SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Joshua Lee Fletcher, Noa Midžić

RAZVOJ ALATA PODRŽANOG UMJETNOM INTELIGENCIJOM ZA ANALIZU KVALITETE OPISA VERZIJA PROGRAMSKOG KÔDA KAO POMOĆ U UČENJU I POUČAVANJU SLOŽENOG KONCEPTA VERZIONIRANJA PRI RAZVOJU PROGRAMSKIH PROIZVODA

Varaždin, 2025.

Ovaj rad izrađen je na Sveučilištu u Zagrebu, Fakultetu organizacije i informatike u Varaždinu u Laboratoriju za primijenjeno softversko inženjerstvo pod vodstvom **izv. prof. dr. sc. Zlatka Stapića** i predan je za natječaj za dodjelu Rektorove nagrade u akademskoj godini **2024./2025.**

Kratice korištene u radu

LLM	veliki jezični model (eng. Large Language Model)
DSRM	metodologija znanosti o oblikovanju (eng. Design Science Research Methodology)
SLR	sustavni pregled literature (eng. Systematic Literature Review)
VS Code	Visual Studio Code
UI	umjetna inteligencija (eng. Artificial Intelligence, AI)
API	programsko sučelje aplikacije (eng. Application Programming Interface)
SDK	razvojni paket softvera (eng. Software Development Kit)
CLI	sučelje naredbenog retka (eng. Command Line Interface)
SCCS	Sustav za upravljanje izvornim kôdom (eng. Source Code Control System)
RVS	Sustav za upravljanje revizijama (eng. Revision Control System)
CVS	Sustav za upravljanje istovremenim verzijama (eng. Concurrent Versions System)
SVN	Subversion
CCS	Konvencionalne commit poruke (eng. Conventional Commits)

Sadržaj

Kratice korištene u radu ii

1. Uvod 1
1.1. Metode i tehnike 2
1.1.1. Znanost o oblikovanju 2
1.1.2. Sustavni pregled literature 4
1.1.3. Primjena znanosti o oblikovanju u radu 5
1.2. Identifikacija problema i motivacija 6
1.3. Definiranje ciljeva rješenja 7

2. Sustav za upravljanje verzijama 8
2.1. Verzija 8
2.1.1. Semantičko verzioniranje 8
2.1.2. Verzija u kontekstu programskog kôda 9
2.2. Upravljanje verzijama 10
2.2.1. Povijest sustava za upravljanje verzijama programskog kôda 10
2.2.1.1. Source Code Control System (SCCS) 10
2.2.1.2. Revision Control System (RCS) 11
2.2.1.3. Concurrent Versions System (CVS) 11
2.2.1.4. Subversion (SVN) 12
2.2.2. Distribuirani sustav za upravljanje verzijama 12
2.2.2.1. Git 12
2.2.2.2. Kontrolni sažetak kao naziv verzije 14
2.3. Dobre prakse za pisanje Git commit poruka 14
2.3.1. Prema Torvaldsu 15
2.3.2. Prema specifikaciji Conventional Commits 16
2.3.3. Prema Git Krakenu 17
2.3.4. Metodologija procjene kvalitete commit poruka u predloženom alatu . . . 18
2.3.4.1. Odabrane smjernice za evaluaciju commit poruka 19

3. Veliki jezični modeli 21
3.1. Umjetna inteligencija i jezični modeli 21
3.2. Arhitektura i funkcioniranje velikih jezičnih modela 22
3.3. Edukativna primjena velikih jezičnih modela 23
3.4. Usporedba najvažnijih modela za analizu commit poruka 24
3.4.1. Metodologija pretrage 24

3.4.1.1. Selekcija izvora . . . 24
3.4.1.2. Ekstrakcija podataka . . . 25
3.4.1.3. Primjeri opisa metodologije studija . . . 25
3.4.1.4. Specifična evaluacijska mjerila . . . 25
3.4.1.5. Performanse generiranja kôda i analize . . . 26
3.4.1.6. Tehničke mogućnosti integracije . . . 27
3.4.2. Rezultati . . . 28
3.4.2.1. Pokrivenost modela . . . 29
3.4.2.2. Otvorenost i dostupnost API-ja . . . 29
3.4.2.3. Brzina i jednostavnost integracije u softverske alate . . . 29
3.4.2.4. Praktična upotrebljivost u razvoju alata za ocjenu kvalitete commit poruka . . . 30
3.4.3. Rasprava . . . 30
3.4.4. OpenAI GPT . . . 31
3.4.5. Google Gemini . . . 31
3.4.6. Anthropic Claude . . . 31
3.4.7. Meta LLaMA 3 . . . 32
3.4.8. Odabir modela za integraciju u predloženo programsko rješenje . . . 32

4. Dizajn i razvoj programskog rješenja . . . 34
4.1. Funkcionalnosti . . . 34
4.2. Korištene tehnologije . . . 36
4.3. Tijek korištenja alata . . . 37
4.4. Implementacija . . . 42
4.4.1. Klijentski dio . . . 43
4.4.2. Poslužiteljski dio . . . 46

5. Verifikacija i validacija . . . 51
5.1. Plan provedbe verifikacije i validacije . . . 51
5.2. Demonstracija programskog rješenja . . . 51
5.3. Evaluacija programskog rješenja . . . 52
5.3.1. Provođenje ankete za evaluaciju alata . . . 52
5.3.1.1. Pitanja . . . 52
5.3.1.2. Kvantitativni rezultati . . . 53
5.3.1.3. Kvalitativni rezultati . . . 57
5.3.2. Provođenje polustrukturiranog intervjua . . . 59
5.4. Procjena učinkovitosti . . . 60

6. Zaključak . . . 61

Popis literature . . . 65

Popis slika . . . 67

Popis tablica 68

Popis isječaka koda 69

Sažetak 70

Summary 71

1. Uvod

Tijekom razvoja programskog proizvoda, skupine promjena se bilježe i spremaju kao verzije programskog kôda, pri čemu svaka nova verzija označava određeni korak naprijed u razvoju. Te se verzije pohranjuju u repozitorij pomoću sustava za upravljanje verzijama, poput Gita, koji omogućuje vođenje povijesti izmjena, pregled promjena između svake verzije kôda, identifikaciju autora i vremena svake promjene.

Osim toga, svaki commit, odnosno zabilježena promjena, popraćen je commit porukom, što je kratki tekstualni opis promjene koji čini osnovni oblik dokumentacije o razvoju softvera. Iako iskusni programeri intuitivno pišu jasne i korisne commit poruke, studenti i manje iskusni programeri često griješe jer pišu nejasne, nepotpune ili nedosljedne poruke, što otežava timsku suradnju i razumijevanje tijeka razvoja. Posebno predstavlja problem u kasnijim analizama kôda i promjenama na kôdu, te može višestruko smanjiti buduću produktivnost timova.

Upravo radi toga se javlja potreba za sustavom koji može analizirati kvalitetu commit poruka i pružiti korisne povratne informacije. Takav sustav bio bi posebno koristan u edukacijskom kontekstu jer bi pomogao studentima da razviju profesionalne navike u dokumentiranju, a mentorima bi olakšao nadzor nad radom studenata te smanjio količinu ručnog pregledavanja svake commit poruke radi davanja personaliziranih povratnih informacija svakom studentu.

U ovom radu pristupa se razvoju takvog sustava korištenjem metodologije znanosti o oblikovanju (eng. design science), koja omogućuje znanstveno utemeljen razvoj programskih artefakata za rješavanje konkretnih problema. S obzirom na to da su commit poruke izrađene u prirodnom jeziku, za njihovu analizu koristi se tehnologija velikih jezičnih modela (eng. Large Language Model, LLM). Ti su modeli trenirani na velikim količinama tekstualnih podataka i prilagođavaju odgovore sukladno kontekstu, što ih čini idealnima za primjenu u ovakvom edukativnom kontekstu za ocjenjivanje tekstualne dokumentacije poput commit poruka.

Putem API-ja (eng. Application Programming Interface) moguće je LLM-u zadati specifične kriterije evaluacije commit poruka, a povratna informacija može uključivati komentare o jasnoći, sažetosti ili stilskoj dosljednosti. Na taj način LLM postaje dijelom sustava koji služi kao edukativni asistent, koji studentima omogućuje trenutačnu analizu njihovog rada, a pritom ih usmjerava prema boljoj dokumentacijskoj praksi.

Rješenje razvijeno u ovom radu dizajnirano je metodologijom znanosti o oblikovanju, te je zamišljeno kao proširenje za Visual Studio Code s integracijom prema GitHub API-ju i vanjskom LLM uslugom putem API-ja. Sustav dohvaća commit poruke iz odabranog projektnog repozitorija, prosljeđuje ih LLM-u uz definirane kriterije evaluacije te korisniku prikazuje strukturiranu povratnu informaciju i ocjenu na unaprijed definiranoj skali. Implementacija je modularna kako bi se alat mogao prilagoditi različitim edukacijskim okruženjima i jednostavno integrirati u postojeće radne tokove.

U skladu s načelima znanosti o oblikovanju, rad uključuje sljedeće istraživačke faze:

- identifikaciju problema neadekvatnih commit poruka među studentima,

- definiranje ciljeva rješenja u obliku alata za evaluaciju commitova,
- dizajn i implementaciju tog alata korištenjem integracije s velikim jezičnim modelom,
- demonstraciju funkcionalnosti alata kroz praktične primjere i fokus grupu,
- evaluaciju kvalitete i učinkovitosti alata primjenom fokus grupa sa stručnjacima, nastavnicima i studentima, uključujući raspravu o rezultatima, doprinosima i mogućnostima daljnjeg razvoja,
- komunikaciju rezultata i doprinosa istraživanja kroz znanstveni rad i dokumentaciju.

Izrađeno programsko rješenje se jednostavno instalira iz razvojnog okruženja Visual Studio Code. Alatu se može pristupiti i na javnom repozitoriju preko poveznice: `https://github.com/jfletcher20/commitment-issues`.

1.1. Metode i tehnike

U ovom radu koristi se metodologija **znanosti o oblikovanju** (eng. Design Science Research Methodology, DSRM) [1], koja je posebno prikladna za istraživanja čiji je cilj razvoj i evaluacija korisnih artefakata. U kontekstu ovog istraživanja riječ je o razvoju edukativnog alata za analizu Git commit poruka uz pomoć velikog jezičnog modela. Znanost o oblikovanju temelji se na iterativnom ciklusu koji povezuje stvarne probleme s razvojem rješenja, pri čemu se znanstveno validira korisnost i učinkovitost predloženog rješenja.

Osim metodologije znanosti o oblikovanju, proveden je **sustavni pregled literature** (eng. systematic literature review, SLR) [2] sa ciljem komparativne analize nekoliko poznatih velikih jezičnih modela, kako bi se odabrao najprikladniji model za potrebe našeg rješenja. Također je analizirana relevantna literatura o dobrim praksama pisanja commit poruka, na temelju koje je oblikovan skup smjernica. Te smjernice čine osnovu za standardizirano ocjenjivanje i generiranje povratnih informacija u okviru analize studentskih commit poruka.

1.1.1. Znanost o oblikovanju

U ovom istraživanju odabran je pristup znanosti o oblikovanju zbog prikladnosti za istraživačke zadatke vezane za razvoj inovativnih programskih rješenja. Ovaj istraživački pristup se fokusira na dizajn i evaluaciju proizvoda osmišljenih za rješavanje problema identificiranih u praksi [1].

Kako bi se takva istraživanja mogla provoditi sustavno i transparentno, Peffers et al. [1] su predložili ovu metodologiju. Ona definira šest temeljnih aktivnosti koje se mogu slijediti u razvoju znanstveno utemeljenog proizvoda, uključujući softverska rješenja:

1. **Identifikacija problema i motivacija** – Definiranje problema koji zahtijeva rješenje te obrazloženje njegove važnosti. Ova faza usmjerava istraživanje prema stvarnim potrebama korisnika.

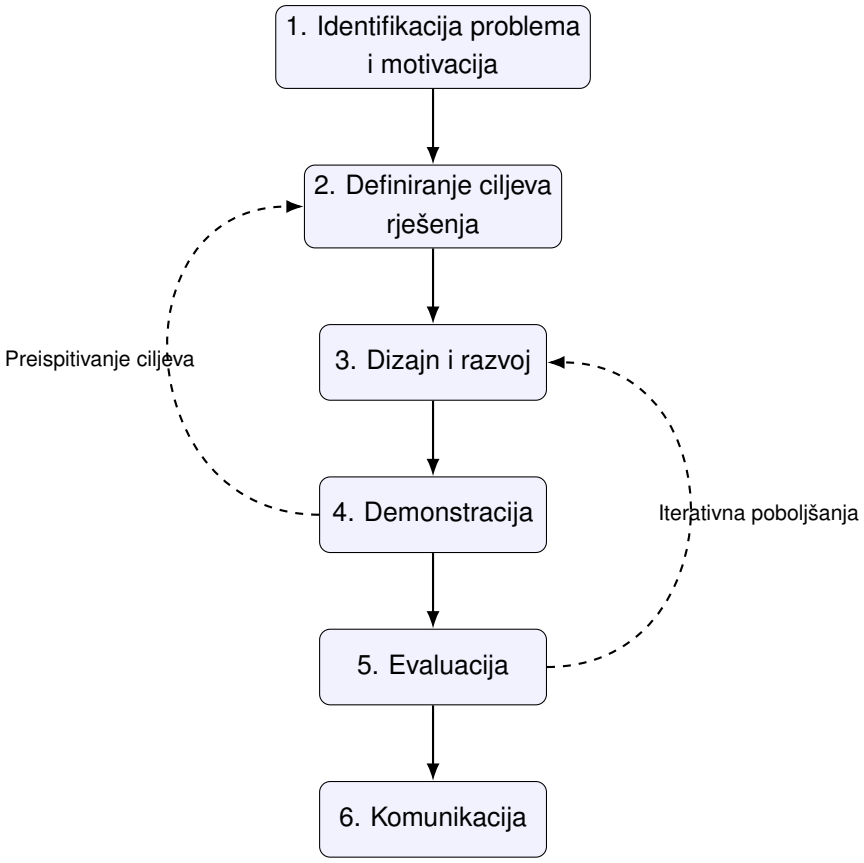
2. **Definiranje ciljeva rješenja** – Postavljanje mjerljivih ciljeva koje proizvod treba zadovoljiti. Ciljevi se izvode iz razumijevanja problema i postojećih rješenja.

3. **Dizajn i razvoj** – Izrada proizvoda ili artefakta koji može biti model, metoda, programski proizvod ili druga vrsta tehnološkog rješenja koje je temeljeno na postojećem znanju i teoriji.

4. **Demonstracija** – Ova faza je također poznata kao verifikacija i služi kao primjena artefakta u kontekstu stvarnog problema kako bi se pokazala njegova funkcionalnost, od jednostavnog prikaza da rješenje "radi" u praksi, pa sve do opsežnije formalne evaluacije gotovog artefakta. Neki autori poput Eekelsa i Roozenburga [1] kombinirali su obje faze u svojim radovima. To može uključivati eksperimente, simulacije, studije slučaja ili druge načine praktične primjene. Važno je da autori znaju kako alat koristiti u stvarnoj situaciji da bi jasno pokazali njegovu funkcionalnost.

5. **Evaluacija** – Ova faza je također poznata kao validacija i služi kao procjena uspješnosti artefakta u ispunjavanju zadanih ciljeva. Ovisno o vrsti, evaluacija može ići od usporedbe funkcionalnosti s definiranim ciljevima, preko mjerenja performansi, do prikupljanja povratnih informacija kroz ankete, intervjue ili simulacije [1]. Može biti kvantitativna (npr. vrijeme odziva, broj grešaka) ili kvalitativna (npr. zadovoljstvo korisnika), a ovisno o rezultatima istraživači mogu odlučiti hoće li dodatno dorađivati artefakt ili nastaviti prema sljedećoj fazi.

6. **Komunikacija** – Prikazivanje rezultata istraživanja i samog artefakta znanstvenoj i stručnoj zajednici kroz akademske radove, prezentacije ili tehničku dokumentaciju.



Slika 1: Model procesa znanosti o oblikovanju (DSRM), prilagođeno prema [1].

DSRM ne zahtijeva strogo linearnu provedbu. Istraživač može započeti u bilo kojoj fazi, ovisno o prirodi problema i dostupnim resursima, što metodologiju čini fleksibilnom i prilagodljivom različitim istraživačkim kontekstima.

Upotrebom DSRM pristupa u ovom radu osigurava se strukturiran i znanstveno utemeljen razvoj alata za analizu commit poruka, čime se doprinosi obrazovnom procesu i praksi razvoja softvera. Takav okvir omogućuje transparentnost u planiranju, izradi i validaciji rješenja koje ima stvarnu primjenu u edukacijskom kontekstu. Osim primjene na konkretan problem s kojim se susreću studenti i mentori, uporaba ove metodologije doprinosi i razvoju znanstvenih znanja o primjeni umjetne inteligencije u obrazovanju i programskom inženjerstvu.

1.1.2. Sustavni pregled literature

Sustavni pregled literature predstavlja strukturirani i transparentni postupak identifikacije, selekcije, vrednovanja i sinteze postojećih istraživanja, sa ciljem da se odgovori na unaprijed definirano istraživačko pitanje [2]. Za razliku od tradicionalnih narativnih pregleda, sustavni pregled nastoji biti što objektivniji kako bi proces bio ponovljiv kroz jasno definirane korake, odnosno kriterije pretrage.

U kontekstu ovog istraživanja, sustavni pregled literature proveden je uz pomoć alata **Elicit** [3], koji koristi metode strojne obrade teksta za automatsku pretragu i analizu znanstvenih radova. Svrha ovog pregleda bila je komparativna analiza najpopularnijih velikih jezičnih

modela kako bismo odabrali model za integraciju u predloženo programsko rješenje, koji će vršiti analizu commit poruka. Radi toga smo formulirali istraživačko pitanje te kriterije po kojima uspoređujemo modele na temelju dostupnih studija.

Kao istraživačko pitanje definirali smo: „Usporedi velike jezične modele GPT-4 (OpenAI), Claude 3 (Anthropic), Gemini (Google) i LLaMa (Meta) prema kriterijima: otvorenost i dostupnost API-ja, brzina i jednostavna integracija u softverske alate, podrška za razumijevanje programskog kôda i commit poruka, te praktična upotrebljivost u razvoju alata za ocjenu kvalitete commit poruka."

Proces sustavnog pregleda putem Elicita slijedio je standardne faze metodologije. Najprije je provedena **selekcija izvora** relevantnih radova i tehničke dokumentacije. Potom je izvršena **ekstrakcija podataka** kroz strukturirane tablice koje je alat generirao, uključujući **primjere formata** i **specifična evaluacijska mjerila**. Dodatno su analizirane **performanse generiranja kôda i analize** te **tehničke mogućnosti integracije** pojedinog modela. Na temelju toga oblikovani su rezultati kroz nekoliko kategorija: **pokrivenost modela**, **otvorenost i dostupnost API-ja**, **brzina i jednostavnost integracije**, te **praktična upotrebljivost u edukativnom kontekstu**, posebno u razvoju alata za evaluaciju commit poruka.

Na taj način alat Elicit omogućio je strukturirano prikupljanje i analizu podataka, što je značajno ubrzalo i pojednostavilo provedbu sustavnog pregleda literature za definirano istraživačko pitanje. Provedba sustavnog pretraživanja literature predstavljena je u kontekstu usporedbe velikih jezičnih modela, koji su obrađeni u poglavlju 3.

1.1.3. Primjena znanosti o oblikovanju u radu

U sljedeća dva poglavlja (Poglavlja 1.2 i 1.3) definirani su problem i ciljevi rješenja. Također je u poglavlju 2 analiziran kontekst u kojem commit poruke nastaju, kroz prikaz načina rada sustava za upravljanje verzijama i uloge commit poruka unutar tih sustava, što pomaže u jasnijem definiranju konteksta problema.

U nastavku tog poglavlja je provedeno istraživanje relevantne literature i preporuka stručnjaka o dobrim praksama pisanja commit poruka. Kao rezultat, oblikovane su odabrane smjernice koje služe kao standardizirani okvir na temelju kojeg veliki jezični model u našem rješenju može dosljedno ocjenjivati različite vrste commit poruka. Oblikovanjem ovih smjernica se pristupa fazi dizajna softverskog rješenja za identificirani problem.

Ova faza metodologije se nastavlja kroz poglavlje 3, u kojem je prikazano kako je uz pomoć alata za pretraživanje i analizu znanstvene literature Elicit provedeno sustavno pretraživanje literature sa ciljem usporedbe dostupnih velikih jezičnih modela prema unaprijed definiranim kriterijima. Na temelju tog pregleda odabran je model koji najbolje odgovara zahtjevima našeg rješenja.

Nakon toga je izrađena softverska arhitektura i implementirano rješenje korištenjem odabranih tehnologija. Ovi koraci su detaljno opisani u poglavlju 4 i njima završava treća faza metodologije, odnosno dizajn i razvoj rješenja.

Potom su provedene sljedeće dvije faze metodologije, demonstracija i evaluacija, koje se u okviru metodologije znanosti o oblikovanju smatraju ključnima za provjeru valjanosti i korisnosti razvijenog artefakta. One su prikazane u poglavlju 5, koje opisuje način provedbe verifikacije i validacije rješenja putem fokus grupe koja je uključivala ukupno šest sudionika podijeljenih u dvije skupine, stručnjake i krajnje korisnike. Skupinu stručnjaka činila su dva sveučilišna nastavnika i tri asistenta s iskustvom u poučavanju kolegija vezanih uz programsko inženjerstvo te jedan stručnjak iz industrije s višegodišnjim iskustvom u razvoju softverskih rješenja i timskom radu. Korisničku skupinu činilo je osmero studenata različitih razina studija s većim ili manjima iskustvom u korištenju Gita u fakultetskim projektima. Rad s fokus grupom je bio podijeljen u dvije faze, gdje je rješenje prvo demonstrirano članovima fokus grupe kako bi se prikazale funkcionalnosti alata. Nakon toga, sudionicima je omogućeno da isprobaju alat na vlastitim repozitorijima. Potom su sudionici ispunili anketni upitnik i/ili sudjelovali u polustrukturiranom intervjuu, kroz koji se dobivao detaljniji uvid u njihova iskustva s korištenjem alata. Ova faza je tako omogućila uvid u to kako se alat ponaša u praksi i koliko ga korisnici intuitivno razumiju i prihvaćaju.

U posljednjoj fazi metodologije, rezultati istraživanja i sam alat dokumentirani su u ovom radu te će biti predstavljeni i kroz druga stručna i akademska sredstva kao što su konferencijska izlaganja i primjena na kolegijima vezanim uz programsko inženjerstvo. Na taj način doprinosi se širenju znanja u području računarstva, s posebnim naglaskom na edukaciju budućih programera i primjenu naprednih tehnologija umjetne inteligencije (UI, eng. Artificial Intelligence, AI) u obrazovne svrhe.

1.2. Identifikacija problema i motivacija

Pri razvoju programskog proizvoda, kvalitetna dokumentacija je ključan aspekt kako bi se osigurala razumljivost, održivost i poboljšana mogućnost suradnje između članova razvojnih timova.

Git commit poruke, kao temeljni oblik dokumentacije napretka u razvoju konkretnog softverskog proizvoda, često predstavljaju izazov za studente i programere s ograničenim iskustvom u profesionalnim okruženjima. Uobičajene pogreške su da commit poruka nije dovoljno sažeta, opisivanje nebitnih informacija, nejasno i nedovoljno objašnjenje promjena u programskom kôdu, nedosljednost u stilu pisanja commit poruka, nepoštivanje smjernica organizacije ili institucije kojemu pripadaju, nedostatak poveznice na zadatak (eng. task, issue) na koju se odnosi . Ove poteškoće otežavaju praćenje razvojnog procesa, ugrožavaju održivost programskog proizvoda te učinkovitost timskog rada.

