UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Milana Kovačević

RAZVOJ PLATFORME ZA DISTRIBUIRANO IZRAČUNAVANJE U OBLAKU

master rad

Beograd, 2022.

Mentor:

prof. dr Saša Malkov
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip Marić
Univerzitet u Beogradu, Matematički fakultet

doc. dr Ivan Čukić
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Porodici i najbližima
za strpljenje i podršku tokom studiranja

Naslov master rada: Razvoj platforme za distribuirano izračunavanje u oblaku

Rezime: Distribuirani sistemi predstavljaju skup nezavisnih mašina, koje su međusobno povezane mrežom. Mašine međusobno sarađuju i raspoređuju poslove kako bi rešile zadati problem. Distribuirani sistemi za izračunavanje su specijalizovani za izvršavanje resursno zahtevnih zadataka koristeći visok nivo paralelizma koji se postiže raspodelom poslova na čvorove. Ovim se dostižu visoke performanse i mogućnost izvršavanja poslova koji često ne bi mogli da se izvrše na jednoj mašini. Glavni čvor koordiniše izvršavanje, a ostali izvršavaju zadatke. Ovaj rad opisuje sistem za distribuirano izvršavanje poslova, nazvan DCS, implementiran u programskom jeziku C#. Sistem DCS predstavlja Sofver kao servis rešenje za obradu poslova. Prednost sistema je njegova modularnost, jer ga je moguće proširiti i specijalizovati da izračunava različite poslove od interesa. U radu je fokus na infrastrukturi sistema. Predstavljena je njegova arhitektura kao i način organizovanja i izvršavanja poslova. Dodatno, predstavljen je i način njegovog pokretanja u okviru klastera Kubernetes, korišćenjem *Docker* kontejnera. U okviru rada je predstavljano i pokretanje sistema DCS u oblaku, koristeći platformu Azure i servis Azure Kubernetes Service. DCS je integrisan sa ostalom infrastrukturom u oblaku, koristeći pomoćne servise za praćenje rada sistema i detekciju grešaka.

Ključne reči: distribuirani sistemi, distribuirano izračunavanje, Azure tehnologije, oblak, Softver kao servis, Kubernetes, Docker

Sadržaj

Glava 1

Uvod

U uvodnom delu su ukratko dati pregled i opšte informacije o distribuiranim sistemima, kako bi se stekla šira slika o oblasti ovog rada. Opisani su osnovni pojmovi i dati primeri relevantnih postojećih sistema za distribuirano izračunavanje. U nastavku su opisane korišćene tehnologije i alati potrebni za razumevanje rada. Detalji realizacije praktičnog dela su opisani u poglavlju 2, gde su dati pregled funkcionalnosti, detalji implementacije, kao i način pokretanja sistema u oblaku i njegovog testiranja i praćenja. U poglavlju 3, opisani su rezultati implementiranog rešenja kroz primer i analizu performansi. Na kraju, predložena su moguća unapređenja sistema i pravci njegovog daljeg razvoja, kao i zaključak celokupnog rada.

1.1 Distribuirani sistemi

Distribuirani sistemi se sastoje od skupa fizički odvojenih mašina, tzv. čvorova, koje su međusobno povezane mrežom. Na ovim mašinama su pokrenute softverske jedinice koje međusobno dele odgovornost, poslove, komuniciraju i sinhronizuju se, kako bi rešile zadati problem.

Prednosti ovako struktuiranog softvera su značajne. Neke od njih su:

1. Postizanje visokog nivoa paralelizacije prilikom raspodele poslova na mašine,
2. Pouzdanost postignuta prevazilaženjem problema jednog mesta otkazivanja sistema (eng. *single point of failure*). Ovim se takođe povećava dostupnost softvera i njegova otpornost na greške,
3. Skalabilnost postignuta promenom broja mašina uključenih u sistem, čime se sistem može prilagoditi potrebama.

4. Geo-distribuiranje povezivanjem mašina putem interneta.

Glavna mana distribuiranih sistema je oslanjanje na transport poruka kroz mrežu. Prezasićenjem mreže može doći do povećanog kašnjenja prilikom transporta informacije od jednog dela sistema do drugog. Dodatna mana je što je značajno teže obezbediti sistem nego što je to slučaj sa sistemina sa jednom mašinom. Na kraju, implementacija i održavanje distribuiranih sistema su načelno kompleksniji nego rad sa atomičnim sistemima.

Distribuirani sistemi se mogu razvrstati u zavisnosti od načina povezivanja mašina. Neki od najčešće zastupljenih oblika arhitekture su:

1. Klijent-server - Arhitektura u kojoj postoji jasna podela poslova između servera, koji je snabdevač podataka ili servisa, i klijenta, koji šalje zahteve serveru.

2. Peer-to-peer - Arhitektura u kojoj su sve mašine u sistemu su ravnopravne. Ne postoji centralna jedinica, već mašine međusobno komuniciraju i dele poslove i podatke. Svaka mašina pokreće isti softver. Primer sistema koji koristi peer-to-peer arhitekturu je *Torrent*.

3. *Middleware* - Arhitektura koja uključuje komponentu koja služi za povezivanje aplikacija. Jedan od primera je *middleware* zasnovan na porukama, čija je svrha transportovanje poruka između dve ili više aplikacija. Pojednostavljuje se integracija servisa u kompleksnim distribuiranim sistemima.

4. Višeslojne arhitekture - Arhitektura u kojoj su komponente podeljene u jedinice, u slojevima. Zahtev dolazi do prve jedinice gde se obrađuje i posleđuje sledećoj jedinici, prateći slojeve sistema. Rezultat se šalje obrnutim redosledom.

Distribuirani sistemi se mogu podeliti prema nameni na sledeći način:

1. Distribuirani sistemi za izračunavanje - Sistemi koji se koriste za dostizanje željenih performansi, oslanjajući se na visok stepen paralelizacije. Mogu se podeliti u dve grupe:

   a) Klasteri predstavljaju grupu uvezanih čvorova koji međusobno sarađuju, deleći lokalnu mrežu i koristeći isti operativni sistem. Glavni čvor (eng. *master*) je zadužen za primanje zahteva, prosleđivanje zadataka na čvorove „robove" (eng. *slaves*) i slanje rezultata nazad do korisnika.

   b) Sistemi za mrežno izračunavanje se sastoje od uvezanih podgrupa od kojih je svaka nezavisni distribuirani sistem. Kontrolni čvor predstavlja vezu između podgrupa sistema.

2. Distribuirani informacioni sistemi - „Uređeni sistemi koji prikupljaju, skladište, obrađuju i isporučuju informacije o stanju domena" [21], s tim da imaju distribuiranu arhitekturu.

   a) Sistemi za obradu distribuiranih transakcija obezbeđuju glavne osobine transakcija, ali u distribuiranom okruženju. Glavne karakteristike transakcija su atomičnost (nevidljiva za ostale, dok se ne završi), konzistentnost, izolacija (ne utiču na druge transakcije) i trajnost (završena transakcija je nepromenljiva). Glavni deo sistema koji upravlja transakcijama se naziva *TP Monitor* (eng. *Transaction Processing Monitor*) ili jednostavnije, menadžer transakcija (eng. *Transaction Manager*).

   b) Integracija aplikacija *Enterprise* povezuje različite poslovne softvere u jedno rešenje.

3. Prožimajući distribuirani sistemi uvode svakodnevne objekte u sistem koji ih povezuje. To uključuje naprave za praćenje zdravstvenog stanja poput pametnog telefona, sata, sisteme pametnih kuća, kao i drugih sistema koji prikupljaju podatke putem senzora.

Delom distribuiranih sistema se može smatrati i *RPC* (eng. *Remote Procedure Call*). *RPC* predstavlja deo softvera koji apstrahuje način izvršavanja neke komande. Komanda može biti implementirana na proizvoljan način, a često uključuje komunikaciju sa drugim delovima sistema, čime predstavlja vezu između komponenti. Softver koji koristi *RPC* nema potrebe da toga bude svestan, već se fokusira na druge stvari.

Više detalja o distribuiranim sistemima se može naći u [14] i [15].

1.2 Sistemi za distribuirano izračunavanje

Sistemi za dristribuirano izračunavanje imaju cilj da efikasno obrade zahtevne poslove. Za dostizanje željenih performansi, sistem se oslanja na visok nivo paralelizacije koju postiže raspodelom poslova na čvorove. Inicijalna pretpostavka sistema je da su poslovi resursno zahtevni, tj. da je potrebna veća količina resursa (memorija,

procesor) za njihovo izračunavanje. Ovo često znači da posao ne može da se izvrši na jednoj mašini, ili, ukoliko je to moguće, izršavanje ne zadovoljava očekivane performanse. Distribuirani sistemi nose sa sobom cenu sinhronizacije poslova između čvorova, kao i slanje podataka kroz mrežu. Međutim, oslanjajući se na pretpostavku o zahtevnosti poslova, ova cena je prihvatljiva, jer je nadoknađena ubrzanjem koje se postiže paralelizacijom.

Količina podataka u svetu kao i broj zahteva za njegovo obrađivanje neprekidno rastu. Obrada velikih podataka je postala potreba svakodnevice. Sistemi za distribuirano izračunavanje predstavljaju rešenje za obradu velike količine podataka i potražnja za njima raste u koraku sa globalnom količinom podataka.

Primeri postojećih sistema

Danas postoji više sistema za obradu podataka koji koriste distribuiranu arhitekturu. Svako od navedenim rešenja ima prednosti i mane, a u zavisnosti od potreba korisnika, neki sistem je bolji ili lošiji izbor.

Neki od sistema za distribuirano izračunavanje su navedeni u nastavku:

1. Apache Spark [24] - Spark je sistem otvorenog koda za obradu podataka, i deo je ekosistema Hadoop koji pruža korisnicima skup servisa za obradu velike količine podataka.

2. Databricks [9] predstavlja dodatnu nadogradnju sistema Spark.

3. Snowflake [23] je Softver kao servis rešenje za obradu podataka, koji u poslednje vreme stiče sve veću popularnost.

4. Azure Functions [6] je *serverless* rešenje za obradu podataka. *Serverless* znači da korisnik nema uvid u resurse koje koristi Azure Functions, već ga koristi po potrebi, na zahtev korisnika. Korisnik definiše logiku u blokovima koda koji se nazivaju funkcije, koje se po potrebi pokreću nad prosleđenim ulaznim podacima. Sistem automatski skalira resurse potrebne za izvršavanje funkcija. Analogno rešenje na platformi *AWS* je *AWS Lambda*.

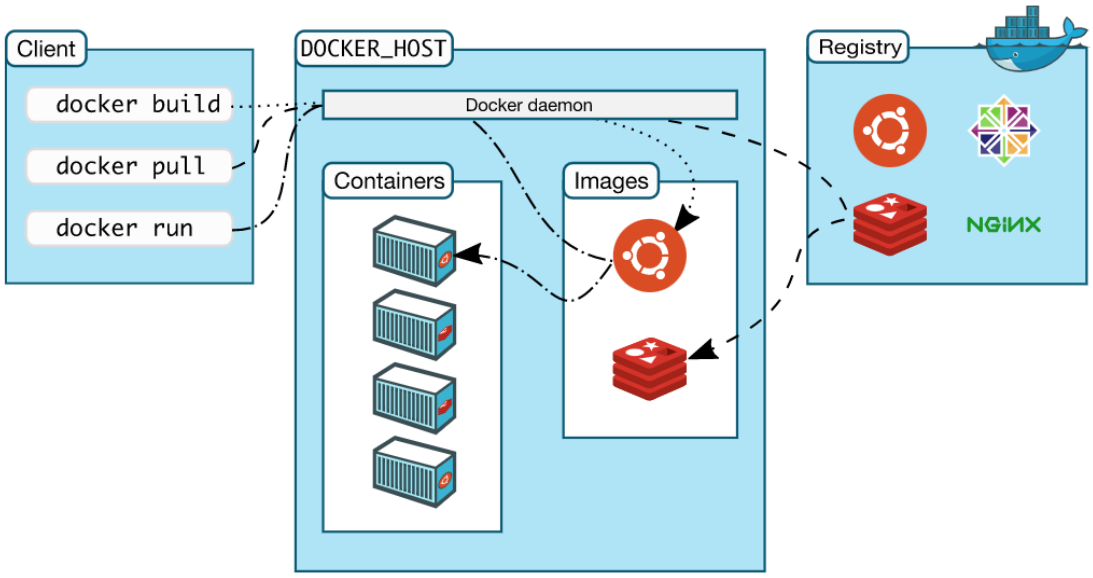
1.3 Opis korišćenih tehnologija i alata

U ovom odeljku su opisane tehnologije i alati korišćeni prilikom programske realizacije sistema.

Platforma *Docker*

Za pokretanje aplikacija je korišćena platforma *Docker* [11]. Prednosti korišćenja ove platforme su mnogostuke. Za početak, ona razdvaja razvijanje aplikacije od infrastrukture na kojoj će biti pokrenuta. Aplikacija je spakovana u izolovano okruženje koje se naziva kontejner (eng. *container*). Kontejneri sadrže sve ono što je neophodno za pokretanje aplikacije, a u idealnom slučaju samo ono što je neophodno, u vidu strukture koja se naziva slika (eng. *image*). To čini da su kontejneri lagani za prenošenje, za razliku od virtuelnih mašina, koje mogu da pruže istu funkcionalnost. Spakovana aplikacija može biti pokrenuta neograničeni broj puta, u različitim okruženjima: prilikom ručnog i automatskog testiranja, u produkciji, itd. Prilikom pokretanja kontejnera, može se dodatno precizirati njegova konfiguracija (na primer, mapiranje portova).

Arhitektura platforme je predstavljena na slici 1.1, preuzetoj iz zvanične dokumentacije [12].



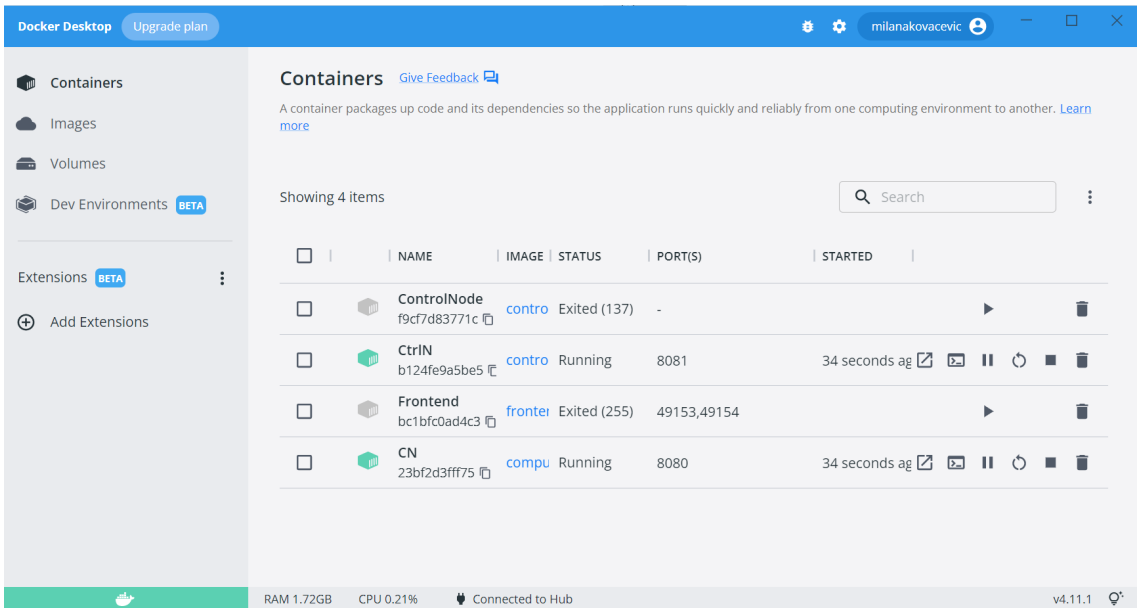
Slika 1.1: Arhitektura platforme *Docker*

Koraci potrebni da se napravi slika aplikacije se definišu u datoteci *Dockerfile*. Svaka instrukcija u ovom fajlu kreira po jedan sloj slike (eng. *layer*). U slučaju *.NET Core* aplikacija, prvi korak je učitavanje željenog radnog okvira, zatim, (primera radi) pokretanje prevođenja izvornog koda i smeštanje izvršnog koda na željenu lokaciju. Krajnji korak je uglavnom definisanje komande za pokretanje aplikacije. Bitno je naglasiti da tokom razvijanja aplikacije, *Docker* prepoznaje koji su se slo-

jevi slike promenili, te ponovo kreira samo njih i njihove naredne slojeve. Ovo čini generisanje slike efikasnom i brzom operacijom koja ne usporava programera, koji najčeće menja samo neke delove aplikacije. Kreirane slike se čuvaju u registru slika koji može biti lokalni ili negde u oblaku.

