UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Jovana Bošković

OPTIMIZACIJA PROCESA KONTINUIRANE INTEGRACIJE I ISPORUKE KROZ SELEKTIVNO TESTIRANJE ZASNOVANO NA ANALIZI POKRIVENOSTI KODA

master rad

Beograd, 2024.

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ivan Čukić, docent
Univerzitet u Beogradu, Matematički fakultet

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Optimizacija procesa kontinuirane integracije i isporuke kroz selektivno testiranje zasnovano na analizi pokrivenosti koda

Rezime: Kontinuirana integracija i isporuka (CI/CD) predstavljaju procese u savremenom razvoju softvera koji omogućavaju brzo i efikasno uvođenje promena u proizvodnji softvera. Optimizacija ovih procesa je bitna za poboljšanje produktivnosti timova, smanjenje ciklusa izvršavanja i održavanje visokog kvaliteta koda. Zbog toga, ovaj rad istražuje kako se procesi CI/CD mogu unaprediti kroz selektivno testiranje koje se zasniva na detaljnoj analizi pokrivenosti koda.

Osnovna ideja unapređenja sastoji se iz konstruisanja alata pod nazivom *Select Relevant Tests* koji automatski identifikuje delove koda koji su izmenjeni i selektivno pokreće samo one testove koji su relevantni za te izmene. To vodi ka smanjenju vremena potrebnog za testiranje i omogućava brže integracije i isporuke. Alat koristi softver za merenje pokrivenosti koda *gcov* kako bi prikupio neophodne podatke i stvorio preslikavanje između testova i koda koji se testira. Alat je evaluiran na testovima i promenama projekta *Cppcheck* i prosečna ušteda broja pokretanja testova pri korišćenju alata iznosi 72.86% što dalje pozitivno utiče i na vreme testiranja kao i na sam CI/CD proces.

Ključne reči: kontinuirana integracija i isporuka CI/CD, optimizacija, selektivno testiranje, pokrivenost koda, automatizacija testiranja, alat *gcov*, razvoj softvera

Sadržaj

1 Uvod **1**

2 Kontinuirana integracija i isporuka softvera **3**

2.1 Uloga automatizacije u CI/CD procesima 5

2.2 Značaj i koncepti pokrivenosti koda 7

2.3 Pregled postojećih alata i praksi 9

3 Analiza i dizajn alata *Select Relevant Tests* **15**

3.1 Funkcionalni i nefunkcionalni zahtevi 15

3.2 Opis implementacije alata 16

3.3 Prikaz integracije alata u projekat otvorenog koda 26

4 Evaluacija i testiranje **29**

4.1 Metodologija i proces evaluacije alata 29

4.2 Analiza rezultata i efikasnosti alata 33

5 Zaključak **38**

Bibliografija **40**

Glava 1

Uvod

Kontinuirana integracija i isporuka (CI/CD) predstavljaju procese u savremenom razvoju softvera koji omogućavaju brzo i efikasno uvođenje promena u proizvodnji softvera. CI/CD je skraćenica koja označava kombinovani proces kontinuirane integracije (eng. *Continuous Integration, CI*) i kontinuirane isporuke ili primene (eng. *Continuous Delivery/Deployment, CD*). Primena ovih procesa ključna je za moderni razvoj softvera, jer omogućava timovima da kontinuirano integrišu i isporučuju softverske proizvode uz visok stepen automatizacije [36].

CI/CD proces počinje integracijom novog koda koji programeri redovno spajaju sa centralnim repozitorijumom. Svaka integracija pokreće niz automatizovanih prevođenja i testova, čime se osigurava da je svaka promena u kodu testirana. Ovaj pristup ne samo da smanjuje rizik od kompleksnih konflikata u kodu, već i omogućava brzo otkrivanje i ispravljanje grešaka. Takođe, ovaj pristup razvoju softvera rezultira u značajnom ubrzanju procesa puštanja proizvoda na tržište. U današnje vreme, tržište se konstantno menja, a sposobnost da se brzo odgovori na te promene može biti odlučujuća za uspeh jedne kompanije.

CI/CD omogućava timovima da često i bez odlaganja dostavljaju nove funkcionalnosti i ispravke, čime se kompanijama pruža mogućnost da očuvaju svoju konkurentnost i brzo odgovore na potrebe korisnika. Pravilno implementirani CI/CD procesi smanjuju rizik od grešaka koje nastaju zbog ljudskog faktora, zahvaljujući automatizovanim testovima i stalnim integracijama, što osigurava kontinuiranu proveru i poboljšanje kvaliteta koda.

Integracijom koda i njegovim kontinuiranim testiranjem, problemi se otkrivaju i rešavaju dok su još uvek mali, što sprečava njihovu eskalaciju u veće izazove koje bi bilo teže i skuplje rešiti. U skladu sa ovim principima, optimizacija ovih procesa je bitna za poboljšanje produktivnosti timova, smanjenje ciklusa izvršavanja i održavanje visokog kvaliteta koda.

U okviru ovog rada, razvijen je alat koji ima za cilj da unapredi procese kontinuirane integracije i isporuke. Osnovna ideja alata je da omogući selektivno testiranje, odnosno testiranje koje je fokusirano na testiranje koje je povezano sa izmenjenim delovima koda.

Integracijom sa alatom *gcov* [32], opisano rešenje omogućava brzo prepoznavanje relevantnih testova koji se automatski izvršavaju, čime se unapređuje efikasnost razvojnog ciklusa. Alat je dizajniran tako da se lako uklapa u platformu *Jenkins* [7], što omogućava njegovu široku primenu i jednostavnu integraciju u različite radne procese. Razvoj ovog alata nije samo tehničko unapređenje — on je odgovor na zahtev za većom agilnošću i sposobnošću prilagođavanja brzim promenama.

Tekst teze je organizovan na sledeći način. U poglavlju 2 detaljno su razmotreni procesi kontinuirane integracije i isporuke i njihova važnost u okviru savremenog razvoja softvera, uloga automatizacije u CI/CD procesima i uticaj automatizacije na efikasnost i produktivnost razvojnih timova, kao i značaj i koncepti pokrivenosti koda i pregled postojećih alata. U poglavlju 3 su definisani funkcionalni i nefunkcionalni zahtevi, kao i sama implementacija alata za selektivno testiranje. Predstavljen je proces implementacije, uključujući tehničke detalje koji su važni za razumevanje kako alat funkcioniše u praksi i kako se njegova implementacija odražava na efikasnost razvojnog toka. Takođe je opisana integracija alata *gcov* i *Jenkins*, i prikazano kako se ovi alati mogu uspešno kombinovati da podrže i unaprede CI/CD ciklus. Na kraju je predstavljen i primer kako se alat integriše u postojeći projekat otvorenog koda. U poglavlju 4 je opisana metodologija i procesi koji se koriste za procenu alata, kriterijumi uspešnosti, kao i evaluacija njegovih performansi i koristi koje donosi. U poslednjem poglavlju sumirani su ključni uvidi i rezultati dobijeni tokom istraživanja, zajedno sa izazovima koji su se pojavili u procesu. Pored toga, predstavljene su preporuke za potencijalna poboljšanja i dalji razvoj konstruisanog rešenja.

Glava 2

Kontinuirana integracija i isporuka softvera

Kontinuirana integracija i isporuka (CI/CD) su temelji savremenog pristupa razvoju softvera koji omogućavaju timovima da automatizuju testiranje i isporuku softvera, smanjujući time rizik i ubrzavajući ciklus isporuke. Razumevanje ovih procesa pomaže nam da shvatimo kako moderni razvojni timovi postižu brz tempo razvoja dok istovremeno održavaju kvalitet i stabilnost svojih aplikacija [36].

Proces kontinuirane integracije i isporuke predstavlja skup koraka koji su dizajnirani da poboljšaju razvojni ciklus softvera, od trenutka kada se kôd piše pa sve do njegovog puštanja u rad u produkcijskom okruženju. Glavni cilj ovog procesa je da omogući brzo i efikasno uvođenje izmena u kodu, dok se istovremeno održava visok standard kvaliteta softverskih proizvoda. CI/CD uključuje kontinuirane procese automatizacije i konstantno praćenje kroz sve faze ciklusa aplikacije — od integracije i testiranja, preko dostavljanja, do konačne isporuke u produkcijsko okruženje. Ovi procesi, koji se često nazivaju „CI/CD lanac" (eng. *CI/CD pipeline*), su ključni za timove koji rade na razvoju i održavanju softvera u okviru agilnih metodologija.

Kontinuirana integracija

Kontinuirana integracija je pristup u razvoju aplikacija koji omogućava programerima da istovremeno rade na različitim delovima iste aplikacije. Umesto da

se sve promene u kodu objedinjuju na dan kada se vrši spajanje (često nazivan *merge day*), što može biti veoma vremenski zahtevno, CI omogućava da se izmene koda redovno i automatski integrišu. To znači da se kôd koji programeri razvijaju u izolovanom okruženju redovno spaja sa centralizovanom bazom koda, a automatizovani procesi prevođenja i testiranja potvrđuju valjanost promena. Time se sve funkcije, klase i moduli proveravaju, osiguravajući da nove izmene ne narušavaju funkcionalnost aplikacije. Ako se pojave bilo kakvi sukobi ili greške, CI olakšava njihovo brzo otkrivanje i ispravljanje.

Kontinuirane isporuke/primene

Nakon uspešne integracije i validacije koda kroz CI, kontinuirana isporuka preuzima proces sa ciljem da automatizuje puštanje proverenog koda iz repozitorijuma u testno ili proizvodno okruženje. CD osigurava da je kôd stalno spreman za puštanje u produkciju, smanjujući vreme koje je potrebno da se promene učine dostupnim korisnicima. Automatizovani testovi i procesi isporuke koda deo su ove faze, omogućavajući operativnom timu da brzo i efikasno implementira aplikaciju u proizvodno okruženje.

Poslednji korak, kontinuirana primena, predstavlja nadogradnju na kontinuiranu isporuku i podrazumeva automatsko puštanje aplikacije u produkciju. U praksi, to znači da bilo koja promena napravljena od strane programera može biti puštena u javnost u roku od nekoliko minuta nakon što prođe sve etape automatizovanog testiranja. Ova brzina puštanja omogućava kontinuiranu adaptaciju i integraciju povratnih informacija od korisnika, čineći proces razvoja aplikacije agilnijim i sa manjim rizikom od grešaka u produkciji. Promene u aplikaciji se objavljuju postepeno, umesto sve odjednom, što dodatno smanjuje rizik. Ovaj pristup zahteva značajna ulaganja u razvoj i održavanje automatizovanih testova koji podržavaju različite faze testiranja i isporuke unutar CI/CD procesa.

