursima koje koriste podovi.

Docker Swarm omogućava brzu i jednostavnu skalabilnost, ali nema ugrađenu podršku za automatsko skaliranje na nivou Kubernetes-a. Docker Swarm skalira instance na zahtev putem jednostavnih komandi kao što su *docker service scale*, što je korisno za manje i jednostavnije aplikacije. Iako Swarm ne podržava automatsko skaliranje bazirano na metrikama, omogućava brzo skaliranje servisa na više *nodova*. [6]

8.1.3 Balansiranje Opterećenja

Docker Swarm nudi automatsko balansiranje opterećenja između kontejnera unutar klastera, što omogućava besprekornu komunikaciju između kontejnera. Swarm koristi ugrađeni *load balancer* koji automatski raspoređuje saobraćaj između dostupnih instanci servisa. Ovo omogućava korisnicima da brzo i lako distribuiraju opterećenje bez dodatne konfiguracije.

Kubernetes nema ugrađeno automatsko balansiranje opterećenja, ali omogućava integraciju eksternih alata za balansiranje opterećenja. Kubernetes koristi servise i *Ingress* resurse za upravljanje saobraćajem. Servisi omogućavaju pristup *pod*-ovima preko stabilnih IP adresa ili DNS imena, dok *Ingress* omogućava sofisticirano upravljanje HTTP i HTTPS saobraćajem koristeći eksterni *load balancer*. Integracija sa alatima kao što su *Traefik* ili *NGINX* može pružiti dodatne mogućnosti za balansiranje opterećenja i upravljanje saobraćajem. [15]

8.1.4 Visoka Dostupnost

Oba alata pružaju visok nivo dostupnosti, ali na različite načine. Kubernetes koristi napredne strategije zakazivanja i repliciranja usluga da bi osigurao visok nivo dostupnosti. Na primer, Kubernetes automatski preusmerava saobraćaj sa neispravnih *pod*-ova i zamenjuje ih novim koristeći mehanizme kao što su *disruption budgets* i *replicaset* kontroleri. Kubernetes takođe podržava *multi-master* arhitekturu koja omogućava visoku dostupnost kontrolnog plana.

Docker Swarm koristi kontrolu dostupnosti na nivou menadžerskih *nodova* i omogućava lako dupliranje mikroservisa. Swarm koristi *Raft* konsenzusni algoritam za održavanje stanja klastera, što omogućava da se menadžeri lako oporavljaju u slučaju kvara. Docker Swarm menadžeri mogu premestiti radni *node* na drugi resurs u slučaju kvara hosta, čime se obezbeđuje kontinuitet servisa. [15]

8.1.5 Mreža

Docker Swarm pruža ugrađene mrežne mogućnosti koje su jednostavne za upotrebu, uključujući *overlay* mreže koje omogućavaju kontejnerima da komuniciraju preko više hostova. Swarm koristi ugrađeni *DNS* servis za ime-resoluciju unutar klastera, što omogućava lako povezivanje servisa.

Kubernetes nudi složeniji mrežni model sa različitim opcijama poput *Calico*, *Flannel* i *Weave*. Ove mrežne *plugin*-ove omogućavaju napredne mrežne funkcionalnosti kao što su mrežne politike, enkapsulacija saobraćaja i sigurnost na mrežnom nivou. Kubernetes podržava koncept *Network Policies* koje omogućavaju definisanje detaljnih pravila za saobraćaj između *pod-ova*, što je ključno za aplikacije sa visokim zahtevima za sigurnost.

8.1.6 Komanda-linijski Alati i GUI

Docker Swarm koristi standardni Docker CLI za upravljanje klasterom, što olakšava korišćenje za korisnike koji su već upoznati sa Docker-om. Sve operacije se mogu obaviti putem poznatih Docker komandi, što smanjuje vreme učenja i pojednostavljuje administraciju.

Kubernetes zahteva instalaciju *kubectl* alata za rad sa klasterom, koji pruža široke mogućnosti za upravljanje *deployment-ima*, servisima, podovima i drugim resursima unutar Kubernetes okruženja. *kubectl* omogućava napredne funkcionalnosti kao što su deklarativno upravljanje konfiguracijama i kompleksne operacije nad klasterom. Što se tiče *GUI* podrške, Kubernetes dolazi sa ugrađenim *dashboard-om* koji pruža korisnički interfejs za upravljanje klasterom i monitoring stanja resursa. Docker Swarm zahteva integraciju sa trećim partijama kao što su *Portainer*, *Dockstation*, *Swarmpit*, i *Shipyard* za pružanje *GUI* funkcionalnosti. [14][15]

