Nadzor rada mikroservisa i osiguravanje visoke dostupnosti

Abramović, Maja

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:578273

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-07-17**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

Sveučilište u Rijeci – Fakultet informatike i digitalnih tehnologija

Diplomski studij informatike, smjer informacijski i komunikacijski sustavi

Maja Abramović

Nadzor rada mikroservisa i osiguravanje visoke dostupnosti

Diplomski rad

Mentor: doc. dr. sc. Vedran Miletić

Rijeka, 25. prosinca 2022.

Sažetak

Usporedno s razvojem softvera nastali su novi pristupi arhitekture i održavanja softvera. Pristupačnost internetske veze i hardvera omogućuju krajnjem korisniku da neprekidno koriste softver, a razvojni timovi moraju omogućiti visoku dostupnost. Mikroservisna arhitektura je odgovor kako prilagoditi softverski kod radu u redundancijama, a tehnološki mikroservise podupiru kontejnerizacija i orkestracija kontejnera. Kubernetes je vodeći orkestrator kontejneriziranih okolina i omogućuje nesmetan, visoko dostupan rad mehanizmima stvaranja redundancije. Kao dodatna mjera sigurnosti, softver se može nadzirati alatima, a poželjno ga je konfigurirati tako da su promjene u trendovima očite.

Ključne riječi: mikroservisna arhitektura; kontinuirana integracija; kontinuirana isporuka; kontinuirani razvoj; kontejnerizacija; orkestracija; Kubernetes; k8s; nadzor mikroservisa; dostupnost mikroservisa; visoka dostupnost; Prometheus; Grafana;

Sadržaj

1. Uvod . . . 1

2. Pregled temeljnih pojmova područja . . . 2

2.1. Mikroservisna arhitektura . . . 2

2.2. Kontinuirana integracija i kontinuirana isporuka . . . 5

2.3. Kontejnerizacija i orkestracija . . . 6

3. Orkestrator Kubernetes . . . 10

3.1. Instalacija alata Kubernetes u grozdu s više čvorova . . . 11

3.1.1. Podešavanje na svim čvorovima . . . 13

3.1.2. Podešavanje na glavnom čvoru grozda . . . 16

3.1.3. Podešavanje na čvorovima radnicima . . . 18

3.1.4. Lokalno upravljanje grozdom . . . 19

3.2. Osnovni objekti orkestratora Kubernetes . . . 19

3.3. Kubernetes alati komandne linije . . . 23

4. Dostupnost i nadzor mikroservisa . . . 27

4.1. Dostupnost mikroservisa . . . 27

4.2. Nazor mikroservisa . . . 30

4.3. Osiguranje visoke dostupnosti Kubernetes grozda . . . 32

5. Alati Prometheus i Grafana . . . 35

5.1. Opis komponenti sustava Prometheus . . . 36

5.2. Prikupljanje, pohrana i obrada podataka . . . 37

6. Primjer nadzora mikroservisne arhitekture . . . 40

6.1. Postavljanje alata Prometheus i komponenti alatom Helm . . . 40

6.2. Postavljanje vlastitog softvera u orkestrirano okružje . . . 46

6.3. Prikazi metrika u Grafana sučelju . . . 53

7. Zaključak . . . 62

Literatura 63

Poglavlje 1

Uvod

Povijest softvera prati dugi niz godina razvoja, a tehnološki i sociološki razvoj društva je uvjetovao promjene u samom načinu primjene. Nekoć internetska veza nije bila dostupna 24 sata u danu te sedam dana u tjednu, već je pojam same veze bio vezan uz visoke troškove. Danas se internetske usluge poslužuju bez prestanka te zahtijevaju osiguranje visoke dostupnosti.
Sve važne komponente softvera su nekoć bile nastanjene na jednom čvoru te su usluge bile tolerantne na pogreške, ali ne visoko dostupne. Ovakav pristup stvarao je potencijalne probleme jer što je niže u procesu nastala pogreška, to je veći utjecaj mogla imati na gubitak podataka. Redundancije usluga su bile teške za postići, a nepouzdane zbog potencijalnog raspršivanja pogreške.
Internetska veza postala je opće dostupna usluga, visoka dostupnost je postala uvjet koji treba podržati hardverski i softverski. Visoku dostupnost općenito omogućuje mikroservisni pristup organizacije razvoja softvera ili mogućnost kontejnerizacije, korištenje orkestratora koji omogućuju replikacije i druge mehanizme za održavanje dostupnih sustava. Prelazak softvera s klasične monolitne arhitekture na mikroservisnu pokazao se kao korak ka sigurnijem održavanju visoke dostupnosti.
Međutim, nije bilo dovoljno prilagoditi samo softver visokoj dostupnosti, već i temeljne navike razvojnih timova. Da bi se smanjile šanse pogrešaka, uvele su se kontinuirane aktivnosti testiranja softvera i automatizacije procesa dostave koda u orkestrirane okoline. Da bi se na pogreške moglo reagirati u pravo vrijeme, uvedene su nove uloge u razvojne timove poput DevOps inženjera, inženjera sigurnosti i inženjera za osiguranje pouzdanosti (engl. *site reliability engineer*, skraćeno SRE).
Diplomski rad je organiziran kako slijedi: u drugom poglavlju se daje pregled temeljnih pojmova područja orkestracije mikroservisa, a u trećem poglavlju se nastavlja s detaljnim iznošenjem informacija o za to korištenim alatima. u sedmom poglavlju iznosi se zaključak rada.

Poglavlje 2

Pregled temeljnih pojmova područja

2.1 Mikroservisna arhitektura

Mikroservisne aplikacije su servisi modelirani oko poslovne domene tako da ih je moguće samostalno isporučiti. Pojedini servis enkapsulira softversku funkcionalnost te ju poslužuje drugim servisima putem računalnih mreža. Povezujući više servisa gradi se kompleksna mreža, a tada se prema vanjskom korisniku servisi predstavljaju kao cjelina. Zbog toga što su servisi prividna cjelina, iz vanjskih sustava se pojedini servis tretira kao crna kutija [1].
Mikroservisi se grade oko temeljnog koncepta skrivanja informacija [2]. Koncept objašnjava da se treba težiti ka skrivanju što više informacija unutar pojedine komponente, a manje dijelove informacija se može otkrivati na izloženim mrežnim punktovima. Ovaj pristup jasnije pojašnjava koliko su promjene riskantne. Općenito, manje su riskantne one promjene koje ne dolaze u dodir s vanjskim strankama, ne dolaze u dodir s mrežnim sučeljem te su unazad kompatibilne. Promjene unutar jednog servisa ne bi trebale uzrokovati promjene u drugima, pojedini servis treba biti izoliran i neovisno moguće isporučiti.
Pojava mikroservisne arhitekture nije učinila monolitnu i servisno orijentiranu arhitekturu manje validnim opcijama. Vrijedi naglasiti da pojmovi monolita i naslijeđenog koda (engl. *legacy code*) nisu sinonimi te da se takav kod još uvijek razvija. Monolit je zadana arhitektura dok se analizom ne ustanovi da bi gradnja takvog softvera bila previše kompleksna. Arhitektura samog monolita ima razne opcije izgradnje te je često dovoljna za implementaciju poslovnih pravila. Neke od ključnih prednosti su brža prilagodba softverskih inženjera na projekt, jednostavnije rješavanje problema te nadgledanje aplikacijskih dnevnika. Kod monolita je česta pojava kod koji se ponovno koristi tako da je moguće skratiti vrijeme potrebno da se značajka razvije.
Arhitektura koja je preteča mikroservisnoj arhitekturi je servisno orijentirana arhitektura. Ova arhitektura je nastala kao odgovor na teško skaliranje velikih monolitnih softvera u nadi da će se što više softvera moći ponovno koristiti te zamijeniti bez da korisnik uopće primijeti. U korijenu, ideja servisno orijentiranih aplikacija zvuči primamljivo, međutim pokazala se pro-

blematična u temeljnim konceptima koji su riješeni mikroservisnom arhitekturom. Kada bi se monolit počeo odvajati, servisno orijentirane aplikacije su imale fundamentalne probleme poput definicije komunikacijskih protokola (recimo, SOAP), teško definiranje servisne granularnosti, problem dostupnosti middleware alata, dijeljenje baze podataka. U usporedbi, najčešći protokoli komunikacije među mikroservisima su REST, GraphQL, Remote Procedure Calls (RPC) i brokeri poruka; preporučeno je da svaki servis koristi vlastitu bazu podataka, a koristi se middleware koji je dostupan.

Jedan od ključnih argumenata zašto koristiti mikroservisnu arhitekturu je da je pojedini servis neovisno moguće isporučiti – promjene pojedinog servisa ne uzrokuju promjene niti u jednom drugom. Ova jednostavna ideja skriva kompleksniju pozadinu i probleme koje treba riješili. Što je više servisa u arhitekturi, javljaju se sve češće isporuke. Ideja neovisne isporuke sugerira mogućnost automatizacije procesa isporuke što je više potrebno nego kod monolitnih aplikacija gdje je česta navika da se softver isporučuje ručno.

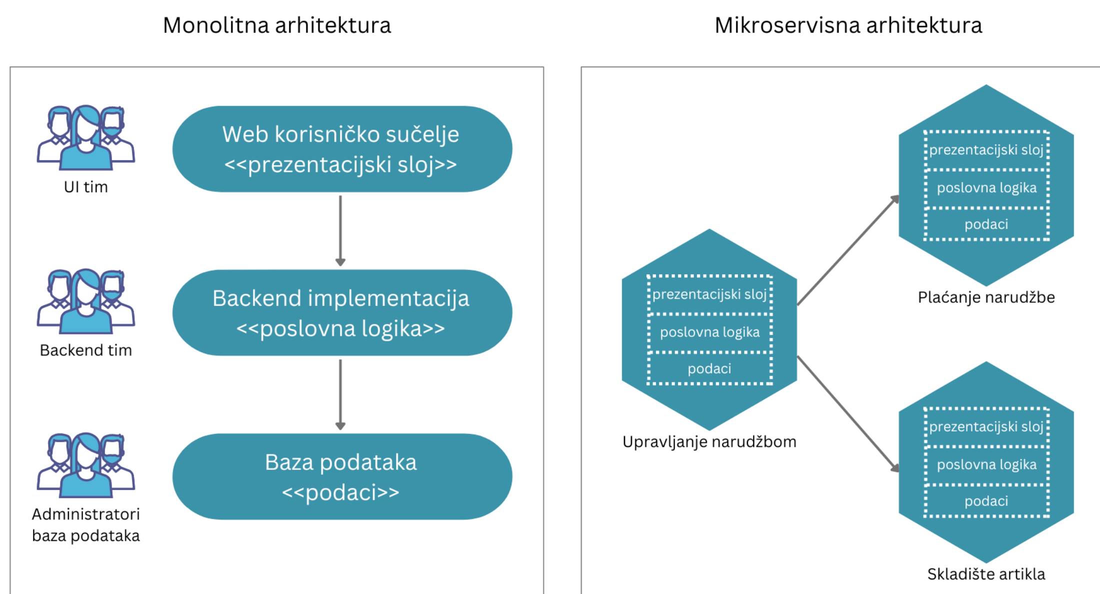
Baza podataka je također bitan aspekt neovisnosti arhitekture, često je problematična za održavanje, no ako ju koristi više servisa, raste broj operacija nad bazom te kompleksnost arhitekture. Iz tog razloga je preporuka da svaki servis koristi vlastitu bazu. Za razumijevanje uvjeta da svaki mikroservis ima vlastitu bazu podataka se može promatrati argument da ukoliko drugi servis treba podatke koje na neki način generira i pohranjuje drugi servis, do tih podataka treba doći komunikacijom sa servisom, a ne tako da vrši operacije nad istom bazom. Na taj način se održava koncept skrivanja informacija te smanjuje šansu da će se podaci koristiti maliciozno.

Svaki servis implementira unaprijed definirana poslovna pravila, a odluku kako će izgledati kompleksna mreža servisa donosi arhitekt prije nego što softverski tim počne s implementacijom. Često se koristi tehnika domenski vođenog dizajna tako da se izvorni kod strukturira na način da reprezentira poslovnu domenu iz stvarnog poslovnog slučaja.

Primjer domenski vođenog dizajna moguće je objasniti na klasičnom primjeru koji se koristi za MVC metodologiju – primjer online kupovine. Klasični monolitni softver razdvaja tri sloja: prezentacijski sloj, implementiranu softversku logiku i podatke u bazi. Za razliku od monolita, koristeći mikroservisnu arhitekturu možemo značajke podijeliti na tri servisa, a sva tri enkapsuliraju tri osnovna sloja kod monolita. Primjerice, problem možemo podijeliti na tri servisa: upravljanje narudžbama, plaćanje narudžbe i stanje skladišta. Unutar svakog servisa nalazi se poslovna implementacija unutar koda, web sučelje koje će biti dostupno korisniku te logika komunikacije servisa s bazom podataka.

Naziv arhitekture implicira da servisi moraju biti maleni, ali teško je postaviti granice veličine softvera. Postoje metrike poput broja linija koda, međutim ne moraju biti vjerna reprezentacija obujma softvera. Mikroservisni pristup teži ka tome da se odrede minimalne funkcionalne cjeline, a cjeline koje bi se ponavljale treba pretvoriti u zasebne servise sučelja.

Pojavom mikroservisa struka dolazi u doticaj s mnogo nove tehnologije koju je potrebno uklju-



Slika 2.1: Razlika između monolitnog i mikroservisnog pristupa

čiti. Korištenu tehnologiju uglavnom diktiraju arhitekture softvera koje su potrebne za implementaciju poslovnih pravila, međutim u odluci aktivno sudjeluju organizacije s obzirom na obrazovanje svojeg kadra i postojećih licenci koje imaju. Rast broja servisa koje treba pratiti je zahtijevao nove tehnologije distribuiranog praćenja dnevnika, softver se počeo koristiti u kontejneriziranim okružjima pa potrebno odabrati orkestracijsko okruženje. Orkestracijsko okruženje je važna komponenta odluke hoće li infrastruktura živjeti u vlastitom data centru organizacije ili u oblaku. S obzirom da servisi trebaju druge usluge poput distribuiranih baza, raznih skladišta podataka i brokera, organizacije moraju razmišljati o latenciji. Rješenja u oblaku nude već gotova rješenja te su često najisplativija.
Zbog izolacije svakog pojedinog servisa je moguće tehnološki stog učiniti heterogenim. Ako su funkcionalnosti važne vanjskom sustavu mrežno izložene, nije važno koja tehnologija se koristi u pozadini servisa te je moguće u jednoj mreži servisa koristiti veći broj programskih jezika, različite vrste spremanja podataka, a time je moguća lakša adaptacija na moderne tehnologije. Postoji moguća zamka kod ovoga pristupa, a to je predstavljanje previše različitih tehnologija. Tada se može dogoditi da tim odgovoran za održavanje servisa nema kapacitet za količinu znanja koju bi trebali imati.
Još jedna prednost mikroservisa je mogućnost skaliranja. U monolitu nastaje problem jer se sve funkcionalnosti moraju skalirati zajedno. Za svaki servis se može očekivati neka vrsta opterećenja poput mrežne, memorijske ili hardverske. Mikroservise se može lakše instancirati i balansirati promet između njih što je vrlo važna funkcionalnost kada različiti servisi imaju različita opterećenja.

Razvijanje mikroservisa znači da se za pojedini servis radi manja količina koda i funkcionalnosti nego za cijeli monolit, a ta činjenica potiče organizacije na drugačiju organizaciju timova. Mikroservise obično razvijaju manji timovi, a u manjim grupama je komunikacija manje zahtjevna te je lakše pratiti tko je autor koda. Ukoliko je timska komunikacija ispravna, lakšim dokazivanjem vlasništva koda će se greške ranije otkloniti. Aspekt iskustva razvojnog tima može postati problem kod razvijanja mikroservisa. Osim što je potrebno da razvojni tim ima veliku količinu znanja, ovisno o tome koliko se servisa razvija, hardver koji koristi pojedini član tima može postati nedovoljan čak i s lakšim (engl. *lightweight*) alatima u pozadini. Postoji mogućnost razvoja u oblaku, međutim taj pristup može biti skup, a neefikasan. U praksi, kada su servisi postavljeni u razvojno okruženje u oblaku, postavljanje se obavlja planirano i najavljeno jer kada bi razvojni tim stalno radio promjene na oblaku, mogle bi se međusobno pregaziti. Ovakav pristup zahtjeva također može oštetiti postojeći koncept kontinuirane integracije. Ideja je da takva okruženja budu stabilna, stoga bi pojedinac trebao koristiti lokalne resurse.

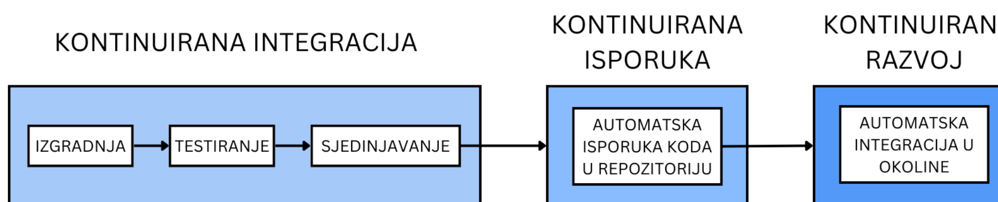
2.2 Kontinuirana integracija i kontinuirana isporuka

Kontinuirana integracija i kontinuirana isporuka (engl. *continuous integration/continuous delivery*), u nastavku CI/CD, je metoda redovite isporuke softvera korisnicima korištenjem automatizacije za razne faze razvijanja softvera. Ovaj koncept pokazuje se osjetno zahtjevan na početku razvoja novog softvera, ali kada ga se s vremenom prilagodi, postane jedan od najvažnijih dijelova aplikacijskog ciklusa. Iako je često korištena poštapalica (engl. *buzzword*) upravo akronim CI/CD, potrebno je razumjeti razliku između kontinuirane integracije i kontinuirane isporuke.

Kontinuirana integracija podrazumijeva automatizaciju osnovnih operacija nad softverskim kodom poput kompajliranja, građenja, pokretanje testnih skripti, skladištenja aplikacijskih artefakta te sjedinjavanje (engl. *merge*) novog koda s ostatkom baze koda. Ovaj korak do konačne isporuke koda se pokazao važan primarno jer svaki član razvojnog tima najčešće ima slobodu kako oblikovati vlastito testno okruženje. S obzirom da svaki član tima može imati različitu verziju programskih jezika i popratnih alata, kontinuirana integracija se najčešće obavlja nad jednom, strogo definiranom verzijom programskog jezika i popratnih alata, a ta verzija je obično prilagođena onoj koja će se koristiti u testnoj i produkcijskoj okolini.

Kontinuirana isporuka podrazumijeva da se ispravan, već ispitan kod pohranjuje u granama (engl. *branch*) namijenjenima okolinama, najčešće produkcijskoj okolini. Ovaj pristup omogućuje jasan odabir produkcijskog koda onim članovima tima koji se bave operativnim zadacima (primjerice, DevOps inženjeri ili inženjerima za pouzdanost). Kontinuirana isporuka se često griješi za kontinuirano razvijanje (engl. *continuous deployment*), a ono podrazumijeva automatizaciju integracije novog servisa na radne okoline. Korištenje kontinuiranog razvijanja znači

da razvojni tim može u pitanju minuta integrirati kod koji su razvijali te se proces integracije koda osigurava s dvije prethodne kontinuirane faze.