Automatizacija testiranja je ključni element CI/CD procesa, jer omogućava brzo i kontinuirano testiranje koda, što je neophodno za efikasno isporučivanje softvera. Automatizovani testovi obezbeđuju da se greške otkriju i isprave ranije, čime se smanjuju troškovi i ubrzava razvojni ciklus. Pokrivenost koda je jedna od metrika koja pokazuje koliko je koda testirano. Visoka pokrivenost je indikator

da je softver temeljno proveren i spreman za produkciju. Merenje pokrivenosti testovima pruža uvid u obim koda koji je prošao kroz proces verifikacije pomoću automatizovanih testova.

2.1 Uloga automatizacije u CI/CD procesima

Automatizovano testiranje je proces upotrebe softverskih alata koji izvršavaju seriju testova na novorazvijenom softveru ili određenoj verziji kako bi se identifikovale potencijalne greške u kodu, uska grla i druge prepreke koje utiču na performanse. Test skripte se razvijaju i pokreću, nakon čega se dobijeni rezultati porede sa predviđenim ishodima [28].

Automatizovani testovi se mogu klasifikovati po tipu testiranja ili fazazama testiranja. Ovakva klasifikacija pomaže u organizovanju i strukturiranju procesa testiranja kako bi se obezbedila provera softvera.

Po tipu testiranje moze biti funkcionalno i nefunkiconalno. Funkcionalno testiranje proverava da li softver ili aplikacija rade u skladu sa zahtevanim specifikacijama, dok nefunkcionalno testiranje meri koliko dobro softver ili aplikacija rade.

Postoje različite vrste testova koji se koriste za proveru softvera, a klasifikacija može varirati u zavisnosti od izvora. U neke od često korišćenih tipova testova spadaju testovi poverenja, integracioni i regresioni testovi, kao i testovi bezbednosti, performansi i prihvatanja. [35].

Testovi poverenja (eng. *smoke tests*) su vrsta funkcionalnih testova koji se fokusiraju na najkritičnije funkcije softverskog rešenja kako bi se osiguralo da je sistem spreman za dalje detaljno testiranje bez rizika od ozbiljnih problema.

Integracioni testovi (eng. *integration tests*) kombinuju sve pojedinačne komponente i funkcionalnosti softverskog rešenja i testiraju ih zajedno kako bi se osiguralo da integracija različitih delova ne uzrokuje probleme.

Regresioni testovi (eng. *regression tests*) uključuju kombinaciju funkcionalnih i nefunkcionalnih testova koji se koriste za proveru da li su novim izme-

nama unazadili funkcionalnost ili performanse softvera. Cilj je identifikovati nove greške koje su se pojavile nakon ažuriranja ili izmena u kodu.

Testovi bezbednosti (eng. *security tests*) obuhvataju funkcionalne i nefunkcionalne testove koji proveravaju softver na prisustvo bezbednosnih ranjivosti. Oni otkrivaju slabosti i potencijalne tačke za napad u sistemu.

Testovi performansi (eng. *performance tests*) su obično nefunkcionalni testovi koji pomažu u evaluaciji kriterijuma poput brzine odziva i stabilnosti softvera pod različitim uslovima opterećenja i stresa. Testiraju kako softver upravlja sa zahtevima u realnom radnom okruženju.

Testovi prihvatanja (eng. *acceptance tests*) su funkcionalni testovi koji određuju koliko je softver prihvatljiv za krajnje korisnike. Ovo je finalna faza testiranja koju softversko rešenje mora proći pre nego što bude pušteno u upotrebu, kako bi se osiguralo da zadovoljava sve korisničke zahteve i očekivanja.

Testiranje softvera se sprovodi u različitim fazama tokom procesa razvoja i održavanja. Faze testiranja uključuju: testiranje jedinica koda, integraciono i sistemsko testiranje [29].

Testiranje jedinica koda (eng. *unit testing*) je faza testiranja najmanjih delova koda, kao što su pojedinačne funkcije ili klase.

Integraciono testiranje (eng. *integration testing*) je faza u kojoj se pojedinačne komponente, nakon testiranja jedinica koda, spajaju i testiraju zajedno. Cilj je otkriti i rešiti probleme koji nastaju tokom integracije, obezbeđujući da moduli ispravno funkcionišu zajedno.

Sistemsko testiranje (eng. *system testing*) je faza u kojoj se testira celokupan sistem nakon integracije. Cilj je osigurati da softver ispunjava sve zahteve iz specifikacije.

2.2 Značaj i koncepti pokrivenosti koda

Pokrivenost koda je mera koja pokazuje koliki deo izvornog koda je izvršen tokom procesa testiranja. Ona se obično izražava procentualno i koristi se za procenu kvaliteta softverskih testova. Značaj pokrivenosti koda leži u njenoj sposobnosti da pruži merljive podatke o kvalitetu i efikasnosti testiranja softvera [31].

Kriterijumi pokrivenosti koda su zahtevi koje skup testova mora da ispuni tokom testiranja softvera [2]. U softverskom testiranju, postoji više kriterijuma pokrivenosti koji mere adekvatnost testiranja programa. Osnovni tipovi kriterijuma pokrivenosti u softveru uključuju pokrivenost funkcija, naredbi i grana, koji su dalje detaljno predstavljeni [26].

Pokrivenost funkcija je test kriterijum koji kvantifikuje koliko funkcija u testiranom kodu je izvršeno. Pokrivenost funkcija se računa tako što se podeli broj izvršenih funkcija sa ukupnim brojem funkcija u kodu. Pokrivenost funkcija je korisna za proveru da li su sve funkcije izvršene tokom testa, ali ne može osigurati da su sve naredbe unutar tih funkcija izvršene. Drugim rečima, veliki deo koda koji ima potpunu pokrivenost funkcija možda nikada neće biti izvršen tokom procesa testiranja. Stoga se pokrivenost funkcija često smatra slabim kriterijumom pokrivenosti ako se koristi samostalno. Kao rezultat, u softverskom testiranju obično koristimo pokrivenost funkcija sa drugim kriterijumima pokrivenosti kao što su pokrivenost naredbi i pokrivenost grana.

Pokrivenost naredbi je još jedan kriterijum testiranja koji kvantifikuje koliko se naredbi u testiranom kodu izvršava. Računa se kao broj izvršenih naredbi podeljen sa ukupnim brojem naredbi u kodu koji se testira. Potpuna pokrivenost naredbama osigurava da je svaka naredba u kodu izvršena bar jednom. Za razliku od pokrivenosti linija (biće objašnjeno kasnije), koja meri odnos izvršenih linija prema ukupnom broju linija, pokrivenost naredbama je precizniji kriterijum pokrivenosti. Svaka linija koda može sadržati više od jedne naredbe, i sve potencijalne naredbe moraju se uzeti u obzir prilikom izračunavanja pokrivenosti naredbama.

Pokrivenost grana ima za cilj da izmeri koliko su različite grane izvršavanja koda obuhvaćene tokom testiranja. Određuje se tako što se broj testiranih

ishoda grana (tj. različitih puteva koje program može da pređe na osnovu uslovnih izraza) podeli sa ukupnim brojem mogućih ishoda grana koji postoje u programu. Potpuna pokrivenost granama podrazumeva da su tokom testiranja izvršene sve moguće grane u kodu, odnosno svaki uslov i odluka u kodu su testirani tako da su rezultirali i tačnim i netačnim ishodom. Zbog toga je pokrivenost granama praktičniji kriterijum od pokrivenosti naredbama, jer ako se postigne potpuna pokrivenost granama, automatski je postignuta i potpuna pokrivenost naredbama.

Pored pomenutih kriterijuma pokrivenosti, koji osiguravaju da su osnovni elementi koda testirani, važno je pomenuti da postoje i drugi tipovi koji doprinose sveobuhvatnosti testiranja.

Pokrivenost uslova se odnosi na testiranje svakog logičkog uslova unutar odluka u kodu. To znači da se proverava kako se kod ponaša za sve moguće vrednosti tih uslova, osiguravajući da su sve kombinacije istinitih i lažnih vrednosti ispravno obrađene. Pokrivenost grana, s druge strane, osigurava da su sve moguće grane odluka testirane barem jednom. Dakle, pokrivenost uslova dopunjuje pokrivenost grana pružajući uvid u logičke odluke unutar koda.

Pokrivenost putanja osigurava da su sve potencijalne rute izvršavanja kroz kôd temeljno testirane, uključujući složene scenarije sa višestrukim petljama i uslovima.

Pokrivenost linija nudi brzu procenu obuhvaćenosti koda, merenjem koliko je linija koda izvršeno tokom testiranja.

Pokrivenost petlji se fokusira na testiranje petlji sa različitim brojem iteracija, što je posebno važno za proveru graničnih slučajeva i ispravnost obrade kolekcija.

Sve ove metrike zajedno pružaju uvid u to koliko je softver testiran i pomažu u identifikaciji delova koda koji zahtevaju dodatnu pažnju u testiranju.

2.3 Pregled postojećih alata i praksi

U ovom poglavlju biće predstavljeni postojeći alati koji su važni za implementaciju i efikasno funkcionisanje *CI/CD* lanca (eng. *pipeline*), kao i alati koji omogućavaju merenje pokrivenosti koda. Razumevanje ovih alata je značajno za razvoj i primenu alata selektivnog testiranja zasnovanog na analizi pokrivenosti koda.

Alati za kontinuiranu integraciju i isporuku

Alati za kontinuiranu integraciju i isporuku omogućavaju timovima da automatizuju procese razvoja, implementacije i testiranja. Neki od tih alata su specijalizovani za upravljanje kontinuiranom integracijom, dok drugi se fokusiraju na procese razvoja i isporuke. U poznate alate spadaju:

Jenkins [7] je platforma otvorenog koda za kontinuiranu integraciju (*CI*), koja omogućava automatizaciju procesa izgradnje softvera za različite projekte. Kompatibilna je sa svim programskim jezicima i podržava rad na operativnim sistemima, kao što su *Windows*, *Linux* i *macOS*. *Jenkins* će biti opisan detaljno u nastavku jer će se koristiti u dizajnu alata koji je razvijen u toku ove teze.

Bitrise [4] je *CI/CD* platforma namenjena za razvoj mobilnih aplikacija. Omogućava automatizaciju procesa izgradnje, testiranja i isporuke mobilnih aplikacija, podržavajući razne okvire i alate, što olakšava razvojni proces za programere mobilnih aplikacija.