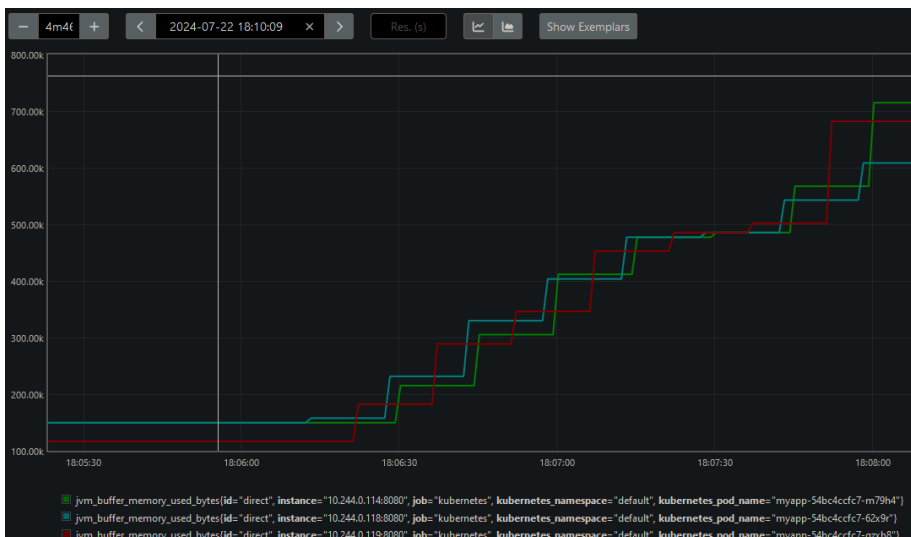
8.2 Poređenje po metrikama

8.2.1 Poređenje u količini iskorišćene buffer memorije



Slika 16 -

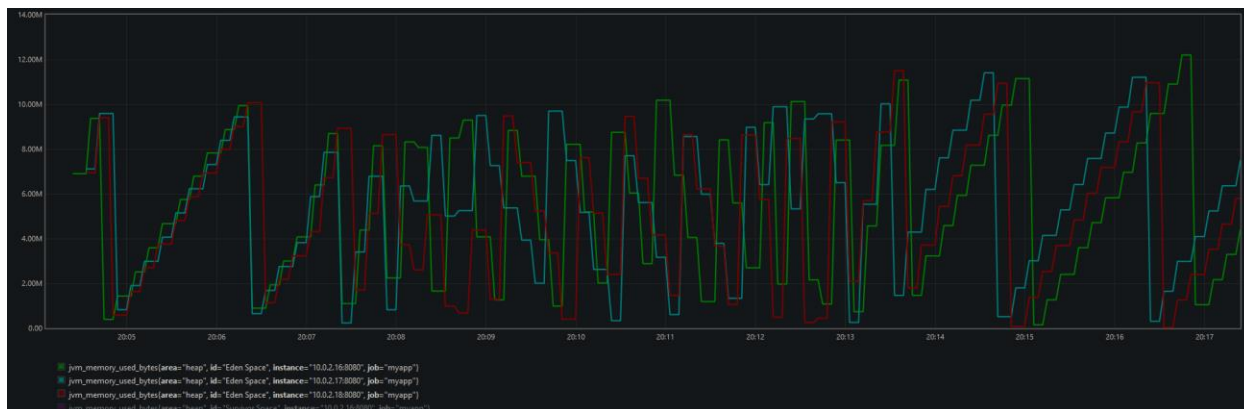
kubernetes_jvm_buffer_memory_used_bytes



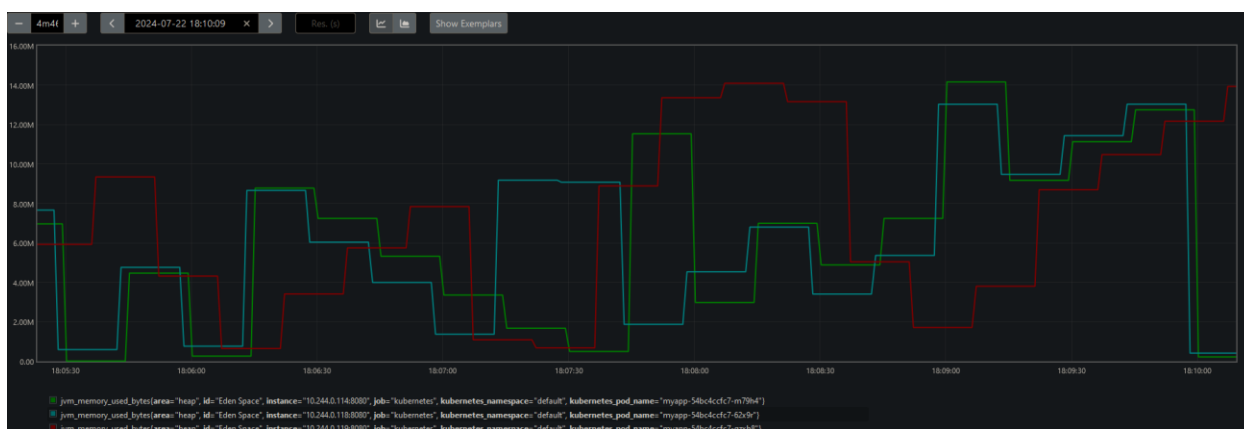
Slika 15 - kubernetes_jvm_buffer_memory_used_bytes

Instancije Kubernetes-a pokazuju veće korišćenje *JVM* memorijskog *buffer-a* u poređenju sa Docker Swarm-om, dostižući vrhunac od oko 600k bajtova naspram 400k bajtova kod Swarm-a. Dok oba sistema pokazuju postepeni porast upotrebe memorije, Kubernetes ima izraženije korake i varijacije na početku, što ukazuje na dinamički obrazac alokacije. Swarm se brže stabilizuje sa minimalnim fluktuacijama, sugerišući konzistentniju potrošnju memorije. Na većem obimu, ovi trendovi će verovatno ostati isti, s tim da napredne funkcije Kubernetes-a mogu dovesti do efikasnije upotrebe resursa, ali i veće potrošnje memorije zbog dinamičkih prilagođavanja. Swarm može zadržati niži *overhead* i predvidljive obrasce korišćenja, iako se mogu pojaviti ograničenja u skalabilnosti. Ove razlike ističu različite mogućnosti upravljanja resursima i performansama svakog sistema za orkestraciju.

8.2.2 Poređenje po upravljanju memorijskog prostora



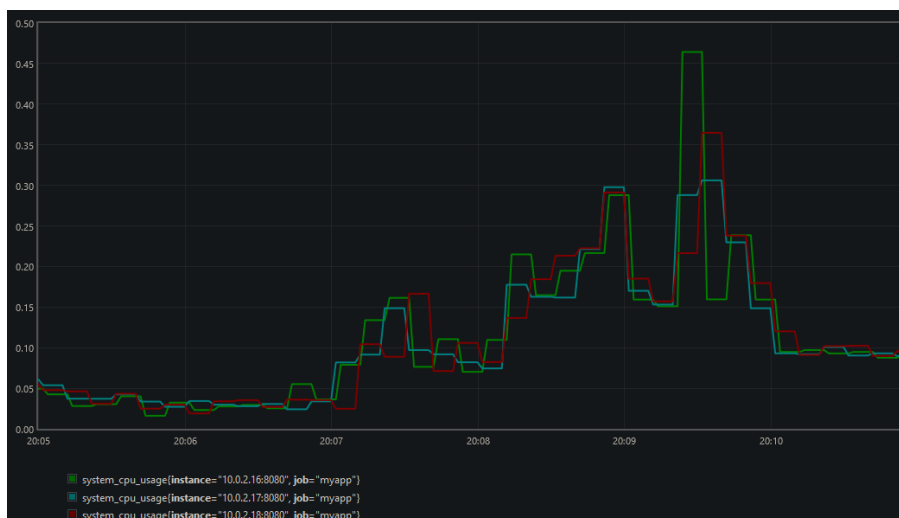
Slika 17 - swarm_jvm_memory_used_bytes



Slika 18 - kubernetes_jvm_memory_used_bytes

Poređenje korišćenja *JVM* memorije u *Eden Space-u* između Docker Swarm-a i Kubernetes-a pokazuje da obe platforme imaju ciklične obrasce, što ukazuje na redovne aktivnosti prikupljanja smeća. *Eden Space* je deo *heap* memorije u *JVM-u* koji služi za kratkotrajno skladištenje novostvorenih objekata pre nego što budu premješteni u druge delove *heap-a* ili uklonjeni tokom prikupljanja smeća. Swarm prikazuje česte usponi i padove, sa korišćenjem memorije koje varira između 0 i 12M, što odražava efikasno recikliranje memorije. Nasuprot tome, Kubernetes pokazuje izraženije vrhove i duže periode višeg korišćenja, sa rasponom do 14M, što ukazuje na veće, ali ređe *deallocacije* memorije. Na većem obimu, ovi trendovi sugerišu da Kubernetes može bolje podneti aplikacije sa stalnim visokim opterećenjem memorije, dok bi Swarm-ovo češće recikliranje moglo dovesti do efikasnijeg upravljanja memorijom, ali bi moglo imati problema sa većim, stalnim zahtevima za memorijom.

