Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Robert Samardžija

Uporaba velikih jezikovnih modelov za generiranje dokumentacije iz izvorne kode

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Zoran Bosnić
Somentor: mag. Dragoslav Radin

Ljubljana, 2024

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Robert Samardžija
Naslov: Uporaba velikih jezikovnih modelov za generiranje dokumentacije iz izvorne kode
Vrsta naloge: Diplomska naloga na univerzitetnem programu prve stopnje Računalništvo in informatika
Mentor: prof. dr. Zoran Bosnić
Somentor: mag. Dragoslav Radin

Opis:
Kandidat naj v diplomski nalogi razišče možnost uporabe velikih jezikovnih modelov za potrebe generiranja dokumentacije iz izvorne kode. Zasnuje naj pristop generatorja in z izbranimi metrikami naj evalvira uspešnost delovanja.

Title: Use of large language models for generating source code documentation

Description:
In the thesis, the candidate shall explore the possibility of using large language models for the purpose of generating documentation from source code. The candidate should design the documentation generator and use selected metrics to evaluate its performance.

Zahvaljujem se mentorju prof. dr. Zoranu Bosniću za strokovno usmeritev pri izdelavi diplomske naloge, somentorju mag. Dragoslavu Radinu in ostalim zaposlenim na DevRev, ki so me podpirali in vodili pri razvoju diplomske naloge, ter vsem bližnjim, ki so mi ob pisanju diplomske naloge stali ob strani.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Pregled področja **3**

2.1 Obstoječe rešitve 3

2.2 Generativna umetna inteligenca 4

2.3 Jezikovni modeli 5

2.4 Proces ETL 7

3 Podatki in zasnova rešitve **9**

3.1 Vir podatkov 9

3.2 Arhitektura 13

3.3 Upravljalec AWS 14

3.4 Lokalni upravljalec receptov 15

3.5 Generator dokumentacije 15

3.6 Uporabniški vmesnik 17

4 Implementacija rešitve **19**

4.1 Prenos receptov iz AWS S3 19

4.2 Lokalni upravljalec receptov 21

4.3 Generiranje dokumentacije 23

5 Rezultati in evalvacija **33**
5.1 Izbira kriterijev vrednotenja 33
5.2 Vrednotenje rezultatov 34

6 Zaključek **45**

Literatura **47**

Seznam uporabljenih kratic

kratica	angleško	slovensko
GenAI	generative artificial intelligence	generativna umetna inteligenca
AI	artificial intelligence	umetna inteligenca
LLM	large language model	veliki jezikovni model
GPT	generative pre-trained transformer	generativni prednaučeni transformer
HTML	hypertext markup language	jezik za označevanje nadbesedila
PDF	portable document format	prenosni dokumentni format
API	application programming interface	aplikacijski programski vmesnik
JSON	JavaScript object notation	JavaScript objektna notacija
ETL	extract, transform, load	ekstrahiraj, transformiraj, naloži
SDK	software development kit	paket za razvoj programske opreme
PLM	product lifecycle management	upravljanje življenjskega cikla proizvoda
SPOF	single point of failure	kritična točka odpovedi

Povzetek

Naslov: Uporaba velikih jezikovnih modelov za generiranje dokumentacije iz izvorne kode

Področje generativne umetne inteligence je v letu 2022 v tehnološko stroko in tudi v ostale stroke prineslo revolucijo. Razcvet na področju osnovnih modelov je omogočil ustvarjanje realističnih in kompleksnih vsebin različnih vrst ter odprl vrata novim pristopom na področjih ustvarjalnosti, strojnega prevajanja in odločanja. V diplomski nalogi raziščemo uporabo velikih jezikovnih modelov za generiranje dokumentacije iz izvorne kode. Ogledamo si pristope inženiringa poizvedb, zasnujemo in razvijemo prototip generatorja ter ocenimo zmogljivost velikih jezikovnih modelov na zastavljeni nalogi. Izpostavimo težavo narave delovanja jezikovnih modelov, ki lahko pri različnih izvajanjih ustvarijo nezaželene razlike v rezultatih, in problem prilagajanja naše metode na delovanje specifičnega jezikovnega modela. Delo zaključimo z ugotovitvijo, da implementacija naše metode zadovoljuje potrebe podjetja DevRev in predstavlja alternativo obstoječim generatorjem dokumentacije, ki ne uporabljajo jezikovnih modelov. Predstavimo možne izboljšave, ki vključujejo uporabo jezikovnih modelov iz različnih družin in integracijo prototipa v storitev Airdrop platforme DevRev.