Spinnaker [24] je platforma otvorenog koda namenjena za kontinuiranu isporuku softvera, koja omogućava brzu i sigurnu isporuku softverskih promena. *Spinnaker* pruža jedinstvenu platformu koja omogućava bezbedno raspoređivanje i upravljanje aplikacijom u višestrukim *cloud* okruženjima, uključujući *AWS* [37], *GCP* [8] i *Kubernetes* [3]. Koriste ga organizacije, kao što su *Target* [10], *Airbnb* [1], *Adobe* [27], i mnoge druge.

GoCD [25] je alat otvorenog koda za kontinuiranu isporuku koji omogućava automatizaciju i upravljanje ciklusom izdavanja softvera. Fokusira se na mo-

deliranje i vizualizaciju kompleksnih radnih tokova (eng. *workflows*) i lanaca isporuke (eng. *delivery pipeline*). Omogućava timovima da lako prate i kontrolišu put softvera od izvornog koda do isporuke. Robustan i skalabilan, i pruža timovima alat koji može podržati kako male tako i velike, kompleksne softverske projekte.

Jenkins: detaljan pregled

Jenkins je platforma koja omogućava kontinuiranu integraciju i kontinuiranu isporuku projekata, nezavisno od platforme na kojoj se radi. To je besplatna platforma otvorenog koda koja može upravljati bilo kojom vrstom izgradnje ili kontinuirane integracije. *Jenkins* se može integrisati sa brojnim tehnologijama za testiranje i isporuku [7].

Jenkins omogućava automatizaciju kreiranja, ažuriranja i brisanja poslova u skladu sa repozitorijumima detektovanim u sistemu za upravljanje konfiguracijom softvera. Efikasnost upravljanja poslovima može se poboljšati pravilnim organizovanjem definicija poslova, kako bi se maksimalno iskoristile prednosti automatizacije koje *Jenkins* nudi.

Jenkins podržava tri načina za upravljanje poslovima: korišćenje organizacionih direktirijuma, korišćenje višestrukih grana u lancu i direktno korišćenje lanca. *Jenkins* lanac je skup pluginova koji omogućava modeliranje i automatizaciju procesa kontinuirane isporuke softvera unutar *Jenkins-a*. Definicija lanca se upisuje u *Jenkinsfile*, tekstualnu datoteku koja se čuva u repozitorijumu za kontrolu verzija, omogućavajući verzionisanje i reviziju kao i kod bilo kog drugog dela softvera. Ovo omogućava automatsku izgradnju za sve grane, beleži sve promene i pruža centralizovano mesto gde tim može pregledati i uređivati definicije lanca.

Jenkinsfile se može pisati koristeći dve vrste sintakse: deklarativnu i skriptovanu. Deklarativna sintaksa u *Jenkins-u* omogućava definisanje procesa izgradnje i isporuke kroz jasno strukturiran i lako čitljiv format. Koristi upotrebu blokova koji opisuju faze i korake, pružajući jednostavnost i preglednost. Skriptovana lanac sintaksa koristi *Groovy* skript za fleksibilnost i kontrolu, omogućavajući složenije logičke operacije i prilagođenu logiku unutar lanca. Glavna razlika između njih je u nivou kontrole i složenosti; deklarativna sintaksa je jednostavnija i ograničenija,

dok skriptovana pruža više mogućnosti za prilagođavanje.

Kôd lanca opisuje ceo proces izgradnje, koji obično uključuje faze kao što su izgradnja aplikacije, njeno testiranje i isporuka.

U kontekstu *Jenkins* lanca, *Node*, *Stage* i *Step* imaju specifična značenja:

Node se odnosi na računar u *Jenkins* okruženju koji je sposoban za izvršavanje lanca ili pojedinačnih poslova. *Node* može biti fizički server, virtualna mašina ili čak *docker* kontejner koji ima instaliran *Jenkins* agent.

Stage je deo lanca koji predstavlja logički deo procesa izgradnje i isporuke. Svaki *stage* obično sadrži skup koraka koji se izvršavaju kao deo jedne faze, kao što su izgradnja, testiranje ili isporuka.

Step je najmanja izvršna jedinica u *Jenkins* lancu i predstavlja pojedinačni zadatak koji treba obaviti. Na primer, *step* može biti izvršavanje skripte, prevođenje koda ili pokretanje testova. U lancu, *step*-ovi se određuju unutar faza i mogu iskoristiti različite *Jenkins* dodatke za obavljanje određenih zadataka.

Jenkins nudi i funkcionalnost *Multibranch* lanca. *Multibranch* lanac omogućava *Jenkins*-u da automatski otkrije sve grane u repozitorijumu koje sadrže *Jenkinsfile* i kreira lance za svaku granu. Ova funkcionalnost je posebno korisna za projekte sa više aktivno razvijanih grana, jer omogućava paralelno testiranje i izgradnju za svaku granu.

Blue Ocean je dodatak za *Jenkins* koji pruža vizualizaciju lanca kroz intuitivno korisničko okruženje. Razvijen je s ciljem pojednostavljenja korisničkog iskustva i povećanja preglednosti.

Takođe, *Jenkins* se može koristiti sa *Docker*-om. *Docker* je platforma za pokretanje aplikacija u izolovanim okruženjima nazvanim kontejneri. [21] Ovo omogućava pokretanje *Jenkins* servera u izolovanim kontejnerima, pružajući dodatnu fleksibilnost, konzistentnost i skalabilnost CI/CD procesa.

Za potrebe master rada, biće usvojen pristup korišćenja deklarativnog *Multibranch* lanca. Ovaj metod omogućava jasno strukturisano i lako čitljivo definisanje lanca, pružajući visok nivo fleksibilnosti i kontrole nad procesom upravljanja poslovima. Deklarativni pristup je dobar za situacije koje zahtevaju paralelno testiranje i izgradnju različitih verzija softvera.

Alati za pokrivenost koda

Alati za pokrivenost koda su softverski alati koji se koriste za merenje pokrivenosti koda testovima [6]. Alati tipično funkcionišu tako što instrumentiraju ili analiziraju kôd i njegovo izvršavanje tokom testiranja. Oni prikupljaju podatke i generišu izveštaje koji pokazuju procenat koda koji je izvršen tokom testova.

Proces korišćenja alata za pokrivenost koda obično počinje izvršavanjem pripremljenih testova, koji mogu biti testovi jedinica koda, integracioni testovi ili sistemski testovi. Na osnovu prikupljenih podataka, ovi alati generišu izveštaj o pokrivenosti koda koji može uključivati različite metrike, kao što su broj izvršenih linija koda, grananja, funkcija ili naredbi.

Ako postoje različite vrste testova, alati za pokrivenost koda mogu kreirati posebne izveštaje za svaku vrstu testa i zatim ih kombinovati u jedan zajednički izveštaj. Ovo omogućava sveobuhvatan pregled pokrivenosti koda kroz sve nivoe testiranja, dajući jasnu sliku o tome koje delove koda treba dodatno testirati ili poboljšati. Neki od poznatih alata su navedeni nastavku.

Gcov [32] je alat za merenje pokrivenosti testova koji se koristi za analizu programa zajedno sa prevodiocem *GCC* [22]. *Gcov* se može koristiti i kao alat za profilisanje kako bi se identifikovali delovi koda gde bi optimizacija imala najveći uticaj. Pored toga, *gcov* se može koristiti zajedno sa alatom za profilisanje *gprof* [33], kako bi se procenilo koji delovi koda zahtevaju najviše računarskog vremena. *Gcov* će biti opisan detaljnije jer će se koristiti u dizajnu alata.

Allure TestOps [38] je alat za merenje pokrivenosti koda koji se može koristiti za analizu i dokumentaciju testiranja softvera. Njegove funkcije omogućavaju automatsko generisanje dokumentacije testova, merenje pokrivenosti koda i praćenje napretka testiranja. Ovaj alat omogućava pregled metrika pokrivenosti za individualne test slučajeve i za ukupan skup testova. *Allure TestOps* centralizuje podatke o testiranju, što olakšava saradnju unutar timova i omogućava efikasno praćenje napretka testiranja.

Testwell CTC++ [30] je alat za pokrivenost koda napisanog u jezicima *C*, *C++*, *C#* ili *Java*, koji omogućava detaljnu analizu efikasnosti testiranja. Posebno

se ističe svojim naprednim analitičkim funkcijama, uključujući mogućnost filtriranja i isključivanja delova koda.

Alat *gcov*: detaljan pregled

Alat *gcov* služi za merenje pokrivenosti koda koji generiše informacije koje se mogu koristiti za analizu različitih tipova pokrivenosti koda. *Gcov* dolazi kao pomoćni alat u okviru *GCC* (eng. *GNU Compiler Collection*) i funkcioniše isključivo na kodu koji je preveden pomoću *GCC-a* [26].

Kada se alat *gcov* koristi za analizu pokrivenosti koda, optimalno je da prevođenje koda bude izvršeno bez primene optimizacije. Razlog za to je što proces optimizacije može dovesti do spajanja više linija koda u jednu, što može rezultirati smanjenjem dostupnih informacija za identifikaciju delova koda koji značajno opterećuju sistem. Budući da *gcov* sakuplja statistike na nivou pojedinačnih linija koda, najpreciznije rezultate daje kada se svaka naredba nalazi na svojoj liniji, što olakšava praćenje i analizu izvršavanja svake specifične instrukcije [32].

Instrumentalizacija koda pomoću *gcov-a* u *GCC-u* podrazumeva dodavanje specijalnog koda u prevedeni program koji omogućava prikupljanje podataka i izračunavanje pokrivenosti koda testovima. Kada se koriste određene opcije prevodioca, *GCC* dodaje ovaj kod kako bi se omogućilo praćenje izvršavanja i prikupljanje relevantnih podataka.

Za instrumentalizaciju koda koriste se sledeće opcije *gcc* prevodioca:

-ftest-coverage opcija generiše datoteke sa ekstenzijom *.gcno* koje *gcov* koristi za analizu pokrivenosti programa. Ove datoteke sadrže informacije potrebne za rekonstrukciju grafa osnovnih blokova, uključujući brojeve linija izvornog koda za svaki blok. Tokom prevođenja, prevodilac dodaje brojače za čvorove i grane u grafu kontrole toka (eng. *control flow graph*, skraćeno *CFG*).

-fprofile-arcs opcija dodaje kôd koji kreira datoteke sa ekstenzijom *.gcda* koji sadrže broj izvršavanja svakog osnovnog bloka i grane, kao i neke sažete informacije, pri završetku izvršavanja programa. Prevodilac prati broj izvršavanja osnovnih blokova u realnom vremenu i ažurira brojače u *.gcda* datotekama pri završetku programa.