8.2.3 Poređenje po količini iskorišćenja procesora



Slika 19 - swarm_system_cpu_usage



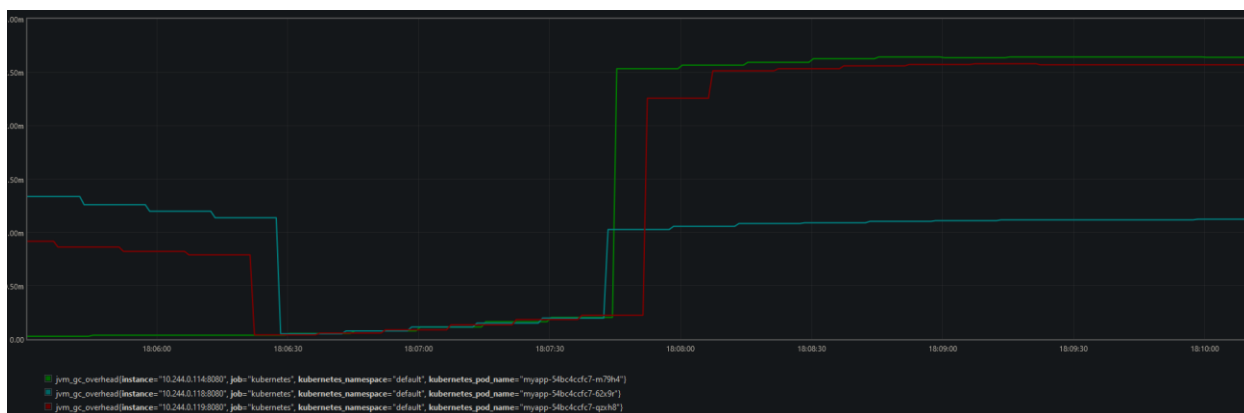
Slika 20 - kubernetes_system_cpu_usage

Poređenje korišćenja CPU resursa između Docker Swarm-a i Kubernetes-a pokazuje da Kubernetes ima veće i dugotrajnije opterećenje CPU-a, sa vrhuncem od oko 0,70 u poređenju sa 0,45 kod Swarm-a. Kubernetes pokazuje konzistentnija perioda visokog opterećenja, dok Swarm ima sporadične skokove sa bržim povratkom na osnovne nivoe. Ovo ukazuje na to da je Kubernetes bolje prilagođen za intenzivne i dugotrajne radne zadatke, dok Swarm može podneti lakše ili promenljive opterećenja. Na većem obimu, ovi trendovi će verovatno ostati isti, pri čemu Kubernetes može bolje upravljati složenijim i resursno zahtevnijim aplikacijama zbog svoje sposobnosti da podnosi veća opterećenja CPU-a. Swarm-ove predvidljive i niže obrasce korišćenja mogli bi dovesti do nižeg *overhead-a*, ali bi mogao imati ograničenja u skaliranju za zadovoljavanje većih zahteva.

8.2.4 Poređenje po GC (garbage collection)



Slika 21 - swarm_jvm_gc_overhead



Slika 22 - kubernetes_jvm_gc_overhead

JVM GC (Garbage Collection) overhead se odnosi na resurse koje *Java Virtual Machine* troši na upravljanje memorijom i čišćenje neupotrebljivih objekata. Poređenje *JVM GC overhead-a* između Docker Swarm-a i Kubernetes-a pokazuje da Swarm počinje sa visokim *overhead-om* (oko 6,8m) koji se naglo smanjuje i stabilizuje na nižim nivoima, što ukazuje na efikasno prikupljanje smeća. Nasuprot tome, Kubernetes počinje sa nižim *overhead-om* (oko 0,5m), ali se stabilizuje na višem nivou, održavajući doslednu potrošnju resursa. Ovo sugerise da Swarm efikasnije smanjuje *overhead* u poređenju sa Kubernetes-ovim stabilnim upravljanjem resursima. Na većem obimu, ovi trendovi će verovatno ostati isti, pri čemu Swarm može ponuditi niži ukupni *GC overhead*, što može biti korisno za aplikacije koje zahtevaju efikasno upravljanje memorijom. Kubernetes-ov konzistentan, ali viši *overhead* može podržavati stabilne performanse aplikacija, ali može dovesti do veće potrošnje resursa.

8.2.5 Poređenje sistemskog opterećenja



Slika 23 - swarm_system_load_average_1m



Slika 24 - kubernetes_system_load_average_1m

Poređenje prosečnog sistemskog opterećenja tokom jednog minuta pokazuje da Swarm ima više i stabilnije početno opterećenje, sa konzistentnim fluktuacijama oko 0.8 do 1.0, što ukazuje na umeren do visok radni zadatak. Opterećenje kod Swarm-a se brzo stabilizuje nakon dostizanja vrhunca, što sugerise efikasno upravljanje radnim zadacima. Nasuprot tome, Kubernetes pokazuje niže početno opterećenje oko 0.2 do 0.3, sa većom varijabilnošću i naglim promenama, odražavajući dinamične varijacije u radnim zadacima. Na većem obimu, ovi obrasci će verovatno ostati isti, pri čemu će Swarm pružiti stabilnije i predvidljivije upravljanje opterećenjem, što ga čini pogodnim za okruženja sa stabilnim radnim zadacima. Kubernetes, sa svojim dinamičnim promenama opterećenja, može biti bolje prilagođen za okruženja sa fluktuirajućim radnim zadacima, ali bi mogao zahtevati robusnije mehanizme za praćenje i prilagođavanje kako bi se efikasno nosio sa varijabilnošću.

9 Praktican Primer

9.1 Docker Swarm

9.1.1 Prvi Korak: Izgradnja Docker Image-a

Docker je platforma koja omogućava razvoj, isporuku i pokretanje aplikacija u izolovanim okruženjima poznatim kao kontejneri. Centralni koncept Docker-a su Docker *image-i* i Docker Hub.

Docker Image je statička datoteka koja sadrži sve potrebne komponente za pokretanje aplikacije, uključujući operativni sistem, aplikacijski kod, *runtime* okruženje, biblioteke i sve druge zavisnosti. Docker *image* je zapravo šablon iz kojeg se stvaraju kontejneri. Svaki put kada pokrenete kontejner, on koristi Docker *image* kao osnovu, čime se osigurava da aplikacija uvek ima sve potrebne komponente za ispravan rad. Docker *image* se kreira na osnovu **Dockerfile-a**, koji je skriptni fajl sa setom uputstava koja Docker koristi za izgradnju *image-a*.

Docker Hub je javni registar za Docker image-e gde korisnici mogu skladištiti i deliti svoje *image-e*. Nakon što napravite Docker *image*, možete ga postaviti na Docker Hub kako bi bio dostupan za preuzimanje i korišćenje na različitim serverima ili klasterima.

