Usporedba radnih karakteristika web servisa u oblaku implementiranih u Pythonu i Rustu

Sever, Luka

Master's thesis / Diplomski rad

2025

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:968119

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-04-01**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

SVEUČILIŠTE JURJA DOBRILE U PULI
TEHNIČKI FAKULTET

Luka Sever

Usporedba radnih karakteristika web servisa u oblaku implementiranih u Pythonu i Rustu

DIPLOMSKI RAD

Pula, ožujak, 2025.

SVEUČILIŠTE JURJA DOBRILE U PULI
TEHNIČKI FAKULTET

Luka Sever

Usporedba radnih karakteristika web servisa u oblaku implementiranih u Pythonu i Rustu

DIPLOMSKI RAD

JMBAG: 0303092150, redoviti student
Studijski smjer: Računarstvo

Kolegij: Raspodijeljeni sustavi
Znanstveno područje : Tehničke znanosti
Znanstveno polje : Računarstvo
Znanstvena grana : Programsko inženjerstvo

Mentor: doc. dr. sc. Nikola Tanković

Pula, ožujak, 2025.

Sažetak

U ovom radu analizirane su performanse mrežnih servisa u oblaku implementiranih u Python-u i Rust-u kako bi se utvrdila razlika u njihovoj učinkovitosti, skalabilnosti i optimizaciji resursa. Mrežni servisi razvijeni su koristeći razvojne okvire FastAPI za Python i Actix-web za Rust te su ispitani u Amazon Web Services (AWS) okruženju, koristeći EC2 instance i DynamoDB bazu podataka. Ispitivanje je provedeno uz pomoć Apache JMeter alata, simulirajući stvarne uvjete opterećenja s ciljem mjerenja ključnih metrika kao što su latencija, vrijeme odgovora, propusnost i iskorištenost procesora.

Rezultati istraživanja pokazali su da Rust nadmašuje Python u pogledu performansi, s nižom latencijom, bržim vremenom odgovora i većom propusnošću zahtjeva. Python je, s druge strane, omogućio jednostavniji razvoj i veću fleksibilnost, ali uz značajne nedostatke radnih značajki pri većem opterećenju sustava. Zaključeno je da je Rust prikladniji za razvoj visoko-performantnih i resursno zahtjevnih aplikacija, dok je Python pogodniji za brzi razvoj i prototipiranje.

Ovo istraživanje pruža korisne uvide u odabir tehnologije za razvoj mrežnih servisa u oblaku te može poslužiti kao temelj za daljnja istraživanja u području optimizacije performansi distribuiranih sustava.

Github link: `https://github.com/i3arsu/Masters_Project`

Ključne riječi : računarstvo u oblaku, Python, Rust, web servisi, AWS

Abstract

This paper analyzes the performance of cloud-based web services implemented in Python and Rust to determine the differences in efficiency, scalability, and resource optimization. The web services were developed using FastAPI for Python and Actix-web for Rust and were tested in an Amazon Web Services (AWS) environment, utilizing EC2 instances and DynamoDB as the database. Performance testing was conducted using Apache JMeter, simulating real-world load conditions to measure key metrics such as latency, response time, throughput, and CPU utilization.

The results showed that Rust outperforms Python in terms of performance, exhibiting lower latency, faster response times, and higher request throughput. On the other hand, Python facilitated faster development and greater flexibility but suffered from significant performance drawbacks under high system load. The findings indicate that Rust is more suitable for high-performance, resource-intensive applications, while Python is better suited for rapid development and prototyping.

This research provides valuable insights into the selection of technologies for developing cloud web services and can serve as a foundation for further studies in optimizing distributed system performance.

Github link: `https://github.com/i3arsu/Masters_Project`

Keywords : cloud computing, Python, Rust, web services, AWS

Sadržaj

1 Uvod **1**
1.1 Motivacija istraživanja . . . 2
1.2 Servisi u Oblaku (*eng. Cloud Services*) . . . 2
1.3 Programski jezici . . . 6

2 Metodologija istraživanja **8**
2.1 Testno okruženje . . . 8
2.2 Baza podataka . . . 8
2.3 Model sustava . . . 9
2.4 Mjerenje latencije . . . 11

3 Implementacija rješenja **12**
3.1 Python . . . 12
3.2 Rust . . . 21
3.3 Infrastruktura . . . 31
3.4 Ispitivanje performansi (Jmeter) . . . 34

4 Rezultati **36**

5 Rasprava **41**

6 Zaključak **43**

Literatura **44**

Popis slika **46**

Popis tablica **47**

1 Uvod

U današnje vrijeme sve više trgovina nudi mogućnost digitalne kupovine pomoću vlastitih web trgovina. S ciljem privlačenja što većeg broja kupaca, implementiraju se mogućnosti iskorištavanja kupona za dodatna sniženja već postojećih cijena. No, razvoj opisanih web aplikacija nosi niz izazova, među kojima su ključni odabir programskog jezika i arhitekture sustava. Performanse i skalabilnost web servisa igraju ključnu ulogu u osiguravanju optimalnog korisničkog iskustva, osobito u uvjetima visokog opterećenja.

S obzirom na rastuću potrebu za brzim i efikasnim sustavima, odabir tehnologije za implementaciju web servisa ima značajan utjecaj na njegovu pouzdanost i odaziv. Za izgradnju aplikacijskog programskog sučelja (**API**, *eng. application programming interface*) često se koriste dva vrlo popularna programska jezika, Python i Rust. Python je široko prihvaćen zbog svoje jednostavnosti, fleksibilnosti i bogatog ekosustava biblioteka. Za razliku od prethodno spomenutog Python-a, Rust se ističe svojom visokom učinkovitošću, sigurnošću memorije i optimiziranim upravljanjem resursima.

Cilj ovog rada analizirati je i usporediti performanse web servisa implementiranih u Python-u i Rust-u, koristeći FastAPI i Actix-web razvojni okvir. Analiza performansi temelji se na ključnim metrikama, kao što su vrijeme odaziva, latencija, propusnost i iskorištenost procesora, uz dodatnu primjenu alata za ispitivanje opterećenja, poput Apache JMeter alata. U okviru istraživanja razvijen je API za obradu narudžbi i primjenu kupona na proizvode u košarici web trgovine, implementiran u oba spomenuta programska jezika. U sklopu analize, API je ispitan koristeći Apache JMeter alat, kako bi se s visokom pouzdanošću izmjerile radne značajke pod različitim razinama opterećenja.

Rad također daje uvid u performanse Python i Rust implementacija te pruža smjernice za što lakši odabir tehnologije ovisno o specifičnim zahtjevima sustava. Rezultati istraživanja mogu poslužiti kao svojevrsni temelj budućim optimizacijama i donošenju informiranih odluka prilikom razvoja web servisa visokih radnih značajki.

Ovim istraživanjem nastoji se pružiti jasna preporuka o tome koji je jezik prikladniji za razvoj specifičnih servisa u oblaku, uzimajući u obzir tehničke radne značajke, troškove razvoja i daljnje održavanje. Rezultati mogu poslužiti kao smjernice za odabir tehnologija prilikom dizajna i implementacije daljnjih servisa, kao i novih aplikacija u oblaku.

Izvorni kôd projekta : `https://github.com/i3arsu/Masters_Project`

1.1 Motivacija istraživanja

Motivacija za ovo istraživanje proizlazi iz velikog oslanjanja na tehnologije u oblaku i distribuirane sustave u razvoju modernih aplikacija. Jezici Python i Rust posebice su zanimljivi jer predstavljaju dva različita pristupa razvoju:

- Python je poznat po svojoj jednostavnosti i fleksibilnosti, često se koristi za brzi razvoj prototipa i aplikacija, no ponekad pati od nedostatka radnih značajki
- Rust pruža vrhunsku učinkovitost i sigurnost, ali zahtijeva ulaganje više vremena i truda u razvoj što proporcionalno može imati utjecaj na vremenski okvir implementacije

Razvoj modernih web servisa zahtijeva ravnotežu između radnih značajki, skalabilnosti i jednostavnosti implementacije. U poslovnom okruženju, posebice u razvoju internet trgovina, brzina obrade zahtjeva može izravno utjecati na zadovoljstvo korisnika, a time i na poslovne rezultate. Spora ili neefikasna web aplikacija može dovesti do gubitka budućih i postojećih kupaca, dok optimizirani sustavi, koji omogućuju bolje korisničko iskustvo i veću konverziju, imaju gotovo suprotan učinak.

Jedan od ključnih izazova u izradi web servisa je odabir tehnologije koja neće „pokleknuti" pod velikim opterećenjima, posebice u situacijama u kojima se od sustava očekuje obrada značajnog broja istovremenih zahtjeva. Python se već dugo koristi kao standard u razvoju web aplikacija zbog svoje jednostavnosti i bogatog ekosustava, ali se često dovodi u pitanje njegova efikasnost u usporedbi s novijim, efikasnijim jezicima, poput Rust-a. Rust nudi visoke performanse i sigurnost memorije, no njegovo usvajanje u web razvoju još uvijek nije u potpunosti istraženo.

1.2 Servisi u Oblaku (*eng. Cloud Services*)

Servisi u oblaku (*eng. Cloud Services*) predstavljaju aplikacije, platforme i računalne resurse koji su dostupni za korištenje korisnicima putem interneta [1]. Razvoj web servisa u oblaku postaje ključan za modernu tehnologiju, jer omogućuje skalabilnost, visoku dostupnost i efikasno upravljanje resursima. Oblačne tehnologije danas podržavaju širok raspon aplikacija, od malih start-up projekata do globalno popularnih platformi poput Netflix-a, Amazon-a i Microsoft Azure-a.

Koncept računarstva u oblaku spominje se prvi put u 60-tim godinama prošlog stoljeća s idejom dijeljenja računalnih resursa kao javnih usluga, čime se namjeravala postići optimizacija korištenja skupe računalne opreme [2]. John McCarthy (1927. - 2011.), svojevrsni pionir umjetne inteligencije, predviđa da će u budućnosti računarstvo postati široko dostupno, poput javne usluge slične električnoj energiji. Međutim, tadašnja tehnologija bila je nedovoljno razvijena za realizaciju njegova koncepta. Prvi pravi temelj za razvoj tehnologija u oblaku postavljen je u 90-tim godinama prošlog stoljeća, kada dolazi do ubrzanog razvoja internetske infrastrukture [2].

U navedenom periodu tvrtke se počinju koristiti centraliziranim serverima za pružanje aplikacija i usluga putem interneta što popratno dovodi do pojave prvih oblika računarstva u oblaku. Prva značajna komercijalna usluga na oblaku je Salesforce (1999.) koja omogućava softver za odnos s klijentima (**CRM**, *eng. Customer Relationship Management*) dostupan isključivo putem interneta, uklanjajući potrebu za lokalnim instalacijama [3]

Tijekom 2000-tih godina dolazi do procvata tehnologija oblaka uslijed prepoznavanja potencijala pružanja računalnih resursa putem interneta [4] velikih tehnoloških kompanija poput Amazon-a, Google-a i Microsoft-a. Podružnica tvrtke Amazon, Amazon Web Services (**AWS**), 2006. godine pokreće svoju prvu infrastrukturu kao uslugu dostupnu svojim klijentima, omogućujući korisnicima pristup serverima i resursima na zahtjev. Tvrtke Google i Microsoft ubrzo slijede AWS svojim vlastitim rješenjima poput servisa Google Cloud i Microsoft Azure, čime se tržište usluga u oblaku značajno proširuje [5].

Servisi u oblaku danas posjeduju ključnu ulogu u digitalnoj industriji, omogućujući velikim i malim tvrtkama, kao i privatnim i poslovnim korisnicima skalabilnost, fleksibilnost i znatnu uštedu troškova i resursa. Računarstvo u oblaku dijeli se u nekoliko glavnih modela [6]:

- Infrastruktura kao usluga (**IaaS, *eng. infrastructure as a service***) - pruža osnovne računalne resurse poput virtualnih servera, pohrane podataka i mrežne infrastrukture (pr. AWS)

- Platforma kao usluga (**PaaS**, *eng. platform as a service*) - omogućuje razvoj i upravljanje aplikacijama bez potrebe za održavanjem infrastrukture (pr. Google App Engine, AWS Elastic Beanstalk i Heroku)

- Softver kao usluga (***SaaS***, *eng. software as a service*) - gotove aplikacije dostupne putem interneta (pr. Google Workspace i Office 365)

1.2.1 Mrežni servisi - Amazon Web Services (AWS)

Tvrtka Amazon Web Services (**AWS**) predstavlja jednog od glavnih pružatelja usluge servisa u oblaku s preko više od 200 različitih usluga koje pokrivaju širok raspon potreba kao što su, primjerice, pohrana podataka, računalni resursi i strojno učenje. Njihova velika popularnost temelji se na skalabilnosti, fleksibilnosti i širokoj dostupnosti [7].

Glavna prednost mrežnih servisa AWS-a naspram sličnim servisima je njegova skalabilnost koja omogućuje korisnicima povećavanje i smanjivanje resursa sustava ovisno o potrebama. Smanjenjem dostupnih resursa ujedno se postiže i smanjenje nepotrebnih troškova. AWS koristi model naplate „*pay-as-you-go*“ što znači da korisnici plaćaju isključivo resurse koje trenutno koriste, bez potrebe za vlastitom infrastrukturom.

Tvrtka AWS posjeduje visoko povezanu globalnu mrežu podatkovnih centara na više kontinenata što korisnicima omogućuje brži prijenos podataka i smanjenje latencije na globalnoj skali. Ujedno, njihovi mrežni servisi posjeduju značajnu razinu sigurnosti i pouzdanosti [7].

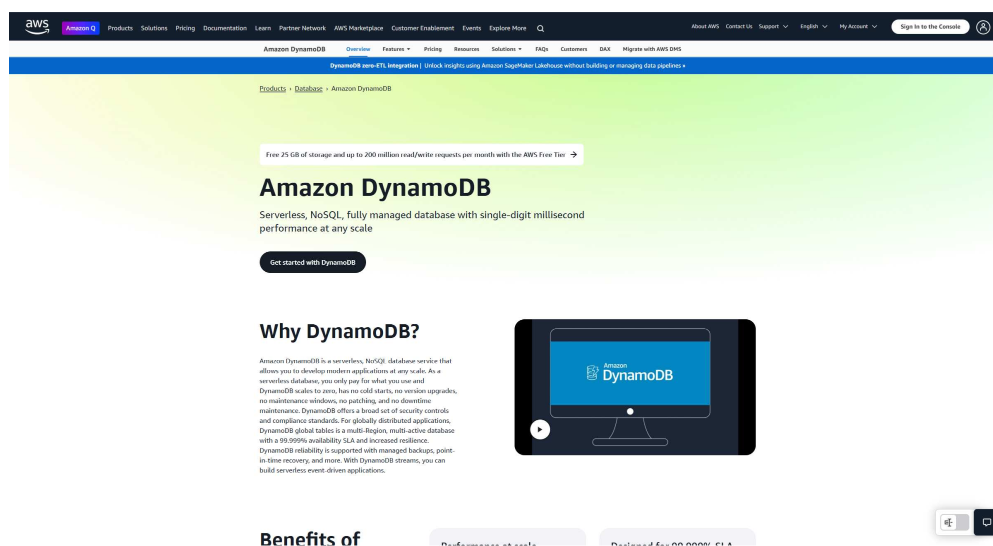
U ovom istraživanju mrežni servisi tvrtke AWS odabrani su kao platforma za implementaciju i ispitivanje servisa u oblaku zbog fleksibilne infrastrukture i niza usluga koje znatno olakšavaju razvoj i evaluaciju. Ključne usluge koje se koriste u ovom radu su usluga Elastic Compute Cloud (EC2) i DynamoDB tvrtke AWS. Infrastruktura AWS-ovih mrežnih servisa omogućava simulaciju stvarnih scenarija što je presudno za evaluaciju radnih značajki i usporedbu tehnologija.

1.2.2 DynamoDB

DynamoDB je visoko skalabilna i distribuirana NoSQL baza podataka koju pruža tvrtka AWS. Dizajnirana je za aplikacije koje zahtijevaju nisku latenciju i visoku dostupnost što je čini idealnim rješenjem za mrežne servise i druge sustave kojima je glavni cilj brz i pouzdan pristup podacima. Baza koristi model ključa i atributa omogućujući jednostavno spremanje podataka bez strogog nacrta.