Slika 2.2: Kontinuirana integracija, kontinuirana isporuka, kontinuirani razvoj

Automatizirane operacije kontinuirane integracije i kontinuirane isporuke obavljaju se pomoću skripti koje nazivamo cjevovodi (engl. *pipelines*). Cjevovodi se pokreću alatima koje klijenti i razvojni tim definiraju na početku razvoja softvera. Alati koji se koriste za upravljanje cjevovodima često imaju neki dio akronima u svojem imenu, ali to ne znači nužno da ne mogu obaviti i drugi dio. Primjer CI/CD alata su Jenkins, Tekton, Spinnaker, GoCD, Concourse, Screwdriver, TravisCI, CircleCI, ArgoCD. Moderna rješenja nastoje smanjiti količinu alata koji se koriste te je nastao koncept GitOps – *framework* koji potiče korištenje CI/CD alata koji su već integrirani unutar softvera za upravljanje izvornim kodom poput Gitlaba, Githuba i BitBucketa.
Kontinuirana integracija pokazala se kao ključni koncept za brzu i ispravnu isporuku koda, ali uz automatizaciju je potrebno postaviti prioritete operacija. Koraci kontinuirane integracije trebali bi se obavljati svakodnevno te se novi kod ne bi smio pridruživati glavnim granama ukoliko ga nije moguće izgraditi te nema testove koji mogu ispitati kvalitetu. Kod projekata koji implementiraju mikroservisnu arhitekturu je iznimno važno osposobiti testove koji mogu validirati kod, a to je česta praksa koju treba implementirati svaki softverski inženjer. Ukoliko se u svakodnevnom korištenju kontinuirane integracije dogodi da gradnja ili testovi ne uspijevaju, sanacija problema bi trebala biti glavni prioritet. Koncept automatizirane integracije se mora obavljati nad granama repozitorija koje moraju ući u deblo. Voditelji tima se moraju pobrinuti da razvojni tim prati dogovorenu normu otvaranja grana, a to je da se grane koriste pametno – stvaraju se iz debla, koriste se što kraće, a onda se pridružuju deblu. Smatra se lošom praksom dodavati kod direktno u glavne grane ili raditi nove grane iz već otvorenih grana te da bi se to reguliralo, moraju postojati definirane norme.

2.3 Kontejnerizacija i orkestracija

Kontejner je način na koji je moguće ujediniti softverski kod i sve njegove ovisnosti tako da je moguće pokretati aplikaciju u izoliranom okolišu koji se na isti način pokreće na svakoj

platformi. Najčešći slučajevi u kojima koristimo kontejnere su [3]:

- prenosivost softvera neovisno o platformi iza kontejnera,
- brzina kojom kontejner postaje dostupan,
- dostupnost kontejnerskih slika iz javnih ili privatnih registratora,
- osim ako korisnik ne kontejnerizira vlastitu aplikaciju, ne mora brinuti o instalaciji;
- kontejneri čine dobar način kako učiti nove tehnologije u izoliranom okolišu.

Prema principu rada, kontejner najviše podsjeća na rad virtualnih strojeva, ali važno je naglasiti da se bitno razlikuju u arhitekturi. Kontejner ima cilj virtualizirati operacijski sustav na kojem se nalazi, dok hipervizor virtualnim strojevima dopušta virtualizaciju na hardverskoj razini. Različiti kontejneri dijele isto jezgra operacijskog sustava (engl. *kernel*), ali izolirano rade u korisničkom prostoru. Kontejner može imati dva stanja:

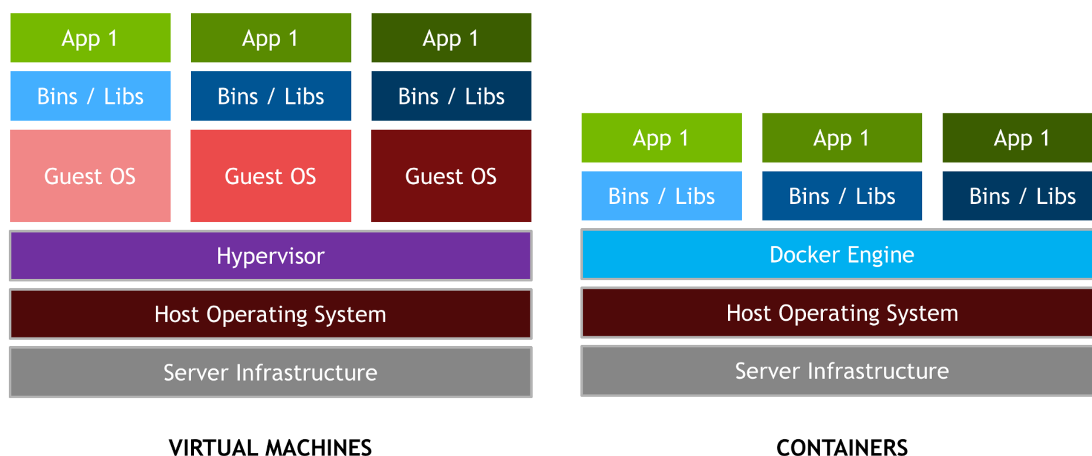
- Neaktivno stanje: Kontejnerom smatramo kontejnersku datoteku ili skup takvih datoteka koji je spremljen na disku. Jednu takvu datoteku nazivamo kontejnerskom slikom, a više slika koje zajedno čine cjelinu kontejnerskim repozitorijem.
- Aktivno stanje: Kontejner je pokrenut i operacijski sustav ga tretira poput ostalih procesa. Proces roditelj kontejnerskom procesu su alati za kontejnerizaciju kao što su Docker i Podman.

Kontejnerske slike često preuzimamo s registarskih servera (engl. *registry server*), a najčešće korišteni javni registarski server je Docker Hub. Moguće je održavati vlastiti registarski server ili barem promijeniti zadani na neki drugi javno dostupni.

Da bi približili zašto je kontejner poseban koncept, potrebno je razmišljati u kontekstu njegovog načina rada. Za razliku od ostalih procesa s kojima upravlja operacijski sustav, kada započinje rad kontejnera šalje drugačije vrste signala nego u slučaju procesa. Alat za kontejnerizaciju iza svakog kontejnera na taj način omogućava da kontejner ima vlastiti imenski prostor jezgre (engl. *kernel namespace*), odnosno mogućnost da individualno mogu definirati mrežne postavke, memorijske točke, korisnike, procese te njihove identifikatore i slične koncepte kojima se koriste operacijski sustavi.

Tradicionalna virtualizacijska rješenja obično rade na principu kreiranja apstraktnog sloja između fizičkog hardvera i softvera koji će biti pokrenut. U tom slučaju hipervizori svakom virtualnom stroju daju postotak hardverskih resursa koje mogu koristiti, za razliku od kontejnera koji je još jedan proces jezgre domaćina. Za kontejner se na taj način može dinamički rezervirati resurs te ga po potrebi ograničiti.

Kontejnerizacija je bazirana na filozofiji kreiranja atomičkih instanci softvera kojima ne predstavlja problem kratka i brza upotreba koju se potencijalno može odbaciti. Takav pristup omogućuje robusno i portabilno rješenje problema. U prošlosti su se nerijetko na projekte dodavale pozicije koje se bavile kreiranjem prirodnog okruženja za softver u razvoju, dok kontejnerizacijski alati potpuno okruženje mogu stvoriti kroz svega nekoliko definirajućih datoteka, poput



Slika 2.3: Razlika rada virtualnog stroja i kontejnera. Izvor: [4]

Dockerfile datoteke i/ili *docker-compose* datoteke.

Alatima za kompozicije kontejnera (primjerice, *Docker Compose* i *Podman Compose*) moguće je grupirati više različitih aplikacija koje se koriste u grupi mikroservisa. Definirajući svaku pojedinu vlastitim definicijskim datotekama se lokalno može imitirati produkcijska okolina te ukoliko okoline koriste iste kontejnerske slike, pogreške i opasna ponašanja se mogu primijetiti u ranijim fazama razvoja.

Ideja orkestracije je posluživati mikroservise dostupne mrežno koristeći API pouzdano i ispravno skalirano. S obzirom da su orkestracijski alati inače posluženi u distribuiranim okruženjima, nastaje potreba za koordinacijom i upravljanjem mrežne komunikacije. Svakodnevno se koristi sve više API komunikacije koja ovisi o visokoj dostupnosti i pouzdanosti. Da bi ovakve usluge bile dostupne i pouzdane, orkestratori moraju brinuti da su servisi visoko dostupni i da imaju mehanizme oporavka u slučaju kritičnih pogrešaka. Kada raste ili pada potreba za mrežno dostupnim servisima, potrebno je skalirati servise tako da se podiže ili smanji broj instanci, a taj posao također obavlja orkestrator. Unatoč brojnim značajkama koje orkestracijski alati nude instalacijom osnovnih paketa, u razvoju je nastala potreba za više kontrole nad aplikacijama i resursima koji su im potrebi. U tu svrhu je osmišljen koncept orkestracije, odnosno procesa isporuke kontejneriziranih aplikacija s kontroliranim životnim ciklusom, mogućnosti skaliranja te kontroliranja hardverskih i mrežnih metrika. Potreba za orkestratorima nije sasvim očita iz konteksta lokalnog razvijanja aplikacije, ali s porastom kompleksnosti na drugim okolinama, kontrola kontejnera postaje mukotrpan posao.

Orkestratori najčešće rade distribuirano kroz više čvorova, najčešće postoji jedan kontrolni čvor te jedan ili više čvorova radnika. Kontrolni čvor se bavi primanjem zahtjeva te organizira ostale čvorove dok radnici obavljaju stvarnu orkestraciju i operacije s kontejnerima. U kontekstu orkestratora se najčešće priča o grupi čvorova (engl. *cluster*), ali orkestracija se može obavljati i na razini jednog čvora. Rješenje na jednom čvoru može biti Docker Compose ili druga rješe-

nja temeljena na Kubernetesu poput alata Minikube, Portainer, Minishift, kind (Kubernetes in Docker) i slično.

Čvorovi orkestracijskog alata pokreću kontejner prema parametrima zadanim u deskriptivnoj datoteci YAML ili JSON formata. Da bi se kontejner pokrenuo, orkestrator mora obaviti nekoliko inicijalnih koraka:

1. Zakazivač (engl. *scheduler*) određuje koji čvor će obavljati operacije s kontejnerom. Odluka se donosi u odnosu na definirane parametre, a najčešće se radi o parametrima vezanim uz potrebne hardverske resurse.
2. Kada je čvor odabran, zakazivač zahtjeva pokretanje kontejnera. Ova operacija se odnosi na korake poput preuzimanja kontejnerske slike iz registratora, podešavanje mreže kontejnera, pridruživanja memorijskog prostora.
3. Nakon rezervacije osnovnih hardverskih i mrežnih resursa, kontejner se pokreće.
4. Ukoliko su podešene provjere zdravlja kontejnera, one se obavljaju. Ukoliko servis nije dostupan, ovisno o postavkama se kontejner može ugasiti ili ponovno pokrenuti.
5. Nakon što su inicijalne provjere zdravlja uspješne, kontejner je aktivan te orkestrator kontinuirano provjerava zdravlje te promatra kontejner.

Neka od najčešćih orkestracijskih rješenja su sljedeća:

- Docker Swarm,
- HashiCorp Nomad,
- Google Kubernetes Engine i Google Cloud Run,
- Red Hat Openshift,
- Amazon EKS, Amazon ECS, Amazon Fargate;
- Azure Kubernetes Service i Azure Container Instances.

Rad s kontejnerima i orkestratorima se opisuje kao nepromjenjiv za razliku od klasičnog posluživanja servisa. Promjene su bilo kakve nadograde na postojeći servis, operacijski sustav na kojemu servis radi ili na popratni dostupni softver. U klasičnim uvjetima, odgovorna osoba ručno obavlja nadograde te provjerava integritet, a u radu s kontejnerima se obavi novi build kontejnera i popratnih orkestracijskih objekata. Iako je moguće ručno vršiti nadogradbe u kontejneru, nije preporučljivo to raditi bez ispravnih operacija kontinuiranog razvoja i integracije. Ručne promjene se izbjegavaju jer su sklone prepisivanju, teško je pratiti vlasništvo promjena te otežavaju *rollout* i *rollback* procese. Pristup nepromjenjivog rada najbolje funkcionira uz deklarativnu vrstu konfiguracije – konfiguracija koja ne treba biti pokrenuta da bi bila razumljiva. Ovakav pristup konfiguraciji pokazao se manje sklon pogreškama te omogućuje korištenje infrastrukture poput koda (engl. *infrastructure as a code*, skraćeno IaaC). Idealan operativni pristup infrastrukturi nepromjenjivog rada je GitOps jer se tada kod strukturira prema okolinama u kojima je integriran. Tada se nadograde, deklarativni pristup i integritet konfiguracije može potvrditi prije produkcije.

Poglavlje 3

Orkestrator Kubernetes

Kubernetes, također poznat kraticom k8s, je orkestrator za isporuku kontejneriziranih aplikacija otvorenog koda [5]. Na temelju prethodnih iskustava i bolnih točki internog orkestracijskog alata Borg, Kubernetes je inicijalno osmišljen i razvijen sa strane Googleovog razvojnog tima. Nakon što je 2014. godine postao softver otvorenog koda, Kubernetes je stekao popularnost radi kvalitetno osmišljenih značajki poput automatskih *rollout* i *rollback* operacija, podrške potpunog životnog ciklusa kontejnera, podrške za horizontalno i vertikalno skaliranje, mogućnosti samostalnog zacjeljivanja te napredne kontrole pristupa s obzirom na uloge (engl. *Role-Based Access Control*). Kubernetes je danas jedan od najvećih projekata otvorenog koda te je uzvodno rješenje (engl. *upstream*) za brojna orkestracijska rješenja u oblaku, primjerice Google Kubernetes Engine, Red Hat Openshift, Amazon EKS, Azure Kubernetes Service i sličnih. Alat je službeno podržan na popularnim Linux distribucijama (CentOS, RHEL, Debian, Ubuntu, SUSE) te na Windows Server distribucijama. Alat je pogodan raditi na raznovrsnom hardveru od malenih računala poput računala Raspberry Pi sve do velikih podatkovnih centara i rješenja u oblaku.

Samostalno zacjeljivanje je značajka zbog koje Kubernetes ne prestaje pratiti status servisa čim je servis upogonjen na željeni status. Kubernetes kontinuirano poduzima mjere protiv neuspjeha servisa. Klasične integracije obično zahtijevaju ljudsku intervenciju na pogreške sustava, a Kubernetes vremenski dostavljenim probama potvrđuje zdravlje servisa te održava broj instanciranih replika točno onim kako je definirano u deklarativnim konfiguracijama. Da bi se omogućilo još bolje samostalno zacjeljivanje, razvojni tim može odlučiti koristiti kontejnerske servise koje im Kubernetes omogućuje korištenjem operatora.

Nepromjenjiv rad i deklarativna konfiguracija omogućuju da Kubernetes može automatski skalirati servise [6]. S obzirom da se ne očekuju promjene unutar kontejnera, skaliranje se može previdjeti prema mrežnom opterećenju do servisa te dostupnosti fizičkog hardvera na kojemu je integriran Kubernetes naspram hardverske potrebe servisa. Operativni tim može samostalno definirati na koji način će se servis skalirati ili dopustiti da Kubernetes automatski skalira. Ku-

bernetes usluge u oblaku čak mogu unaprijed predvidjeti cijenu budućeg rada servisa te na temelju takvog izvještaja ponuditi operacijskom timu odluku kako skalirati.
Kubernetes omogućuje brojne apstrakcije i API rješenja za održavanje mikroservisa:

- Ljuska (engl. *pod*): apstrakcija koja predstavlja grupu jednog ili više kontejnera koji dijele zajedničku mrežu, memoriju i informacije.
- Usluge (engl. *sevices*): osiguravaju balansiranje mrežnog prometa, imenovanje i otkrivanje izoliranih servisa.
- Imenski prostor (engl. *namespace*): omogućuje izolaciju servisa i kontrolu pristupa administracije.
- Ulazak (engl. *ingress*): omogućuje da se usluge više servisa povežu u jedan, vanjski dostupan API.

Kubernetes službeno nudi broj alata komandne linije za upravljanje grozdom te web sučelje koje se mora omogućiti kao zasebno proširenje. Za pristup web sučelju potrebno je definirati naprednu kontrolu pristupa s obzirom na uloge. S obzirom na specifičan slučaj korištenja Kubernetes grozda, kontrola pristupa može omogućiti pristup korisnicima definiranima unutar softvera za upravljanje identitetima ili LDAP protokola. Ukoliko se računi definiraju bez vanjskih ovisnosti, web sučelju moći će se pristupiti unosom pristupnog žetona (engl. *access token*) generiranog zajedno s upravljačkim korisničkim identitetom.

3.1 Instalacija alata Kubernetes u grozdu s više čvorova

Ova instalacija odvija se na glavnom čvoru i dva čvora radnika koja možemo podignuti ručno ili automatizirano alatima kao što je Vagrant. U ovom slučaju, virtualni strojevi bili su podignute koristeći alat Vagrant i hipervizor Virtualbox, ali po želji ih se može podizati ručno, drugim alatima za automatizaciju ili koristeći druge hipervizore. Pritom je potrebno osigurati da su strojevi međusobno dostupni. Ovakav primjer instalacije sličan je stvarnom stanju u testnim, razvojnim i produkcijskim okolinama.
Za potrebu ovog rada korištene su virtualni strojevi s instaliranim operacijskim sustavom Ubuntu 20.04. Minimalni zahtjevi za virtualne strojeve Kubernetes čvorova moguće je pronaći u službenoj dokumentaciji [7]:

- Kompatibilni Linux operacijski sustav
- 2 GB RAM memorije
- 2 procesorske jezgre
- Mrežna povezanost među strojevima
- Jedinstveno ime domaćina
- Odluka o akciji vezanoj uz swap memoriju

Na sustavu domaćinu je potrebno instalirati Virtualbox i Vagrant naredbom `sudo apt ins-`

`tall virtualbox vagrant -y` ili nekim drugim ponuđenim načinom ovisno o operacijskom sustavu domaćina. Dalje je potrebno kreirati zasebni direktorij i u njemu pokrenuti `vagrant init` koji kreira Vagrant file. Vagrant file ispuniti sljedećim sadržajem (ili ga po želji i potrebi promijeniti):

```
NUM_WORKER_NODES=2
IP_NW="10.0.0."
IP_START=10

Vagrant.configure("2") do |config|
    config.vm.box = "ubuntu/focal64"
    config.vm.box_check_update = true

    config.vm.define "master" do |master|
      master.vm.hostname = "master-node"
      master.vm.network "private_network", ip: IP_NW + "#{IP_START}"
      master.vm.provider "virtualbox" do |vb|
          vb.memory = 4048
          vb.cpus = 2
          vb.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
      end
    end

    (1..NUM_WORKER_NODES).each do |i|
      config.vm.define "node0#{i}" do |node|
        node.vm.hostname = "worker-node0#{i}"
        node.vm.network "private_network", ip: IP_NW + "#{IP_START + i}"
        node.vm.provider "virtualbox" do |vb|
            vb.memory = 2048
            vb.cpus = 1
            vb.customize ["modifyvm", :id, "--natdnshostresolver1", "on"]
        end
      end
    end
  end
```

Podizanje virtualnih strojeva započinje naredbom `vagrant up`. Vagrant skripta u ovom slučaju izvodi sljedeće:

- Skripti su zadane varijable koje određuju da će se podignuti dva virtualna stroja za čvorove radnike, statična IP adresa svakog čvora počinje s IP_NW="10.0.0." i da će prvi definirani čvor imati broj 10 u posljednjem dijelu IP adrese (u ovom slučaju glavni čvor).
- Po podizanju svakog virtualnog stroja je potrebno obaviti update instaliranih paketa i dodati statične IP adrese čvorova u datoteku domaćina.
- Svaki čvor je podignut sa zadanim osiguranim resursima u sklopu odabranog hipervizora.

Otvoriti tri (ili koliko je već čvorova definirano) prozora terminala te u svakom pokrenuti `vagrant ssh <ime jednog od tri stroja>`.

3.1.1 Podešavanje na svim čvorovima

Potrebno je u hosts datoteku dodati statične IP adrese strojeva. Na svakom čvoru dodati sve linije tako da naredbom `sudo <željeni text editor> /etc/hosts` dodamo linije:

```
10.0.0.10  master-node
10.0.0.11  worker-node01
10.0.0.12  worker-node02
```

Za provjeru jesu li čvorovi uspješno umreženi pokrenuti `ping <neki od druga dva stroja>`. Potrebno je na svim čvorovima obaviti nadogradnju paketa `sudo apt update -y` tako da osiguramo da će Kubernetes instalacija raditi s najnovijim paketima.

Kubernetes je tek nedavno uveo podršku za swap memoriju te korištenje takve vrste memorije može uzrokovati probleme ovisno o verzijama drugih alata potrebnih za rad. Za manje, osobne okoline je preporučeno onemogućiti swap memoriju [8].

```
$ sudo swapoff -a
# onemogućiti ponovno korištenje swapa pri sljedećem bootu
$ sudo sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

Potrebno je osigurati da je u jezgri uveden modul `br_netfilter` koji nam kasnije treba za ispravnu kontrolu mrežnog prometa među ljuskama (engl. *pod*). U konfiguracijske datoteke zapisujemo potrebne parametre jer ih želimo sačuvati u slučaju ponovnog podizanja sustava.