Ključne besede: veliki jezikovni modeli, generatorji dokumentacije, inženiring poizvedb, GPT.

Abstract

Title: Use of large language models for generating source code documentation

The field of generative artificial intelligence brought about a revolution in technology and other disciplines in the year 2022. The development and incredible success of foundation models enabled the creation of realistic and complex content of various kinds and introduced new approaches in creativity, machine translation and decision-making. In our work, we explore the use of large language models for generating source code documentation. We examine prompt engineering approaches, design and develop a prototype of the generator and evaluate the performance of large language models on the set task. We highlight the challenging nature of language models, whose output can undesirably differ between runs, and the problem of tuning our method to one specific language model. The work concludes with the finding that the implementation of our method satisfies the needs of DevRev and represents an alternative to existing documentation generators that do not use language models. We also present possible improvements that include the use of language models from different families and the integration of our prototype into DevRev's Airdrop service.

Keywords: large language models, documentation generators, prompt engineering, GPT.

Poglavje 1

Uvod

Hiter razvoj platform za podporo procesu upravljanja življenjskega cikla proizvoda (angl. product lifecycle management, PLM) je povečal uspešnost in učinkovitost delovanja podjetij. Platform, ki podpirajo proces PLM, je veliko: od platform za sodelovanje v razvojnem procesu, kot sta GitHub in Bitbucket, platform za upravljanje projektov, kot je Jira, platform za podporo uporabnikom, kot je Zendesk, do platform za izvajanje strategij rasti in prodaje, kot je Salesforce. Kljub široki izbiri različnih platform pa je integracija med temi platformami slaba.

Vizija podjetja DevRev je razviti enovito platformo, ki povezuje ali pa nadomesti predhodno omenjene platforme. Ena izmed glavnih storitev platforme DevRev je storitev Airdrop, ki omogoča enosmerno uvažanje podatkov iz zunanjih sistemov, in tudi dvosmerno sinhronizacijo podatkov med platformo in zunanjimi sistemi. Ključen del delovanja te storitve je pravilna transformacija podatkov med zunanjim sistemom in platformo DevRev.

Airdrop trenutno omogoča uvoz oz. sinhronizacijo podatkov iz platform Jira, GitHub, Linear, Zendesk, Salesforce, HubSpot, ServiceNow, Confluence in Rocketlane. Transformacije med podatkovnimi modeli zunanjih sistemov in podatkovnim modelom platforme DevRev omogočajo transformacijske funkcije, ki pa imajo težko berljivo dokumentacijo in so tudi same težko berljive.

Pred letom 2020 orodja generativne umetne inteligence niso imela velikega pomena in vpliva na tehnološko industrijo. Veliki jezikovni modeli so bili uporabljeni predvsem za zaznavo neželene elektronske pošte, prevajanje, osnovno odgovarjanje na vprašanja in samodejno dopolnjevanje kode v obsegu 1 vrstice. Med letoma 2020 in 2022 je področje ogromno napredovalo. Veliki jezikovni modeli so lahko spisali osnovna enostavna besedila, modeli za dopolnjevanje kode pa so začeli predlagati več kot 1 vrstico. Leta 2022 se je zgodil razcvet – jezikovni modeli so lahko spisali daljša besedila, pri tem pa so se zavedali konteksta besedila. Dopolnjevanje kode se je izboljšalo z uporabo konteksta, natančnost se je povečala. Pojavili so se generativni modeli za slike, logotipe, fotografije in videoposnetke. Glavni igralec na tem področju je postalo ameriško podjetje OpenAI. Z razpoložljivostjo orodja ChatGPT[1] širši javnosti 30. novembra 2022 se je uporabnost področja razširila tudi na druge panoge, ne samo na tehnološko industrijo. Generativna umetna inteligenca je postala eno izmed najbolj zanimivih, popularnih in znanih področij umetne inteligence. Raziskovalci so se začeli osredotočati na izboljšave obstoječih modelov in zasnovo novih, boljših, zmogljivejših modelov, ki bodo primerni za različne načine uporabe.