--coverage opcija je sinonim za kombinaciju opcija *-fprofile-arcs* i *-ftest-coverage*.

Prilikom izvršavanja programa prevedenog sa *-coverage* opcijom, stvaraju se *.gcda* datoteke u direktorijumu objektnih datoteka. Lokacija ovih datoteka se može prilagoditi korišćenjem okolinskih promenljivih *GCOV_PREFIX*, koja dodaje prefiks putanjama, i *GCOV_PREFIX_STRIP*, koja uklanja deo putanje. Na primer, postavke *GCOV_PREFIX*=/target/run i *GCOV_PREFIX_STRIP*=1 usmeravaju *.gcda* datoteke u /target/run/build/. Pre analize *gcov-om*, potrebno je premestiti *.gcda* datoteke na odgovarajuću lokaciju.

Upotrebom podataka iz *.gcno* i *.gcda* datoteka, moguće je izvršiti komandu *gcov* za generisanje statistika o pokrivenosti koda. Rezultat ove operacije je sažetak pokrivenosti koji obuhvata datoteke i funkcije unutar njih, a koji se prikazuje na standardnom izlazu. Dodatno, kreiraju se i detaljni izveštaji o pokrivenosti koda sa *.gcov* ekstenzijom.

Alat *gcov* pruža i dodatni niz opcija za detaljnu analizu pokrivenosti koda. U kontekstu alata koji se razvija u okviru ovog master rada, sledeće opcije alata *gcov* će biti iskorišćene:

- **-n (--no-output)** ova opcija sprečava *gcov* da piše analizirane podatke u standardnu izlaznu datoteku, što može biti korisno ako su potrebne samo *.gcov* datoteke.
- **-f (--function-summaries)** ova opcija prikazuje sažetak pokrivenosti za svaku funkciju, uključujući broj izvršavanja i pokrivenost linija koda unutar funkcije.
- **--demangled-names** ova opcija prikazuje pojednostavljene (eng. *demangled*) nazive funkcija, što je posebno korisno u *C++* kodu gde su nazivi funkcija često zakomplikovani zbog preopterećenja i šablona.
- **-o (--object-directory)** ova opcija omogućava navođenje direktorijuma u kojem se nalaze *.gcno* i *.gcda* datoteke, što je korisno kada nisu u istom direktorijumu kao izvorni kôd.

Glava 3

Analiza i dizajn alata *Select Relevant Tests*

U okviru ovog poglavlja, prvo će biti definisani funkcionalni i nefunkcionalni zahtevi alata. Nakon toga, biće opisana implementacija alata *Select Relevant Tests* i njegova arhitektura. Alat je razvijen u okviru ovog rada i javno je dostupan [5]. Zatim će biti prikazano kako su *gcov* i *Jenkins* integrisani u razvojni tok, kao i kako se projekat može pripremiti za primenu ovog alata.

Alat *Select Relevant Tests* ima za cilj optimizaciju procesa testiranja tako što identifikuje i selektuje relevantne testove potrebne za pokrivanje izmenjenih delova koda u zahtevima za izmene koda. Alat analizira promene u *PR*-ovima i bira testove koji su direktno povezani sa izmenjenim funkcijama, čime se smanjuje broj izvršenih testova.

3.1 Funkcionalni i nefunkcionalni zahtevi

Alat *Select Relevant Tests*, razvijan kao deo ovog rada, treba da bude dizajniran tako da se lako integriše u postojeće CI/CD okruženje, omogućavajući automatski odabir i izvršavanje testova koji su direktno povezani sa nedavnim promenama u kodu. Takođe, alat treba da se lako poveže sa sistemima za upravljanje verzijama, kao što je *Git* [9], kako bi pratio promene i odredio relevantne testove. Važno je da projekat u koji se alat za selektivno testiranje integriše ima mogućnost prevođenja

sa uključenom opcijom za praćenje pokrivenosti koda.

Sa aspekta nefunkcionalnih zahteva, performanse alata za selektivno testiranje su od velikog značaja. Neophodno je da alat funkcioniše brzo i efikasno kako ne bi došlo do usporavanja CI/CD procesa, posebno u situacijama kada se obrađuju obimni skupovi podataka ili se izvršava veliki broj testova. U skladu s tim, alat mora demonstrirati visok nivo skalabilnosti, sposobnost da se prilagodi i održi performanse uprkos rastućem obimu koda i broju testova koji su karakteristični za projekte koji se šire.

Pouzdanost je još jedan faktor: alat mora generisati konzistentne i tačne rezultate kako bi korisnici mogli da se oslone na njega u svakodnevnom radu. Održivost alata osigurava da se može lako ažurirati i nadograđivati, čime se obezbeđuje njegova dugoročna upotrebljivost i minimalizacija prekida u radu, što je neophodno za održavanje neprekidnog razvojnog ciklusa.

3.2 Opis implementacije alata

Alat *Select Relevant Tests* se sastoji od dva ključna dela: CI/CD tokova i samog alata. CI/CD tokovi omogućavaju integraciju alata u postojeće procese kontinuirane integracije i isporuke, dok sam alat optimizuje proces testiranja.

Implemetacija CI/CD tokova

Da bi se alat koristio, prvo je potrebno implementirati CI/CD tokove. Ovi tokovi se sastoje od dva glavna dela: periodičnog toka i *Pull Request* toka. Svaki od njih ima specifične korake i ciljeve koji osiguravaju efikasnu i pouzdanu automatizaciju testiranja i isporuke softvera. *Jenkins* je implementiran korišćenjem *Docker* plugina, sa lokalnim *Docker* [21] okruženjem koje koristi operativni sistem *Debian Bookworm* [34]. Svi *Jenkins* tokovi se izvršavaju na ovom okruženju.

Periodični tok

Periodični tok se pokreće redovno, prema unapred definisanom rasporedu (npr. jednom dnevno ili jednom nedeljno), kako bi se prikupljali podaci o pokrivenosti koda.

Jenkinsfile definiše *CI/CD* lanac za periodični tok koji uključuje faze: pripremu radnog prostora, preuzimanje koda iz *Git* repozitorijuma, prevođenje projekta, izvršavanje testova i prikupljanje podataka o pokrivenosti koda.

Prva faza, *Prepare Workspace*, osigurava da radni prostor bude čist pre nego što se preuzmu i izgrade novi elementi projekta. Koristi se funkcija *deleteDir()* koja briše sve datoteke i direktorijume iz trenutnog radnog prostora. Ovo je bitno za izbegavanje ostataka od prethodnih izgradnji. Listing 3.1 prikazuje fazu pripreme.

```
stage('Prepare Workspace') {
    steps {
        echo "Clearing workspace..."
        deleteDir() // Clear the workspace
    }
}
```

Listing 3.1: Faza pripreme okruženja

U fazi *Checkout*, kôd se preuzima iz *Git* repozitorijuma. Koristi se *Git plugin* da bi se preuzeli podaci iz određene grane (u ovom slučaju *main*) sa odgovarajućim podacima za prijavu. Nakon uspešnog preuzimanja koda, dobija se komit koristeći komandu *git rev-parse HEAD* i čuva se u promenljivoj *GIT_SHA*. Ova vrednost će kasnije biti korišćena za identifikaciju komita u fazi prikupljanja podataka o pokrivenosti. Listing 3.2 pokazuje fazu preuzimanja.

```
stage('Checkout') {
    steps {
        echo 'Checking out from Git...'
        // Replace with your repository URL and branch
        git branch: 'main', url: 'git@github.com:jboskovic/cppcheck-master.git
    ', credentialsId: '9a6c6d07-0b04-4278-ba08-c4659a2eb2c4'
        // Get the Git SHA of the checked-out commit
        script {
          GIT_SHA = sh(script: "git rev-parse HEAD", returnStdout: true).trim()
          echo "Checked out commit: ${GIT_SHA}"
        }
    }
}
```

Listing 3.2: Faza preuzimanja repozitorijuma

Nakon preuzimanja koda, lanac prelazi na fazu *Build* gde se izvršava prevođenje projekta. Komanda *make COVERAGE=1* se koristi za prevođenje projekta sa uključenim opcijama za pokrivenost koda. Ova komanda se izvršava u direktorijumu *cppcheck_project*. Listing 3.3 pokazuje fazu prevođenja.

```
stage('Build') {
    steps {
        // Add build commands here
        echo "Building.."
        sh "cd cppcheck_project && make COVERAGE=1"
    }
}
```

Listing 3.3: Faza prevođenja koda

Sledeća faza je *Test* gde se prvo dobijaju liste testova koristeći komandu *make testclasses COVERAGE=1*, a zatim se izvršavaju svi testovi koristeći skriptu *run_list_of_tests.sh* sa listom testova *all_tests.txt*. Oba koraka se izvršavaju u direktorijumu *cppcheck_project*. Listing 3.4 pokazuje fazu testiranja.

```
stage('Test') {
    steps {
        echo "Get list of tests..."
        sh "cd cppcheck_project && make testclasses COVERAGE=1"
        // Add test commands here
        echo "Testing.."
        sh "cd cppcheck_project && bash run_list_of_tests.sh all_tests.txt"
    }
}
```

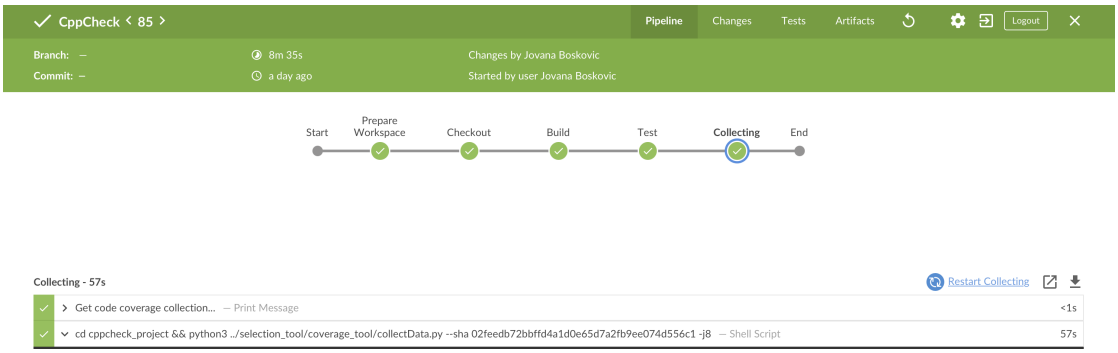
Listing 3.4: Faza testiranja koda

Na kraju, u fazi *Collecting* prikupljaju se podaci o pokrivenosti koda koristeći skriptu *collectData.py*. Skripta se pokreće sa opcijama *--sha* za identifikaciju komita i *-j8* za korišćenje paralelnih procesa tokom prikupljanja podataka. Ovaj korak osigurava da svi relevantni podaci o pokrivenosti budu prikupljeni i arhivirani za dalje korišćenje u *Pull Request* toku. Podaci se čuvaju na lokaciji koja je globalno dostupna *Jenkinsu*, u ovom slučaju */var/jenkins_home*, što je uobičajena putanja za skladištenje *Jenkins* podataka u *Docker* okruženju. Listing 3.5 pokazuje fazu prikupljanja podataka.

```
stage("Collecting") {
    steps {
        echo "Get code coverage collection..."
        sh "cd cppcheck_project && python3 ../selection_tool/coverage_tool/collectData.py --sha ${GIT_SHA} -j8"
    }
}
```

Listing 3.5: Faza prikupljanja podataka

Slika 3.1 prikazuje izgled periodičnog toka u *BlueOcean*-u, jasno ilustrujući opisane faze.