```
$ lsmod | grep br_netfilter
$ sudo modprobe br_netfilter

$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF
```

```
$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
$ sudo sysctl --system

$ cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF

$ sudo modprobe overlay
$ sudo modprobe br_netfilter

$ cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# omogućuje promjenu sistemskih parametara bez da se cijeli sustav ponovno podiže
$ sudo sysctl --system
```

Uzrečicom *Docker* koju koristimo svakodnevno podrazumijevamo tri komponente:

- dockerd (Docker Engine): sistemsku daemon, CLI i API sučelje;
- containerd: daemon odgovoran za prikupljanje kontejnerskih slika i pokretanjem istih u formi kontejnera;
- runc: odgovoran za kreiranje imenskih prostora (engl. *namespaces*) i cgroups resursnih ograničenja.

U Kubernetes okolinama je CRI (Container Runtime Interface) zamijenio Docker Engine, a još je moguće koristiti containerd. CRI je posve prilagođen Kubernetesu i zato je službeni zadani alat, ali još uvijek postoje dodaci trećeg izvora (engl. *third party*) koji omogućuju korištenje Docker Enginea. Instalacija potrebnih paketa:

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
$ sudo apt-get update -y
$ sudo apt-get install -y \
  apt-transport-https \
```

```
  ca-certificates \
  curl \
  gnupg \
  lsb-release

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg \
  --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

$ echo \
  "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee \
  /etc/apt/sources.list.d/docker.list > /dev/null

$ sudo apt-get update -y
$ sudo apt-get install -y docker-ce docker-ce-cli containerd.io
```

Dodatno je potrebno konfigurirati containerd i ponovno pokrenuti daemon:

```
$ sudo mkdir -p /etc/containerd
$ containerd config default | sudo tee /etc/containerd/config.toml
(...)
$ sudo systemctl restart containerd
```

Sada se mogu instalirati Kubernetes komponente:

```
$ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg \
  https://packages.cloud.google.com/apt/doc/apt-key.gpg
$ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] \
  https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee \
  /etc/apt/sources.list.d/kubernetes.list
(...)
$ sudo apt update -y
(...)
$ sudo apt install -y kubelet kubeadm kubectl
(...)
$ sudo apt-mark hold kubelet kubeadm kubectl
(...)
```

3.1.2 Podešavanje na glavnom čvoru grozda

Za integraciju gotovih softverskih rješenja u vlastite Kubernetes imenske prostore korisno je poslužiti se upraviteljom paketa Helm. Helm mora biti instaliran isključivo na glavnom čvoru grozda. Prvo je potrebno dodati ključeve lokalno, dodati repozitorij iz kojega je moguće preuzeti Helm paket, a potom ga instalirati alatom APT. Osim alatom za upravljanje paketima, Helm je moguće postaviti kompajliranjem izvornog koda ili pokretanjem instalacijske skripte koja je dostupna u službenom GitHub repozitoriju alata Helm.

```
$ curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee \
  /usr/share/keyrings/helm.gpg > /dev/null
$ sudo apt-get install apt-transport-https --yes
(...)
$ echo "deb [arch=$(dpkg --print-architecture) \
  signed-by=/usr/share/keyrings/helm.gpg] \
  https://baltocdn.com/helm/stable/debian/ all main" | sudo tee \
  /etc/apt/sources.list.d/helm-stable-debian.list
(...)
$ sudo apt-get update
(...)
$ sudo apt-get install helm
(...)
```

Grozd započinjemo sljedećom naredbom. Vrijedi uzeti u obzir da je IP master čvora eksplicitno naveden jer je tako definiran u Vagrant datoteci, ali može se izvesti kao varijabla. Ukoliko su strojevi bili ručno podizani, potrebno je pripaziti na sljedeći IP. U ovom slučaju, statički IP master čvora definiran je kao 10.0.0.10, a pod CIDR je 192.168.0.0/16. Također, dodana je zastavica da se ignoriraju errori vezani za swap memoriju unatoč tome da je već onemogućena. Razlog tome je pokrivanje od mogućih pogrešaka i upozorenja.

```
$ sudo kubeadm init --apiserver-advertise-address=<IP master čvora> \
  --apiserver-cert-extra-sans=<IP master čvora> \
  --pod-network-cidr=<pod CIDR> --node-name <hostname master čvora> \
  --ignore-preflight-errors Swap
(...)
```

Izlaz ove naredbe će dati naredbu s kojom ćemo spojiti worker čvorove. Novu naredbu potrebno je sačuvati za kasnije.

```
$ sudo kubeadm join 10.0.0.10:6443 --token 9w4ma4.dzs52hd7m3zm6p8d \
  --discovery-token-ca-cert-hash \
```

```
  sha256:5fe33c7f7ad410692779ba9efabf2c9f1efb3a82beebc4cf8fc2c88628962a4b
# alternativno, spremiti istu naredbu u datoteku
$ sudo kubeadm token create --print-join-command > ~/join.sh
```

Potrebno je podesiti Calico plugin koji dodijeljuje podovima IP adrese.

```
$ curl -O https://docs.projectcalico.org/manifests/calico.yaml
(...)
$ kubectl apply -f calico.yaml
(...)
```

Za upravljanje Kubernetesom koristeći korisničko sučelje potrebno je podesiti Dashboard proširenje. Potom je potrebno definirati administratorski račun kojim će biti omogućeno upravljanje. Za prijavu u korisničko sučelje potrebno je priložiti generirani žeton koji se generira u zadnjem koraku.

```
$ kubectl apply -f \
  https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommendec
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
EOF

$ cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

```
EOF


$ kubectl -n kubernetes-dashboard get secret \
  $(kubectl -n kubernetes-dashboard get sa/admin-user \
    -o jsonpath="{.secrets[0].name}") -o \
  go-template="{{.data.token | base64decode}}" >> ~/token
```

Konačno, započinjemo kubelet sljedećim naredbama:

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
$ sudo systemctl restart systemd-resolved
$ sudo systemctl daemon-reload && sudo systemctl restart kubelet
(...)
```

3.1.3 Podešavanje na čvorovima radnicima

Svaki radnički čvor potrebno je priključiti naredbom komandne linije te u nju uključiti pristupni žeton generiran na glavnom čvoru.

```
$ sudo kubeadm join 10.0.0.10:6443 --token 9w4ma4.dzs52hd7m3zm6p8d \
  --discovery-token-ca-cert-hash \
  sha256:5fe33c7f7ad410692779ba9efabf2c9f1efb3a82beebc4cf8fc2c88628962a4b
(...)
```

Čvor radnik bit će dostupan kada se na njegovom domaćinu pokrene sistemski proces daemon `kubelet`.

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
$ NODENAME=$(hostname -s)
$ kubectl label node $(hostname -s) node-role.kubernetes.io/worker=worker-new
$ sudo systemctl restart systemd-resolved
$ sudo systemctl daemon-reload && sudo systemctl restart kubelet
(...)
```

Izlaz prve naredbe predlaže da se na glavnom čvoru pokrene naredba za dohvat čvorova `kubectl get nodes` da se potvrdi ispravno povezivanje svih radnih čvorova. Ako su čvorovi ispravno priključeni, svi čvorovi bi trebali pokazivati status *Ready*.

3.1.4 Lokalno upravljanje grozdom

Kubernetes grozdom se može upravljati s bilo kojeg računala kojemu je Kubernetesov domaćin mrežno dostupan. Lokalno mora biti dostupna konfiguracijska datoteka grozda te mora biti instaliran paket za alat komandne linije kubectl. Alat se na lokalnom računalu priključuje sljedećim naredbama:

```
$ mkdir -p $HOME/.kube
# uz pretpostavku da je u trenutnom direktoriju iskopirana config datoteka
$ cp config $HOME/.kube
$ export KUBECONFIG=$(pwd)/config
```

Potrebno je potvrditi je li se lokalno računalo spojilo uspješno, a to možemo naredbom `kubectl get nodes`.

```
$ kubectl get nodes
NAME            STATUS   ROLES           AGE   VERSION
master-node     Ready    control-plane   25m   v1.25.4
worker-node01   Ready    worker          21m   v1.25.4
worker-node02   Ready    worker          17m   v1.25.4
```

U ranijim koracima bio je instaliran plugin Kubernetes Dashboard, a tom sučelju se sada može pristupiti tako da se u jednom prozoru terminala pokrene naredba `kubectl proxy`, a u web browseru pokrene poveznica slična sljedećoj:

```
http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/\
services/https:kubernetes-dashboard:/proxy/#/login
```

Na login stranici potrebno je unesti token generiran kod stvaranja administratorskog korisnika.

3.2 Osnovni objekti orkestratora Kubernetes

Imenski prostori (engl. *namespaces*) [9] su mehanizam koji omogućuje izolaciju grupe servisa na grozdu unutar kojega je moguće kontrolirati resurse. Izolacija imenskih prostora obuhvaća isključivo objekte poput usluga i ljuski, ali ne one koji se stvaraju na razini cijeloga grozda poput objekata vezanih uz pohranu i čvorova. Iako je primarna ideja imenskih prostora podijeliti servise prema imenu, korisni su u kontekstu upravljanja korisnicima jer se na toj razini mogu dodijeliti korisnička prava. Servisi unutar zajedničkog imenskog prostora se razlikuju atributom oznake (engl. *label*). U početku korištenja novog Kubernetes grozda, automatski se podese sljedeći imenski prostori:

- `default`: Imenski prostor koji korisnik može koristiti bez stvaranja vlastitog imenskog prostora.
- `kube-node-lease`: Imenski prostor unutar kojega žive objekti za zakup resursa definirani prema svakom čvoru u grozdu. Unutar ovoga grozda se provjeravaju otkucaji srca, odnosno jedan od mehanizama za provjeru zdravlja grozda.
- `kube-public`: Iako ovaj imenski prostor nije obvezan, već dodatan, koristi se za komunikaciju čvorova unutar grozda. Ovaj imenski prostor dostupan je javno zbog mogućnosti udaljenog upravljanja grozdom.
- `kube-system`: Imenski prostor rezerviran za interno upravljanje Kubernetes alatom.

Servisi se unutar Kubernetes imenskih prostora pokreću ljuskama (engl. *pods*), odnosno najmanjim izvršljivim artefaktima. Unutar ljuski je moguće imati više kontejnera koji rade zajedno tako da bi se omogućilo dijeljenje resursa i smanjenje latencije među usko spojenim (engl. *tightly coupled*) servisima. Kontejneri koji su zajedno pokrenuti u ljuski imaju sljedeće značajke [6]:

- Svaki kontejner unutar ljuske ima vlastiti *cgroup*, odnosno značajku jezgre Linux za regulaciju resursnih limita i izolacije resursa.
- Kontejneri dijele većinu Linux imenskih prostora.
- Aplikacije unutar ljuske imaju istu IP adresu, vrata (engl. *port*), ime domaćina (engl. *hostname*) te mogu komunicirati autohtonim interprocesnim komunikacijskim kanalima.
- Kontejneri iste ljuske ne moraju biti pokrenuti na istim čvorovima u Kubernetes grozdu.

Ljuske su definirane unutar manifesta, odnosno tekstualnih datoteka definiranih deklarativnim pristupom koje Kubernetes može prihvatiti i obraditi. Manifest se obično kreira korištenjem Kubernetesovog sučelja za komandnu liniju te može biti definiran kroz različite objekte, obično objekte `Deployment` i `StatefulSet`. Unutar manifesta mora biti definirano koje je željeno stanje servisa te u koliko replika bi servis trebao biti dostupan. Kada je manifest prihvaćen, sprema se u perzistentnu pohranu. Zakazivač tada obavi provjeru koji čvor ima uvjete pokrenuti ljusku te uzima u obzir broj definiranih replika jer postavljanje više jednakih replika na isti čvor koji može zakazati stvara mogućnost pogoršanja pouzdanosti sustava.

Kada je aplikacija pokrenuta u imenskom prostoru, nad kontejnerima se automatski obavljaju provjere zdravlja. Automatske provjere su osnovne provjere koje potvrđuju da je glavni proces u kontejneru pokrenut, a ukoliko nije, proces se automatski ponovno pokreće. Često nije dovoljna samo osnovna provjera jer u slučaju nemogućnosti odgovora na zahtjev, osnovna provjera vraća pozitivan rezultat. Ovom motivacijom osmišljena je provjera živosti (engl. *liveness probe*) koja provjerava stanje aplikacije prema logici koja je definirana unutar aplikacije, a to je obično specifično definirana krajnja točka aplikace (engl. *endpoint*). Dok provjera živosti provjerava mogućnost aplikacije da odgovori na zahtjev, provjera spremnosti (engl. *readiness probe*) provjerava mogućnost aplikacije da izvrši zahtjev i vrati rezultat, stoga je u definiciji

ove provjere moguće definirati različite protokole kojima se provjere vrše te izvršavanje naredbi komandnih linija. Ukoliko aplikacija ima duži inicijalni proces pokretanja, Kubernetes nudi provjeru pokretanja, odnosno provjeru koja omogućuje izvršavanje drugih provjera samo ako je ona sama uspješna. Općenito, Kubernetes podržava obavljanje provjera koristeći različite protokole, najčešći su HTTP, RPC i TCP utičnice. Sigurnosne provjere definiraju se u deklarativnim manifestima te ih je moguće prilagoditi svakom kontejneru.
U deklarativnoj datoteci se mogu definirati hardverski zahtjevi svakog kontejnera. Općenito, efikasnost se mjeri metrikom iskorištenja. Iskorištenje je definirano kao količina aktivno korištenih resursa podijeljena s dostupnom količinom resursa. Kubernetes dopušta korištenje dve različite vrste metrika, a to su minimalna i maksimalna količina resursa. Mjerne jedinice količine podataka moguće je definirati binarnim ili dekadskim jedinicama, a potrošnja procesorskih jezgri definira se cjelobrojnim literalima ili jedinicom milijezgre (engl. *milicore*)*.
Svakoj ljusci mogu se priključiti memorijski volumeni, a Kubernetes podržava različite vrste memorije s obzirom na primjenu:

- Perzistentni volumen: Ovaj tip volumena omogućuje zakup memorijskog prostora za trajnu upotrebu. Memorijski zakup se radi nad priključenom *internet Small Computer System Interface* (iSCSI) i Network File System (NFS) memorijama. U kontekstu korištenja memorije, Kubernetes vrti dvije vrste objekata: perzistentni volumen (engl. persistent volume) i zahtjev perzistentnog volumena (engl. persistent volume claim). Objekt perzistentni volumen provizioniraju memorijski prostor te ih je moguće postaviti da rade dinamički, a objekti zahtjevi perzistentnog volumena su objekti koje kreira korisnik administrator samih servisa te definira po kojemu od pristupnih načina namjerava koristiti pohranu. Zahtjevi se mogu ispuniti dinamički ili uz ljudsku interakciju.
- Prolazni volumen (engl. ephemeral volume): Prolazna memorija je ona čiji životni ciklus traje koliko i životni ciklus kontejnera.
- EmptyDir volumen: EmptyDir je vrsta prolaznog tipa koja koristi resurse čvora u Kubernetes grozdu. Životni ciklus memorije počne jednom kada se ljuska pokrene na čvoru, a prekida se u slučaju premještanja ljuske na drugi čvor ili pada servisa.
- HostPath volumen: Postoji mogućnost da ljuska direktno koristi memoriju domaćina Kubernetes čvora, ali ovaj pristup nije preporučen iz sigurnosnih razloga.
- ConfigMap volumen: Kubernetes omogućuje korištenje objekta tipa ConfigMap kroz specifičnu vrstu volumena.

Izgradnja mikroservisa obično zahtjeva parametriziranja vezana uz okolinu u kojoj se nalaze, a resursi za čuvanje konfiguracija, varijabli okoline i drugog sadržaja su objekti tajne i konfiguracijske mape. Poput ostalih vrsta objekta, definiraju se deklarativno. Konfiguracijske mape imaju tri najčešće primjene, a to su:

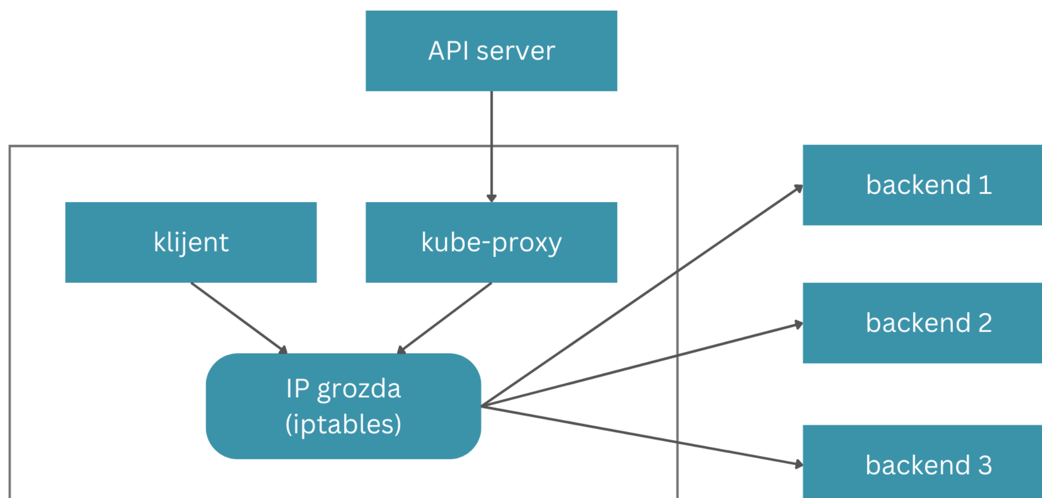
*Jedna milijezgra čini tisućiti dio jezgre centralne procesorske jedinice.

- sadržavanje datoteka koje je potrebno prenijeti u datotečni sustav kontejnera;
- čuvanje varijabli okruženja koje je potrebno dinamički mijenjati unutar kontejnera;
- pripremanje argumenata za izvršavanje naredbi u komandnoj liniji kontejnera. Za razliku od konfiguracijskih mapa, objekti tajne čuvaju podatke osjetljive prirode poput podataka servisnih računa, sigurnosnih tokena, privatnih ključeva. Poput konfiguracijskih mapa, sadržaj tajni je moguće prenijeti u datotečni sustav kontejnera.

Zbog dinamičnosti koju Kubernetes nudi nastaje potreba za novim mrežnim rješenjem. Koncept otkrivanja usluga (engl. *service discovery*) otkriva set alata čiji je cilj pronaći procese te adrese i vrata na kojima slušaju. Klasični DNS zakazuje jer mehanizam nije dovoljno zreo za ponovno rješavanje imena, u cache memoriji se počinju pamtiti ustajala preslikavanja te postoje poteškoće pridruživanja jednog imena s više IP adresa. Kubernetes ovaj problem interno rješava vlastitim otkrivanjem uslugama i vlastitim balansiranjem mrežnog tereta.

Otkrivanje usluga započinje kreiranjem objekta usluge koji definiramo deklarativnom konfiguracijom. U svakoj konfiguraciji usluge je potrebno definirati parametar birača (engl. *selector*) kojim pridružujemo uslugu ljusci. Usluga automatski dodijeli virtualnu IP adresu koja se zove IP grozda, a takva adresa je specifična jer omogućuje dijeljenje tereta prema više ljuski istog birača. Usluge omogućuju definiciju proizvoljnog prosljeđivanja vrata. Ispravno konfigurirane usluge omogućuju dostupnost ljuske unutar cijelog Kubernetes grozda, a ukoliko se dodatno aktivira osobitost usluga vrste NodePort na definiranim vratima, omogućuje se otvaranje servisa prema vanjskoj mreži tako da svaki čvor grozda preusmjeruje promet definiranih vratiju ka ljusci. Pri definiciji NodePort usluge, preporučuje se vlastoručno definirati balansiranje tereta, a ono može biti definirano postojećim Kubernetes objektima, fizičkim balansiranjem, dostupnim rješenjem u oblaku ili nekom od popularnih softverskih opcija. Mogućnost balansiranja tereta prema IP adresama grozda omogućuje interna komponenta `kube-proxy`. Ona promatra nove usluge putem API servera te usklađuje `iptables` zapise i prepisuje odredište na koje paketi moraju stići.

Rješenje poput NodePort usluge rješava problem četvrtog sloja OSI sustava što može biti validno za neke jednostavne potrebe korištenja TCP i UDP protokola. Međutim, norme je potrebno ispoštovati sve do sedmog sloja, odnosno HTTP komunikacije. U okolinama koje nisu kontejnerizirane obično HTTP i HTTPS zahtjevi dolaze do klasičnog alata za balansiranje tereta na vratim 80 ili 443 te prosljeđuje komunikaciju uzvodnom servisu. Kubernetes omogućuje komunikaciju sedmog sloja koristeći pritok (engl. *ingress*) koji oponaša komunikaciju u klasičnim okolinama. Pritok upravlja virtualnim domaćinima, a funkcionira tako da standardizira deklarativnu konfiguraciju, konfiguraciju pretvara u standardni Kubernetes objekt te više objekata sjedinjuje u jednu konfiguraciju za balansiranje tereta. Kubernetes u standardnoj instalaciji ne dodaje pritok, već omogućuje korisniku da odabere vlastiti. Neka od popularnih rješenja su pritoci već dostupni u oblaku, pritok otvorenog koda Contour organizacije Cloud Native



Slika 3.1: Konfiguracija i korištenje IP adrese grozda

Computing Foundation, Nginx Ingress, Traefik, Emissary i Gloo.
Pri osmišljavanju svake mikroservisne okoline koriste se osnovni Kubernetes objekti poput *deployment* objekata, usluga ili definicija memorije poput perzistentnih volumena. Po potrebi, moguće je definirati vlastite, prilagođene resursne definicije (engl. *custom resource definitions*) kao proširenje Kubernetes aplikacijskog sučelja u svrhu stvaranja vlastitih objekata. Prilagođene resursne definicije nisu dostupne u zadanim Kubernetes instalacijama te ih je potrebno omogućiti. Neki objekti koji upravljaju Kubernetesom unutar sistemskog imenskog prostora stvoreni su prema prilagođenim resursnim definicijama, primjerice uslužne mreže (engl. *service mesh*) [10]. Prije osmišljavanja definicije, potrebno je osmisliti njezinu namjenu te ju usporediti s mogućnostima konfiguracijske mape.

3.3 Kubernetes alati komandne linije

Upravljanje Kubernetes grozdom se najčešće obavlja korištenjem alata komandne linije. Postoje obvezni alati za upravljanje grozdom poput alata za administraciju grozda `kubeadm` i alata za komunikaciju s grozdom `kubectl`, no službene preporuke preporučuju druge alate s obzirom na primjenu.
Prethodno priložena instalacija Kubernetes grozda je slična onoj produkcijskoj, međutim za lokalni razvoj se češće upotrijebljavaju alati za generiranje Kubernetes grozda. Najčešći primjer takvog alata je Minikube – alat za kreaciju Kubernetes grozda s jednim čvorom. Alat sličan alatu Minikube koji omogućuje Kubernetes na više čvorova je Kind, međutim niti jedan od

ovih primjera nije preporučen za produkcijske okoline. U svrhe stvaranja produkcijskih Kubernetes grozdova postoje alati ovisni o drugim softverima za provizioniranje infrastrukture poput koda (engl. *infrastructure as a code*, skraćeno IaaC) kao što je Ansible, a primjer takvog Kubernetes alata je Kubespray. Pri vlastitom odabiru alata potrebno je uzeti u obzir da za korištenje alata za generiranje produkcijski spremnih grozdova treba imati znanje pozadinskih softvera, infrastrukture te pravog procesa instalacije poput onog ranije opisanog. Kada se koristi neki od alata za generiranje Kubernetes grozda, korisnik gotovo sigurno ne treba vlastoručno koristiti alat `kubeadm`.
Osim broja alata za stvaranje grozdova, postoje alati koji omogućuju svakodnevno korištenje u svrhu integracije softvera i konfiguracija unutar Kubernetes imenskih prostora. Osnovni službeni alat koji se koristi u svrhu upravljanja Kubernetes objektima je `kubectl`, a temelji se na Kubernetes API pristupu kontrolnoj okolini. S obzirom da se korisnik najčešće koristi ovim alatom, vrijedi spomenuti neke od mogućnosti.

Tablica 3.1: Pregled često korištenih naredbi alata `kubectl`.

Naredba i argumenti	Opis naredbe	Zastavice
`kubectl apply`	Spremanje definicije resursa iz datoteke ili standardnog unosa.	`-f` unos iz datoteke; `-k` spremanje svih konfiguracija iz direktorija; `-n` imenski prostor
`kubectl cluster-info`	Ispis statusa čvorova u grozdu.	`dump` detaljno pojašnjenje statusa
`kubectl delete {{ vrsta objeka }} {{ ime objekta }}`	Brisanje definicije resursa te svih njegovih instanci.	`-n` imenski prostor
`kubectl describe {{ vrsta objeka }} {{ ime objekta }}`	Detaljno opisivanje resursa ispisom ljudski čitljive deklarativne konfiguracije i isječcima iz dnevnika.	`-n` imenski prostor
`kubectl explain {{ vrsta objeka }} {{ ime objekta }}`	Resursi objašnjeni u formatu generirane dokumentacije.	`-n` imenski prostor

Naredba i argumenti	Opis naredbe	Zastavice
`kubectl logs {{ ime ljuske }} {{ ime kontejnera }}`	Ispis aplikacijskih zapisa iz kontejnera unutar ljuske. Ukoliko je samo jedan kontejner dostupan u ljusci, nije potrebno navesti ime kontejnera.	`-n` imenski prostor; `--tail {{ broj }}` ispisuje broj posljednjih redaka zapisa
`kubectl plugin`	Ispis dostupnih i omogućenih proširenja za Kubernetes.	
`kubectl port-forward {{ ime ljuske }} {{ vrata }}`	Omogućavanje prosljeđivanje vrata s glavnog čvora grozda na lokalnog domaćina.	`-n` imenski prostor
`kubectl proxy {{ vrata }}`	Otvaranje zamjenskog poslužitelja između lokalnog domaćina i Kubernetes API poslužitelja na definiranim vratima.	
`kubectl rollout {{ ime definicije ljuske }}`	Upravljanje brojem replika servisa.	`-n` imenski prostor; `--replicas` novi broj replika; `history`; `pause`; `restart`; `resume`; `status`; `undo`
`kubectl run {{ ime ljuske }}`	Pokretanje kontejnerske slike bez prethodno definiranih objekata.	`-n` imenski prostor; `--env []` varijable okruženja; `--port` izložena vrata ljuske

Aplikacije koje žive u Kubernetes imenskim prostorima mogu biti aplikacije u razvoju ili gotova

softverska rješenja koja se preuzimaju u vlastite svrhe. Zbog toga što je princip rada gotovih rješenja sličan u svakom slučaju integracije, već gotove definicije deklarativnih tekstualnih datoteka za Kubernetes objekte moguće je automatizirano integrirati pomoću alata Helm. Alat Helm se može usporediti s alatima za upravljanje paketima na operacijskim sustavima, a primjeri takvih su Advanced Package Tool (skraćeno APT), RPM, Yellowdog Updater Modified (skraćeno YUM), Dandified YUM (skraćeno DNF), Pacman, Zypper, Brew i Chocolatey. Helm nudi alat komandne linije koji u nekoliko koraka omogućuje instalaciju gotovih rješenja u Kubernetes imenske prostore, a radi na principu parametriziranja deklarativnih tekstualnih datoteka. Kako su Helm konfiguracije parametrizirane, pri instalaciji paketa je moguće popuniti vlastite parametre po kojima će se obaviti instalacija, a integritet instaliranog paketa se ne narušava ukoliko se manifesti mijenjaju ručno.
Uvrštavanje softvera na Kubernetes uključuje razumijevanje tri temeljna pojma:

1. Dijagram (engl. *chart*): Naziv za unaprijed konfigurirane Kubernetes resurse koji su spremni za integraciju.
2. Puštanje (engl. *release*): Naziv za obavljenu integraciju Kubernetes resursa.
3. Repozitorij: Lokacije na kojima su dostupni dijagrami.
4. Vrijednosti: Autori najčešće omogućuju prilagodbu dijagrama vlastitoj okolini parametriziranjem kritičnih vrijednosti.

Da bi se dijagram integrirao, potrebno je u Helm popis repozitorija dodati lokaciju na kojoj je dostupan dijagram. Ovaj proces je sličan onome kod alata za upravljanje paketa operacijskih sustava. Jednom kada je lokacija repozitorija definirana, prije integracije dijagrama je potrebno preuzeti i prilagoditi datoteku parametriziranih vrijednosti. U naredbi za instalaciju potrebno je navesti repozitorij s kojeg se preuzima dijagram te datoteka s vrijednostima ukoliko je definirana. Jednom kada se dijagram instalira počinje se nazivati puštanjem, a puštanjem se može naknadno upravljati, nadograđivati i pratiti status.

Poglavlje 4

Dostupnost i nadzor mikroservisa

4.1 Dostupnost mikroservisa

Dostupnost (engl. *availability*) je mogućnost servisa da bude operativan onda kada je potrebno obaviti operaciju. Ovaj koncept se često miješa s pouzdanošću (engl. *reliability*), odnosno mogućnosti servisa da obavi operaciju bez pogrešaka [11]. Dok pouzdanost uključuje pitanja integriteta izlaznih podataka, dostupnost postavlja pitanja šireg spektra, a ona se općenito tiču mogućnosti pristupa softveru. Dva pojma su usko povezana jer ukoliko sustav nije pouzdan, vjerojatno neće biti ni dostupan. Neki od najčešćih razloga koji čine sustav nedostupnim su iscrpljivanje dostupnih resursa, neuviđene mogućnosti povećanja prometa i tereta, povećan broj razdvojenih servisa i razvojnih timova, velike ovisnosti van sustava te tehnički dug prema softveru i hardveru.

Jedan od načina kako održati sustav visoko dostupnim je računanjem postotka vremena u kojemu je sustav dostupan, a postotak računamo tako da podijelimo razliku sekundi ukupnog peroda i sekundi koliko je sustav bio nedostupan s brojem sekundi ukupnog peroda. U kontekstu ove metrike svaki razvojni tim mora odrediti koji je prihvatljivi postotak dostupnosti. Primjerice, prihvatljiva dostupnost se izražava devetkama, odnosno kako je prikazano sljedećom tablicom [12]. Obično se okvirni postotak dostupnosi uzima se u odnosu na redovito održavanje, najčešće održavanje hardvera, ažuriranje novijih verzija softvera te produkcijske okoline.

Tablica 4.1: Postotak dostupnosti i rezultirajuća dozvoljena nedostupnost u različitim vremenskim rasponima (godina, kvartal, mjesec, tjedan i dan).

Dostupnost %	U godini	U kvartalu	U mjesecu	U tjednu	U danu
90% (“jedna devetka”)	36.53 dana	9.13 dana	73.05 sati	16.80 sati	2.40 sati

Dostupnost %	U godini	U kvartalu	U mjesecu	U tjednu	U danu
95% ("jedna i jedna polovina devetke")	18.26 dana	4.56 dana	36.53 sati	8.40 sati	1.20 sati
99% ("dve devetke")	3.65 dana	21.9 sati	7.31 sati	1.68 sati	14.40 minuta
99.5% ("dve i jedna polovina devetke")	1.83 dana	10.98 sati	3.65 sati	50.40 minuta	7.20 minuta
99.9% ("tri devetke")	8.77 sati	2.19 sati	43.83 minuta	10.08 minuta	1.44 minuta
99.95% ("tri i jedna polovina devetke")	4.38 sati	65.7 minuta	21.92 minuta	5.04 minuta	43.20 sekundi
99.99% ("četiri devetke")	52.60 minuta	13.15 minuta	4.38 minuta	1.01 minuta	8.64 sekundi
99.999% ("pet devetki")	5.26 minuta	1.31 minuta	26.30 sekundi	6.05 sekundi	864.00 milisekundi
99.9999% ("šest devetki")	31.56 sekundi	7.89 sekundi	2.63 sekundi	604.80 milisekundi	86.40 milisekundi
99.99999% ("sedam devetki")	3.16 sekundi	0.79 sekundi	262.98 milisekundi	60.48 milisekundi	8.64 milisekundi
99.999999% ("osam devetki")	315.58 milisekundi	78.89 milisekundi	26.30 milisekundi	6.05 milisekundi	864.00 mikrosekundi
99.9999999% ("devet devetki")	31.56 milisekundi	7.89 milisekundi	2.63 milisekundi	604.80 mikrosekundi	86.40 mikrosekundi

Dostupnost obično nije pitanje kojim se tehničko osoblje bavi sve dok se ne pokaže češći trend nedostupnosti, a tada je ispravan korak mjeriti postotak, utvrditi postotak koji odgovara slučaju primjene te uvesti mehanizme koji bi spriječili nedostupnost. Preporučuje se kontinuirano pratiti promjenu postotka dostupnosti te analizirati koje nove navike i promjene čine poboljšanje. U slučaju da mehanizmi za primjenu visoke dostupnosti podbace, potrebno je takve slučaje dokumentirati i opisati za buduće situacije.
Da bi se smanjilo vrijeme potrebno za redovito održavanje, razvoj bi se trebao usmjeriti ka automatizaciji što je više moguće ručnih procesa, a u produkcijske okoline uopće ne uključivati ljudski faktor. Za sve korake koji se kontinuirano ponavljaju je potrebno testirati promjene, omogućiti trećoj strani provjeru novih promjena, koristiti kontrolu verzija, koristiti alate koji omogućuju obavljanje zadataka na više točaka paralelno, omogućiti konzistentnu upotrijebu resursa itd. Koncentracija na ovakve detalje zahtijeva predstavljanje različitih automatizacijskih rješenja, primjerice automatizacije upravljanja virtualnim okružjem i primjena koncepta kontinuirane integracije i kontinuirane isporuke.
Upravljanje rizikom omogućuje prepoznavanje potencijalnih nedostupnosti prije nego što se dogode, a potrebno je utvrditi šansu za takvim događajem te kakve posljedice bi nedostupnost izazvana poznatim rizikom imala na postojeći sustav. U konačnici, u svijetu poslovnog softvera svaka nedostupnost znači gubitak novaca, a može uzrokovati posljedice poput gubitka podataka te pogreške u operacijama. Prepoznati rizici se nabrajaju u matrici rizika koja sadrži osnovne podatke poput tima odgovornog za točku neuspjeha, mogućnost neuspjeha, ozbiljnost posljedica, akcije poduzete za prevenciju rizika te plana u slučaju pojave rizika. Izvore ideja za potencijalne rizike čine znanje razvojnog tima, poznate točke koje zahtijevaju više podrške, poznate prijetnje i slabosti, područja u kojima nedostaju značajke ili koja imaju slabe performanse, uzorci skoka prometa, brige iskazane od strane klijenata i korisnika te tehnički nedostaciu sustavu. Matricu rizika je potrebno redovito ažurirati, po mogućnosti pri svakoj pojavi rizika. Pri svakom ažuriranju, u obzir treba uzeti promjene u ozbiljnosti posljedica i mogućnosti neuspjeha, rankove rizika prema ozbiljnosti, ali ne treba zakinuti ni one manje kritične.