Brojne studije ukazuju na to da loše napisane Git commit poruke štete timu i razvoju sustava. Na primjer, Tian i suradnici [4] utvrdili su da je čak oko 44% commit poruka u aktivnim projektima otvorenog kôda nedovoljno informativno ili nejasno, što otežava suradnju i održavanje kôda. Kako kvaliteta dokumentacije direktno utječe na suradnju i produktivnost razvojnih timova, ovo nosi implikacije i za otežano poštivanje rokova i budžeta softverskih projekata u mnogim kompanijama. Najpoznatija i najcitiranija studija o toj temi je The Standish Group

CHAOS Report [5], koji od 1990-ih prati uspješnost softverskih projekata. Njihovi nalazi često navode da većina softverskih projekata premašuje rokove i budžet, a značajan dio ih uopće ne uspije.

U obrazovnom kontekstu, Ma i Lopes [6] pokazali su da studenti često pišu kratak i nejasan commit, ali da se kvaliteta poruka može znatno poboljšati jednostavnim poticajima u korisničkom sučelju. Problemi sa studentskim commit porukama mogu prozilaziti i iz činjenice da mentori u programskim kolegijima često nemaju vremena posvetiti se detaljnoj analizi svake pojedine commit poruke.

Stoga postoji potreba za pojašnjenjem dobrih praksi za pisanje Git commit poruka te za alatom koji potiče unaprjeđenje vještine pisanja commit poruka, posebice za studente i nedovoljno iskusne programere. Srećom, LLM-ovi nam nude mogućnost personalizacije i automatizacije recenzije commit poruka, što naš alat nudi kao rješenje primjenom principa znanosti o oblikovanju.

1.3. Definiranje ciljeva rješenja

Definiranje ciljeva jedan je od ključnih koraka u procesu znanosti o oblikovanju jer usmjerava istraživanje prema konkretnim i mjerljivim ishodima. Polazišna točka ovoga rada jest prepoznati problem neadekvatnih commit poruka među studentima i kroz metodološki utemeljen okvir razviti rješenje, pritom obuhvaćajući teorijske, metodološke i praktične aspekte istraživanja. Prema tome, ciljevi uključuju ne samo razvoj programskog artefakta, već i njegovo pozicioniranje u obrazovnom kontekstu te evaluaciju njegove učinkovitosti. Na taj se način nastoji osigurati da rješenje bude znanstveno utemeljeno, tehnički izvedivo i pedagoški korisno.

- Identificirati i sistematizirati dobre prakse u pisanju Git commit poruka kao temelja profesionalne dokumentacijske prakse.
- Definirati kriterije evaluacije commit poruka koji se mogu interpretirati i provoditi pomoću velikog jezičnog modela.
- Obaviti sustavno pretraživanje literature za usporedbu nekoliko velikih jezičnih modela prema definiranim kriterijima te odabir najprikladnijeg modela za integraciju s predloženim rješenjem.
- Razviti edukativni alat koji omogućuje korisnicima analizu commit poruka uz povratnu informaciju.
- Pružiti podršku studentima u učenju, a mentorima u poučavanju složenih koncepata povezanih s verzioniranjem programskog kôda pri poučavanju razvoja softverskih proizvoda
- Potaknuti refleksiju kod studenata o svrsi i kvaliteti pisanih commit poruka kako bi poboljšali vlastite vještine dokumentiranja softverskog razvoja.

2. Sustav za upravljanje verzijama

Kako bismo pojasnili složenost učenja i poučavanja koncepata povezanih sa verzioniranjem programskog kôda te pripremom i opisom verzija kôda, u ovom poglavlju pojasnit ćemo što je sustav za upravljanje verzijama i od kojih se sve koncepata sastoji. Dobrim razumijevanjem sustava za verzioniranje razjasnit će se važnost razvoja vještine opisa verzija kôda.

Sustav za upravljanje verzijama se odnosi na sustave i alate razvijene za upravljanje verzijama izvornog kôda (eng. source code control). U sljedećim potpoglavljima bit će detaljnije opisani pojmovi verzija i verzioniranje, kratka povijest sustava za upravljanje verzijama, razvoj skalabilnog distribuiranog sustava Git i njegov utjecaj na današnji svijet te dobre prakse prilikom pisanja poruka o verzijama unutar Git sustava. Kod dobrih praksi bit će prikazane preporuke iz literature te nekoliko široko prihvaćenih izvora u programskoj zajednici. Na temelju njih definirat će se skup najvažnijih dobrih praksi koji će poslužiti kao standard za evaluaciju u poučavanju programskog inženjerstva putem predloženog programskog rješenja.

2.1. Verzija

Verzija predstavlja stabilno (ili čak nestabilno) stanje nekog proizvoda u datom trenutku u vremenu, bilo da je riječ o fizičkom proizvodu poput knjige (različita izdanja pojedine knjige) ili digitalnom proizvodu poput softvera. Verzije izdanja programskog proizvoda često prate konvenciju semantičkog verzioniranja (eng. semantic versioning) pomoću koje je moguće procijeniti značaj promjena između verzija, što čini razumijevanje ove konvencije ključnom u strukturiranom dokumentiranju i praćenju kôda.

2.1.1. Semantičko verzioniranje

Prema specifikacijama semantičkog verzioniranja koju je inicijalno razvio Tom Preston-Werner 2011., svaka verzija programskog proizvoda slijedi format:

$$MAJOR.MINOR.PATCH$$

- **MAJOR** verzija, kada je promjena u API-ju ili programskom proizvodu toliko velika da nije kompatibilna sa starom verzijom
- **MINOR** verzija, kada se nadograđuje funkcionalnost na način koji je kompatibilan s prošlom verzijom programa (eng. backwards compatible)
- **PATCH** verzija, kada se popravlja greška u nekoj funkcionalnosti na način koji je kompatibilan s prošlom verzijom programa

Pritom MAJOR, MINOR i PATCH moraju biti obilježeni brojevima [7]. Prema semantičkom verzioniranju, $0.x.y$ predstavlja nestabilno pred-izdanje (eng. pre-release) programskog

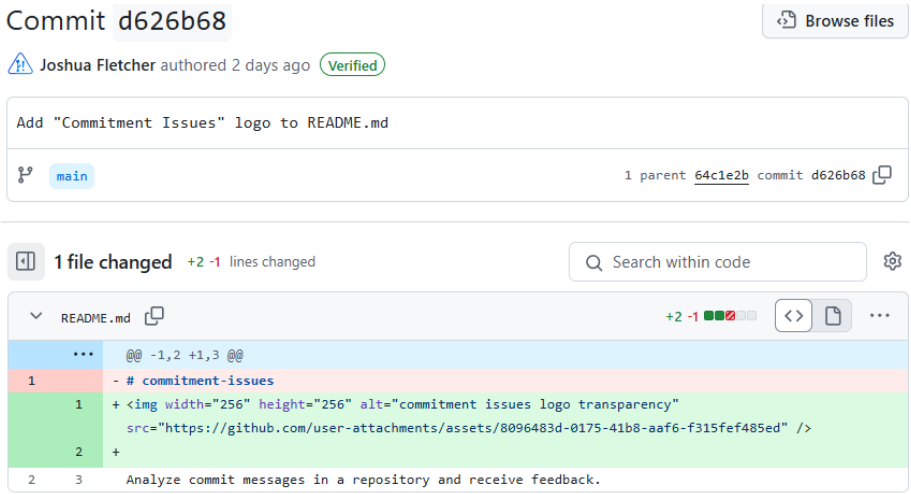
proizvoda koje ima česta ažuriranja i namijenjen je za fazu inicijalnog razvoja programskog proizvoda, gdje su x i y proizvoljni prirodni brojevi. Prva stabilna verzija programskog proizvoda nosi verziju $1.0.0$. Nakon izdanja konkretne verzije programa, svako ažuriranje uvijek povećava odgovarajući broj verzije s obzirom na promjenu.

Dodatno se oznaka verzije može proširiti metapodacima koji se odnose na verziju izgradnje (eng. build version metadata). Ona se zapisuje nakon oznake "+": $v1.3.2 + 3$.

Metapodaci o verziji izgradnje se često koriste za obilježavanje stanja ili namjene određene verzije, kao što su alfa i beta za testiranje ili predizdanje (eng. pre-release).

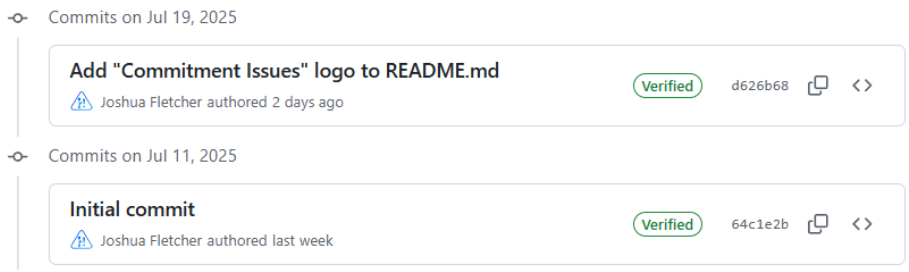
2.1.2. Verzija u kontekstu programskog kôda

U razvoju programskog proizvoda, pojam "verzija" obilježava različito stanje jedne ili više digitalnih datoteka u odnosu na drugo stanje.



Slika 2: Primjer promjena nastalih u commitu d626b68

Promjena stanja, kao što se vidi iz slike 2, u kontekstu programskog kôda može biti promjena u strukturi direktorija ili datoteka, dodavanje, brisanje ili mijenjanje sadržaja datoteka repozitorija. Pakiranje manjeg ili većeg skupa promjena pod određenim znakom i porukom u sustavu za upravljanje verzijama označava spremanje nove verzije programskog kôda. Verzije u programskom kôdu, dakle, označavaju korak napretka programskog kôda u određenom smjeru razvoja. Danas se standardno bilježi i poruka (eng. message) koja ukratko opisuje nastale promjene u svakoj verziji programskog kôda (slika 3).



Slika 3: Primjer promjena nastalih u commitu d626b68

2.2. Upravljanje verzijama

Sustavi za upravljanje verzijama programskog kôda danas predstavljaju jednu od najvažnijih tehnologija kojima se koriste programeri u svakidašnjem razvoju programskih proizvoda, do te mjere da su neizostavan dio svakog razvoja softvera, pogotovo timskog ili profesionalnog. Efikasnim upravljanjem različitim verzijama programskog kôda moguće je poništiti promjene ili oporaviti staro stanje. Isto tako, može se čuvati stabilno stanje programskog kôda, odnosno ono se može spakirati u program od kojeg se očekuje relativno stabilno ponašanje, za razliku od onog kod kojeg je pretpostavljena manja ili viša razina neočekivanog ponašanja.

2.2.1. Povijest sustava za upravljanje verzijama programskog kôda

Postoji više sustava za upravljanje verzijama koji su se razvili kroz povijest i koji postoje danas, ali istaknut će se značajke samo pojedinih koji se smatraju najbitnijima.

2.2.1.1. Source Code Control System (SCCS)

Prvi potpuni sustav za upravljanje verzijama programskog kôda je bio Sustav za upravljanje izvornim kôdom (eng. Source Code Control System, SCCS), koju je razvio Mark Rochkind 1972. godine. SCCS verzioniranje omogućuje pohranom delti, odnosno razlika stanja u datotekama između verzija prije i nakon promjena [8].

```
1 static char SccsId[ ] = "@(#)program.c   1.2 08/29/80";
```

Isječak koda 1: SCCSID primjer. Preuzeto iz [9, poglavlje 5].

Kao što se vidi iz isječka kôda 1, zapisi u SCCS-u su počinjali s "@(#)" te su se prema tome mogli identificirati i pretraživati, slično današnjoj upotrebi "#" na društvenim mrežama za kategoriziranje vlastitih objava radi njihovog lakšeg pretraživanja. Mogli su se dodavati i komentari (poruke) o promjenama u zadanoj verziji, kao što se vidi iz isječka kôda 2.

```
$ sccs delta program.c
comments? corrected typo in widget(), \
null pointer in n_crunch()
1.2
5 inserted
3 deleted
84 unchanged

$ sccs prt program.c
D 1.2                80/08/29
↪  12:35:31                                                pers                2        1
corrected typo in widget(),
null pointer in n_crunch()

D 1.1                79/02/05
↪  00:19:31                                                zeno                1        0
date and time created 80/06/10 00:19:31 by zeno
```

Isječak koda 2: Primjer korištenja SCCS sustava. Preuzeto iz [9, poglavlje 5].

Jedan veliki nedostatak SCCS sustava je da nije moglo više korisnika istovremeno otvoriti i ažurirati sadržaj pojedine datoteke.

2.2.1.2. Revision Control System (RCS)

Walter F. Tichy je svojim sustavom za upravljanje revizijama (eng. Revision Control System, RCS) 1982. godine stvorio sustav s inovacijama reverznih odvojenih delti (eng. separate reverse deltas).

Tichy pojašnjava da je granularnost promjena koje zapisuje RCS na razini linije kôda, odnosno, ako se bilo koji znak mijenja na konkretnoj liniji, smatra se kao da je cijela linija bila izmijenjena. Taj pristup je bio odabran zbog UNIX programa *diff* koji delte izračunava na osnovi linija-po-linija. Također ističe da, za razliku od SCCS-a koji koristi spojene delte (eng. merged deltas), RCS koristi odvojene delte (eng. separate deltas), što je optimizacija koju je detaljno opisao u svome radu [10, poglavlje 3].

2.2.1.3. Concurrent Versions System (CVS)

U početku je istovremeni sustav verzija (eng. Concurrent Versions System, CVS) postojao samo kao skup skripti koje je napisao Dick Grune i koje su koristile naredbe RCS-a, ali se na kraju razvio u zasebni sustav 1986. godine [11]. Glavni napredak u odnosu na RCS je mogućnost suradivanja više korisnika na istim datotekama istovremeno.

CVS koristi arhitekturu klijent-poslužitelj (eng. client-server architecture), odnosno radi se o centraliziranom sustavu. Centralizirani sustavi karakterizirani su po centralnom poslužitelju koji služi kao jedinstveni izvor istinskog stanja repozitorija, odnosno programskog kôda. Na lokalnom uređaju samo se koristi radna kopija repozitorija s poslužitelja. Klijenti se spajaju

na poslužitelj kako bi lokalno spremili radnu kopiju s najnovijim podacima s poslužitelja (eng. check out) i kasnije svoje promjene radnih kopija pohranili na poslužitelju (eng. check in), što omogućuje suradnju među razvojnim timovima, ali ujedno i služi kao sigurnosna mjera protiv gubitka repozitorija s obzirom na postojanje redundantnih kopija.

2.2.1.4. Subversion (SVN)

Kao oblik centraliziranog sustava za upravljanje verzijama, na poticaj tvrtke CollabNet 2000. godine, Karl Fogel i Jim Blandy su s drugim članovima CollabNet tima započeli s razvojem Subversiona (SVN). Zadržao je mnoge funkcionalnosti koje je nudio CVS, sa ciljem da olakša prijelaz onima koji su naviknuti na CVS. Danas nosi naziv "Apache Subversion" te se i dalje razvija kao projekt otvorenog kôda [12].

Za razliku od CVS-a koji prati napretke na razini datoteka, SVN svaki commit gleda kao novu reviziju te je lakše pratiti povijest razvoja projekta, štoviše, SVN commit tretira kao transakciju, tzv. atomski commit (eng. atomic commit) te se ili uspješno izvrši commit ili se ne commita uopće. Ovakav pristup osigurava konzistentnost sustava, što je bio nedostatak CVS sustava gdje je bilo moguće izgubiti podatke commita.

Subversion se i danas koristi u nekim projektima te se smatra boljim od daleko popularnijeg Gita za slučajeve gdje se radi s velikim datotekama.

2.2.2. Distribuirani sustav za upravljanje verzijama

Distribuirani sustavi, za razliku od centraliziranih, nemaju pojedini centralni poslužitelj, nego su bazirani na arhitekturi klijent-klijent (eng. peer-to-peer, P2P) prema kojemu su klijenti povezani jedni s drugima i direktno komuniciraju međusobno, umjesto da moraju preko centralnog poslužitelja kouminicirati, i na taj način se smanjuje opterećenje koje bi ono moralo podnijeti.

Godine 2000. izlazi BitKeeper pod tvrtkom BitMover, komercijalni proizvod i prvi skalabilni distribuirani sustav za upravljanje verzijama [13]. Koristio se često i u projektima otvorenog kôda (eng. open-source projects), uključujući Linux projekt [14].

2.2.2.1. Git

Linus Torvalds je 1991. počeo s razvojem operacijskog sustava Linux. Projekt je bio otvorenog kôda i dok je rasla njegova popularnost, rasla je i potreba za dobrim alatima za međunarodnu suradnju. 2005. godine, dok su surađivali korištenjem BitKeepera, izgubili su licencu za njegovo korištenje te je Linusa to potaknulo da razvije vlastiti alat za distribuirano upravljanje verzijama. Rezultat tog pothvata je Git [14].

Git je distribuirani sustav za upravljanje verzijama, ali je i otvorenog kôda. Svi mogu pridonijeti njegovom razvoju i vidljiva je cijela povijest, do prvog commita ikad (slika 4).

Slika 4: Prvi Git commit je sadržavao sam Git

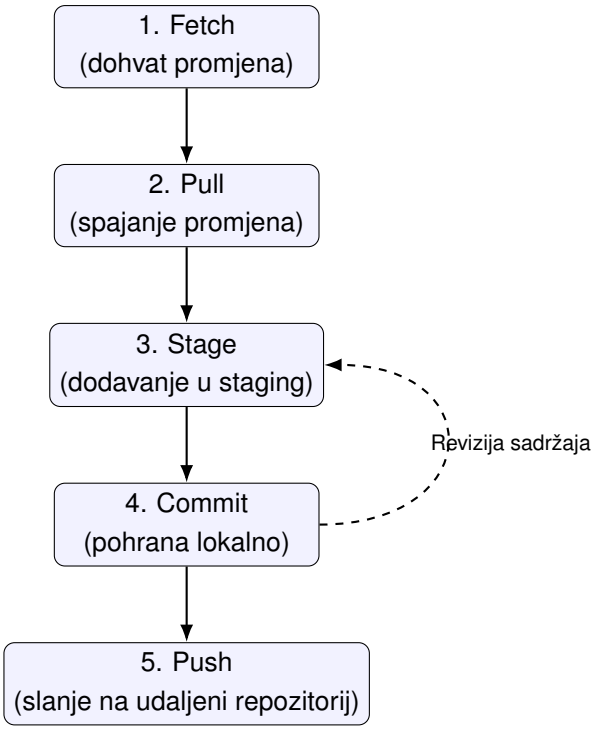
Valja napomenuti da, iako Git nudi sposobnost korištenja P2P arhitekture, popularne platforme kao što su GitHub često potiču donekle centralizirani pristup njegovom korištenju. Unatoč tome, jedna od velikih razlika između Git-a i SVN-a je lokacija izvođenja operacija.

Unutar Git sustava, bez obzira na platformu za njegovo korištenje, različite operacije - kao što su poništavanje commita (eng. revert), rješavanje konflikta (eng. merge conflicts), stvaranje grane (eng. branch) odnosno novog kanala za tok commitova unutar repozitorija, spajanje grana i slično - izvršavaju se na pojedinom čvoru (računalu, klijentu), nakon čega se promjene prenose na ostale čvorove. U SVN sustavu sve se te promjene moraju izvoditi na poslužitelju [15].

Git dominira tržište današnjice. Prema anketama programera, 2025. Git koristi 93.87% programera [15]. Najpopularnije platforme za sustave za upravljanje verzijama GitHub, GitLab i Bitbucket imaju isključivo potporu za Git.

Slika 5 prikazuje osnovni tijek rada u Gitu, što zahtijeva razumijevanje osnovnih pojmova, posebno imajući u vidu korištenje platforme GitHub u kontekstu ovog projekta:

- Remote (hrv. druga lokacija) - lokacija repozitorija na poslužitelju na koju se inicijalno spajaju klijenti kako bi lokalno pohranili repozitorij
- Fetch (hrv. dohvati) - dohvaća informacije o postojanju promjena na poslužitelju
- Pull (hrv. povuci) - preuzima promjene s poslužitelja u lokalni repozitorij
    - Pull Request (PR) (hrv. zahtjev za povlačenjem) – mehanizam koji omogućuje da se promjene iz jedne grane predlože za spajanje u drugu granu, uz pregled i odobrenje ostalih članova razvojnog tima
- Stage (hrv. pripremi) - određivanje koje će od lokalnih promjena biti spremljene u sljedeću verziju (commitu)
- Commit (hrv. spremi) - spremanje promjena koje su u pripremi (eng. staged) kao nove verzije programskog kôda
- Push (hrv. poguraj) - prijenos lokalnih novih verzija na poslužitelj



Slika 5: Osnovni tijek rada s Gitom.

2.2.2.2. Kontrolni sažetak kao naziv verzije

Kao što se vidi iz slike 3, verzije u programskom kôdu u Git sustavu su označene pomoću kontrolnog sažetka (eng. checksum, hash).

d626b68 je skraćeno od SHA-1 sažetka:

d626b68670d02c02d1bfc92e5690521db1d14671

To je automatski generirani kontrolni sažetak koji je Git dodjelio prilikom spremanja commita, koji služi za osiguravanje integriteta verzije prilikom njegovog prijenosa i pregleda, kako bi pokušaji njegove naknadne izmjene ili korupcija podataka u prijenosu bili lakši za otkrivanje. Kontrolnim sažecima je osiguran integritet povijesti Git commitova.

Budući da kontrolni sažetak sam po sebi nije dovoljno informativan, GitHub u prvi plan stavlja commit poruku kojom korisnik opisuje promjene u repozitoriju. Iz tog razloga je od velike važnosti pisati dobre commit poruke.

2.3. Dobre prakse za pisanje Git commit poruka

Tijekom godina oblikovan je niz preporuka za pisanje i strukturiranje Git commit poruka. U nastavku ćemo prikazati preporuke iz različitih izvora, usporediti ih te na temelju komparativne analize izvući opća pravila dobre prakse koja odražavaju konsenzus među autorima. Kao izvori korištene su najčešće citirane konvencije iz literature te autoriteti koje je programska zajednica povijesno i široko prihvatila, poput Torvaldsa. Pravila dobivena međusobnim usklađivanjem i unakrsnim referenciranjem ovih izvora poslužit će kao temelj za vrednovanje commit

poruka u alatu namijenjenom edukaciji u području programskog inženjerstva.

2.3.1. Prema Torvaldsu

Linus Torvalds je kao autor Gita napisao u jednom commitu svoju perspektivu o tome što čini dobru commit poruku. Motivaciju je opisao unutar same commit poruke [16].

Slika 6: Torvaldsova commit poruka objašnjava motivaciju opisivanja praksi pisanja dobrih commit poruka [16]

Prijevod commit poruke iz slike 6 glasi:

"*[Zaglavlje] Dodaj informacije o pravilno formatiranim commit porukama*

[Tijelo] Čini se da mnogi korisnici GitHuba nisu navikli na dobra pravila za commit poruke i možda nikada nisu koristili Git za projekt kojem je stalo do dobrih zapisnika i urednih sažetaka u linijama.