Slika 3.1: Periodični tok

Tok *Pull Request*

Tok *Pull Request* se pokreće za grane koje imaju otvoren *Pull Request*. *Pull Request* je zahtev za integraciju izmena iz jedne grane u drugu, obično iz razvojne grane u glavnu ili neku drugu ciljnu granu.

Jenkinsfile.pr definiše *CI/CD* lanac za tok *PR*, koji koristi alat *Select Relevant Tests* kako bi izabrao testove relevantne za izmene u PR-u. Tok se oslanja na podatke prikupljene tokom periodičnog toka kako bi optimizovao proces testiranja. Lanac se sastoji od nekoliko ključnih faza: pripreme radnog prostora, preuzimanja koda iz Git repozitorijuma, selekcije relevantnih testova, izgradnje projekta i izvršavanja testova.

Prve dve faze, *Prepare Workspace* i *Checkout*, implementirane su isto kao u periodičnom toku. U fazi *Prepare Workspace*, radni prostor se čisti, dok se u fazi *Checkout* kod preuzima iz Git repozitorijuma.

U fazi *Select Relevant Tests*, izvršava se *Python* skripta *selectRelevantTests.py* koja analizira izmene u kodu i selektuje relevantne testove koje treba izvršiti. Alat koristi podatke prikupljene tokom periodičnog toka kako bi identifikovao testove koji su relevantni za izmene u PR-u. Skripti se prosleđuje i *GIT_SHA* varijabla koja identifikuje trenutni komit. Kao što je opisano u periodičnom toku, podaci o pokrivenosti koda su sačuvani na globalnoj putanji */var/jenkins_home*, što omogućava *PR* toku da ima pristup tim podacima za analizu i selekciju relevantnih testova. Listing 3.6 prikazuje fazu selektovanja testova.

```
stage('Select Relevant Tests') {
    steps {
        echo "Selecting relevant tests.."
        sh "python3 selection_tool/selectRelevantTests.py --sha ${GIT_SHA}"
    }
}
```

Listing 3.6: Faza selektovanja testova

Nakon selekcije testova, lanac prelazi na fazu *Build* gde se projekat izgrađuje. Komanda *make* se koristi za izgradnju projekta u direktorijumu *cppcheck_project*.

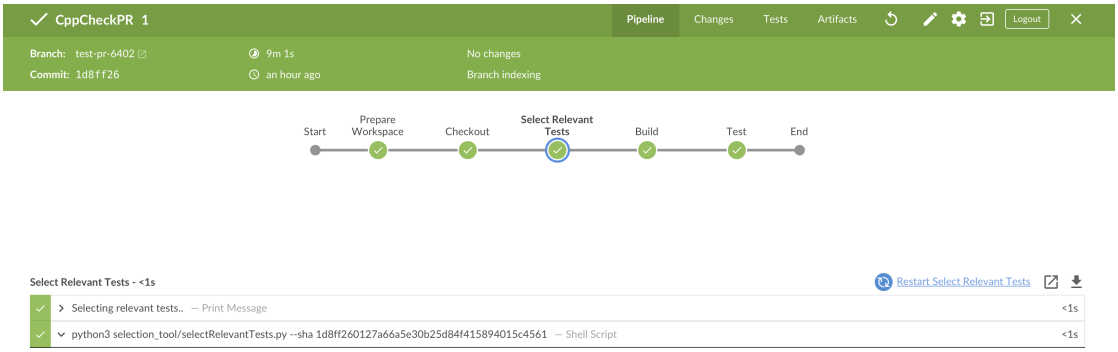
Faza *Test* uključuje izvršavanje testova. Prvo se proverava da li postoji datoteka *selected_tests.txt* u direktorijumu *cppcheck_project*. Ako datoteka postoji, to znači da je alat *Select Relevant Tests* selektovao podskup testova koji su relevantni za izmene u PR-u i ova datoteka se koristi za izvršavanje testova. U suprotnom, generiše se lista svih testova pomoću komande *make testclasses*. Nakon toga, testovi se izvršavaju pomoću skripte *run_list_of_tests.sh* sa odgovarajućom listom testova. Listing 3.7 prikazuje fazu testiranja selektovanih testova.

```
stage('Test') {
    steps {
        script {
            def testListFile = 'all_tests.txt'
            if (fileExists('cppcheck_project/selected_tests.txt')) {
                testListFile = 'selected_tests.txt'
            } else {
```

```
8                 echo "Get list of tests..."
9                 sh "cd cppcheck_project && make testclasses"
10            }
11            echo "Using test list: ${testListFile}"
12            // Run the tests
13            echo "Testing.."
14            sh "cd cppcheck_project && bash run_list_of_tests.sh ${testListFile
    }"
15        }
16    }
17 }
```

Listing 3.7: Faza testiranja

Sledeća slika 3.2 prikazuje izgled periodičnog toka u *BlueOcean*-u, jasno ilustrujući sve opisane faze.



Slika 3.2: *Pull Request* tok

Implementacija alata *Select Relevant Tests*

Alat *Select Relevant Tests* implementiran je kao Python [23] skripta koja se sastoji od dva glavna dela *CollectData* i *SelectTests*. Prvi deo služi za prikupljanje podataka o pokrivenosti koda i koristi se u periodičnom toku, dok drugi deo koristi prikupljene podatke za selektovanje relevantnih testova i koristi se u *Pull Request* toku.

CollectData

CollectData je skripta namenjena prikupljanju detaljnih podataka o pokrivenosti koda za svaki pojedinačni test. Korisnik može da definiše parametre kao što su:

- **--sha**: Identifikator komita za koji se prikupljaju podaci.
- **-j**: Broj paralelnih procesa za izvršavanje zadataka.
- **--dont-delete-gcda-files**: Opcija za zadržavanje *gcda* datoteka nakon završetka prikupljanja podataka.

Skripta *CollectData* automatski prepoznaje koji su testovi izvršeni tako što prolazi kroz direktorijum koji sadrži podatke o pokrivenosti koda. Unutar tog direktorijuma, skripta identifikuje poddirektorijume koji odgovaraju pojedinačnim testovima. Svaki poddirektorijum sadrži *gcda* datoteke koje predstavljaju podatke o pokrivenosti za određeni test. Na osnovu tih datoteka, alat prikuplja detaljne informacije o pokrivenosti koda, omogućavajući granularno praćenje pokrivenosti za svaki test. Skripta *CollectData.py* je prikazana u listingu 3.8.

Inicijalizuju se objekati *Storage* i *Collector*. *Collector* kreira više paralelnih procesa koji preuzimaju testove iz reda i prikupljaju podatke o pokrivenosti za svaki test. Svaki proces preuzima listu *gcda* datoteka iz odgovarajućeg direktorijuma i pokreće alat *gcov* kako bi dobio podatke o pokrivenosti. Izlaz *gcov*-a se parsira kako bi se identifikovale funkcije koje su pokrivene, a zatim se čuvaju u objektu *Storage*.

Objekat *Storage* definiše način na koji se podaci čuvaju. Podaci se indeksiraju i mapiraju kako bi se efikasno organizovali i pretraživali. Konkretno, za svaku datoteku se čuvaju mapiranja koja prikazuju koje funkcije su pokrivene u toj datoteci i koji testovi pokrivaju te funkcije. Da bi se uštedela memorija, mapiranja se čuvaju u obliku indeksa umesto punih imena funkcija i testova. Podaci se čuvaju u nekoliko *JSON* datoteka:

- *tests_indexed.json*: Sadrži informacije o indeksima testova.
- *files_indexed.json*: Sadrži informacije o indeksima datoteka.

- *functions_indexed.json*: Sadrži informacije o indeksima funkcija.
- *functions_to_tests.json*: Sadrži mapiranja koja povezuju funkcije sa testovima koji ih pokrivaju.

Sve *JSON* datoteke se arhiviraju pod jedinstvenim identifikatorom komita. Čuvaju se u direktorijumu sa nazivom koji se sastoji od datuma i identifikator komita u formatu „Y-m-d-H:M:S_sha". U ovom formatu, Y predstavlja godinu, m mesec, d dan, H sat, M minut, a S sekund kada je komit napravljen. Ovaj jedinstveni naziv omogućava istorijski pregled pokrivenosti i brzo izvlačenje relevantnih informacija.

```
if __name__ == '__main__':
    time_for_collection_started = time.time()

    args = parse_args(sys.argv[1:])

    # get all executed test files
    tests = from_gcda_dir_path_collect_test_names()
    # object Storage is multiprocess safe and is used for storing collected
    data
    storage = Storage(tests, args.sha)

    # object Collector is used for collecting the data in parallel
    collector = Collector(storage, args.dont_delete_gcda, coverage_dir)
    collector.run(args.j)

    # save collected data to json files in a directory that's named using sha
    and date of sha
    storage.save_data()

    time_for_collection_ended = time.time()
    end_time_for_collection = time_for_collection_ended -
    time_for_collection_started
    print("Collection completed. Took time {}".format(end_time_for_collection))
```

Listing 3.8: Skripta *CollectData.py*

SelectTests

SelectTests je skripta koja koristi prikupljene podatke za selektivno testiranje tokom *Pull Request* toka. Skripta se sastoji iz dva komponente: *ParsePR* i *CoverageData*. Prva komponenta, *ParsePR*, započinje proces analizom izmena napravljenih u *PR-u* kako bi identifikovala promenenjene datoteke i funkcije. Druga komponenta, *CoverageData*, na osnovu tih promena i prikupljenih podataka selektuje testove koje je potrebno pokrenuti. Skripta *SelectTests.py* je prikazana u listingu 3.9.

Komponenta *ParsePR* koristi *Git* komande kako bi dobila osnovnu verziju PR-a (eng. *baseline*) i identifikovala sve promenjene datoteke i funkcije.