Glavna prednost DynamoDB baze podataka je njegova skalabilnost. Baza se automatski prilagođava potrebama aplikacije tako što smanjuje ili povećava kapacitet u skladu s opterećenjem. Navedeni način rada omogućuje kontinuirano posjedovanje visokih radnih značajki, čak i tijekom naglog povećanja upita prema bazi. DynamoDB nudi mogućnost složenih upita kroz sekundarne indekse što direktno djeluje na efikasnost pretraživanja podataka izvan primarnog ključa. Dodatno, DynamoDB podržava replikaciju podataka između AWS regija, čime se dodatno osiguravaju podaci u slučaju iznenadnih kvarova [8].

U ovom istraživanju DynamoDB baza podataka odabrana je za obje implementacije web servisa u oblaku zbog svoje jednostavne integracije s infrastrukturom AWS mrežnih servisa i podrške za veliku količinu podataka uz nisku latenciju. Zbog svojih radnih značajki i visoke pouzdanosti, DynamoDB omogućuje simuliranje stvarnog opterećenja uz osiguravanje relevantnosti dobivenih rezultata. Na slici 1. prikazana je početna internetska stranica pružatelja usluge DynamoDB baze podataka.



Slika 1: Početna stranica DynamoDB servisa (izvor: https://aws.amazon.com/dynamodb/)

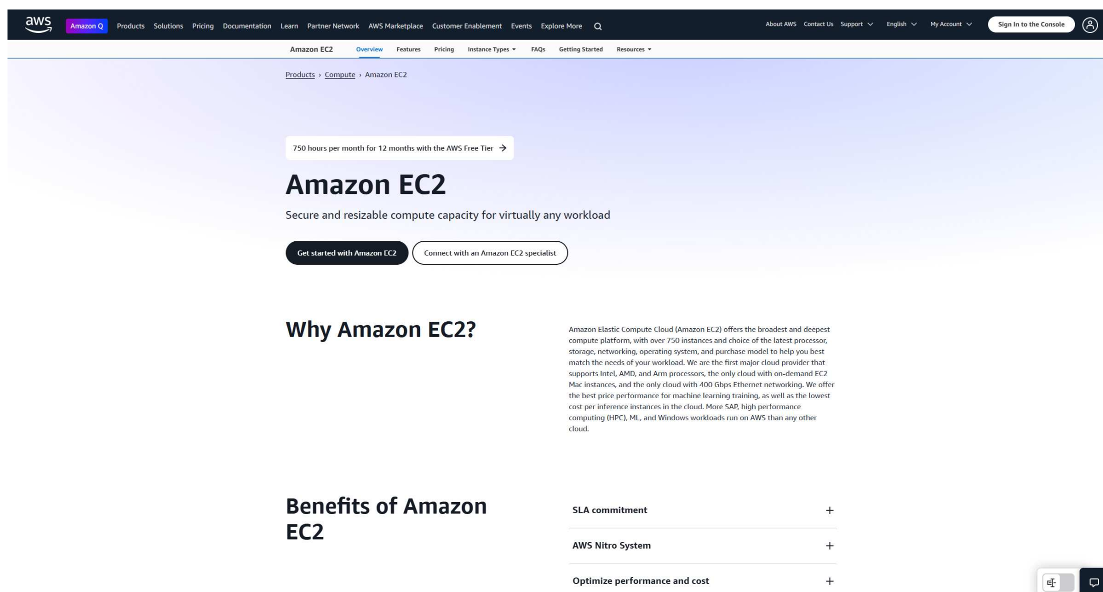
1.2.3 Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) ključna je usluga koju pruža AWS. Svojim korisnicima pruža skalabilne računalne resurse u oblaku. Usluga EC2 dizajnirana je kako bi se omogućilo brzo pokretanje instanci (virtualnih poslužitelja) čime se eliminira potreba za fizičkim hardverom i posebnom infrastrukturom. Tvrtka AWS omogućava korisnicima odabir između različitih vrsta i veličina instanci koje se prilagođavaju specifičnim potrebama aplikacije.

Usluga EC2 podržava širok raspon operacijskih sustava od popularnih distribucija poput Linux Ubuntu sustava, Amazon Linux i CentOS-a pa sve do Windows Server-a. [9] Svaka instanca smještena je unutar privatnog virtualnog oblaka (**VPC**, *eng. Amazon Virtual Private Cloud*) što omogućuje visoku razinu kontrole sigurnosti, kao i same mrežne infrastrukture. Korisnicima se pruža mogućnost definiranja mrežnog prometa pomoću sigurnosnih grupa, kao i ograničavanje pristupa specifičnim IP adresama. Ujedno, usluga EC2 omogućava jednostavno povezivanje s drugim uslugama AWS-a, poput S3 i DynamoDB.

Glavna značajka usluge EC2 je sposobnost skaliranja kapaciteta prema potrebama aplikacija. Horizontalno skaliranje omogućuje dodavanje više instanci radi povećanja računalne moći, dok vertikalno skaliranje omogućuje povećanje resursa pojedine instance. [10]

U ovom istraživanju usluga Amazon EC2 koristi se za implementaciju i pokretanje web servisa razvijenih u Python-u i Rust-u. Svaki servis se pokreće na vlastitoj instanci s operacijskim sustavom Ubuntu, odabranim zbog svoje pouzdanosti i kompatibilnosti. Na slici 2. prikazana je početna internetska stranica pružatelja usluge EC2.



Slika 2: Početna stranica EC2 servisa (izvor: https://aws.amazon.com/ec2/)

1.3 Programski jezici

U navedenom poglavlju iznijeti će se usporedbe između odabranih programskih jezika (Python i Rust) te objasniti njihova upotreba u kontekstu rada.

1.3.1 Python

Python je programski jezik poznat po svojoj jednostavnosti, čitljivosti i širokoj primjeni u raznim područjima. Razvio ga je 1991. godine Guido van Rossum (1956. -) [11] nakon čega Python ubrzo postaje jednim od najpopularnijih programskih jezika. Filozofija Python-a temelji se na konceptima jednostavnog, eksplicitnog i intuitivnog kôda što ga čini idealnim za brzi razvoj aplikacija.

Python podržava više stilova programiranja, uključujući proceduralno, objektno-orijentirano i funkcionalno programiranje. Glavni razlog popularnosti je veoma opsežna biblioteka (*eng. library*) koja pruža širok asortiman funkcija i metoda za rad s datotekama, mrežnom komunikacijom, manipulacijom podataka i drugim zadacima. Dodatno, Python nudi bogatu podršku za razvoj umjetne inteligencije i web servisa implementacijom biblioteka.

U kontekstu razvoja web aplikacija i servisa, Python nudi veliku raznolikost razvojnih okvira (*eng. framework*), kao što su Flask, FastAPI i Django. Razvojni okviri Flask i Django često se koriste za razvoj kompleksnih aplikacija, dok je noviji FastAPI optimiziraniji za moderne „RESTful“ API-je. RESTful API (*eng. Representational State Transfer Application Programming Interface*) je arhitekturni stil za izgradnju mrežnih servisa koji omogućuje komunikaciju između klijenta i poslužitelja putem standardnih HTTP metoda poput GET, POST, PUT i DELETE. [12]

U ovom istraživanju Python se koristi za implementaciju web servisa zbog dostupnosti alata poput FastAPI-a i zbog same popularnosti jezika [13]. Servis se koristi za interakciju s DynamoDB bazom podataka i za odgovor na zahtjev klijenata putem HTTP protokola. Iako je Python poznat po višoj latenciji u usporedbi s jezicima koji zahtijevaju kompilaciju, njegova fleksibilnost omogućuje bržu implementaciju i ispitivanje servisa u kontekstu rada.

1.3.2 Rust

Rust je suvremeni programski jezik razvijen 2010. godine od tvrtke Mozilla Research s ciljem pružanja visokih radnih značajki i sigurnosti u programiranju [14]. Rust se ističe po jedinstvenom sustavu upravljanja pohranom, kao i po mogućnosti izrade aplikacija bez uobičajenih problema poput curenja memorije ili uvjetnih utrka (*eng. race conditions*). Sve navedeno čini ga idealnim jezikom za razvoj softverskih rješenja gdje pouzdanost i brzina igraju ključnu ulogu.

Rust se svrstava u kategoriju jezika sustava (*eng. system programming language*), slično jezicima C i C++, koji od razvojnog programera očekuju nisku razinu kontrole nad hardverom, pri čemu zauzvrat pružaju dodatne sigurnosne značajke. Rust-ovu ključnu značajku čini rad na načelu vlasništva i posjedovanja memorije (*eng. ownership model*) umjesto načela skupljanja smeća (*eng. garbage collection*).

Prema navedenom načelu omogućena je rigorozna kontrola posjedovanja i upravljanja podacima u danom trenutku. [14]. Rust, kao i Python, podržava mnoštvo paradigmi programiranja što ga čini svestranim jezikom primjenjivim u svakoj vrsti projekta. Rustov kompilator nudi detaljne informacije o greškama i upozorenjima što doprinosi kvaliteti pisanja kôda omogućujući rano otkrivanje i izbjegavanje grešaka.

U kontekstu razvoja web aplikacija, Rust se sve više upotrebljava zbog svoje iznimne brzine i sigurnosti [15]. Popularni razvojni okviri poput Actix-a i Rocket-a dodatno olakšavaju razvoj RESTful mrežnih servisa. Razvojni okvir Actix posebno se ističe kao jedan od najbržih i najkorištenijih okvira za izradu web aplikacija zahvaljujući svojoj mogućnosti da obrađuje upite asinkrono. Rust kao jezik ima strmu krivulju učenja zbog svog strogo tipiziranog sustava i vlasničkog modela upravljanja memorijom, no njegova velika prednost leži u činjenici da korisniku nudi veliku sigurnost i napredne performanse što ga čini pouzdanim izborom za razvoj profesionalnih aplikacija.

U ovom istraživanju Rust se koristi za implementaciju web servisa koji komuniciraju s DynamoDB bazom podataka putem RESTful API-ja. Kao razvojni okvir koristi se Actix.

2 Metodologija istraživanja

Kako bi se dobio jasan uvid u radne značajke web servisa implementiranih u Python-u i Rust-u, provedeno je istraživanje temeljeno na mjerenju ključnih metrika rada sustava.

2.1 Testno okruženje

U svrhu osiguranja dosljednosti i reproducibilnosti rezultata, aplikacijska programska sučelja pokrenuta su na identičnim AWS EC2 instancama s istovrsnim specifikacijama procesora i memorije prikazanim u tablici 1.

Ime instance	Ime arhitekture	vCPUs	RAM (GiB)
python-webserver	t2.micro	1	1
rust-webserver	t2.micro	1	1

Tablica 1: Tablični prikaz instanci i njihovih specifikacija (izvor: Autor)

Tablica 2. prikazuje sve dostupne T2 instance i njihove radne značajke. T2 instanca je odabrana zbog svoje dostupnosti u besplatnom planu rada AWS računa [16].

Instance	vCPU	CPU Credits / Sat	Memorija	Pohrana	Performanse mreže
t2.nano	1	3	0.5	EBS-Only	Niske
t2.micro	1	6	1	EBS-Only	Niske do Srednje
t2.small	1	12	2	EBS-Only	Niske do Srednje
t2.medium	2	24	4	EBS-Only	Niske do Srednje
t2.large	2	36	8	EBS-Only	Niske do Srednje
t2.xlarge	4	54	16	EBS-Only	Srednje
t2.2xlarge	8	81	32	EBS-Only	Srednje

Tablica 2: Prikaz svih vrsta T2 instanci i njihovih radnih značajki (izvor: https://aws.amazon.com/ec2/instance-types/)

Ispitivanja su provođena u periodima od 45 minuta s 500 simuliranih korisnika. Tijekom ispitivanja sustavi su bili pod konstantnim opterećenjem.

2.2 Baza podataka

Svaka tablica unutar DynamoDB baze podataka mora imati definiran primarni ključ koji omogućava brzo dohvaćanje podataka:

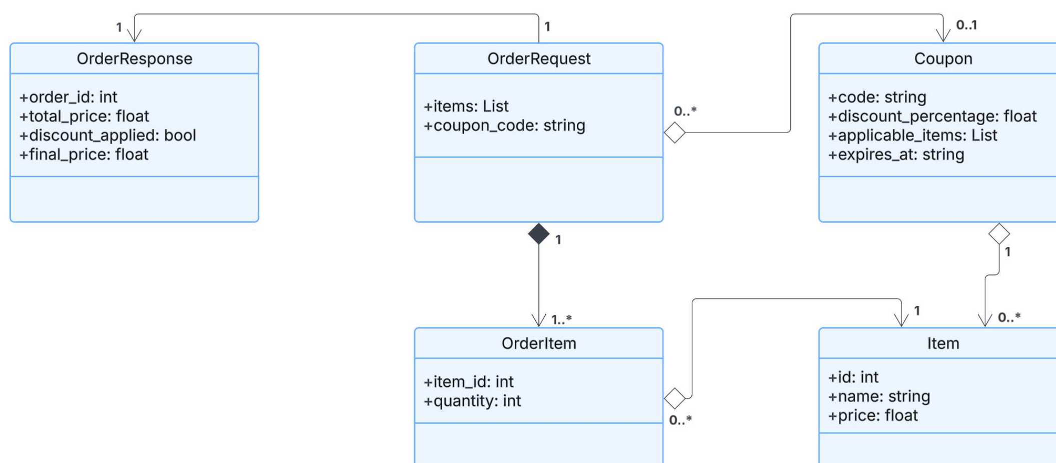
- Jednostavni ključ (*eng. partition key*) sastoji se samo od *hash* ključa. Vrijednost ključa koristi se za određivanje particije na kojoj će podatci biti pohranjeni.
- Složeni ključ (*eng. partition key + sort key*) kombinira *hash* ključ i *sort* ključ što omogućava lakšu organizaciju podataka unutar iste particije prema redoslijedu prethodno definiranom *sort* ključem.

Hash ključ koristi *hash* funkciju kako bi podatke rasporedio u različite particije, pri tome pazeći da se vrijednosti ravnomjerno distribuiraju kako bi se izbjeglo preopterećenje određenih čvorova. *Sort* ključ omogućuje sekvencijsko pretraživanje prilikom kojeg se svi podaci s istim jednostavnim ključem mogu sortirano dohvatiti.

Također osim primarnog ključa, DynamoDB omogućuje i definiranje sekundarnih indeksa radi bržeg pretraživanja. Globalni sekundarni indeks (**GSI**, *eng. global secondary index*) može koristiti i drukčiji *partition* i *sort* ključ od primarne tablice, osiguravajući alternativne načine pretraživanja. Lokalni sekundarni indeks (**LSI**, *eng. local secondary index*) dijeli isti *partition* ključ kao i glavna tablica, ali koristi drukčiji *sort* ključ čime omogućuje alternativni pogled na isti skup podataka.