V nalogi bomo raziskali nov pristop generiranja dokumentacije iz izvorne kode, bolj specifično za dokumentacijo transformacijskih funkcij. Ker tradicionalne metode z namenskimi generatorji dokumentacije zaradi specifičnosti naše naloge (predvsem zaradi uporabe programskega jezika jq) ne pridejo v poštev, bomo razvili in ovrednotili prototip aplikacije, ki bo za generiranje dokumentacije uporabljal velike jezikovne modele. V drugem poglavju bomo predstavili trenutno stanje na področju in predstavili teoretične temelje, na osnovi katerih bomo zgradili našo rešitev. V tretjem poglavju bomo predstavili podatkovni vir in zasnovo prototipa naše rešitve. V četrtem poglavju bomo glede na zasnovo prototipa opisali proces razvoja prototipa in predstavili najpomembnejše tehnike uporabe velikih jezikovnih modelov, v petem poglavju pa bomo pridobljene rezultate ovrednotili.

[1] `https://chatgpt.com/`

Poglavje 2

Pregled področja

V tem poglavju bomo pregledali obstoječe rešitve za generiranje dokumentacije iz izvorne kode in na kratko predstavili generativno umetno inteligenco, jezikovne modele ter razvoj modelov GPT. Nazadnje bomo še predstavili proces ETL, področje, na katerem bomo aplicirali našo rešitev.

2.1 Obstoječe rešitve

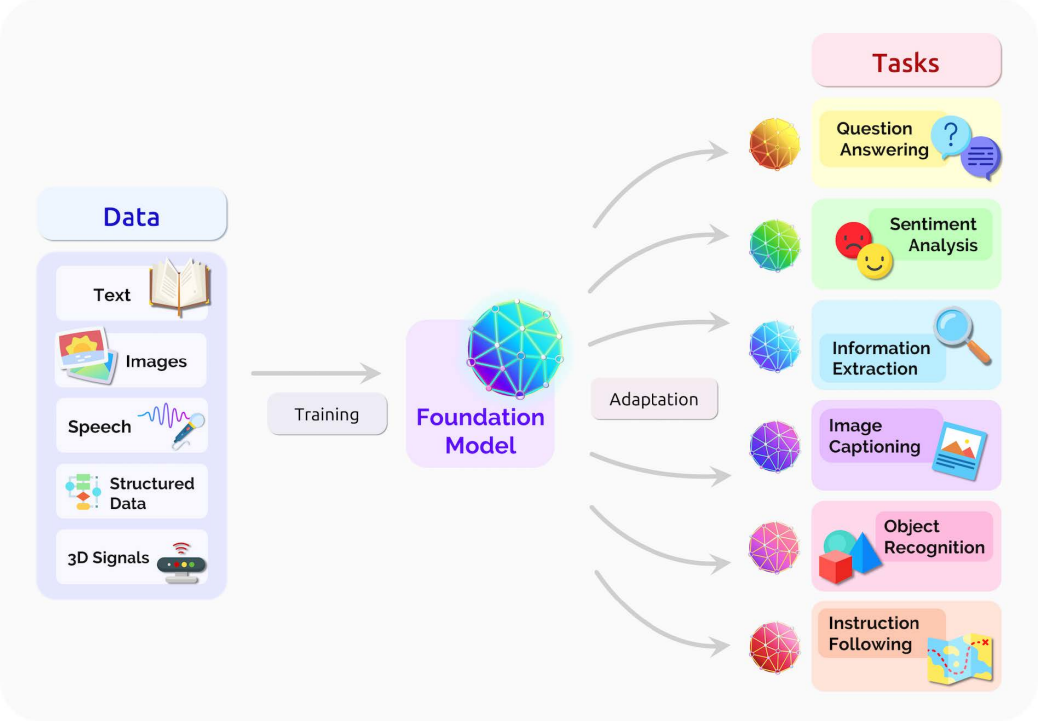
Za generiranje dokumentacije iz izvorne kode danes primarno uporabljamo 3 pristope.