[Podnožje] Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>"

"Signed-off-by" u prijevodu znači "potpisuje", ali se ne prevodi u citatu jer se radi o Gerrit preporukama koje Torvalds slijedi prilikom pisanja podnožja Git poruka [17]. Iz commita jasno se vidi što je Torvalds smatrao dobrim općim smjernicama za pisanje commit poruka [16]:

Unutar sadržaja promjena navedenog commita vidimo Linusove preporuke za pisanje dobrih commit poruka:

- "*[Pravilo] linija zaglavlja: objašnjava commit u jednoj liniji*
- *Tijelo commit poruke je nekoliko redaka teksta, objašnjava detaljnije o promjeni, moguće je dati pozadinu o problemu koji se rješava i slično.*

- *Tijelo commit poruke može imati nekoliko odlomaka, i molimo vas da pravilno prelamate riječi i da stupci budu kraći od oko 74 znaka otprilike. Na taj način će 'git log' prikazivati stvari lijepo čak i kada su uvučene linije.*
- *Reported-by (hrv. objavljuje): onaj-koji-je-objavio*
- *Signed-off-by (hrv. potpisuje): Tvoje Ime tvojemail@tvojposlužitelj.com*
- *gdje linija zaglavlja zaista treba biti smislena, i zaista treba biti samo jedna linija. Ta linija zaglavlja je ono što prikazuju alati poput gitk i shortlog, i treba sažeti promjenu u jednoj čitljivoj liniji teksta, neovisno o dužem objašnjenju.*"

Kasnije je u istom repozitoriju ažurirana uputa. Thiago Macieira ažurira sadržaj s dodatnim uputama za zaglavlje i tijelo commit poruka [18]:

- "*[Pravilo] linija zaglavlja: objasni commit u jednoj liniji (koristi imperativ)*
- *[Pravilo za tijelo commit poruke] pobrinite se da objasnite svoje rješenje i zašto radite ono što radite, za razliku od pukog opisivanja onoga što radite. Recenzenti i vaše "buduće ja" mogu pročitati patch, ali možda neće razumjeti zašto je određeno rješenje implementirano.*
- *Molimo da koristite glagole u imperativu u commit porukama, kao što su: Popravi grešku koja..., Dodaj datoteku/značajku...*"

Git Kraken pojašnjava da se koristi imperativ u liniji zaglavlja zbog perspektive prema kojoj se promjena naređuje sustavu, što će detaljnije biti opisano u poglavlju o preporukama Git Krakena [19].

Možemo uočiti da je ove nove smjernice Torvalds već pratio i tada dok je u svojoj inicijalnoj commit poruci [16] objašnjavao svrhu pisanja ovih uputa.

2.3.2. Prema specifikaciji Conventional Commits

Conventional Commits specifikaciju (hrv. konvencionalni commitovi, standardizirani commitovi, CCS) je razvila programerska zajednica kroz projekt otvorenog kôda čija je inicijalna verzija nastala 2017. godine. Detaljno definira pravila za pisanje Git poruka. Točno je određeno kako podijeliti zaglavlje i tijelo te podnožje, s time da su tijelo i podnožje commit poruka opcionalni [20]:

<tip>[opcionalni doseg]: <opis>

[opcionalno tijelo]

[opcionalno/a podnožje/a]

Pritom važe sljedeća pravila:

- <tip> je imenica, npr. *feat* (ako se u commitu dodaje nova funkcionalnost), *fix* (ako se u commitu popravlja greška)
- nakon tipa može slijediti [opcionalni doseg], što je imenica koja se odnosi na konkretnu funkcionalnost na koju utječe commit, npr. *fix(downloader)*
- <opis> mora slijediti nakon dvotočke ":", ukratko opisuje promjene u commitu, npr. *fix(downloader): issue with download not resuming after reconnecting to internet*
- tijelo može sadržavati detaljniji opis od onoga u zaglavlju, ali mora postojati jedna prazna linija između tijela i zaglavlja
- podnožje može biti dodano, ali mora postojati jedna prazna linija između podnožja i tijela

CCS definira i pravila formatiranja podnožja te druge aspekte i ograničenja u detalje. Jedan od razloga zašto ima toliko specifično definirana pravila je kako bi bilo moguće generirati zapisnik o izmjenama (eng. changelog) korištenjem alata koji će parsirati sve commit poruke na automatizirani način, što olakšava programerima posao pisanja dodatne dokumentacije prilikom objave nove verzije programa.

2.3.3. Prema Git Krakenu

Kao alat stvoren da ponudi grafičko korisničko sučelje za Git, Git Kraken također ima službeni članak o tome što smatra najboljim praksama za pisanje Git commit poruka [19].

Git Kraken ističe da dobra commit poruka počinje od dobrog commita, a za to što čini dobru commit poruku stavlja naglasak na sljedeće činjenice:

- Zaglavlje - idealno maksimalno 50 slova, ali može sadržavati do 72 slova
- Tijelo - idealno maksimalno 72 slova, ali nema gornju granicu za količinu slova
- Idealna commit poruka je dovoljno kratka da se lako pročita, ali dovoljno detaljna da se commit lako razumije
- Konzistentnost u commit porukama je ključna, radilo se o individualnom ili o timskom radu
- Pojedini timovi koriste specifične znakove (npr. *) kako bi istaknuli da tijelo commit poruke sadrži više informacija
- Ne završavati Git commit poruke s interpunkcijskim znakovima
- Izbjegavati nepotrebna velika slova, provjeriti pravopis
- Upotrijebiti imperativ u zaglavlju
- Referencirati zadatak koji commit rješava

- Često commitati promjene, to daje mogućnost što veće preciznosti i jasnoće pri pisanju commit poruka
- Conventional commits predlaže dobar format početka zaglavlja (npr. Fix-, Feat-, Perf-)

Imperativ (npr. "Add X" umjesto "(I) added X", hrv. "Dodaj X" umjesto "Dodao/la sam X") preporučuje se jer, osim što jasno sažima promjenu, stavlja fokus na perspektivu u kojoj se promjena izravno nalaže sustavu, kao da se daje naredba prilikom povlačenja promjena sa centralnog repozitorija na lokalnu kopiju, umjesto da commit poruka služi samo kao zapis o onome što je autor napravio. Drugim riječima, commit poruka je dobro strukturirana ako pravilno nadopunjuje rečenicu: "If applied, my commit will [commit poruka]". Zato se i preporučuje da se Git commit poruke pišu na engleskom, naročito u slučaju međunarodnih projekata. Iz tog razloga odlučili smo naš alat prvenstveno implementirati na engleskom jeziku. Ipak, LLM ima mogućnost vraćanja prijedloga za zaglavlje i tijelo commit poruke na istom jeziku na kojem je napisana commit poruka, u skladu s pravilom konzistentnosti jezika u repozitoriju. Ovo pravilo konzistentnosti bit će pojašnjeno u sljedećem potpoglavlju, odnosno u metodologiji odabira smjernica za pisanje commit poruka, koje ujedno čine evaluacijski standard na temelju kojeg alat provodi analizu.

2.3.4. Metodologija procjene kvalitete commit poruka u predloženom alatu

Na temelju dosadašnjeg pregleda literature o dobrim praksama u pisanju commit poruka, oblikovali smo sljedeću metodologiju na temelju koje će se provoditi evaluacija kvalitete commit poruka u našem predloženom softverskom rješenju.

Pojednostavljanjem stavki koje smatramo ključnim značajkama analiziranih izvora, možemo napraviti sljedeću tablicu usporedbe. Iz tablice 1 jasno vidimo koje su preporuke podržane (✓), nisu izričito navedene u navedenoj literaturi ili su opcionalne (–), ili su pak u kontradikciji s uputama navedene literature (×).

Tablica 1: Usporedba preporuka za commit poruke između različitih izvora

Značajka	Torvalds	CCS	Git Kraken	Bodovi
Zaglavlje u jednoj liniji	✓	✓	✓	$3/3$
Imperativ u zaglavlju	✓	✓	✓	$3/3$
Tijelo poruke objašnjenje	✓	✓	✓	$3/3$
Ograničenje duljine linija	✓	✓	✓	$3/3$
Podjela na zaglavlje, tijelo, podnožje	✓	✓	–	$2.5/3$
Konzistentna struktura	–	✓	✓	$2.5/3$
Česti commitovi	–	✓	✓	$2.5/3$
Referenciranje zadatka	–	✓	✓	$2.5/3$
Opcionalno tijelo i podnožje	×	✓	✓	$2/3$
Format `<tip promjene> opis`	×	✓	✓	$2/3$
Upotreba "Signed-off-by"	✓	–	×	$1.5/3$
Podržava obradu u changelog	×	✓	×	$1/3$

Stavke koje su podržane unutar svih triju izvora bit će definirane kao obavezne, ostale će biti rangirane prema bodovima ili izostavljene. Odnosno, sve stavke s ocjenom 2 ili više bit će uključene u alat. Primarni razlog za ovakvu selekciju je ušteda tokena pri svakoj interakciji s LLM-om, s obzirom na njihovu ograničenu količinu (uloga tokena je detaljnije pojašnjena u sljedećem poglavlju). U sljedećoj planiranoj verziji alata će se s većim brojem dostupnih tokena uključiti i sve preporuke iz tablice koje imaju 2 ili više boda, a koje već nisu uključene u trenutnoj verziji.

Bodovi su dodijeljeni prema vrijednostima:

✓ $= +1$

$- = +0.5$

× $= +0$

2.3.4.1. Odabrane smjernice za evaluaciju commit poruka

Iz tablice 1, možemo definirati niz općeprihvaćenih standarda za pisanje dobrih commit poruka.

1. Zaglavlje commit poruke mora biti 72 znakova ili manje, u jednoj liniji
2. Zaglavlje mora početi glagolom u imperativu koji opisuje tip commit poruke, imperativ se koristi kroz cijelo zaglavlje

3. Tijelo commit poruke je opcionalno, ali ako je korišteno onda služi za detaljnije objašnjenje promjena u tom commitu bez suvišnih riječi
4. Commit poruku pisati na engleskom jeziku ili na jeziku i stilu konzistentnom s ostalim commitovima u repozitoriju
5. Pisati commit poruke na gramatički ispravan način
6. U commit poruci uključiti referencu na zadatak u čijem rješenje taj commit sudjeluje ili na pull request na koji se odnosi, ako postoji

Velikom jezičnom modelu će na temelju ovih odabranih smjernica biti formalno definirana pravila koja će koristiti pri analizi commit poruka te pri davanju ocjena i prijedloga za njihovo unaprjeđenje. Ta pravila, koja čine temelj evaluacijskog postupka, formulirana su na sljedeći način:

1. Duljina zaglavlja 72 ili manje - ako prelazi 72, smatrati prekršajem
2. Imperativ u zaglavlju - ako nije imperativ, smatrati prekršajem
3. Sažetost tijela commit poruke - ako je nepotrebno dugačko (ili nejasno), smatrati prekršajem
4. Gramatika - ako je značajno loša gramatika u zaglavlju ili u tijelu (očite greške), do mjere da otežava razumijevanje commit poruke, smatrati prekršajem
5. Konzistentnost stila repozitorija - provjera konzistentnosti jezika s obzirom na druge commitove u repozitoriju te konzistentnosti stila, ako nije konzistentno smatrati prekršajem
6. Referenca na zadatak - opcionalni dodatni kriterij ocjenjivanja; ako za repozitorij postoje zadaci (alat sam prepoznaje iz konteksta referenciranog repozitorija), a nije referenciran zadatak smatrati prekršajem

Ocjena koju alat dodjeljuje commit porukama kreće se od 0 do 5, gdje 5 označava "izvrsno", a 0 "izuzetno loše". Svaka poruka u početku ima maksimalnu ocjenu 5, koja se smanjuje pri svakom prekršaju pravila, sve do minimalne vrijednosti 0. Commit poruke s ocjenom manjom od 3 smatraju se nezadovoljavajućima.

LLM će opravdati ocjene na način da će vratiti popis pravila koje smatra prekršenima te dati prijedlog nove Git commit poruke za zaglavlje i tijelo koja nema te nedostatke. Nadalje, kako bismo za LLM definirali dobru uputu (eng. prompt) za ocjenjivanje commit poruka, prvo trebamo produbiti razumijevanje o velikim jezičnim modelima.

3. Veliki jezični modeli

Alat koji razvijamo temelji se na primjeni velikih jezičnih modela (LLM). Kako bismo LLM-u definirali dobru uputu, odnosno prompt, kako ispravno ocijeniti opis verzije programskog kôda, u ovom poglavlju ćemo pojasniti pojmove velikih jezičnih modela, načina na koji funkcioniraju, a poseban naglasak će biti na pisanju uputa odnosno promptova temeljem kojih alat vraća odgovor.

Isto tako, bit će ukratko predstavljena umjetna inteligencija s posebnim naglaskom na obradu prirodnog jezika, arhitekturu transformatora te način funkcioniranja velikih jezičnih modela. Također ćemo se osvrnuti na dosadašnju edukativnu primjenu ovih modela i smjestiti naše rješenje u taj kontekst. Zatim će se pomoću alata za pretraživanje i analizu znanstvene literature Elicit provesti strukturirana komparativna analiza četiri velika jezična modela - OpenAI GPT, Google Gemini, Anthropic Claude i Meta LLaMa - prema kriterijima mogućnosti integracije s Gitom, te sposobnosti razumijevanja programskog kôda i analize commit poruka. Rezultati Elicitove analize bit će uključeni u raspravu, uz razmatranje aspekata koji nisu obuhvaćeni analizom, poput usporedbe cjenovnih planova API-ja i njihove brzine te jednostavnosti u primjeni unutar softverskog razvoja. Cilj analize je odabrati najprikladniji model za integraciju u naše rješenje s obzirom na kontekst problema i primjenu u obrazovnom kontekstu.

3.1. Umjetna inteligencija i jezični modeli

Umjetna inteligencija (UI) predstavlja područje računalne znanosti za izgradnju sustava koji mogu izvršiti zadatke koji zahtijevaju ljudsku inteligenciju - primjerice, planiranje, prepoznavanje uzoraka, razumijevanje jezika i učenje. UI se temelji na kombinaciji algoritama, statistike i podatkovne znanosti, a neke najvažnije primjene su u područjima računalnog vida te obrade prirodnog jezika (eng. Natural Language Processing, NLP) [21].

U kontekstu LLM-ova, najvažnija poddisciplina UI je strojno učenje, posebno duboko učenje (eng. deep learning) koje koristi višeslojne neuronske mreže za modeliranje složenih jezičnih i logičkih odnosa. Takvi sustavi više ne funkcioniraju po pravilima koja eksplicitno programira čovjek, već uče na temelju velikih skupova podataka koji su često na razini milijardi rečenica ili linija kôda [22]. U tom smislu, veliki jezični modeli oslanjaju se na sadržaj proizveden iz gotovo cjelokupne pisane ljudske povijesti i funkcioniraju kao sustavi za učinkovito generiranje odgovora na pitanja iz različitih domena.

Još jedan važan pojam kod LLM-ova predstavlja prompt, odnosno ulazni tekst koji se šalje modelu sa ciljem dobivanja odgovora. Način na koji je prompt formuliran izravno utječe na relevantnost i točnost dobivenih rezultata. Radi toga se razvio i pojam "inženjerstvo promptova" (eng. prompt engineering) koji označava praksu osmišljavanja i prilagođavanja prompta na način da model daje korisnije odgovore te da se izbjegnu halucinacije [23]. Halucinacije su situacije kada jezični model generira netočne, izmišljene ili nelogične informacije koje nisu utemeljene na ulaznim podacima. U našem rješenju one bi se mogle pojaviti, primjerice, ako model izmisli pravilo koje nije zadano ili predloži commit poruku koja odstupa od definirane

strukture.

U ovom projektu prompt i prompt engineering imaju bitnu ulogu jer definiraju kako će Gemini model analizirati commit poruke. Definiranje pravila po kojima želimo da model vrši evaluaciju commit poruka te sistemske upute modelu osiguravaju da se model drži isključivo zadanih pravila te smanjuju rizik od halucinacija.

3.2. Arhitektura i funkcioniranje velikih jezičnih modela

Temelj velikih jezičnih modela je transformator (eng. transformer), arhitektura koju su prvi opisali Vaswani et al. [24] u radu “Attention is All You Need”. Transformatori omogućuju paralelnu obradu ulaznih podataka, za razliku od ranijih modela kao što su rekurentna neuronska mreža (eng. Recurrent Neural Network, RNN) i Long Short-Term Memory (LSTM) koji su koristili sekvencijalnu obradu. Centralni mehanizam unutar transformatora je “self-attention”, koji omogućuje modelu da za svaki element ulaza procjeni važnost svih ostalih elemenata u toj sekvenci. Na primjer, prilikom analize rečenice "Refaktoriram funkciju jer je nečitka", model može ocijeniti da je riječ "jer" ključ za postavljanje odnosa između radnje i razloga. Radi toga su transformatori prikladni za zadatke koji zahtijevaju razumijevanje konteksta, poput analize commit poruka.

LLM-ovi se treniraju na velikim količinama tekstualnih podataka od javnih web stranica do open-source kôda kako bi naučili statističke obrasce jezika. Tijekom treniranja model uči predviđati sljedeći token (što može biti riječ, dio riječi ili znak) u nizu koji bi inače slijedio s najvećom vjerojatnošću, što ga čini sposobnim za generiranje koherentnog teksta. U ovom smislu, modeli nemaju sposobnost samostalnog rezoniranja kao što bi se očekivalo od istinske "umjetne inteligencije", već sastavljaju svaki odgovor po statističkim obrascima na kojima su trenirani iz pruženih podataka. Osim predviđanja sljedećeg tokena u nizu, modeli se često dodatno fino podešavaju (eng. fine-tuning) ili putem Reinforcement Learning from Human Feedback (RLHF), kako bi njihovi odgovori bili relevantniji u stvarnim kontekstima [25].

Za analizu commit poruka, važno je da je odabrani model treniran tako da razumije strukturu i informativnost tih poruka, kao i kontekst vezan uz programski kôd. Neki modeli su dodatno trenirani na velikim količinama programskog sadržaja, bilo da se radi o isječcima kôda, tehničkoj dokumentaciji, komentarima u kôdu, Git commit porukama ili pitanjima i odgovorima s platformi poput Stack Overflowa, što ih čini prikladnijima za ovaj zadatak.

Veliki jezični modeli su često dostupni putem API-ja (eng. Application Programming Interface), programskog sučelja preko kojeg vanjske aplikacije mogu slati upite i primati odgovore modela. Korištenje API-ja omogućuje integraciju LLM-ova u postojeće sustave bez potrebe za lokalnim pokretanjem modela, što je važno jer treniranje i izvođenje ovakvih modela zahtijeva značajne hardverske resurse. U kontekstu našeg rješenja, pristup preko API-ja omogućuje analizu commit poruka uz minimalnu infrastrukturnu složenost, stoga se radi toga u usporedbi ispituje i dostupnost te troškovna pristupačnost pojedinih API rješenja. Ograničavajući faktor pri svakom API pozivu je broj raspoloživih tokena koje model koristi za obradu ulaza i generiranje izlaza. Budući da svaki upit i odgovor troše određeni broj tokena, ukupna duljina upute i

generiranog odgovora ne smije prijeći određeni limit. To znači da korištenje predugačkih uputa dovodi do prekoračenja ograničenja, zbog čega je potrebno pažljivo dizajnirati promptove kako bi se u raspoloživi broj tokena saželo sve bitne informacije.

Uz API, modeli mogu biti dostupni i putem SDK-a (eng. Software Development Kit), programskih alata za rad u određenom programskom jeziku, kao i preko CLI (eng. Command Line Interface) okruženja. Takvi alati olakšavaju integraciju u razvojne procese i nude fleksibilnije testiranje i automatizaciju. U usporedbi se uzima u obzir i dostupnost ovih alata jer izbor između API-ja, SDK-a ili CLI-ja utječe na složenost razvoja i troškove održavanja sustava.

3.3. Edukativna primjena velikih jezičnih modela

Veliki jezični modeli donose bitne inovacije u edukaciji, osobito u učenju programiranja, pisanju kôda i razvoju softverskih vještina. Njihova sposobnost da razumiju tekstualni kontekst, analiziraju kôd, daju savjete i generiraju dokumentaciju omogućava stvaranje inteligentnih edukacijskih alata koji studentima mogu pružiti personalizirane povratne informacije. LLM-ovi omogućuju automatsko ispravljanje pogrešaka u kôdu, generiranje objašnjenja za programske koncepte i evaluaciju kvalitete tekstualnih opisa unutar kôda, kao što su commit poruke. Time se stvara okruženje u kojem studenti ne moraju čekati povratnu informaciju nastavnika, već odmah dobivaju smjernice za poboljšanje vlastitog rada.

Prema istraživanju Choi et al. [26], uvođenje LLM-a u edukaciju poboljšava angažman studenata i ubrzava proces usvajanja znanja. Studenti koriste modele poput GPT-a za traženje pojašnjenja, analizu greški i refaktoriranje vlastitog kôda. Pedagogijske prednosti uključuju poboljšano razumijevanje studenata [27].

U kontekstu ovog rada, LLM se koristi za analizu commit poruka kao kratkih, ali važnih tekstualnih jedinica koje dokumentiraju promjene u kôdu. Budući da studenti često pišu neinformativne ili nekonzistentne commit poruke, naš alat koristi LLM za njihovu evaluaciju i davanje prijedloga za poboljšanje. Ova metoda direktno pomaže studentima da usvoje dobre inženjerske prakse u radu s verzijskom kontrolom (Git), što je bitan aspekt profesionalnog softverskog razvoja.

Primjenom LLM-a u ovom kontekstu ostvarujemo sljedeće edukativne ciljeve:

- povratna informacija studentima u stvarnom vremenu
- razvoj profesionalne dokumentacijske prakse
- smanjenje opterećenja nastavnika u ocjenjivanju softverskih artefakata
- poticanje refleksije kod studenata o svrsi i kvaliteti commit poruka.

Odnosno, cilj je razviti alat koji služi kao obrazovni asistent, ne samo kao tehnički validator.

3.4. Usporedba najvažnijih modela za analizu commit poruka

Za razvoj alata koji ocjenjuje commit poruke i daje prijedloge za poboljšanje, relevantni su modeli koji kombiniraju razumijevanje jezika i dobru integraciju s programskim okruženjem. U nastavku se uspoređuju četiri suvremena LLM-a: OpenAI GPT, Google Gemini, Anthropic Claude i Meta LLaMA, svaki s obzirom na mogućnost integracije, razumijevanje kôda i potencijal za edukaciju programera. Pomoću Elicita smo proveli pretraživanje studija s obzirom na ove kriterije.

3.4.1. Metodologija pretrage

Koristili smo sljedeće istraživačko pitanje za Elicit: “Usporedi velike jezične modele GPT-4 (OpenAI), Claude 3 (Anthropic), Gemini (Google) i LLaMa (Meta) prema sljedećim kriterijima:

- Otvorenost i dostupnost API-ja (postoji li javni API, je li dokumentiran, koliko je lako integrirati model)
- Brzina i jednostavna integracija u softverske alate (npr. putem Node.js, SDK-ova, CLI alata)
- Podrška za razumijevanje programskog kôda i Git commit poruka (kako su modeli trenirani na kôdu, kako analiziraju sažetke promjena)
- Praktična upotrebljivost u razvoju alata za ocjenu kvalitete commit poruka (npr. u edukacijske svrhe)"

3.4.1.1. Selekcija izvora

Proveli smo selekciju izvora koji su ispunjavali sljedeće kriterije:

- **Pokriće LLM-a**: Ispituje li studija barem jedan od navedenih LLM-ova (GPT-4, Claude 3, Gemini, LLaMa)?
- **Tehnička integracija**: Proučava li studija tehničke aspekte integracije (API integracija, implementacija SDK-a ili programski pristup) s razvojnim alatima ili okvirima (eng. framework)?
- **Analiza kôda**: Proučava li studija izvedbu modela u razumijevanju kôda ili analizi commit poruka uz empirijsku evaluaciju?
- **Mjerni podaci izvedbe**: Uključuje li studija empirijsku evaluaciju tehničkih mjernih podataka (vremena odgovora, propusnost ili latencija)?
- **Tehnička fokusiranost**: Proučava li studija tehničke aspekte integracije izvan općeg razumijevanja jezika ili sposobnosti prirodnog jezika?

- **Verzija modela**: Proučava li studija trenutne verzije modela (ne ranije verzije)?
- **Vrsta studije**: Uključuje li studija empirijsku evaluaciju (ne samo mišljenja ili teorijske rasprave)?
- **Fokus implementacije**: Fokusira li se studija na praktične aspekte implementacije, a ne samo na obuku modela, arhitekturu ili analizu troškova?

Svi kriteriji su uzeti u obzir zajedno i donesena je holistička procjena o tome hoće li rad biti uključen u daljnju analizu.