Kontejner sadrži specifikaciju operativnog sistema na kojem se pokreće slika, i on može biti *Windows*, *Mac* i *Linux*. Kontejneri koji odgovaraju drugom operativnom sistemu se pokreću u odgovarajućoj virtuelnoj mašini.

Korišćeno okruženje za lokalno kreiranje i pokretanje kontejnera je *Docker Desktop* [13] prikazan na slici 1.2. *Docker Desktop* u sebi sadrži virtuelnu mašinu *Linux* u okviru koje se pokreću kontejneri *Linux*.



Slika 1.2: Docker Desktop

Platforma Kubernetes

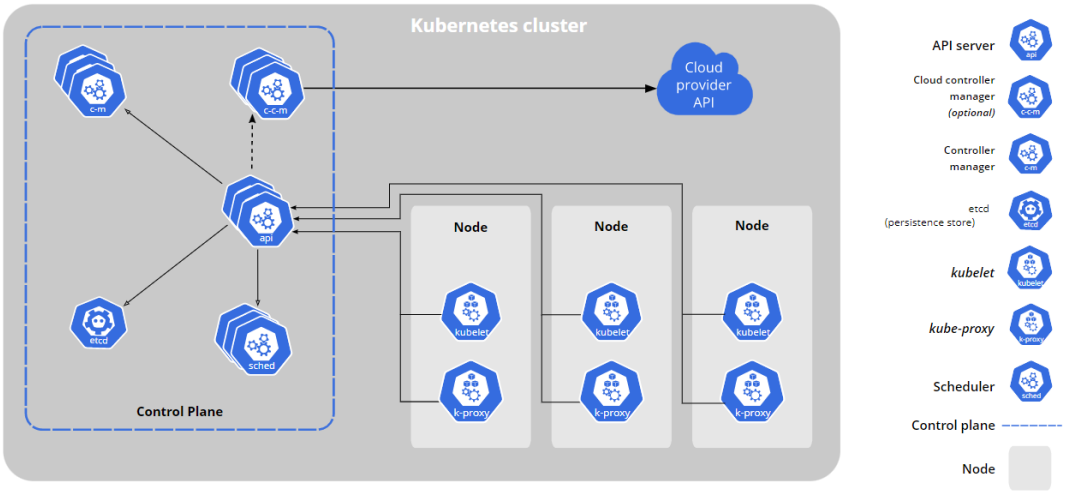
Pokretanje sistema na praktičan i skalabilan način, omogućila je platforma *Kubernetes* [16]. Ona pruža potrebnu infrastrukturu za pokretanje aplikacija zapakovanih u *Docker* kontejnere, kao i za upravljanje njima i ostalim pratećim delovima sistema.

Kubenetes omogućava:

1. Pronalaženje servisa koristeći *DNS* ime ili *IP* adresu,
2. Balansiranje saobraćaja kroz mrežu u zavisnosti od opterećenosti,

3. Korišćenje diska i drugih skladišta podataka,
4. Automatsko ažuriranje (eng. *update*) verzija aplikacije kao i vraćanje na prethodnu verziju,
5. Upravljanje resursima i pakovanje kontejnera na mašine,
6. *Self-healing* - automatski restart kontejnera koji ne ispunjavaju uslove zdravlja,
7. Upravljanje konfiguracijom, šiframa, sertifikatima i drugim osetljivim informacijama - uz samostalno ažuriranje bez potrebe za promenom slike kontejnera.

Na slici 1.3 prikazana je arhitektura klastera Kubernetes, preuzeta iz zvanične dokumentacije [17].



Slika 1.3: Arhitektura klastera Kubernetes

Klaster Kubernetes se sastoji iz dve celine: kontrolnog dela i skupa radnih mašina (eng. *nodes*). Radne mašine na sebi pokreću *mahune* (eng. *pod*) u okviru kojih je smešten jedan ili više kontejnera, a u okviru svakog se nalazi aplikacija. Mahuna je najmanja jedinica koju je moguće kreirati i pokrenuti na klasteru, a kontejneri unutar njega imaju istu specifikaciju i dele lokalnu mrežu.

Kako bi pokrenute aplikacije bile dostupne na mreži, kreira se apstrakcija koja se naziva *Servis*. Ova apstrakcija povezuje grupu mahuna i definiše način na koji im se pristupa: koristeći jedinstveno *DNS* ime ili *IP* adresu (eng. *Service endpoints*). Slanjem poruka na ovu adresu, Kubernetes sam usmerava i balansira saobraćaj ka mahunama, od kojih svaka ima jedinstvenu *IP* adresu.

U okviru svake radne mašine je pokrenuto još nekoliko sistemskih procesa:

1. *kuberlet* - Zadužen za pokretanje kontejnera na mašini, kao i za praćenje rada kontejnera i njihovog zdravlja.

2. *kube-proxy* - Zadužen za podešavanje pravila mreže koja omogućavaju slanje i prijem poruka mahunama, koristeći specifikacije definisane servisima.

3. *Container runtime* - Zadužen za pokretanje kontejnera.

Kontrolni deo klastera služi za upravljanje klasterom, a funkcionalnosti su obezbeđene kroz nekoliko komponenti:

1. *kube-apiserver* - Server API koji prima zahteve upućene klasteru.

2. *etcd* - Služi kao skladište podatka u klasteru.

3. *kube-scheduler* - Zadužen za smeštanje novih mahuna na mašine u skladu sa dostupnim i traženim resursima, kao i drugim specifikacijama (na primer, međusobni afinitet servisa).

4. *kube-controller-manager* - Sastoji se od nekoliko kontrolera:

   a) *node-controller* - Omogućava reagovanje u slučajevima kada mašine postanu nedostupne.

   b) *job-controller* - Izvšava dodatne poslove na klasteru.

   c) *endpoints-controller* - Podešava endpointe, tj. omogućava pronalaženje servisa i mahuna koje obuhvata. Ažurira adrese u slučaju promena (dodavanja i brisanja mahuna).

   d) *Service Account & Token Controller* - Resursi u okviru klastera se mogu podeliti po imenskim prostorima (eng. *namespace*). Ovi kontroleri omogućavaju podešavaju prava pristupa u okviru imenskih prostora.

5. *cloud-controller-manager* - Predstavlja vezu između klastera i snabdevača resursa u oblaku (eng.*cloud provider*).

Sve aplikacije su pokrenute na fizičkim (ili virtuelnim) mašinama koje su date na raspolaganju klasteru tokom njegovog kreiranja ili ažuriranja. Neki kontejneri/mahune mogu biti pokrenuti na istoj mašini, ali to zavisi od potražnje i raspodele resursa, kao i od drugih specifikacija. Jedna od prednosti Kubernetesa je što on vodi

računa o tome gde je koji proces pokrenut, te pokušava da obezbedi najbolju otpornost na greške u sistemu (na primer, restartovanje mašine) i time obezbedi visoku dostupnost pokrenutih servisa.

Dodatno, visoka dostupnost servisa se obezbeđuje kroz kontrolisani proces ažuriranja aplikacija, koristeći mehanizam koji se naziva eng. *rolling update*. Ovaj sistem podrazumeva da se inkrementalno zamenjuju mahune, sa novokreiranim mahunama koje su pokrenute sa novom verzijom. Tokom ažuriranja, saobraćaj do mahuna se automatski raspoređuje samo ka dostupnim mahunama.

Za definisanje resursa i njihovih specifikacija, kao i konfiguracije na klasteru koriste se datoteke u formatu *yaml*. Prosleđivanjem ovih datoteka klasteru se rade promene na klasteru.

Za komunikaciju sa kontrolnim delom Kubernetes klastera se koristi kilijent *kubectl*. Neke od komandi su predstavljene u nastavku.

```
// Opšti oblik komande
kubectl [command] [TYPE] [NAME] [flags]
// Pravljenje resursa i njihovih specifikacija
kubectl apply -f deploy.yaml
// Izlistavanje servisa na klasteru
kubectl get svc
// Izlistavanje mahuna
kubectl get pods
// Pristup bash konzoli na mašini u okviru koje je pokrenuta mahuna
kubectl exec mypod-m96mk -it mypod-m96mk  -- /bin/bash
// Podešava automatsko skaliranje broja mahuna
kubectl autoscale deployment computenode --cpu-percent=50 --min=1 --max=10
```

Platforma *Microsoft Azure*

Programska implementacija sistema je uključila njegovo pokretanje u oblaku, a za to je korišćena platforma *Microsoft Azure* [3]. Ova platforma pruža veliki broj softverskih i infrastrukturnih rešenja, kao i propratnih servisa i mogućnosti koje poboljšavaju celokupno iskustvo korišćenja.

Najrelevantniji resursi korišćeni za pokretanje sistema u na *Azure* platformi su:

1. *Azure Kubernetes Service* (skr. AKS) [2] - Resurs koji predstavlja klaster *Kubernetes*.

2. *Azure SQL Database* (skr. AzureSQLDB) [5] - Resurs koji predstavlja relacionu bazu podataka.

3. *Azure Active Directory* [1] (skr. AAD) - Resurs koji omogućava kreiranje identiteta (npr. korisnike, grupe) i njihovo podešavanje (prava pristupa, login...).

4. *Azure Monitor* (skr. AM) [4] - Skup propratnih funkcionalnosti koje pružaju uvid u ponašanje Azure resursa. Sakuplja logove i metrike, daje alate za njihovo analiziranje, kao i mogućnost uzbunjivanja (eng. *alerts*) i reagovanja na dešavanja od interesa. Ima podršku i za tehnike mašinskog učenja nad sakupljenim podacima. Opis integracije sa servisom *Azure Monitor* se nalazi u odeljku 2.6 - Praćenje rada sistema.

Dodatni alati

Swagger / OpenAPI

OpenAPI [22] je specifikacija *REST API*-a (eng. *Application Programming Interface*) koja ne zavisi od programskog jezika. *Swagger* [25] predstavlja skup alata koji koriste *OpenAPI* specifikaciju.

Korišćenjem skupa alata *Swagger*, moguće je dokumentovati serverski *web REST API* na standardizovan način, u datoteci formata *json*. Za pravljenje ove specifikacije, koristi se alat *Swashbuckle* [26].

Za generisanje klijenta u radnom okviru *.NET Core*, koristi se alat *NSwag* [20]. Na osnovu prethodno generisane datoteke *json*, on generiše klasu koja sadriži *HTTP* klijenta i potrebnu dokumentaciju. Ovo je praktičan i efikasan način da se automatski generiše kod klijenta pomoću kojeg se šalju zahtevi serveru.

Opis korišćenja skupa alata *Swagger* se nalazi u odeljcima 2.3 - Implementacija i 2.5 - Okvir za testiranje.

Glava 2

Sistem DCS

Kao sastavni deo master rada razvijen je projekat nazvan *Distributed Computation System* (skr. *DCS*).

Glavni zadatak sistema je da omogući korisniku obradu podataka na distribuirani način. U jednostavnom slučaju, sistem obrađuje prosleđeni niz atomičnih poslova, a zatim agregira njihove rezultate. Ovo predstavlja distribuiranu implementaciju funkcionalnosti eng. *map-reduce*. U naprednijem slučaju, sistem DCS je moguće usko specijalizovati za obradu određenog tipa posla. Tada se prilikom implementacije koristi i poznavanje prirode posla, što omogućava da se on podeli na podjedinice. Krajnji rezultat se takodje dobija agregiranjem podrezultata, s tim da podjedinice posla mogu biti međusobno zavisne, te je za celokupno izvršavanje posla potrebno pratiti plan distribuiranog izvršavanja.

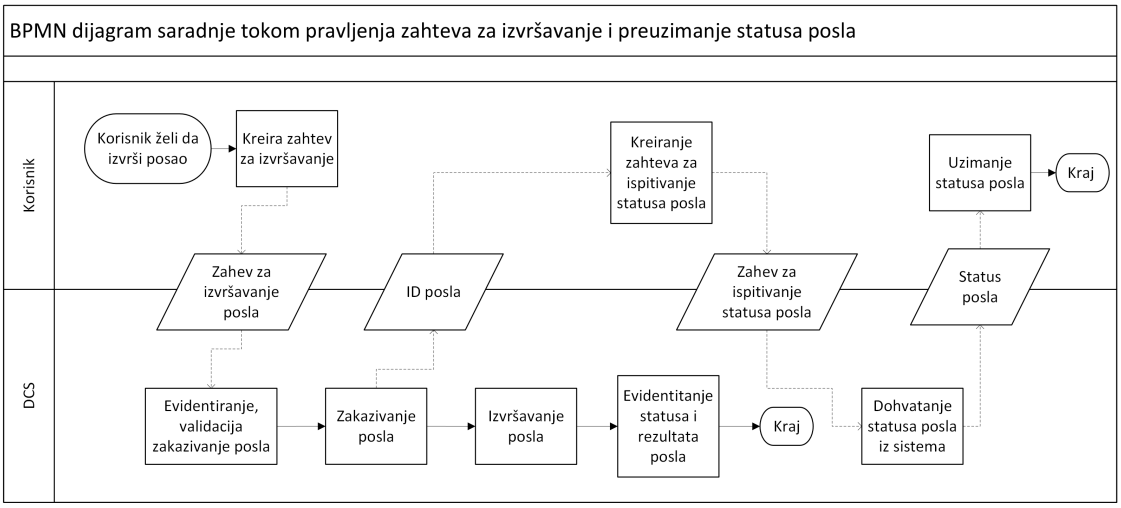
Jedan od glavnih ciljeva sistema DCS je njegova modularnost i primenljivost. Njega je, uz jednostavne izmene, moguće proširiti kako bi podržao različite tipove poslova koje su od značaja njegovim korisnicima, oslanjajući se pritom na zajedničku infrastrukturu za obradu poslova. Ova infrastruktura je opisana u narednim poglavljima.