Osnovna verzija u kontekstu *Pull Request*-a predstavlja referentnu tačku ili osnovnu verziju koda s kojom se upoređuju izmene unete u *Pull Request* [9]. Da bi se izračunala osnovna verzija, koristi se komanda *git merge-base*, koja vraća *hash* zajedničkog komita.

Za identifikaciju promena koristi se komanda *git diff*. Ova komanda prikazuje razlike između različitih verzija datoteka u *Git* repozitorijumu, pružajući informacije kao što su imena promenjenih datoteka, broj izmenjenih linija, opseg izmena, a često i ime bloka ili funkcije u kojoj su napravljene izmene. Imena promenjenih datoteka pojavljuju se sa prefiksima a/ i b/, označavajući originalnu i izmenjenu verziju datoteke. Opseg izmena je označen linijom koja počinje sa @@, a često uključuje i naziv funkcije ili bloka koda koji je izmenjen. Da bi se fokusirali na relevantne datoteke koja sadrže podatke o pokrivenosti, datoteke se filtriraju prema ekstenzijama .c, .cpp, .h, i .hpp, uklanjajući one koje nisu relevantne.

Mana korišćenja *git diff* je što ne može uvek tačno da identifikuje ime promenjene funkcije u *C++* datotekama. Razlog je u složenosti i fleksibilnosti *C++* sintakse, koja može uključivati preopterećenje funkcija, šablone i anonimne funkcije. Ovi elementi mogu otežati precizno određivanje mesta gde funkcija počinje i završava, što može rezultirati nepravilnim identifikovanjem izmenjenih funkcija. Takođe, u situacijama kada su funkcije dodate ili izbrisane, *git diff* pogrešno identifikuje koja funkcija je promenjena, hvatajući prvu funkciju iznad kao izmenjenu. U ovakvim slučajevima, alat identifikuje pogrešnu funkciju i na osnovu nje selektuje testove. Pretpostavka je da, ako se funkcija izbriše ili doda, neophodno je izmeniti

i funkcije koje koriste tu dodatu ili izbrisanu funkciju. Ovo rezultira time da alat selektuje više testova nego što je potrebno, ali ne manje.

Nakon prikupljanja izmena, *ParsePR* generiše mapu promenjenih datoteka i funkcija u njima, omogućavajući preciznu identifikaciju delova koda koji su izmenjeni.

Komponenta *CoverageData* koristi podatke prikupljene komponentom *ParsePR* kako bi selektovala relevantne testove za izvršavanje tokom *Pull Request* toka. Prvo, komponenta uzima vreme kreiranja *baseline* komita i koristi ga za pretraživanje svih postojećih kolekcija podataka o pokrivenosti. Cilj je pronaći kolekciju koja je najskorije napravljena u odnosu na vreme kreiranja *PR-a*. Ova kolekcija će najpreciznije opisivati stanje koda u trenutku kada je komit napravljen, omogućavajući preciznu identifikaciju relevantnih testova. Ukoliko je pronađena kolekcija stara ili ne postoji, izvršavaju se svi testovi.

Na osnovu mape izmena generisane od strane komponente *ParsePR*, *CoverageData* prolazi kroz sve izmenjene datoteke i funkcije. Kolekcija podataka o pokrivenosti koja je pronađena sadrži informacije o pokrivenosti za svaku datoteku i funkciju. *CoverageData* koristi ove informacije da identifikuje testove koji su relevantni za izmenjene funkcije. Za svaku izmenjenu funkciju, komponenta koristi mapu iz kolekcije koja povezuje funkcije sa testovima kako bi identifikovala testove koji pokrivaju te funkcije. Testovi koji pokrivaju izmenjene funkcije se selektuju i dodaju u skup testova koji treba da se izvrše. Na kraju, skup selektovanih testova se čuva u jednoj datoteci koju CI/CD sistem može da pročita i koristi tokom procesa testiranja.

```
1  class SelectRelevantTests:
2      def __init__(self):
3          print('Starting PR Parsing...')
4          self.parser = ParsePR()
5          print('PR Parsing finished successfully!')
6          self.call_coverage_decision_tool()
7
8      def call_coverage_decision_tool(self):
9          input_for_coverage_tool = self.parser.collected_changes
10         print('\n###################################')
11         print('Coverage decision tool run start...')
12         print('###################################\n')
```

```
13          self.coverage_tool = CoverageData(input_for_coverage_tool, self.parser.
    baseline)
14          print("Coverage decision tool run finished successfully!\n")
15          print('###################################')
16          print("Selecteed tests")
17          print(self.coverage_tool.output)
18          print('###################################')
19          self.write_selected_test_to_file(self.coverage_tool.output)
20
21      def write_selected_test_to_file(self, list_of_tests):
22          if list_of_tests != ["all"]:
23              file_name = project_name + "/selected_tests.txt"
24              print("Write selected tests to a file {}".format(file_name))
25              try:
26                  with open(file_name, 'w') as file:
27                      for test in list_of_tests:
28                          file.write(f"{test}\n")
29                      print("Successfully wrote {} tests to {}".format(len(
    list_of_tests), file_name))
30              except Exception as e:
31                  print(f"An error occurred: {e}")
32          else:
33              print("Tool for Selecting Relevant Tests selected all tests")
34          print('###################################')
```

Listing 3.9: Skripta *SelectTests*

3.3 Prikaz integracije alata u projekat otvorenog koda

Da bi se precizno prikupljali podaci o pokrivenosti koda po testu, potrebno je integrisati odgovarajući kod u deo koji se poziva pre svakog testa. Ovaj kod mora da kreira direktorijum za svaki test, gde će se čuvati podaci o pokrivenosti.

Kôd 3.10 demonstrira kako se pomoću *C++* funkcije može pripremiti test okruženje za prikupljanje ovih podataka. Funkcija postavlja potrebne promenljive okruženja i kreira direktorijum za svaki test, što osigurava da se podaci o pokrivenosti

prikupljaju i čuvaju odvojeno.

```
bool TestFixture::prepareTestForCoverage(std::string testname){
    std::string testDir = "./coverage_per_test/" + std::string(testname); //
    Adjust the path as needed
    if (mkdir(testDir.c_str(), 0777) != 0) {
        // If directory creation fails and it's not because the directory
    exists
        if (errno != EEXIST) {
            std::cerr << "Failed to create directory for test: " << testDir
                    << ", Error: " << strerror(errno) << std::endl;
            return false;
        }
    }

    // Set the environment variables to point to the new directory
    if (setenv("GCOV_PREFIX", testDir.c_str(), 1) != 0) {
        std::cerr << "Failed to set GCOV_PREFIX environment variable." << std::
    endl;
        return false;
    }

    if (setenv("GCOV_PREFIX_STRIP", "0", 1) != 0) {
        std::cerr << "Failed to set GCOV_PREFIX_STRIP environment variable." <<
     std::endl;
        return false;
    }
}
```

Listing 3.10: Funkcija za pripremu test okruženja

Da bi *gcov* mogao da sačuva podatke o pokrivenosti u kreirani direktorijum, potrebno je postaviti odgovarajuće promenljive okruženja. Promenljiva *GCOV_PREFIX* se postavlja na putanju kreiranog direktorijuma, dok se *GCOV_PREFIX_STRIP* postavlja na vrednost „0” da bi se osigurala pravilna struktura putanja. Ako postavljanje bilo koje od ovih promenljivih okruženja nije uspešno, funkcija prijavljuje grešku i vraća *false*. Ako su svi koraci uspešno izvršeni, funkcija vraća *true*, signalizirajući da je okruženje za prikupljanje podataka o pokrivenosti uspešno pripremljeno.

Integracija ove funkcionalnosti u razvojni tok podrazumeva da se pre izvršavanja svakog testa poziva funkcija *prepareTestForCoverage* sa odgovarajućim imenom testa. Nakon pripreme okruženja, test se izvršava kao i obično, dok *gcov* automatski prikuplja podatke o pokrivenosti i smešta ih u kreirani direktorijum. Nakon izvršavanja testova, prikupljeni podaci o pokrivenosti se mogu analizirati i transformisati u odgovarajući format (npr. JSON) za dalju upotrebu.

Da bi ovaj proces bio uspešan, važno je omogućiti pokretanje pojedinačnih testova. Ovo znači da svaki test mora biti izolovano izvršen kako bi se njegovi rezultati mogli detaljno analizirati. Pojedinačno pokretanje testova omogućava precizno prikupljanje podataka o pokrivenosti za svaki test.

Glava 4

Evaluacija i testiranje

U ovom poglavlju ćemo se fokusirati na evaluaciju i testiranje alata *Select Relevant Tests*. Alat je primenjen na projektu *cppcheck*, demonstrirajući njegovu upotrebu na konkretnom primeru. *Cppcheck* [11] je statički analizator koda za *C/C++* projekte koji detektuje nedefinisano ponašanje i rizične kodne konstrukcije, sa ciljem da minimizira broj lažno pozitivnih rezultata. Dizajniran je da može analizirati C/C++ kod čak i kada koristi nestandardnu sintaksu, što je često prisutno u projektima za sisteme sa ugrađenim računarom (eng. *embedded systems*).

Evaluacija alata sprovodi se kroz niz eksperimenata koji uključuju pokretanje alata na različitim PR-ovima i upoređivanje testova koje alat selektuje sa testovima koji zaista pokrivaju izmenjene delove koda. Cilj je da se proceni koliko je alat precizan i efikasan u selekciji relevantnih testova.

4.1 Metodologija i proces evaluacije alata

Da bismo procenili efikasnost alata za selektivno testiranje zasnovano na analizi pokrivenosti koda, koristimo empirijski pristup koji uključuje analizu pokrivenosti linija koda kako bismo identifikovali stvarno pogođene testove izmenama. Zatim upoređujemo te testove sa testovima koje alat selektuje. Ova analiza omogućava da procenimo koliko dobro alat prepoznaje relevantne testove i merimo uštedu u broju izvršenih testova. Proces se sprovodi u nekoliko ključnih koraka:

1. **Pokretanja na PR-u**

   Selektovanje testova: Alat se pokreće za svaki *Pull Request* i identifikuje relevantne testove na osnovu izmenjenih funkcija u kodu. Selektovani testovi se beleže u posebnu datoteku.