3.4.1.2. Ekstrakcija podataka

Za dizajn studije i metodologiju, zadatak je bio opisati osnovnu metodologiju korištenu u studiji za usporedbu velikih jezičnih modela. Konkretno, trebalo je identificirati vrstu studije, poput komparativne analize, evaluacije izvedbe ili empirijske procjene. Zatim je bilo potrebno opisati specifičan pristup koji je korišten za usporedbu jezičnih modela, primjerice zadatke generiranja kôda, inženjering upita ili evaluaciju izvedbe. Također su se trebala napomenuti bilo koja jedinstvena obilježja eksperimentalnog dizajna. Ove informacije su se prvenstveno tražile u odjeljcima o metodama ili uvodu, a u slučajevima kada je korišteno više metodoloških pristupa, oni su navedeni prema važnosti.

3.4.1.3. Primjeri opisa metodologije studija

Tijekom ekstrakcije podataka Elicit je, između ostalog, identificirao obrasce istraživačkog dizajna, odnosno tipične načine na koje autori provode evaluaciju velikih jezičnih modela. Ovi obrasci služe kao primjeri standardiziranih opisa metodologija koje se koriste u literaturi.

Njihova svrha je sažeto prikazati tip studije (npr. komparativna analiza, empirijsko istraživanje) i konkretnu metodu usporedbe modela (natjecanja u kodiranju, generiranje kôda, prevođenje jezika). Time se omogućuje brza usporedba različitih radova prema istraživačkom dizajnu.

U ovoj fazi je Elicit zadao uputu da model izdvoji i prikaže način istraživačkog dizajna i metodologije korišten u studiji. Primjeri formata kojima je opisana metodologija studije mogu biti:

- "Komparativna analiza izvedbe LLM-a korištenjem LeetCode i GeeksforGeeks natjecanja u kodiranju"
- "Empirijska procjena LLM-a kroz zadatke generiranja kôda i prevođenja na više programskih jezika"

3.4.1.4. Specifična evaluacijska mjerila

Izdvojena su specifična mjerila koja su korištena za usporedbu velikih jezičnih modela, s naglaskom na ključna područja istraživačkog pitanja:

- Otvorenost i dostupnost API-ja
- Brzina integracije
- Sposobnost razumijevanja kôda
- Korisnost za razvoj softverskih alata

Ove informacije su pretražene u metodologiji, rezultatima ili raspravi znanstvenih publikacija, uključujući bilo kakve kvantitativne ili kvalitativne metrike. Ako kriteriji nisu izričito definirani, zabilježeno je "Kriteriji evaluacije nisu specificirani".

Za pretraživanje specifičnih evaluacijskih mjerila korištenih u studijama modelu su dani primjeri standardiziranog formata, poput:

- API pristupnost: [Specifična metoda evaluacije]
- Brzina integracije: [Metrike ili pristup evaluaciji]
- Razumijevanje kôda: [Specifični testovi ili benchmarkovi]

3.4.1.5. Performanse generiranja kôda i analize

U ovom dijelu istraživanja prikupljene su informacije o tome kako se veliki jezični modeli snalaze u zadacima koji uključuju generiranje i analizu programskog kôda. Naglasak je stavljen na metrike koje kvantitativno prikazuju snagu i ograničenja pojedinih modela u praktičnim zadacima, čime se omogućuje usporedba njihove primjenjivosti u realnim scenarijima razvoja softvera.

Zabilježene su specifične metrike izvedbe za zadatke povezane s kôdom:

- Točnost generiranja kôda
- Sposobnost razumijevanja i analize kôda/commit poruka
- Performanse u programerskim izazovima ili natjecanjima

Pretraženi su kvantitativni rezultati u sekciji rezultata. Uključeni su:

- Specifični postotci izvedbe
- Komparativni rangovi između modela
- Bilo koje značajne snage ili slabosti

Tijekom pretraživanja performansi, modelu su zadani primjeri poželjnog formata izvještavanja u studijama koje se pretražuju, primjerice:

- "Izvedba GPT-4: 85% uspješnosti u LeetCode natjecanjima u kodiranju"

- "Točnost prevođenja kôda: [Specifični postotak ili komparativna metrika]"

Ako mjerni podaci izvedbe nisu pruženi, zabilježeno je da "Mjerni podaci izvedbe nisu izvještavani".

3.4.1.6. Tehničke mogućnosti integracije

Analizirane su i tehničke mogućnosti integracije velikih jezičnih modela u razvojna okruženja i postojeće sustave. Ova dimenzija posebno je važna jer određuje koliko je praktično i izvedivo ugraditi pojedini model u alate koje koriste programeri, bilo u industriji, bilo u obrazovnom kontekstu.

Dokumentirane tehničke značajke integracije modela:

- Dostupnost SDK-ova ili razvojnih alata
- Jednostavnost integracije u razvojna okruženja softvera
- Podrška za različite programske jezike ili platforme

Pretraženi detalji u metodologiji, rezultatima i raspravi o:

- Podršci za programske jezike
- Integracijskim okvirima (Node.js, alati za komandno sučelje itd.)
- Bilo kojim specifičnim tehničkim ograničenjima ili prednostima

Za ove podatke korišteni su primjeri standardiziranog odnosno poželjnog formata u studijama koje se pretražuju, poput:

- Dostupnost SDK-a: [Da/Ne, s detaljima]
- Platforme za integraciju: [Popis podržanih okruženja]
- Značajna tehnička ograničenja: [Specifična ograničenja]

3.4.2. Rezultati

U nastavku su prikazani rezultati pretraženih i analiziranih studija. Svaka studija daje doprinos razumijevanju mogućnosti i ograničenja pojedinih modela, a tablica 2 sažima njihove glavne naglaske u odnosu na pokrivenost modela, tehničku integraciju i ključne nalaze.

Tablica 2: Rezultati studija za karakteristike različitih velikih jezičnih modela

Studija	Fokus studije	Pokrivenost modela	Tehnička integracija i glavni nalazi
Hou i Ji [28]	Komparativna analiza generiranja kôda i prevođenja u kodiranim natjecanjima	GPT-4, Gemini Ultra, Claude 2, ostali	Generiranje kôda na više jezika (Python3, C++, Java, JavaScript); nema izričite rasprave o API-u ili integraciji. Hou i Ji izvještavaju da je GPT-4 nadmašio ostale velike jezične modele i većinu ljudskih programera, s jakim sposobnostima prevođenja i ispravljanja pogrešaka.
Patil et al. [29]	Generiranje API poziva i smanjenje halucinacija korištenjem API-Bench (dataset za generaciju API poziva)	Gorilla (LLaMA-bazirani), GPT-4, GPT-3.5, Claude	APIBench dataset; integracija s HuggingFace, TorchHub, TensorFlow Hub; sustav za prepoznavanje dokumenata za adaptaciju. Patil navodi da je Gorilla nadmašio GPT-4 u točnosti API poziva i smanjenju halucinacija, te je bio otporan na promjene u dokumentaciji.
Shin et al. [30]	Generiranje kôda za Nonlinear Mixed Effects Modeling (NONMEM) za farmakometriju	ChatGPT 4.0, Gemini Ultra 1.0	Generacija kôda za NONMEM; nema detalja o SDK-u, API-ju ili integraciji. Oba modela generirala su predloške kôda, ali su zahtijevala stručnu korekciju; izlazi nisu bili ponovljivi ili bez pogrešaka.
Haider et al. [31]	Generiranje komentara za automatizirani pregled kôda	Llama (open-source), GPT-3.5, Gemini-1.0 Pro	Integracija putem OpenAI/Google API-ja; Node.js (JavaScript okruženje), Express (web okvir za Node.js), React.js (JavaScript knjižnica za korisnička sučelja), TypeScript (tipizirani superset JavaScripta); fino podešavanje putem Quantized Low-Rank Adapter (QLoRA). Haider izvještava da je inženjering upita i fino podešavanje poboljšalo generiranje komentara za pregled kôda; integracija je demonstrirana u web tehnologiji.
Elgedawy et al. [32]	Generiranje kôda s naglaskom na sigurnost	GPT-3.5, GPT-4, Bard, Gemini	Integracija temeljena na Pythonu (Django, Flask); nema detalja o SDK-u ili API-ju. GPT-3.5 i GPT-4 dosljedno su generirali funkcionalni kôd; Gemini i Bard bili su manje pouzdani, osobito pod sigurnosnim ograničenjima.

3.4.2.1. Pokrivenost modela

GPT-4 bio je evaluiran u tri studije; Gemini (bilo koja verzija) u tri studije; GPT-3.5 u dvije studije; Claude (bilo koja verzija) u dvije studije; modeli bazirani na LLaMA u dvije studije; ChatGPT 4.0, Gemini-1.0 Pro, Bard, Gorilla i Claude 2 evaluirani su svaki u zasebnoj studiji.

Nijedan model nije bio evaluiran u svih pet studija.

3.4.2.2. Otvorenost i dostupnost API-ja

Tablica 3: Procjena otvorenosti i dostupnosti API-ja ispitanih modela

Model	**Dostupnost API-ja**	**Kvaliteta dokumentacije za API**	**Metode integracije**
GPT-4	Javni API (OpenAI), nije detaljno opisano u studijama	Nema spomena u studijama	Implicirana putem OpenAI API-ja, nije sustavno evaluirana
Gemini	Javni API (Google), korišten u nekim studijama	Nema spomena u studijama	Pristup preko Google API-ja
LLaMA (Meta)	Otvoreni izvori težina (Llama, Gorilla), API nije jasan	Nema spomena u studijama	Integracija putem Gorilla sustava, APIBench, HuggingFace

Pronašli smo spominjanje javnog API-ja za Gemini, koji je korišten u nekim studijama, te za GPT-4, koji je bio dostupan, ali nije izravno korišten u uključenim studijama.

Za LLaMA, otvorene težine modela su bile dostupne (pobliže objašnjeno u raspravi u poglavlju 3.4.3), ali API nije bio dostupan.

Nijedna od uključenih studija nije izravno evaluirala Claude 3. Isto tako, nismo pronašli spominjanje kvalitete dokumentacije o integraciji preko API-ja za bilo koji od modela u uključenim studijama.

3.4.2.3. Brzina i jednostavnost integracije u softverske alate

Tablica 4: Procjena brzine i jednostavnosti integracije modela u softverske alate

Model	**Podrška za programske jezike**	**Složenost integracije**
GPT-4	Python3, C++, Java, JavaScript	Nije sustavno evaluirano
Gemini	Korišten u zadacima pregleda kôda i NONMEM	Integracija putem Google API-ja u web tehnologiji
LLaMA (Meta)	Integracija putem Gorilla s HuggingFace i sličnim alatima	Nije sustavno evaluirano

Podrška za programske jezike bila je eksplicitno opisana za GPT-4, koji podržava Python3, C++, Javu i JavaScript. Za Gemini smo ustanovili da je korišten u zadacima pregleda kôda i NONMEM-u, ali nismo pronašli detalje o podršci za programske jezike. LLaMA je integriran putem Gorilla s HuggingFace i sličnim alatima, ali također nismo pronašli specifične informacije o podršci za programske jezike.

Složenost integracije nije sustavno evaluirana za GPT-4 i LLaMA. Za Gemini je opisana integracija putem Google API-ja u web tehnologiji. Nije bilo spominjanja podrške za SDK-ove.

3.4.2.4. Praktična upotrebljivost u razvoju alata za ocjenu kvalitete commit poruka

Integracija u softverske alate bila je najviše konkretno prikazana kod Haidera [31], gdje su veliki jezični modeli korišteni u skupu web tehnologija kao što su Node.js i React.js za generiranje komentara za pregled kôda.

Integracija temeljena na Pythonu (Django, Flask) opisana je kod Elgedawyja [32], ali nedostajali su detalji o SDK-u ili API-ju.

Patil [29] demonstrira adaptaciju na promjene API-ja ili dokumentacije putem sustava za prepoznavanje dokumenata Gorilla, što implicira da LLaMA može biti posebno praktičan model za softverski razvoj.

Nijedna studija nije izravno evaluirala korisnost modela za edukacijske alate ili ocjenu kvalitete commit poruka, iako su nalazi o pregledu kôda i generiranju kôda relevantni za takve primjene.

3.4.3. Rasprava

Uključene studije pružaju najjače dokaze za generiranje i pregled kôda, s GPT-4 i LLaMA modelima (posebno Gorilla) koji demonstriraju jake performanse u zadacima vezanim uz kôd. Tehnička integracija najčešće je ostvarena putem javnih API-ja (OpenAI, Google) i uobičajenih web okvira kao što su Node.js i React.

Kvaliteta dokumentacije i lakoća uvođenja pojedinog LLM-a u softverski projekt nisu sustavno evaluirani u uključenim studijama. Uključene studije izvještavaju o dosljednoj visokoj podršci za razumijevanje programskog kôda za GPT-4 i LLaMA-bazirane modele, dok Gemini pokazuje promjenjive performanse, osobito pod sigurnosnim ograničenjima. Nijedna od uključenih studija nije izravno evaluirala Claude 3, niti se bilo koja fokusirala na analizu commit poruka. Integracija je često pretpostavljena putem API pristupa, ali podrška za SDK, CLI ili automatizaciju nije sustavno mjerena preko benchmark testova.

Osim ovih rezultata dobivenih komparativnom analizom iz postojećih studija, usporedili smo i planove cijena za svaki od njih, kao i dodatne značajke iz drugih studija koje nisu bile pokrivene pretraživanjem putem Elicita, a koje je vrijedno uzeti u obzir za potrebe razvoja predloženog softverskog rješenja.

3.4.4. OpenAI GPT

GPT-4 je trenutno jedan od najnaprednijih dostupnih modela [33]. Treniran je na mješavini internetskih podataka, kôdu i strukturiranom znanju, a ističe se po preciznoj obradi tehničkog jezika. Model pokazuje visoku točnost u razumijevanju prirodnog jezika i programskog kôda (posebno Python, JavaScript i Git konvencije), što ga čini prikladnim za analizu commit poruka.

Prema Brown et al. [34], GPT modeli u višim verzijama pokazuju emergentno ponašanje, odnosno sposobnost obavljanja zadataka za koje nisu bili izričito trenirani. To bi moglo biti korisno u edukacijskom kontekstu, gdje model može prilagoditi savjete razini znanja studenta.

OpenAI nudi pristup GPT-4-turbo i GPT-3.5-turbo putem API-ja, s cijenama od $0.01 za 1.000 ulaznih (prompt) tokena i $0.03 za 1.000 izlaznih (output) tokena kod GPT-4-turbo. GPT-3.5-turbo je jeftiniji, sa cijenama od $0.0005 za ulazne i $0.0015 za izlazne tokene. OpenAI ne nudi besplatni sloj za API, tako da se korištenje naplaćuje od prvog tokena.

3.4.5. Google Gemini

Googleov Gemini model razvijen je s naglaskom na učinkovitost i integraciju s Googleovim ekosustavom, što mu može dati prednost što se tiče mogućnosti integracija s Googleovim uslugama i servisima.

Gemini API od Googlea nudi pristup modelima poput Gemini 1.5 Pro preko Google AI Studija i Vertex AI-ja. Dok se korištenje preko Vertex AI-ja naplaćuje (npr. $0.0035 po 1.000 ulaznih tokena), Gemini API je trenutno besplatan za korištenje unutar Google AI Studija uz dnevna i minutna ograničenja. Ograničenje iznosi otprilike 5 zahtjeva po minuti i 25 zahtjeva dnevno za model Gemini 2.5 Pro. Nakon prelaska tih ograničenja, korištenje se blokira dok se dnevna kvota ponovno ne resetira ili osim ako se korisnik ne prebaci na neku plaćenu verziju. To čini Gemini jedinom od ove četiri opcije koja omogućuje besplatno API testiranje i razvoj do određenog broja zahtjeva.

3.4.6. Anthropic Claude

Claude je model razvijen uz naglasak na sigurnost, transparentnost i kontrolirano ponašanje [35]. Njegova arhitektura i treniranje također se temelje na transformatoru, ali s dodatnim naglaskom na smanjenje halucinacija i neprovjerenih tvrdnji. Claude modeli pokazali su se stabilnima u obradi strukturiranih podataka i u kreiranju sigurnog outputa, što je korisno kada radimo s edukacijskim alatima za studente.

Claude podržava obradu većih input sekvenci, što bi potencijalno omogućilo dubinsku analizu većih commit povijesti. Claude API koji nudi Anthropic dostupan je u više varijanti: Claude 3 Opus, Sonnet i Haiku. Cijene variraju, primjerice Claude 3 Sonnet stoji $3 po milijun ulaznih tokena i $15 po milijun izlaznih tokena. Ni jedan od Claude modela ne nudi besplatni sloj korištenja putem API-ja, tako da se svi zahtjevi naplaćuju.

3.4.7. Meta LLaMA 3

LLaMA modeli, razvijeni od strane Mete (Facebooka), temelje se na otvorenom kôdu i trenirani su na velikim količinama javno dostupnog teksta [36]. Meta Llama API je još uvijek u ograničenoj preview fazi, dostupan pojedinim korisnicima ili partnerima. Meta nije službeno objavila cjenik za API korištenje, no verzije poput LLaMA 2 su dostupne za komercijalnu upotrebu kao open-source modeli koje korisnici mogu hostati sami bez naplate za generiranje modela. Lokalno pokretanje bez potrebe za internetom ili vanjskim API ključem omogućuje razvoj alata koji su idealni za okruženja s ograničenim pristupom internetu ili kada je privatnost najvažniji faktor, što pak nije slučaj u kontekstu ovog rada.

Otvorene težine (eng. open weights) odnose se na parametre modela, poput težina neuronskih mreža, koji su dostupni javnosti nakon što je model treniran. To omogućuje korisnicima da preuzmu već istrenirane modele i primijene ih bez potrebe za ponovnim treniranjem. Razlika između otvorenog kôda i otvorenih težina je u tome što otvoreni kôd omogućuje pristup samoj implementaciji modela, dok otvorene težine omogućuju pristup parametrima modela. LLaMA modeli nude i otvoreni kôd i otvorene težine, što omogućava veću fleksibilnost u primjeni, ali traži vlastitu infrastrukturu za pokretanje. Za razliku od modela poput GPT-4, Claudea ili Geminija, koji nude zatvorene težine, LLaMA omogućuje preuzimanje parametara modela, što korisnicima daje veću slobodu, ali s većim tehničkim zahtjevima.

S druge strane, API-ji koje nude GPT-4, Gemini i Claude nude centralizirane i gotove API usluge koje se mogu koristiti putem interneta, uz dostupne dokumentacije, API ključeve i jednostavne integracije sa softverskim projektima, dok LLaMA zahtijeva od korisnika da postave vlastiti sustav ili poslužitelj za njegovo pokretanje. Iako LLaMA nudi veću slobodu prilagodbe, korisnici su odgovorni za upravljanje infrastrukturom i pokretanje modela, dok su ostali modeli lakši za implementaciju putem gotovih API rješenja.

3.4.8. Odabir modela za integraciju u predloženo programsko rješenje

Prednost API-ja u slučaju Geminija, GTP-4 i Claudea jest da omogućuju jednostavnu integraciju s NodeJS-om kao poslužiteljskom stranom i Git webhookovima, što bi omogućilo automatiziranje procjene kvalitete commit poruka svaki put kada korisnici naprave novi push na repozitorij.

Što se tiče LLaMa modela, s obzirom na nedostatak gotovog API rješenja te potrebu za dodatnom konfiguracijom i resursima za lokalno izvođenje LLaMA modela (poput GPU-a i optimizacije memorije), smatramo da će za naše rješenje bolje odgovarati jedan od API modela (GPT-4, Gemini ili Claude), koji omogućavaju jednostavniju i bržu implementaciju.

Isto tako, kako je Claude slabije zastupljen u pretraženim studijama, primarno se fokusiramo na odabir između OpenAI i Gemini modela. U kontekstu projekta biramo Gemini zbog dobre ravnoteže između kvalitete i cijene. Cijena je razumna odnosno besplatna na ograničene količine i nudi dovoljno veliku količinu besplatnih tokena prije nego što se javi potreba za

plaćanjem. Isto tako, podrška za dokumentaciju je na razini s onom za OpenAI i Claude, kao i općenita kvaliteta generiranja odgovora. Iako je pregled literature pokazao da OpenAI modeli mogu imati laganu prednost u nekim kontekstima generiranja i analize programskog kôda, smatramo da za potrebe razvoja predloženog softverskog rješenja Gemini modeli pružaju više nego zadovoljavajuću kvalitetu odgovora, pogotovo uz precizno strukturiran prompt.

Što se tiče cijene, upit u kojem se analizira posljednjih deset commit poruka prosječno troši nekoliko stotina tokena, što je zanemarivo u odnosu na besplatnu kvotu od više stotina tisuća tokena mjesečno. Time se potvrđuje da je Gemini, uz svoju kombinaciju kvalitete i povoljnog cjenovnog modela, najprikladniji izbor za potrebe ovog istraživačkog rada.

4. Dizajn i razvoj programskog rješenja

4.1. Funkcionalnosti

U ovom potpoglavlju je naveden popis funkcionalnih zahtjeva koje planirano programsko rješenje treba ispunjavati s obzirom na ranije diskutirane ciljeve (tablica 5).

S obzirom da je u popis funkcionalnih zahtjeva uvršten i zahtjev da LLM sam daje prijedlog bolje commit poruke, vrijedno je procijeniti postoji li rizik kako bi studenti jednostavno koristili alat da kopiraju predložene poboljšane commit poruke i koriste ih bez obraćanja pažnje na manjkavosti u vlastitim originalnim commit porukama te načinima na koje se one mogu poboljšati. Međutim, procjena je da u ovom slučaju ne postoji rizik od toga jer način na koji je alat strukturiran dozvoljava prijedloge poboljšanih poruka samo za one commitove koji su već napravljeni, te neće izbacivati prijedlog za commit poruku koju student trenutno piše. Kako je jedan od ciljeva predloženog programskog rješenja da ga studenti koriste za kritičko razmišljanje o vlasititm vještinama dokumentiranja svog programskog kôda i pisanja commit poruka, u ovom slučaju se LLM koristi samo za analizu prošlih commit poruka. Na temelju tih povratnih informacija, studenti kontinuirano mogu poboljšavati svoje buduće commit poruke i ponovno ih analizirati putem alata, sve dok ne podignu kvalitetu svojih poruka dovoljno da LLM ocijeni kako one slijede sve dobre prakse (maksimalna ocjena). Na taj se način ostvaruje edukativna svrha korištenja velikih jezičnih modela – oni služe kao podrška u razvoju vještina dokumentiranja i kritičkog razmišljanja, umjesto da budu samo prečac koji bi dugoročno onemogućio studentima da se oslanjaju na vlastite sposobnosti ako ih u međuvremenu ne razviju.