Kada je Docker *image* postavljen na Docker Hub, može se koristiti za kreiranje kontejnera. **Kontejner** je instanca Docker *image-a* koja radi izolovano na host mašini. Ova izolacija omogućava konzistentnost i prenosivost aplikacija između različitih okruženja, kao što su razvojno, testno i produkciono okruženje.

Kreacija *image-a* i postavljanje na Docker Hub mogu se izvršiti pomoću sledećih komandi:

- `docker build --file Dockerfile image_name`
- `docker push image_name`

Dockerfile 1: Go Lang Api

U nastavku je primer Dockerfile-a za Go Lang projekat. Ovaj Dockerfile koristi više faza (multistage build) kako bi se optimizovao proces izgradnje i smanjila veličina konačnog *image-a*.

```
# Pin specific version for stability
# Use a multi-stage build to optimize the image
# Use Debian-based Golang image for easier build utilities
FROM golang:1.19-bullseye AS build-base
WORKDIR /app
# Copy go.mod and go.sum to leverage layer caching for dependencies
COPY go.mod go.sum ./
# Use cache mount to speed up installation of existing dependencies
RUN --mount=type=cache,target=/go/pkg/mod \
  --mount=type=cache,target=/root/.cache/go-build \
  go mod download
# Development stage
FROM build-base AS dev
# Install air for hot reload & delve for debugging
RUN go install github.com/cosmtrek/air@latest && \
  go install github.com/go-delve/delve/cmd/dlv@latest
# Copy the entire project for development
COPY . .
# Command to start air for live reload during development
CMD ["air", "-c", ".air.toml"]
# Production build stage
FROM build-base AS build-production
# Add a non-root user for better security
RUN useradd -u 1001 nonroot
# Copy the entire project for production build
COPY . .
# Compile healthcheck binary with static linking
```

```
RUN go build \
  -ldflags="-linkmode external -extldflags -static" \
  -tags netgo \
  -o healthcheck \
  ./healthcheck/healthcheck.go
# Compile the main application binary with static linking
RUN go build \
  -ldflags="-linkmode external -extldflags -static" \
  -tags netgo \
  -o api-golang
# Deployment stage
FROM scratch
# Set gin mode to release for production
ENV GIN_MODE=release
WORKDIR /
# Copy the passwd file for the non-root user
COPY --from=build-production /etc/passwd /etc/passwd
# Copy the healthcheck binary from the build stage
COPY --from=build-production \
  /app/healthcheck/healthcheck healthcheck
# Copy the main application binary from the build stage
COPY --from=build-production /app/api-golang api-golang
# Use the non-root user for running the application
USER nonroot
# Expose the application port
EXPOSE 8080
# Command to run the application
CMD ["/api-golang"]
```

Objašnjenje Dockerfile-a

1. Prva faza: *build-base*
 - Koristimo *golang* bazu sa *Debian bullseye* distribucijom.
 - Postavljamo radni direktorijum na */app*.
 - Kopiramo *go.mod* i *go.sum* datoteke i preuzimamo sve zavisnosti pomoću *go mod download*. Korišćenjem *cache mount-a* ubrzavamo proces preuzimanja.
2. Druga faza: *dev*
 - Instaliramo alate za *hot reload* (*air*) i *debugging* (*delve*).
 - Kopiramo ceo projekat u radni direktorijum.
 - Pokrećemo *air* za *hot reload* tokom razvoja.
3. Treća faza: *build-production*
 - Dodajemo *non-root* korisnika radi sigurnosti.
 - Kopiramo ceo projekat i kompajliramo binarni *fajl api-golang* sa statičkim linkovanjem.
4. Finalna faza: *scratch*
 - Koristimo minimalni *scratch image* za konačnu verziju.
 - Postavljamo *GIN mod* na *release* za produkciju.
 - Kopiramo binarni fajl i postavljamo *non-root* korisnika.
 - Eksponiramo port 8080 i postavljamo komandnu liniju za pokretanje aplikacije.

Dockerfile 2: Node api

```
# Pin specific version for stability
# Use slim version of Node.js for reduced image size
FROM node:19.6-bullseye-slim AS base
# Specify working directory other than the root directory
WORKDIR /usr/src/app
# Copy only package.json and package-lock.json for better layer caching during dependency installation
COPY package*.json ./
# Development stage
FROM base as dev
# Use cache mount to speed up the installation of existing dependencies
RUN --mount=type=cache,target=/usr/src/app/.npm \
  npm set cache /usr/src/app/.npm && \
  npm install
# Copy the entire project for development
COPY . .
# Command to start the development server
CMD ["npm", "run", "dev"]
# Production stage
FROM base as production
# Set NODE_ENV to production
ENV NODE_ENV production
# Install only production dependencies
# Use cache mount to speed up the installation of existing dependencies
RUN --mount=type=cache,target=/usr/src/app/.npm \
  npm set cache /usr/src/app/.npm && \
  npm ci --only=production
# Use a non-root user for better security
USER node
# Copy the healthcheck script with appropriate ownership
COPY --chown=node:node ./healthcheck/ .
# Copy remaining source code AFTER installing dependencies to leverage layer caching
COPY --chown=node:node ./src/ .
# Expose the application port
EXPOSE 3000
# Command to run the application
CMD ["node", "index.js"]
```

Objašnjenje Dockerfile-a

1. Prva faza: base
 - Koristimo node bazu sa *Debian bullseye* distribucijom.
 - Postavljamo radni direktorijum na */usr/src/app*.
 - Kopiramo *package.json* i *package-lock.json* datoteke i instaliramo zavisnosti koristeći *npm cache*.
2. Druga faza: dev
 - Kopiramo ceo projekat u radni direktorijum.
 - Pokrećemo aplikaciju u razvojnom modu koristeći *npm run dev*.
3. Treća faza: *production*
 - Postavljamo *NODE_ENV* promenljivu na *production*.
 - Instaliramo samo produkcijske zavisnosti.
 - Koristimo *non-root* korisnika radi sigurnosti.
 - Kopiramo *healthcheck* skriptu i preostali izvorni kod.
 - Eksponiramo port 3000 i postavljamo komandnu liniju za pokretanje aplikacije.