2.1 Funkcionalnosti

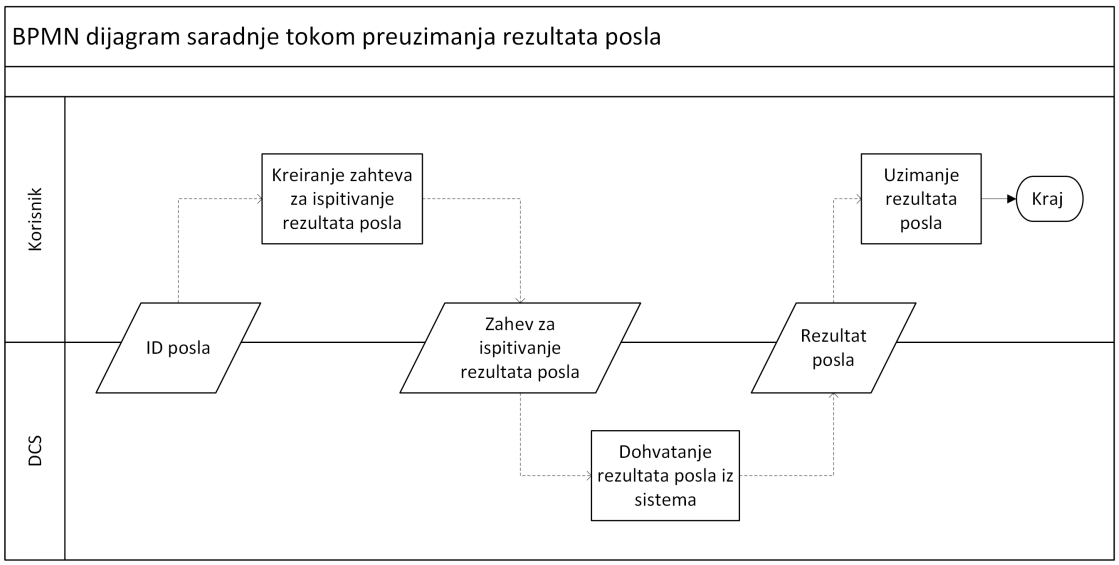
U ovom poglavlju je prikazan pregled funkcionalnosti sistema za distribuirano izračunavanje DCS. Glavna funkcionalnost sistema je obrada korisnikovih zahteva za izvršavanje poslova.

U nastavku se nalaze BPMN dijagrami (eng. *Business Process Modeling Notation*) koji opisuju saradnju između korisnika i sistema. Na slici 2.1 je predstavljen tok

saradnje koji uključuje pravljenje posla i njegovo izvršavanje. Na slici 2.2 je predstavljena saradnja prilikom preuzimanja rezultata, a na slici 2.3 je opisana saradnja prilikom otkazivanja prethodno napravljenog posla.

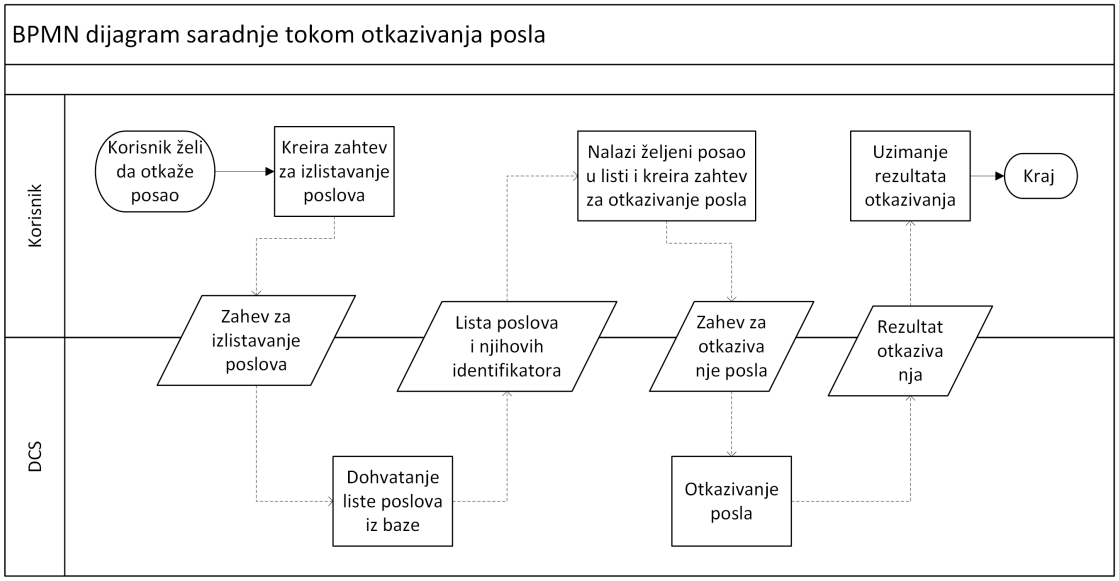


Slika 2.1: BMPN dijagram saradnje - Pravljenje zahteva za izvršavanje i provera statusa posla



Slika 2.2: BMPN dijagram saradnje - Preuzimanje rezultata

U narednim odeljcima su opisane funkcionalnosti kroz slučajeve upotrebe, iz ugla korisnika i iz ugla dva tipa administratora: bezbednosnog i klaster administratora.



Slika 2.3: BMPN dijagram saradnje - Otkazivanje posla

Slučajevi upotrebe - Korisnik

Slučajevi upotrebe iz ugla korisnika su:

1. Pravljenje posla - Prvi korak koji korisnik radi kako bi zaposlio sistem DCS da izvršava posao.

2. Pregled poslova - Izlistavanje jednog ili više poslova kako bi korisnik preuzeo status o uspešnosti, kao i identifikatore poslova.

3. Preuzimanje rezultata izvršavanja posla - Kada je posao izvršen, korisnik preuzima rezultate.

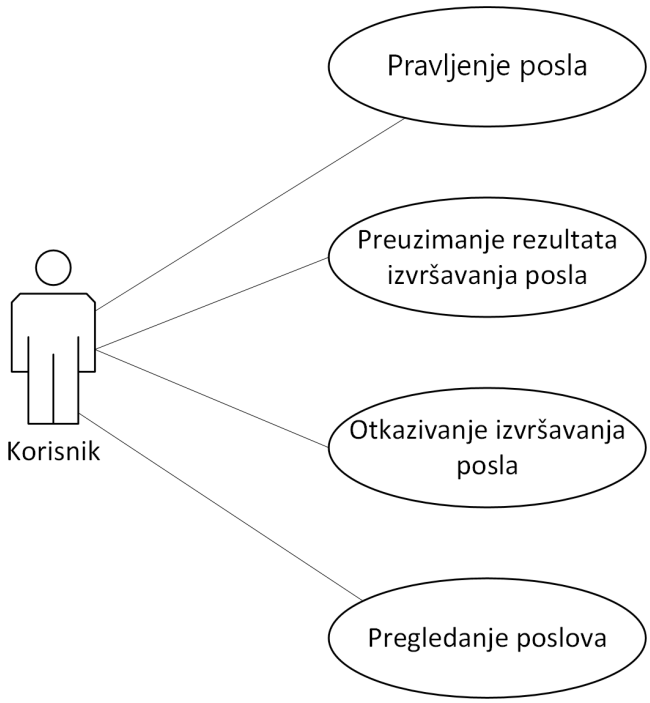
4. Otkazivanje posla - Korisnik ima mogućnost da prekine izvršavanje prethodno napravljenog posla.

Ovi slučajevi upotrebe su prikazani na slici 2.4, a detaljnije su opisani u narednim odeljcima.

Pravljenje posla

1. Naziv: Pravljenje posla.

2. Akter: Korisnik koji želi da izvrši posao nad ulaznim podacima.



Slika 2.4: UML dijagram slučajeva upotrebe - Korisnik

3. Kratak opis: Korisnik šalje zahtev za kreiranje posla. Sistem validira zahtev i vraća potvrdu o uspešnosti primanja zahteva.

4. Preduslovi: Korisnik ima pristup internetu. Korisnik ima neophodna prava da bi poslao zahtev sistemu. Sistem je u funkciji.

5. Postuslovi: Sistem je evidentirao novi zahtev za posao i prosledio ga na asinhrono izvršavanje.

6. Tok događaja:

   a) Korisnik pravi zahtev za dodavanje posla. Zahtev se sastoji od:

      i. Specifikacije tipa posla,

      ii. Niza ulaznih podataka.

   b) Korisnik šalje zahtev sistemu preko interneta koristeći definisani interfejs.

   c) Sistem proverava prava korisnika.

   d) Sistem validira novopridošli zahtev za izvršavanje posla.

   e) Sistem evidentira novi zahtev za izvršavanje u bazi.

f) Sistem stavlja zahtev u red za asinhrono izvršavanje.

g) Korisnik dobija potvrdu da je posao prihvaćen i identifikator po kojem je posao zaveden u sistemu.

7. Alternativni tok događaja:

   a) Korisnik nema prava da podnese zahtev. Ukoliko u koraku 6c korisnik nema neophodna prava, sistem odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6b glavnog toka.

   b) Zahtev je neispravan. Ukoliko u koraku 6d sistem prepozna neispravan zahtev, odbija ga uz odgovarajuću grešku. Neispravan zahtev može biti:

      i. Neispravna specifikacija tipa posla. Zahtevani tip posla mora biti podržan od strane sistema,

      ii. Format ulaznih podataka nije odgovarajući.

      Proces se nastavlja u koraku 6a glavnog toka.

   c) Sistem je preopterećen i nema dovoljno resursa da zakaže novi posao. Ukoliko u koraku 6f sistem proceni da nema dovoljno resursa za izvršavanje posla, on odbija zahtev uz grešku da je sistem preopterećen i da korisnik pokuša kasnije. Proces se nastavlja u koraku 6b glavnog toka.

8. Podtokovi: /

9. Specijalni zahtevi: /

10. Dodatne informacije: /

Pregled poslova

1. Naziv: Pregled poslova.

2. Akter: Korisnik koji želi da preuzme listu evidentiranih poslova u sistemu.

3. Kratak opis: Korisnik šalje zahtev za izlistavanje evidentiranih poslova. Sistem validira zahtev i vraća tražene informacije o evidentiranim poslovima.

4. Preduslovi: Korisnik ima pristup internetu. Korisnik ima neophodna prava da bi poslao zahtev sistemu. Sistem je u funkciji. Posao je izvršen i rezultati su dostupni u bazi.

5. Postuslovi: Sistem je prosledio rezultate izvršavanja korisniku.
6. Tok događaja:
   a) Korisnik pravi zahtev za izlistavanje poslova.
   b) Korisnik šalje zahtev sistemu preko interneta koristeći definisani interfejs.
   c) Sistem proverava prava korisnika.
   d) Sistem šalje listu evidentiranih poslova dostupnih u bazi.
   e) Korisnik dobija listu evidentiranih poslova.
7. Alternativni tok događaja:
   a) Korisnik nema prava da podnese zahtev. Ukoliko u koraku 6c korisnik nema neophodna prava, sistem odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6b glavnog toka.
8. Podtokovi: /
9. Specijalni zahtevi: /
10. Dodatne informacije: Rezultujuća lista poslova sadrži informacije o svakom evidentiranom poslu, i to:
    a) Identifikator posla,
    b) Status posla,
    c) Vreme početka izvršavanja posla,
    d) Vreme završetka izvršavanja posla (ako je posao završen).

Preuzimanje rezultata posla

1. Naziv: Preuzimanje rezultata posla.
2. Akter: Korisnik koji želi da pruzme rezultate prethodno zakazanog posla.
3. Kratak opis: Korisnik šalje zahtev za preuzimanje rezultata izvršavanja. Sistem validira zahtev i vraća tražene rezultate.
4. Preduslovi: Korisnik ima pristup internetu. Korisnik ima neophodna prava da bi poslao zahtev sistemu. Sistem je u funkciji. Posao je izvršen i rezultati su dostupni u bazi.

5. Postuslovi: Sistem je prosledio rezultate izvršavanja korisniku.
6. Tok događaja:
    a) Korisnik pravi zahtev za preuzimanje rezultata posla. Zahtev se sastoji od identifikatora posla za koji želi da preuzme rezultate.
    b) Korisnik šalje zahtev sistemu preko interneta koristeći definisani interfejs.
    c) Sistem proverava prava korisnika.
    d) Sistem validira novopridošli zahtev za preuzimanje rezultata posla.
    e) Sistem šalje rezultate korisniku.
    f) Korisnik dobija rezultate.
7. Alternativni tok događaja:
    a) Korisnik nema prava da podnese zahtev. Ukoliko u koraku 6c korisnik nema neophodna prava, sistem odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6b glavnog toka.
    b) Zahtev je neispravan. Ukoliko u koraku 6d sistem prepozna da posao sa datim identifikatorom ne postoji, odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6a glavnog toka.
8. Podtokovi: /
9. Specijalni zahtevi: /
10. Dodatne informacije: Ukoliko je posao uspešno izvršen, rezultat izvršavanja se sastoji od traženog rezultata izračunavanja. Ukoliko je posao neuspešno obrađen, rezultat se sastoji od informacije o grešci.

Otkazivanje posla

1. Naziv: Otkazivanje posla.
2. Akter: Korisnik koji želi da otkaže prethodno zakazani posao.
3. Kratak opis: Korisnik šalje zahtev za otkazivanje posla. Sistem validira zahtev i vraća potvrdu o uspešnosti otkazivanja.
4. Preduslovi: Korisnik ima pristup internetu. Korisnik ima neophodna prava da bi poslao zahtev sistemu. Sistem je u funkciji.

5. Postuslovi: Sistem je otkazao posao i ažurirao evidenciju posla u bazi.
6. Tok događaja:
    a) Korisnik pravi zahtev za otkazivanje posla. Zahtev se sastoji od identifikatora posla koji želi da otkaže.
    b) Korisnik šalje zahtev sistemu preko interneta koristeći definisani interfejs.
    c) Sistem proverava prava korisnika.
    d) Sistem validira novopridošli zahtev za otkazivanje posla.
    e) Sistem sinhrono otkazuje sve operacije povezane sa poslom.
    f) Korisnik dobija potvrdu da je posao otkazan.
7. Alternativni tok događaja:
    a) Korisnik nema prava da podnese zahtev. Ukoliko u koraku 6c korisnik nema neophodna prava, sistem odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6b glavnog toka.
    b) Zahtev je neispravan. Ukoliko u koraku 6d sistem prepozna da posao sa datim identifikatorom ne postoji ili nije aktivan, odbija zahtev uz odgovarajuću grešku. Proces se nastavlja u koraku 6a glavnog toka.
8. Podtokovi: /
9. Specijalni zahtevi: /
10. Dodatne informacije: /

Slučajevi upotrebe - Administrator

Dodatne funkcionalnosti sistema uključuju slučajeve upotrebe u kojima je učesnik administrator. Ovo uključuje podešavanje bezbednosti i prava pristupa sistemu, kao i prilagođavanje sistema kako bi mogao na što efikasniji način da obradi zahteve korisnika.

Slučajevi upotrebe iz ugla administratora su:

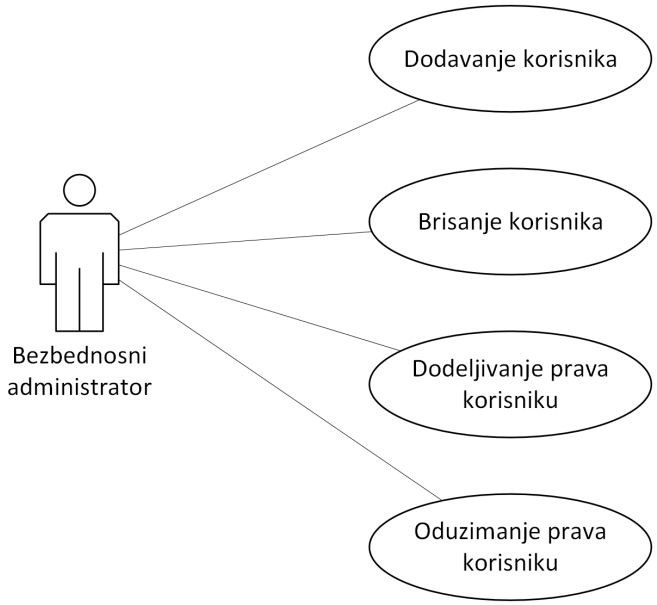
1. Podešavanja prava pristupa,
2. Podešavanje klastera.