Najbolj razširjen pristop je uporaba orodij kot so Javadoc, Doxygen in JSDoc, ki iz kombinacije sintakse, komentarjev in posebnih oznak (angl. tag) dokumentacijo zgenerirajo v formatih HTML, PDF ali Markdown. Ta vrsta dokumentacije je pomembna pri modernem razvoju programske opreme, ki pogosto uporablja veliko zunanjih knjižnic in API-jev [4] in je namenjena predvsem razvijalcem.

Uporabniške priročnike in navodila za uporabo programske opreme v večini podjetij piše posebej usposobljeno osebje, t.i. pisci tehnične dokumentacije (angl. technical writer) [8]. Priročniki so pomembni za končne uporabnike aplikacij, ki se na tehnične podrobnosti delovanja aplikacij ne spoznajo v enaki meri kot razvijalci.

Z razširjenostjo in integracijo velikih jezikovnih modelov (več v razdelku 2.3.1) v aplikacije se je pojavil nov pristop: generiranje dokumentacije z uporabo orodij AI, kot so ChatGPT ali GitHub Copilot. Ta pristop postaja zanimiv zaradi obsežnosti korpusa velikih jezikovnih modelov, ki vsebuje tako tehnične vsebine, kot so izvorna koda, in splošne vsebine, kot so navodila za uporabo brez tehničnega žargona. Najnaprednejši veliki jezikovni modeli (angl. large language models, LLM) lahko zato ustvarijo tehnično ali splošno namensko dokumentacijo.

2.2 Generativna umetna inteligenca



Slika 2.1: Osnovni modeli združujejo učne podatke različnih modalnosti in omogočajo natančno nastavljanje [2].

Generativna umetna inteligenca (angl. generative artificial intelligence, kratica GenAI) so sistemi umetne inteligence, ki ustvarjajo nove vsebine, kot so besedilo, slike, videoposnetki in zvok [15]. Poganjajo jih osnovni modeli (angl. foundation models) - veliki modeli strojnega učenja, naučeni na ogromnih podatkovnih množicah različnih modalnosti. Primarno so to besedila, slike in govor, ki so ponazorjeni na levi strani slike 2.1 pod oznako *Data*, kjer so osnovni modeli prikazani v sredini pod oznako *Foundation Model*. Osnovne modele lahko prilagodimo za obsežen nabor končnih nalog (nekaj primerov je prikazanih na sliki 2.1 pod oznako *Tasks*), med katere spadajo povzemanje in klasifikacija besedil, odgovarjanje na vprašanja, ekstrakcija informacij, pogovorni modeli, generiranje slik itd. Omogočajo nam tudi natančno nastavljanje (angl. fine-tuning), t.j. prilagajanje za specifične primere uporabe, pri čemer pa v primerjavi z velikostjo podatkovne množice za učenje osnovnega modela potrebujemo zelo majhno število vzorčnih podatkov.

2.3 Jezikovni modeli

Jezikovni modeli so vrsta generativne umetne inteligence, ki napoveduje in generira besedilo. V osnovi so to modeli strojnega učenja, ki žetonu (angl. token) ali zaporedju žetonov v daljšem zaporedju ostalih žetonov določajo verjetnost pojavitve [6].

Žetoni so osnovne enote v jezikovnih modelih. To so lahko posamezni znaki, deli besed, besede ali fraze. Zaporedja žetonov lahko predstavljajo nekaj besed, eno poved, zaporedje povedi, odstavek ali pa celotno besedilo. Jezikovni modeli so torej zmožni izračunati verjetnosti pojavitve ne samo za posamezne besede, ampak tudi za daljše sekvence besedila.

2.3.1 Veliki jezikovni modeli

Eksplozija velikih jezikovnih modelov (angl. large language model, LLM) se je začela leta 2017 z izidom članka “Attention Is All You Need” [14]. V članku so Googlovi raziskovalci predstavili novo arhitekturo za modele glo-

bokega učenja imenovano Transformer, ki je postala osnova za vse trenutno najnaprednejše velike jezikovne modele.

2.3.2 GPT

GPT (angl. kratica za generative pre-trained transformer) je vrsta velikih jezikovnih modelov, ki jo je leta 2018 predstavilo podjetje OpenAI [11]. Taki modeli temeljijo na transformerski arhitekturi, učenje pa poteka v dveh fazah:

1. nenadzorovano predhodno učenje, kjer se model uči na velikem besedilnem korpusu,

2. nadzorovano natančno nastavljanje, kjer se model prilagodi za specifično nalogo. Ta faza ni obvezna.