Dockerfile 3: React front client

```
# Use specific syntax version for better Dockerfile parsing
# Base image for building the application
FROM node:19.4-bullseye AS build
# Specify working directory other than the root directory
WORKDIR /usr/src/app
# Copy only package.json and package-lock.json for better layer caching during dependency installation
COPY package*.json ./
# Use cache mount to speed up the installation of existing dependencies
RUN --mount=type=cache,target=/usr/src/app/.npm \
  npm set cache /usr/src/app/.npm && \
  npm install
# Copy the entire project for building
COPY . .
# Run the build command to compile the application
RUN npm run build
# Use a separate stage for the deployable image
FROM nginxinc/nginx-unprivileged:1.23-alpine-perl
# Copy nginx configuration with --link to avoid breaking cache if the base image changes
COPY --link nginx.conf /etc/nginx/conf.d/default.conf
# Copy the built application from the build stage to the Nginx HTML directory
COPY --link --from=build /usr/src/app/dist/ /usr/share/nginx/html
# Expose the application port
EXPOSE 8080
```

Objašnjenje Dockerfile-a

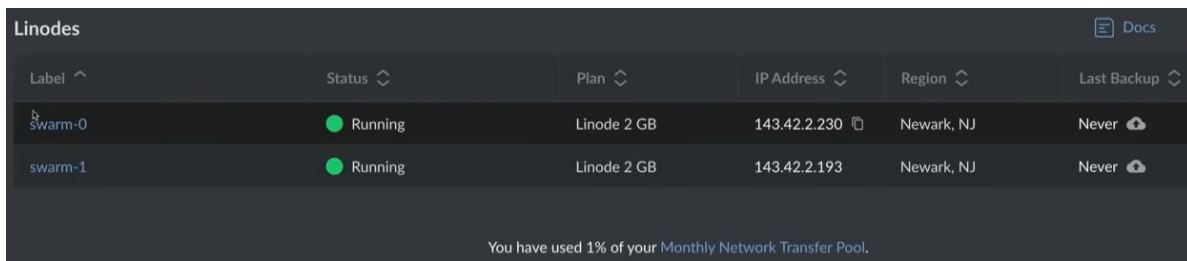
1. Prva faza: *build*
 - Koristimo *node* bazu sa *Debian bullseye* distribucijom.
 - Postavljamo radni direktorijum na */usr/src/app*.
 - Kopiramo *package.json* i *package-lock.json* datoteke i instaliramo zavisnosti koristeći *npm cache*.
 - Kopiramo ceo projekat i kompajliramo *frontend* koristeći *npm run build*.
2. Druga faza: *deployable image*
 - Koristimo *nginx* bazu sa *Alpine* distribucijom.
 - Kopiramo *nginx* konfiguraciju i kompajlirani *frontend-u* odgovarajuće direktorijume.
 - Ekspoziramo port 8080 za pristup aplikaciji.

9.1.2 Drugi Korak: Kreiranje Swarm Nodova

U ovom koraku ćemo se fokusirati na kreiranje i konfiguraciju Swarm *nodova*. Docker Swarm omogućava orkestraciju kontejnera preko više *nodova*, što pruža visoku dostupnost i skalabilnost vaših aplikacija. U ovom primeru ćemo koristiti *Linode Cloud* za kreiranje *nodova*.

Kreiranje Swarm Nodova

Kreirat ćemo više *nodova* koji će biti deo Docker Swarm klastera. U ovom primeru ćemo koristiti dva *noda* jedan kao *manager*, a drugi kao *worker*.



Linodes						Docs
Label ^	Status ^	Plan ^	IP Address ^	Region ^	Last Backup ^	
swarm-0	● Running	Linode 2 GB	143.42.2.230	Newark, NJ	Never	
swarm-1	● Running	Linode 2 GB	143.42.2.193	Newark, NJ	Never	

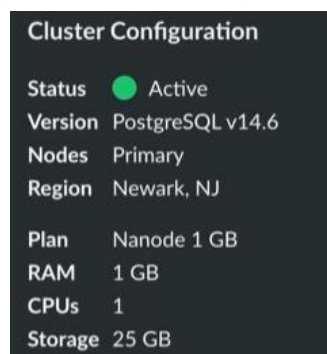
You have used 1% of your [Monthly Network Transfer Pool](#).

Slika 25 - Podignuti docker swarm nodovi

Kreiranje Baze

Prvo, potrebno je kreirati bazu podataka koju će koristiti vaša aplikacija. Baza podataka može biti bilo koja, ali u ovom primeru ćemo koristiti *PostgreSQL*. Nakon što se baza kreira u njen *white-list nodova* koji smeju da joj pristupe ćemo dodati IP adrese naših novo kreiranih.

Takođe lokalno se treba napraviti Docker tajna koja će nositi login informacije, kao *password*, za pristup bazi. Sve ovo se treba dodati lokalno u projektu i terminalu.



Cluster Configuration	
Status	● Active
Version	PostgreSQL v14.6
Nodes	Primary
Region	Newark, NJ
Plan	Nanode 1 GB
RAM	1 GB
CPU's	1
Storage	25 GB

Slika 26 - Podignuta baza

9.1.3 Treci Korak: Podešavanje nodova i swarm-a

Prvo se moramo povezati na *nodove*, što možemo postići korišćenjem *SSH* protokola i komande:

- ***ssh username@IpAddress***
- ***ssh root@143.42.2.230***

Kada uspostavimo sesiju, posto su čvorovi nakon kreacije prazni, prvo moramo da im podesimo okruženje za rad. U ovom slučaju, pošto želimo da ih pokrećemo u swarm modu, moramo da im podesimo lokalne instance Docker-a. To možemo da uradimo preko već napravljene skripte koja se nalazi na <https://get.docker.com/>.

```
root@localhost:~# curl https://get.docker.com | sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 21926  100 21926    0     0  199k      0  0:00:01  0:00:01  0:00:00 202k
# Executing docker install script, commit: c2de0811708b6d9015ed1a2c80f02c9b70c8ce7b
+ sh -c apt-get update -qq >/dev/null
```

Slika 27 - Instalacija dockera na nodu

Da bismo bili sigurni da je sve ispravno izvršeno, možemo pokrenuti jednostavnu naredbu *docker --version*. Ako dobijemo odgovor koji sadrži verziju Docker-a, to znači da je instalacija uspešna.

```
Server: Docker Engine - Community
Engine:
Version:      24.0.1
API version:  1.43 (minimum version 1.12)
Go version:   go1.20.4
Git commit:   463850e
```

Slika 28 - Provera pravilne instalacije dockera

Zatim ćemo na jednom čvoru pokrenuti swarm mod, dok ćemo drugi čvor pridružiti prvom koristeći komandu *docker swarm join --token [master_token]*. U ovom trenutku naš swarm je pokrenut i povezan, ali još uvek ne izvršava ništa jer mu nismo dodelili projekte koje želimo da se izvršavaju.

```
root@localhost:~# docker swarm init
Swarm initialized: current node (ky1nb83oqquljm51semxtrh2l) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5lvpiippmilesrsv4iro4fx8dhxvb8ld8p3ccntdvvousp1oq-0g2bu6cauj18muwqwebx7af8k 143.42.2.230:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Slika 29 - Pokretanje swarm menadžera

```
root@localhost:~# docker swarm join --token SWMTKN-1-5lvpiippmilesrsv4iro4fx8dhxvb8ld8p3ccntdvvousp1oq-0g2bu6cauj18muwqwebx7af8k 143.42.2.230:2377
This node joined a swarm as a worker.
```

Slika 30 - Povezivanje workera za menadžera

```
root@localhost:~# docker node ls
ID                HOSTNAME        STATUS        AVAILABILITY    MANAGER STATUS    ENGINE VERSION
ky1nb83oqquljm51semxtrh2l * localhost      Ready         Active           Leader             24.0.1
op1bum9fnmkxa6ehw2eomjsxe localhost      Ready         Active           Leader             24.0.1
```

Slika 31 - Provera statusa nodova

Radi lakšeg korišćenja, možemo izvršiti naredbu:

- `export DOCKER_HOST="ssh://root@143.42.230"`

Ova komanda će povezati naš lokalni Docker Desktop sa Docker čvorom, omogućavajući nam upravljanje swarm-om preko API hosta swarm-a.