Podešavanja prava pristupa

Prava pristupa sistemu, tj. delovima sistema dodeljuje administrator za bezbednost. Prava pristupa pojedinačnim delovima sistema se kontrolišu kroz različita prava koja mogu biti dodeljena korisniku. Tipovi prava su:

1. Iskustvo krajnjeg korisnika - Mogućnost slanja zahteva sistemu kroz definisani javni interfejs,
2. Pravo za praćenje rada sistema - Pristup telemetriji i logovima za praćenje rada sistema,
3. Administrator klastera - Pristup klasteru za devops akcije,
4. Administrator bezbednosti.

Slučajevi upotrebe iz ugla administratora za bezbednost su prikazani na slici 2.5. Ovih slučajevi upotrebe su jednostavni i intuitivni, te nisu opisani u nastavku.



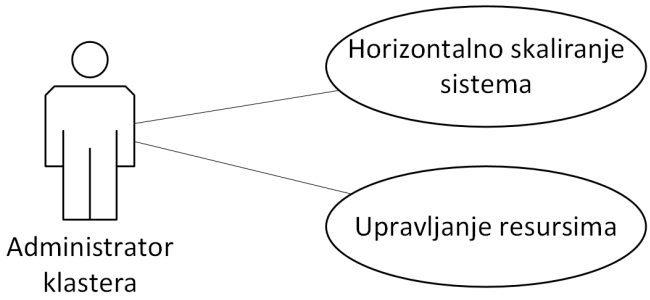
Slika 2.5: UML dijagram slučajeva upotrebe - Administrator za bezbednost

Podešavanje klastera

Administrator klastera ima mogućnost da menja konfiguraciju sistema kako bi ga prilagodio potrebama krajnjeg korisnika. To uključuje horizontalno i vertikalno

skaliranje sistema. Horizontalno skaliranje sistema podrazumeva menjanje broja pokrenutih servisa koji se koriste tokom izvršavanja posla. Vertikalno skaliranje podrazumeva ažuriranje konfiguracije kojom se dodeljuju resursi servisima (dostupna memorija i procesorsko vreme).

Slučajevi upotrebe iz ugla administratora klastera su prikazani na slici 2.6. Detalji ovih slučajeva upotrebe su konceptualno jasni, a uključuju poznavanje implementacionih detalja, te nisu dalje razrađivani.



Slika 2.6: UML dijagram slučajeva upotrebe - Administrator klastera

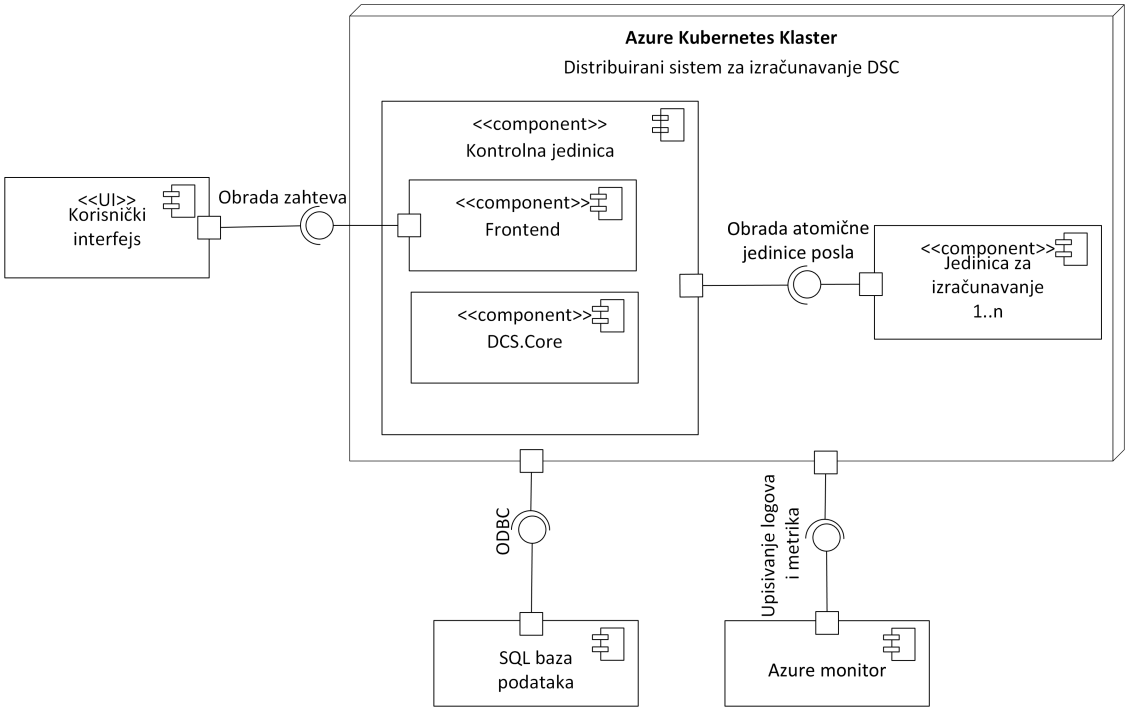
2.2 Arhitektura sistema

Sistem DCS je implementiran po uzoru na arhitekturu klijent-server.

Centralni deo sistema DCS se sastoji od dva tipa aplikacija: Kontrolne jedinice (eng. *Control Node*, skr. CtrlN) i jedinice za izračunavanje (eng. *Compute Node*, skr. CmpN). U sistemu postoji tačno jedna aplikacija kontrolne jedinice, koja je zadužena za dve logički odvojene celine: primanje zahteva od korisnika i orkestriranje izvršavanja prethodno zakazanih poslova. Sa druge strane, u zavisnosti od potreba, DCS se sastoji od jedne ili više jedinica za izračunavanje koje su zadužene za izvršavanje atomičnih (nedeljivih) poslova. CmpN prima zahteve za izvršavanje atomičnih poslova od CtrlN.

Arhitektura sistema je prikazana na slici 2.7.

Arhitekturalni obrazac korišćen prilikom razvoja sistema je Prezentacija-Apstrakcija-Kontrola (eng. *Presentation-Abstraction-Control*, skr. PAC). Sistem je hijerarhijski podeljen na agente CtrlN i CmpN. CtrlN je agent višeg novoa, koji je pozvezan sa CmpN agentima. Ovakvom arhitekturom se uvodi podela odgovornosti različitih delova servisa.



Slika 2.7: UML dijagram komponenti - Arhitektura sistema

PAC je slojevita arhitektura, te svaki od agenata CtrlN i CmpN ima PAC slojeve. Prvi sloj aplikacija je *prezenter* koji predstavlja *REST API*. Ovaj sloj aplikacije je okrenut ka spolja i definiše interfejs za komunikaciju sa servisom. U slučaju CtrlN, to je *API* preko kog prima korisnikove zahteve, a u slučaju CmpN, to je *API* preko kog prima zahteve od CtrlN. Drugi sloj aplikacija je *kontroler* koji predstavlja centralni deo agenta. U okviru kontrolera se nalazi glavna logika. Treći sloj aplikacija je *apstrakcija* i predstavlja model podataka i interfejs prema bazi podataka.

2.3 Implementacija

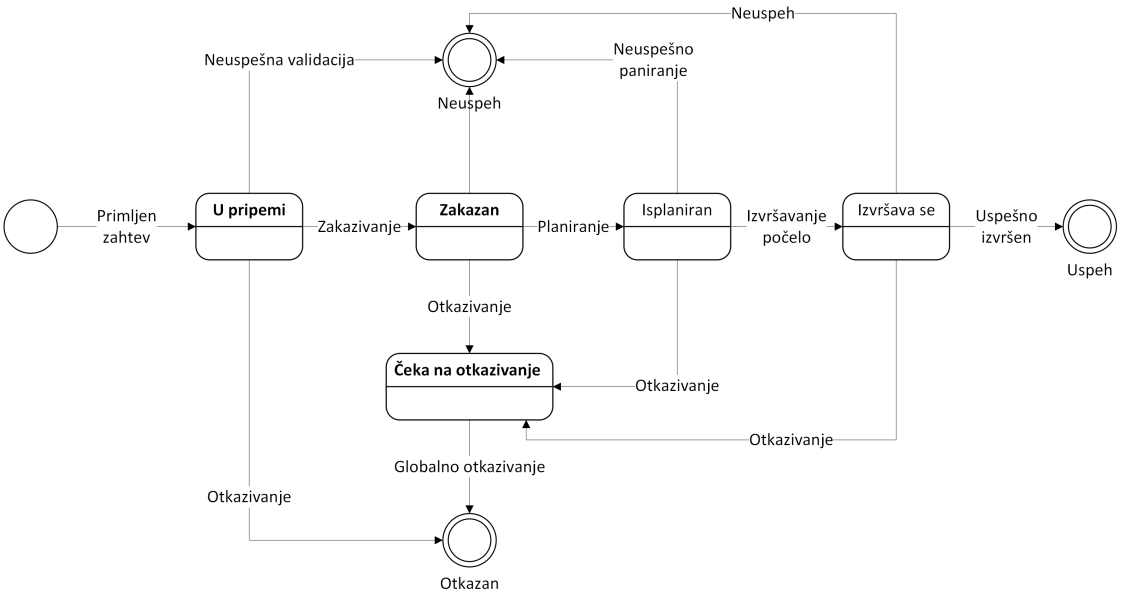
Sistem DCS je javno dostupan na servisu *GitHub* na adresi https://github.com/milana-kovacevic/DistributedComputationSystem. Za implementaciju je korišćen programski jezik C#, i radni okvir *.NET Core 6.0*. Korišćeno je razvojno okruženje *Microsoft Visual Studio Community 2022* i operativni sistem *Windows 10*.

Glavna logika distribuiranog izvršavanja nalazi se u okviru kontrolne jedinice. Jedinice za izvršavanje su jednostavni servisi koji imaju jasan cilj koji podrazumeva izvršavanje atomičnog posla i vraćanje rezultata izvršavanja. U nastavku, razrađuje se implementacija CtrlN komponente (ukoliko nije naglašeno drugačije).

Kontrolna jedinica se sastoji od dve komponente koje se nazivaju: *Frontend* (skr. FE) i Orkestrator distribuiranja (skr. DO). Detaljnije, zaduženja CtrlN su, redom:

1. *Frontend*
    a) Autentifikacija klijenta,
    b) Primanje i validacija zahteva,
2. Orkestrator distribuiranja
    a) Podela zahteva na potposlove i pravljenje plana izvršavanja,
    b) Orkestiranje izvršavanja potposlova i agregacija podrezultata,
    c) Čuvanje rezultata u bazi ili nekom drugom skladištu.

Moguća stanja kroz koja prolazi posao su prikazana na slici 2.8, a u nastavku je objašnjen način njegove obrade.



Slika 2.8: UML dijagram stanja - Posao

Kada je FE primio i evidentirao zahtev za izvršavanje posla, on ubacuje posao u red za izvršavanje. U slučaju da je sistem zauzet obradom drugih poslova, novi zahtev će čekati dok sistem ne postane spreman da ga obradi. Za implementaciju reda za čekanje, korišćeno je postojeće rešenje bezbedno pri radu sa nitima *BlockingCollection* [8]. Posebna nit aplikacije uzima poslove iz reda i šalje ih, jednog po jednog, na dalju obradu komponenti DO.

Navedena zaduženja 2a i 2b uključuju glavni deo logike distribuiranog izvršavanja. Pre nego što se počne sa izvršavanjem, neophodno je podeliti posao na potposlove i definisati njihove zavisnosti. Rezultat ovoga je predstavljen u vidu stabla u kojem listovi predstavljaju početne atomične poslove, unutrašnji čvorovi agregacije podrezultata, a grane prenos međurezultata. Koren predstavlja krajnje rešenje. U slučaju da korisnikov zahtev već sadrži niz atomičnih poslova, podela početnog zahteva je trivijalna. Na osnovu stabla zavisnosti, pravi se plan izvršavanja. Sličan način planiranja koriste baze podataka [10], kao i sistemi poput Sparka [24] predstavljen u odeljku 1.2. Plan izvršavanja zavisi od tipa posla, a u slučaju *map-reduce* funkcionalnosti podrazumeva se da se svaki atomični posao nezavisno obrađuje (funkcijom *map*), a krajnje rešenje se dobija definisanom agregacijom atomičnih rezultata (funkcijom *reduce*). Na osnovu plana izvršavanja, komponenta DO sprovodi dalje izvršavanje posla: šalje atomične poslove preko mreže do izabranih CmpN na izvršavanje. Klijent korišćen za komunikaciju između komponenti je generisan pomoću skupa alata *Swagger* opisanih u odeljku 1.3. Zakazivanje i izvršavanje atomičnih poslova se dešava asinhrono i ono ne blokira DO da uzme naredni posao iz reda i započne njegovu obradu. Dobijeni rezultati se šalju odvojenoj komponenti koja zna da agregara rezultate na osnovu plana izvršavanja. U zavisnosti od kompleksnosti agregacije, i ona može da se prosledi CmpN na izračunavanje.

DO dodatno prati uspešnost izvršavanja poslova i u slučaju uspešnog završetka, upisuje rezultat u bazu. Ukoliko dođe do greške prilikom izvršavanja, posao može da se ponovi ili da se evidentira greška u krajnjem rezultatu. Ponovno pokretanje poslova je potrebno kako bi se povećao stepen pouzdanosti izvršavanja. U distribuiranim sistemima treba imati u vidu da često može doći do problema u komunikaciji među servisima, bilo zbog preopterećenosti mreže ili eventualnih restarta individualnih aplikacija. Ovo može uzrokovati prolaznu grešku, koju je moguće prevazići novim pokušajem. Ukoliko i nakon ponovnog izvršavanja dođe do greške, posao se završava i kao krajnji rezultat se upisuje greška.