Tablica 5: Funkcionalni zahtjevi alata

Zahtjev	**Svrha**	**Metrika**
Integracija s uređivačem kôda VS Code	Omogućiti studentima jednostavno korištenje alata u njihovom radnom okruženju	Dostupan VS Code plugin/proširenje
Integracija s GitHubom	Povezati s odabranim javnim repozitorijem	Postoji opcija za unos poveznice za repozitorij u korisničkom sučelju
Dohvat commitova i njihovih detalja	Dohvatiti commit poruke s detaljima poput autora, grane na kojoj je objavljena, referenciranih taskova	Postoji opcija za prikaz commit poruka s navedenim detaljima u korisničkom sučelju
Filtriranje commitova po grani, autoru i/ili broju najnedavnijih commitova	Dohvatiti commit poruke filtrirane po autoru, grani i/ili broju najnedavnijih commitova	Postoji opcija za unos imena autora, grane te broja najnedavnijih commit poruka za dohvat u korisničkom sučelju

Zahtjev	Svrha	Metrika
Integracija s LLM-om	Iskoristiti potencijal velikih jezičnih modela za evaluaciju commit poruka	API poziv odabranom LLM-u vraća evaluaciju po predefiniranom formatu za jedan ili više odabranih commitova
Davanje povratne informacije o prekršajima na temelju definiranih dobrih praksi	Pomaganje studentima u uočavanju problema u commit porukama	Broj identificiranih prekršaja po poruci
Davanje ocjene commit poruke	Kvantificirati kvalitetu poruke	Ocjena za svaku commit poruku u korisničkom sučelju na skali od 0 (najniža - commit poruka ne slijedi niti jednu definiranu dobru praksu) do 5 (najviša - commit poruka slijedi sve definirane dobre prakse)
Davanje prijedloga za bolju commit poruku za zaglavlje i/ili tijelo poruke, u slučaju da jedno od toga ili oboje krši neko pravilo	Korištenje LLM-a za davanje primjera boljih commit poruka na temelju navedenih pravila	Prijedlog bolje poruke za zaglavlje i/ili tijelo commit poruke ispod commit poruke čije zaglavlje i/ili tijelo krše neko od pravila
Prikaz evaluacija u panelu na bočnoj traci unutar uređivača kôda	Evaluacije commit poruka prikazuju se u dediciranom panelu na bočnoj traci (eng. sidebar) uređivača kôda (VS Code), čime se izbjegava prekid trenutnog rada na kôdu u glavnom editoru i omogućuje paralelni prikaz rezultata evaluacije tijekom razvoja	Postoji bočni panel u kojem se prikazuju evaluacije commitova kao rezultat klika na opciju za analizu commitova
Prikaz evaluacija u HTML formatu unutar glavnog editora	Evaluacije commit poruka prikazuju se u HTML formatu unutar glavnog editora	Postoji prikaz u HTML formatu unutar glavnog editora koji oponaša izgled web stranice
Konfiguracija API ključeva	Korisnik može konfigurirati vlastite API ključeve za pristup GitHubu i Gemini modelu	Unutar konfiguracije VS Code proširenja postoji opcija za unos API ključeva za Github i Gemini model

4.2. Korištene tehnologije

Programsko rješenje razvijeno je kao plugin, odnosno proširenje za uređivač kôda, sa ciljem da korisnicima omogući paralelni rad na programskom kôdu i istovremenu analizu commit poruka. Na taj način kvaliteta dokumentiranja provjerava se tijekom samog procesa razvoja, bez potrebe za dodatnim alatima izvan radnog okruženja.

Odabran je Visual Studio Code (VS Code) zbog svoje široke popularnosti među studentima (što znači da će većina studenata imati najviše iskustva upravo s ovim alatom), manje kompleksnosti u odnosu na naprednija okruženja poput Microsoft Visual Studija, jednostavne instalacije i korištenja proširenja te opće primjenjivosti ovog uređivača kôda u edukativnom kontekstu. Proširenje se u potpunosti integrira u ovo okruženje, što daje intuitivno korisničko sučelje te olakšava konfiguraciju i korištenje alata za analizu commit poruka unutar već poznatog i raširenog korištenog uređivača.

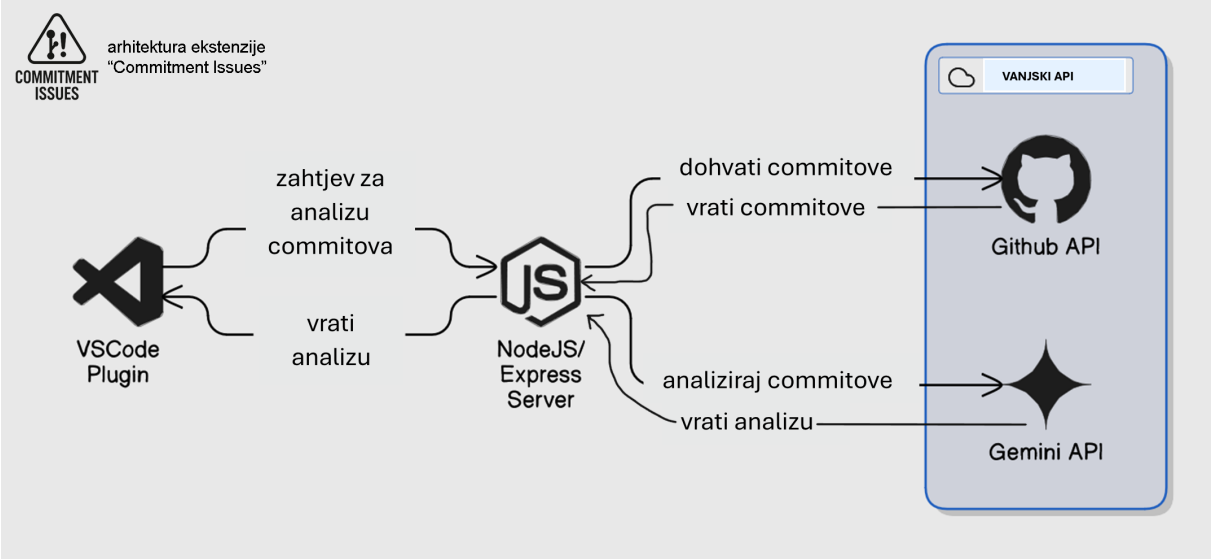
Za izradu poslužiteljskog dijela programskog rješenja se koristi Node.js s okvirom Express, pisan u programskom jeziku TypeScript. Odabran je TypeScript nad JavaScriptom, budući da TypeScript omogućuje statičko tipiziranje i otkrivanje grešaka u fazi razvoja, a time bolju održivost kôda. Ovo je posebno važno kod izrade edukacijskih alata koje će u budućnosti koristiti i potencijalno nadograđivati različiti korisnici.

Sve komponente sustava, njihova svrha, kao i tehnologije koje ih ostvaruju, opisane su u tablici 6.

Tablica 6: Korištene tehnologije

Tehnologija	**Svrha / opis uporabe**	**Dio sustava**
Node.js	Izvršno okruženje u kojem se pokreće backend logika alata	Poslužiteljski
Express	Okvir koji omogućuje definiranje HTTP endpointa i rukovanje zahtjevima prema GitHub i Gemini API-ju	Poslužiteljski
TypeScript	Osigurava veću razinu tipiziranja (eng. type-safety) u odnosu na JavaScript i olakšava održavanje kôda	Poslužiteljski
VS Code proširenje	Frontend komponenta integrirana u uređivaču kôda koja korisniku omogućuje unos parametara i prikaz rezultata evaluacije	Klijentski
GitHub API	Omogućuje dohvat commit poruka i pripadnih metapodataka iz odabranog repozitorija	Poslužiteljski (vanjski API)
Gemini LLM API	Analizira commit poruke i vraća ocjene i prekršaje prema definiranim pravilima.	Poslužiteljski (vanjski API)

Pokretanjem analize unutar instaliranog proširenja, ono se preko GitHub API-ja povezuje s odabranim **javnim** repozitorijem kako bi dohvaćao commitove kao što se vidi iz slike 7, a evaluacija commit poruka provodi se korištenjem Gemini LLM API-ja kojemu podatke o commitovima prosljeđuje poslužitelj.



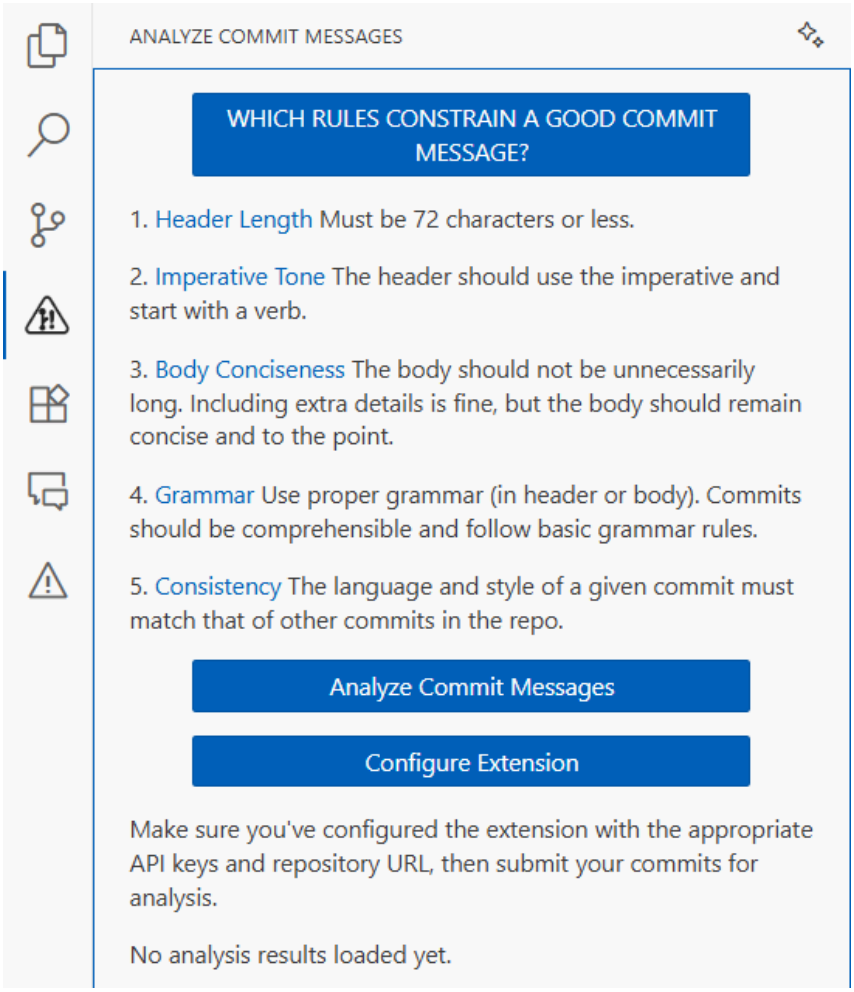
Slika 7: Arhitektura programskog rješenja

4.3. Tijek korištenja alata

Proširenja su dostupna na VS Code Marketplaceu, što znači da će nakon objave ovog programskog rješenja biti dovoljno unutar alata VS Code pretražiti naziv proširenja, "Commitment Issues", i kliknuti na opciju instalacije kako bi proširenje postalo odmah dostupno unutar sučelja uređivača.

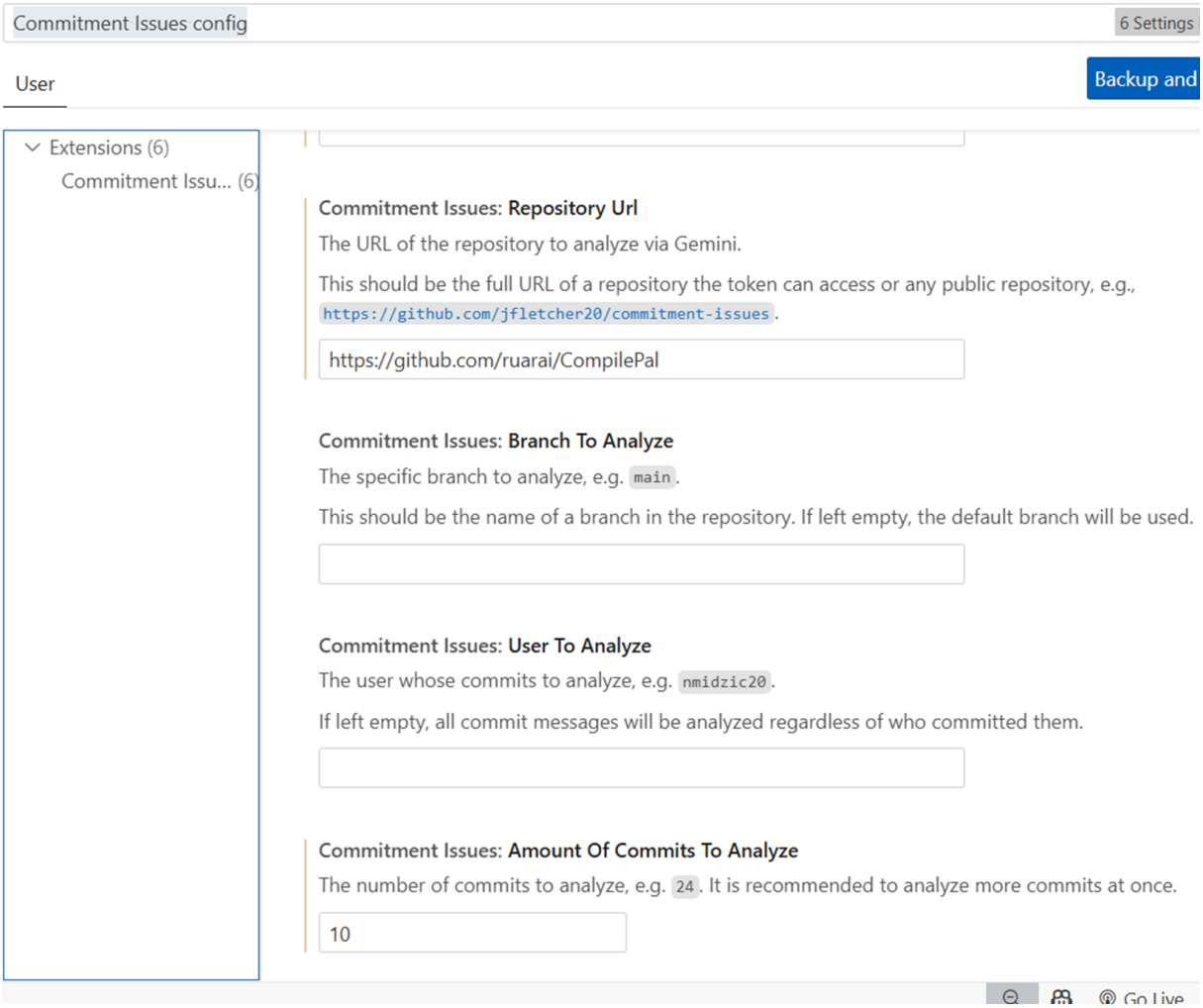
Jednom kada se otvori proširenje, ono prikazuje pravila po kojima LLM evaluira commit poruke, kako bi ih studenti mogli koristiti kao referencu i opcionalno pokušali popraviti svoje commit poruke i prije commitanja te analize.

Ispod popisa pravila se nalaze opcija za analizu commit poruka, kao i opcija za konfiguraciju proširenja, kao što se vidi na sljedećoj slici (slika 8).



Slika 8: Izgled VS Code proširenja

Kao što uputa u proširenju navodi, potrebno je prvo konfigurirati proširenje tako da se postave API ključevi za Github i Gemini model koje studenti sami nabavljaju, te se postavlja URL repozitorija s kojeg se dohvaćaju commit poruke. Sučelje za konfiguraciju dostupno je klikom na gumb "Configure Extension" ili preko komande ">Open Commitment Issues Config" u tražilici na vrhu uređivača. Izgled konfiguracijske stranice prikazan je na slici 9.



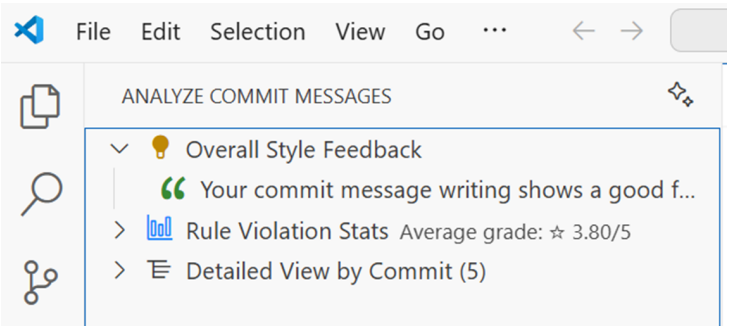
Slika 9: Konfiguracija proširenja

Slike 10 do 16 prikazuju proces analize commit poruka jednom kada se konfigurira proširenje i pokrene analiza klikom na opciju "Analyze Commit Messages". Slika 10 prikazuje obavijest u koja se pojavljuje u donjem desnom kutu uređivača kôda koja informira korisnika da je proces analize u tijeku, dok slike 15 do 16 prikazuju rezultate analize u bočnom prozoru uređivača kôda.

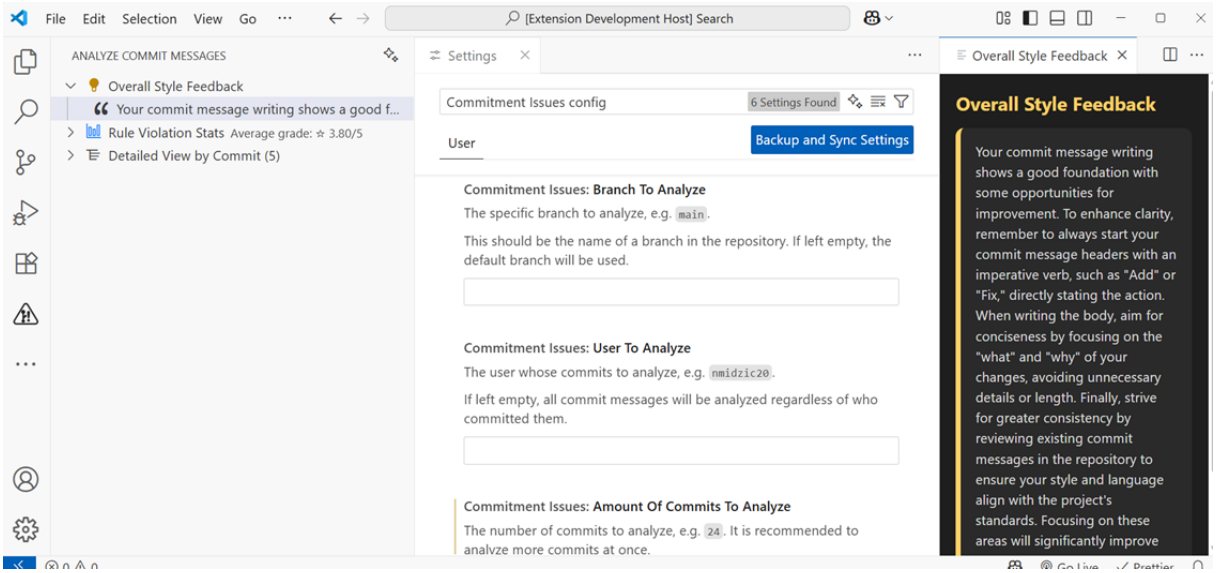
Slika 10: Pokretanje analize commitova

Prikaz rezultata u bočnom prozoru podijeljen je na jednostavan i detaljan pogled. U jednostavnom pogledu koji čini stavka "Overall Style Feedback", korisnik dobiva općeniti komentar od strane LLM-a o stilu pisanja commit poruka, zajedno sa savjetom za poboljšanje stila pisa-

nja bez referenciranja specifičnih commitova. Logika prema kojoj LLM daje ovakve općenite komentare pri svakoj analizi uzima u obzir sve commit poruke koje su analizirane u toj rundi, odnosno statistike o tome koliko poruka krši neko pravilo, koja pravila su prekršena te prosječnu ocjenu o kvaliteti svih commit poruka. Ta će logika biti detaljnije pojašnjenja u poglavlju 4.4. kod predstavljanja implementacije u kôdu. Kako je u jednostavnom pogledu poanta savjeta da vrati općeniti naputak u kojem smjeru korisnik treba krenuti da popravi stil pisanja svojih commit poruka, ove detaljne statistike se ne navode u samom savjetu, te su dostupne tek u detaljnom pogledu. Jednostavan pogled prikazan je na slikama 11 i 12.

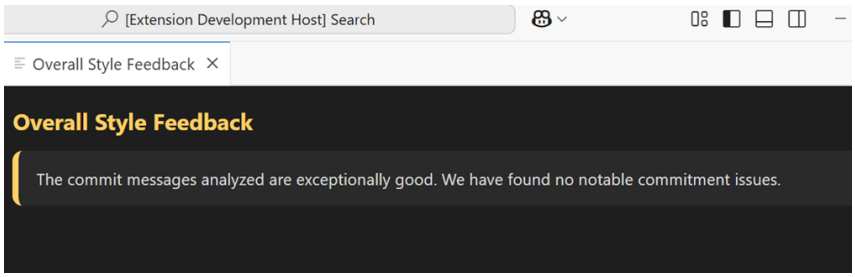


Slika 11: Jednostavan pogled na analizu



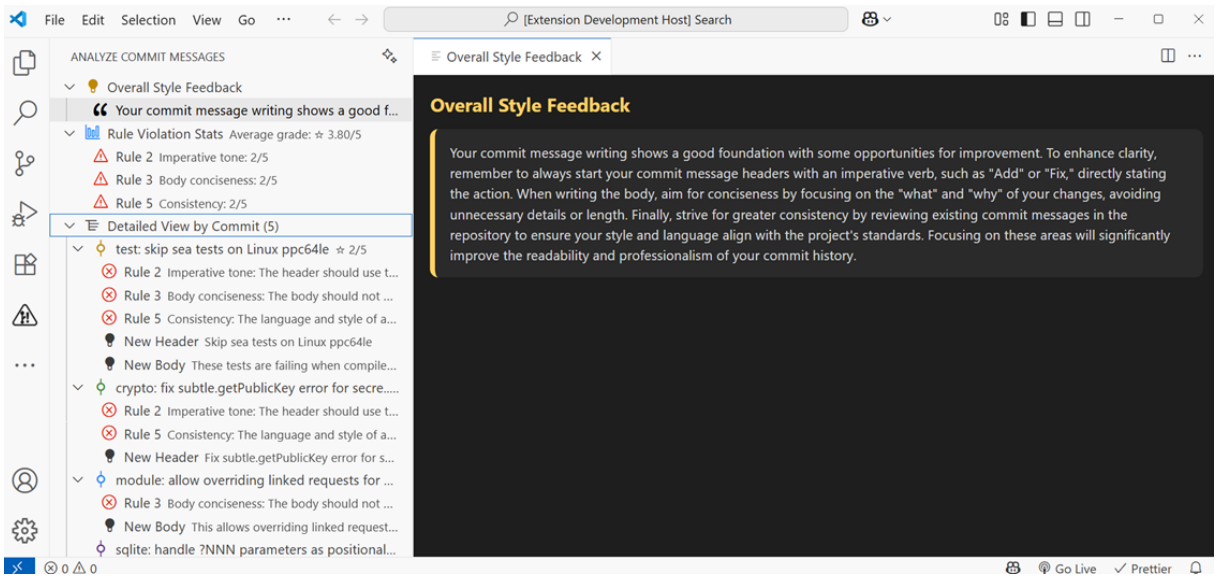
Slika 12: Jednostavan pogled nakon klika na savjet LLM-a

U slučajevima kada analiza pokaže da niti jedan commit nema ocjenu manju od 3 te je prosječna ocjena svih commitova najmanje 4.5, LLM neće davati savjet za popravke, već će se korisniku samo vratiti poruka da analizirani commitovi već slijede većinu ili sve dobre prakse - odnosno da ne treba posebno mijenjati stil pisanja commit poruka jer je već prilično dobar, kao što je prikazano na slici 13.



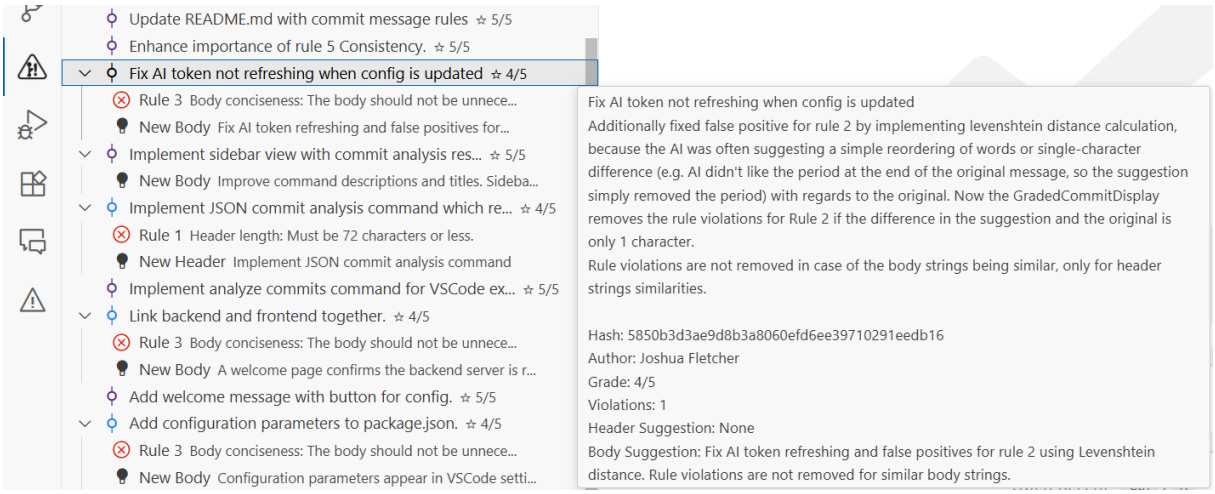
Slika 13: Slučaj kada commit poruke u prosjeku već slijede većinu ili sve dobre prakse

Detaljni pogled čine stavke "Rule Violation Stats" te "Detailed View by Commit (x)" gdje "x" u zagradi označava broj commitova koji se analizirao u toj rundi. Za razliku od jednostavnog pogleda, ove stavke detaljnog pogleda su inicijalno sakrivene i otvaraju se korisničkom akcijom. Korisnik klikom na "Rule Violation Stats" dobiva listu prekršaja uočenih u analiziranim commitovima (bez navođenja specifičnog commita). Klikom na "Detailed View by Commit", dobiva se pregled svih analiziranih commitova (slika 14).