Osim redovitog upravljanja rizikom, prema poznatim rizicima trebaju postojati planovi smanjenja štete. Tipičan primjer je mogućnost prestanka rada baze podataka – tada se prema planovima smanjenja štete koriste replicirane instance baze na drugim instancama ili primjena hardverske zaštite podataka korištenjem RAID tehnike rada diskova. Ukoliko se pogreška dogodi na jednom od identificiranih rizika, unaprijed je moguće definirati plan oporavka servisa koji se definira akcijama koje zaustavljaju problem, akcijama koje smanjuju ili zaobilaze problem, obavještavanje korisnika te dežurnih osoba u razvojnom timu. Ključno je da se plan oporavka definira jasno i dovoljno detaljno tako da se problem može što prije otkloniti.
Ukoliko se aktivnosti upravljanja rizikom primijenjuju na mikroservisnu arhitekturu, svaki servis može imati vlastite prijetnje koje je potrebno uočiti. Općenito, servisi međusobno komuni-

ciraju te su ovisni jedni o drugima, a već u fazi arhitekture sofvera treba predvidjeti mehanizme koje će sofver koristiti ukoliko neki od ovisnih servisa zakaže. Unaprijed treba uzeti u obzir neke od mogućih znakova da ovisnost zakazuje:

- Povratna informacija operacije nije razumljiva, vraća fatalnu pogrešku ili ne vraća potrebni rezultat;
- Povratna informacija nije stigla na ciljani servis ili dolazi sporo.

U slučaju problema s povratnom informacijom, potrebno je obaviti ispravne akcije na ciljanom servisu.

- Dobrohotna degradacija: Ukoliko ovisnost zakaže, može li servis nastaviti s radom? Ukoliko je moguće, treba omogućiti djelomično obavljanje operacije.
- Dobrohotno povlačenje: Ukoliko nije moguće obaviti operaciju djelomično, treba omogućiti servisu neuspjeh koji ne djeluje razarajuće.
- Što ranije podbacivanje: Ukoliko je moguće unaprijed zaključiti da operacija neće biti uspješna, ne treba ju niti provoditi. Razlozi tom pristupu su očuvanje resursa, održavanje responzivnosti servisa te smanjenje kompleksnosti pogreške.
- Validiranje korisničkog unosa.

4.2 Nazor mikroservisa

Nadzor (engl. *monitoring*) je težak za čvrsto definirati jer u kontekstu svakog sustava može značiti nešto drugo. Općenito, nadzor mikroservisa obuhvaća kontrolu metrika komponenti sustava mikroservisa, uzbunjivanje (engl. *alerting*) u slučaju neobičnih ponašanja softvera, planiranje s obzirom na povijesne podatke ponašanja softvera. Nadzor se tretira kao izvor informacija za održavanje zdravih produkcijskih i poslovnih sustava [13]. Nadzorom servisa se bave različite uloge u razvoju softvera, a najčešće su to sistemski inženjeri, devops inženjeri, inženjeri pouzdanosti te inženjeri za osiguranje kvalitete. Nadzor se obično dijeli na sljedeće komponente:

- Metrike (engl. *metrics*): Brojčane vrijednosti vezane za točke u vremenu u kojima su zabilježeni sistemski resursi, aplikacijske operacije i poslovne karakteristike. Obično su ove vrijednosti agregiranje zajedno s identifikatorom događaja.
- Zapisivanje (engl. *logging*): Osim što se spremaju metrike, u sklopu događaja u vremenu se mogu zapisati različiti podaci, primjerice izlaz aplikacije te metapodaci.
- Traganje (engl. *tracing*): Koncept koji proširuje zapisivanje tako da jedinstven događaj spaja s redovnim spremljenim zapisom.
- Uzbunjivanje: Ukoliko se jasno definiraju uvjeti, na temelju primjećenih uvjeta nadzor može kontaktirati čovjeka.
- Vizualizacija: Sve zapisane brojčane vrijednosti je moguće grafički prikazati.

S obzirom da je svaki sustav specifičan, treba postaviti pitanje koje metrike treba prikupljati.

Google predlaže zlatno pravilo četiri signala [14], a oni su:

- Latencija: vrijeme potrebno da se zahtjev posluži;
- Promet: broj zahtjeva koji se kreiraju;
- Pogreške: stopa zahtjeva na koje sustav ne odgovori;
- Saturacija: količina rada koju sustav ne obradi ili stavlja u radni stog.

U novijoj literaturi se uz nadzor spominje uočljivost (engl. *observability*) kao mjera kojom se može pronaći uzrok stanja sustava iz korisničke interakcije, a ne metrika.

Visoka dostupnost zahtjeva planiranje konfiguracije čvorova unaprijed, a najčešće dvije konfiguracije čvorova podrazumijevaju:

- Konfiguracija aktivnog ka aktivnom: Grozd se sastoji od barem dva čvora koji među sobom imaju balans tereta te obavljaju isti posao simultano. Teret se raspoređuje po *round robin* algoritmu, a redundancija omogućuje dostupnost u slučaju kritičnih pogrešaka.
- Konfiguracija aktivnog ka pasivnom: Grozd se sastoji od barem dva čvora koji među sobom imaju balans tereta, ali samo jedan aktivno radi dok se drugi tretira kao rezerva u slučaju kritične pogreške.

Pristup aktivnog ka pasivnom je stariji te se u modernom softveru najčešće teži aktivnog ka aktivnom pristupu. Kubernetes služi kao primjer implementacije oba načina rada visoke dostupnosti jer radnici mogu raditi isti posao simultano, ali kontrolna okolina je samo jedna. Kontrolnu okolinu nije preporučeno obavljati simultano na više čvorova, već ju se replicira na drugu lokaciju koja preuzima odgovornost u slučaju zakazivanja one glavne.

Kategorije u koje se dijeli nadzor su najčešće nadzor crne kutije i nadzor bijele kutije. Crna kutija je sustav kojega se promatra izvan sustava te mu je pristup ograničen. Primjer akcija koje se vrše u sklopu nadzora crne kutije su provjere reagira li domaćin na ICMP zahtjeve*, ispitivanje je li definirani TCP port otvoren, odgovara li aplikacija s očekivanim podacima i statusnim kodovima na zahtjeve, provjera je li proces aktivan na domaćinu i slično. Osim što nadzor crne kutije ispituje neposrednu okolinu servisa, bavi se ispitivanjem balansiranja tereta te vatrozida.

Nadzor bijele kutije omogućuje podatke o unutarnjim stanjima i performansama kritičnih stanja, a korištenje telemetrije može se pokazati kao snažna strategija promatranja zdravlja komponenti. Podaci o bijeloj kutiji obično dolaze iz izvezenih zapisa (engl. *logs*), strukturiranih agregiranih događaja koji se prijavljuju nadzornim alatima ili agregiranih podataka zadržanih u memoriji dostupnih na krajnjoj točki, primjerice izvoz podataka za alat Prometheus. Nije nužno da sve usluge imaju mogućnost izvoza podataka. U praksi se često koriste servisi specifično namijenjeni prilagođavanju zapisa za obradu nadzora.

U kontekstu razlikovanja sustava nadzora po principu prikupljanja podataka, alati za nadzor se

*Internet Control Message Protocol je protokol za provjeru ispravnog rada Internetskog protokola. Njegova često korištena implementacija je alat otvorenog koda `ping`.

dijele u dvije skupine: nadzor gurnutih podataka i nadzor povučenih podataka. Sustavi nadzora s gurnutim podacima podrazumijevaju da se podaci odmah šalju sustavu za nadzor te nude mogućnost odabira frekvencije kojom će se podaci gurati. U slučaju takvih sustava nadzora, nastaje problem kompleksnih i nepraktičnih slanja podataka, a najpopularnije rješenje takvog nadzora je ELK stog, odnosno kombinacija alata Elasticsearch, Logstash i Kibana. U kontrastu, sustavi povučenih nadzora samostalno prikupljaju podataka s definiranih krajnjih točki, a jedno od poznatih takvih rješenja je alat Prometheus. Najčešći argument u korist nadzora povučenih podataka je da kod nadzora gurnutih podataka u gusto naseljenim okolinama mogu nastati mrežni zastoji zbog nebalansiranog tereta. Servisi se sami prijavljuju nadzoru gurnutih podataka te takav nadzor ima više poteškoća detektirati treba li im se neki servis javiti, je li još dostupan i treba li nešto poduzeti po tom pitanju.

4.3 Osiguranje visoke dostupnosti Kubernetes grozda

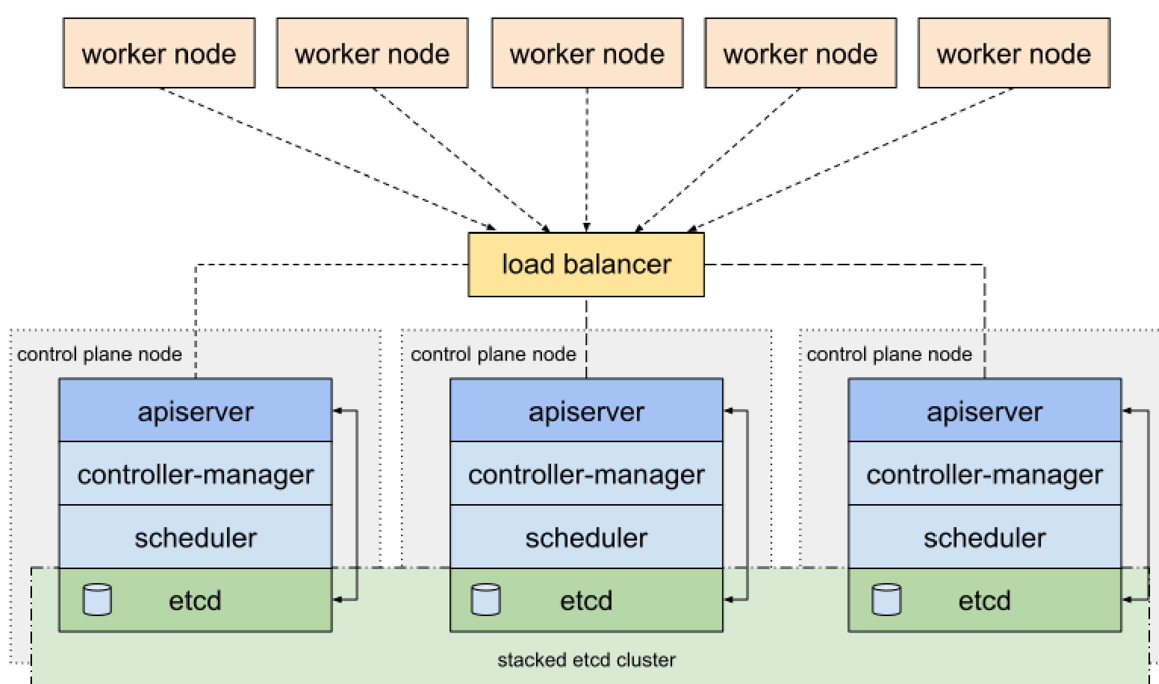
Dostupnost je pojam koji obuhvaća softver i infrastrukturu kao važne aspekte osiguravanja kvalitete. Visoka dostupnost Kubernetes grozda ovisi o dizajnu grozda, a unaprijed je potrebno definirati osnovnu topologiju u koju je moguće dinamički dodavati i smanjivati broj dostupnih čvorova.
U svrhu stvaranja rezervnih kopija stanja podataka u grozdu koristi se konzistentna i visoko dostupna pohrana na principu ključa i vrijednosti *etcd*. U pohranu se spremaju svi Kubernetes objekti, a potrebno je redovito održavati rezerve za slučaj katastrofalne pogreške. Osim što sprema rezerve, omogućuje enkriptiranje rezervi tako da ostanu sigurne. Rezerve stanja podataka moguće je definirati na dva načina: ugrađenim snimkom ili snimkom volumena [15]. Pri osmišljavanju topologije treba uzeti u obzir da se ne preporučuje dinamički skalirati *etcd* grozdove jer se time ne mijenjaju performanse pohrane. Spremanje stanja te korištenje rezerve se omogućuje moćnim alatom komandne linije, a moguće ga je automatizirati po želji. Prije stvaranja produkcijskog Kubernetes grozda je poželjno upoznati se s radom alata *etcd*, a ključne informacije su sljedeće:

- Zbog toga što je *etcd* grozd distribuiran sustav po principu vođe, *etcd* grozd uvijek ima neparan broj članova. Jedna od zadaća čvora vođe je promatrati otkucaje srca svih članova grozda da bi se održala stabilnost grozda.
- Važno je osigurati da se ne iscrpe svi dostupni resursi jer je *etcd* grozd ovisan o ispravnom radu mreže i zapisima na tvrdim memorijama. Trošenje svih resursa može uzrokovati zastoj nekog od čvorova, a u tom slučaju se onemogućuju promjene stanja u *etcd* grozdu, a zbog toga se također onemogućuje promjena stanja u Kubernetes grozdu.
- *etcd* grozd može činiti jedan ili više neparnih članova te je među njima potrebno uspostaviti sigurnosne mjere poput alata za balansiranje tereta, razmjenu ključeva te sigurnu

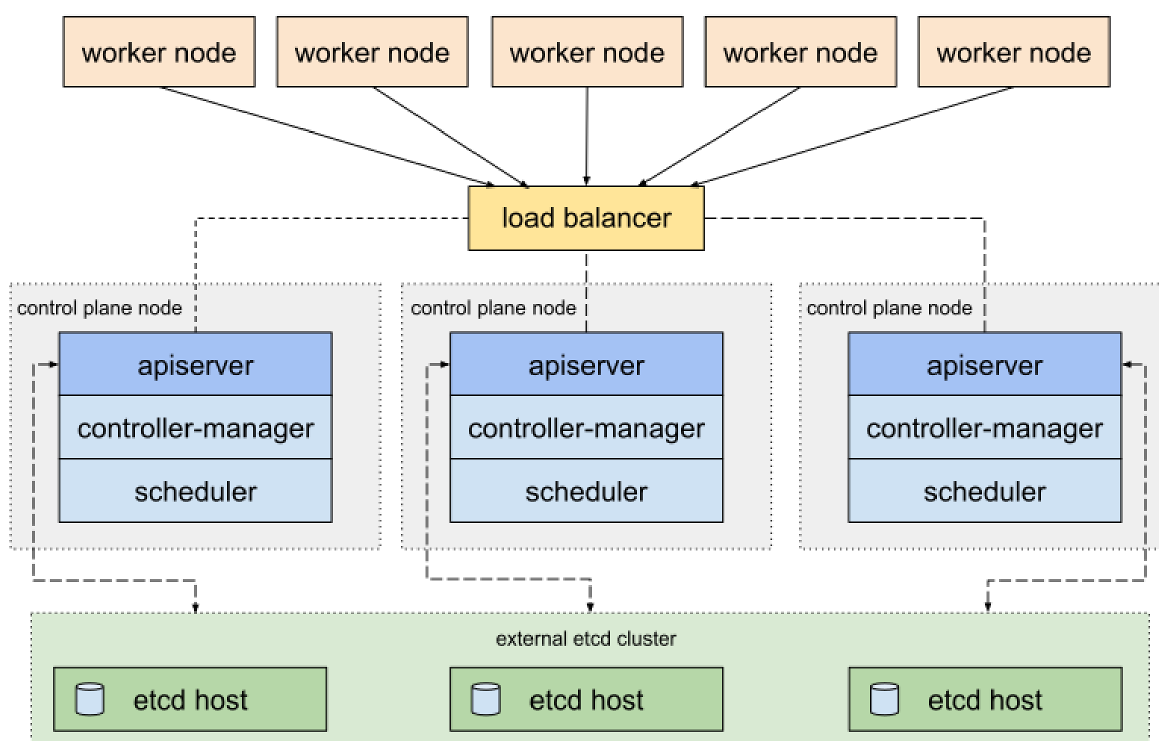
HTTPS komunikaciju.

Svaka kontrolna ravnina Kubernetes grozda sadrži instance osnovnih sistemskih koncepata: `kube-apiserver`, `kube-scheduler` te `kube-controller-manager`. Ovisno o željenom pristupu pisanja stanja u pohranu, razmatraju se dvije opcije topologija [15]:

- Topologija stoga: Čvorovi kontrolnih ravnina su također čvorovi *etcd* grozda. Ovo čini čvor kontrolne ravnine i čvor *etcd* grozda usko spojenima te ukoliko jedno od toga zakaže, međusobno si ugrožavaju stabilnost. Takav slučaj implicira da je za visoko dostupnu Kubernetes uslugu potrebno imati barem tri čvora kontrolne ravnine.
- Vanjska topologija: Čvorovi etcd grozda odvojeni su od Kubernetes čvorova te komuniciraju koristeći `kube-apiserver` poput drugih radnih čvorova Kubernetes grozda. Ova opcija je bolje prilagođena visokoj dostupnosti jer se čvorovi dvaju grozdova ne narušavaju međusobno.



Slika 4.1: Topologija stoga – Kubernetes čvorovi i etcd čvorovi. Izvor: [15]



Slika 4.2: Vanjska topologija – Kubernetes čvorovi i etcd čvorovi. Izvor: [15]

Poglavlje 5

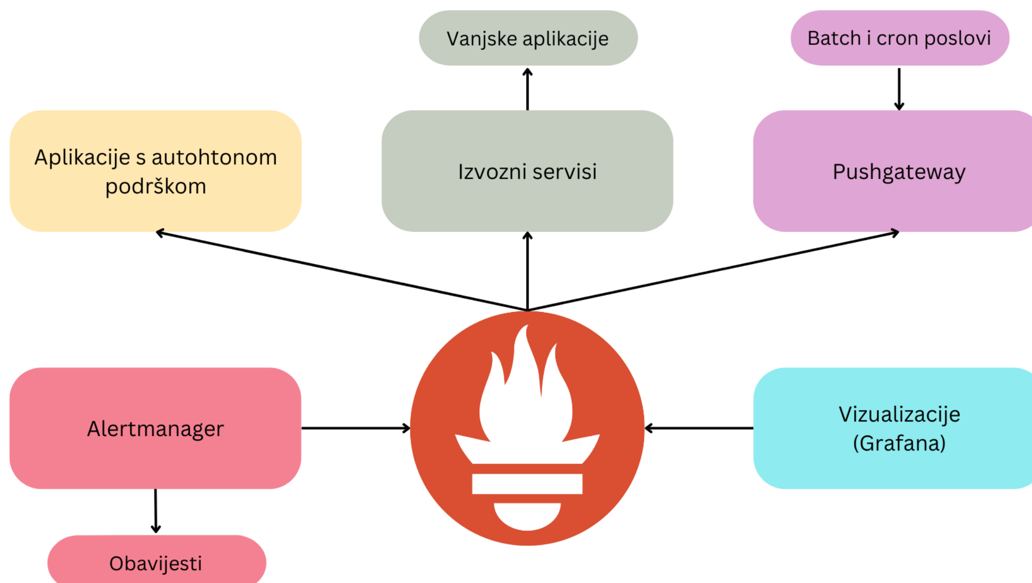
Alati Prometheus i Grafana

Alat Prometheus je alat za nadzor otvorenog koda baziran na vremenskim događajima, inicijalno napravljen za SoundCloud mikroservise po uzoru na Googleov Borgmon. Funkcionira na način da prikuplja podatke HTTP zahtjevima prema drugim domaćinima i servisima na njihovim krajnjim točkama. U svrhu manipulacije podacima, Prometheus ima vlastiti upitni jezik – PromQL.

Kod implementacije alata Prometheus u vlastitim okolinama razmatramo nekoliko komponenti:

- Prometheus server koji prikuplja povijesne podatke, pohranjuje ih, obavješćuje te čini podatke spremne za upite;
- Alertmanager komponenta koja prima upozorenja Prometheus servera te obavlja daljnju komunikaciju upozorenja;
- Push Gateway koji prima gurnute podatke poslova kratkog trajanja poput tempiranih poslova (engl. *cron*) i serijskih poslova (engl. *batch*);
- Aplikacije koje podržavaju format podataka koji Prometheus može preuzeti na HTTP krajnjim točkama i iskoristiti za analizu;
- Izvozni servisi (engl. *exporters*) koje razvija Prometheus zajednica;
- Web aplikacije na kojima se poslužuju kontrolne ploče (engl. *dashboards*) prvog ili trećeg izvora.

Zbog toga što aplikacije imaju ograničenja na način kako će dijeliti svoje metrike, zajednica kontinuirano radi na razvijanju zasebnih izvoznih servisa. Ideja izvoznih servisa je da neprilagođene podatke transformira u one čitljive Prometheusovom serveru te da izloži krajnju točku s koje je moguće preuzeti događaje u ispravnom formatu. Izvozni servisi su obično u neposrednoj okolini promatrane aplikacije, često integrirani po obrascu prikolice (engl. *sidecar*) zato što trebaju imati pristup zrnu (engl. *kernel*) domaćina na kojemu je aplikacija montirana. Tada servis može pratiti osnovne metrike crne kutije poput broja zapisa u memoriju, broj ulaza i izlaza, hardverske performanse i slično. Zajednica održava izvozne servise za popularne programske jezike i softvere, a popis službeno i neslužbeno podržanih servisa dostupan je u službenoj do-



Slika 5.1: Komponente okoline koju nadzire Prometheus

kumentaciji alata Prometheus. Sami izvozni servisi su najčešće nezahtjevnih performansi, a ukoliko se tijekom razvoja pokaže kao bolja opcija, izvorni servis može biti autorski.

5.1 Opis komponenti sustava Prometheus

Alertmanager komponenta je dio sustava koja je odgovorna za obavještavanje korisnika o zastojima. S obzirom da je dostupnost glavna motivacija za korištenje nadzornih sustava, smisleno je da nadzorni sustav ima komponentu koja osigurava brzu reakciju. Unutar postavki servisa Alertmanager se definiraju rute prema kojima se šalju obavijesti, a to podrazumijeva komunikacijske kanale, njihove podatke te podatke o korisnicima koji trebaju biti obaviješteni. U svrhu upravljanja obavijestima i pravilima, Alertmanager nudi web sučelje radi osiguranja korisnički prijateljske konfiguracije. Prometheus koristi Alertmanager API za pohranjivanje uzbuna u pohranu, agent za obavijesti prati promjene u pohrani te dojavljuje korisniku po definiranoj ruti, a one najčešće su elektroničkom poštom, SMS porukom ili automatiziranim robotima na aplikacije za čavrljanje kao što je Slack, Discord, Microsoft Teams, RocketChat ili neki drugi.
Ukoliko je instalacija Prometheusa planirana u Kubernetes grozdu, dodatna komponenta Prometheus okoline je Prometheus Operator. Operator* je proces koji stvara sintezu između Prometheusa i Kubernetes prilagođenih resursa. Glavna značajka Operatora je dinamički ažurirati Prometheus konfiguraciju u svrhu nesmetanog dodavanja novih čvorova na promatranje. Prila-

*Koncept operatora osmislio je razvojni tim CoreOS-a, a u kontekstu Kubernetes administracije podrazumijeva dio softvera koji implementira najbolje prakse za konfiguraciju željenog softvera.

gođeni resurs koji koristi Operator naziva se ServiceMonitor, a takav objekt sadrži informacije o krajnjoj točki koja će se promatrati, njezinu lokaciju te odgovarajuće oznake kako Operator prepoznaje da se radi o ispravnom servisu.
Krajnje točke s kojih će se prikupljati podaci za nadzor mogu biti uključene unutar servisa ili izvan. U slučaju kada iz sigurnosnih razloga nije preporučljivo izložiti servis, može se uvesti pomoćni servis koji služi isključivo serviranju dnevničkih zapisa čitljivih Prometheusu, a takav servis se naziva izvoznik (engl. *exporter*). Izvoznici mogu biti priključeni svim vrstama servisa neovisno o načinu posluživanja, okolini i infrastrukturi. Za transakcije između servisa i izvoznika je važno samo da su mrežno povezani. U Kubernetes okolinama, prilagođeni resursi vrste ServiceMonitor će promatrati krajnju točku izvoznika u svrhu prikupljanja podataka. U zajednici razvoja otvorenog koda je česta praksa da se u sofver već integrira osobitost koja omogućuje nadzor čestim alatima, a jedan od takvih je Prometheus. Zajednica koja se bavi razvojem alata Prometheus također razvija izvoznike za često korištene softvere i knjižnice za programske jezike, a informacije o takvim projektima dostupne su u službenoj dokumentaciji. Izvoznik je moguće vlastoručno kreirati i održavati ukoliko je to poželjno za razvoj.

5.2 Prikupljanje, pohrana i obrada podataka

Prometheus za potrebe nadzora koristi prilagođenu strukturu baze podataka utemeljenu na vremenskim serijalima. Podaci se s krajnjih točaka prikupljaju u definiranim vremenskim intervalima te se u realnom vremenu mogu prikazivati u grafičkom ili tabličnom formatu. Razlog zašto Prometheus ne koristi relacijske ili NoSQL baze podataka je da one ne daju željene performanse u slučaju prikupljanja velikih količina podataka s većeg broja kranjih točaka te posluživanja u realnom vremenu. Model podataka u ovom slučaju je kombinacija dvije vrste baza podataka, a podaci se spremaju tako da sadrže tri glavne komponente: vremenska marka, ključna vrijednost metrike te kontekst prikupljanja metrike koji je izražen kroz asocirane vrijednosti ključeva i vrijednosti.
Kada se podaci prikupljaju s krajnjih točaka, čuvaju se u RAM memoriji Prometheus domaćina najviše dva sata, a onda se arhiviraju. S obzirom da pisanje u memoriju predstavlja jednu od najskupljih vrsta operacija, na taj način se dramatično smanjuje broj operacija koje upisuju na disk, a ubrzava se proces odgovaranja na upite. Taj pristup ima slabost, a to je da ukoliko domaćin doživi fatalnu pogrešku, postoji mogućnost gubitka podataka. Prometheus rješava taj problem pisanjem zapisa unaprijed (engl. *write-ahead log*) tako da perzistentno čuva stanje memorije. Jednom kada istekne dva sata čuvanja podataka, podaci koji se spremaju u perzistentnu memoriju postaju nepromijenjivi. Ukoliko se podaci brišu iz perzistentne memorije, po njihovom brisanju u memoriji ostaju datoteke koje zapisuju metapodatke brisanih podataka.
Vlastiti prisup pohrani podataka je uvjetovao osmišljanju vlastitog upitnog jezika te je zato para-

lelno s alatom Prometheus razvijen njegov vlastiti upitni jezik PromQL. Ako promatramo informaciju kroz tri temeljne komponente (vremensku marku, ključnu vrijednost i kontekst metrike), jedna vremenska serija izražava se izrazom <ime_metrike>[{<ključ_1="vrijednost_1">,<ključ_n="vr
<identifikator_numerička_vrijednost> te s priključenom vremenskom markom točnosti u milisekundama. Dopuštena imena metrika sadrže malena i velika slova abecede engleskog jezika, donje povlake (_), dvotočke (:) te arapske numerale od 0 do 9. Vrijednosti konteksta izražene parovima ključeva i vrijednosti smiju biti imenovana na isti način kao imena metrika, ali ne smiju sadržavati dvotočke.

Kod planiranja nadzora treba uzeti u obzir opširnost sustava koji promatra sustav nadzora. S obzirom da Prometheus sprema vremenske serijale, ima vlastite mehanizme kako da održi spremanje vrste metrike na memorijski prihvatljiv način bez da degradira performanse nadzora. Potrebno je uzeti u obzir okviran broj jedinstvenih metrika koje će biti nadzirane, hoće li to biti velik broj servisa s malim brojem metrika ili mali broj servisa s velikim brojem metrika, ali prema uvjetima okoline u kojoj se nadzor nalazi treba postaviti granice, odnosno kardinalnost nadzora. Metrike je potrebno pažljivo definirati na način da metrici nije subjekt podatak koji bi povećavao dimenzionalnost pohrane podataka, primjerice e-mail adrese, korisnička imena ili transakcije koje se događaju unutar softvera promatranog po principu bijele kutije. Prometheus generira četiri temeljne vrste metrika izraženih grafovima.

Tablica 5.1: Vrste metrika alata Prometheus.

Vrsta metrike	Opis metrike	Primjer
Brojač (engl. *counter*)	Kumulativna metrika čije se vrijednosti uvijek povećavaju. Jedini slučaj u kojemu brojač pada je kada se metrika ponovno započinje, a tada joj je prva vrijednost nula.	broj svih paketa koje je Prometheus primio u periodu vremena
Mjerač (engl. *gauge*)	Metrika koja mjeri količinu zadane vrijednosti u periodu vremena.	slobodan memorijski prostor dostupan Grafana instanci

Vrsta metrike	Opis metrike	Primjer
Histogram	Metrika izražavanja unaprijed uračunatih agregacija statističkog značenja.	broj HTTP zahtjeva koje je Prometheus primio u periodu vremena
Sažetak (engl. *summary*)	Metrika izražavanja unaprijed uračunatih agregacija statističkog značenja uključujući metapodatke poput latencije ili veličine.	maksimalno trajanje grupe u kvantitativnoj vrijednosti po sekundama

Prometheusov način rada definira se u ljudski čitljivim konfiguracjama, ali nudi se opcija alata komandne linije kojim je moguće naredbama uključiti iste osobitosti. U slučaju klasične instalacije Prometheusa, konfiguracijska datoteka naziva se `prometheus.yml`, a bez nje se Prometheus odbija pokrenuti. Unatoč čestom priključivanju Prometheusa i Grafane kao dva alata koji dolaze u paru, Prometheus također nudi vlastito web sučelje na kojemu je moguće vidjeti krajnje točke koje Prometheus promatra, testirati PromQL upite te vidjeti trenutnu konfiguraciju koja uključuje sve parametre definirane unutar konfigurabilne datoteke i naknadno priključene korištenjem alata komandne linije. Vrijedi napomenuti da Prometheus i Grafana ne ovise jedan o drugome te mogu služiti individualno u kombinaciji s drugim alatima.

Poglavlje 6

Primjer nadzora mikroservisne arhitekture

Sljedeća arhitektura aplikacija osmišljena je u svrhu primjera nadzora mikroservisnih aplikacija u orkestriranom kontejneriziranom okruženju. Alat Prometheus nadzirat će grupu manjih servisa koji zajedno imaju svrhu demonstrirati više načina komunikacije servisa na primjeru pozivanja aplikacijskog sučelja popisa sveučilišta po državama. Servisi koji čine primjer su Prometheus te njegove komponente, UniApp aplikacijsko sučelje, alat za distribuirani poredak zadataka Celery, web poslužitelj Nginx u ulozi obrnutog zamjenika (engl. *reverse proxy*) i distribuirani posrednik za dijeljenje poruka RabbitMQ.

6.1 Postavljanje alata Prometheus i komponenti alatom Helm

S obzirom da se uz Prometheus koriste dodatni servisi, jednostavan način za brzu i laku integraciju je koristiti alat Helm. Dijagrame za instalaciju Prometheusa također službeno održava razvojni tim te su sve podržane verzije dostupne za preuzimanje. Za početak, potrebno je u repozitorije dijagrama dodati `prometheus-community`. S obzirom da Prometheus zajednica razvija izvoznike za popularna softverska rješenja, iz ovog repozitorija je moguće preuzeti broj korisnih dijagrama. Sljedećnom naredbom dodaje se službeni Prometheus Charts repozitorij te se osvježuju podaci o dodanim repozitorijima.