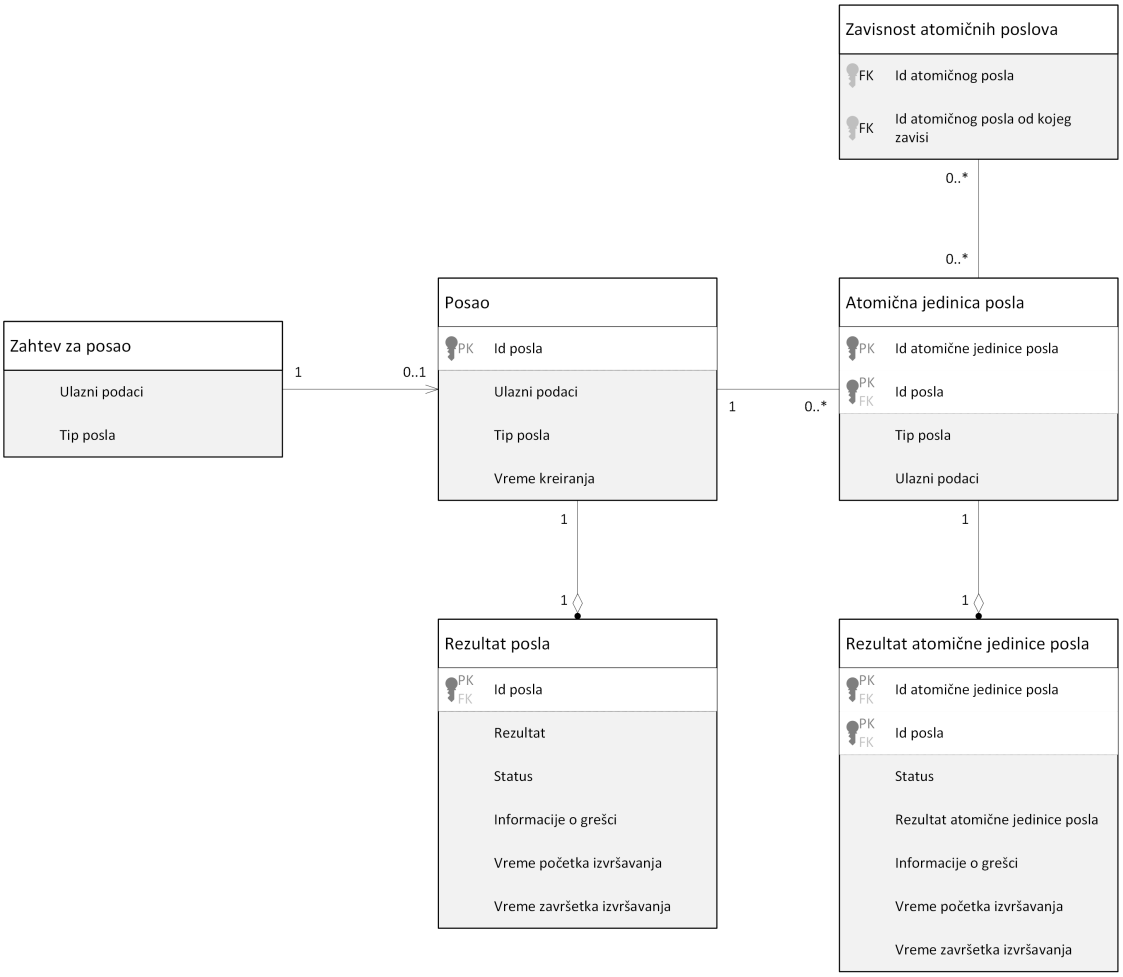
Klijent ima mogućnost da otkaže aktivan posao. Ova operacija uključuje i otkazivanje svih njegovih potposlova koji su u procesu izvršavanja ili koji čekaju na izvršavanje, te se ukupno otkazivanje dešava globalno u celom sistemu. Kada zahtev za otkazivanje stigne do komponente DO, on proverava status posla i status njegovih potposlova. Ukoliko u tom trenutku postoji barem jedan atomični posao koji čeka na izvršavanje ili se izvršava, DO prekida dalje zakazivanje i asinhrono otkazuje poslove koji su u toku izvršavanja. Tada se ažurira status posla i čeka se da se otkažu

svi atomični poslovi. Kada sistem globalno otkaže sva izvršavanja, posao je otkazan i klijentu se vraća potvrda o uspešnosti.

Baza podataka

Sistem DCS koristi relacionu bazu podataka za evidentiranje poslova. DCS ažurira stanja u kojem se nalazi posao kako bi korisnik mogao da prati šta se dešava u sistemu. Dodatno, baza podataka služi i za čuvanje rezultata i međurezultata poslova.

ER (eng. *Entity-Relation*) dijagram modela posla je prikazan na slici 2.9.



Slika 2.9: ER dijagram entiteta u bazi podataka

Autentifikacija

Za autentifikaciju klijenta je korišćen servis *AAD* [1] opisan u uvodnom poglavlju u odeljku 1.3. Korisniku su dodeljena prava korišćenja servisa time što je njegov identitet, član grupe korisnika definisane u okviru servisa AAD.

Proces autentifikacije je prikazan na slici 2.10. Korisnik pre kontaktiranja sistema DCS šalje zahtev servisu *AAD* kako bi preuzeo token za autentifikaciju. Ovaj token ima rok trajanja (uobičajeno je 60 minuta) i specifičan je za autentifikaciju na DCS sistem. Kada klijent ima token, prosleđuje ga u *header* delu *http* zahteva u polju za autentifikaciju. Sistem DCS prima zahtev i pre nego što počne da ga obrađuje, izvrši validaciju *AAD* tokena. Token sadrži otisak (eng. *thumbprint*) koji garantuje njegovu validnost. Ukoliko je token ispravan, nastavlja se sa obradom zahteva, a ukoliko nije, korisnikov zahtev se odbija uz odgovarajuću grešku.

Za bezbednu komunikaciju u produkcionom okruženju, korišćen je protokol *https*.

2.4 Pokretanje u oblaku

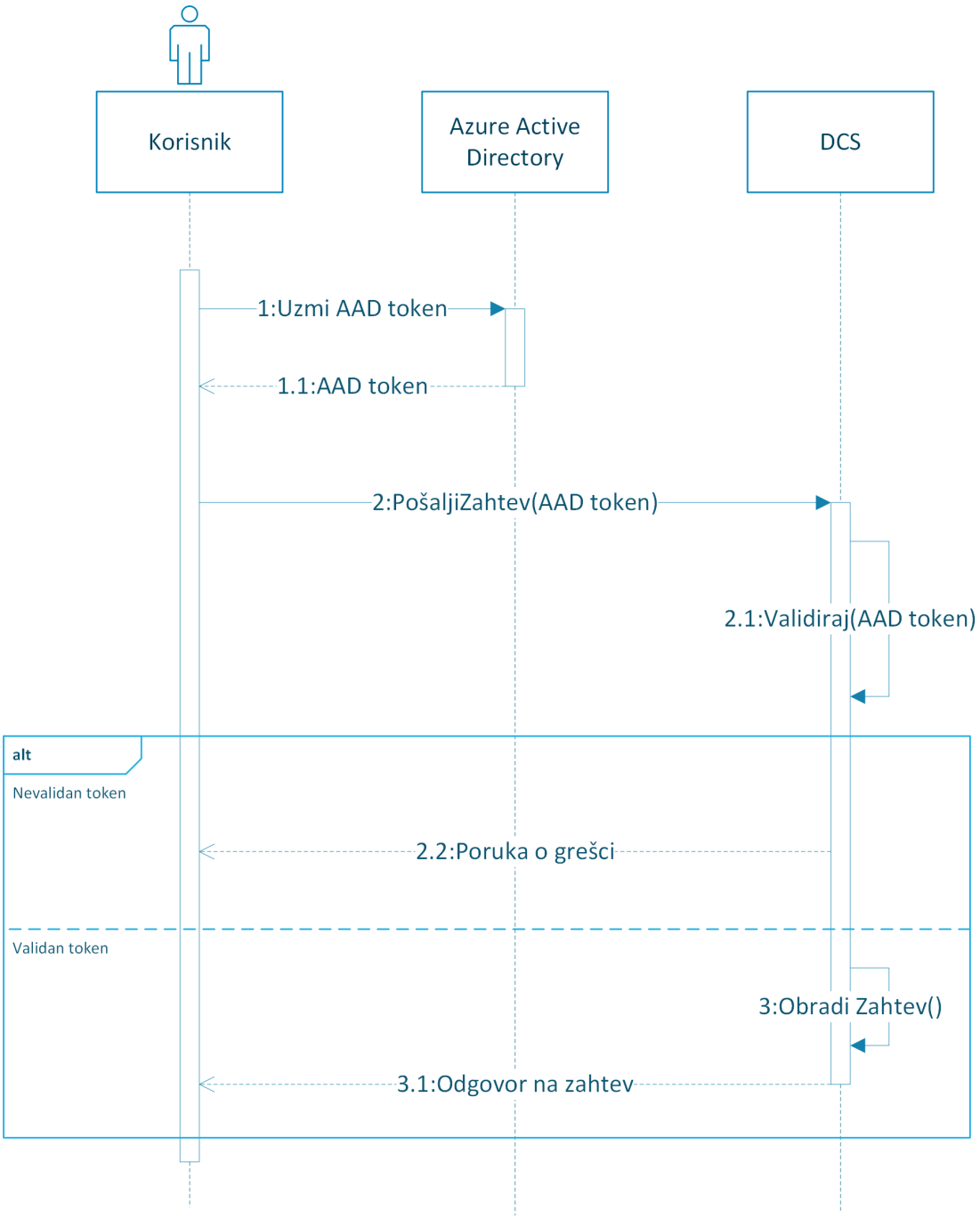
Opisana implementacija dobija smisao i punu moć pokretanjem sistema DCS na klasteru Kubernetes predstavljenom u uvodu u delu 1.3. Komponente *Control Node* i *Compute Node* se pokreću u okviru *Docker* kontejnera (uvodni deo, odeljak 1.3). Izbor radnog okvira *.NET Core* omogućava pokretanje servisa na različitim platformama *Windows*, *Mac* i *Linux*, bez izmena izvornog koda. Za izradu sistema su korišćeni *Docker*kontejneri sa operativnim sistemom *Linux*. Za pokretanje sistema DCS u oblaku je korišćena platforma *Microsoft Azure* [3]. Za upravljanje i pristup resursima u okviru ove platforme koristi se portal *Azure* [7].

Na adresi projekta na servisu *GitHub* 2.3 se nalaze datoteke potrebne za kreiranje i konfigurisanje servisa pokrenutih u okviru klastera Kubernetes, kao i pomoćne skripte koje automatizuju proces postavljanja nove verzije aplikacija na klaster.

U nastavku, na slici 2.11 je prikazana datoteka u formatu *yaml* koja definiše kreiranje aplikacije *Control Node* na klasteru Kubernetes.

Kako bi se efikasno pristupalo aplikaciji preko mreže, kreira se i odgovarajući servis koristeći datoteku u formatu *yaml* prikazanu na slici 2.12. Tada je moguće pristupiti servisu koristeći njegovo ime ili dodeljenu statičku adresu servisa.

Specifikacije za aplikaciju i servis komponente *Compute Node* su analogne, s tim da on može imati i više od jedne replike. Kreiranjem servisa CmpN, omogućava se da CtrlN komunicira prema njemu koristeći samo poznatu *IP* adresu servisa.



Slika 2.10: UML dijagram sekvence - Autentifikacija

Ova adresa se ne menja tokom postojanja servisa, a klaster Kubernetes automatski usmerava zahteve prema dostupnim, odgovarajućim mahunama, tj. kontejnerima. Više o ovome se nalazi u odeljku 1.3. Na ovaj način, sistem DCS se oslanja na postojeći mehanizan balansiranja saobraćaja u okviru klastera. Iz ugla CtrlN, on

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: controlnode
  namespace: distributed-system-dev-ns
  labels:
    app: controlnode
spec:
  replicas: 1
  selector:
    matchLabels:
      app: controlnode
  strategy: {}
  template:
    metadata:
      labels:
        app: controlnode
    spec:
      containers:
        - name: controlnode-container
          image: matfmastercr.azurecr.io/controlnode:v1.1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              protocol: TCP
          env:
            - name: ASPNETCORE_URLS
              value: http://+:80
```

Slika 2.11: Definisanje kreiranja aplikacije *Control Node*

uvek šalje zahteve na jednu adresu CmpN, a tu adresu čita iz promenljive postavljene u okruženje (eng. *environment variable*).

Servisi su definisani tako da primaju samo lokalni saobraćaj, odnosno saobraćaj u okviru mreže klastera. Koriste lokalni port 80, što je standardni port za http, a smatra se da je komunikacija unutar klastera bezbedna. CtrlN je potrebno otvoriti prema korisniku putem interneta, odnosno, dodeliti mu javnu *IP* adresu, što je moguće uraditi na više načina. Jedan od načina je kreiranjem servisa tipa *Load-Balanser*, koji mu automatski dodeljuje eksternu *IP* adresu. Međutim, ovaj način dodeljivanja adrese komplikuje podešavanja bezbednosnog protokola *https*, koji zahteva instalaciju sertifikata *TLS* na serverskoj mašini. Zbog ovoga je kreiran dodatni *API* objekat koji se naziva *Ingress*, a način njegovog kreiranja je prikazan na slici 2.13. Ovaj objekat predstavlja vezu između spoljašnjosti i unutrašnjosti klastera.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: controlnode
  namespace: distributed-system-dev-ns
  labels:
    app: controlnode
spec:
  type: ClusterIP
  ports:
  - port: 80
    name: 80-80
    protocol: TCP
    targetPort: 80
  selector:
    app: controlnode
  internalTrafficPolicy: Local
```

Slika 2.12: Definisanje kreiranja servisa *Control Node*

Dodeljena mu je statička *IP* adresa sa uvezanim *DNS* imenom, koji su prethodno kreirani preko portala. Ovaj objekat takođe usmerava zahteve ka željenim servisima na osnovu definisanih pravila. U sučaju DCS, sve zahteve (označene prefiksom /) usmerava na CtrlN. Za instaliranje i automatsko rotiranje sertifikata, kreiran je i objekat *ClusterIssuer*, koji koristi javnog snabdevača *TLS* sertifikatima, *Let's encrypt* [18]. Uz ovakvo podešavanje, produkcionom sistemu DCS se može pristupiti preko javno dostupne adrese *matf-distr-comp-sys.westeurope.cloudapp.azure.com* koristeći *https*.

Broj mahuna aplikacije CmpN je moguće automatski skalirati kreiranjem *autoscale* objekta. Komanda koja skalira broj CmpN u odnosu na procenat iskorišćenosti procesora je sledeća:

```
kubectl autoscale deployment computenode --cpu-percent=60 --min=2 --max=7 \
 --namespace=distributed-system-dev-ns
```

2.5 Okvir za testiranje

Za izradu testova korišćen je radni okvir *XUnit* [27].

Svi testovi se nalaze na servisu *GitHub* na adresi projekta u poddirektorijumu tests.