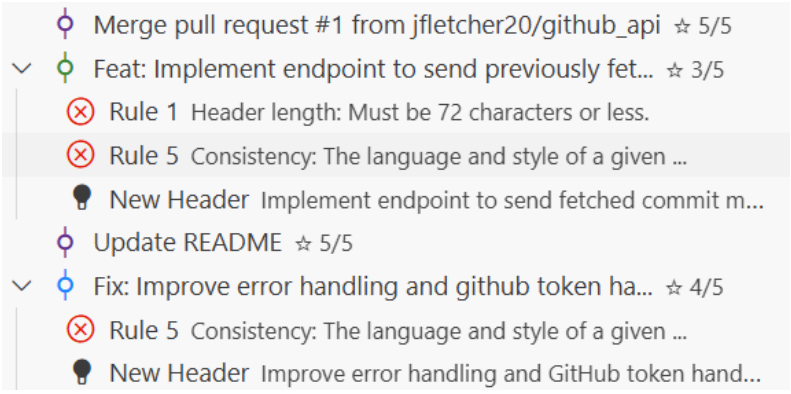
Slika 14: Detaljan pogled na analizu

Kako je prikazano na slici 15, prelaskom miša preko pojedine stavke svakog commita, korisnik dobiva detalje kao što su zaglavlje i tijelo originalne commit poruke, hash, ime autora, ocjena poruke te broj prekršenih pravila. Klikom na bilo koji commit se u pregledniku otvara GitHub stranica u repozitoriju za taj commit sa svim njegovim podacima.



Slika 15: Rezultati analize unutar bočnog prozora u proširenju

Nadalje, ako postoje prekršena pravila, u bočnom pogledu uređivača kôda se ispod commit poruke navodi koja su to pravila. U slučaju da su zaglavlje ili tijelo poruke ti koji krše neko od pravila, LLM će predložiti i primjer bolje commit poruke zaglavlja i/ili tijela. Ovo je prikazano na slici 16. Kao što je ranije navedeno, ova funkcionalnost je omogućena samo za već napravljene commitove (ne i trenutnu commit poruku koju student piše), kako bi se studente potaknulo na samostalno poboljšavanje svojih commit poruka sve dok ne dobiju maksimalnu ocjenu od strane LLM-a.



Slika 16: Detaljniji pogled na rezultate analize

4.4. Implementacija

U nastavku slijedi opis osnovnih dijelova implementacije proširenja u kôdu, odnosno njegovog klijentskog te poslužiteljskog dijela. Za detaljniji uvid u implementaciju, može se pristupiti javnom repozitoriju na poveznici: `https://github.com/jfletcher20/commitment-issues`.

Treba naglasiti da se u ovom poglavlju ne prikazuju sve tehničke pojedinosti implementacije, poput potpunog kôda, konfiguracijskih datoteka ili detalja pojedinih pomoćnih modula.

Takvi elementi, iako bitni za funkcioniranje sustava, premašili bi opseg rada te bi otežali razumijevanje glavne ideje i obrazovne svrhe alata. Stoga se fokus stavlja na pregled arhitekture i ključnih komponenti koje omogućuju funkcionalnost sustava, dok su detaljni kodni zapisi i tehničke specifikacije dostupni u priloženom javnom repozitoriju. Na ovaj način naglasak ostaje na strukturi i logici rješenja te njegovoj svrsi u obrazovnom kontekstu, a ne na operativnim detaljima implementacije.

Pregledom arhitekture prikazane na slici 7 te dijagrama toka na slici 19, olakšava se praćenje opisa implementacije koji slijedi. Na taj način, uz pregled glavnih komponenti, postaje intuitivnije sagledati i sve popratne funkcionalnosti programa – od načina komunikacije klijenta s poslužiteljem i upravljanja konfiguracijom, do prikaza rezultata analize, evidentiranja prekršaja pravila dobrih praksi te obrade commit poruka i njihovih analiza u oblik razumljiv korisniku.

4.4.1. Klijentski dio

Datoteka `extension.ts` služi kao ulazna točka VS Code proširenja i definira sve osnovne funkcionalnosti koje se pokreću prilikom aktivacije proširenja. Kao što se vidi iz isječka kôda 3, u funkciji `activate` pokreće se Express poslužitelj koji će obrađivati zahtjeve za dohvat i analizu commit poruka. Nakon pokretanja poslužitelja registriraju se komande koje će korisnik moći pozivati unutar VS Codea, te se inicializira provider koji omogućuje prikaz rezultata u bočnom panelu uređivača kôda.

Nadalje, registrirane komande omogućuju otvaranje konfiguracijske stranice proširenja, pokretanje analize commit poruka, te prikaz rezultata analize unutar uređivača kôda. Rezultati se dohvaćaju preko HTTP zahtjeva prema lokalno pokrenutom poslužitelju.

```
export function activate(context: vscode.ExtensionContext) {

        _backendServer = backend();
        ...

        const commitAnalysisProvider = new AnalysisResultsProvider(context);
        vscode.window.registerTreeDataProvider(VSCodeExtensionViews.ROOT_COMMIT_ANALYSIS_RESULTS,
        ↪  commitAnalysisProvider);

        context.subscriptions.push(
                vscode.commands.registerCommand("commitment-issues.openConfig", ()
                ↪  => {
                        vscode.commands.executeCommand("workbench.action.openSettings",
                        ↪  "Commitment Issues config");
                })
        );

        context.subscriptions.push(
                vscode.commands.registerCommand("commitment-issues.analyzeCommitMessages",
                ↪  async () => {
                        vscode.window.withProgress({
                                location: vscode.ProgressLocation.Notification,
                                title: `Analyzing ${ConfigurationManager.amount}
                                ↪  commits on ${ConfigurationManager.branch}...`,
                                cancellable: false
                        }, async (progress) => {
                                await commitAnalysisProvider.analyze();
                                vscode.window.showInformationMessage("Analysis
                                ↪  complete!");
                        });
                })
        );
        ...
}
```

Isječak koda 3: Funkcija `activate` iz datoteke `extension.ts`

Isječak kôda 4, izdvojen iz datoteke `settings.ts`, definira postavke koje korisnik može konfigurirati u sklopu VS Code proširenja. Enum `ExtensionConfigurationSettings` navodi sve dostupne parametre poput API ključeva za GitHub i Gemini, URL-a repozitorija, imena grane i autora, te broja najnedavnijih commitova koje je potrebno analizirati. Klasa `ConfigurationManager` služi kao centralno mjesto za dohvat i spremanje tih postavki preko VS Code API-ja za rad s konfiguracijom (`workspace.getConfiguration`). Kroz skup statičkih `get` metoda omogućuje se jednostavan pristup pojedinoj postavci unutar proširenja, pri čemu se po potrebi vraća zadana vrijednost (npr. predefinirana poruka ako API ključ još nije unesen). Na taj način ova komponenta upravlja svim korisnički definiranim parametrima koji su potrebni za povezivanje s GitHubom i Gemini modelom te za filtriranje commitova tijekom analize.

```
export enum ExtensionConfigurationSettings {
  root = "CommitmentIssues",
  gemini = "geminiApiKey",
  github = "githubAccessToken",
  repoUrl = "repositoryUrl",
  branch = "branchToAnalyze",
  user = "userToAnalyze",
  amountOfCommits = "amountOfCommitsToAnalyze",
}
/* primjer upotrebe:
 * vscode.workspace.getConfiguration(ExtensionConfigurationSettings.root)
 *   .get<string>(ExtensionConfigurationSettings.github) */
```

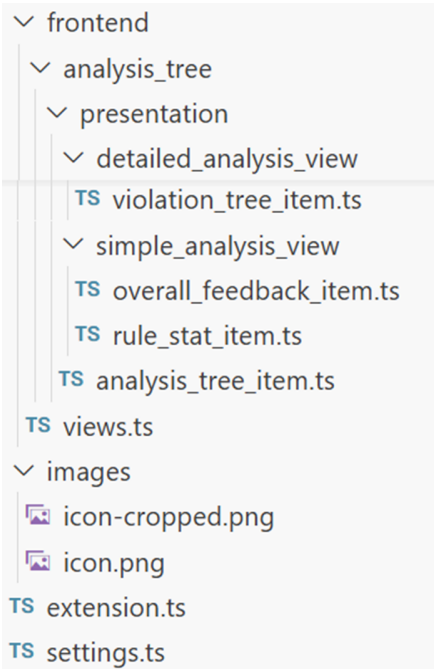
Isječak koda 4: Postavke konfiguracije proširenja iz datoteke `settings.ts`

Preostali dio klijentske strane sadrži nekoliko pomoćnih klasa koje služe za prikaz rezultata evaluacije commit poruka u obliku stabla u bočnom prozoru VS Codea.

Klasa `AnalysisResultsProvider` dohvaća rezultate s poslužiteljske strane, transformira ih u prikazive objekte i upravlja njihovim prikazom. Prezentacija stabla prikaza podijeljena je u ranije istaknuti jednostavan i detaljan pogled, gdje jednostavan pogled čine klase `OverallFeedbackRootItem`, `OverallFeedbackPreviewItem`, kao i `RuleStatItem` i `RuleStatsRootItem`, a detaljan pogled klase `CommitsRootItem` i `CommitTreeItem` za sam commit, te `SuggestionTreeItem` i `ViolationTreeItem`.

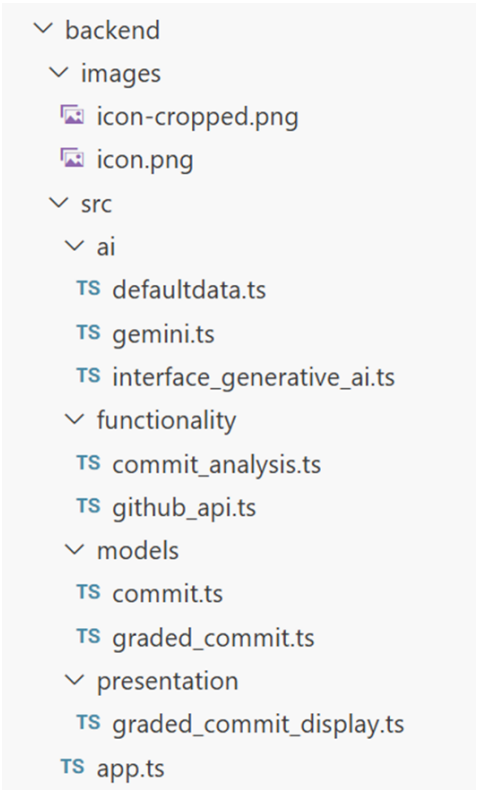
Klasa `CommitsTreeItem` predstavlja pojedini evaluirani commit te prikazuje informacije poput ocjene, autora i naslova poruke. `ViolationTreeItem` prikazuje prekršena pravila povezana s određenim commitom, dok `SuggestionTreeItem` prikazuje prijedloge poboljšanja. Sve navedene klase nasljeđuju baznu klasu `AnalysisTreeItem`, koja definira osnovne karakteristike svakog elementa u stablu (naziv, ikona, opis), te zajednički omogućuju pregledan prikaz rezultata analize u vizualnom sučelju proširenja.

Dodatno, `OverallFeedbackRootItem` služi kao naslovni čvor za prikaz općeg stila pisanja commit poruka, dok klasa `OverallFeedbackPreviewItem` prikazuje skraćenu verziju komentara s mogućnošću otvaranja punog teksta na zahtjev korisnika. Statistički pregled pravila omogućuju `RuleStatsRootItem`, koji u sažetku prikazuje prosječnu ocjenu i otvara popis pravila, te `RuleStatItem`, koji za svako pojedino pravilo prikazuje broj prekršaja i njegov opis. Na taj način korisnik u bočnom prozoru radnog okruženja uređivača kôda dobiva hijerarhijski organiziran prikaz – od opće ocjene i komentara stila, preko statistike pravila, do detaljnih rezultata za svaku pojedinu commit poruku.



Slika 17: Prikaz strukture klijentskog dijela VS Code proširenja

4.4.2. Poslužiteljski dio



Slika 18: Prikaz strukture poslužiteljskog dijela

Slično kao što datoteka `extension.ts` predstavlja ulaznu, odnosno središnju točku za klijentski dio alata, tako datoteka `app.ts` fukcionira za poslužiteljski dio alata. U njoj se inicijalizira Express aplikacija putem objekta `app` te se pokreće HTTP poslužitelj na lokalnoj adresi `http://localhost:3066` unutar funkcije `backend`. Ova adresa se koristi prilikom slanja zahtjeva sa klijentske strane na poslužiteljsku.

U istoj datoteci se također instancira jezični model Gemini preko sučelja `GenerativeAI`, koji će se koristiti prilikom evaluacije commit poruka. Također su definirane tri glavne rute: `/fetch-commits` koja dohvaća commit poruke iz odabranog repozitorija putem GitHub API-ja, `/analyze-commits` koja poziva funkciju `analyzeCommitsFromRepo` i vraća rezultate analize u HTML ili JSON formatu, te `/style-comment` koja koristi rezultate analize kako bi izgradila novi prompt za LLM na temelju kojeg se vraća općeniti komentar na stil pisanja commitova. Tako poslužiteljska strana pruža REST sučelje kojim klijentska strana, odnosno VS Code proširenje dohvaća commitove i dobiva evaluacije za prikaz korisniku.

Klasa `Gemini` implementira sučelje `GenerativeAI` i predstavlja spoj prema Gemini modelu koji se koristi za analizu commit poruka. U okviru klase definiraju se sistemske upute (`systemInstructions`) koje se šalju modelu te pomoćne funkcije za generiranje sadržaja i upravljanje pravilima. Funkcija `genAiResponse` šalje konkretan prompt putem `GoogleGenAI` klijenta i vraća odgovor u JSON formatu. Funkcija `analyzeCommits` priprema listu commitova za analizu i šalje ih Gemini modelu na evaluaciju. Tijekom pripreme commitova dodaje se oznaka `|72|` kada je zaglavlje poruke predugačko, jer su jezični modeli poput Geminija bolje prilagođeni analizi teksta nego brojanju znakova. Stoga bi model često pogrešno procijenio duljinu zaglavlja, pa je samo brojanje riješeno u kôdu, a informacija o prekoračenju ograničenja šalje se LLM-u da je uzme u obzir tijekom evaluacije.

U sistemskim uputama se posebno naglašava modelu da ne koristi *conventional commits* specifikacije te da ne smije automatski dodavati reference na pull requestove ili zadatke, s obzirom da u tom području često ima problema s halucinacijama. Radi toga je posebno važno da sistemske upute na ovaj način točno preciziraju okvire unutar kojih model smije vršiti evaluaciju commit poruka. Isto tako, ukoliko commit poruka ne krši nijedno pravilo, prijedlog (`suggestion`) ostaje prazan.

Funkcija `analyzeCommitsFromRepo` predstavlja glavnu integracijsku točku poslužiteljskog dijela. Ona najprije poziva funkciju `fetchCommitMessages` kako bi dohvatila commit poruke iz GitHub repozitorija, zatim šalje te commitove odabranom modelu sučelja `GenerativeAI` (u našem slučaju klasu `Gemini`) i prima JSON odgovor s ocjenama i prekršajima prema definiranoj shemi 5.

```
export class Gemini implements GenerativeAI {
  ...
  getResponseSchema(): SchemaUnion {
    return {
      type: Type.ARRAY,
      items: {
        type: Type.OBJECT,
        properties: {
          commitHash: { type: Type.STRING },
          violations: {
            type: Type.ARRAY,
            items: {
              type: Type.OBJECT,
              properties: { rule: { type: Type.NUMBER } },
            },
          },
          suggestion: { type: Type.STRING },
          bodySuggestion: { type: Type.STRING },
        },
        propertyOrdering: [
          "commitHash",
          "violations",
          "suggestion",
          "bodySuggestion",
        ],
      },
    };
  }
  ...
}
```

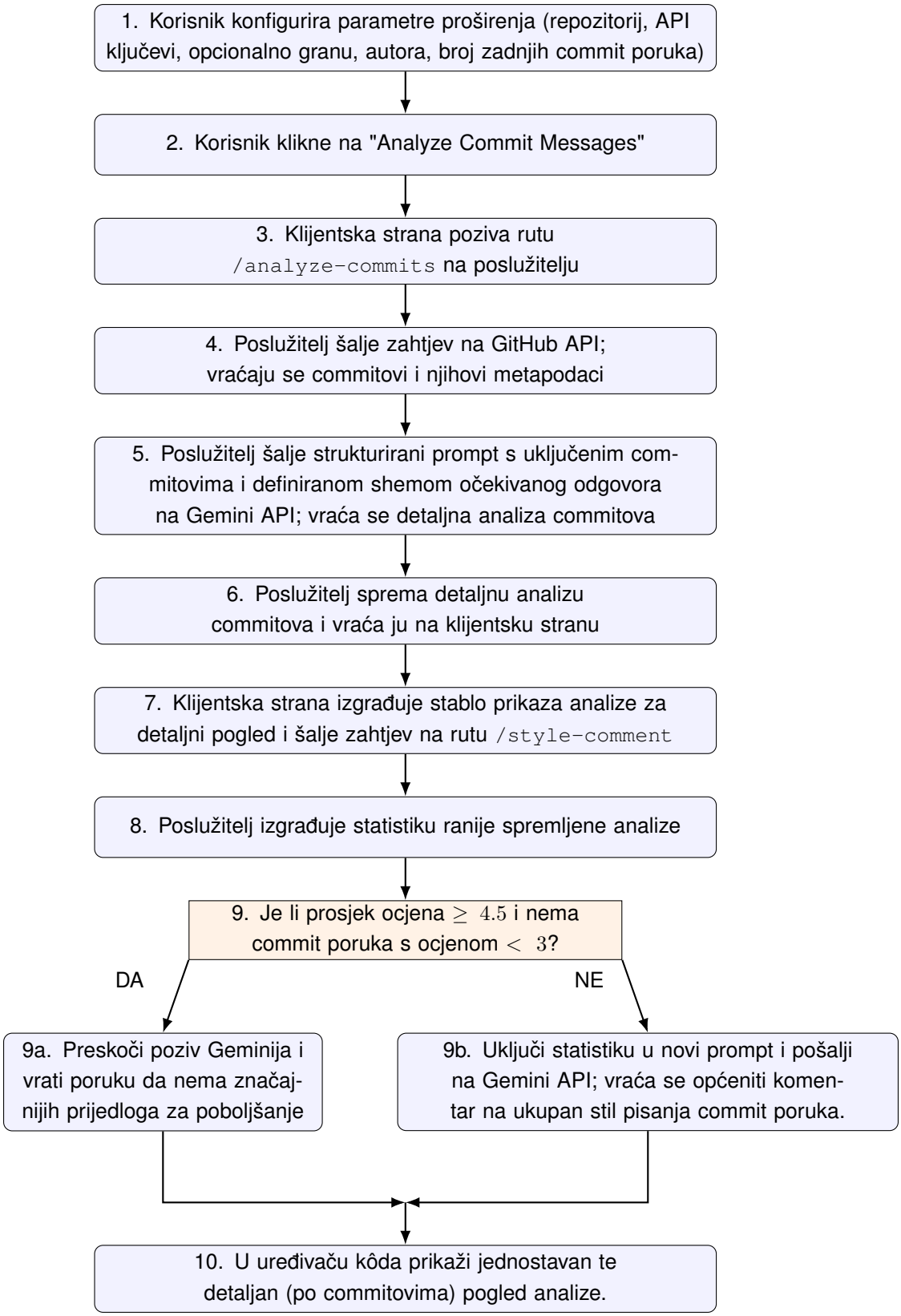
Isječak koda 5: Shema odgovora za rezultate analize iz datoteke `gemini.ts`

Nakon toga odgovori se pretvaraju u objekte `GradedCommit` te zajedno s originalnim commitovima formiraju prikazive entitete `GradedCommitDisplay` koji se vraćaju klijentskom dijelu za prikaz u korisnički odabranom formatu (bočni prozor, HTML webview ili JSON format).

Funkcija `fetchCommitMessages` dohvaća commitove preko `GitHub API`-ja, uz opcionalno filtriranje po grani i autoru. Ujedno provjerava postoje li otvoreni zadaci u repozitoriju kako bi ta informacija bila dostupna tijekom analize. Klasa `Commit` predstavlja pojedinu commit poruku (zajedno s metapodacima), klasa `GradedCommit` pohranjuje rezultate evaluacije (prekršena pravila i prijedloge), dok `GradedCommitDisplay` spaja izvornu poruku i evaluaciju i pruža metode za izradu strukturiranog HTML/JSON prikaza za korisničko sučelje.

Klasa `Gemini` sadrži i funkciju `generateStyleFeedbackFromStats` koja na temelju agregiranih statistika o prekršenim pravilima upućuje drugi (ulančani) zahtjev LLM-u za opći komentar o stilu pisanja commit poruka. Statistika uključuje parove oblika `"pravilo - broj prekršaja/ukupan broj commitova"`, a prompt upućen modelu traži da se mjesta s malo ili nimalo prekršaja naglase kao snaga, dok se za češće prekršaje ponude konkretni savjeti za poboljšanje.

Kada korisnik pokrene analizu commitova klikom na "Analyze Commit Messages", poziva se ruta `analyze-commits`, koja prvo šalje zahtjev na GitHub API radi dohvaćanja zatraženih commitova, zatim po primitku tih commitova šalje zahtjev na Gemini API kako bi LLM vratio rezultate detaljne analize i sugestija za svaku commit poruku. Rezultati analize se spremaju u memoriju (`lastAnalysis`), potom se šalje zahtjev na rutu `/style-comment`, gdje se ti rezultati sumiraju preko pomoćne funkcije `buildViolationStats`, koja računa prosječnu ocjenu, ukupan broj commitova i bilježi eventualne commitove s ocjenom manjom od 3. Ako su prosječna ocjena veća od 4.5 i nijedan commit nema ocjenu manju od 3, vraća se unaprijed definirana poruka pohvale bez pozivanja modela. U suprotnom se poziva funkcija `generateStyleFeedbackFromStats` koja šalje novi prompt LLM-u za generiranje općenitog komentara na temelju ukupne analize, i vraća se personalizirani komentar temeljen na cjelokupnom stilu pisanja commit poruka. Tijek cijelog procesa prikazan je na dijagramu na slici 19.



Slika 19: Cjelokupni radni tok analize commitova.

Na ovaj način sustav uz detaljne evaluacije pojedinačnih commitova pruža i sažet, mentorski orijentiran pregled koji korisniku daje širu sliku o kvaliteti pisanja commit poruka.

5. Verifikacija i validacija

Verifikacija i validacija predstavljaju dvije najvažnije faze u metodologiji znanosti o oblikovanju, a cilj im je potvrditi korisnost razvijenog artefakta u praksi. U okviru ovog rada provedene su te dvije faze kroz demonstraciju i evaluaciju programskog rješenja. One omogućuju provjeru koliko je alat usklađen s definiranim zahtjevima te koliko ga korisnici intuitivno razumiju i procjenjuju korisnim u stvarnom kontekstu upotrebe unutar edukacije programskog inženjerstva.