```
$ helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts
$ helm repo update
(...)
```

Dijagrami za Helm instalacije obično sadrže datoteke u kojima se mogu definirati vlastite vrijednosti na parametre instalacije. Ukoliko se oni koje se neće mijenjati izbrišu iz takve dato-

teke, uzimaju se zadane vrijednosti. Kada se ispuni datoteka, potrebno je izvršiti instalaciju. Dijagram tada postaje puštanje naziva prom, a naziv puštanja je važan zbog definicije drugih objekata.

```
$ helm show values prometheus-community/kube-prometheus-stack > prom-values.yaml
# nakon uređenja datoteke prom-values.yaml
$ helm install prom prometheus-community/kube-prometheus-stack -f prom-values.yaml
```

Objekte koje instalacija dijagrama definira unutar Kubernetesa moguće je unaprijed vidjeti unutar repozitorija ili na nekom od podržanih web sjedišta. Nakon instalacije je najbolje stvorene objekte dobiti korištenjem Kubernetesovog alata komandne linije.

```
# Ukoliko je imenski prostor u koji je Kubernetes instaliran populiran drugim servisin
$ kubectl get all

NAME                                                    READY   STATUS    RESTART
pod/alertmanager-prom-kube-prometheus-stack-alertmanager-0   2/2     Running   1 (4m31
pod/prom-grafana-587c8db8b8-v4znz                         3/3     Running   0
pod/prom-kube-prometheus-stack-operator-64697d46f7-jbj66  1/1     Running   0
pod/prom-kube-state-metrics-54c4ff848b-bkxwt              1/1     Running   0
pod/prom-prometheus-node-exporter-q595t                   1/1     Running   0
pod/prometheus-prom-kube-prometheus-stack-prometheus-0    2/2     Running   0

NAME                                           TYPE        CLUSTER-IP       EXTERNA
service/alertmanager-operated                  ClusterIP   None             <none>
service/kubernetes                             ClusterIP   10.96.0.1        <none>
service/prom-grafana                           ClusterIP   10.99.190.174    <none>
service/prom-kube-prometheus-stack-alertmanager ClusterIP  10.107.219.12    <none>
service/prom-kube-prometheus-stack-operator    ClusterIP   10.104.177.103   <none>
service/prom-kube-prometheus-stack-prometheus  ClusterIP   10.107.97.48     <none>
service/prom-kube-state-metrics                ClusterIP   10.106.30.111    <none>
service/prom-prometheus-node-exporter          ClusterIP   10.99.175.240    <none>
service/prometheus-operated                    ClusterIP   None             <none>

NAME                                         DESIRED   CURRENT   READY   UP-TO-DATE
daemonset.apps/prom-prometheus-node-exporter   1         1         1       1

NAME                                             READY   UP-TO-DATE   AVAILABLE
deployment.apps/prom-grafana                     1/1     1            1
```

```
deployment.apps/prom-kube-prometheus-stack-operator    1/1     1            1
deployment.apps/prom-kube-state-metrics                1/1     1            1

NAME                                                             DESIRED   CURRENT   R
replicaset.apps/prom-grafana-587c8db8b8                          1         1         1
replicaset.apps/prom-kube-prometheus-stack-operator-64697d46f7   1         1         1
replicaset.apps/prom-kube-state-metrics-54c4ff848b               1         1         1

NAME                                                                   READY   AGE
statefulset.apps/alertmanager-prom-kube-prometheus-stack-alertmanager   1/1     5m1s
statefulset.apps/prometheus-prom-kube-prometheus-stack-prometheus       1/1     5m1s
```

Promatranjem izlaza prethodne naredbe vide se svi objekti kreirani Helmovom automatiziranom instalacijom. Najvažniji za promatranje je prvi popis koji ispisuje statuse zrna. Pokrenuta zrna su servisi za nadzor, a to su redom Alertmanager, Grafana, Operator, Kube State Metrics, Node Exporter i Prometheus Server. Stvoreni DaemonSet je izvoznik performansnih metrika svakog Kubernetes čvora. Ovaj popis objekata nije apsolutan jer se varijabilni podaci poput imena zrna i IP adresa u grozdu mogu mijenjati dinamički.
Usluge se izlažu za potrebe provjere rada usluge servisa te za međusobnu komunikaciju ovisnosti. Usluge koje definiraju mrežna ponašanja za Prometheus i Grafanu će biti one čija će se vrata proslijediti zbog korištenja web sučelja. Iz sljedećih predložaka usluga može se zaključiti da su vrata na koja se pristupa Grafana sučelju 3000 TCP vrata, a vrata na kojima se poslužuje Prometheus 9090 TCP vrata. Iz izlaza sljedećih naredbi se može primjetiti da se pri automatskoj Helm instalaciji dodaju anotacije i oznake koje su važne za daljnje funkcioniranje te otkrivanje radom Operatora. Definiraju se parametri Port i TargetPort, odnosno vrata na koja servis sluša i vrata koja će biti izložena na zrnu.

```
$ kubectl describe service/prom-grafana

Name:              prom-grafana
Namespace:         default
Labels:            app.kubernetes.io/instance=prom
                   app.kubernetes.io/managed-by=Helm
                   app.kubernetes.io/name=grafana
                   app.kubernetes.io/version=9.3.0
                   helm.sh/chart=grafana-6.45.0
Annotations:       meta.helm.sh/release-name: prom
                   meta.helm.sh/release-namespace: default
```

```
Selector:           app.kubernetes.io/instance=prom,app.kubernetes.io/name=grafana
Type:               ClusterIP
IP Family Policy:   SingleStack
IP Families:        IPv4
IP:                 10.99.44.28
IPs:                10.99.44.28
Port:               http-web  80/TCP
TargetPort:         3000/TCP
Endpoints:          192.168.158.3:3000
Session Affinity:   None
Events:             <none>

$ kubectl describe service/prom-kube-prometheus-stack-prometheus

Name:               prom-kube-prometheus-stack-prometheus
Namespace:          default
Labels:             app=kube-prometheus-stack-prometheus
                    app.kubernetes.io/instance=prom
                    app.kubernetes.io/managed-by=Helm
                    app.kubernetes.io/part-of=kube-prometheus-stack
                    app.kubernetes.io/version=42.2.0
                    chart=kube-prometheus-stack-42.2.0
                    heritage=Helm
                    release=prom
                    self-monitor=true
Annotations:        meta.helm.sh/release-name: prom
                    meta.helm.sh/release-namespace: default
Selector:           app.kubernetes.io/name=prometheus,prometheus=prom-kube-prometheus-s
Type:               ClusterIP
IP Family Policy:   SingleStack
IP Families:        IPv4
IP:                 10.104.103.76
IPs:                10.104.103.76
Port:               http-web  9090/TCP
TargetPort:         9090/TCP
Endpoints:          192.168.158.5:9090
Session Affinity:   None
```

```
Events:            <none>
```

U kontekstu integracije nadzora unutar Kubernetes okružja, objekti koji se dodaju za definiciju nadzora su prilagođene resursne definicije ServiceMonitor. Opis objekta definicije objekta ServiceMonitor sadrži detaljnu razradu svih parametara koji su dopušteni za definiciju objekata. Primjerice, definicija resursa ServiceMonitor može se dohvatiti lokalno te pregledati. U preuzetoj YAML datoteci vrijedi pogledati opis obaveznih parametara, primjerice opis za krajnje točke.

```
$ kubectl get crd servicemonitors.monitoring.coreos.com -oyaml > ~/servicemonitor.yaml
(...)
```

Unutar datoteke se mogu potražiti opisi putanje i vrata koje je potrebno pratiti. U nastavku slijedi skraćena verzija sadržaja datoteke.

```yaml
properties:
  endpoints:
    description: A list of endpoints allowed as part of this ServiceMonitor.
    items:
      description: Endpoint defines a scrapeable endpoint serving Prometheus metrics.
      properties:
        path:
          description: |
            HTTP path to scrape for metrics. If empty, Prometheus
            uses the default value (e.g. `/metrics`).
          type: string
        port:
          description: |
            Name of the service port this endpoint refers to.
            Mutually exclusive with targetPort.
          type: string
```

Primjer jednog konačnog objekta za promatranje je ServiceMonitor objekt samoga Prometheus poslužitelja. U nastavku slijedi opis objekta `servicemonitor/prom-kube-prometheus-stack-prometheus`. U prethodnom opisu objekta ServiceMonitor istaknuti su parametri *path* i *port*, odnosno putanja i vrata na koja je servis dostupan, a u slučaju Prometheus poslužitelja promatrana je putanja `/metrics` i vrata koja su u objektu usluge imenovana kao `http-web`, odnosno vrata 9090 za TCP vezu.

```
$ kubectl describe servicemonitor prom-kube-prometheus-stack-prometheus

Name:         prom-kube-prometheus-stack-prometheus
Namespace:    default
Labels:       app=kube-prometheus-stack-prometheus
              app.kubernetes.io/instance=prom
              app.kubernetes.io/managed-by=Helm
              app.kubernetes.io/part-of=kube-prometheus-stack
              app.kubernetes.io/version=42.2.0
              chart=kube-prometheus-stack-42.2.0
              heritage=Helm
              release=prom
Annotations:  meta.helm.sh/release-name: prom
              meta.helm.sh/release-namespace: default
API Version:  monitoring.coreos.com/v1
Kind:         ServiceMonitor
Metadata:
  Creation Timestamp:  2022-12-05T09:56:21Z
  Generation:          1
  Managed Fields:
    API Version:  monitoring.coreos.com/v1
    Fields Type:  FieldsV1
    Manager:         helm
    Operation:       Update
    Time:            2022-12-05T09:56:21Z
  Resource Version:  3432
  UID:               22b46fc2-3547-4f7a-92a4-22e6c416df09
Spec:
  Endpoints:
    Path:  /metrics
    Port:  http-web
  Namespace Selector:
    Match Names:
      default
  Selector:
    Match Labels:
      App:             kube-prometheus-stack-prometheus
```

```
      Release:          prom
      Self - Monitor:   true
Events:                 <none>
```

6.2 Postavljanje vlastitog softvera u orkestrirano okružje

Zbog demonstracije načina na koji se softver po mjeri priključuje orkestriranim okružjima preuzet je primjer komunikacije porukama koristeći broker poruka RabbitMQ web framework FastAPI, alat za upravljanje poretka poslova Celery te web sučelje za upravljanje poretka poslova Flower prema članku autora Sumana Dasa [16]. Originalni primjer mikroservisne komunikacije pisan u programskom jeziku Python je za potrebe ovoga rada prilagođen radu u kontejneriziranom okružju te je ispred originalne usluge aplikacijske usluge za ispis podataka o sveučilištima dodan proxy koristeći Nginx.

Kao u mnogo slučajeva problema rješivih mikroservisnom softverskom arhitekturom, lagani web framework kao što je FastAPI obično nudi rješenje asinkrone paralelizacije za stavljanje poslova u pozadinu. Kada procesiranje mora obaviti teži, složeniji posao, predlaže se korištenje alata koji omogućuje rad u više procesa te je zbog tog pristupa u arhitekturu uveden alat Celery. Celery redove poslova obavlja distribuirano te zbog toga treba distribuirane radnike, a komunikacija u Celery grozdu se obavlja preko brokera za razmjenu poruka. U ovome slučaju je korišteni broker RabbitMQ, ali često korišteni brokeri su također Redis, ZeroMQ, Kafka, ActiveMQ, IBM-MQ i ostali.

Kada su svi servisi nastanjeni u orkestrirano okružje, rad servisa čine sljedeći koraci:

1. Klijent koristeći zamjenski web poslužitelj šalje zahtjev FastAPI aplikaciji.
2. FastAPI aplikacija šalje poruku u definirani red na brokeru sa zahtjevom.
3. Celery radnici prime poruku od brokera poruka, obave posao te ishod vrate brokeru u različiti red.
4. FastAPI promatra red brokera na kojemu se očekuje ishod zahtjeva.
5. Flower promatra rad Celery radnika tako da se pretplati na redove na brokeru.

Svaki od ovih servisa je potrebno kontejnerizirati te tri servisa ovisna o programskom jeziku Python imaju istu pripremu, ali konačno različite pozivne komande procesa koji će biti aktivan u kontejneru i različita izložena vrata kontejnera. Kontejneri aplikacija pokrenuti će se prema sljedećem primjeru Docker datoteke (engl. *Dockerfile*):

```
FROM python:3.9.6-slim
ENV PYTHONUNBUFFERED=1
WORKDIR /app
COPY . /app
```

```
RUN pip3 install --upgrade pip
RUN pip3 install -r requirements.txt
```

Servisi će se pokretati bazirani na minimalnoj slici programskog jezika Python verzije 3.9.6. Pri kontejnerizaciji Python aplikacija se obično definira varijabla okruženja koja omogućuje praćenje zapisa u realnom vremenu. U kontejner je potrebno kopirati programski kod, a potom instalirati knjižnice. Svaka od aplikacija izlaže potrebna vrata i pokreće se na specifičan način u posljednjim redovima Docker datoteke:

```
# Pokretanje FastAPI aplikacije
EXPOSE 9000
CMD python main.py

# Pokretanje Celery radnika
CMD celery -A main.celery worker --loglevel=info -E -Q \
    universities,university --concurrency=3

# Pokretanje Flower sučelja
EXPOSE 5555
CMD celery -A main.celery flower --port=5555
```

FastAPI u zadanim postavkama koristi Uvicorn sučelje pristupnika asinkronog poslužitelja koje sluša zahtjeve na vratima 9000. Radnik Celery grozda ne mora biti izložen vanjskoj mreži ukoliko je broker poruka u istom okruženju, a u slučaju ove integracije na Kubernetes jest. Radnik će na brokeru komunicirati koristeći dva reda: *universities* red koji radnicima poslužuje poslove koje treba obaviti te *university* red na koji radnici objavljuju dobivene rezultate. Flower izlaže vrata 5555 radi pristupa web sučelju, a prema Celery konfiguracijskim datotekama može prepoznati radnike čiji rad treba pratiti.
Sljedeći korak prema dostavljanju servisa na Kubernetes prenijeti kontejnerske slike aplikacija u registar slika koji je dostupan lokalnom Docker klijentu i unutar samog Kubernetes grozda. Ovaj korak je različit u svakom slučaju stvaranja softvera po mjeri te ovisi o infrastrukturi. Za produkcijske okoline se obično koriste slike iz pouzdanih registara u oblaku ili u vlastitim podatkovnim centrima te se implementiraju mehanizmi kontinuirane integracije i isporuke da bi proces bio što manje sklon ljudskoj pogreški. Na lokalnoj instanci Dockera potrebno je prvo napraviti prijavu s korisničkim računom ukoliko registar nije javan, a potom izgraditi slike servisa sljedećim naredbama. Slike drugih servisa su dostupne u javnim registrima kao što su Docker Hub i Quay.