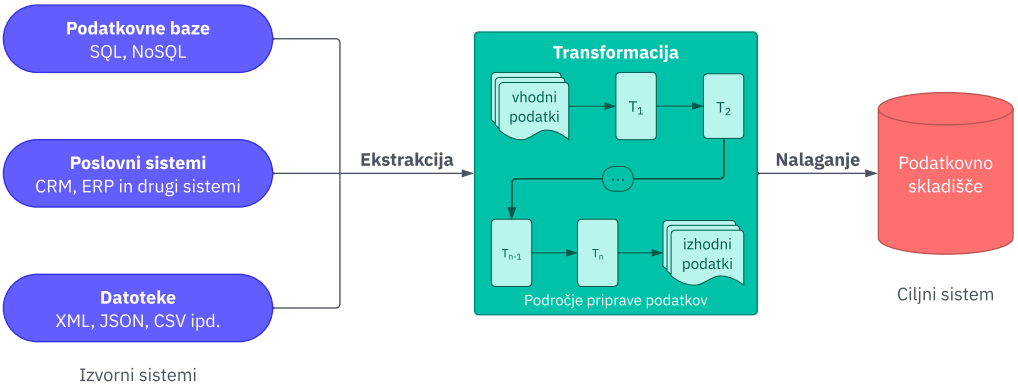
Prvotnemu modelu iz leta 2018 je sledil model GPT-2, s katerim so raziskovalci pri OpenAI pokazali, da tudi splošni modeli premorejo širok nabor zmožnosti [12]. Učna množica podatkov za GPT-2 ni vsebovala podatkov iz specifičnih domen, temveč 8 milijonov spletnih strani ter 7 tisoč neizdanih knjig raznih zvrsti. Kljub temu je GPT-2 na specifičnih domenah (npr. Wikipedija ali novice) dosegel boljše rezultate kot takrat najnaprednejši domensko-specifični modeli. Še zmeraj pa model po zmogljivosti ni dosegel specializiranih modelov za naloge, kot so bralno razumevanje, prevajanje ali povzemanje.

Modeli GPT-3, njegova izboljšava GPT-3.5 in GPT-4 so naslednje iteracije v seriji modelov "GPT-n" podjetja OpenAI. Raziskovalci podjetja v članku "Language Models are Few-Shot Learners" pokažejo, da se zmogljivost jezikovnih modelov s povečanjem velikosti učne množice in števila parametrov znatno izboljša, med drugim tudi na bolj specifičnih nalogah [3, 10]. Modeli od vključno GPT-3 naprej potrjujejo to trditev in so pokazali izjemne zmogljivosti tudi na nalogah, kot so prevajanje, odgovarjanje na vprašanja, sklepanje ipd. Novejši modeli, kot je GPT-4, premorejo tudi večmodalne sposobnosti, kot je sprejemanje slik na vhodu v model.

Kljub temu pa noben model ni popoln in ima svoje pomanjkljivosti in omejitve. Učne množice za modele namreč obsegajo velik del interneta, ki lahko vsebuje neresnične, neprimerne in protizakonite vsebine [5]. To težavo so v podjetju OpenAI rešili z uvedbo varnostnih mehanizmov, ki preprečujejo generiranje takih vsebin. Kljub implementirani rešitvi pa obstajajo tehnike, ki omogočajo, da varnostne mehanizme zaobidemo, imenovane "jailbreaking". Med druge težave zaradi narave modelov pa spadajo tudi halucinacije, ponavljanje besedila in zmedenost pri dolgih kontekstih [9, 13].

2.4 Proces ETL

Proces ETL (angl. kratica za Extract, Transform, Load), idejno in po stopnjah predstavljen na sliki 2.2, je tristopenjski proces za podatkovne integracije, kjer podatke iz izvornega sistema, zajete v njegovem podatkovnem modelu, zajamemo, nato transformiramo v ciljni podatkovni model in na koncu naložimo v ciljni sistem [7]. Z razvojem podatkovnih baz, strojnega učenja in podatkovne analitike je ETL postal pomemben del poslovne inteligence sodobnih podjetij. Vse stopnje procesa so ponavadi popolnoma avtomatizirane v sklopu sistema ETL.