5.1. Plan provedbe verifikacije i validacije

Za provedbu verifikacije i validacije odabrali smo pristup fokus grupe, jer omogućuje kombiniranje strukturiranog prikupljanja podataka (putem ankete) i otvorene diskusije kroz koju se mogu dobiti detaljniji uvidi u korisnička iskustva. Provedene su dvije fokus grupe.

Fokus grupe bile su sastavljene od dvije skupine sudionika - stručnjaka i krajnjih korisnika. Stručnu skupinu činila su dva sveučilišna nastavnika i tri asistenta s iskustvom u poučavanju kolegija vezanih uz programsko inženjerstvo te jedan stručnjak iz industrije s višegodišnjim iskustvom u razvoju softverskih rješenja i timskom radu. Korisničku skupinu činili su studenti različitih razina studija (preddiplomski i diplomski) s većim ili manjim iskustvom u korištenju Git sustava u okviru fakultetskih projekata. Takav odabir sudionika osigurao je kombinaciju perspektiva onih koji prenose znanje, onih koji ga koriste u praksi, kao i onih kojima je alat primarno namijenjen.

Na početku smo definirali protokol po kojem smo proveli verifikaciju i validaciju alata. Sastojao se od sljedećih koraka:

- Okupiti sudionike na online sastanku ili uživo.
- Kroz 15-20 minuta sudionicima demonstrirati alat i kako funkcionira.
- Omogućiti sudionicima da isprobaju alat na jednom ili dva vlastita projekta (repozitorija).
- Pitati sudionike za dojmove i prodiskutirati alat kroz polustrukturirani intervju.
- Dati sudionicima da odgovore na upitnik od nekoliko pitanja putem Google Forms.

5.2. Demonstracija programskog rješenja

Faza demonstracije provedena je tako da su članovima fokus grupe prikazane glavne funkcionalnosti alata. Tijekom demonstracije pokazani su konkretni primjeri analize commit poruka na odabranom repozitoriju. Rezultati su uključivali:

- ocjene pojedinih commit poruka

- prikaz prekršenih pravila
- generirane prijedloge poboljšanja (zaglavlje i tijelo)
- agregirani komentar o ukupnom stilu commit poruka

Na ovaj način fokus grupa je mogla vidjeti puni radni tok korištenja alata, od konfiguracije parametara do prikaza rezultata u uređivaču kôda. Time su sudionici dobili jasnu predodžbu o tome kako bi se alat mogao koristiti u nastavi i svakodnevnom radu. Sudionici su u bilo kojem trenutku mogli postavljati pitanja ili zatražiti dodatna pojašnjenja o funkcionalnostima alata.

5.3. Evaluacija programskog rješenja

U petoj fazi metodologije znanosti o oblikovanju naglasak je na evaluaciji, tj. provjeri korisnosti alata u praksi. Nakon demonstracije, članovi fokus grupe dobili su priliku samostalno isprobati alat na vlastitim repozitorijima. Tijekom tog procesa sudionici su mogli vidjeti kako se alat ponaša na njihovim konkretnim projektima i koliko su rezultati analize primjenjivi na njihov kontekst.

Nakon isprobavanja alata sudionici su ispunili anketu u Google Forms te sudjelovali u polustrukturiranom intervjuu. Anketna pitanja pokrivala su aspekte razumljivosti alata, jednostavnosti korištenja, korisnosti u nastavnom i projektnom kontekstu, kao i potencijalne koristi za studente u poboljšanju vještina pisanja commit poruka. Osim toga, sudionici su mogli navesti što im se najviše i najmanje svidjelo te koje bi dodatne funkcionalnosti voljeli vidjeti.

5.3.1. Provođenje ankete za evaluaciju alata

U sklopu fokus grupe krajnjih korisnika bili su uključeni i studenti iz inozemstva, te su pitanja ankete za njih bila prevedena i na engleski jezik kako bi se omogućilo i njihovo sudjelovanje.

5.3.1.1. Pitanja

Kako bi se dobila sveobuhvatna slika o iskustvima korisnika, evaluacijska anketa je obuhvaćala pitanja zatvorenog i otvorenog tipa. Zatvorena pitanja omogućila su kvantitativnu analizu stavova sudionika, primjerice kroz ocjene razumljivosti i jednostavnosti korištenja alata, dok su otvorena pitanja davala prostor za dublje razmišljanje i osobne komentare o prednostima i nedostacima alata. U tablici 7 su prikazana konkretna pitanja korištena u anketi.

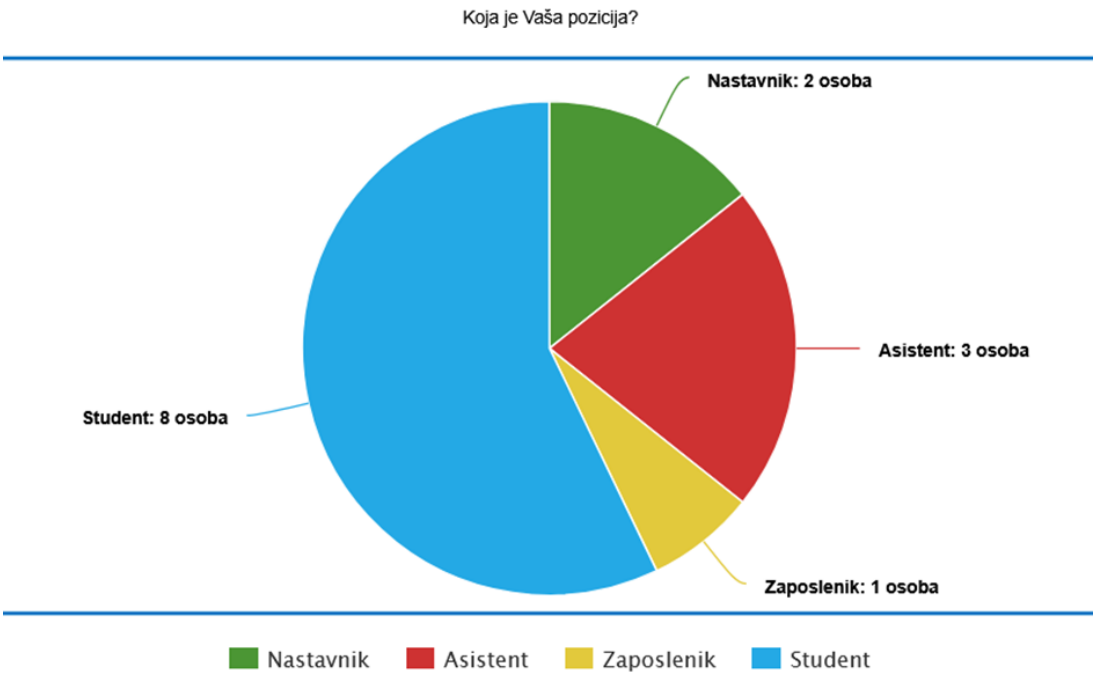
Tablica 7: Pitanja korištena u anketi fokus grupe

#	Pitanje	Tip pitanja	Primjeri odgovora
1	Koja je vaša pozicija?	Višestruki izbor (jedan odgovor)	Student; Nastavnik; Stručnjak iz industrije
2	Koliko vam je alat bio jasan i lagan za korištenje?	Likert skala (1–5)	1 = Nimalo jasan ... 5 = Vrlo jasan
3	Biste li koristili ovakav alat u svojoj nastavi/projektima?	Višestruki izbor (jedan odgovor)	Da, redovito; Moguće, povremeno; Vjerojatno ne
4	Koliko bi vam alat olakšao praćenje i evaluaciju vlastitog (ako ste student), odnosno studentskog (ako ste nastavnik) rada?	Likert skala (1–5)	1 = Nimalo ... 5 = Uvelike
5	Mislite li da bi alat pomogao studentima da poboljšaju svoje vještine dokumentiranja i pisanja commit poruka?	Likert skala (1–5)	1 = Nimalo ... 5 = Uvelike
6	Što vam se najviše svidjelo u alatu?	Otvoreni odgovor (kratki tekst)	npr. “Integracija s VS Code”, “Vizualni prikaz ocjena”, “Jednostavno korištenje”
7	Koje dodatne funkcionalnosti ili mogućnosti biste voljeli imati?	Otvoreni odgovor (dulji tekst)	npr. “Povijest ocjena”, “Izvoz u CSV/JSON”, “Pravila prilagođena kolegiju”
8	Što vam se najmanje svidjelo u alatu?	Otvoreni odgovor (dulji tekst)	npr. “Predugo trajanje analize”, “Nejasne poruke o greškama”

5.3.1.2. Kvantitativni rezultati

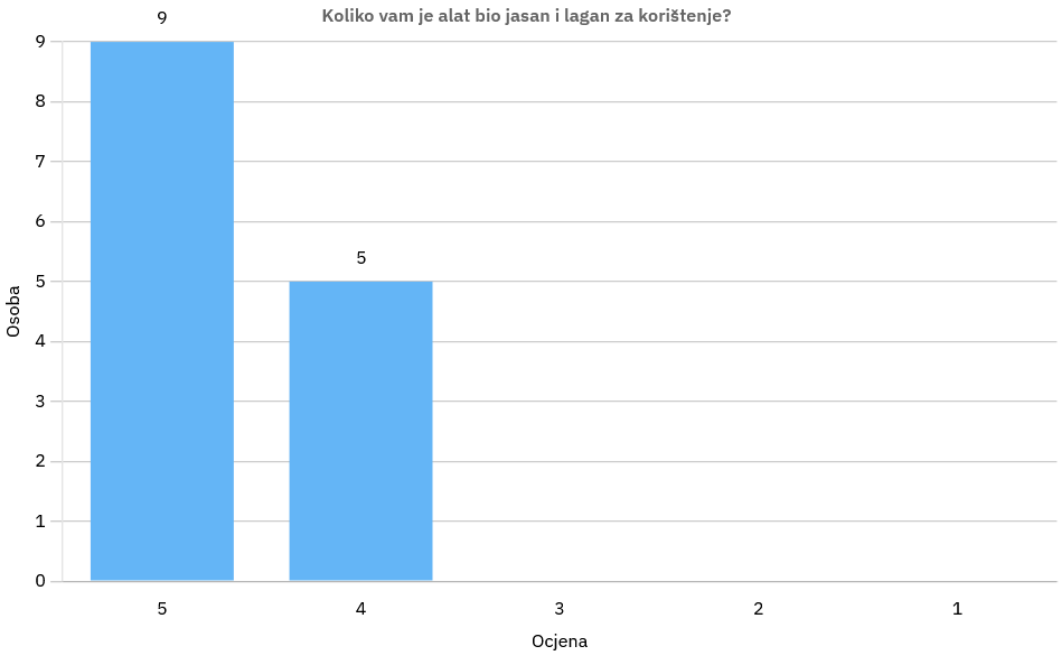
U ovom poglavlju predstavljeni su kvantitativni rezultati provedene ankete, odnosno pitanja zatvorenog tipa koja koriste Likertove skale, da/ne te odabir između ponuđenih opcija. Ti odgovori se mogu numerički obraditi i prikazati grafički, što je ovdje učinjeno kroz kružne i stupčaste dijagrame.

Kao što se vidi iz raspodjele sudionika predstavljene na slici 20, ukupno smo imali 14 sudionika, od kojih je bilo 8 studenata i 6 stručnjaka. Od stručnjaka, 5 ih je imalo poziciju u edukaciji programskog inženjerstva - točnije, 2 nastavnika i 3 asistenta - dok je jedan stručnjak bio zaposlenik iz industrije - softverski inženjer na senior poziciji, odnosno s dugogodišnjim iskustvom rada u softverskim kompanijama.



Slika 20: Pitanje 1: Koja je Vaša pozicija?

Iz slike 21 vidimo da je većina sudionika procijenila alat jasnim i intuitivnim za korištenje najvišom ocjenom. Kako je najniža ocjena i dalje bila 4 (vrlo dobar), svi su se sudionici složili da im alat nije bio previše složen te je time ispunio cilj pristupačnosti.



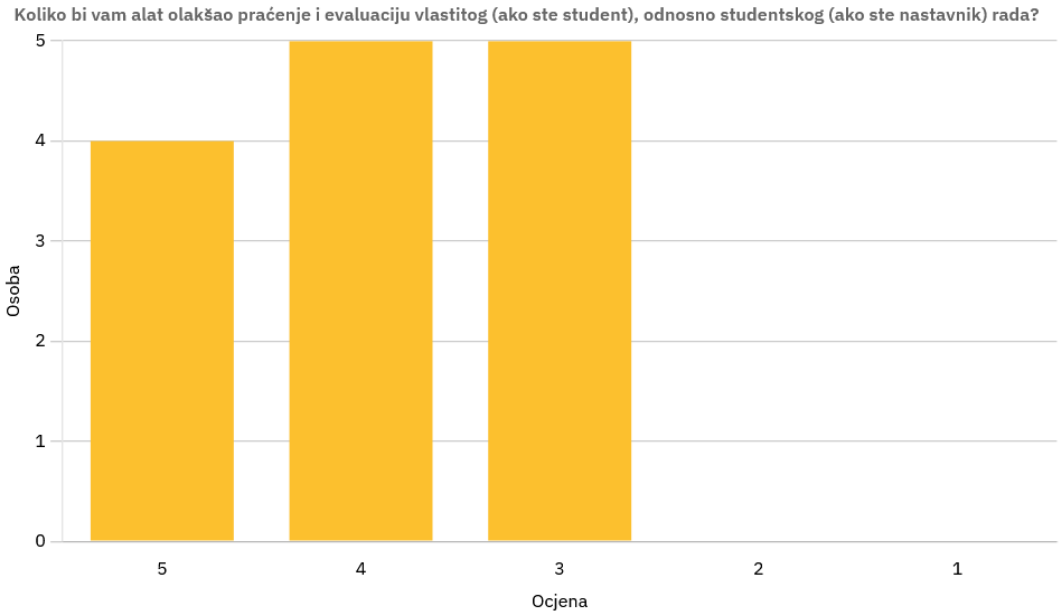
Slika 21: Pitanje 2: Koliko vam je alat bio jasan i lagan za korištenje?

Za pitanje o korištenju alata u nastavi (za nastavnike) odnosno u vlastitim projektima (za studente), slika 22 pokazuje da je velika većina sudionika (85.7%, odnosno 12 ljudi) bila

otvorena za mogućnost povremenog korištenja alata po potrebi, dok je manji broj (14.3%, odnosno dvoje ljudi) izjavio da bi ga sigurno redovito koristili. Kako je alat zamišljen kao potpora u nastavnom procesu, kao i samostalnom razvoju, vidimo vrijednost već i u povremenom korištenju, kako bi studenti mogli s vremena na vrijeme imati strukturirani način za procjenu kako se njihove vještine dokumentiranja softvera i pisanja commit poruka poboljšavaju kroz vrijeme. U tom smislu je ohrabrujuća činjenica da su svi sudionici procijenili da vide korist u uporabi alata, odnosno niti jedan sudionik nije izjavio da ga ne bi uopće koristio.

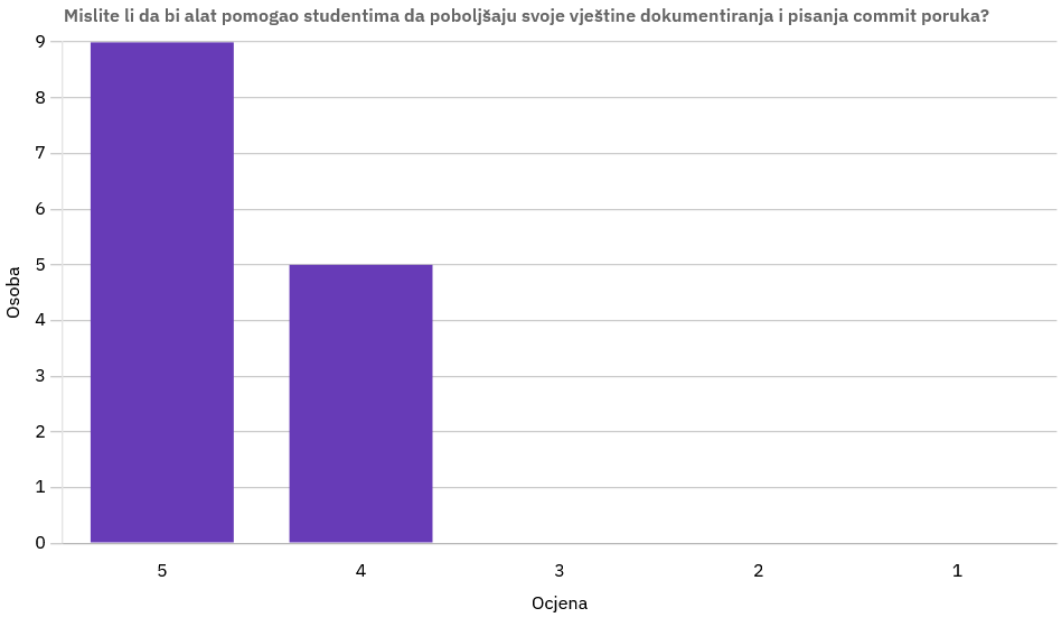
Slika 22: Pitanje 3: Biste li koristili ovakav alat u svojoj nastavi/projektima?

Odgovori na pitanje o procjeni koliko bi alat olakšao praćenje i evaluaciju vlastitih commit poruka (za studente), odnosno studentskih commit poruka (za nastavnike), predstavljeni na slici 23, pokazuju veći raspon procjena od drugih odgovora. Procjene su u rangu od 3 (osrednje) do 5 (najviše), što ukazuje na to da je konsenzus općenito pozitivan, no da razlike u procjeni mogu postojati zbog toga što je, primjerice, dio sudionika procijenio da alat još treba određenu prilagodbu ili dodatnu funkcionalnost kako bi se bolje prilagodio njihovoj specifičnoj potrebi, odnosno načinu kako bi ga koristili u nastavnom procesu. Ovi razlozi su detaljnije istraženi u diskusiji kroz polustrukturirane intervjue (poglavlje 5.3.2).



Slika 23: Pitanje 4: Koliko bi vam alat olakšao praćenje i evaluaciju vlastitog (ako ste student), odnosno studentskog (ako ste nastavnik) rada?

Konsenzus za pitanje o potencijalu alata za poboljšanje studentskih vještina je pozitivan (slika 24), s obzirom da je većina sudionika (9 sudionika, tj. 64.3%) za tu procjenu dala najvišu ocjenu 5, dok su svi ostali sudionici dali vrlo dobru ocjenu 4. Ovo je bila jedna od ključnih procjena evaluacije, pogotovo od strane nastavnika, o čemu smo detaljnije raspravljali kroz diskusiju (poglavlje 5.3.2).



Slika 24: Pitanje 5: Mislite li da bi alat pomogao studentima da poboljšaju svoje vještine dokumentiranja i pisanja commit poruka?

5.3.1.3. Kvalitativni rezultati

U ovom dijelu prikazan je sažetak povratnih informacija dobivenih od sudionika na temelju pitanja iz ankete koja su bila otvorenog tipa gdje sudionici slobodno odgovaraju. Ovdje se podaci analiziraju kroz tematsko grupiranje, interpretaciju i citiranje reprezentativnih odgovora, a ne kroz statističku obradu. Poseban naglasak stavljen je na citiranje reprezentativnih odgovora kako bi se vjernije prenijela iskustva i stavovi sudionika.

Na anketno pitanje "Što vam se najviše svidjelo u alatu?", većina sudionika istaknula je kako im se najviše svidjela jednostavnost alata i njegova integracija sa VS Code okruženjem. Kako je jedan sudionik naveo: „Sviđa mi se jednostavnost i integracija u VSC. Naravno sadržaj poruka je bitan, ali već i ovo ima svoju primjenu i vidim gdje bih to mogao koristiti."

Sudionici su također naglasili vrijednost detaljne analize commitova: „Detaljna analiza po commitovima kao i po greškama (violations)", dok je drugome najviše odgovarala „Jasnoća opisa grešaka i prijedlog moguće zamjene".

Pozitivno su ocijenjeni i dodatni elementi poput pregleda statistike te rezimea, odnosno podjele na jednostavan te na detaljan pogled u bočnom prozoru: „Sviđa mi se da ima summary statistika i detaljni prikaz koji se može pregledavati", ali i mogućnost personalizacije pregleda: „Mogućnost filtriranja commiteva po korisniku".

Nekoliko sudionika je istaknulo vrijednost povratne informacije: „Jasni feedback s porukom", dok je drugima najkorisnija bila primjena pravila za konzistentnost jezika u repozitoriju (pravilo 5): „Provjera konzistentnosti jezika kojim se piše opis commita u kontekstu repozitorija".

Na pitanje "Koje dodatne funkcionalnosti ili mogućnosti biste voljeli imati?", sudionici su mogli navesti vlastite prijedloge za proširenje funkcionalnosti koje bi im koristile u radu. Nastavnici su posebno naglasili potrebu za masovnom analizom više repozitorija odjednom: „Volio bih da mogu navesti više repozitorija i da se analiziraju masovni svi ti repozitoriji. Kao nastavniku najviše mi koristi kada mogu automatizirati neke provjere odjednom na svim studentima i da nakon toga samo pregledam." Iz ovoga možemo uvidjeti potrebu za pojačanim naglaskom na automatizaciji alata kako bi se olakšao radni tok u njegovoj primjeni u nastavnom procesu.

Osim toga, još jedan prijedlog koji se ponavljao je i šira podrška za repozitorije izvan GitHub platforme: „Svakako bih volio podršku izvan GitHuba jer koristim i vlastite custom repozitorije i GitLab". Drugi prijedlozi uključivali su „Integraciju s više LLM-ova", kao i mogućnost analize commit poruke prije potvrđivanja commita: „Analiza poruke i opisa prije nego je commit potvrđen."

Neki sudionici su još predložili i naprednije mogućnosti poput automatskog prepisivanja povijesti: „Preview i/ili automatski rewrite povijesti prema sugestijama", kao i opciju da alat generira commit poruke umjesto korisnika: „Da piše umjesto tebe commit poruku." Ova dva prijedloga smo ocijenili kao jedine koje vjerojatno ne bismo uzeli u obzir tijekom nadogradnje alata.

Prvi prijedlog se ne može provesti iz razloga što commit, kao osnovna jedinica povijesti u sustavima za upravljanje verzijama, ima ulogu povijesnog, odnosno trajnog zapisa o stanju

projekta u određenom trenutku, kao što je objašnjeno u poglavlju 2. Commit poruka, zajedno s hashom koji se izračunava na temelju svakog commita, predstavlja nepromjenjivi identifikator promjene, a izmjena već postojećih commitova značila bi brisanje ili prepisivanje povijesti repozitorija. Takav postupak doveo bi do gubitka izvornog traga promjene u kôdu, a time i gubitka integriteta i povjerenja u repozitorij, kao i mogućih konflikata u kôdu u suradničkom radu. Stoga alat može predlagati poboljšanja za prošle commit poruke kako bi korisnici na temelju toga prilagodili svoj stil pisanja za buduće commit poruke, ali automatsko prepisivanje povijesti nije izvedivo ni u skladu s idejom samog commita. Štoviše, ovakva priroda funkcioniranja commitova naglašava samu važnost poboljšavanja stila pisanja smislenih commit poruka.

Što se tiče drugog prijedloga, on je za razliku od prvog prijedloga tehnički izvediv, no kako je ranije spomenuto, već je u fazi dizajna rješenja odlučeno da neće biti implementiran zbog rizika da bi studenti primarno koristili takvu funkcionalnost za korištenje alata da piše commit poruke umjesto njih koje oni onda ne bi niti morali sami pogledati, što bi bilo nauštrb njihovom kritičkom razmišljanju te postiglo suprotan učinak u smislu da bi zapravo umanjilo njihove vještine dokumentiranja kôda, a posljedično i njihovu upućenost u stanje samog kôda na određenom projektu. Međutim, za ovu stavku su nastavnici tijekom polustrukturiranog intervjua spomenuli moguću alternativu koja bi zadržala komponentu kritičkog razmišljanja, što će biti detaljnije komentirano u sljedećoj sekciji koja se odnosi na intervjue.