   Identifikacija izmenjenih linija: Alat koristi opciju *git diff* za identifikaciju izmenjenih linija koda u PR-u. Izmenjene linije se organizuju u mapu gde su ključevi imena datoteka, a vrednosti liste izmenjenih linija unutar tih datoteka. Ova mapa se čuva za dalju analizu. Kôd koji identifikuje promenjene linije prikazan je u Listing 4.1.

```
1    def get_changed_lines_from_PR(self):
2        try:
3            output = subprocess_call(
4                'git --no-pager diff --ignore-space-change --ignore-blank-
    lines --unified=0 `git merge-base origin/main HEAD` -- \'*.cpp\' \'*.h
    \' \'*.c\''
5            )
6        except Exception as e:
7            exit_with_message(f"Getting changed files from PR failed {e}")
8
9        # Regex patterns to match file headers and hunks
10       file_header_pattern = re.compile(r"^diff --git a/(.*) b/(.*)")
11       hunk_header_pattern = re.compile(r"^@@ -\d+(?:,\d+)? \+(\d+)(?:,(\
    d+))? @@")
12
13       current_file = None
14       changed_lines = {}
15
16       # Parse the git diff output
17       for line in output.stdout.splitlines():
18           file_match = file_header_pattern.match(line)
19           if file_match:
20               current_file = file_match.group(2)
21               current_file = current_file.split("cppcheck_project/")[1]
22               continue
23
24           hunk_match = hunk_header_pattern.match(line)
```

```
25              if hunk_match and current_file:
26                  start_line = int(hunk_match.group(1))
27                  num_lines = int(hunk_match.group(2) or 1)
28                  current_line = start_line
29                  for i in range(num_lines):
30                      # Skip deleted lines (lines starting with '-')
31                      if line.startswith('-'):
32                          continue
33                      # Only add lines that start with '+' (added lines)
34                      if line.startswith('+') or not line.startswith('-'):
35                          if current_file not in changed_lines:
36                              changed_lines[current_file] = []
37                          changed_lines[current_file].append(current_line)
38                      current_line += 1
39
40          print("Changed lines per file ", changed_lines)
41          return changed_lines
```

Listing 4.1: Identifikacija izmenjenih linija

2. **Pokretanje periodičnog toka**

   Prevođenje i pokrivenost: Nad istim PR-om se pokreće periodični tok koji prevodi PR sa uključenom opcijom za pokrivenost koda. Ovo omogućava prikupljanje detaljnih podataka o tome koje linije koda su pokrivene kojim testovima.

   Izvršavanje testova: Svi testovi se pokreću, i za svaki test se prikupljaju podaci o linijama koda koje taj test pokriva. Kôd koji prikuplja pokrivene linije po testu prikazan je u Listing 4.1.

   Kreiranje datoteke *lines_to_tests.json*: Na osnovu prikupljenih podataka kreira se datoteka *lines_to_tests.json*, koja sadrži mapu gde se datoteke mapiraju na linije koda, a svaka linija koda na listu testova koji je pokrivaju. Ova datoteka pruža tačan uvid u pokrivenost koda na nivou linija.

3. **Upoređivanje rezultata**

Selektovanje testova na osnovu izmenjenih linija: Koristeći datoteku *lines_to_tests.json* i spisak izmenjenih datoteka i linija iz PR-a, selektuju se testovi koji pokrivaju te izmenjene linije.

Validacija selekcije: Proces validacije uključuje upoređivanje testova selektovanih od strane alata sa testovima identifikovanim na osnovu izmenjenih linija koda. Upoređivanjem ovih skupova testova, procenjuje se tačnost selekcije. Ova analiza omogućava identifikaciju potencijalnih nedostataka ili prekomernih selekcija, čime se ocenjuje efikasnost alata u identifikaciji relevantnih testova.

```
# parse the json object from the gcov output for one test and return a json of executed lines and functions per file
def parse_full_json_object(self, json_object):
    new_json_object_lines = {}
    new_json_object_functions = {}

    executed_lines, functions = [], []
    json_object = json.loads(json_object)
    for file_object in json_object['files']:
        file = file_object['file']
        file_index = str(self._storage.insert_file_indexed(file))
        executed_lines = []
        functions = []

        for line_info in file_object['lines']:
            if line_info['count'] > 0:
                executed_lines.append(line_info['line_number'])

        for function_info in file_object['functions']:
            if function_info['execution_count'] > 0:
                function_name = function_info['demangled_name']
                function_index = self._storage.insert_function_indexed(function_name)
                functions.append(function_index)

        if executed_lines != []:
            # Ensure the file_index exists in both dictionaries
            if file_index not in new_json_object_lines:
```

```
27                  new_json_object_lines[file_index] = []
28              if file_index not in new_json_object_functions:
29                  new_json_object_functions[file_index] = []
30
31              # Merge executed lines and functions without duplicates
32              all_lines = list(set(new_json_object_lines[file_index] +
    executed_lines))
33              new_json_object_lines[file_index] = all_lines
34              all_functions = list(set(new_json_object_functions[file_index]
    + functions))
35              new_json_object_functions[file_index] = all_functions
36
37          return new_json_object_functions, new_json_object_lines
```

Listing 4.2: Prikupljanje pokrivenih linija

4.2 Analiza rezultata i efikasnosti alata

Analiziramo rezultate i efikasnost alata *Select Relevant Tests* koristeći stvarne *Pull Request*-ove sa projekta *cppcheck*. Za evaluaciju alata korišćeni su sledeći *Pull Request*-ovi:

PR 6388 [12] — U ovom PR-u su izvršene manje izmene u funkciji *void setParentExprId* iz fajla *cppcheck_project/lib/symboldatabase.cpp*, i dodata nova funkcija *void exprid11* u fajlu *cppcheck_project/test/testvarid.cpp*.

PR 6392 [13] — U ovom PR-u su izvršene manje izmene u funkciji *void Tokenizer::setVarIdPass2()* iz fajla *cppcheck_project/lib/tokenize.cpp*, i dodata nova funkcija *void varid71()* u fajlu *cppcheck_project/test/testvarid.cpp*

PR 6393 [14] — U ovom PR-u su izvršene promene i optimizacije u funkcijama za proveru reda evaluacije u fajlu *cppcheck_project/lib/checkother.cpp*, uključujući dodavanje i brisanje funkcija i promene potpisa funkcija. Takođe su izvršene izmene u fajlovima *cppcheck_project/lib/checkother.h* i *cppcheck_project/test/testother.cpp*.

PR 6401 [15] — U ovom PR-u su izvršene manje izmene u funkcijama *void CheckCondition::multiCondition2()* i *void oppositeInnerCondition()* iz fajla *cppcheck_project/lib/checkcondition.cpp*, kao i izmene u testovima u fajlu *cppcheck_project/test/testcondition.cpp*.

PR 6402 [16] — U ovom PR-u su dodate nove provere i optimizacije u funkcijama *void CheckClass::initializerListOrder()* i *initializerListArgument* iz fajla *cppcheck_project/lib/checkclass.cpp*, kao i izmene u testovima u fajlu *cppcheck_project/test/testclass.cpp*.

PR 6403 [17] — U ovom PR-u su izvršene izmene u funkciji *const Token* Tokenizer::isFunctionHead* iz fajla *cppcheck_project/lib/tokenize.cpp* i u funkciji *void varid70* iz fajla *cppcheck_project/test/testvarid.cpp*.

PR 6409 [18] — U ovom PR-u su dodate nove provere i optimizacije u funkcijama *static std::vectorValueFlow::Value getInitListSize* i *void outOfBounds()* iz fajla *cppcheck_project/lib/valueflow.cpp*, kao i izmene u testovima u fajlu *cppcheck_project/test/teststl.cpp*.

PR 6411 [19] — U ovom PR-u su izvršene manje promene u funkciji *void CmdLineParser::printHelp()* iz fajla *cppcheck_project/cli/cmdlineparser.cpp*.

PR 6412 [20] — U ovom PR-u su izvršene promene u funkcijama za analizu toka vrednosti u fajlu *cppcheck_project/lib/valueflow.cpp*, uključujući optimizacije i proširenje funkcionalnosti u funkcijama *Action analyze(const Token* tok, Direction d)*, *static void valueFlowForwardAssign*, *static bool isVariableInit* i *void valueFlowAfterAssign()*. Takođe su izvršene izmene u testovima u fajlu *cppcheck_project/test/testvalueflow.cpp*.

Analiza se zasniva na podacima prikupljenim iz ovih PR-ova, što omogućava realističnu evaluaciju alata u kontekstu stvarnih scenarija razvoja softvera.

Projekat *cppcheck* sadrži ukupno 76 testova, a dizajniran je tako da se uvek svi testovi izvršavaju prilikom svake promene u kodu. Koristeći alat *Select Relevant Tests*, selektujemo samo one testove koji su relevantni na osnovu izmenjenih funkcija u kodu. Na taj način možemo izračunati kolika je ušteda ostvarena selektovanjem samo relevantnih testova u poređenju sa izvršavanjem svih 76 testova.

Za svaki PR, alat identifikuje i beleži broj relevantnih testova, a uštedu izračunavamo kao odnos broja selektovanih testova prema ukupnom broju testova. Na primer, ako alat selektuje 10 testova od ukupno 76, to znači da je izvršeno samo 13.16% svih testova, što rezultira uštedom od 86.84% u broju izvršenih testova.

Pored analize uštede u broju izvršenih testova, takođe procenjujemo broj lažno pozitivnih i lažno negativnih rezultata. Lažno pozitivni rezultati su testovi koje je alat selektovao, ali koji ne pokrivaju izmenjene linije koda. Lažno negativni rezultati su testovi koje alat nije selektovao, a koji pokrivaju izmenjene linije koda. Ova analiza pomaže u proceni tačnosti alata i identifikaciji potencijalnih nedostataka.

Kombinovanjem ovih metrika (ušteda u broju izvršenih testova, broj lažno pozitivnih i broj lažno negativnih rezultata) možemo dobiti sveobuhvatan uvid u efikasnost i pouzdanost alata *Select Relevant Tests*. Ovakav pristup omogućava identifikaciju područja za poboljšanje i optimizaciju procesa testiranja u razvoju softvera.

Rezultati su prikazani u tabeli 4.1.