```
$ sudo docker build -t referenca-registra/app:v1 -f Dockerfile_app .
(...)
```

```
$ sudo docker build -t referenca-registra/worker:v1 -f Dockerfile_worker .
(...)
$ sudo docker build -t referenca-registra/flower:v1 -f Dockerfile_flower .
(...)
```

Servisi Python aplikacija neće raditi ukoliko RabbitMQ nije podignut na lokalnoj Kubernetes instanci. Instalacija brokera poruka već je automatizirana pomoću dostupnih dijagrama za alat Helm, a vjerodostojan izvor kontejnerskih slika i Helm dijagrama je Bitnami. Instalacija brokera RabbitMQ bit će obavljena tako da će se među Helm repozitorije dodati Bitnami, preuzeti će se popis parametara za dijagram te onda obaviti instalacija i potvrditi kreacija puštanja. U slučaju ovoga rada, datoteka vrijednosti je sadržavala promjene zaporke i korisničkog imena za upotrijebu usluga brokera, promjenu u popisu dodatnih mogućnosti brokera te je omogućeno automatsko kreiranje objekta ServiceMonitor.

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami && helm repo update
(...)
$ helm show values bitnami/rabbitmq > rabbit-values.yaml
# u uređivaču teksta vlastitog izbora urediti datoteku s vrijednostima,
# a potom izvršiti instalaciju
$ helm install zec bitnami/rabbitmq -f rabbit-values.yaml
(...)
```

Zbog toga što su se u datoteci s vrijednostima promijenili zadano korisničko ime i lozinka, unutar istog imenskog prostora gdje će se integrirati servisi treba priključiti tajnu s pristupnim podacima. Kubernetes objekt za čuvanje tajni zahtjeva podatke base64 enkodirane podatke, stoga vrijednosti u objektu nisu ljudski čitljive. Ove vrijednosti bit će priključene aplikaciji, Celery radniku i Flower sučelju.

```
apiVersion: v1
kind: Secret
metadata:
  name: rabbit-creds
type: Opaque
data:
  CELERY_BROKER_URL: YW1xcDovL2tvcmlzbmlrOmxvemlua2FAemVjLXJhYmJpdG1xOjU2NzIvLw==
  RABBITMQ_DEFAULT_USER: a29yaXNuaWs=
  RABBITMQ_DEFAULT_PASS: bG96aW5rYQ==
```

FastAPI aplikaciji je priključen web poslužitelj Nginx kao zaštita ispred pristupa same aplikacije. Posrednik poslužitelj je obično priključen radi filtracije mrežnog prometa, sigurnosnih provjera zahtjeva i balansiranja mrežnog tereta. U ovom slučaju, radi se o jednostavnom primjeru

koji omogućuje da sama aplikacija nije izložena prometu Kubernetes grozda, već je izložena samo prometu u zrnu. Aplikacija i Nginx zajedno su integrirani u istom zrnu po uzorku dizajna *sidecar*, a to je uzorak po kojemu usko vezani servisi žive kao jedna cjelina te je najčešće samo jedan servis izložen. Da bi se omogućilo korištenje takvog uzorka, bilo je potrebno prilagoditi zadanu konfiguraciju servisa Nginx tako da na prisup vratima 8080 Nginx komunicira s definiranom krajnjom točkom aplikacije. Prilagođenu konfiguraciju potrebno je spremiti u obliku Kubernetes objekta konfigurabilne mape. Dodatno je u konfiguraciju dodano da Nginx na vratima 8000 omogućuje čitanje vlastitih metrika prilagođenima alatu Prometheus na krajnjoj točki /metrics, a za to je bilo potrebno iskoristiti metodu stub_status.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sidecar-nginx
data:
  nginx.conf: |
    error_log /var/log/nginx/error.log;

    events {
        worker_connections 1024;
    }

    http {
        log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                          '$status $body_bytes_sent "$http_referer" '
                          '"$http_user_agent" "$http_x_forwarded_for"';

        access_log  /var/log/nginx/access.log  main;

        server {
            listen       8080;

            location / {
                proxy_pass http://localhost:9000/;
            }
        }
        server {
            listen       8000;
```

```
            location /metrics {
                stub_status on;
            }
        }
    }
```

Zajednička usluga za zrno koje sadrži aplikaciju i Nginx sadrži definiciju vrata 8080 i 8000, odnosno vrata na kojima su dostupne aplikacija i metrike.

```yaml
kind: Service
apiVersion: v1
metadata:
  name: sidecar-uniapp
  labels:
    app.kubernetes.io/instance: sidecar-uniapp
    app.kubernetes.io/name: sidecar-uniapp
spec:
  ports:
    - protocol: TCP
      name: 8080-tcp
      port: 8080
      targetPort: 8080
    - protocol: TCP
      name: 8000-tcp
      port: 8000
      targetPort: 8000
  selector:
    app: sidecar-uniapp
  type: ClusterIP
status:
  loadBalancer: {}
```

Aplikacija i posrednik se zajednički pokreću u istom zrnu, a to znači da ih je potrebno zajedno spomenuti u Deployment objektu. Obrazac za stvaranje zrna spominje osnovne podatke o dva kontejnera koji će biti aktivni, a neki od tih podataka su kontejnerska slika, tajne i konfiguracijske datoteke, komande koje treba pokrenuti unutar kontejnera te izložena vrata. Ovo zrno demonstrira na koji način rade provjere živosti i provjere spremnosti, a to je tako da se osigura da šalju zahtjeve na krajnju točku na kojoj servis može odgovoriti. S obzirom da aplikacija

nije izložena, provjere živosti i spremnosti joj pristupaju preko posrednika. U kod aplikacije je dodan Starlette Instrumentor za FastAPI izvoz metrika te aplikacija poslužuje putanju /metrics.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sidecar-uniapp
  labels:
    app.kubernetes.io/instance: sidecar-uniapp
    app.kubernetes.io/name: sidecar-uniapp
spec:
  selector:
    matchLabels:
      app: sidecar-uniapp
  template:
    metadata:
      labels:
        app: sidecar-uniapp
    spec:
      volumes:
      - name: sidecar-nginx
        configMap:
          name: sidecar-nginx
          defaultMode: 420
      containers:
      - name: uniapp
        image: referenca-registra/app:v1
        env:
        - name: CELERY_BROKER_URL
          valueFrom:
            secretKeyRef:
              name: rabbit-creds
              key: CELERY_BROKER_URL
        - name: CELERY_RESULT_BACKEND
          value: rpc://
        <...>
      - name: nginx
        command:
```

```
          - nginx
        args:
          - '-g'
          - daemon off;
        volumeMounts:
          - name: sidecar-nginx
            mountPath: /etc/nginx/nginx.conf
            subPath: nginx.conf
        image: nginx:1.20
        ports:
        - containerPort: 8080
          protocol: TCP
        - containerPort: 8000
          protocol: TCP
        livenessProbe:
          httpGet:
            path: /metrics
            port: 8080
            scheme: HTTP
          <...>
        readinessProbe:
          httpGet:
            path: /metrics
            port: 8080
            scheme: HTTP
          <...>
        <...>
```