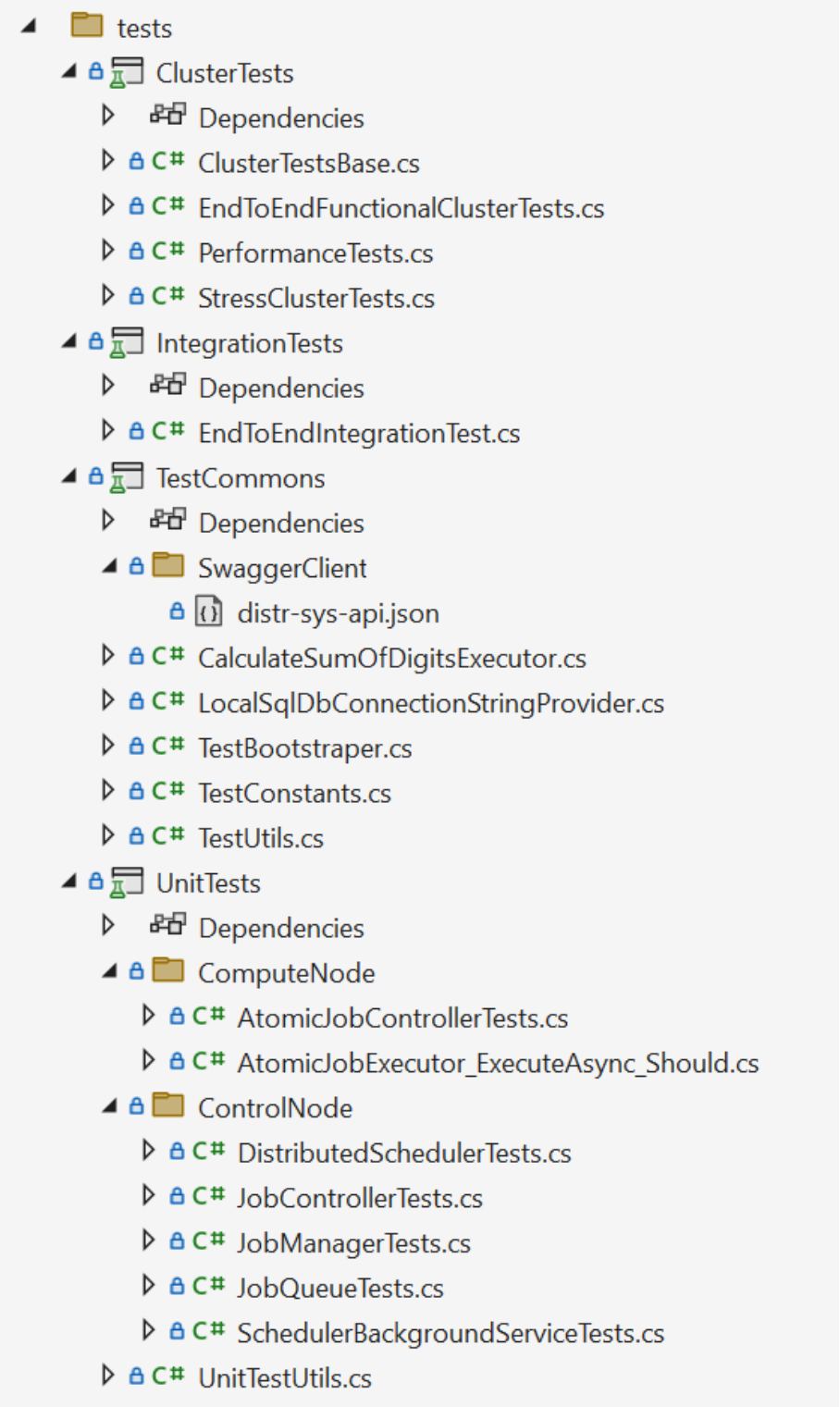
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cn-tls
  namespace: distributed-system-dev-ns
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    cert-manager.io/cluster-issuer: letsencrypt
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - matf-distr-comp-sys.westeurope.cloudapp.azure.com
    secretName: tls-secret
  rules:
  - host: matf-distr-comp-sys.westeurope.cloudapp.azure.com
  - http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: controlnode
            port:
              number: 80
```

Slika 2.13: Kreiranje objekta ingress za pristup CtrlN preko *DNS* imena i *https* protokola

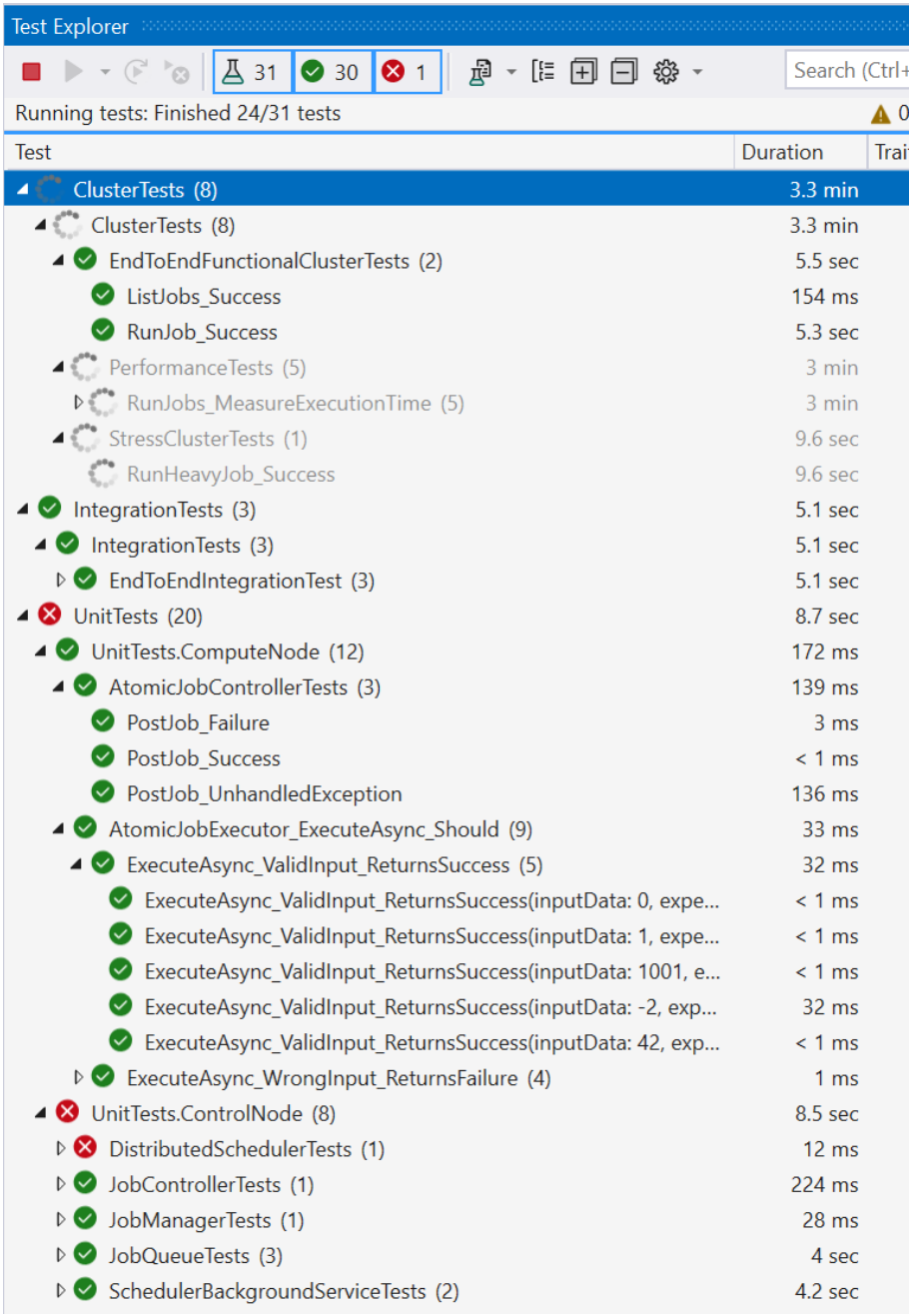
Radni okvir *XUnit* ima podršku za pisanje dva tipa testova: činjenice (eng. *Fact*) i teorije (eng. *Theory*). Činjenice su testovi koji ne primaju argumente, dok teorije predstavljaju parametrizovane testove. Specifikacijom atributa iznad tela funkcije teorijskog testa, definišu se ulazi nad kojima se test pokreće, što značajno olakšava generisanje testova sa različitim ulazima.

Struktura foldera sa testovima je prikazana na slici 2.14, a na slici 2.15 se nalazi vizuelni prikaz rezultata izvršavanja testova u okruženju *Microsoft Visual Studio*.

Sistem je testiran na više nivoa, počevši od testiranja jedinica koda za svaku komponentu, zatim testova integracije, i, na najvišem nivou, funkcionalnih testova koji ukljucuju klaster testove.



Slika 2.14: Struktura foldera sa testovima



Slika 2.15: Pokretanje testova u okruženju *Microsoft Visual Studio*

Testiranje jedinica koda

Komponente *Control Node* i *Compute Node* prate odgovarajući testovi jedinica koda. Ovi testovi se nalaze u zasebnom projektu. Tokom izrade testova jedinica koda, korišćen je i radni okvir *Moq* [19] za „podmetanje" vrednosti u zavisnim delova koda. U nastavku se nalazi slika 2.16 sa primerom testa jedinice koda u okviru komponente

CtrlN.

```
1  public class DistributedSchedulerTests
2  {
3      private readonly ServiceProvider serviceProvider;
4      private readonly Mock<IComputeNodeClientWrapper>
           mockedComputeNodeClient = new();
5
6      public DistributedSchedulerTests()
7      {
8          // Configure services using common bootstraper for tests
9          var services = new ServiceCollection();
10         TestBootstraper.ConfigureServices_Frontend(services);
11
12         // Setup mocked ComputeNodeClient
13         mockedComputeNodeClient.Setup(m =>
               m.RunAsync(It.IsAny<int>(), It.IsAny<int>(),
               It.IsAny<FrontendAtomicJobType>(),
               It.IsAny<string>()))
14              .Returns(Task.FromResult(new
                    FrontendAtomicJobResult()));
15         services.AddScoped((services) =>
               mockedComputeNodeClient.Object);
16
17         services.AddScoped<IScheduler, DistributedScheduler>();
18         serviceProvider = services.BuildServiceProvider();
19     }
20
21     [Fact]
22     public async Task ScheduleJobAsync_Success()
23     {
24         var scheduler =
               serviceProvider.GetService<IScheduler>();
25         Assert.NotNull(scheduler);
26
27         var jobToBeScheduled = UnitTestUtils.GetDummyJob();
28         await scheduler.ScheduleJobAsync(jobToBeScheduled);
29
30         mockedComputeNodeClient.Verify(client =>
               client.RunAsync(It.IsAny<int>(),
               jobToBeScheduled.JobId,
               It.IsAny<FrontendAtomicJobType>(),
               It.IsAny<string>()), Times.AtLeastOnce());
31         Assert.NotEmpty(scheduler.GetInProgressTasks());
32     }
33 }
```

Slika 2.16: Primer testa koji koristi radni okvir *Moq*

Integracioni testovi

Postoje dva tipa integracionih testova:

1. Testovi integracije unutar komponenti,
2. Testovi integracije celih komponenti.

Testovi integracije unutar komponenti testiraju međusobne zavisnosti podkomponenti CtrlN i CmpN. Na primer, zavisnost FE i DO u okviru CtrlN.

Testovi integracije celih komponenti CtrlN i CmpN se pokreću nad lokalnim okruženjem, i očekuju da je okruženje pripremljeno za njihovo izvšavanje. U okviru *Docker* kontejnera potrebno je pokrenuti jednu CtrlN aplikaciju, kao i jednu CmpN aplikaciju, sa ranije definisanim adresama i portovima. Koristi se automatski generisana klijentska klasa *DistributedComputationSystemClient* nastala korišćenjem alata *Swagger* u toku prevođenja projekta sa testovima. Ovaj klijent šalje zahteve servisu CtrlN, a u telu testova se proverava da li je odgovor od sistema očekivan.

Testovi nad klasterom

Testovi nad klasterom testiraju sistem iz ugla korisnika u produkcionom okruženju. Oni se pokreću nad klasterom u oblaku, i u njima se takođe koristi generisani klijent. Dele se na nekoliko grupa:

1. Funkcionalni testovi - Testiraju osnovne funkcionalnosti servisa u produkcionom okruženju.
2. Stres-testovi - Skup testova koji se izvršavaju kako bi se analiziralo i unapredilo ponašanje sistema pod opterećenjem.
3. Testovi performansi - Skup testova koji se pokreću nad servisom pokrenutim u oblaku, sa fokusom na praćenje vremena potrebnog da sistem odradi očekivani posao. Cilj ovog testiranja je uvid u performanse sistema u odnosu na resurse dodeljene klasteru.

Na slici 2.17 se nalazi primer koji testira funkcionalnost sistema od početka do kraja: zakazivanje posla, preuzimanje statusa, i provera rezultata.

Stres-testovi i testovi performansi ispituju granice sistema. Oni su neophodni kako bi se na vreme razumela i unapredila uska grla sistema, kao i da bi se razumele mogućnosti sistema u skladu sa dodeljenim resursima.

2.6 Praćenje rada sistema

U ovom odeljku su prikazane mogućnosti praćenja rada sistema uz pomoć servisa *Azure Monitor* za praćenje ponašanja resursa u oblaku. Kako su glavne komponente sistema pokrenute u okviru klastera *Azure Kubernetes Service*, dat je pregled načina na koji se analiziraju podaci o radu klastera. Na analogan način se može pratiti i ponašanje baze podataka *Azure SQL Database* koju koristi sistem.

Metrike

Metrike sistema predstavljaju koncizne informacije o njegovom radu u (skoro) realnom vremenu. Kroz metrike možemo da vidimo trenutnu iskorišćenost resursa klastera, status mašina, mahuna, kontejnera, itd.

Na slici 2.18 je prikazan pogled na sekciju *Insight* pripadajućeg klastera kojem se pristupa preko portala [7].

Na slici 2.19 je prikazan pregled pokrenutih servisa na klasteru.

Logovi

Logovi predstavljaju informacije o radu sistema koje ispisuju pokrenuti servisi u tekstualnom formatu. Oni su automatski sakupljeni sa mašina i prosleđeni na mesto za čuvanje, gde im je moguće pristupiti kroz propratne alate.

Za pregled logova korišćen je alat *Azure Log Analytics* dostupan u okviru servisa *Azure Monitor*. Ovaj alat pruža moćan deskriptivni jezik kojim se mogu pretraživati, filtrirati, sortirati i analizirati logovi. Rezultate upita je takođe moguće vizuelizovati u vidu grafikona.

Na slici 2.20 se nalazi prikaz stranice za pregled logova kontejnera koji sadrži aplikaciju *Control Node*.

Na slici 2.21 se nalazi metrika o broju uspešnih i neuspešnih izvršenih poslova u periodu od sat vremena, agregiranih na 10 minuta.

Uzbunjivači

Korisnik koji ima pravo pristupa logovima i metrikama može da napravi automatsku uzbunu (eng. *Alert*) u nekim situacijama od značaja. Ovaj servis periodično proverava dostupne metrike i logove, i proverava da li su se ispunjeni uslovi definisani tokom njegovog kreiranja. Ukoliko zaključi da postoji problem, podiže uzbunu

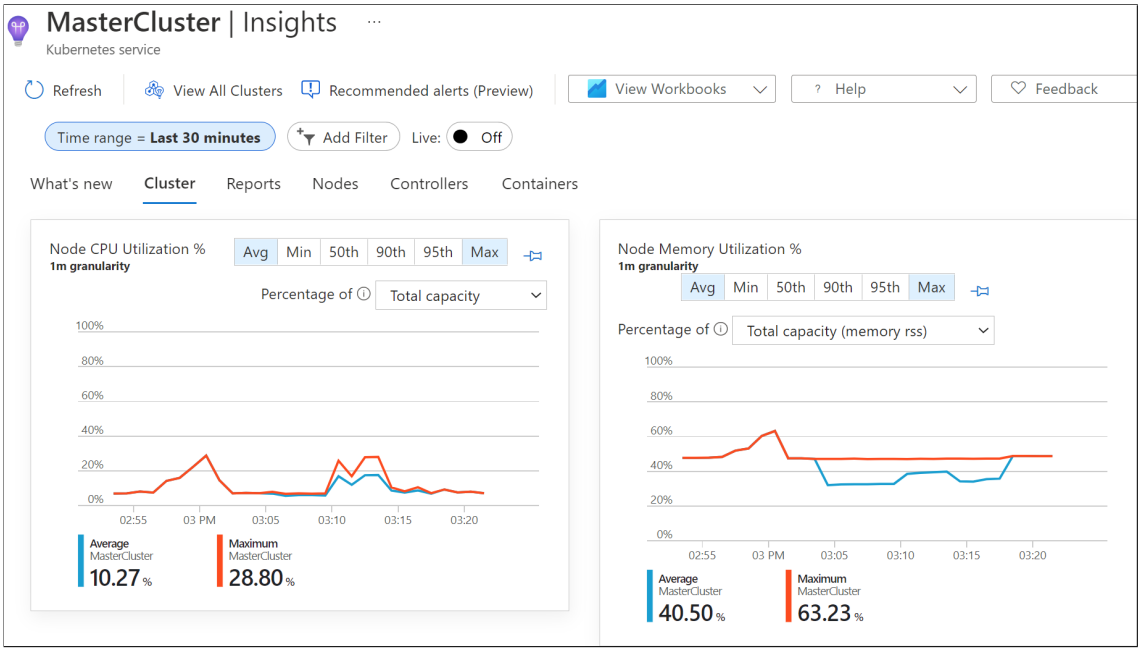
na ranije definisan način (mejlom, SMS-om, pozivom, obaveštenjem) i, ukoliko postoje, automatski pokreće definisane akcije. Problemi u produkcionom okruženju se javljaju, i u tim slučajevima je cilj što pre reagovati i obezbediti da sistem bude zdrav i na raspolaganju korisnicima.

U nastavku se na slici 2.22 nalazi primer aktivnog upozoranja u slučaju grešaka koje bacaju komponente sistema.

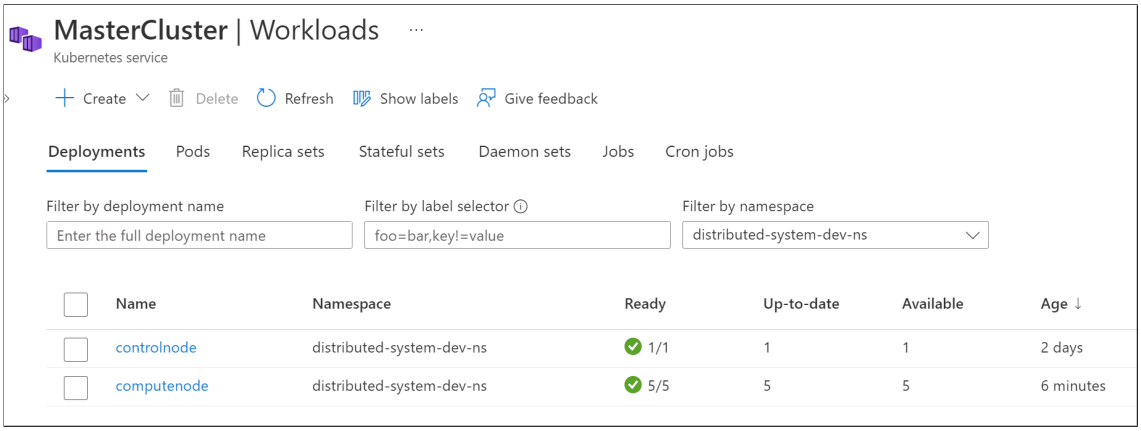
Na slici 2.23 nalazi primer aktivnog opšteg upozorenja u slučaju nedostatka memorije na mašinama. Analogno, postoji i uzbunjivač koji prati procenat iskorišćenosti dostupnog procesora.