```
> docker node ls
ID                HOSTNAME    STATUS    AVAILABILITY    MANAGER STATUS    ENGINE VERSION
ky1nb83oqquljm51semxtrh2l * localhost    Ready     Active           Leader            24.0.1
op1bum9fnmkxa6ehw2eomjsxe localhost    Ready     Active           -                 24.0.1
```

Slika 32 - Kontrola prebačena lokalnom docker desktop-u

Docker swarm file

Sledeći korak je pokretanje Docker Swarm fajla, koji će preuzeti slike naših projekata i postaviti ih u swarm. Ovaj proces omogućava raspoređivanje i upravljanje našim aplikacijama unutar swarm-a.

Pokrecemo docker swarm file pomocu komnade:

- `docker stack deploy -c docker-swarm.yml example_app`

Najvažniji deo ovog koda je:

- **mode: replicated** - Ova opcija definiše da će se zadatak replicirati na više čvorova. To znači da će se instance naših aplikacija pokrenuti na različitim čvorovima unutar swarm-a, što poboljšava dostupnost i otpornost na greške.
- **replicas: 1** - Ovaj parametar postavlja broj replika, odnosno instanci aplikacije koje će biti pokrenute. U ovom slučaju, jedna instanca aplikacije će biti pokrenuta.
- **update_config: order: start-first** - Ova konfiguracija određuje redosled ažuriranja. Kada se vrši ažuriranje, nova instanca će biti pokrenuta pre nego što se stara zaustavi. To osigurava da nema prekida u radu aplikacije tokom ažuriranja.

```

version: '3.7'
services:
  client-react-nginx:
    image: sidpalas/devops-directive-docker-
course-client-react-nginx:5
    deploy:
      mode: replicated
      replicas: 1
      update_config:
        order: start-first
    init: true
    networks:
      - frontend
    ports:
      - 80:8080
    healthcheck:
      test: ["CMD", "curl", "-f",
"http://localhost:8080/ping"]
      interval: 30s
      timeout: 5s
      retries: 3
      start_period: 10s
    api-node:
      image: sidpalas/devops-directive-docker-
course-api-node:9
      read_only: true
      deploy:
        mode: replicated
        replicas: 1
        update_config:
          order: start-first
      init: true
      environment:
        -
DATABASE_URL_FILE=/run/secrets/database
-url
      configs:
        - db.crt
      secrets:
        - database-url
      networks:
        - frontend

```

```

      - backend
    healthcheck:
      test: ["CMD", "node",
"/usr/src/app/healthcheck.js"]
      interval: 30s
      timeout: 5s
      retries: 3
      start_period: 10s
    api-golang:
      image: sidpalas/devops-directive-docker-
course-api-golang:8
      read_only: true
      deploy:
        mode: replicated
        replicas: 2
        update_config :
          order: start-first
      init: true
      environment:
        -
DATABASE_URL_FILE=/run/secrets/database
-url
      secrets:
        - database-url
      networks:
        - frontend
        - backend
      healthcheck:
        test: ["CMD", "/healthcheck"]
        interval: 30s
        timeout: 5s
        retries: 3
        start_period: 10s
      networks:
        frontend:
        backend:
      secrets:
        database-url:
          external: true
      configs:
        db.crt:
          file: ./db.crt

```

```
> docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
k5vktj6j7r7a	example-app_api-golang	replicated	2/2	sidpalas/devops-directive-docker-course-api-golang:8
grt0v3nabun1	example-app_api-node	replicated	1/1	sidpalas/devops-directive-docker-course-api-node:9
wlyfqclm081o	example-app_client-react-nginx	replicated	1/1	sidpalas/devops-directive-docker-course-client-react-nginx:5

Slika 33 - Prikaz pokrenutog swarma

Sada, ako pogledamo oba *noda*, možemo videti da imaju pokrenute kopije API-a. Ukoliko jedan čvor padne, drugi će preuzeti njegov rad, osiguravajući kontinuitet usluge.

```
> docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS	ENGINE VERSION
ky1nb83oquljm51semxtrh2l *	localhost	Ready	Active	Leader	24.0.1
op1bum9fmmkxa6ehw2eomjsxe	localhost	Ready	Active		24.0.1