Slika 2.2: Diagram tipičnega procesa ETL

2.4.1 Extract: ekstrakcija podatkov

V koraku ekstrakcije se iz podatkovnega vira izvornega sistema (angl. source system) v področje priprave podatkov (angl. staging area) sistema ETL naložijo izvorni podatki, ki ponavadi izvirajo iz podatkovnih baz, različnih vrst poslovnih sistemov ali datotek (glej levo stran slike 2.2) [7]. Uspešna izvedba ekstrakcije zahteva obsežne predhodne raziskave podatkovnih modelov in shem izvornih sistemov, dobro načrtovanje obvladovanja sprememb v podatkih in robustno implementacijo same ekstrakcije.

2.4.2 Transform: transformacija podatkov

Po uspešni ekstrakciji sistem podatke počisti in pripravi v obliko, primerno za končni podatkovni vir v podatkovnem skladišču [7]. To je najpomembnejši in najtežji korak v celotnem procesu ETL. V tem koraku sistem filtrira in odstranjuje odvečne podatke, preverja vrsto in veljavnost podatkov, izloča dvojne podatke (deduplikacija), izvaja transformacije (izračune, pretvorbe), oblikuje podatke v zahtevano končno obliko in obvešča vse ostale komponente sistema o napakah. Omenjene transformacije podatkov smo na sliki 2.2 ponazorili kot zaporedje transformacij od T_1 do T_n, seveda pa lahko ta proces glede na obliko podatkov in zahteve poteka tudi vzporedno ali kot kombinacija zaporednih in vzporednih transformacij.

2.4.3 Load: nalaganje transformiranih podatkov

Ko sistem pridobi pripravljene transformirane podatke, jih s področja priprave podatkov naloži v podatkovno skladišče (na sliki 2.2 prikazan kot rdeči valj, kjer nalaganje prikazuje puščica v valj) [7]. Po končanem prvotnem uvozu podatkov ta del sistema skrbi za redne posodobitve že obstoječih podatkov in postopno nalaganje novih ter spremenjenih podatkov (angl. incremental loading). Posodobitve se izvajajo po določenem urniku ali so sprožene ročno, uvožene podatke pa lahko tudi izbrišemo.

Poglavje 3

Podatki in zasnova rešitve

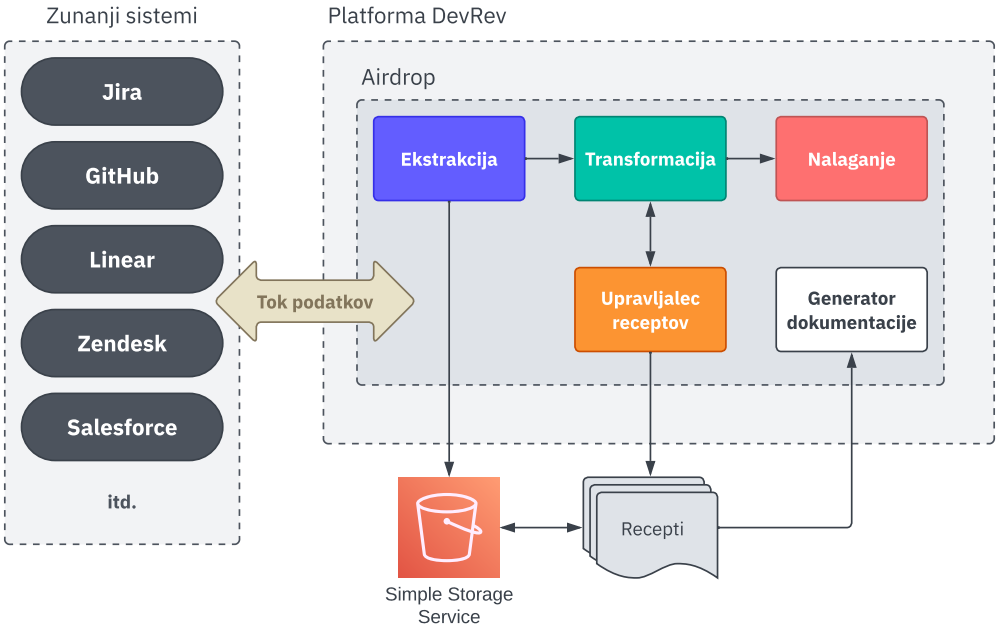
V tem poglavju bomo opisali, od kod pridobimo podatke za našo aplikacijo, in predstavili visokonivojski načrt oz. arhitekturo aplikacije.