Konačno, neki sudionici su u anketi spomenuli i želju za većom prilagodljivošću: „Dodavanje svojih rule-ova”, ali i za metrikama poput veličine commita: „Prijedlog o veličini commit-a, (ne sadržajno, no commit ne bi trebao biti 1000+ linija add/remove)”. Opciju za dodavanje vlastitih pravila vidimo kao nešto što bi nastavnici mogli koristiti u slučaju da žele definirati vlastite prakse pisanja commit poruka koje se traže u okvirima njihovih kolegija i/ili smjernica za određene studentske projekte, te bi ju bilo korisno imati za olakšanje korištenja alata od strane nastavnika. Isto tako, definiranje dodatnog pravila da LLM tijekom analize uzima u obzir i veličinu samog kôda tijekom jednog commita bi svakako moglo biti korisno za proširivanje opsega analize, kako je u dokumentiranju kôda važno ne samo pisati smislene commit poruke, već i stavljati u commit manje jedinice kôda od kojih svaka idealno ostvaruje samo jednu promjenu kao svoju svrhu, ne više promjena (što je pogotovo bitno za razumljivost kôda iz perspektive stručnjaka koji imaju iskustvo rada u softverskim kompanijama). Kako smo se u ovom radu ograničili na analizu samih commit poruka, ovakva funkcionalnost nije bila uključena u trenutnoj verziji alata.

Na pitanje "Što vam se najmanje svidjelo u alatu?", dio sudionika naveo je tehnička ograničenja, primjerice: „Da je trenutno ovisan o GitHubu” ili „Ograničenje broja analiziranih commitova”. Ova ograničenja su primarno radi svrhe stvaranja prve funkcionalne verzije alata koja se fokusirala samo na integraciju s GitHubom kao s dominantno korištenom platformom, te je broj analiziranih commitova bio ograničen radi uštede tokena. Sljedeća funkcionalna verzija alata bi povećala ove opcije, pogotovo u vidu analize većeg broja commita kako bi omogućila analizu na razini cijelog jednog, ili čak više, repozitorija, što bi bilo osobito korisno u nastavnom procesu za analizu studentskih repozitorija od strane nastavnika i dobivanja ocjena i pregleda u generalno poštivanje dobrih praksi commit poruka po studentu.

Dio ispitanika nije imao posebnih primjedbi, primjerice: „Ništa specifično, alat ima jasno ograničenu svrhu i jasno prezentira rezultate”, ili jednostavno: „Sve je bilo okej. Alat mi se u potpunosti svidio.”

Jedan od komentara dotaknuo se i šireg konteksta korištenja: „Postoji mogućnost da se korisnik ponese s gamifikacijom te da se jako fokusira na postizanje visokog ratinga umjesto da se fokusira komunicirati sebi/suradnicima svrhu commita.”, što je zanimljiva perspektiva koju nismo razmatrali, no isto tako otvara mogućnost pozitivne primjene gamifikacije kao većeg poticaja za redovito korištenje alata.

5.3.2. Provođenje polustrukturiranog intervjua

Kako bi se detaljnije validirala korisnost razvijenog alata, provedeni su i polustrukturirani intervjui sa sudionicima. Intervjui su omogućili dublje razumijevanje stavova sudionika te kontekstualizaciju rezultata dobivenih putem ankete.

U intervjuu sa stručnjakom iz industrije (senior developer), reakcije su bile izrazito entuzijastične i alat mu se činio vrlo korisnim. Stručnjak je dodatno komentirao da razmišlja bi li mogao uvjeriti svoj tim unutar kompanije da ga koristi, iako je komentirao mogući otpor unutar timova u industrijskoj praksi zbog preferencije za brzinom koja ponekad osujećuje praksu dobrog dokumentiranja.

U intervjuu sa studentima, komentari su u većini slučajeva bili u skladu s odgovorima iz ankete. Posebno se istaknuo slučaj jednog studenta koji je, dok je čekao analizu vlastitog repozitorija, procijenio da će ukupna ocjena biti oko 3,3. Rezultat alata zaista je bio vrlo blizu (3,25), što upućuje na to da je alat davao rezultate koji su se podudarali s korisničkim očekivanjima (u čemu zasigurno pomaže i jasno navođenje pravila unutar proširenja po kojima se analiziraju commitovi), čime se povećava povjerenje u njegovu pouzdanost.

Intervjui s nastavnicima dali su, očekivano, najkorisniju povratnu informaciju, osobito u smislu validacije primjenjivosti alata u edukativnom procesu. Jedan od komentara bio je kako nema potrebe za forsiranjem korištenja umjetne inteligencije za zadatke za koje je dovoljan program. Ova primjedba odnosi se na odobravanje činjenice da alat već samostalno provodi provjeru tehničkih pravila, poput broja znakova u zaglavlju, budući da se pokazalo da je veliki jezični model u takvim zadaćama nepouzdaniji.

Nastavnici su također istaknuli potencijalne izazove u korištenju, primjerice mogućnost da bi nastavnicima bilo nepraktično ručno unositi URL svakog studentskog repozitorija. Na taj je komentar pojašnjeno da je alat primarno zamišljen za korištenje od strane samih studenata, dok nastavnici samo osiguravaju pristup i okvirnu podršku.

Što se tiče primjenjivosti u edukativnom kontekstu, nastavnici su predložili nekoliko mogućih scenarija korištenja. Primjerice, studentima bi se moglo omogućiti da mjesec dana rade na svojim projektima za kolegij i tek tada pokrenu alat kako bi dobili početnu analizu. Nakon toga, u drugoj fazi rada na projektu, studenti bi mogli ponovno pokrenuti alat i očekivalo bi se poboljšanje ocjena stila commit poruka kako napreduju kroz semestar. Na taj bi se način alat koristio kao podrška učenju i refleksiji. Također, spomenuta je ideja da bi bilo korisno da se pri

pisanju commit poruke u realnom vremenu prikazuje informacija o tome jesu li neka pravila prekršena. U diskusiji se naglasilo da u tom kontekstu alat ne bi trebao davati konkretne prijedloge za zaglavlje ili tijelo poruke (kao što to radi za već postojeće commitove), kako bi se izbjeglo nekritičko kopiranje, već samo poticaj studentima da sami preispitaju i poboljšaju poruku prije potvrđivanja commita.

5.4. Procjena učinkovitosti

Detaljni osvrt na kvantitativne i kvalitativne rezultate provedene ankete te komentara iz polustrukturiranih intervjua ostvaren je u prethodnim sekcijama zajedno s dokumentacijom rezultata. Ovo poglavlje daje općeniti osvrt na cjelokupnu povratnu informaciju od strane fokus grupe, odnosno je li faza verifikacije i validacije pokazala korisnost alata, pogotovo za edukativni kontekst.

Analiza povratnih informacija fokus grupe pokazala je da je alat u većini slučajeva ocijenjen kao intuitivan i jednostavan za korištenje. Stručnjaci (nastavnici i industrijski stručnjaci) istaknuli su da alat može poslužiti kao vrijedan dodatak nastavnom procesu jer omogućuje praćenje kvalitete commit poruka, što je dosad bilo zanemareno područje. Studenti su se složili da bi alat mogao pomoći u boljem razumijevanju toga što čini dobru commit poruku te da im je korisna konkretna povratna informacija LLM-a s prijedlozima za poboljšanje.

Iako su navedene neke mogućnosti za poboljšanje (npr. proširenje broja pravila po kojima se procjenjuju commit poruke, integracija s dodatnim platformama osim VS Codea i GitHuba), opći zaključak je da je alat pokazao svoju korisnost i opravdao očekivanja u obrazovnom i praktičnom kontekstu. Time je u skladu s načelima metodologije znanosti o oblikovanju pokazano da razvijeni artefakt ima stvarnu vrijednost i doprinosi rješavanju prepoznatog problema, te da s predloženim poboljšanjima može samo još više doprinijeti tome cilju.

6. Zaključak

Ovaj rad prikazuje primjenu metodologije znanosti o oblikovanju u razvoju programskog rješenja čija je svrha podržati studente i programere u razvoju ključne vještine pisanja kvalitetnih commit poruka. Commit poruke, kao temeljni oblik dokumentiranja verzija kôda, imaju presudnu ulogu u timskom radu i dugoročnoj održivosti softverskih projekata, stoga njihova kvaliteta izravno utječe na učinkovitost razvoja.

Kroz rad je pokazano kako se veliki jezični modeli mogu iskoristiti na jednostavan i troškovno prihvatljiv način u edukativnom kontekstu. Naglasak nije stavljen na pružanje gotovog i jedinstvenog rješenja za svaku commit poruku, već na poticanje kritičkog razmišljanja kod studenata. Alat služi kao potpora u procesu samoevaluacije jer analizira snage i slabosti prošlih commit poruka na temelju dobrih praksi definiranih u literaturi i široko prihvaćenih u profesionalnoj zajednici, što omogućuje korisnicima da sami prepoznaju i usvoje kvalitetnije obrasce pisanja.

Glavni doprinosi ovog rada su:

- razvoj alata koji analizira commit poruke prema definiranim pravilima temeljenima na dobrim praksama iz literature i prihvaćenima od strane programske zajednice,
- integracija alata u popularan uređivač kôda (VS Code), čime se omogućuje studentima i ostalim korisnicima lako dostupna povratna informacija tijekom rada na projektima,
- demonstracija korisnosti alata kroz fokus grupu koja je uključila i stručnjake i krajnje korisnike,
- primjena velikih jezičnih modela u edukativnom potencijalu alata s naglaskom na zadržavanje i poticanje kritičkog razmišljanja,
- doprinos nastavnom procesu kroz alat koji smanjuje opterećenje nastavnicima i studentima pomaže razvijati vještine pisanja kvalitetnih commit poruka.

Ograničenja istraživanja uključuju manji broj sudionika u evaluaciji, relativno uski skup pravila na kojima se alat temelji te ovisnost o API-jima velikih jezičnih modela, što sa sobom nosi rizik dodatnih troškova i promjena uvjeta korištenja od strane pružatelja usluge. Ipak, s obzirom na prirodu upotrebe i dostupne besplatne kvote Gemini modela, većini korisnika alat ostaje praktičan i lako dostupan.

Smjerovi budućeg razvoja uključuju proširenje pravila evaluacije, omogućavanje analize na razini jednog ili više repozitorija te integraciju s dodatnim alatima i okruženjima (primjerice GitHub Pull Requests, GitLab ili Microsoft Visual Studio). Time bi se dodatno povećala prilagodljivost i stupanj automatizacije u obrazovnom procesu, uz zadržavanje jednostavnosti i intuitivnosti korištenja koja je posebno pohvaljena kroz validaciju. Nadalje, šira evaluacija s većim brojem korisnika omogućila bi veću potvrdu korisnosti alata i detaljniji uvid u njegove nedostatke.

Logičan nastavak istraživanja bio bi razvoj rješenja neovisnog o VS Code okruženju, s mogućnošću korištenja namjenskog poslužitelja i modela "platforme kao usluge" (PaaS) za analizu commit poruka. Takav pristup otvorio bi put prema širem korištenju i integraciji u različite razvojne okoline, čime bi se dodatno osnažila njegova praktična i obrazovna vrijednost.

Popis literature

[1] K. Peffers, T. Tuunanen, M. A. Rothenberger i S. Chatterjee, „A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, sv. 24, br. 3, str. 45–77, 2007. DOI: 10.2753/MIS0742-1222240302.

[2] S. Turney. „Systematic Review | Definition, Example & Guide." (2022.), adresa: https://www.scribbr.com/methodology/systematic-review/ (pogledano 25. 8. 2025.).

[3] Elicit, *Elicit: The AI Research Assistant*, 24. 1. 2023.

[4] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang i H. Liu, „What Makes a Good Commit Message?" *Proceedings of the 44th International Conference on Software Engineering*, 2022., str. 2389–2401.

[5] S. Group, „CHAOS Report," The Standish Group International, teh. izv., 1995., Available: https://www.standishgroup.com/.

[6] I. Ma i C. V. Lopes, „Improving the Quality of Commit Messages in Students' Projects," *2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG)*, 2023., str. 1–4. DOI: 10.1109/SEENG59157.2023.00005.

[7] T. P.-W.
bibinitperiod et al. „Semantic Versioning 2.0.0," Semantic Versioning 2.0.0. (2025.), adresa: https://semver.org/ (pogledano 23. 7. 2025.).

[8] M. Rochkind, „The Source Code Control System," *Software Engineering, IEEE Transactions on*, sv. SE-1, br. 4, str. 364–370, 1975., ISSN: 0098-5589. DOI: 10.1109/TSE.1975.6312866.

[9] Sun Microsystems, Inc., „SCCS Source Code Control System," *Programming Utilities Guide*, Broj dokumenta: 802-5880, Mountain View, CA, U.S.A.: Sun Microsystems, Inc., 1997., pogl. 5.

[10] W. F. Tichy, „Design, implementation, and evaluation of a Revision Control System," *Proceedings of the 6th International Conference on Software Engineering*, serija ICSE '82, Tokyo, Japan: IEEE Computer Society Press, 1982., str. 58–67.

[11] D. Grune, „Concurrent Versions System, A Method for Independent Cooperation," Vrije Universiteit, Amsterdam, IR 113, 1986.

[12] B. Collins-Sussman, B. W. Fitzpatrick i C. M. Pilato, *Version Control with Subversion*, Subversion 1.7 (compiled from r6069). O'Reilly Media, 2011.

[13] BitMover. „BitKeeper - Distributed source management and version control," BitKeeper. (2000.), adresa: https://web.archive.org/web/20000617153000/http://bitkeeper.com/bk07.html (pogledano 23. 7. 2025.).

[14] T. Blau. „Git turns 20: A Q&A with Linus Torvalds," The GitHub Blog. (2025.), adresa: https://github.blog/open-source/git/git-turns-20-a-qa-with-linus-torvalds/ (pogledano 23. 7. 2025.).

[15] R. team. „Version Control Systems Popularity in 2025," RhodeCode blog. (2025.), adresa: https://rhodecode.com/blog/156/version-control-systems-popularity-in-2025 (pogledano 26. 7. 2025.).

[16] L. Torvalds. „Git commit b659015: "Add some information about properly formatted commit messages"." (2011.), adresa: https://github.com/torvalds/subsurface-for-dirk/commit/b6590150d68df528efd40c889ba6eea476b39873 (pogledano 23. 7. 2025.).

[17] Gerrit Code Review. „Signed-off-by Lines." verzija v3.12.2-1016-g59ec58973b. (2025.), adresa: https://gerrit-review.googlesource.com/Documentation/user-signedoffby.html (pogledano 23. 7. 2025.).

[18] Thiago Macieira. „Git commit 99b8e85: "Update the README file part about commit messages"." (2014.), adresa: https://github.com/torvalds/subsurface-for-dirk/commit/99b8e85d739387c0c8342535b28e010cdac74a5f (pogledano 23. 7. 2025.).

[19] GitKraken. „Writing a Good Git Commit message," GitKraken. (2025.), adresa: https://www.gitkraken.com/learn/git/best-practices/git-commit-message (pogledano 23. 7. 2025.).

[20] Conventional Commits. „Conventional Commits Specification v1.0.0." A lightweight convention on top of commit messages. (2019.), adresa: https://www.conventionalcommits.org/en/v1.0.0/ (pogledano 26. 7. 2025.).

[21] S. J. Russell i P. Norvig, ur., *Artificial Intelligence: A Modern Approach* (Pearson Series in Artificial Intelligence), 4. izdanje, u sur. s M. Chang, J. Devlin, A. Dragan i dr. Harlow, UK: Pearson Education Limited, 2022., 1166 str., ISBN: 978-1-292-40113-3.

[22] Y. LeCun, Y. Bengio i G. Hinton, „Deep learning," *Nature*, sv. 521, br. 7553, str. 436–444, 2015.

[23] B. v. d. M. L. Barkley, „Investigating the Role of Prompting and External Tools in Large Language Models," *arXiv preprint arXiv:2410.19385*, 2024.

[24] A. Vaswani, N. Shazeer, N. Parmar i dr., „Attention is All You Need," *arXiv preprint arXiv:1706.03762*, 2017.

[25] L. Ouyang, J. Wu, X. Zhai i et al., „Training language models to follow instructions with human feedback," *arXiv preprint arXiv:2203.02155*, 2022.

[26] S. Choi i H. Kim, „The impact of a large language model-based programming learning environment on students' motivation and programming ability," *Education and Information Technologies*, sv. 30, str. 8109–8138, 2025. DOI: 10.1007/s10639-024-13107-x.

[27] I. P. Alomar i I. P. Mkaouer, „Podrška kulturi pregleda koda koristeći LLM-ove,” *Naziv časopisa ili konferencije*, 2024.

[28] W. Hou i Z. Ji, „Comparing Large Language Models and Human Programmers for Generating Programming Code,” *Advancement of Science*, 2024.

[29] S. G. Patil, T. Zhang, X. Wang i J. E. Gonzalez, „Gorilla: Large Language Model Connected with Massive APIs,” *Neural Information Processing Systems*, 2023.

[30] E. Shin, Y. Yu, R. R. Bies i M. Ramanathan, „Evaluation of ChatGPT and Gemini Large Language Models for Pharmacometrics with NONMEM,” *Journal of Pharmacokinetics and Pharmacodynamics*, 2024.

[31] M. A. Haider, A. B. Mostofa, S. M. Hossain, A. Iqbal i T. Ahmed, „Prompting and Fine-Tuning Large Language Models for Automated Code Review Comment Generation,” *arXiv.org*, 2024. DOI: 10.48550/arXiv.2411.10129.

[32] R. Elgedawy, J. Sadik, S. Dutta i dr., „Occasionally Secure: A Comparative Analysis of Code Generation Assistants,” *arXiv.org*, 2024.

[33] OpenAI, „GPT-4 Technical Report,” OpenAI, teh. izv., 2023.

[34] T. B. Brown, B. Mann, N. Ryder i et al., „Language Models are Few-Shot Learners,” *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[35] Anthropic, *Introducing Claude*, https://www.anthropic.com/index/introducing-claude, 2023.

[36] H. Touvron, T. Lavril, G. Izacard i et al., „LLaMA: Open and Efficient Foundation Language Models,” *arXiv preprint arXiv:2302.13971*, 2023.

Popis slika

1. Model procesa znanosti o oblikovanju (DSRM), prilagođeno prema [1]. 4

2. Primjer promjena nastalih u commitu d626b68 9
3. Primjer promjena nastalih u commitu d626b68 10
4. Prvi Git commit je sadržavao sam Git 13
5. Osnovni tijek rada s Gitom. 14
6. Torvaldsova commit poruka objašnjava motivaciju opisivanja praksi pisanja dobrih commit poruka [16] 15

7. Arhitektura programskog rješenja 37
8. Izgled VS Code proširenja 38
9. Konfiguracija proširenja 39
10. Pokretanje analize commitova 39
11. Jednostavan pogled na analizu 40
12. Jednostavan pogled nakon klika na savjet LLM-a 40
13. Slučaj kada commit poruke u prosjeku već slijede većinu ili sve dobre prakse . . 41
14. Detaljan pogled na analizu 41
15. Rezultati analize unutar bočnog prozora u proširenju 42
16. Detaljniji pogled na rezultate analize 42
17. Prikaz strukture klijentskog dijela VS Code proširenja 46
18. Prikaz strukture poslužiteljskog dijela 46
19. Cjelokupni radni tok analize commitova. 50

20. Pitanje 1: Koja je Vaša pozicija? 54
21. Pitanje 2: Koliko vam je alat bio jasan i lagan za korištenje? 54
22. Pitanje 3: Biste li koristili ovakav alat u svojoj nastavi/projektima? 55

23. Pitanje 4: Koliko bi vam alat olakšao praćenje i evaluaciju vlastitog (ako ste student), odnosno studentskog (ako ste nastavnik) rada? 56
24. Pitanje 5: Mislite li da bi alat pomogao studentima da poboljšaju svoje vještine dokumentiranja i pisanja commit poruka? 56

Popis tablica

1. Usporedba preporuka za commit poruke između različitih izvora 19

2. Rezultati studija za karakteristike različitih velikih jezičnih modela 28
3. Procjena otvorenosti i dostupnosti API-ja ispitanih modela 29
4. Procjena brzine i jednostavnosti integracije modela u softverske alate 29

5. Funkcionalni zahtjevi alata 34
6. Korištene tehnologije 36

7. Pitanja korištena u anketi fokus grupe 53

Popis isječaka koda

1. SCCSID primjer preuzeto 10
2. Primjer korištenja SCCS sustava preuzeto 11
3. Funkcija `activate` iz datoteke `extension.ts` 44
4. Postavke konfiguracije proširenja iz datoteke `settings.ts` 45
5. Shema odgovora za rezultate analize iz datoteke `gemini.ts` 48

Sažetak

Imena autora: Joshua Lee Fletcher, Noa Midžić
Naslov rada: Razvoj alata podržanog umjetnom inteligencijom za analizu kvalitete opisa verzija programskog kôda kao pomoć u učenju i poučavanju složenog koncepta verzioniranja pri razvoju programskih proizvoda

Istraživački rad analizira problem neadekvatnih vještina dokumentiranja softverskih projekata prvenstveno kroz Git commit poruke, koji je prepoznat kao značajna prepreka kvalitetnom razvoju programskih proizvoda u industriji te u obrazovanju u programskom inženjerstvu. Polazište rada je metodologija znanosti o oblikovanju, koja pruža znanstveni okvir za istraživanje i razvoj programskog rješenja. Unutar tog okvira definiraju se kriteriji dobre prakse u pisanju commit poruka na temelju literature i stručnih izvora. Kroz sustavni pregled literature uspoređuju se veliki jezični modeli te se procjenjuje njihova prikladnost za provjeru ovih kriterija. Na temelju tih spoznaja oblikovan je i razvijen edukativni softverski alat integriran u razvojno okruženje, koji studentima pruža analizu i povratnu informaciju o kvaliteti njihovih commit poruka. U obrazovnom kontekstu alat služi kao didaktičko pomagalo koje studentima omogućuje refleksiju o vlastitim praksama dokumentiranja, dok nastavnicima pruža podršku u poučavanju složenih koncepata verzioniranja kôda. Validacija i evaluacija provedene putem fokus grupa sa studentima, nastavnicima i stručnjacima iz industrije pokazale su znanstvenu i praktičnu vrijednost alata te njegov potencijal za unapređenje obrazovanja u području programskog inženjerstva.

Ključne riječi: verzioniranje kôda, Git commit poruke, veliki jezični modeli, edukativni alati, znanost o oblikovanju

Summary

Authors: Joshua Lee Fletcher, Noa Midžić
Title: AI-Supported Tool for Analyzing the Quality of Software Code Version Descriptions as an Aid in Learning and Teaching the Complex Concept of Version Control in Software Development

This research analyzes the problem of inadequate documentation skills in software projects, primarily through Git commit messages, identified as a significant obstacle to high-quality software development both in industry and in software engineering education. The study is grounded in the design science research methodology (DSRM), which provides a scientific framework for the investigation and development of software solutions. Within this framework, best-practice criteria for writing commit messages are defined based on literature and expert sources. Through a systematic literature review, large language models (LLMs) are compared and their suitability for verifying these criteria is evaluated. Based on these insights, an educational software tool was designed and developed, integrated into a development environment, providing students with analysis and feedback on the quality of their commit messages. In the educational context, the tool functions as a didactic aid, enabling students to reflect on their documentation practices, while offering instructors support in teaching the complex concepts of code versioning. Validation and evaluation carried out through focus groups with students, educators, and industry experts demonstrated both the scientific and practical value of the tool and highlighted its potential to enhance software engineering education.

Keywords: code versioning, Git commit messages, large language models (LLM), educational tools, design science