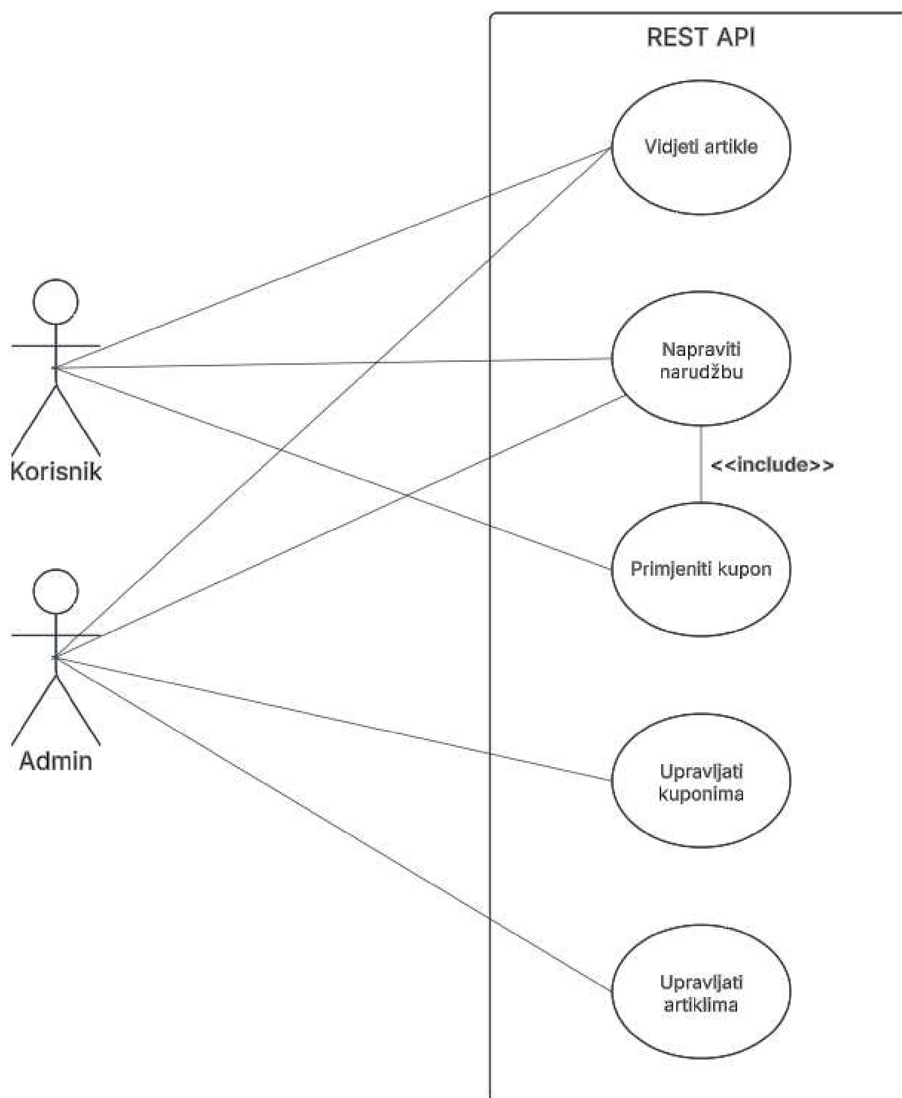
2.3 Model sustava

Podaci su modelirani identično za oba servisa kako bi se osigurala dosljednost podataka. Na slici 3. prikazan je grafički prikaz jedinstvenog jezika za modeliranje (**UML**, *eng. unified modeling language*) koji prikazuje sve modele podataka i njihove međusobne odnose.



Slika 3: UML Dijagram modela podataka (izvor: Autor)

Prethodno navedeni grafički prikaz predstavlja osnovni model podataka internet trgovine s podrškom za sniženja uz pomoć kupona koji je implementiran u Rust i Python servisu. Za implementaciju u Python servisu korištena je biblioteka `Pydantic` koja omogućuje deklarativnu validaciju podataka te ujedno podržava i konverziju tipova. Za implementaciju u Rust servisu korištena je biblioteka `Serde` koja omogućuje statički tipizirane strukture, ali bez popratne automatske konverzije tipova.



Slika 4: Dijagram korištenja sustava (UML Use case Dijagram) (izvor: Autor)

Graf na slici 4 prikazuje rad sustava kao i njegovo korištenje od strane korisnika. Sustav je osmišljen na način da kupac može vidjeti dostupne artikle, postaviti narudžbu i primijeniti kupon na narudžbu, no ne može upravljati artiklima i kuponima. Za razliku od kupca, administrator ima pristup svim funkcionalnostima sustava.

2.4 Mjerenje latencije

Mjerenje latencije (kašnjenje) sustava ključno je za evaluaciju performansi i osiguranje optimalnog korisničkog iskustva, osobito u sustavima koji zahtijevaju brzi odaziv i visoku dostupnost. Latencija se definira kao vrijeme proteklo od trenutka slanja zahtjeva prema API-ju do trenutka primitka odgovora i predstavlja jedan od ključnih pokazatelja učinkovitosti sustava. Cilj mjerenja latencije u kontekstu rada je kvantificirati razlike u performansama između aplikacija razvijenih u Python-u i Rust-u te procijeniti njihovu skalabilnost i stabilnost pod različitim razinama opterećenja.

U svrhu mjerenja latencije korišten je Apache JMeter, popularan alat za ispitivanje performansi mrežnih servisa. Alat JMeter omogućuje simulaciju prometa koji oponaša stvarne uvjete rada. On, nadalje, podržava raznovrsne zahtjeve, uključujući GET i POST metode te omogućuje prilagodbu parametara poput intenziteta opterećenja, broja paralelnih korisnika i složenosti zahtjeva. Tijekom ispitivanja simulirani su različiti scenariji opterećenja kako bi se ispitalo ponašanje sustava pri povećanom broju korisnika i zahtjeva [17]. U test su uključene dvije vrste prometa:

- GET zahtjevi: dohvaćanje podataka o artiklima čime se ispitivala brzina čitanja iz baze podataka i sposobnost API-ja da odgovori na postavljene zahtjeve sa što manjim mogućim kašnjenjem
- POST zahtjevi: primjena kupona na narudžbu prilikom koje su ispitani složeniji procesi poput validacije kupona i izračuna ukupnog sniženja cijene

Promet je prilagođen stvarnim scenarijima u kontekstu internet trgovine:

- Veličina narudžbe: stvoreni su zahtjevi s različitim brojem artikala i količinama kako bi se oponašale male i velike narudžbe
- Primjena kupona: korišteni su nasumično odabrani kuponi za simulaciju stvarnih situacija u kojima korisnici pokušavaju iskoristiti popuste
- Varijabilno opterećenje: ispitivanja su uključivala statičko opterećenje s konstantnim brojem korisnika te dinamičko opterećenje koje simulira nagle poraste prometa

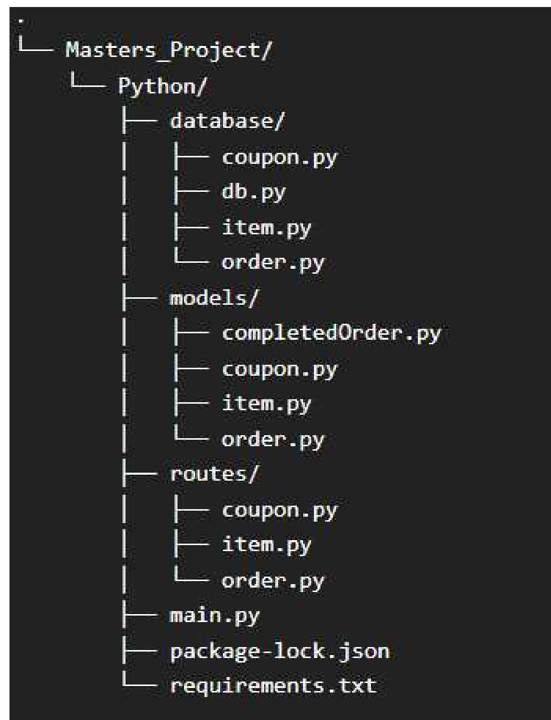
Cilj provedenog testiranja bio je ispitati sljedeće značajke:

- Latenciju pri opterećenju: analiza proteklog vremena do odgovora na zahtjeve pod određenim opterećenjem
- Stabilnost sustava: praćenje dosljednosti vremena i odgovora tijekom duljih vremenskih razdoblja
- Skalabilnost: utvrditi u kojem kapacitetu sustavi razvijeni u Python-u i Rust-u reagiraju na povećanje broja zahtjeva.

Navedenim pristupom omogućeno je realistično ispitivanje radnih značajki sustava upravo zbog korištenja scenarija usklađenih s tipičnim obrascima ponašanja korisnika.

3 Implementacija rješenja

Kako bi se izbjeglo traženje kôda prilikom definiranja strukture projekta preporučuje se organiziranje datoteka. Projektna struktura, prikazana na slici 5., organizirana je tako da je u svakom trenutku osigurano logičko razdvajanje modela, ruta i povezivanja s bazom podataka.



Slika 5: Struktura projekta (izvor: Autor)

3.1 Python

Sljedeće poglavlje opisuje implementaciju API-ja u programskom jeziku Python koristeći se FastAPI razvojnim okvirom. Prilikom kreiranja aplikacije veliki značaj pridodan je samoj jednostavnosti razvoja, čitljivosti kôda i fleksibilnosti uz podršku za automatsku dokumentaciju i asinkrono izvršavanje.

3.1.1 Postavljanje virtualnog okruženja

Za implementaciju servisa u Python-u postavljeno je razvojno okruženje pomoću Python-ovog virtualnog okruženja. Komanda za stvaranje okruženja u kojoj parametar `myenv` predstavlja ime:

```
python3 -m venv myenv
```

Uz pomoć alata pip dohvaćene su potrebne biblioteke poput FastAPI, uvicorn, aiodynamo i pydantic.

3.1.2 Stvaranje glavnog dokumenta za pokretanje servisa

Kako bi se servis mogao nesmetano pokrenuti, potrebno je stvoriti glavni dokument. Na sljedećem kôdu prikazan je glavni dokument servisa unutar kojeg su definirane glavne varijable i postavke samog servisa.

```
import uvicorn
from fastapi import FastAPI
from pathlib import Path
from mangum import Mangum
from routes.item import item_router
from routes.order import order_router
from routes.coupon import coupon_router
from database.db import DynamoDBClientManager


app = FastAPI()
handler = Mangum(app)

app.include_router(item_router, prefix="/item")
app.include_router(coupon_router, prefix="/coupon")
app.include_router(order_router, prefix="/order")

@app.on_event("shutdown")
async def closeClient():
    await DynamoDBClientManager.close_client()

if __name__ == "__main__":
    uvicorn.run(f"{Path(__file__).stem}:app", host="0.0.0.0", port
   =8000, env_file=".env")
```

Varijabla `app` stvara instancu poslužitelja pomoću funkcije `FastAPI()`. Varijabla `handler`, koristeći se bibliotekom Mangum kako bi FastAPI aplikacija mogla raditi na AWS Lambda platformi, omogućuje rad servera bez ručnog upravljanja (*eng. serverless architecture*). Kako bi servis pravilno preusmjerio korisnika na ispravnu rutu, potrebno je definirati *router*. Naredba `app.include_router()` omogućuje korištenje ruta definiranih u drugim datotekama koje sadrže logiku za specifične funkcionalnosti. Parametar `prefix` određuje osnovni put (URL) za svaku rutu. Kako bi se servis pokrenuo pri pozivanju glavne datoteke, potrebno je izvršiti provjeru izravnog pokretanja datoteke. Ako je provjera istinita, izvršava se naredba `uvicorn.run()` koja pokreće FastAPI aplikaciju i prima određene parametre. Prvi parametar dinamički određuje put do aplikacije, drugi parametar omogućuje vidljivost aplikacije na svim mrežnim sučeljima, treći parametar određuje na kojem će se portu aplikacija pokrenuti i zadnji parametar koji daje uputu aplikaciji da učita konfiguracijske varijable iz datoteke pod imenom ".env".

3.1.3 Dodavanje osnovnih ruta

Nakon osnovne konfiguracije poslužitelja potrebno je postaviti rute koje će odgovarati na dobivene upite i zahtjeve. Aplikacija je organizirana prema funkcionalnim cjelinama, pri čemu se logika i rute odvajaju u zasebne module. Osnovna aplikacija koristi glavnu datoteku za registraciju ruta.

Dodavanje ruta objasnit će se na modulu `routes/item`. U modulu `routes/item` definirane su sve rute povezane s artiklima koje omogućuju operacije poput stvaranja novih artikala, dohvaćanja artikla i brisanja artikla.

```
from fastapi import APIRouter
from database.item import create_item, get_item, get_items,
    remove_item
from models.item import Item

item_router = APIRouter()

@item_router.post("/create", response_model=Item)
async def create(item: Item):
    response = await create_item(item.model_dump())
    return response

@item_router.get("/id/{id}")
async def get_one_item(id: str):
    response = await get_item(id)
    return response

@item_router.get("/all")
async def get_all_items():
    response = await get_items()
    return response

@item_router.delete("/remove/{id}")
async def delete_item(id: str):
    response = await remove_item(id)
    return response
```

Ruta `/create`, tipa POST, omogućuje stvaranje novog artikla. Korisnik šalje JSON podatke o artiklu, a funkcija `create_item` , iz modula `database/item`, obrađuje zahtjev. Ako je artikl uspješno stvoren, vraća se odgovor sa statusnim kôdom 200 (OK) i pripadajućom porukom. U slučaju greške, vraća se odgovarajući kôd greške s opisom. Za validaciju podataka funkcija koristi strukturu `Item`.

Ruta `/id/{id}`, tipa GET, koristi funkciju `get_item` iz modula `database/item` , koja dohvaća artikl na temelju kôda. Ako je artikl pronađen, vraća se odgovor sa statusnim kôdom 200 (OK) i detaljima artikla u JSON formatu. Ako artikl nije pronađen, vraća se odgovor sa statusnim kôdom 404 (Not Found).

Ruta `/all`, tipa GET, koristi funkciju `get_all_items` iz modula `database/item`, koja dohvaća sve artikle iz baze podataka. Rezultat se vraća kao JSON popis artikala sa statusnim kôdom 200 (OK). U slučaju greške vraća se odgovarajuća poruka sa statusnim kôdom.

Ruta `/remove/{id}`, tipa DELETE, koristi funkciju `delete_item` iz modula `database/item`, koja uklanja artikl iz baze podataka na temelju kôda. Ako je brisanje uspješno, vraća se potvrda sa statusnim kôdom 200 (OK), a u slučaju greške vraća se odgovarajuća poruka s kôdom.

3.1.4 Definiranje modela za validaciju podataka

Kako bi aplikacija prilikom obrade zahtjeva izbjegla nepotrebne greške poput nesuglasnosti podataka, stvaraju se unaprijed određeni modeli koji opisuju izgled podataka uz pomoć `Pydantic` biblioteke. Na slici 4 prikazan je grafički prikaz UML koji opisuje odnose između modela i njihovu povezanost. Sljedeći kôd prikazuje primjer definiranja Pydantic modela za tablicu Item.

```
class Item(BaseModel):
    id: str = Field(default_factory=lambda: str(uuid4()))
    name: str
    price: float
```

Artikl je definiran na način da sadrži svoj jedinstveni identifikator (`id`) tipa `string`, ime (`name`) tipa `string` i cijenu (`price`) tipa `float`. Prilikom stvaranja artikla `id` se automatski generira, dok `name` i `price` korisnik samostalno definira.

Nadalje, model kupona definiran je na način da sadrži kôd (`code`) tipa `string`, postotak sniženja (`discount_percentage`) tipa `float`, dopuštene artikle (`applicable_items`) u obliku liste `string`-ova i datum isteka (`expires_at`) tipa `string`. Pošto kuponi nisu obvezni dio internet trgovine, korisnik unosi sve podatke ručno pri stvaranju kupona, s mogućnošću izostavljanja polja poput postotka sniženja, dopuštenih artikala i datuma isteka.

Model pod nazivom `OrderItem` predstavlja pojedinačni artikl u narudžbi. Model sadrži atribute `item_id` tipa `string`, koji služi kao identifikator artikla te `quantity` tipa `integer` (cijeli broj), koji određuje količinu.

Model pod nazivom `OrderRequest` predstavlja zahtjev za obradom narudžbe, uključujući artikle i neobvezni kupon za popust. Model sadrži atribute `items` tipa *list*, koji predstavlja jednu stavku narudžbe i `coupon_code` tipa `string` za proizvoljni unos kupona.

Model pod nazivom `OrderResponse` predstavlja odgovor aplikacije po završetku obrade narudžbe. Model, nadalje, sadrži atribute `order_id` tipa `string`, koji predstavlja jedinstveni identifikator narudžbe, `total_price` tipa `float` (decimalni broj), koji predstavlja ukupan iznos narudžbe, odnosno cijenu prije primjene popusta, `discount_applied` tipa `boolean`, koji daje informaciju o tome je li kupon uspješno primijenjen te `final_price` tipa `float`, koji predstavlja konačnu cijenu narudžbe nakon primjene kupona.