Tabela 4.1: Rezultati evaluacije alata: broj selektovanih testova, broj relevantnih testova, broj lažno pozitivnih i lažno negativnih testova, kao i procenat uštede u broju izvršenih testova

PR	**Selekt. testovi**	**Relevantni testovi**	**Lažno poz.**	**Lažno neg.**	**Ušteda (%)**
6388	0	10	0	10	100%
6392	55	35	21	0	27.63%
6393	11	9	2	0	85.53%
6401	10	1	9	0	86.84%
6402	8	2	6	0	89.47%
6403	57	57	0	0	26.32%
6409	7	8	1	2	90.79%
6411	1	1	0	0	98.68%
6412	17	17	0	0	77.63%

Projekat sadrži ukupno 76 testova

Na osnovu prikupljenih podataka možemo zaključiti sledeće:

- Prosečna ušteda u broju izvršenih testova pri korišćenju alata *Select Relevant Tests* iznosi približno 75.87%. Ovo znači da se u proseku broj izvršenih testova smanjuje za 75.87%, što značajno optimizuje proces testiranja.
- Minimalna ušteda u broju izvršenih testova iznosi 26.32%, što je najmanja zabeležena. Za ovaj PR je zabeleženo da je alat selektovao sve testove koji treba da se izvrše. Ukupno je selektovano 57 testova, a svi su potvrđeni kao relevantni za izmenjene linije koda. Ova ušteda je stoga razumljiva i opravdana, jer su svi relevantni testovi bili selektovani i izvršeni kako bi se osigurala ispravnost koda.
- Maksimalna ušteda u broju izvršenih testova iznosi 100%, što je najveća zabeležena ušteda među analiziranim PR-ovima. Međutim, za ovaj primer je zabeleženo lažno negativno, što znači da alat nije uspeo da selektuje relevantne testove koji pokrivaju izmenjene linije koda. Zbog toga, ovaj rezultat nije pouzdan i može se odbaciti jer ne odražava stvarnu efikasnost alata.

Prosečna ušteda broja pokretanja testova pri korišćenju alata *Select Relevant Tests* iznosi približno 72.86% kada ne uzimamo u obzir PR sa lažno negativnim rezultatom. Ovo znači da se značajno smanjuje broj izvršenih testova potrebnih za proveru izmenjenog koda.

Lažno pozitivni rezultati mogu se pojaviti iz nekoliko razloga:

Pokrivenost funkcija. Alat koristi pokrivenost funkcija umesto pokrivenosti pojedinačnih linija koda. To znači da ako je bilo koji deo funkcije izmenjen, svi testovi koji pokrivaju tu funkciju će biti selektovani, čak i ako konkretne linije koda koje su izvršene tokom testiranja nisu promenjene. Zbog toga se selektuje više testova nego što je potrebno.

Dodavanje ili brisanje funkcija. Parser PR-a može pogrešno identifikovati izmenjenu funkciju usled dodavanja ili brisanja funkcija u kodu. U takvim

slučajevima, alat može prijaviti da je promenjena prva funkcija iznad dodate ili obrisane funkcije, što rezultira selekcijom testova koji se odnose na tu funkciju.

Nepouzdane informacije u kolekciji. Ponekad PR može uhvatiti kolekciju koja sadrži nepouzdane ili zastarele informacije. To se može desiti ako je korišćenje funkcije izbrisano u nekom prethodnom PR-u koji nije obuhvaćen trenutnom kolekcijom.

Lažno negativni rezultati mogu se pojaviti iz nekoliko razloga:

Greške u parsiranju izmenjenih funkcija. Parser PR-a može pogrešno isparsirati izmenjene funkcije zbog složenosti C++ sintakse. Ovo se dešava kada alat ne prepozna tačno ime promenjene funkcije, što može dovesti do toga da alat selektuje pogrešne testove koji ne pokrivaju stvarno izmenjene linije koda. Na primer, ako se izmeni funkcija koja ima kompleksnu sintaksu, parser može propustiti da tačno identifikuje tu funkciju, što znači da testovi povezani sa izmenama u toj funkciji neće biti selektovani.

Zastarela kolekcija. Lažno negativni rezultati se mogu pojaviti ako PR koristi zastarelu kolekciju testova. Ovo se dešava kada funkcija izmenjena u trenutnom PR-u nije prepoznata kao relevantna jer su izmene ili korišćenje te funkcije dodate u nekom prethodnom PR-u koji nije obuhvaćen trenutnom kolekcijom.

Izmene van funkcija. Lažno negativni rezultati mogu se pojaviti i kada se izmena odnosi na promenu podataka u klasi, a ne unutar same funkcije. Ovo uključuje promene vrednosti varijabli, tipa promenljivih ili strukture podataka. U takvim slučajevima, alat neće prepoznati sve relevantne testove jer se fokusira na izmene unutar funkcija.

Glava 5

Zaključak

U ovom master radu istraženi su načini unapređenja procesa kontinuirane integracije i isporuke (CI/CD) kroz selektivno testiranje zasnovano na analizi pokrivenosti koda. Razvijen je alat pod nazivom *Select Relevant Tests*, koji automatski identifikuje izmenjene delove koda i selektuje samo one testove koji su relevantni za te izmene, koristeći alat za merenje pokrivenosti koda *gcov*.

Kroz evaluaciju alata na stvarnim *Pull Request*-ovima sa projekta *cppcheck*, pokazano je da ovaj pristup može značajno smanjiti broj izvršenih testova, omogućavajući brže integracije i isporuke. Analiza podataka je pokazala da prosečna ušteda u broju izvršenih testova pri korišćenju alata *Select Relevant Tests* za sve analizirane PR-ove iznosi približno 75.87%, što znači da se u proseku broj izvršenih testova smanjuje za ovaj procenat, čime se značajno optimizuje proces testiranja. Kada se isključi PR sa lažno negativnim rezultatom (PR 6388), prosečna ušteda u broju izvršenih testova iznosi približno 72.86%, što takođe predstavlja značajnu optimizaciju.

Ovi rezultati pokazuju da alat *Select Relevant Tests* može značajno optimizovati proces testiranja, smanjujući ukupan broj izvršenih testova i bolje iskorišćavajući resurse korišćene za testiranje. Međutim, identifikovani su i neki nedostaci, poput lažno pozitivnih i lažno negativnih rezultata, koji su uglavnom uzrokovani složenostima u parsiranju koda i zastarelim kolekcijama testova.

Dalji rad na unapređenju alata mogao bi se fokusirati na smanjenje broja lažno pozitivnih i lažno negativnih rezultata kroz poboljšane algoritme za analizu koda

i ažuriranje kolekcija testova. Takođe, implementacija dodatnih metričkih alata i integracija sa drugim alatima za merenje pokrivenosti koda mogla bi doprineti još većoj tačnosti i efikasnosti alata.

Bibliografija

[1] Inc. Airbnb. Airbnb. on-line at: https://www.airbnb.com.

[2] Paul Ammann and Jeff Offutt. Introduction to Software Testing. *Cambridge University Press, Cam- bridge, UK*, pages 15–17, 2016.

[3] The Kubernetes Authors. Kubernetes. on-line at: https://kubernetes.io.

[4] Bitrise. Bitrise. on-line at: https://bitrise.io.

[5] Jovana Boskovic. Select Relevant Tests. on-line at: https://github.com/jboskovic/cppcheck-master.

[6] Browserstack. Code Coverage Tools. on-line at: https://www.browserstack.com/guide/code-coverage-tools.

[7] Bugfinder. Jenkins Started Guide Ebook. on-line at: https://bugfender.com/wp-content/themes/bugfender-wordpress-theme/assets/docs/Jenkins-Starter-Guide-Ebook.pdf.

[8] Google Cloud. GCP. on-line at: https://cloud.google.com.

[9] The Git Development Community. Git. on-line at: https://git-scm.com.

[10] Target Corporation. Target. on-line at: https://www.target.com.

[11] cppcheck. Cppcheck. on-line at: https://github.com/danmar/cppcheck.

[12] Cppcheck. PR 6388, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/983881b7ac6b0b0f6e5efe397d60b6b2d14d524f.

[13] Cppcheck. PR 6392, 2024. on-line at: ttps://github.com/jboskovic/cppcheck-master/commit/5aeda583b25d263e15031b378180d751ec8df568.

[14] Cppcheck. PR 6393, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/pull/13/files.

[15] Cppcheck. PR 6401, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/1a0d9cfc8dbfacb8f1f072a6ad1d52e522f21560.

[16] Cppcheck. PR 6402, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/1d8ff260127a66a5e30b25d84f415894015c4561.

[17] Cppcheck. PR 6403, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/c0c5b6f976baf861333ea52fce3151fce4e96eea.

[18] Cppcheck. PR 6409, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/11ea2f25526a5bad2be0aa498b6e0a25d982dfd7.

[19] Cppcheck. PR 6411, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/commit/a15d60e0979fe01e7e2100e3e52ed92ac5b27393.

[20] Cppcheck. PR 6412, 2024. on-line at: https://github.com/jboskovic/cppcheck-master/pull/12/files.

[21] Inc. Docker. Docker. on-line at: https://www.docker.com.

[22] Free Software Foundation. GNU gcc, 2013. on-line at: http://gcc.gnu.org/.

[23] Python Software Foundation. Python. on-line at: https://www.python.org.

[24] GitHub. Spinnaker. on-line at: https://github.com/spinnaker/spinnaker.

[25] GOCD. GoCD. on-line at: https://www.gocd.org/index.html.

[26] Zhenmai Hu. A Software Package for Generating Code Coverage Reports With Gcov. *B.A.Sc., Changsha University of Science and Technolog*, pages 10–11, 2013.

[27] Adobe Inc. Adobe. on-line at: https://www.adobe.com.

[28] Test Insitute. Automated Software Testing. on-line at: https://www.test-institute.org/Automated_Software_Testing.php.

[29] Jasmina Osivčić. FazeTestiranja. on-line at: https://iu-travnik.com/wp-content/uploads/2020/01/Jasmina-OsivÄĐiÄĞ-âĂŞ-ZAVRÅăNI-RAD-METODE-I-METODOLOGIJE-U-TESTIRANJU-SOFTVERA pdf.

[30] Testwell Oy. Testwell. on-line at: https://www.testwell.fi.

[31] Sten Pittet. An introduction to code coverage, 2021. on-line at: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[32] GNU Project. Gcov. on-line at: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[33] GNU Project. Gprof. on-line at: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[34] The Debian Project. Debian. on-line at: https://www.debian.org/releases/bookworm/.

[35] RedHat. Automation testing types. on-line at: https://katalon.com/resources-center/blog/automation-testing-types.

[36] RedHat. What is CI/CD. on-line at: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[37] Amazon Web Services. AWS. on-line at: https://aws.amazon.com/about-aws/?nc2=h_header.

[38] Qameta Software. Allure. on-line at: https://qameta.io.

Biografija autora

Jovana Bošković rođena je 1. jula 1997. godine, u Kruševcu. Osnovne studije završila je na Matematičkom fakultetu Univerziteta u Beogradu, smer Informatika 2020. godine sa prosečnom ocenom 9,18 gde je stekla zvanje diplomiranog informatičara. Nakon toga, nastavila je svoje obrazovanje upisavši master studije na istom fakultetu. Od 2021. godine zaposlena je u kompaniji *Cisco*.