```
> docker node ps self
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
i15e3ovyiiu7	example-app_api-golang.2	sidpalas/devops-directive-docker-course-api-golang:8	localhost	Running	Running about a minute ago	
kpovjtnq2r0x	example-app_api-node.1	sidpalas/devops-directive-docker-course-api-node:9	localhost	Running	Running about a minute ago	

```
> docker node ps op1bum9fmmkxa6ehw2eomjsxe
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
l281dda5684h	example-app_api-golang.1	sidpalas/devops-directive-docker-course-api-golang:8	localhost	Running	Running about a minute ago	
jnmxbi4tznpr	example-app_client-react-nginx.1	sidpalas/devops-directive-docker-course-client-react-nginx:5	localhost	Running	Running about a minute ago	

Slika 34 - Prikaz stanja nodova

9.2 Kubernetes

9.2.1 Prvi Korak: Izgradnja Docker Image-a

Kubernetes takođe radi s kontejnerima, tako da prvo moramo napraviti *image* naših projekata i postaviti ih na Docker Hub. Pošto je ovo objašnjeno u poglavlju 8.1.1, nećemo ponavljati taj postupak.

9.2.2 Drugi korak: Kreacija yaml fajla

YAML (Yet Another Markup Language) fajl je standard za serijalizaciju podataka koji je čitljiv za ljude i često se koristi u konfiguracionim fajlovima i za razmenu podataka između jezika sa različitim strukturama podataka. U kontekstu Kubernetes-a, *YAML* fajlovi se koriste za definisanje željenog stanja različitih Kubernetes resursa, kao što su ***Deployments***, ***Services*** i ***Pods***. Ovi konfiguracioni fajlovi omogućavaju korisnicima da specificiraju sve detalje potrebne za Kubernetes da kreira i upravlja ovim resursima.

Kubernetes Servisi su ključna komponenta Kubernetes mrežnog modela, pružajući stabilnu tačku pristupa za set Pod-ova. Servisi apstrahuju osnovne Pod-ove i nude funkcije kao što su balansiranje opterećenja, otkrivanje servisa i mrežna izolacija. Svaki servis u Kubernetes-u je definisan u *YAML* fajlu, gde korisnik specificira kako servis treba da usmerava saobraćaj ka odgovarajućim Pod-ovima, omogućavajući neometanu i pouzdanu komunikaciju unutar klastera. Za projekte sa posebnim aplikacijama ili komponentama, definisanje odvojenih servisa osigurava bolju izolaciju, skalabilnost i upravljanje.

Objasnenje strukture narednih yaml fajlova

spec: Ovo polje definiše željeno stanje *deployment-a*. Sadrži nekoliko podpolja koja određuju kako *Pod*-ovi treba da budu konfigurisani i upravljani.

- **replicas: 3:** Specificuje broj replika *pod*-ova koje treba održavati *deployment*. U ovom slučaju, Kubernetes će osigurati da tri instance *pod*-a uvek budu pokrenute.
- **selector:** Koristi se za identifikaciju skupa *pod*-ova na koje se *deployment* odnosi. Usklađuje *Pod*-ove na osnovu labela.
 - **matchLabels:** Lista ključ-vrednost parova. *deployment* kontroler koristi ovo za selekciju *pod*-ova koje upravlja. Ovde, usklađuje *pod*-ove sa labelom *app: client-react-nginx*.
- **template:** Opisuje *Pod*-ove koje će *Deployment* kreirati.
 - **metadata:** Sadrži metapodatke za *pod*-ove, kao što su labela.
 - **labels:** Ključ-vrednost parovi koji se koriste za identifikaciju i organizaciju *pod*-ova. U ovom primeru, *pod*-ovi su označeni sa *app: client-react-nginx*.
 - **spec:** Specificuje konfiguraciju za *pod*-ove.
 - **containers:** Lista kontejnera koji će se pokrenuti unutar *pod*-ova.
 - **name: client-react-nginx:** Ime kontejnera.
 - **image: sidpalas/devops-directive-docker-course-client-react-nginx:5:** Slika kontejnera koja se koristi. Ova slika sadrži aplikacioni kod i radno okruženje.
 - **ports:** Definiše mrežne portove koje kontejner izlaže.
 - **containerPort: 8080:** Port na kontejneru koji je izložen.

Deployment yaml file: api

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fast-api
  labels:
    app: fast-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: fast-api
  template:
    metadata:
      labels:
        app: fast-api
    spec:
      containers:
      - name: fast-api
        image: sidpalas/k8s-getting-started:0.0.1
        ports:
        - containerPort: 80
      resources:
        requests:
          cpu: 200m
          memory: 300Mi
        limits:
          memory: 400Mi
```

Service yaml file 1: service

```
apiVersion: v1
kind: Service
metadata:
  name: fast-api
spec:
  selector:
    app: fast-api
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

9.2.3 Treći korak: podešavanje Kubernetes klastera

U ovoj demonstraciji, Kubernetes klaster će biti postavljen na *Civo* platformi.

Cluster Information

Name:	k8s-getting-started	Node total:	3 nodes
K3s Version:	v1.22.11+k3s1	External IP:	https://212.2.240.69:6443
Network:	default	API Endpoint:	https://212.2.240.69:6443
Kubeconfig:	Click to download	DNS Name:	cabaf957-af67-4542-b903-2c4i
		CNI:	Flannel

Node pools [Create new pool](#)

Pool: **b9e13b** (3 x Small - Standard 1 CPU 2 GB 40 GB) [Scale Up/Down](#) [Delete](#)

k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-h7uwb	k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-ue4tn	k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-wrcs9 IP: 212.2.240.69
--	--	--

Slika 35 - Demonstracioni kubernetes klaster

Prvo ćemo preuzeti *Kubeconfig* našeg čvora koji nam služi za autorizaciju korisnika i pristup klasteru, a zatim ćemo ga podesiti pomoću komande

- `export KUBECONFIG=[path to config location]`

Nakon toga, pokretanjem naredbe `kubectl get nodes` možemo videti sve aktivne čvorove.

```
> export KUBECONFIG=/Users/palas/Desktop/k8s-getting-started/civo-kubeconfig
> kubectl get nodes
NAME                                     STATUS    ROLES    AGE    VERSION
k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-ue4tn Ready     <none>   11m    v1.22.11+k3s1
k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-h7uwb Ready     <none>   11m    v1.22.11+k3s1
k3s-k8s-getting-started-fc76-a88f9c-node-pool-3ab2-wrcs9 Ready     <none>   11m    v1.22.11+k3s1
```

Slika 36 - Podesavanje kubeconfig fajla

Sledeći korak je aplikacija naših *yaml* fajlova, sto možemo uraditi pomoću komande

- `kubectl apply -f [yaml file path]`

```
> kubectl apply -f .
deployment.apps/fast-api created
service/fast-api created
```

Slika 37 - Aplikacija yaml fajlova na klaster

```
> kubectl get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
fast-api-6b55fb995f-tm8dt	1/1	Running	0	27s
fast-api-6b55fb995f-db28l	1/1	Running	0	27s
fast-api-6b55fb995f-qhlvz	1/1	Running	0	29s

Slika 38 - Status pod-ova sa pokrenutim yaml fajlovima

Sada, kako bismo proverili da li se aplikacija pravilno podigla u našim podovima, možemo direktno pristupiti sa naše lokalne mašine pomoću *port-forwardovanja*. *Port-forwardovanje* do jednog od podova u Kubernetes klasteru omogućava vam direktan pristup specifičnom podu sa vaše lokalne mašine. Ovo može biti korisno za *debugovanje*, testiranje ili pristupanje servisima koji se izvršavaju unutar vašeg klastera, ali nisu spolja dostupni.

Posto je u ovom primeru aplikacija *web api* u *browser-u* mozemo ukucati *kubectl:[local port]* kako bi smo pristupili podu direktno

- *kubectl port-forward [pode name] [port from: port to]*
- *kubectl port-forward fast-api-6b55fb995f-tm8dt 8080:80*