3.1.5 Poslovna logika i komunikacija s bazom podataka

Za spajanje na AWS DynamoDB bazu podataka korištena je biblioteka `aioboto3` zbog svoje podrške za asinkrone upite. Biblioteka također koristi AWS CLI (eng. *command line interface*) za konfiguraciju tajnih ključeva koji su potrebni za pristup AWS uslugama.

Naredba

aws configure

stvara novu, ili uređuje već postojeću konfiguracijsku datoteku, unutar koje se nalaze podaci poput tajnog i pristupnog ključa.

```
import aioboto3

class DynamoDBClientManager:
    _session = None
    _client = None

    @staticmethod
    async def get_client():
        if DynamoDBClientManager._client is None:
            if DynamoDBClientManager._session is None:
                DynamoDBClientManager._session = aioboto3.Session
()
            DynamoDBClientManager._client = await
DynamoDBClientManager._session.client("dynamodb").__aenter__()
        return DynamoDBClientManager._client

    @staticmethod
    async def close_client():
        if DynamoDBClientManager._client is not None:
            DynamoDBClientManager._client.__aexit__(None, None,
None)
            DynamoDBClientManager._client = None
```

Prethodno prikazan kôd razvijen je kako bi se mogla uspostaviti veza s bazom podataka. Klasa je dizajnirana na način da olakša ponovno korištenje resursa iz baze podataka i da omogući asinkroni pristup tablicama. Klasa `DynamoDBClientManager` sadrži atribut `_session` koji čuva referencu na sesiju i atribut `_client` koji sprema instancu klijenta za komunikaciju s bazom podataka.

Metoda `get_client()` razvijena je s ciljem dohvaćanja i inicijalizacije zajedničkog klijenta DynamoDB. Metoda koristi `aioboto3.Session` za stvaranje asinkronog resursa. Ako je resurs već inicijaliziran, funkcija ga samo vraća. Ako nije, otvara se novi resurs unutar asinkronog konteksta i sprema se u `_client`.

Nadalje, metoda `close_client()` razvijena je s ciljem zatvaranja klijenta i sesije uslijed završetka rada servisa. Metoda provjerava postoji li inicijalizirana instanca klijenta. Ako postoji, ona zatvara sesiju i briše referencu.

Kako bi čitljivost kôda bila što bolja, kôd za komuniciranje s bazom podataka odvojen je u zasebne datoteke koje su razvrstane logičkim slijedom.

DynamoDB koristi drukčiji JSON format od uobičajenog, stoga je potrebno sve podatke pretvoriti u DynamoDB format. Kako bi se podaci uspješno pretvorili, koristi se metoda `serialize` klase `TypeSerializer` i pomoćne funkcije iz modula `utils/dynamodb_utils`, prikazane sljedećim kôdom.

```python
serializer = TypeSerializer()
deserializer = TypeDeserializer()

def convert_decimal(obj: Any) -> Any:
    if isinstance(obj, Decimal):
        return int(obj) if obj % 1 == 0 else float(obj)
    elif isinstance(obj, list):
        return [convert_decimal(item) for item in obj]
    elif isinstance(obj, dict):
        return {k: convert_decimal(v) for k, v in obj.items()}
    return obj

def to_dynamodb_json(data: Any) -> dict:
    if isinstance(data, dict):
        return {key: to_dynamodb_json(value) for key, value in
   data.items()}
    if isinstance(data, list):
        return {"L": [to_dynamodb_json(item) for item in data]}
    return serializer.serialize(data)

def deserialize(data: Any) -> Any:
    if isinstance(data, list):
        return [deserialize(item) for item in data]
    if isinstance(data, dict):
        try:
            return convert_decimal(deserializer.deserialize(data))
        except TypeError:
            return {k: deserialize(v) for k, v in data.items()}
    return data
```

Funkcija `create_item` prima parametar `item` tipa `dict` i služi za stvaranje novih artikala u bazi podataka. Referenca na klijenta dohvaća se pomoću metode `DynamoDBClientManager.get_client()`. Po završetku dohvaćanja reference klijenta pripremaju se podaci za unos u DynamoDB bazu podataka. Metoda `put_item` služi za unošenje podataka u bazu te kao parametar prima `TableName` (ime tablice) i stavku za unos `Item`. Funkcija kao odgovor vraća poruku *"Item created successfully!"* i statusni kôd 200 (OK). Ako prilikom izvođenja funkcije dođe do greške, ona se uhvati, nakon čega se vraća opis greške sa statusnim kôdom 500 (*ClientError*).

```
serializer = TypeSerializer()

async def create_item(item: dict) -> JSONResponse:
    client = await DynamoDBClientManager.get_client()
    try:
        item['price'] = Decimal(str(item['price']))
        item['id'] = str(uuid4())  # Generate unique ID
        dynamo_item = to_dynamodb_json(item)

        await client.put_item(TableName="Item", Item=dynamo_item)
        return JSONResponse(content="Item created successfully!",
   status_code=200)

    except ClientError as e:
        return JSONResponse(content={"error": f"AWS Client Error:
   {e.response.get('Error', {}).get('Message', 'Unknown error')}"
   }, status_code=500)
```

Funkcija `get_items` služi dohvaćanju svih artikala iz baze podataka. Referenca na klijenta dohvaća se pomoću metode `get_client()`. Metoda `scan` koristi se za pretraživanje svih stavki unutar tablice te podržava parametre poput `Limit` i `Scan` za upravljanje pretragom. Rezultat metode čini rječnik koji sadrži ključ `Items`, koji je potrebno pretvoriti u normalni JSON format s pomoćnom funkcijom `deserialize`. Kako bi dohvaćeni podaci bili u skladu s unaprijed definiranim modelom, pretvaraju se u instancu klase `Item` i vraćaju kao popis objekata. Ako se prilikom izvođenja funkcije dogodi pogreška, uhvati se iznimka i vraća JSON odgovor s opisom greške i statusnim kôdom 500 (*ClientError*).

```
async def get_items() -> JSONResponse:
    client = await DynamoDBClientManager.get_client()
    try:
        # Scan for items with a limit
        response = await client.scan(TableName="Item", Limit=100)
        deserialized_items = deserialize(response.get('Items', [])
   )

        # Convert to Item models
        items = [Item(**item) for item in deserialized_items]
        return JSONResponse(content=[item.dict() for item in items
   ], status_code=200)

    except ClientError as e:
        return JSONResponse(content={"error": f"AWS Client Error:
   {str(e)}"}, status_code=500)
```

Funkcija `get_item` prima parametar `id` tipa `string` i služi dohvaćanju specifičnog artikla iz baze podataka. Referenca na klijenta se dohvaća pomoću metode `get_client()`. Nakon dohvaćanja reference koristi se metoda `get_item` za dohvaćanje jedne stavke na temelju primarnog ključa. Parametar `Key` čini rječnik koji određuje ključ stavke. U gore navedenom primjeru ključ je id. Rezultat metode predstavlja rječnik koji sadrži detalje o stavki koje je potrebno pretvoriti u JSON format pomoću metode `deserialize`. Ako stavka postoji u bazi podataka, vraća se kao JSON odgovor sa stavkom i kôdom 200 (OK).

Ako stavka ne postoji, iznimka se uhvati i, ovisno o vrsti iznimke, (`ItemNotFound` ili `ClientError`) šalje JSON odgovor s kôdom 404 (Nije pronađeno) ili 500.

```
async def get_item(id: str) -> JSONResponse:
    client = await DynamoDBClientManager.get_client()
    try:
        response = await client.get_item(
            TableName="Item",
            Key={"id": serializer.serialize(id)}
        )

        if 'Item' not in response:
            raise ItemNotFound(f"Item with ID: {id} does NOT exist
    .")

        item_data = deserialize(response['Item'])
        return JSONResponse(content=item_data, status_code=200)

    except ClientError as e:
        return JSONResponse(content={"error": f"AWS Client Error:
    {e.response.get('Error', {}).get('Message', 'Unknown error')}"
    }, status_code=500)
    except ItemNotFound:
        return JSONResponse(content={"error": f"Item with ID: {id}
     does NOT exist."}, status_code=404)
```

Funkcija `remove_item` prima parametar `id` tipa `string` i služi uklanjanju artikla iz baze podataka. Referenca na klijenta se dohvaća pomoću metode `get_client`. Nakon dohvaćanja reference koristi se metoda `delete` s parametrima `TableName` i `Key`. Parametar `TableName` određuje nad kojom tablicom se izvršava operacija, a parametar `Key` određuje ključ pretrage. Funkcija kao odgovor vraća poruku *"Item removed successfully!"* i statusni kôd 200 (OK). Ako se prilikom izvođenja dogodi greška, ona se uhvati te se vraća pripadajuća poruka sa statusnim kôdom.

```
async def remove_item(id: str) -> JSONResponse:
    client = await DynamoDBClientManager.get_client()
    try:
        response = await client.delete(
            TableName="Item",
            Key={"id": serializer.serialize(id)}
        )

        if 'Item' not in response:
            raise ItemNotFound(f"Item with ID: {id} does NOT exist
    .")

        return JSONResponse(content="Item removed successfully!",
    status_code=200)

    except ClientError as e:
        return JSONResponse(content={"error": f"AWS Client Error:
    {e.response.get('Error', {}).get('Message', 'Unknown error')}"
    }, status_code=500)
    except ItemNotFound:
        return JSONResponse(content={"error": f"Item with ID: {id}
     does NOT exist."}, status_code=404)
```

Poslovna logika za `database/coupon` identična je `database/item` logici, pošto se koriste istim operacijama nad drugom tablicom. Modul `database/order` nema direktnu komunikaciju s bazom podataka, već koristi definirane funkcije iz ostalih modula za dohvaćanje podataka. Modul se sastoji od jedne pomoćne funkcije za izračun konačnog iznosa narudžbe nakon primjene kupona i glavne funkcije za obradu narudžbe.

```
def calculatePrice(order: OrderRequest, coupon: Optional[Coupon],
    fetched_items: List[dict]) -> float:
     total_price = sum(item['price'] * order_item.quantity for item
    , order_item in zip(fetched_items, order.items))

     if not coupon:
         return total_price

     # Check if the coupon is expire
     if coupon.expires_at and datetime.datetime.fromisoformat(
    coupon.expires_at) < datetime.datetime.now():
         raise HTTPException(status_code=400, detail="Coupon has
    expired")

     discount = 0.0

     # Calculate the discount
     if coupon.applicable_items:
         for item, order_item in zip(fetched_items, order.items):
             if item['id'] in coupon.applicable_items:
                 discount += item['price'] * order_item.quantity *
    (coupon.discount_percentage / 100)

     final_price = max(0.0, total_price - discount)  # Final price
    should not be negative
     return final_price
```

Funkcija `calculatePrice` prima parametre `order` (predstavljaju narudžbu), `coupon` (objekt kupona) i `fetched_itms` (lista dobivenih artikala). Cijena, odnosno iznos narudžbe, računa se iterativno tako što se množi cijena svakog artikla sa zadanom količinom te se potom zbrajaju cijene svih artikala. Ako kupon nije primijenjen na narudžbu, funkcija vraća zbrojenu cijenu bez daljnjih kalkulacija. Ako je on pak primijenjen, provjerava je li kupon istekao. Ako kupon sadrži listu artikala, popust se primjenjuje samo na te artikle i vraća se konačna cijena s primijenjenim popustom.

Funkcija `applyCoupon` prima parametar `order` tipa `OrderRequest` i služi za primjenu kupona i obradu narudžbe. Ako narudžba sadrži kôd kupona, kupon se pokuša dohvatiti pomoću funkcije `get_coupon` iz modula `database/coupon`. Ako kupon postoji, stvara se objekt tipa `Coupon` pomoću podataka iz odgovora. Ako kupon nije moguće pronaći, podiže se iznimka. Nakon provjere kupona dohvaćaju se svi artikli iz narudžbe i spremaju u listu. Poziva se pomoćna funkcija `calculatePrice` za izračun konačne cijene narudžbe uzimajući u obzir moguće popuste. Nakon obrade stvara se objekt tipa `OrderResponse` koji sadrži sve informacije o narudžbi, odnosno jedinstveni ID narudžbe, konačnu cijenu te informaciju o primjeni popusta.

Funkcija kao odgovor vraća JSON odgovor s podacima o narudžbi i statusnim kôdom 200 (OK). Ako se prilikom izvođenja dogodi greška, ona se uhvati i vraća odgovor s pripadajućom porukom i statusnim kôdom.

```
async def applyCoupon(order: OrderRequest):
    coupon = None
    if order.coupon_code:
        response = await get_coupon(order.coupon_code)

        if response.status_code == 200:
            response_data = json.loads(response.body.decode())
            coupon = Coupon(**response_data)
        else:
            raise HTTPException(status_code=400, detail="Coupon
 not found")

    fetched_items = []
    for item in order.items:
        item_response = await get_item(item.item_id)
        item_data = json.loads(item_response.body.decode())
        fetched_items.append(item_data)

    total_price = sum(item['price'] * order_item.quantity for item
 , order_item in zip(fetched_items, order.items))
    final_price = calculatePrice(order, coupon, fetched_items)

    order_response = OrderResponse(
        order_id = str(uuid4()),
        total_price = total_price,
        discount_applied = bool(coupon),
        final_price = final_price
    )

    return JSONResponse(content=order_response.dict() ,
 status_code=200)
```

3.2 Rust

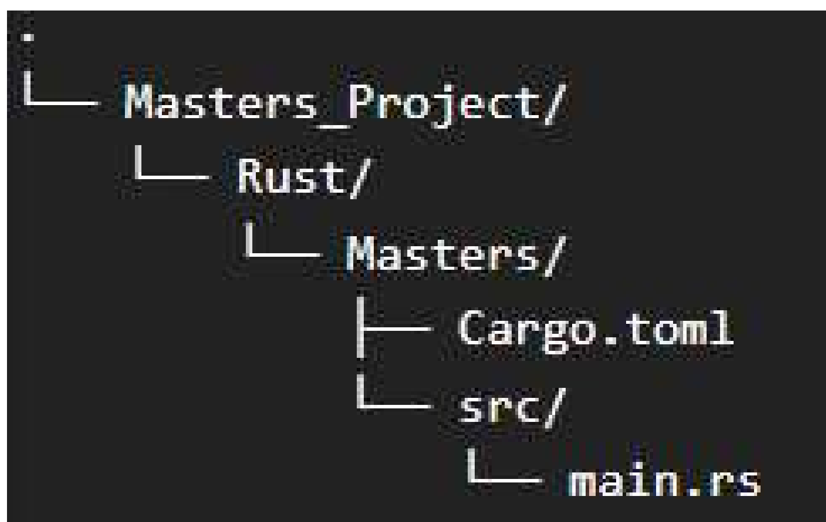
Naredno poglavlje opisuje implementaciju API-ja u programskom jeziku Rust koristeći se Actix-web razvojnim okvirom. Poseban naglasak stavljen je na visoku učinkovitost, asinkrono izvršavanje i optimalno upravljanje memorijom. Implementacija demonstrira sposobnost Rust-a u postizanju visoke propusnosti i niske latencije uz potpunu iskorištenost CPU resursa

3.2.1 Stvaranje novog projekta

Za implementaciju servisa u Rust-u stvoren je novi projekt pomoću alata Cargo. Cargo čini rust-ov pripadajući alat za upravljanje bibliotekama, sličan alatu pip za Python. Komanda za stvaranje novog Rust projekta pod imenom "Masters" nalazi se ispod:

```
cargo new Masters
```

Na slici 6. prikazana je struktura stvorenog projekta. Datoteka `Cargo.toml` predstavlja manifest projekta koji sadrži potrebne informacije poput verzija biblioteka, verzije projekta i ostalih značajki za pravilnu izgradnju projekta.



Slika 6: Prikaz strukture stvorenog projekta (izvor: Autor)

3.2.2 Postavljanje glavnog dokumenta

Glavni dokument u Rust projektu čini datoteka `main.rs`, koja se prema zadanim postavkama nalazi u mapi `src`. Navedena datoteka predstavlja početnu točku za izvršavanje programa. U ovom projektu, datoteka `main.rs` konfigurirana je na način da pokreće osnovni mrežni servis koristeći biblioteku Actix-web koja omogućuje izgradnju RESTful API servisa.