```
1  [Fact]
2  public async void RunJob_Success()
3  {
4      var inputData = new Collection<AtomicJobRequestData>()
5      {
6          new AtomicJobRequestData() { InputData ="42" },
7          new AtomicJobRequestData() { InputData ="142" },
8      };
9      string expectedTotalSum = "1453";
10
11     var request = new JobRequestData()
12     {
13         JobType = JobType.CalculateSumOfDigits,
14         InputData = inputData
15     };
16
17     // Create job.
18     var job = await _client.CreateAsync(request);
19     Assert.NotNull(job);
20
21     // Verify job is created.
22     var jobFromSystem = await _client.JobsAsync(job.JobId);
23     Assert.NotNull(jobFromSystem);
24
25     // Poll and verify job state until it's successfully
           completed.
26     await TestUtils.PollUntilSatisfied(
27         job.JobId,
28         (jobId) =>
29         {
30             var jobFromSys =
                   _client.JobsAsync(jobId).GetAwaiter().GetResult();
31             return jobFromSys.State == JobState.Succeeded;
32         },
33         timeout: defaultTimeout);
34
35     // Verify aggregated result.
36     var jobResult = await _client.JobResultsAsync(job.JobId);
37     Assert.Equal(string.Empty, jobResult.Error);
38     Assert.Equal(expectedTotalSum, jobResult.Result);
39     Assert.Equal(JobState.Succeeded, jobResult.State);
40
41     // Delete job.
42     await _client.DeleteAsync(job.JobId);
43
44     // Now getting job should throw 404.
45     var exception = Assert.ThrowsAsync<ApiException>(async ()
           => await _client.JobsAsync(job.JobId));
46     Assert.Equal<int>((int)HttpStatusCode.NotFound,
           exception.Result.StatusCode);
47 }
```

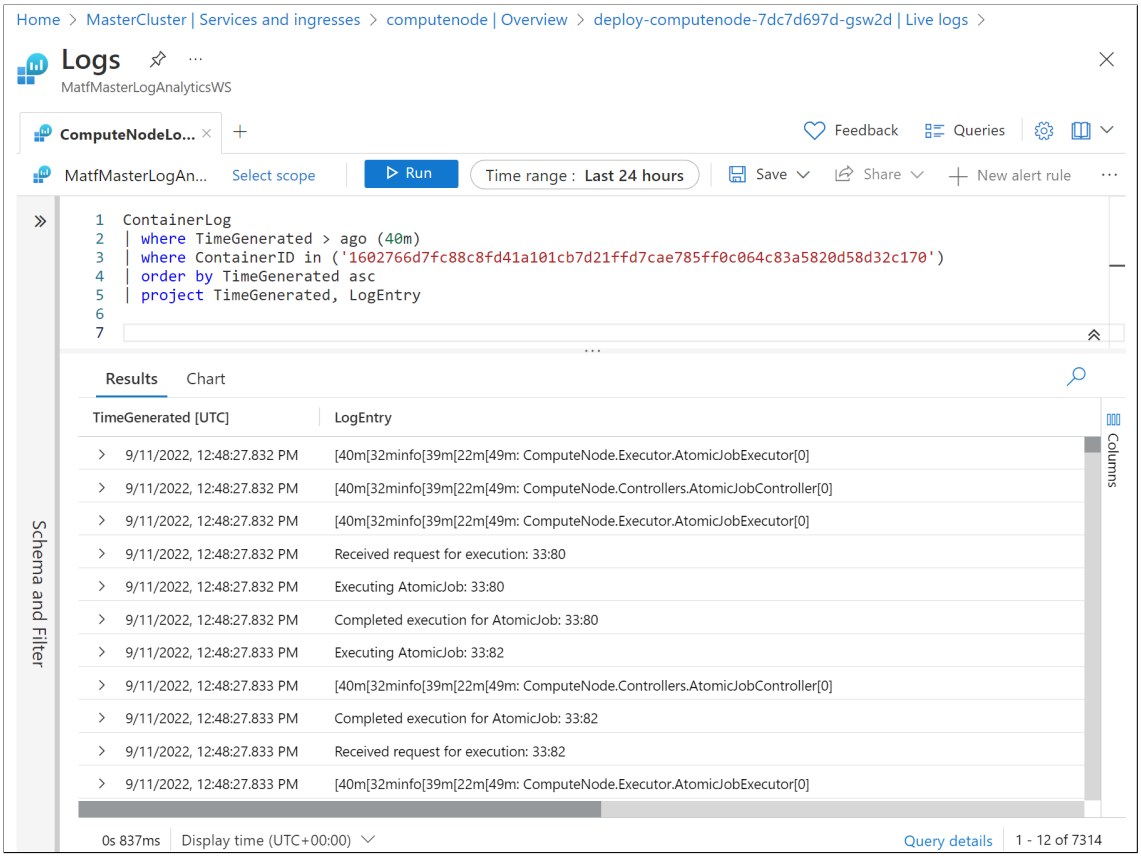
Slika 2.17: Primer funkcionalnog testa



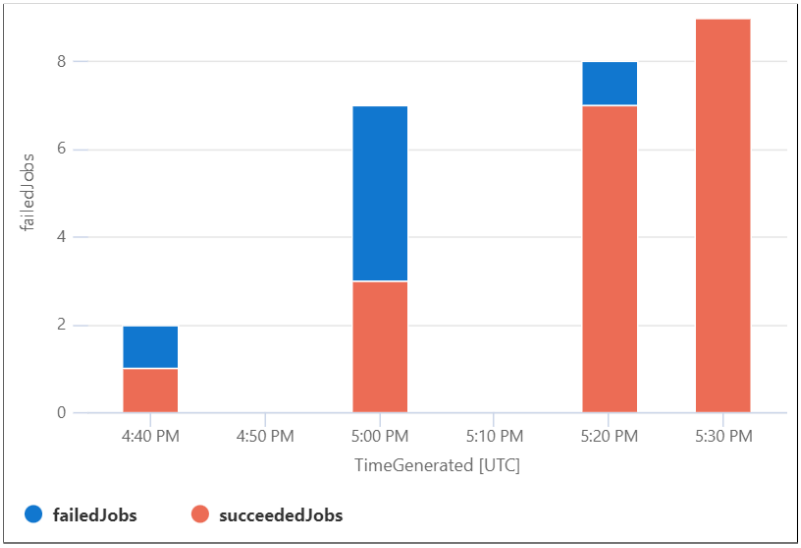
Slika 2.18: Pregled metrika klastera



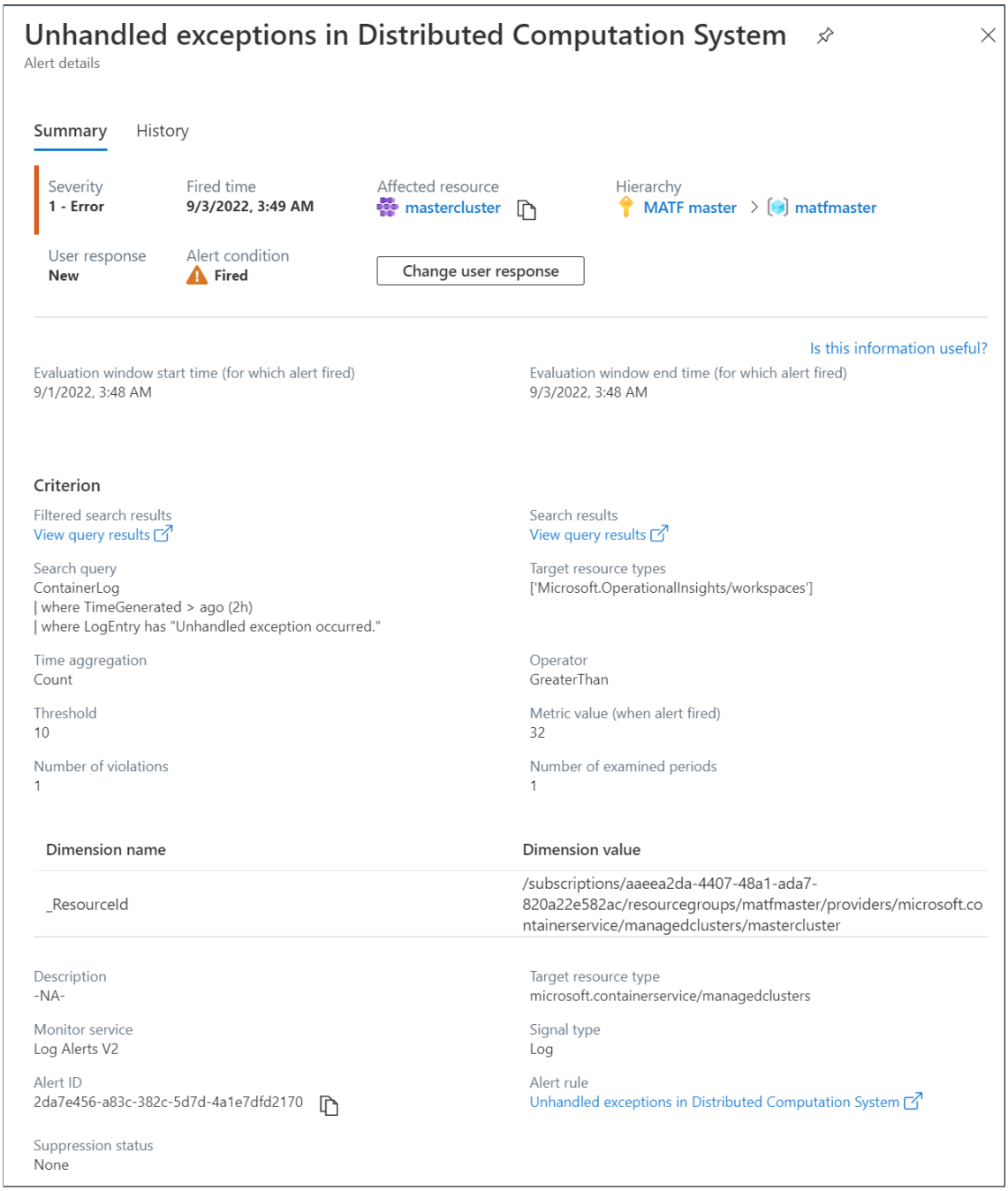
Slika 2.19: Pogled na metrike o statusu klastera



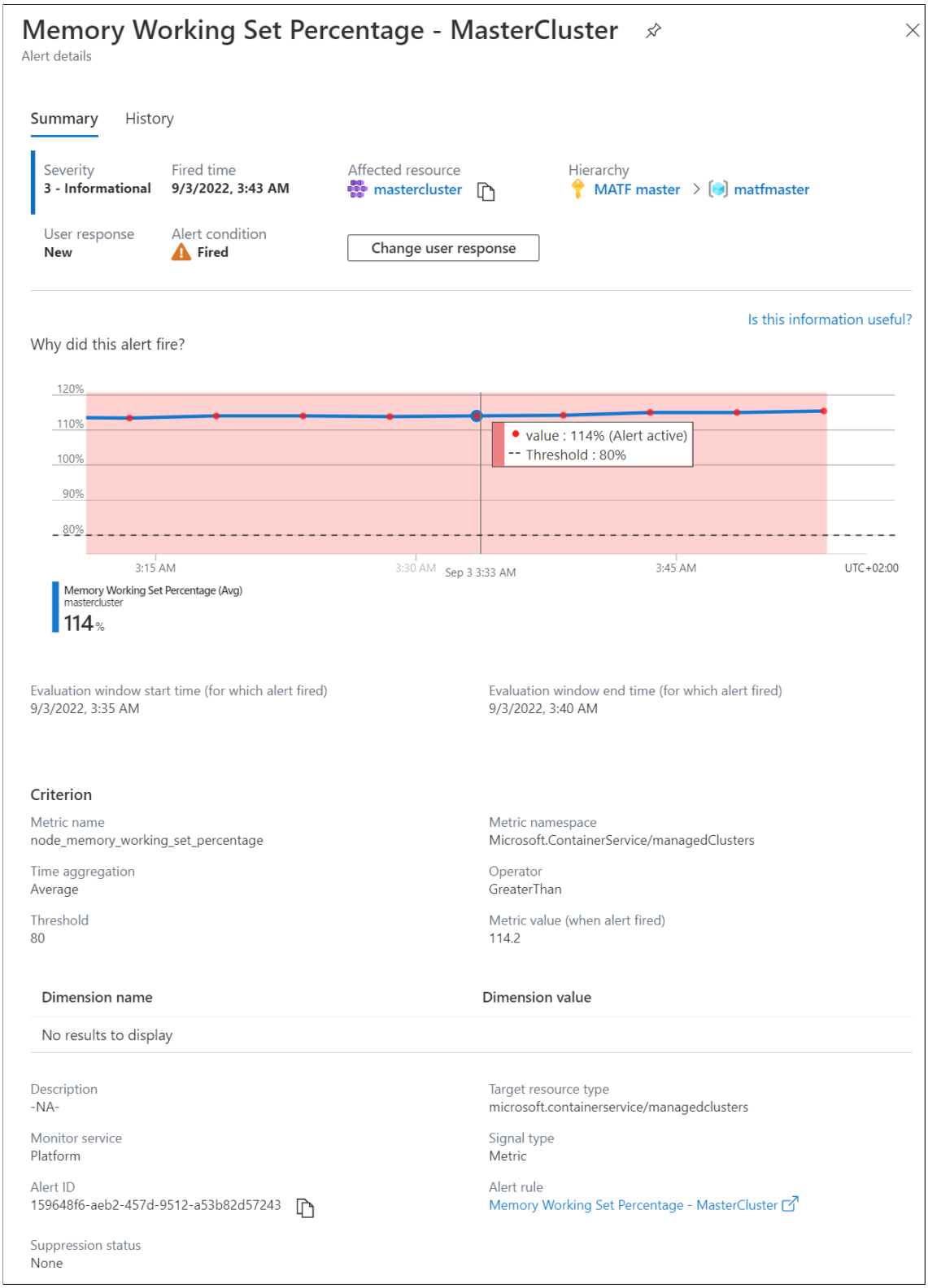
Slika 2.20: Pregled logova komponente *Compute Node*



Slika 2.21: Metrika o broju uspešnih i neuspešnih poslova kroz vreme



Slika 2.22: Aktivni uzbunjivač u slučaju neočkivanih grešaka prisutnim u logovima kontejnera



Slika 2.23: Uzbunjivač u slučaju nedovoljno memorije u klasteru

Glava 3

Rezultati

3.1 Implementirani poslovi

Prva verzija sistema DCS implementira *map-reduce* funkcionalnost nad nizom brojeva. Rezultat izračunavanja je ukupan zbir cifara brojeva od 1 do svakog elementa niza pojedinačno. Svaki element niza predstavlja jedan ulazni podatak za atomični posao. Ovaj primer veštački pravi zahtevan posao koji treba obraditi, ali je dovoljan da dokaže koncept i smislenost sistema.

Primer.

Ulazni niz: [11, 3, 1]

Izlaz: $55 = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + (1 + 0) + (1 + 1)) + (1 + 2 + 3) + (1)$

Otkazivanje zakazanih poslova nije podržano u inicijalnoj verziji.

3.2 Analiza performansi

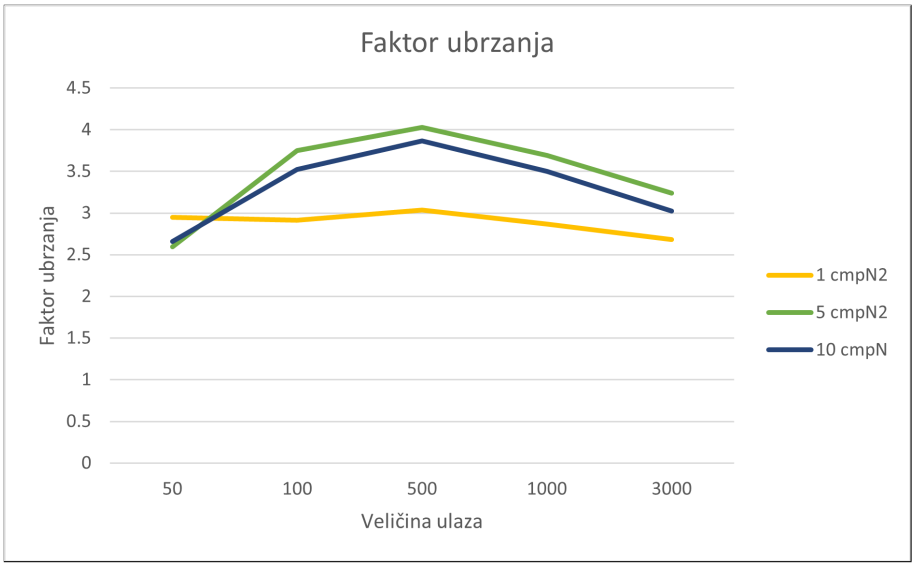
Za ispitivanje performansi sistema, izvršeno je merenje dužine trajanje obrade posla u nekoliko okruženja: sekvencijalno na jednoj mašini, nad klasterom u oblaku koji se sastoji od 1, 5, odnosno 10 jedinica za izračunavanje. Rezultati su prikazani u tabeli 3.1. Veličina ulaza predstavlja broj elemenata niza generisanih na slučajan način. Generisani brojevi se nalaze u rasponu od 400000 do 450000. Ovaj raspon je izabran empirijski, jer se nad velikim brojevima primećuje dovoljno povećanje vremena izvršavanja. Rezultati merenja su predstavljeni u sekundama, zaokruženi na drugu decimalu. Rezultati merenja so prosečna vrednost nakon tri pokretanja izračunavanja sa istim ulaznim podacima.

veličina ulaza	lokalno, sekvencijalno	1 cmpN	5 cmpN	10 cmpN
50	1.09	0.37	0.42	0.41
100	2.36	0.81	0.63	0.67
500	12.05	3.97	2.99	3.12
1000	23.6	8.23	6.39	6.74
3000	78.16	29.17	24.12	25.84

Tabela 3.1: Rezultati izvršavanja u različitim okruženjima

Iz navedenih rezultata se vidi poboljšanje vremena izvršavanja. Dodavanjem jedinica za izvršavanje, smanjuje se vreme izvršavanja. Ovo je i očekivano, s obzirom na povećanje nivoa paralelizma. Međutim, dodavanje jedinica za izvršavanje može da ubrza izvršavanje do određene granice. Iz rezultata se vidi da je vreme izvršavanja, prilikom pokretanja poslova nad sistemom sa 5, odnosno 10 jedinica za izvršavanje, veoma slično.

Na slici 3.1 se nalazi grafički prikaz faktora ubrzanja distribuiranog izvršavanja, u odnosu na sekvencijalno izvršavanje.



Slika 3.1: Faktor ubrzanja dobijen distribuiranjem

U prvoj iteraciji testiranja performansi, poboljšanje nije bilo veliko koliko bi se očekivalo, u poređenju sa lokalnim, sekvencijalnim pokretanjem algoritma. Korišćen je klaster sa ograničenim resursima. Klaster u oblaku ima ograničenja poput količine podataka i zahteva koji se propuštaju kroz mrežu, i to usporava rad sistema. Takođe, i pripadajuća baza podataka ima ograničenja potrošnje resursa i broja zahteva koje

obrađuje, te može početi da odbija zahteve koji dolaze od DCS. Kada je klasteru i bazi podataka dodeljena veća količina resursa, rezultati su se primetno poboljšali. Ovo ide u prilog skalabilnosti sistema, jer su se, kroz jednostavne konfiguracione promene, dobili bolji rezultati.

3.3 Dalji razvoj sistema

Predloženo je nekoliko ideja u kojem pravcu bi sistem DCS mogao da se razvija. Neki od pravaca daljeg razvoja su:

1. Generalizacija tipa poslova,
2. Podrška za različite frontende,
3. Napredno rutiranje zahteva ka jedinicama za izvršavanje.

Pored navedenih mogućnosti, sistem bi mogao da preusmeri pisanje rezultata na proizvoljnu lokaciju, na osnovu tipa posla. Ukoliko su ulazni podaci ili rezultat veliki, te ih nije praktično čuvati u bazi podataka, DCS je moguće proširiti kako bi koristio neko drugo skladište za pristup ulaznim podacima i za čuvanje rezultata.

Generalizacija tipa poslova

Prikazani sistem u inicijalnoj implementaciji može da obrađuje samo poslove koji se sastoje od niza atomičnih poslova. Dodavanje podrške za obradu novih tipova poslova je moguća uz sitnije izmene, čija je suština u dodavanju klase koja izvršava specifični atomični posao. Ova klasa implementira *ISpecificJobExecutor* interfejs u projektu jedinice za izvršavanje. Izbor tipa posla koji bi sledeći bio implementiran, zavisi od potražnje na tržištu.

Potpunu generalizaciju obrade atomičnih jedinica posla moguće je izvesti proširivanjem korisničkog interfejsa za zakazivanje posla, tj. proširenjem interfejsa tako da mu ulazni parametar bude izvorni kod funkcije koju je potrebno izvršiti nad atomičnim poslovima. Tada bi korisnik, pored ulaznih podataka, prosledio i deo koda koji želi da se izvrši nad atomičnim poslovima na sistemu. Za podršku ovog slučaja upotrebe, potrebno je da sistem zna da prevede izvorni kod funkcije i rezultujuću funkciju prosledi jedinici za izvršavanje. Ovaj pristup podrazumeva bezbednosne provere unetog koda, kako bi se osigurali od napada na klaster. Razvoj sistema

u ovom pravcu bi naginjao ka postojećem rešenju *Azure Functions* spomenutog u odeljku 1.2 pod rednim brojem 4.