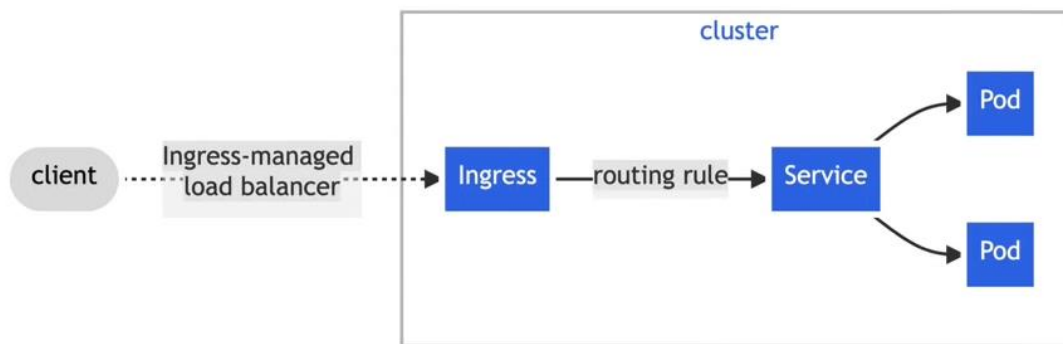
```
> kubectl port-forward fast-api-6b55fb995f-tm8dt 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Slika 39 - Port-forward-ovanje do zeljenog pod-a

9.2.4 Četvrti korak: otvaranje podova javnosti (Ingress)

Trenutno je naš kod podignut u klaster i uspešno pokrenut, ali nije dostupan nikome osim nama. Ovo ćemo popraviti pomoću *Ingress-a*, koji će omogućiti pristup javnosti.

Ingress je *API* objekt u Kubernetes-u koji upravlja eksternim pristupom servisima u klasteru, obično preko HTTP i HTTPS protokola. On omogućava definisanje pravila za *rutiranje* spoljnog saobraćaja ka unutrašnjim servisima, pružajući fleksibilnost u načinu na koji su servisi dostupni spoljnim korisnicima. Kroz *Ingress* možemo specificirati URL putanje, hostove, *SSL/TLS* sertifikate i druge postavke kako bismo kontrolisali pristup aplikacijama unutar Kubernetes klastera.



Slika 40 - Način rada ingress-a

Napraviti ćemo Ingress YAML fajl koji će preusmeravati komunikaciju sa našeg javnog *endpoint-a* za klaster na naš DNS, koji je u ovom slučaju podešen na *Cloudflare* sajtu.

- `kubectl apply -f ingress.yaml`

Tako da sada, kada komunikacija pristupi javnom *endpoint-u* klastera, *Ingress* je prebacuje na naš servis, koji dalje prosleđuje ka našem podu gde radi naša aplikacija.

Ingress yaml file

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: fast-api
spec:
  rules:
  - host: k8s.example.com
    http:
      paths:
      - path: /
        pathType: Exact
      backend:
        service:
          name: fast-api
          port:
            number: 80
```

DNS


A few more steps are required to complete your setup. [Hide](#)

- ✓ [Use wizard to add an SPF record](#) and define what mail servers are allowed to send mail for your domain. [New Alert](#)
- ✓ [Use wizard to add a DMARC policy](#) and choose what happens to outgoing mail that fails authentication. [New Alert](#)

DNS management for **example.com**

Search DNS Records

[Search](#) [Advanced](#) [+ Add record](#)

Type	Name	Content	Proxy status	TTL	Actions
A	k8s	212.2.240.69	 Proxied	Auto	Edit ▶

Slika 41 - DNS koji rutuje komunikaciju do pod-a preko ingressa

10.0 Zaključak

Ovaj rad je pokazao da Kubernetes i Docker Swarm, iako pružaju slične osnovne funkcionalnosti za orkestraciju kontejnera, značajno se razlikuju u pogledu složenosti, fleksibilnosti i funkcionalnosti. Kubernetes je idealan za velike organizacije sa složenim aplikacijama koje zahtevaju visoku dostupnost, napredne mrežne opcije i skalabilnost. Kubernetes-ova fleksibilnost i širok spektar integracija sa eksternim alatima čine ga moćnim alatom za upravljanje *kontejnerizovanim* aplikacijama u produkcionim okruženjima sa visokim tehničkim zahtevima.

S druge strane, Docker Swarm je pogodniji za manje timove i jednostavnije aplikacije zbog svoje jednostavnosti i brzog postavljanja. Swarm je popularan među programerima koji preferiraju brzo startovanje aplikacija na klasteru, bez mnogo konfiguracije, što ga čini idealnim za manje kompleksne projekte.

Izbor između Kubernetes-a i Docker Swarm-a zavisiće od specifičnih potreba i nivoa tehničkog znanja tima. Organizacije treba da uzmu u obzir složenost aplikacija, zahteve za skalabilnost, dostupnost resursa i tehničku ekspertizu pre nego što donesu odluku koji alat će koristiti za orkestraciju kontejnera. U svakom slučaju, oba alata predstavljaju moćna rešenja za modernu orkestraciju kontejnera i pružaju solidne temelje za upravljanje *kontejnerizovanim* aplikacijama.

11.0 Literatur

- 1) Huda, A. N., & Kusumawardani, S. S. (2022). "Kubernetes Cluster Management for Cloud Computing Platform: A Systematic Literature Review," JUTI: Jurnal Ilmiah Teknologi Informasi, 20(2), pp. 75–83.
- 2) Kratzke, N., & Quint, P.-C. (2017). "Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study," The Journal of Systems and Software, 126, pp. 1-16. doi:10.1016/j.jss.2017.01.001
- 3) Vasireddy, I., Ramya, G., & Kandi, P. (2023). "Kubernetes and Docker Load Balancing: State-of-the-Art Techniques and Challenges," International Journal of Innovative Research in Engineering and Management, 10(6), pp. 49-54.
- 4) Nesic, A., "Primer razvoja softvera putem DevOps principa,".
- 5) Deket, M., "Comparison of Docker Swarm and Kubernetes,".
- 6) Docker, "Docker Documentation,".
- 7) Kubernetes, "What is Kubernetes?,".
- 8) Docker, "What is Docker?,".
- 9) Google Cloud, "Google Kubernetes Engine Documentation,".
- 10) Amazon Web Services, "Amazon EKS Documentation,".
- 11) Amazon Web Services, "Amazon EC2 Documentation,".
- 12) Microsoft Azure, "Azure Kubernetes Service (AKS),".
- 13) Microsoft Azure, "Azure Virtual Machines,".
- 14) GeeksforGeeks, "Introduction to Docker Swarm Mode,".
- 15) IBM, "Docker Swarm vs Kubernetes,".
- 16) Better Stack, "Docker Swarm vs Kubernetes,".
- 17) phoenixNAP, "Docker Swarm vs Kubernetes,".
- 18) S. K. Sharma, M. K. Soni, "Performance Comparison Between Virtual Machines and Docker Containers," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 100-113, 2020.
- 19) R. Morabito, "Power and Performance Benchmarks for OpenStack, Docker and Linux Container Virtualization," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 163-176, 2018.
- 20) B. Burns, B. Grant, D. Oppenheimer, E. Brewer and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, pp. 70-93, 2016.
- 21) Amirullah, R. M. Ijtihadie and H. Studiawan, "Optimasi Daya Data Center Cloud Computing Pada Workload High Performance Computing (HPC) Dengan Scheduling Prediktif Secara Realtime," JUTI: Jurnal Ilmiah Teknologi Informasi, vol. 15, no. 1, pp. 1-10, 2017.
- 22) IBM Cloud Education, "Containerization".