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .service(hello)
            .service(echo)
            .route("/hey", web::get().to(manual_hello))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Prethodno naveden kôd čini sadržaj osnovne verzije datoteke `main.rs` koja inicijalizira poslužitelja. Funkcija `HttpServer::new` korištena je za definiranje instanci poslužitelja, dok je metoda `bind` iskorištena za određivanje porta na kojem će poslužitelj primati zahtjeve.

3.2.3 Dodavanje osnovnih ruta

Nakon osnovne konfiguracije poslužitelja, potrebno je definirati rute koje će odgovarati na dobivene zahtjeve. U ovom projektu rute su strukturirane uz pomoć različitih modula kako bi se održala preglednost kôda te kako bi se olakšalo buduće proširivanje funkcionalnosti. Sljedeći kôd prikazuje primjer dodavanja ruta grupiranih u tri domene: `coupon`, `item` i `order`. Dodavanje ruta objasnit će se na modulu `routes::coupon`.

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .service(web::resource("/").route(web::get().to(index)
   ))
            .service(web::scope("/coupon").configure(routes::
   coupon::init_routes))
            .service(web::scope("/item").configure(routes::item::
   init_routes))
            .service(web::scope("/order").configure(routes::order
   ::init_routes))
    })
    .bind("0.0.0.0:8000")?
    .run()
    .await
}
```

U modulu `routes::coupon`, prikazanim sljedećim kôdom, definirane su sve rute povezane s kuponima koje omogućuju operacije poput dodavanja novog kupona, dohvaćanja kupona i brisanja kupona.

```
#[get("/get/{code}")]
async fn get_coupon_by_code(path: web::Path<String>) -> impl
   Responder {
    let code = path.into_inner();
    match coupon::get_coupon_by_code(&code).await {
        Ok(coupon) => HttpResponse::Ok().json(coupon),
        Err(_) => HttpResponse::NotFound().body("Coupon not found"
   ),
    }
}

#[post("/create")]
async fn create_coupon(coupon: web::Json<Coupon>) -> Result<
   HttpResponse, CouponError> {
    match coupon::create_coupon(coupon).await {
        Ok(response) => Ok(HttpResponse::Ok().json(response)),
        Err(err) => Err(err),
    }
}

#[get("/getall")]
async fn get_all_coupons() -> Result<HttpResponse, CouponError> {
    match coupon::get_all_coupons().await{
        Ok(response) => Ok(HttpResponse::Ok().json(response)),
        Err(err) => Err(err),
```

```
    }
}

#[delete("/remove/{code}")]
async fn remove_coupon(path: web::Path<String>) -> Result<
    HttpResponse, CouponError> {
    let code = path.into_inner();
    match coupon::delete_coupon_by_code(&code).await {
        Ok(response) => Ok(HttpResponse::Ok().json(response)),
        Err(err) => Err(err),
    }
}

pub fn init_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(get_coupon_by_code);
    cfg.service(create_coupon);
    cfg.service(get_all_coupons);
}
```

Ruta `/get/{code}`, tipa GET, koristi funkciju `get_coupon_by_code` koja dohvaća kupon na temelju kôda. Ako je kupon pronađen, vraća se odgovor sa statusnim kôdom 200 (OK) i detaljima kupona u JSON formatu. Ako kupon nije moguće pronaći, vraća se odgovor sa statusnim kôdom 404 (Not Found). Funkcija koristi modul `db::coupon` za dohvaćanje kupona iz baze podataka. Ako funkcija za dohvaćanje vrati grešku, klijent dobiva odgovor s porukom o nepostojanju kupona. Ruta `/create` , tipa POST, omogućuje stvaranje novog kupona. Korisnik šalje JSON podatke o kuponu, a funkcija `create_coupon` obrađuje zahtjev. Ako je kupon uspješno kreiran, vraća se odgovor sa statusnim kôdom 200 (OK) i detaljima stvorenog kupona. U slučaju greške vraća se odgovarajući statusni kôd. Funkcija koristi strukturu `Coupon` za validaciju i upravljanje podacima te poziva funkciju `create_coupon` iz modula `db::coupon` za unos podataka u bazu podataka. Ruta `/getall`, tipa GET, vraća sve kupone pohranjene u bazi podataka. Rezultat se vraća u obliku JSON popisa kupona sa statusnim kôdom 200 (OK). U slučaju greške vraća se odgovarajuća poruka sa statusnim kôdom. Ruta koristi funkciju `get_all_coupons` iz modula `db::coupon` za dohvaćanje podataka iz baze.

Ruta `/remove/{code}`, tipa DELETE, koristi funkciju `delete_coupon_by_code` iz modula `db::coupon` kako bi uklonila kupon na temelju kôda. Ako je brisanje uspješno, vraća se potvrda sa statusnim kôdom 200 (OK). U slučaju greške vraća se odgovarajuća poruka s kôdom.

Za razliku od modula `coupon i item`, koji imaju identične rute i operacije, modul `order`, prikazan kôdom koji slijedi, sadrži jednu rutu tipa POST s imenom `apply`. Ruta `/apply` omogućuje korisnicima primjenu kupona na narudžbu. Korisnik šalje JSON podatke o narudžbi, uključujući stavke unutar narudžbe, kao i detalje kupona koji se na nju pokušava primijeniti.

Funkcija apply_coupon iz modula db::coupon obrađuje zahtjev i vraća odgovor koji sadrži konačne podatke o narudžbi nakon primjene popusta.

```
#[post("/apply")]
async fn apply_coupon(order_request: web::Json<OrderRequest>) ->
    Result<HttpResponse, OrderProcessingError> {
     match order::apply_coupon_to_order(order_request).await {
          Ok(order_response) => Ok(HttpResponse::Ok().json(
    order_response)),
          Err(err) => Err(err),
     }
}

pub fn init_routes(cfg: &mut web::ServiceConfig) {
     cfg.service(apply_coupon);
}
```

Sve rute iz modula routes registrirane su pomoću funkcije init_routes. Navedena funkcija osigurava ispravnost konfiguracije rute unutar domene.

3.2.4 Poslovna logika i komunikacija s bazom podataka

Za spajanje na AWS DynamoDB bazu podataka u Rust koristi se biblioteka aws_sdk_dynamodb zbog asinkronog načina rada. Niže je prikazan kôd za uspostavljanje veze s bazom podataka koristeći se singleton uzorkom. Navedeni uzorak limitira inicijalizaciju klase na jednu instancu, stoga nije potrebno ponovno inicijalizirati vezu s bazom pri svakom upitu. Funkcija get_dynamo_client služi za dohvaćanje ili inicijalizaciju veze s bazom podataka.

```
static DYNAMODB_CLIENT: OnceCell<Arc<Client>> = OnceCell::
    const_new();

pub async fn get_dynamodb_client() -> Arc<Client> {
     DYNAMODB_CLIENT
          .get_or_init(|| async {
               let shared_config = aws_config::load_from_env().await;
               let client = Client::new(&shared_config);
               Arc::new(client)
          })
          .await
          .clone()
}
```

Modul db::coupon sadrži poslovnu logiku za sve rute povezane s kuponima. Funkcija get_coupon_by_code prikazana sljedećim kôdom omogućuje dohvaćanje kupona temeljem njegovog jedinstvenog kôda. Funkcija koristi metodu get_item za pristup tablici Coupon. Ako kupon postoji, podaci se transformiraju u instancu modela Coupon te se kao odgovor vraća dohvaćeni kupon, ili pak greška s odgovarajućim statusnim kôdom.

```
pub async fn get_coupon_by_code(code: &str) -> Result<Coupon,
    CouponError> {
     let client = get_dynamodb_client().await;

     let request = client
         .get_item()
         .table_name("Coupon")
         .key("code", AttributeValue::S(code.to_string()))
         .send()
         .await.map_err(|e| CouponError::ParseError)?;

     if let Some(item) = request.item {
         let discount = item.get("discount_percentage")
             .ok_or_else(|| CouponError::ParseError)?
             .as_n()
             .map_err(|_| CouponError::ParseError)?
             .parse::<u32>()
             .map_err(|_| CouponError::ParseError)?;

         let applicable_items = item.get("applicable_items").
    and_then(|v| {
             v.as_l().ok().map(|list| {
                 list.iter()
                     .filter_map(|val| val.as_s().ok().map(|s| s.
    to_string()))
                     .collect::<Vec<String>>()
             })
         });

         let expires_at = item.get("expires_at").and_then(|v| {
             v.as_s().ok().map(|s| s.to_string())
         });

         let coupon = Coupon {
             code: item.get("code")
                 .ok_or_else(|| CouponError::ParseError)?
                 .as_s()
                 .map_err(|_| CouponError::ParseError)?
                 .to_string(),
             discount,
             applicable_items,
             expires_at,
         };
         Ok(coupon)
     } else {
         Err(CouponError::NotFound)
     }
}
```

Funkcija `create_coupon`, prikazana kôdom niže, dodaje novi kupon u bazu podataka. Funkcija koristi metodu `put_item` u svrhu pohrane svih atributa kupona u tablicu `Coupon`. Prije spremanja izvršava se provjera za opcionalne vrijednosti poput `applicable_items i expires_at` koje se pretvaraju u određeni tip podataka za DynamoDB. Funkcija kao odgovor vraća poruku "*Coupon created successfully!*" i statusni kôd 200 (OK) ili opis greške s odgovarajućim statusnim

kôdom.

```
pub async fn create_coupon(coupon: web::Json<Coupon>) -> Result<
    String, CouponError> {
    let client = get_dynamodb_client().await;

    let new_coupon = coupon.into_inner();

    let mut request = client
        .put_item()
        .table_name("Coupon")
        .item("code", AttributeValue::S(new_coupon.code.clone()))
        .item("discount_percentage", AttributeValue::N(new_coupon.
    discount.to_string()));

    if let Some(applicable_items) = &new_coupon.applicable_items {
        let items = applicable_items.iter()
            .map(|item| AttributeValue::S(item.clone()))
            .collect::<Vec<AttributeValue>>();
        request = request.item("applicable_items", AttributeValue
    ::L(items));
    }

    if let Some(expires_at) = &new_coupon.expires_at {
        request = request.item("expires_at", AttributeValue::S(
    expires_at.clone()));
    }

    request
        .send()
        .await
        .map_err(|err| CouponError::DynamoDbError)?;

    Ok("Coupon created successfully.".to_string())
}
```

Funkcija `get_all_coupons` prikazana sljedećim kôdom omogućuje dohvaćanje svih kupona iz tablice koristeći se metodom `scan`. Nakon dohvaćanja iz baze podaci se raščlanjuju i pretvaraju u odgovarajući tip podatka (npr. `string`, u32). Ako pojedini atribut nedostaje ili je pogrešno formatiran, funkcija vraća prilagođenu grešku (npr. `CouponError::MissingAttribute` ili `CouponError::ParseError`) s odgovarajućim statusnim kôdom. Nakon obrade svih stavki, korisniku funkcija kao odgovor vraća listu kupona u formatu `Vec<Coupon>`.

```
pub async fn get_all_coupons() -> Result<Vec<Coupon>, CouponError>
    {
    let client = get_dynamodb_client().await;

    let request = client
        .scan()
        .table_name("Coupon")
        .send()
        .await
        .map_err(|err| CouponError::DynamoDbError)?;

    let items = request.items.unwrap_or_default();
```

```
    let fetched_coupons: Result<Vec<Coupon>, CouponError> =
    items.into_iter().map(|item| {
        let code = item.get("code")
            .and_then(|v| v.as_s().ok())
            .ok_or(CouponError::MissingAttribute)?
            .to_string();

            let discount = match item.get("discount_percentage") {
                Some(attr) => attr.as_n()
                    .map_err(|_| CouponError::MissingAttribute)?
                    .parse::<u32>()
                    .map_err(|_| CouponError::ParseError)?,
                None => {
                    eprintln!("Warning: 'discount' attribute
   missing for coupon with code '{}'. Defaulting to 0.", code);
                    0
                }
            };

        let applicable_items = item.get("applicable_items")
            .and_then(|v| v.as_l().ok())
            .map(|attr_list| {
                attr_list.iter().filter_map(|attr| attr.as_s().ok
   ().map(String::from)).collect()
            });

        let expires_at = item.get("expires_at")
            .and_then(|v| v.as_s().ok())
            .map(String::from);

        Ok(Coupon { code, discount, applicable_items, expires_at
   })
    }).collect();

    fetched_coupons
}
```

Funkcija `delete_coupon_by_code`, prikazana kôdom niže, omogućuje uklanjanje kupona iz tablice temeljem njegova kôda. Funkcija koristi metodu `delete_item` te kao odgovor vraća odgovor sa statusnim kôdom 200 ili pak grešku s opisom i odgovarajućim statusnim kôdom.

```
pub async fn delete_coupon_by_code(code: &str) -> Result<(),
   CouponError> {
    let client = get_dynamodb_client().await;
    let request = client
        .delete_item()
        .table_name("Coupon")
        .key("code", AttributeValue::S(code.to_string()))
        .send()
        .await;
    match request {
        Ok(_) => Ok(()),
        Err(_) => Err(CouponError::DynamoDbError),
    }
```

```
}
```

Modul `db::item` sadrži identičnu poslovnu logiku kao i modul `db::coupon`, no glavnu razliku predstavlja tablica nad kojom se izvršavaju operacije i model za validaciju podataka. Za razliku od modula `db::coupon i db::item`, modul `db::order`, prikazan kôdom niže, sadrži jednu glavnu i jednu pomoćnu funkciju.

Funkcija `apply_coupon_to_order` obrađuje zahtjev narudžbe, računa ukupnu cijenu artikla, primjenjuje kupon i vraća odgovor s detaljima narudžbe. Svakoj narudžbi pridodan je jedinstveni identifikator stvoren pomoću biblioteke `uuid`. Funkcija `get_items_by_ids` dohvaća sve artikle iz narudžbe temeljem njihovih identifikatora (`item_id`). Ako prilikom dohvaćanja artikli ne budu pronađeni, funkcija vraća grešku `ItemFetchError` i odgovarajući statusni kôd. Ukupan iznos, odnosno ukupna cijena, narudžbe izračunata je zbrajanjem cijena svih artikala, uzimajući u obzir njihove količine. Korištena je funkcija `iter().zip()` kako bi se dohvaćeni artikli povezali s njihovim količinama iz zahtjeva. Ako je korisnik unio kupon, funkcija `get_coupon_by_code` dohvaća kupon i provodi validaciju. U slučaju da kupon ne postoji, vraća se greška `INvalidCouponError`.

```
pub async fn apply_coupon_to_order(
    order_request: web::Json<OrderRequest>,
) -> Result<OrderResponse, OrderProcessingError> {
    let client = get_dynamodb_client().await;

    let order_id = Uuid::new_v4().to_string();

    // Fetch items from the database
    let item_ids: Vec<String> = order_request.items.iter().map(|
   item| item.item_id.clone()).collect();
    let items = get_items_by_ids(&client, &item_ids)
    .await
    .map_err(|_| OrderProcessingError::ItemFetchError)?;

    let total_price: f64 = items.iter()
        .zip(order_request.items.iter())
        .map(|(item, order_item)| item.price * order_item.quantity
    as f64)
        .sum();

    let (discount_applied, final_price) = if let Some(coupon_code)
    = &order_request.coupon_code {
        match get_coupon_by_code(coupon_code).await {
            Ok(coupon) => {
                if coupon_applies_to_items(&coupon, &order_request
   .items) {
                    let discount_amount = (coupon.discount as f64)
    / 100.0 * total_price;
                    (true, total_price - discount_amount)
                } else {
                    return Err(OrderProcessingError::
   CouponNotApplicable);
                }
            }
            Err(_) => return Err(OrderProcessingError::
   InvalidCouponError),
```

```
        }
    } else {
        (false, total_price)
    };

    let order_response = OrderResponse {
        order_id,
        total_price,
        discount_applied,
        final_price,
    };

    Ok(order_response)
}
```

Pomoćna funkcija `coupon_applies_to_items` provjerava odnosi li se kupon na artikle u narudžbi. Ako se kupon ne može primijeniti, funkcija vraća grešku `CouponNotApplicable`. Nakon obrade narudžbe i primjene kupona, funkcija vraća odgovor s detaljima narudžbe uključujući ukupnu cijenu artikala, kao i konačnu cijenu nakon primjene popusta.