Dalja unapređenja bi mogla da uključe obradu poslova koji nisu nužno specificirani nizom svojih atomičnih podjedinica. Ovo podrazumeva da programer razume suštinu izvršavanja novog tipa posla, te generiše plan za distribuirano izvršavanje. Kompleksni posao od korisnika bi prvo trebalo podeliti na više manjih, dok plan izvršavanja može da uključuje i agregaciju podrezultata zbog eventualnih zavisnosti potposlova. Razvoj sistema u ovom pravcu bi naginjao ga sistemima poput Sparka [24] spomenutog u odeljku 1.2.

Podrška za različite frontende

U okviru implementirane kontrolne jedinice, komponenta *Frontend* daje korisniku na korišćenje *REST API*. Dalji razvoj servisa bi mogao da podrži i druge klijent - server načine komunikacije, uključujući komunikaciju preko *Web Socketa*. Takođe, moguće je izraditi *Web* platformu i vizuelni korisnički interfejs za lakše korišćenje sistema.

Sistem je moguće proširiti, tako da uključuje podršku i za druge tipove autentifikacije. Osnovna autentikacija uključuje integrisanje sa provajderom identiteta *Azure Active Directory*. Ovo je moguće unaprediti dodavanjem odvojene komponente za podršku za druge načine autentifikacije (na primer, korišćenjem *Google* naloga).

Navedena poboljšanja bi dodatno opteretila postojeću implementaciju kontrolne jedinice. Sa daljim razvojem u ovom delu servisa, trebalo bi razdvojiti implementaciju *Frontend* servisa od orkestratora distribuiranja, i nastaviti razvoj imajući u vidu pristup razvijanja kroz mikro-servise. To bi uključilo izradu drugih *Frontend* servisa koji bi komunicirali sa svojim klijentima, a poslove slali na centralni servis zadužen za distribuirano orkestriranje.

Napredno rutiranje zahteva ka jedinicama za izvršavanje

Urađena implementacija se, prilikom izvršavanja atomičnih jedinica posla na jedinicama za izvršavanje, oslanja na Kubernetes podršku za balansiranje zahteva koji idu kroz mrežu do servisa. Drugi, samostalniji način zakazivanja poslova bi uključio prepoznavanje adresa jedinica za izvršavanje (tj. njihovih mahuna), a zatim bi tu informaciju iskoristio za proizvoljno, pametnije usmeravanje poslova na izvršavanje

na tim adresama. U kalkulacije bi uključio i procene kompleksnosti posla, kao i dostupne resurse.

Ova implementacija je zahtevnija, ali pruža priliku za bolje performanse. Međutim, pre početka same implementacije, potrebno je izvršiti opsežno testiranje i analiziranje postojućeg sistema kako bi se opravdala potreba za novim orkestratorom distribuiranog izvršavanja. Ukoliko osnovna varijanta algoritma za orkestriranje ne zadovoljava kriterijume performansi, onda ima smisla raditi ovakva unapređenja.

Ukoliko je unapređenje opravdano, za početak, potrebno je napraviti infrastrukturu za pronalaženje adresa u okviru komponente koja vrši zakazivanje poslova. Dodatno, treba implementirati algoritam koji procenjuje kompleksnost posla u vidu potrebnih resursa za njegovo izračunavanje. Na kraju, potrebno je i implementirati novi orkestrator. Nakon svega ovoga, potrebno je uraditi testiranje sistema sa novom implementacijom, kao i testiranje AB sistema kako bi se doneli zaključci koji je princip bolji i zašto.

3.4 Zaključak

U radu je predstavljen sistem za dristribuirano izračunavanje DCS. Opisani su detalji sistema, što uključuje mehanizam za primanje zahteva za obradu poslova, zakazivanje posla, distribuirano izvršavanje i praćenje poslova. Opisan je način praćenja rada sistema, kao i načini njegovog testiranja.

Nadogradnje sistema ne zahtevaju veću refaktorizaciju, jer je sistem projektovan da bude modularan, uspostavljajući svu potrebnu infrastrukturu. Ovo čini da se sistem može lako proširiti, kako bi uveo dodatne funkcionalnosti za obradu novih tipova poslova. Postojeći sistem implementira jednostavnu funkcionalnost, kako bi predstavio prednosti predloženog pristupa.

Implementirani sistem DCS pokazuje značajno ubrzanje u odnosu na izvršavanje na jednoj mašini. Pokretanjem sistema u oblaku, on je veoma skalabilan i prilagodljiv potrebama korisnika. Kako bi se dostigle najbolje perfomanse sistema za određeni tip poslova, potrebno je prilagoditi konfiguraciju sistema i njegovih komponenti izvršavanju poslova od interesa.

Postoji više pravaca u kojem je moguće dalje razvijati sistem, a za definisanje pravog pravca je potrebno izvršiti ispitivanja potreba korisnika i tržišta. Na osnovu analiza dostupnih metrika i logova, moguće je doneti odluke u koje delove sistema je potrebno ulagati kako bi se dodatno poboljšale performanse.

Bibliografija

[1] Azure Active Directory. https://azure.microsoft.com/en-gb/services/active-directory/.

[2] Azure Kubernetes Service. https://docs.microsoft.com/en-us/azure/aks/.

[3] Microsoft Azure. https://azure.microsoft.com/.

[4] Azure Monitor. https://azure.microsoft.com/en-us/services/monitor/.

[5] Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/.

[6] AzureFunctions. https://azure.microsoft.com/en-us/services/functions/.

[7] Microsoft Azure Portal. https://portal.azure.com/.

[8] BlockingCollection. https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1?view=net-6.0.

[9] Databricks. https://www.databricks.com/.

[10] Kalen Delaney, Bob Beauchemin, Conor Cunningham, Jonathan Kehayias, Benjamin Nevarez, and Paul S. Randal. *Microsoft SQL Server Internals*. O'Reilly Media, Inc, 2012.

[11] Docker. https://docs.docker.com/.

[12] Arhitektura Doker plaforme, preuzeta iz zvanične dokumentacije. https://docs.docker.com/get-started/overview/.

[13] DockerDesktop. https://www.docker.com/products/docker-desktop/.

[14] Vijay K. Garg. *Elements Of Distributed Computing.* John Wiley & Sons, Inc., 2002.

[15] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2008.

[16] Kubernetes zvanična web stranica. `https://kubernetes.io/`.

[17] Kubernetes Arhitektura. `https://kubernetes.io/docs/concepts/overview/components/`.

[18] Lets encrypt. `https://letsencrypt.org/`.

[19] Moq. `https://github.com/moq/moq`.

[20] NSwag. `https://github.com/domaindrivendev/Swashbuckle.AspNetCore`.

[21] A. Olivé. *Conceptual Modeling of Information Systems.* Springer, 2007.

[22] OpenAPI. `https://spec.openapis.org/oas/v3.1.0`.

[23] Snowflake. `https://www.snowflake.com/`.

[24] Spark. `https://spark.apache.org/`.

[25] Swagger 3.1.0. `https://swagger.io/`.

[26] Swashbuckle. `https://github.com/domaindrivendev/Swashbuckle.AspNetCore`.

[27] XUnit. `https://xunit.net/`.

Biografija autora

Milana Kovačević je rođena u Zrenjaninu, 29. novembra 1995. godine. Osnovno i srednje obrazovanje (Zrenjaninska gimnazija, prirodno-matematički smer) završila je u rodnom gradu, uz sticanje diplome „Vuk Karadžić". Takođe je završila nižu muzičku školu, instrument klavir. 2014. godine je upisala osnovne studije na modulu Informatika na Matematičkom fakultetu Univerziteta u Beogradu. Osnovne studije je završila 2017. godine sa prosečnom ocenom 9.86, kao primalac stipendije Dositeja. Master akademske studije je upisala 2017. godine, takođe na Matematičkom fakultetu na modulu Informatika. Položila je sve ispite predviđene planom i programom master akademskih studija sa prosečnom ocenom 9.15.

Nakon završetka osnovnih studija, nastavlja da se paralelno razvija i u industriji, radom u kompaniji *Microsoft*. Tokom rada se susreće sa sistemima za obradu i čuvanje podataka u okviru platforme *Azure*, a radom na jednom od njih, stiče i praktično znanje o distribuiranim sistemima i tehnologijama za rad u oblaku.