Celery radnik i Flower sučelje se pokreću u dva izolirana zrna, a definiraju vlastite Deployment i uslužne objekte poput onih za aplikaciju. Za aplikaciju i Flower je bilo potrebno kreirati ServiceMonitor objekte koji opisuju krajnje točke i sadrže oznake prema kojima će ih Operator prepoznati. S obzirom da je ime Prometheusovog puštanja prom, obavezna oznaka za dodati je `release: prom`. Unutar svih usluga, Deployment objekata i ServiceMonitor objekata servisa koji će biti promatrani je potrebno dodati oznaku imena instance i servisa.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
```

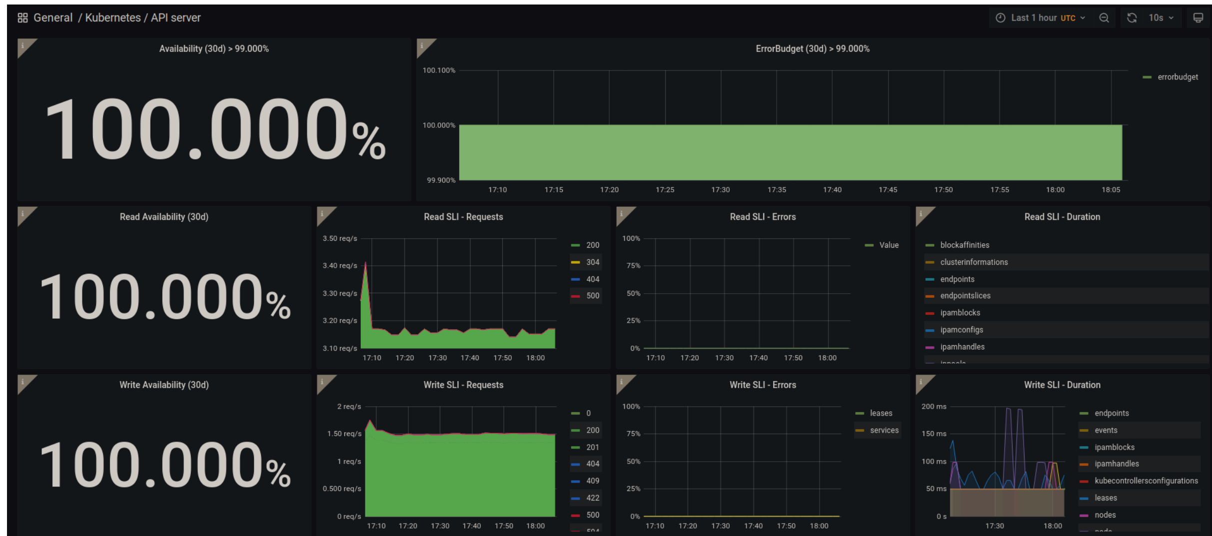
```
  labels:
    app.kubernetes.io/instance: flower
    app.kubernetes.io/name: flower
    release: prom
  name: flower
  namespace: default
spec:
  endpoints:
  - honorLabels: true
    interval: 30s
    path: /metrics
    port: 5555-tcp
    scheme: http
    scrapeTimeout: 30s
  jobLabel: prom
  namespaceSelector:
    matchNames:
    - default
  selector:
    matchLabels:
      app.kubernetes.io/instance: flower
      app.kubernetes.io/name: flower
```

6.3 Prikazi metrika u Grafana sučelju

Kada se Prometheus postavlja korištenjem alata Helm, praćenje metrika vezanih uz rad Kubernetes grozda omogućeno je u zadanim postavkama te se zajedno sa servisima vezanim uz nadzor dodaju izvoznici podataka crne kutije za svaki čvor te izvoz internih metrika rada sustava. Ukoliko dohvatimo pokrenuta zrna, moguće je uočiti četiri zrna ključna za nadzor Kubernetesa. S obzirom da ovaj grozd čini tri čvora, tri su različita servisa za izvoz podataka čvorova.

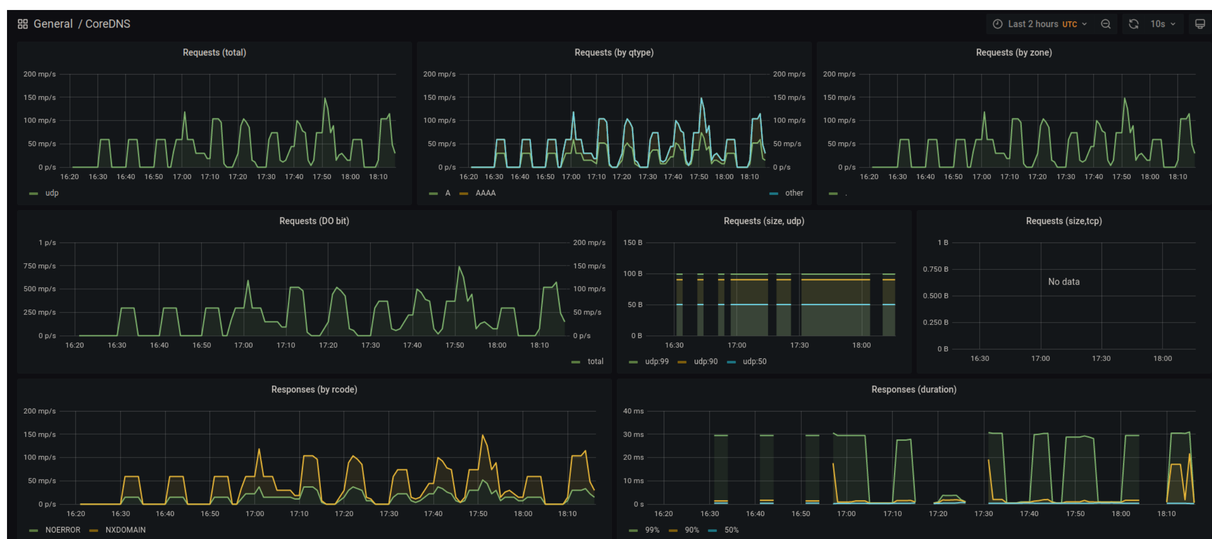
```
$ kubectl get pods
(...)
prom-kube-state-metrics-54c4ff848b-5sk9k         1/1     Running      (
prom-prometheus-node-exporter-8kxgn              1/1     Running      (
prom-prometheus-node-exporter-dgqth              1/1     Running      (
prom-prometheus-node-exporter-v5c2j              1/1     Running      (
(...)
```

Odmah po instalaciji u Grafana sučelju nude se brojne ploče metrika za promatranje Kubernetesa. Općenito, najčešće se promatraju hardverski resursi, mrežni teret, ponašanje zrna u odnosu na njihova ograničenja te specifičnosti vezane uz ključne dijelove Kubernetesa poput Kubelet i API Server procesa. Od mnogo izbora je važno pratiti prave metrike koje su važne. Na više ploča se ponavljaju isti podaci te je moguće da različiti izvoznici prate jednake podatke.
API Server ploča sadrži temeljne podatke o dostupnosti grozda te je na njoj moguće vidjeti trendove pisanja i čitanja memorije, trajanja odgovora na zahtjeve i postotak dostupnosti.



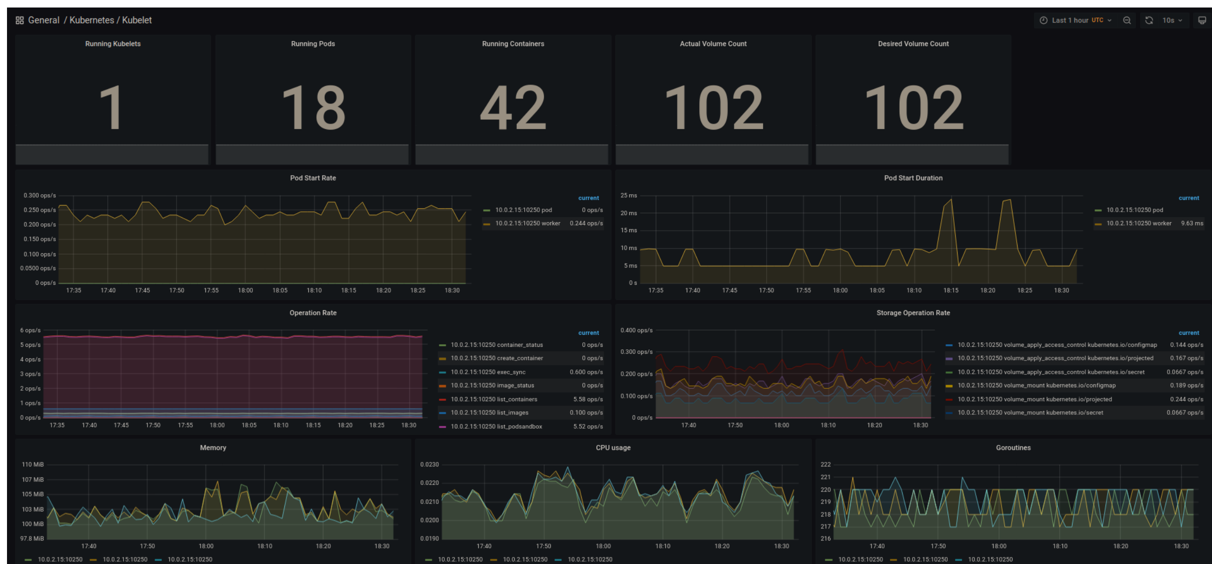
Slika 6.1: API Server ploča i mjere dostupnosti

CoreDNS je DNS poslužitelj koji je osnova Kubernetesove mrežne komunikacije te poslužuje idealne metrike za praćenje mrežne stabilnosti unutar grozda. Detekcija grešaka u mreži se može ustanoviti prema trajanju zahtjeva i odgovora, a potencijalni napadi se mogu primjetiti ukoliko primljeni zahtjevi imaju neuobičajeno velika tijela.



Slika 6.2: CoreDNS ploča i mrežne metrike

Kubelet ploča prikazuje osnovne podatke o čvoru na kojemu se nalazi Kubelet proces, broju zrna kojim upravlja, brzini podizanja zrna te memorijskih operacija koje se događaju zbog upravljanja objektima.



Slika 6.3: Kubelet ploča i metrike upravljanja zrnima

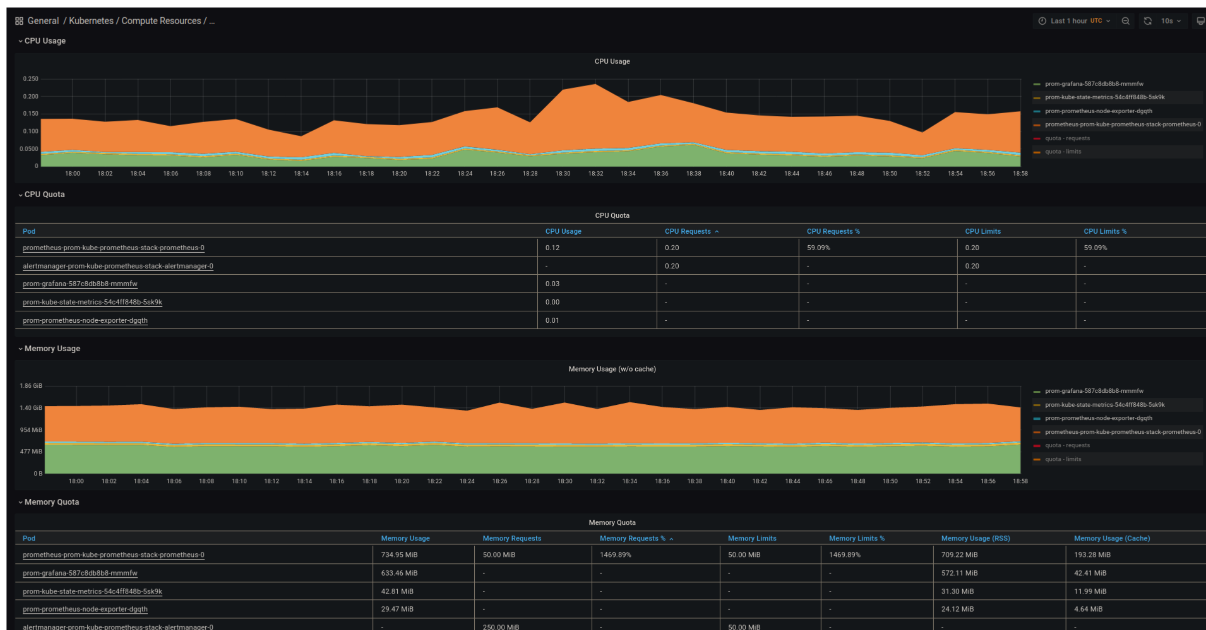
Izvoznici čvorova omogućuju informacije vezane uz resurse korištene na pojedinom čvoru, a dostupne ploče razlikuju se na temelju izvora zrna. Primjerice, potrošnja resursa može se provjeriti na temelju potrošnje u imenskom prostoru, pojedinom čvoru grozda te na razini cijelog grozda. Podaci su pogodni za izražavanje grafikonima i informativnim tablicama, a dostupni su podaci o procesorskim resursima, memoriji te mreži. Potrošnju resursa nije nužno filtrirati prema Kubernetes konceptima, već je dostupna ploča koja prikazuje općenite podatke potrošnje na čvoru.

Osim što Prometheus prikuplja metrike za sve usluge koje imaju ServiceMonitor, tako prikuplja i vlastite metrike vezane uz podatke koje prikuplja. Iz prikazanih grafova na prilogu {#fig:prometheus-dashboard} mogu se prepoznati praćeni servisi, koliko vremena Prometheus provede u kojoj fazi obrade podataka te koliko upita je obradio.

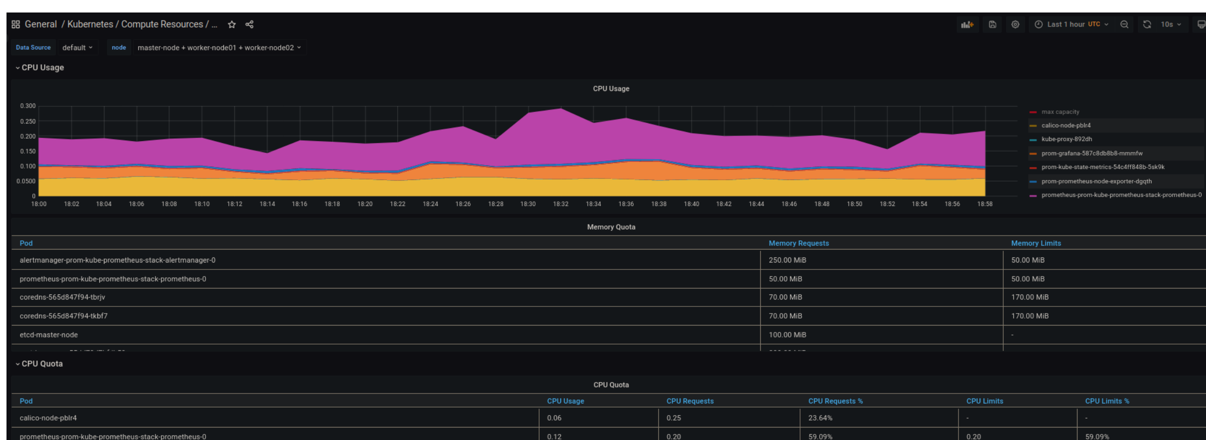
RabbitMQ je često korišteni alat za razmjenu poruka te se njegova ploča redovno razvija od strane zajednice. Iako je ova kontrolna ploča raznovrsnija što je RabbitMQ grozd veći, nudi informacije o broju izmjenjenih poruka, potrošenim i dostupnim resursima, broju pretplatnika i redova, aktivnih konekcija te neobrađenih poruka.

Poput brokera poruka, Celery zajednica također održava vlastitu ploču koja integrira podatke o čvorovima radnicima, njihovom radnom teretu i prosječnom vremenu obavljanja posla. U slučaju prikazanom prilogom {#fig:celery-dashboard}, Prometheus prikuplja podatke s tri radnika koji obrađuju zahtjeve.

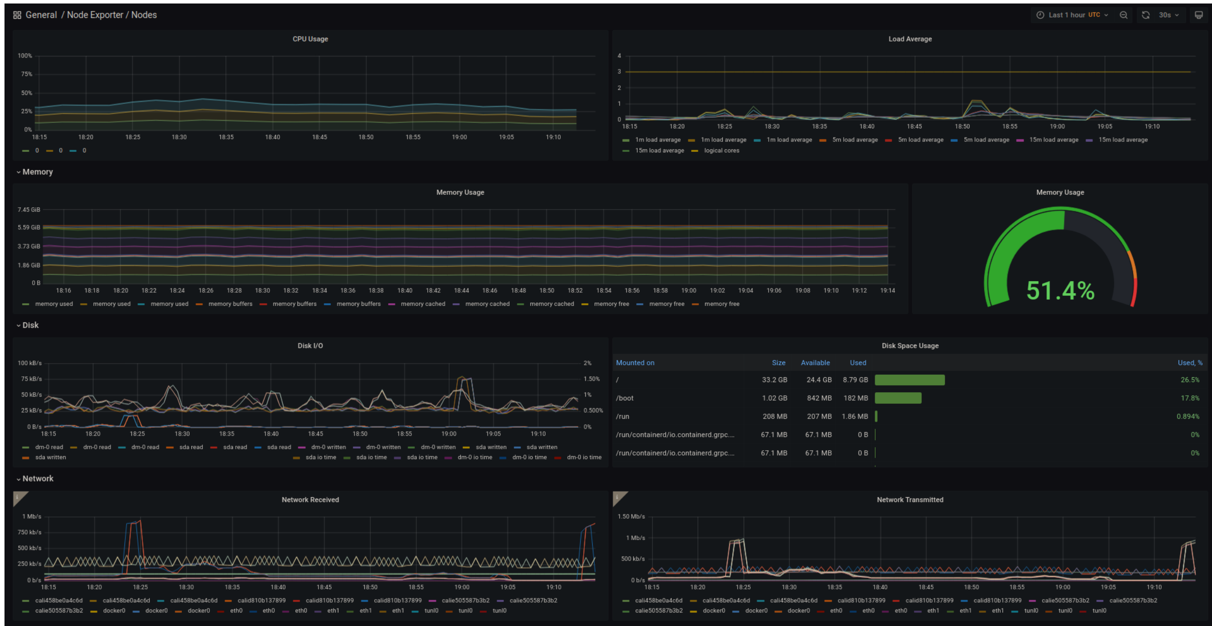
Kod izgradnje softvera po mjeri, često se događa da ne postoji gotovo rješenje za metrike koje je potrebno nadzirati. Za tu svrhu Prometheus i Grafana nude sučelja za testiranje upita jezika



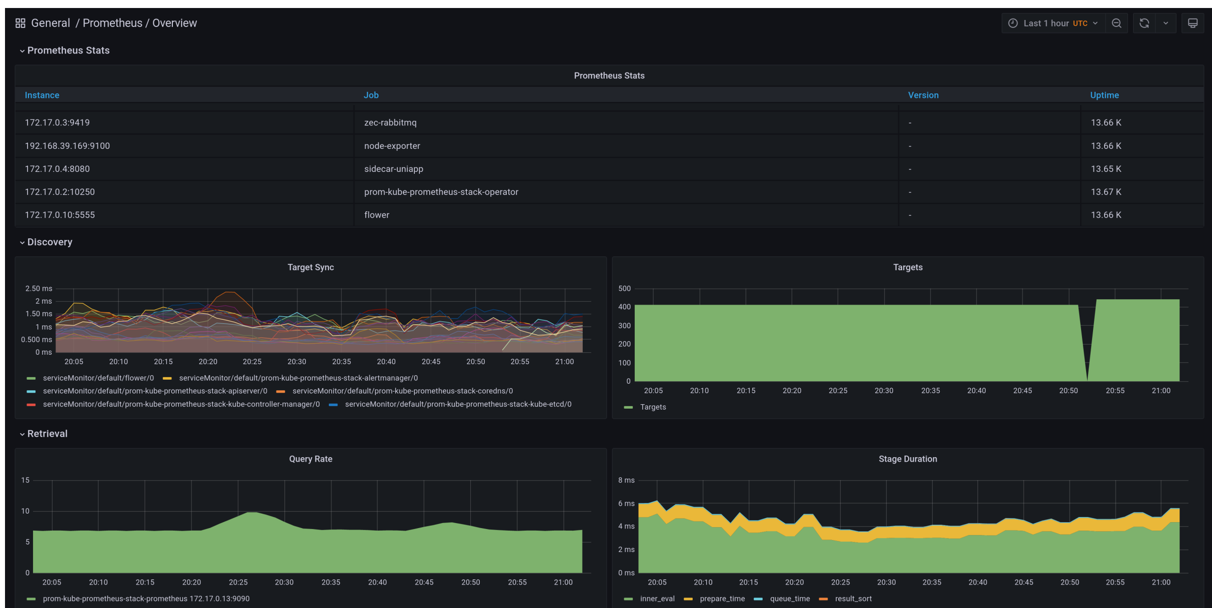
Slika 6.4: Računalni resursi prema imenskom prostoru



Slika 6.5: Računalni resursi prema čvorovima grozda

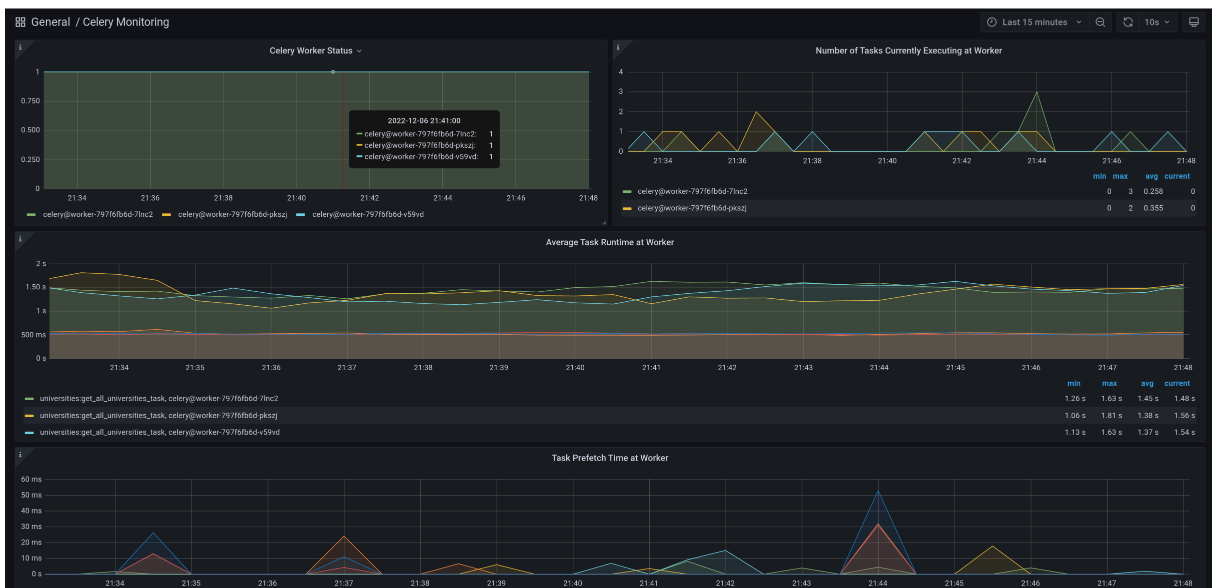


Slika 6.6: Općenita potrošnja resursa na čvoru



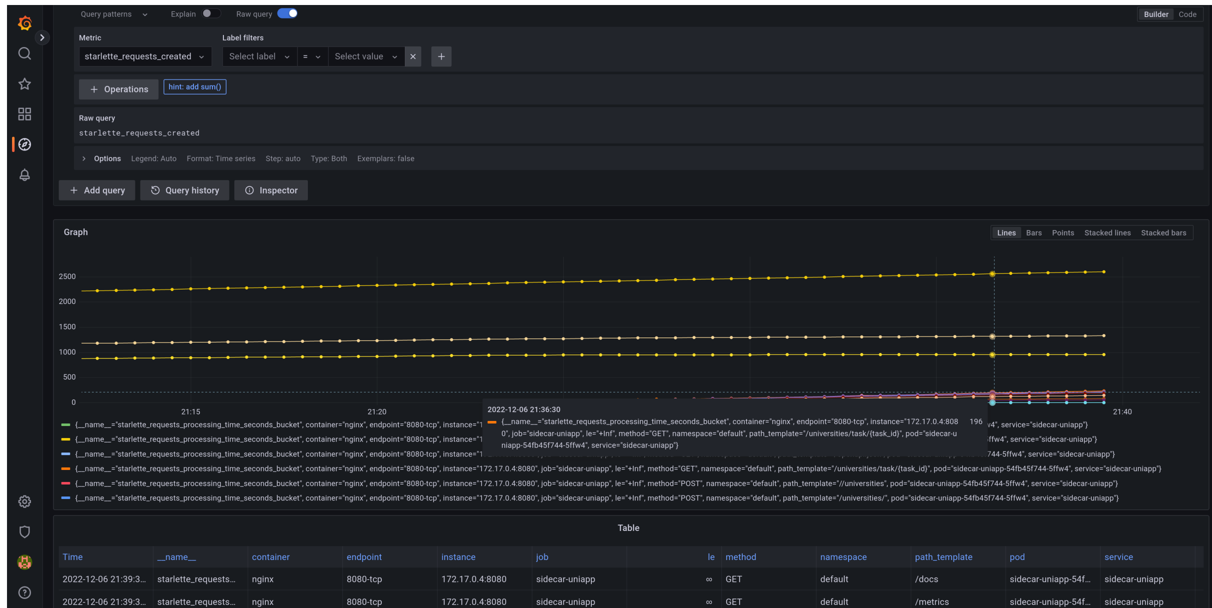
Slika 6.7: Prometheus ploča

Slika 6.8: RabbitMQ ploča



Slika 6.9: Celery ploča

PromQL. Primjerice, u slučaju FastAPI aplikacije, bio je uključen Starlette izvoznik koji je omogućio da Prometheus prikupi podatke, a korisnik ih uredi u vlastite grafikone i tablice. U nastavku slijedi primjer generiranja grafikona i tablice zahtjeva primljenih unutar FastAPI aplikacije.



Slika 6.10: Starlette metrika izražena upitnim jezikom PromQL

Da bi promatranje metrika bilo zaista korisno u realnom vremenu, potrebno je postaviti pravila za obavještavanje. Obavještavanje se koristi na način da se definira set pravila prema kojima se šalju PromQL upiti unutar vremenskih obrazaca. U slučaju održavanja Prometheus okoline na Kubernetesu, pravila se definiraju prilagođenim resursnim definicijama PrometheusRule. Takvi objekti se sortiraju prema servisima i tematici metrika, a sadrže formule koje je potrebno ispitati. Primjerice, u nastavku je prikazana definicija PrometheusRule objekta za API Server te jedno od definiranih pravila u objektu. Sljedeće pravilo ima izraz koji po grozdu zbraja sve HTTP zahtjeve te pohranjuje trendove za prethodnih 30 dana.

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  annotations:
    meta.helm.sh/release-name: prom
    meta.helm.sh/release-namespace: default
  labels:
    app: kube-prometheus-stack
    heritage: Helm
    release: prom
```

```
  managedFields:
    - apiVersion: monitoring.coreos.com/v1
      manager: helm
  name: prom-kube-prometheus-stack-kube-apiserver-availability.rules
  namespace: default
spec:
  groups:
    - interval: 3m
      name: kube-apiserver-availability.rules
      rules:
        - expr: >-
            sum by (cluster, code)
            (code_verb:apiserver_request_total:increase30d
            {verb=~"POST|PUT|PATCH|DELETE"})
          labels:
            verb: write
          record: code:apiserver_request_total:increase30d
```

Izraze kojima se definiraju pravila te njihove izlaze moguće je vidjeti u Prometheus sučelju pod karticom Rules. Sljedeći prilog {#fig:prometheus-rule} prikazuje primjer pravila zadanog kroz Kubernetes objekt PrometheusRule u sučelju.

```
record: code:apiserver_request_total:increase30d
expr: sum by (cluster, code) (code_verb:apiserver_request_total:increase30d{verb=~"POST|PUT|PATCH|DELETE"})
labels:
  verb: write
```

OK		11.114s ago	0.544ms

Slika 6.11: Definicija pravila u Prometheus sučelju

U istoj definiciji pravila je potrebno uključiti obavještavanje. Takva informacija se pojavljuje u Prometheus sučelju pod karticom Alerts. U svrhu demonstracije je nekoliko ključnih komponenti bilo isključeno, a Prometheus je prijavio nedostupnosti.

TargetDown (4 active)

```
name: TargetDown
expr: 100 * (count by (job, namespace, service) (up == 0) / count by (job, namespace, service) (up)) > 10
for: 10m
labels:
  severity: warning
annotations:
  description: {{ printf "%.4g" $value }}% of the {{ $labels.job }}/{{ $labels.service }} targets in {{ $labels.namespace }} namespace are down.
  runbook_url: https://runbooks.prometheus-operator.dev/runbooks/general/targetdown
  summary: One or more targets are unreachable.
```

Labels	State	Active Since	Value
alertname=TargetDown job=sidecar-uniapp namespace=default service=sidecar-uniapp severity=warning	FIRING	2022-12-07T13:13:50.254501731Z	50
alertname=TargetDown job=kube-controller-manager namespace=kube-system service=prom-kube-prometheus-stack-kube-controller-manager severity=warning	FIRING	2022-12-07T13:13:50.254501731Z	100
alertname=TargetDown job=kube-scheduler namespace=kube-system service=prom-kube-prometheus-stack-kube-scheduler severity=warning	FIRING	2022-12-07T13:13:50.254501731Z	100
alertname=TargetDown job=kube-etcd namespace=kube-system service=prom-kube-prometheus-stack-kube-etcd severity=warning	FIRING	2022-12-07T13:13:50.254501731Z	100

Slika 6.12: Prijava nedostupnosti servisa u Prometheus sučelju

Nije očekivano da administratori kontinuirano prate Prometheus sučelja da bi uočili pogreške, već je potrebno automatizirati obavijesti. Za tu svrhu je u Prometheus usluge uključen AlertManager te radi na način da mu Prometheus šalje obavijesti u JSON obliku ukoliko uoči događaj koji je potrebno podijeliti.

Poglavlje 7

Zaključak

U ovom radu opisane su prednosti i razlike monolitne i mikroservisne arhitekture softvera te kako ih integrirati u orkestrirane okoline s osiguranom redundancijom i nadzorom. Iako se ovaj rad odnosio na nadzor mikroservisnog softvera, predstavljene prakse primjenjive su na klasičan softver u uobičajenim, nekontejneriziranim okolinama. Općenito je preporučeno što je više moguće uključiti prakse koje omogućavaju da u kritičnim aktivnostima poput dostave softvera nema ljudskog elementa.
Osmišljanje mikroservisnog softvera može biti skupi proces gdje osim samog rada sa softverom treba ispravno educirati razvojni tim te u fazi arhitekture unaprijed definirati alate i prakse koje će pratiti kontinuirani razvoj, isporuku i integraciju. Veliki gubitak resursa i vremena uzima edukacija, a šira je od same primjene metodologije. Primjerice, jedan alat kao što je Prometheus zahtjeva učelje upitnog jezika za dublje razumijevanje problematike. Slična problematika ponavlja se s alatima za kontinuiranu isporuku i integraciju, premda takvi alati često nude dva načina definicije procesa: konfiguracijama ili skriptama.
Cijela skupina novih alata, automatizacije, migracija na usluge u oblak i edukacija omogućuje visoku dostupnost te pravovremene i prilagođene reakcije na potencijalne probleme. Da bi softver ostao kvalitetan, potrebno je uložiti u kvalitetna rješenja i ne stati na tom koraku, već kontinuirano pratiti trendove te usavršavati postojeće postavke.

Literatura

[1] S. Newman, *Building microservices: Designing fine-grained systems*, Second edition. Beijing Boston Farnham Sebastopol Tokyo, 2015.

[2] D. L. Parnas, "Information distribution aspects of design methodology," 1971.

[3] K. Matthias and S. P. Kane, *Docker: Up and running: Shipping reliable containers in production*, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2015.

[4] faizanbashir arvindpdmn, "Containerization," *Devopedia*. Mar-2017.

[5] 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.

[6] B. Burns, J. Beda, and K. Hightower, *Kubernetes up & running: Dive into the future of infrastructure*, Second edition. Beijing [China] ; Boston [MA]: O'Reilly Media, 2019.

[7] "Installing kubeadm," *Kubernetes*.

[8] B. Bhikkaji, "Kubernetes Tip: Why disable swap on linux ?" *Tailwinds-MajorDomo*. Jun-2020.

[9] "Namespaces," *Kubernetes*.

[10] M. Hausenblas and S. Schimanski, *Programming Kubernetes: Developing cloud-native applications*, First edition. Sebastopol, CA: O'Reilly Media, 2019.

[11] L. Atchison, *Architecting for scale: High availability for your growing applications*, First edition. Sebastopol, California: O'Reilly, 2016.

[12] "High availability," *Wikipedia*. Dec-2022.

[13] J. Bastos and P. Araújo, *Hands-On Infrastructure Monitoring with Prometheus*, 1st edition. Packt Publishing, 2019.

[14] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., *Site reliability engineering: How Google runs production systems*, First edition. Beijing ; Boston: Oreilly, 2016.

[15] "Operating etcd clusters for Kubernetes," *Kubernetes*.

[16] S. Das, "Async Architecture with FastAPI, Celery, and RabbitMQ," *Crux Intelligence*. May-2022.