```
// Helper function to check if the coupon applies to the items
fn coupon_applies_to_items(coupon: &Coupon, items: &[OrderItem])
    -> bool {
    if let Some(applicable_items) = &coupon.applicable_items {
        items.iter().any(|item| applicable_items.contains(&item.
    item_id))
    } else {
        true // If no applicable items are specified, assume the
    coupon is applicable to all items
    }
}
```

3.2.5 Validacija podataka

Za validaciju ulaznih podataka koristi se Rust-ova biblioteka `serde` koja služi za serijalizaciju i deserializaciju podataka. Model `Coupon` opisuje strukturu kupona unutar aplikacije i baze podataka. Sastoji se od varijable `code` tipa `string`, varijable `discount` tipa `u32` (`integer`), varijable `applicable_items` tipa `Option<Vec<String>>` (lista `string`-ova) i opcionalne varijable `expires_at` tipa `string`. Primjer strukture podataka prikazan je sljedećim kôdom.

```
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, Debug)]
pub struct Coupon {
    pub code: String,
    pub discount: u32,
    pub applicable_items: Option<Vec<String>>,
    pub expires_at: Option<String>,
}
```

Model `Item` opisuje strukturu artikla unutar aplikacije i baze podataka. Sastoji se od varijable `code` tipa `String`, varijable `name` tipa `String` i varijable `price` tipa `f64` (`float`).

Modeli `OrderItem`, `OrderRequest` i `OrderResponse` opisuju sve strukture povezane s narudžbama u aplikaciji. Model `OrderItem` sastoji se od varijable `item_id` tipa `String` i varijable `quantity` tipa `u32` (`integer`). Model `OrderRequest` sastoji se od varijable `items` tipa `Vec<OrderItem>`(lista OrderItem-a) i opcionalne varijable `coupon_code` tipa `String`. Model `OrderResponse` sastoji se od varijable `order_id` tipa `String`, varijable `total_price` tipa `f64` (`float`), varijable `discount_applied` tipa `bool` (`boolean`) i varijable `final_price` tipa `f64` (`float`).

3.3 Infrastruktura

Kako bi se ispitivanje moglo uspješno provesti, potrebno je postaviti okruženje u kojemu će se servisi pokrenuti.

3.3.1 EC2

Kako bi rezultati ispitivanja bili precizni, korištena je AWS EC2 instanca za svaki od servisa. Prijavom i ulaskom u AWS EC2 korisniku je predstavljeno glavno sučelje, prikazano slikom 7., pomoću kojeg se stvara EC2 instanca i na kojemu se mogu vidjeti svi podaci vezani uz resurse. Prije stvaranja instance potrebno je odabrati regiju u gornjem desnom kutu stranice, kao što je vidljivo na primjeru 'Europe (Frankfurt)'.

Pritiskom na "*Launch instance*" korisnika se preusmjerava na sučelje za stvaranje instance.

U ovom diplomskom radu, za usporedbu, stvorene su dvije instance sa 'Ubuntu Server 24.04 LTS' šablonom. Obje instance koriste 't2.micro' tip s 1 CPU jezgrom i 1 GiB memorije zbog svoje dostupnosti u besplatnom paketu. Konfiguracija mreže odvija se automatski.

Spajanje na stvorene instance omogućeno je preko `SSH` ili preko web klijenta pritiskom na instancu i odabirom "*Connect*".

Za postavljanje web aplikacije u Python-u potrebno je dohvatiti projekt s GitHub-a. Kako bi projekt bio dohvaćen, potrebno je prvo instalirati `git` pomoću naredbe:

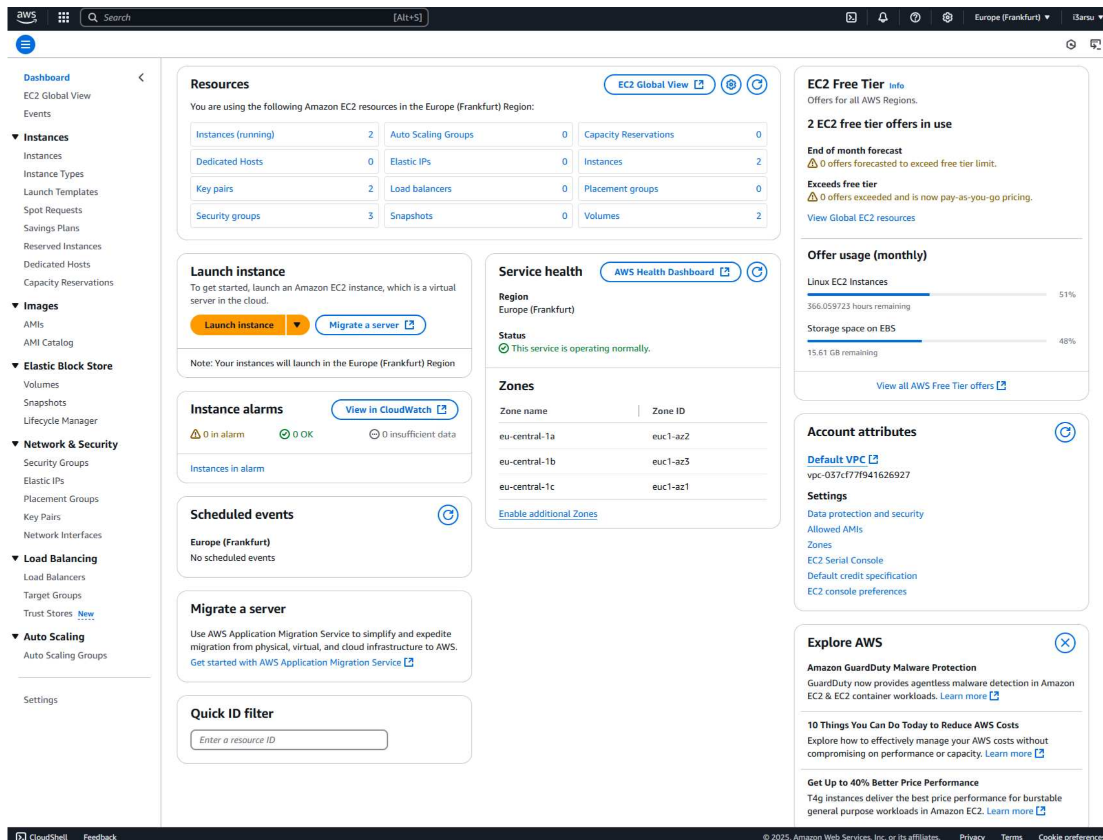
```
apt-get install git
```

Nakon instalacije i postavljanja git softvera, koristi se naredba:

```
git clone <github_link>
```

koja preuzima (klonira) projekt s GitHub-a. Kako bi se projekt pokrenuo, potrebno je instalirati sve biblioteke odgovarajućih verzija u posebnom virtualnom okruženju. Popis svih biblioteka nalazi se u datoteci `requirements.txt`, a komandu za instaliranje iz datoteke čini:

```
pip install -r requirements.txt
```



Slika 7: Glavno sučelje AWS EC2 (Dashboard) (izvor: Autor)

Nakon instalacije biblioteka te prije samog pokretanja aplikacije potrebno je postaviti vatrozid (eng. *firewall*) koji dopušta upite s mrežnog priključka (eng. *port*). Kako bi se postavio vatrozid, potrebno je pokrenuti naredbe:

```
sudo ufw allow 8000/tcp
sudo ufw enable
```

Prije pokretanja aplikacije potrebno je urediti datoteku `'.env'` s tajnim ključevima koji služe spajanju na AWS račun. Kada se vatrozid uspješno postavi i kada se uredi datoteka ".env", aplikacija je spremna za pokretanje i za primanje zahtjeva korisnika. Naredba za pokretanje servisa u pozadini je:

```
nohup python3 main.py &
```

Pošto je Rust kompilirani jezik, nije potrebno preuzeti cijeli projekt s GitHub-a, već je dovoljno preuzeti sastavljenu datoteku spremnu za pokretanje. Za preuzimanje posljednjeg izdanja aplikacije koristi se softver `wget`, a naredbu za preuzimanje čini:

```
wget <link_na_datoteku>
```

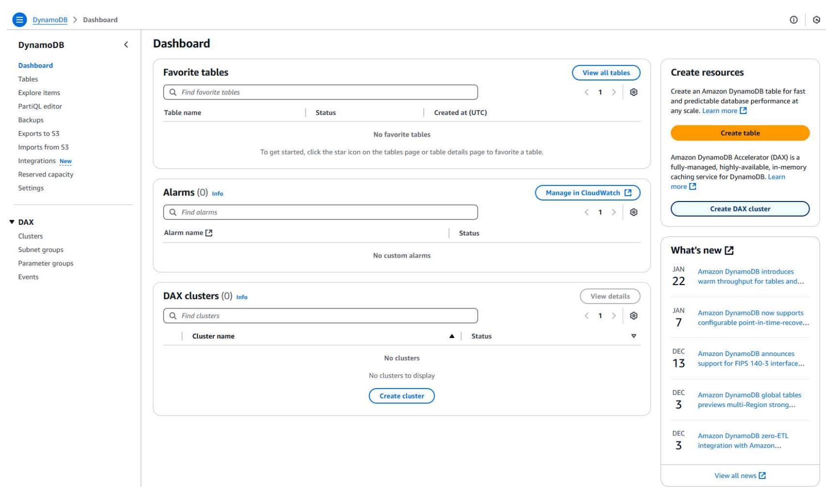
Link koji vodi do datoteke moguće je dohvatiti kopiranjem hiperveze na stranici GitHub-a. Nakon preuzimanja, datoteku je potrebno raspakirati i postaviti vatrozid pomoću naredbi:

```
unzip Masters.zip
sudo ufw allow 8000/tcp
sudo ufw enable
```

Datoteka je spremna za pokretanje s istom komandom kao i u Python servisu.

3.3.2 DynamoDB

Za postavljanje DynamoDB baze podataka potrebno je razumjeti kakav se tip podataka pohranjuje. Pošto DynamoDB pohranjuje podatke u tablicama koje su organizirane prema ključevima, za svaki podatak određuje se primarni ključ (primjer u tablici `Coupon` gdje primarni ključ čini `code`). Kako bi tablica bila uspješno postavljena, potreban je pristup AWS računu za prijavu u web sučelje (slika 8.).

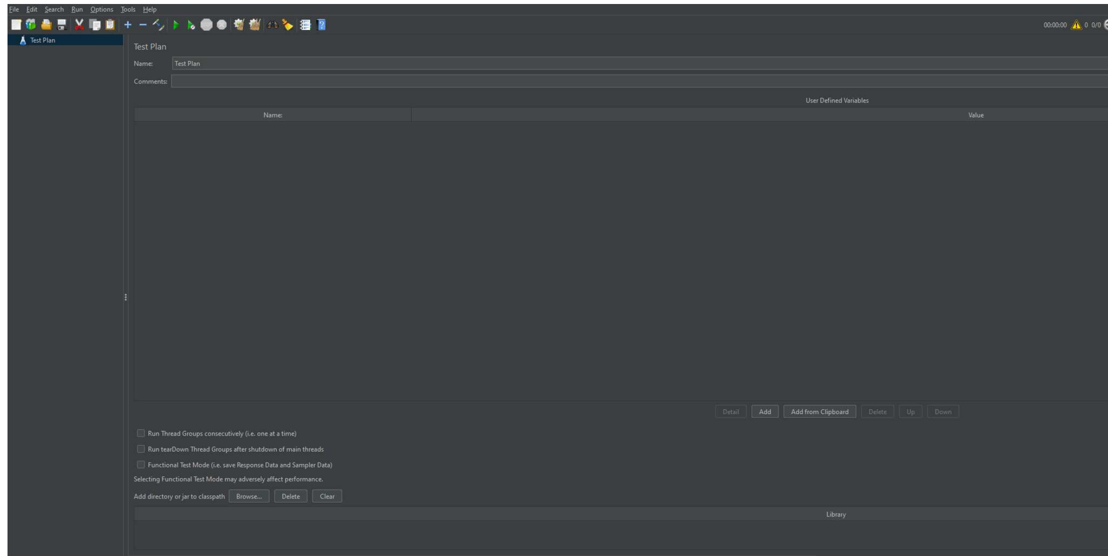


Slika 8: Web sučelje AWS DynamoDB (izvor: Autor)

Pritiskom na "*Create table*" korisnik se preusmjerava do sučelja za upisivanje podataka o tablici. Za ovaj projekt stvorene su dvije tablice: *Coupon* i *Item*. Tablica `Coupon` kao particijski ključ koristi `code` tipa String, a tablica Item koristi `id` tipa String. Ostale postavke, kao što su maksimalni kapacitet upisa i čitanja, puštene su na zadanim vrijednostima s mogućnošću kasnijeg podešavanja.

3.4 Ispitivanje performansi (Jmeter)

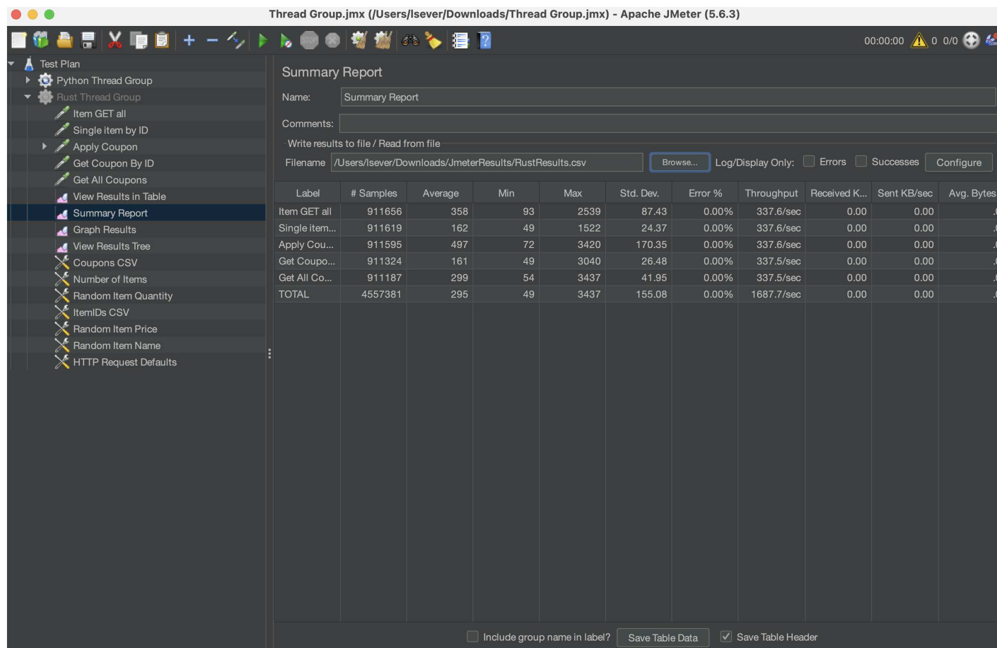
Apache Jmeter je besplatni alat za testiranje web aplikacija i API-ja. Pošto je alat napisan u programskom jeziku Java, za njegovo pokretanje potreban je paket alata za razvoj u Java programskom jeziku (**JDK**, *eng. Java Development Kit*) instaliran na sustavu. Alat posjeduje dvojak načina rada, odnosno bez i s grafičkim sučeljem (GUI, *eng. graphical user interface*) (slika 9.).



Slika 9: Sučelje Apache Jmeter (izvor: Autor)

Nakon pokretanja JMeter alata, potrebno je stvoriti dvije *thread* grupe, jedna za Python servis, a druga za Rust servis. Unutar *thread* grupe nalaze se HTTP zahtjevi za svaku od ruta u servisu. Za dodavanje HTTP zahtjeva potrebno je unijeti URL i odgovarajuće parametre. Kako bi testiranje bilo što realističnije, svi parametri su nasumično izabrani. Za nasumične vrijednosti količine artikala u košarici i imena artikala korištena je `RandomVariable` funkcionalnost.

Kako bi svaki zahtjev bio ispravan, ID artikla i kôd kupona moraju biti valjani, stoga, umjesto nasumičnog odabira, podaci se uzimaju iz .csv datoteke pomoću `CSV Data Set Config` funkcionalnosti. Za praćenje statistike i rezultata ispitivanja korišteni su JMeter-ovi prisluškivači (eng. *listeners*). Prisluškivači posjeduju mogućnost spremanja dobivenih rezultata u `.csv` datoteke, koje se tada mogu detaljnije analizirati i usporediti. Na slici 10. prikazano je sučelje s rezultatima testiranja Rust instance.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
Item GET all	911656	358	93	2539	87.43	0.00%	337.6/sec	0.00	0.00	.0
Single item...	911619	162	49	1522	24.37	0.00%	337.6/sec	0.00	0.00	.0
Apply Cou...	911595	497	72	3420	170.35	0.00%	337.6/sec	0.00	0.00	.0
Get Coupo...	911324	161	49	3040	26.48	0.00%	337.5/sec	0.00	0.00	.0
Get All Co...	911187	299	54	3437	41.95	0.00%	337.5/sec	0.00	0.00	.0
TOTAL	4557381	295	49	3437	155.08	0.00%	1687.7/sec	0.00	0.00	.0

Slika 10: Rezultati Rust instance u JMeter sučelju (izvor: Autor)

4 Rezultati

Cilj ispitivanja bio je usporediti ključne metrike, uključujući vrijeme odaziva, kašnjenje i propusnost sustava, kako bi se ocijenile radne značajke svakog API-ja u različitim uvjetima opterećenja. Testiranja su provedena koristeći alat JMeter, koji je slao zahtjeve prema svakom API-ju pod istim uvjetima opterećenja. Mjerene su sljedeće metrike:

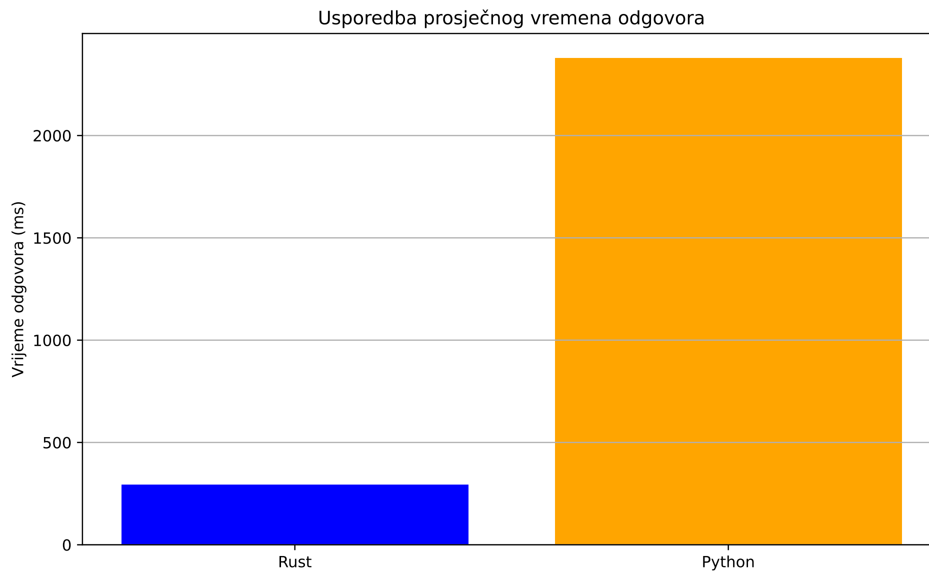
- Vrijeme odaziva (eng. *response time*): ukupno vrijeme od slanja zahtjeva do zaprimanja odgovora
- Kašnjenje (eng. *latency*): vrijeme potrebno API-ju da započne obradu zahtjeva
- Propusnost (eng. *throughput*): broj zahtjeva koje API može odraditi u jednoj sekundi

Oba API-ja implementirana su na AWS EC2 instancama iste konfiguracije kako bi se osigurali identični uvjeti ispitivanja.

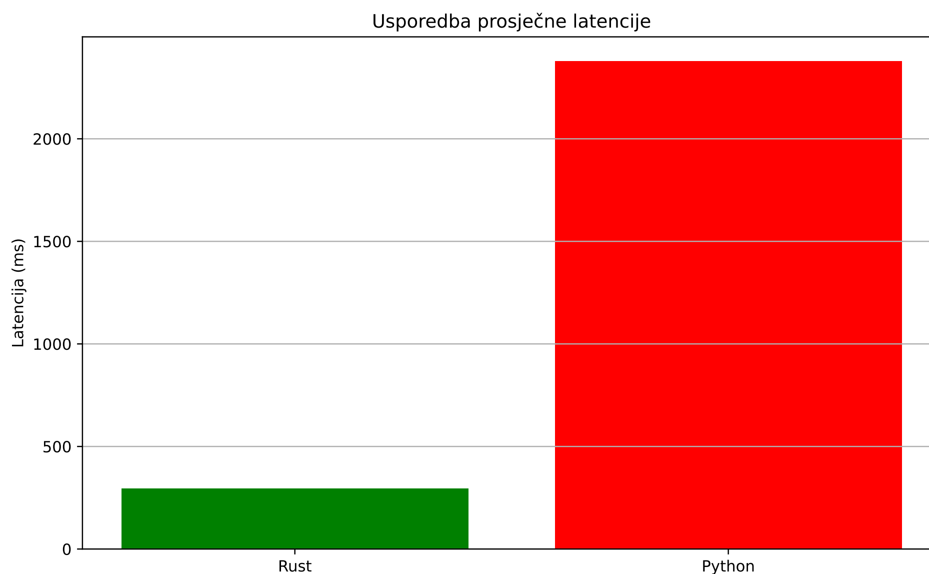
Metrika	Rust API	Python API
Vrijeme odaziva (ms)	295.27	2379.61
Kašnjenje (ms)	295.03	2378.11
Propusnost (req/s)	1691.71	210.22
CPU%	100%	80%
Uspješnost zahtjeva	100%	100%

Tablica 3: Tablični prikaz rezultata ispitivanja ključnih metrika. Mjerne jedinice: ms - milisekunda, req/s - zahtjev/sekunda. Kratice: CPU% - iskorištenost procesora

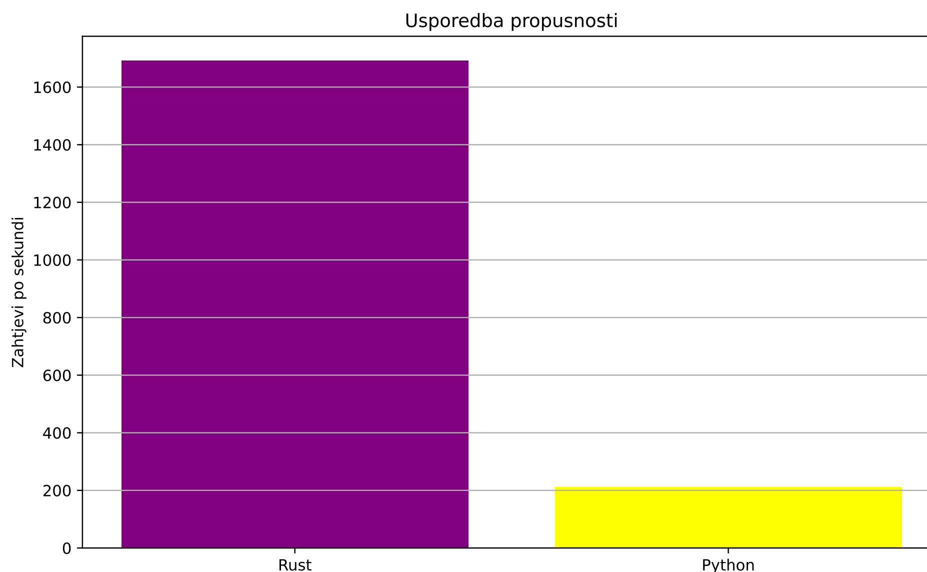
Analizom rezultata (tablica 3.) vidljivo je da je prosječno vrijeme odaziva Rust servisa koje iznosi 295.27 ms znatno kraće od Python servisa, čije vrijeme odaziva iznosi 2379.61 ms. Rezultat ukazuje na činjenicu da Rust servis ima značajnu prednost nad Python servisom u brzini obrade zahtjeva. Rust API također pokazuje niže prosječno kašnjenje (295.03 ms) u odnosu na Python API (2378.11 ms). Navedena metrika posebice je važna u kontekstu sustava u kojem je brzina reakcije ključna. Nadalje, Rust postiže gotovo dvostruko veću propusnost (1691.71 req/s) u usporedbi s Python-om (210.22 req/s) što pokazuje bolju sposobnost rukovanja velikim brojem istovremenih zahtjeva. Prikaz rezultata pojedinih ispitivanja dostupan je na sljedećim grafičkim prikazima (slika 11., slika 12., slika 13.)



Slika 11: Grafički prikaz vremena odaziva u milisekundama(izvor: Autor); Plava: Rust instanca; Žuta: Python instanca; Y os: Vrijeme odaziva u ms

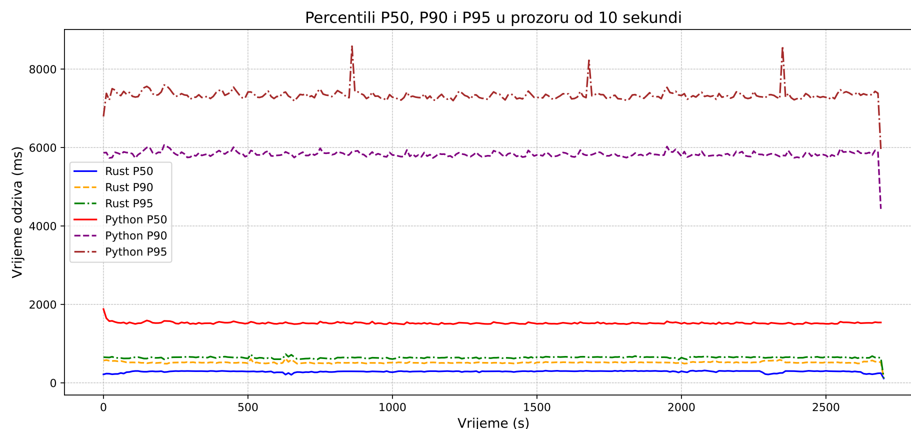


Slika 12: Grafički prikaz kašnjenja u milisekundama (izvor: Autor); Zelena: Rust instanca;Crvena: Python instanca;Y os: Kašnjenje u ms



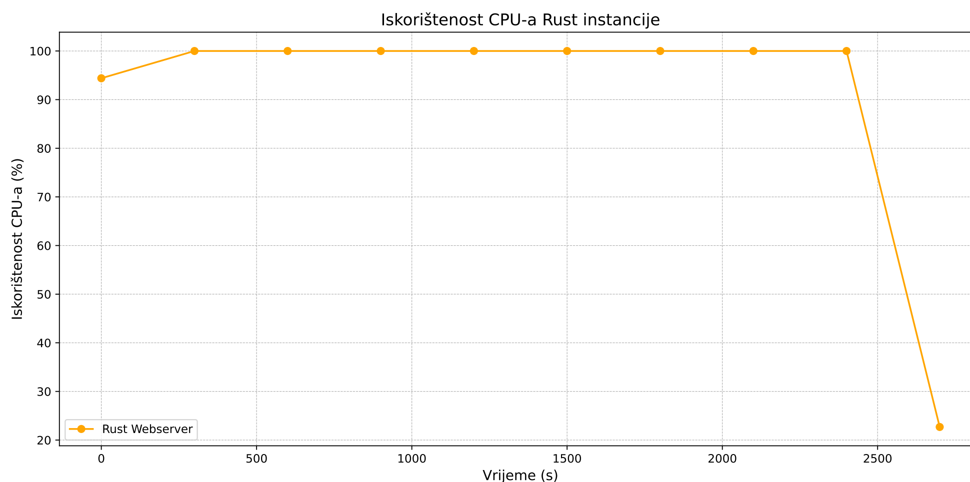
Slika 13: Grafički prikaz propusnosti u zahtjevima po sekundi (req/s) (izvor: Autor); Ljubičasta: Rust instanca; Žuta: Python instanca

Graf na slici 14. predstavlja distribuciju radnih značajki kroz proteklo vrijeme. Rust API ima znatno niži medijan (P50 percentil) naspram Python API-ju što pokazuje da Rust API brže odgovara na većinu zahtjeva. Rust API također ima niži devedeseti percentil (P90, vrijeme potrebno za obradu 90% zahtjeva) što ukazuje na činjenicu da Rust API posjeduje bolje performanse u lošijim scenarijima u usporedbi s Python API-jem. Devedeset i peti se percentil (P95) usredotočuje na najsporijih 5% zahtjeva što posljednično pomaže u detekciji *outlier*-a (ekstremno sporih zahtjeva). Rust API ima znatno niži P95 percentil naspram Python API-ju. Python-ovi P90 i P95 percentili su značajno viši, što sugerira veću nestabilnost i sporije vrijeme odaziva za zahtjeve pod većim opterećenjem. Rust API pokazuje manju disperziju u vrijednostima percentila, što znači da ima dosljedniji odaziv, dok Python API ima znatno veće fluktuacije i P95 vrijednost.

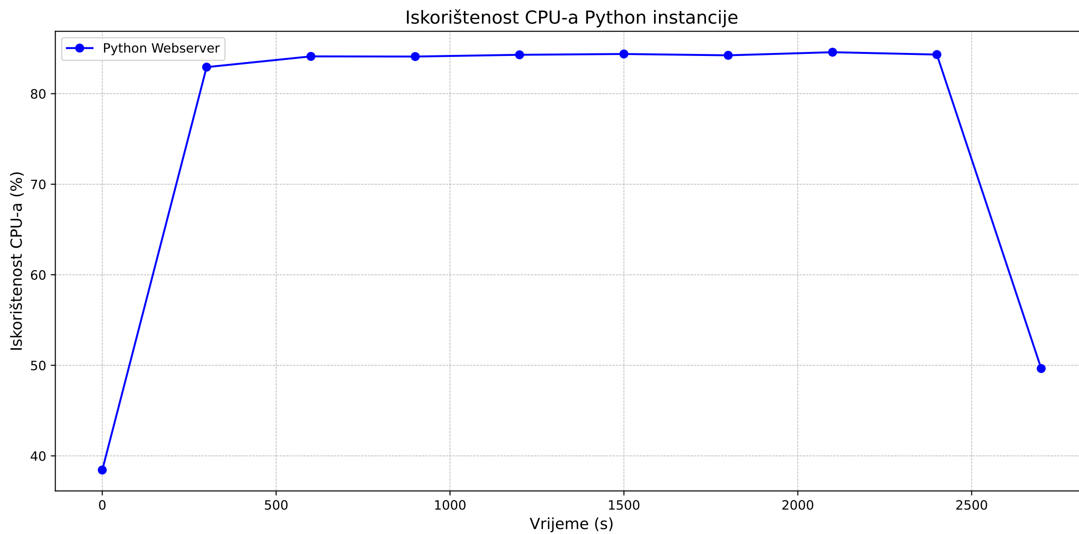


Slika 14: Distribucija radnih značajki kroz vrijeme (izvor: Autor) Plava linija: Rust P50; Žuta linija: Rust P90; Zelena linija: Rust P95; Crvena linija: Python P50; Ljubičasta linija: Python P90; Smeđa linija: Python P95

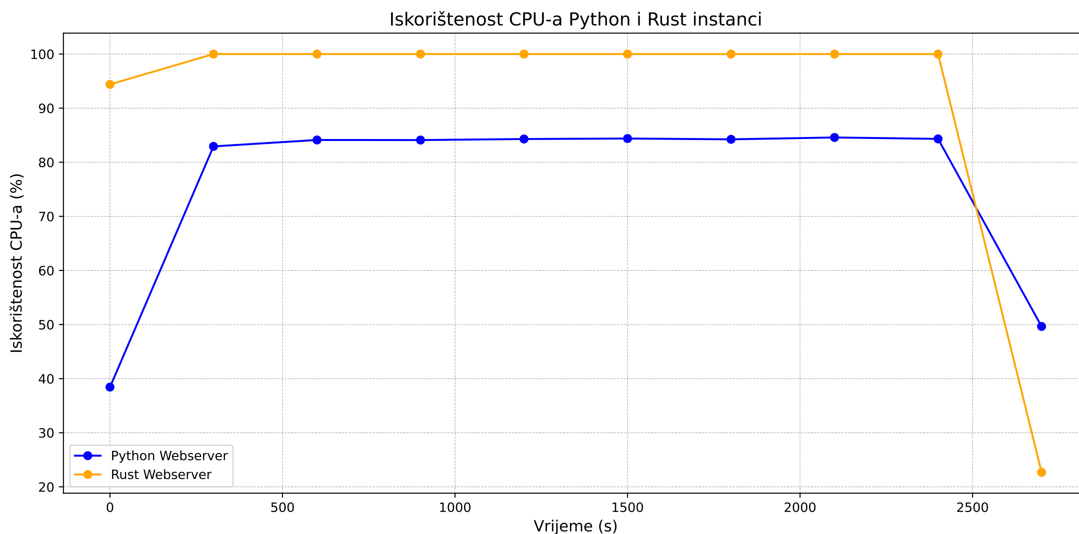
Grafički prikazi dostupni na slikama 15. i 16. prikazuju iskorištenost procesora instance. Rust instanca pri početku ispitivanja pokazuje rast do 100% iskorištenosti procesora, trend koji se bilježi sve do kraja ispitivanja te konačno pada ispod 30% po njegovu završetku. Python instanca, za razliku od Rust instance, naglo raste na 82% iskorištenosti procesora pri početku ispitivanja, oscilira kroz zadani vremenski period oko vrijednosti od 84% iskorištenosti do samog kraja ispitivanja, gdje konačno pada ispod 50% iskorištenosti. Graf na slici 17 prikazuje kombinirani graf obiju instanci kako bi se bolje vidjela usporedba instanci.



Slika 15: Graf iskorištenosti procesora za Rust instancu (izvor: Autor)



Slika 16: Graf iskorištenosti procesora za Python instancu (izvor: Autor)



Slika 17: Graf iskorištenosti procesora za obije instance (izvor: Autor); X os: Postotak iskorištenosti; Y os: Vrijeme u sekundama; Plava linija: Python instanca; Narančasta linija: Rust instanca

Rust API pokazao je superiorne performanse u pogledu vremena odziva, latencije, iskorištenosti procesora i propusnosti, dok je Python API, unatoč jednostavnijem razvoju i fleksibilnosti, imao znatno slabije rezultate pod većim opterećenjem.

5 Rasprava

Brzina i pouzdanost glavne su vrline mrežnih servisa današnjice, stoga je veoma važno pomno odabrati programski jezik koji pruža najbolje radne značajke. Ovo istraživanje usredotočeno je na dva programska jezika različitih karakteristika koji se koriste za razvoj web aplikacija, Rust i Python. Svaki od tih jezika ima svoje prednosti i nedostatke koji utječu na njegovu korisnost i svrhu.

Razvijanje servisa u Python okruženju značajno je jednostavnije i preglednije naspram razvoja u Rust-u, koji zahtijeva podosta striktnu sintaksu i posjeduje rigorozniju skupinu pravila. U više navrata, prilikom pisanja kôda za Rust servis, zatekao bih se pregledavajući dokumentaciju, tražeći načine za što jednostavnijom implementacijom određenih funkcionalnosti, kao i boljim načinima pretvaranja pojedinih tipova podataka. Krivulja učenja Rust jezika veoma je strma, no njegove radne značajke opravdavaju težinu učenja. Također, način na koji Rust uvozi module drukčiji je od ostalih programskih jezika poput Python-a, C++ -a i slično, stoga prilagodba i učenje dodatno oduzimaju vrijeme što pridodaje težini jezika. Ako bismo stavili brzinu programiranja oba servisa u kontekst postotaka, za razvoj u Rust servisu bilo je potrebno izdvojiti oko pedeset posto više vremena.

Jedan od glavnih problema prilikom razvijanja oba servisa činilo je asinkrono procesiranje zahtjeva u Python-u. Pošto biblioteka `boto3`, koja služi interakciji s AWS servisima, ne podržava asinkrone operacije bez korištenja biblioteka poput `asyncio` za asinkrono izvršavanje, problem je riješen korištenjem `aioboto3` biblioteke dizajnirane za asinkrone zahtjeve prema AWS servisima. Nakon implementacije biblioteke, radne značajke servisa značajno su se povećale što je posljedično zahtijevalo potpunu izmjenu pozivanja kôda.

Još jedan problem nastao je prilikom kompiliranja Rust projekta na instanci zbog slabijih radnih značajki. Kompilacija servisa dosegla bi određenu biblioteku čija je obrada bila zahtjevnija, ali ključna za rad servisa, nakon čega se servis jednostavno zamrznuo i konačno srušio. Ponovno pokretanje instance nije riješilo navedeni problem, stoga se, umjesto nadogradnje instance koja bi znatno povećala troškove samog servisa, dalo prednost Rust servisu nad Python servisom u vidu radnih značajki. Rust servis bio je kompiliran na vlastitom računalu te kasnije pokrenut na instanci. Kompilacija je postavljena kako bi odgovarala operativnom sustavu instance (Linux/UNIX).

Prilikom ispitivanja ključnih metrika često je dolazilo do nepredviđenih komplikacija s postotkom uspješnih i neuspješnih zahtjeva. Rust servis bi redovito dobivao određen postotak neuspješnih zahtjeva i značajan rast u kašnjenju (100 puta veći od prosječnog kašnjenja). Uzrok kašnjenja bila je preopterećenost sustava (1000+ korisnika u sekundi) što se naknadno ispravilo smanjenjem broja istovremenih korisnika (dretvi) na 500. Vrlo je važno napomenuti da je Rust servis i dalje pružao konkurentne rezultate unatoč izuzecima. Navedeni problem ostavlja otvorena vrata daljnjim ispitivanjima u kojima bi, hipotetski, postojale instance raznih jačina te koje bi prikazivale time bolju sliku o sposobnosti Rust jezika kao vodećeg izbora u razvoju API-ja.

Promatrajući rezultate ispitivanja da se naslutiti da Rust ima naizgled dvostruko bolje vrijednosti radnih značajki u svim ispitanim područjima. Rezultat ispitivanja iskorištenosti procesora daje do znanja da Rust, za razliku od Python-a, koristi kapacitet CPU-a u potpunosti. Navedena činjenica ukazuje na sposobnost Rust-a u korištenju svih raspoloživih resursa sustava što se može objasniti preko Python-ovog korištenja interpreterskih modela izvršavanja što posljedično ostavlja određeni dio CPU-a neiskorištenim. Rust također posjeduje stabilnije vrijeme odaziva i obrade zahtjeva u usporedbi s Python-om što ukazuje na bolju sposobnost rukovanja velikim brojem zahtjeva bez značajnog pada u performansama. Naizgled jedini način na koji Python kao jezik za razvoj mrežnih servisa može sustići Rust u performansama je da na neki način pruži značajnu prednost u vrijednostima radnih značajki.

Također, postoje načini pomoću kojih bi se istraživanje moglo unaprijediti, no oni nisu obrađeni u kontekstu ovoga rada. Prvi način podrazumijevao bi uvođenje šireg spektra ispitnih scenarija koji bi sadržavali dinamičko opterećenje. Dinamičko opterećenje bi znatno pomoglo u ispitivanju otpornosti sustava na iznenadne promjene prometa. Drugi način odnosio bi se na dodavanje većih količina bazi podataka, primjerice PostgreSQL ili MongoDB, kako bi se procijenile prenosivost i razlike u radnim značajkama. Treći način podrazumijevao bi detaljnu analizu kôda uz pomoć alata poput perf za Rust i cProfile za Python što bi potencijalno olakšalo identifikaciju uskog grla unutar kôda. Isto tako, može se uključiti još nekoliko dodatnih jezika, poput Java i C#.

Konačno, temeljem cjelokupnog istraživanja, dobivenih rezultata i osobnog iskustva u razvijanju web aplikacija, jezik koji bih odabrao ovisi o samoj upotrebi aplikacije. Ako aplikacija zahtijeva visoku skalabilnost i pouzdanost, kao u slučaju, primjerice, financijskih aplikacija i aplikacija koje svoju upotrebu vide u području medicine, Rust bi predstavljao idealan izbor za njihov razvitak. Ako pak aplikacija zahtijeva konstantni daljnji razvitak, u vidu redovite nadogradnje i dodavanja novih značajki, a same radne značajke ne čine kritični, te time i najvažnije osobitosti sustava, Python bi predstavljao idealan izbor za razvitak takvih aplikacija.

6 Zaključak

U ovom diplomskom radu istraživana je izvedba mrežnih servisa razvijenih u programskim jezicima Python i Rust s ciljem usporedbe njihovih performansi u oblaku. Eksperimenti su provedeni korištenjem JMeter alata za testiranje opterećenja, dok su API-jevi bili implementirani koristeći FastAPI za Python i Actix za Rust programski jezik. Analiza je obuhvatila ključne metrike, uključujući vrijeme odgovora, latenciju, propusnost sustava i iskorištenost procesorskih resursa.

Rezultati testiranja ukazali su na značajne razlike u performansama između dviju tehnologija. Rust API ostvario je znatno bolje rezultate u svim mjerljivim metrikama. Rust prema tome pokazuje superiorniju skalabilnost sustava pod velikim opterećenjima što ga čini pogodnijim za razvoj sustava koji zahtijevaju visoku pouzdanost i optimalnu obradu velikog broja zahtjeva.

Unatoč prednostima Rust-a u pogledu performansi, implementacija u Python-u pokazala se jednostavnijom i bržom što ga čini pogodnijim za projekte gdje je fleksibilnost važnija od maksimalne optimizacije. Tijekom razvoja bio sam suočen s mnogim izazovima, uključujući asinkrono rukovanje zahtjevima u Python-u, kao i učenjem Rust sintakse, no oba rješenja su, konačno, postigla zadovoljavajuće rezultate potrebne za njihovu praktičnu primjenu.

Temeljem cjelokupnosti dobivenih rezultata može se zaključiti da Rust predstavlja bolji izbor u razvoju mrežnih servisa visokih performansi, osobito u scenarijima gdje su brzina, skalabilnost i optimizacija resursa ključni. Python, s druge strane, ostaje prikladniji za brži razvoj i jednostavnije održavanje što ga čini pogodnim jezikom za razvoj aplikacija kod kojih performanse nisu primarna briga. U kontekstu budućih istraživanja bilo bi korisno proučiti daljnju optimizaciju Python API-ja, primjerice implementacijom višestrukih *worker* procesa, analizom performansi različitih okruženja u oblaku, uključujući AWS, Google Cloud i Azure te ispitivanjem dugoročne stabilnosti i potrošnje resursa u oba rješenja u produkcijskim uvjetima.

Ovo istraživanje pruža konkretne smjernice za odabir tehnologije u razvoju mrežnih servisa u oblaku te može poslužiti kao polazišna točka za daljnje optimizacije i usporedbe modernih programskih jezika i razvojnih okvira.

Literatura

[1] RedHat. What are cloud services? *https://www.redhat.com/en/topics/cloud-computing/what-are-cloud-services*. [pristupljeno 2.12.2024].

[2] Abelson H. Garfinkel, S. Architects of the information society : thirty-five years of the laboratory for computer science at mit. *History*, page 1, 1999.

[3] B. O'Connell. History of salesforce: Timeline and facts. *TheStreet*, Feb 2020.

[4] Luo Z. Du Y. Guo L. Qian, L. Cloud computing: An overview. volume 5931, pages 626–631, 2009. [10.12.2024].

[5] Announcing amazon elastic compute cloud (amazon ec2) - beta. *https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2—beta/*, 2006. [pristupljeno 12.12.2024].

[6] N. B. Ruparelia. *Cloud Computing*. The MIT Press, 08 2023.

[7] Global infrastructure - aws. *https://aws.amazon.com/about-aws/global-infrastructure/*. [pristupljeno 2.12.2024.].

[8] Amazon Web Services. Amazon dynamodb features. *https://aws.amazon.com/dynamodb/features/*. [pristupljeno 15.2.2025.].

[9] Amazon ec2 features - amazon web services. *https://aws.amazon.com/ec2/features/*. [pristupljeno 5.12.2024.].

[10] What is amazon ec2 auto scaling? - amazon ec2 auto scaling. *https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html*. [pristupljeno 9.12.2024].

[11] B. Venners. The making of python a conversation with guido van rossum, part i. *https://www.artima.com/articles/the-making-of-python*, Jan 2003.

[12] A. Šubert. Rest api: Uvod, Jun 2023. [pristupljeno 7.12.2024.].

[13] Mabunda N. E. Ali A. Mabotha, E. Performance evaluation of a dynamic restful api using fastapi, docker and nginx. *2024 Global Energy Conference (GEC)*, pages 174–181, 2024.

[14] C. Thompson. How rust went from a side project to the world's most-loved programming language. *https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/*, Feb 2023. [pristupljeno 10.12.2024].

[15] P. Jansen. Tiobe index for december 2024. *https://www.tiobe.com/tiobe-index/rust/*. [pristupljeno 10.12.2024].

[16] Rizaldy A. Hartono N. Harwalis H. Nur Hidayat, A. M. Pengujian kinerja web server elastic cloud compute (ec2) free tier pada amazon web service (aws) menggunakan jmeter. *Jurnal Sistem Informasi dan Informatika (Simika)*, 2024.

[17] Ginanjar A. Ihsan Y. Hendayun, M. Analysis of application performance testing using load testing and stress testing methods in api service. *Jurnal Sisfotek Global*, 2023.

Popis slika

1 Početna stranica DynamoDB servisa (izvor: https://aws.amazon.com/dynamodb/) 4
2 Početna stranica EC2 servisa (izvor: https://aws.amazon.com/ec2/) 5
3 UML Dijagram modela podataka (izvor: Autor) 9
4 Dijagram korištenja sustava (UML Use case Dijagram) (izvor: Autor) 10
5 Struktura projekta (izvor: Autor) 12
6 Prikaz strukture stvorenog projekta (izvor: Autor) 22
7 Glavno sučelje AWS EC2 (Dashboard) (izvor: Autor) 32
8 Web sučelje AWS DynamoDB (izvor: Autor) 33
9 Sučelje Apache Jmeter (izvor: Autor) 34
10 Rezultati Rust instance u JMeter sučelju (izvor: Autor) 35
11 Grafički prikaz vremena odaziva u milisekundama(izvor: Autor); Plava: Rust instanca; Žuta: Python instanca; Y os: Vrijeme odaziva u ms 37
12 Grafički prikaz kašnjenja u milisekundama (izvor: Autor); Zelena: Rust instanca;Crvena: Python instanca;Y os: Kašnjenje u ms . . . 37
13 Grafički prikaz propusnosti u zahtjevima po sekundi (req/s) (izvor: Autor); Ljubičasta: Rust instanca; Žuta: Python instanca 38
14 Distribucija radnih značajki kroz vrijeme (izvor: Autor) Plava linija: Rust P50; Žuta linija: Rust P90; Zelena linija: Rust P95; Crvena linija: Python P50; Ljubičasta linija: Python P90; Smeđa linija: Python P95 39
15 Graf iskorištenosti procesora za Rust instancu (izvor: Autor) 39
16 Graf iskorištenosti procesora za Python instancu (izvor: Autor) . . 40
17 Graf iskorištenosti procesora za obije instance (izvor: Autor); X os: Postotak iskorištenosti; Y os: Vrijeme u sekundama; Plava linija: Python instanca; Narančasta linija: Rust instanca 40

Popis tablica

1 Tablični prikaz instanci i njihovih specifikacija (izvor: Autor) 8
2 Prikaz svih vrsta T2 instanci i njihovih radnih značajki (izvor: https://aws.amazon.com/ec2/instance-types/) 8
3 Tablični prikaz rezultata ispitivanja ključnih metrika. Mjerne jedinice: ms - milisekunda, req/s - zahtjev/sekunda. Kratice: CPU% - iskorištenost procesora 36