3.1 Vir podatkov

Airdrop je storitev v okviru platforme DevRev, ki izvaja proces ETL. V fazi ekstrakcije iz zunanjih izvornih sistemov, kot so Jira, GitHub in Linear, naloži izvorne podatke v storitev za shranjevanje podatkov Amazon Simple Storage Service (tudi AWS S3 ali samo S3). Shemo delovanja storitve Airdrop v povezavi z zunanjimi sistemi smo prikazali na sliki 3.1. Zaradi poenostavitve v shemo nismo vključili ostalih notranjih komponent, ki sicer sodelujejo pri delovanju platforme, ampak so za namene naše naloge nepomembne.

S3 deluje na podlagi objektov, ki so shranjeni v vedrih (angl. bucket) [1]. Objekti predstavljajo osnovne enote in so sestavljeni iz ključa (imena), datoteke in metapodatkov, ki dodatno opisujejo objekt. Vedra so namenjena logični organizaciji objektov, pri čemer si vsi objekti znotraj posameznega vedra delijo dostopne pravice in nastavitve. Vsak objekt v S3 lahko enolično identificiramo s kombinacijo vedra in ključa objekta.

Ključni korak transformacije izvede posebna komponenta, imenovana **upravljalec receptov** (angl. recipe manager). Izvorni podatki lahko poleg polj, ki jih ponuja izvorni podatkovni model, vključujejo tudi polja po meri (angl.



Slika 3.1: Storitev Airdrop podatke eno- ali dvosmerno prenaša med platformo DevRev in zunanjimi sistemi. Pri procesu ima pomembno vlogo upravljalec receptov, ki odkriva recepte za transformacije med izvornim podatkovnim modelom in podatkovnim modelom platforme DevRev. Naša aplikacija, generator dokumentacije, deluje kot pomočnik za lažjo razlago teh receptov.

custom fields). Upravljalec receptov zato za vsako uvoženo enoto[1] izvede **odkrivanje recepta**. V določenih primerih se lahko tudi zgodi, da se več polj iz izvornega sistema transformira v eno polje na platformi DevRev ali obratno. Komponenta včasih ni zmožna odkriti vseh preslikav oz. transformacij, zato je v določenih primerih za dokončanje recepta potreben uporabniški vnos. Rezultat odkrivanja recepta je recept v obliki JSON datoteke, ki se shrani v S3. Podatki in recept so shranjeni v vnaprej določeni in enotni podatkovni shemi: `<organizacija>/<zunanji sistem>/<uporabnik>/<enota>`.

[1]Enota je zaključena celota, znotraj katere za vse primerke objektov enakega tipa veljajo enake zakonitosti. Za Jiro je to projekt, za GitHub pa repozitorij.

Za namene naše naloge podatki niso pomembni, zato se bomo osredotočili na recept.

3.1.1 Recept

Recept je datoteka JSON, ki vsebuje metapodatke, transformacije v obliki filtrov jq in module jq, ki se uporabljajo v transformacijah. Za lažjo predstavo smo drevesno strukturo recepta v obliki diagrama ponazorili na sliki 3.2.

JSON in jq

JSON je priljubljen odprt podatkovni format, namenjen ljudem razumljivi izmenjavi podatkov. Temelji na parih ključev in vrednosti, podpira različne podatkovne tipe in je izjemno priljubljen za komunikacijo med spletnimi strežniki in odjemalci.

jq je prilagodljiv in visokozmogljiv funkcijski programski jezik za obdelavo dokumentov JSON[2]. Vsak program v jq se imenuje filter. Uporablja se za poizvedovanje v strukturah JSON in omogoča filtriranje, transformacije in ekstrakcijo podatkov. Ponuja veliko knjižnico vgrajenih filtrov, pogosto pa se uporablja v obdelavi podatkov, skriptah in avtomatizaciji procesov. Programi so sestavljeni iz zaporedja filtrov in cevovodov (prepoznavnih po simbolu `|`), ki izhod enega filtra vodijo na vhod drugega filtra.

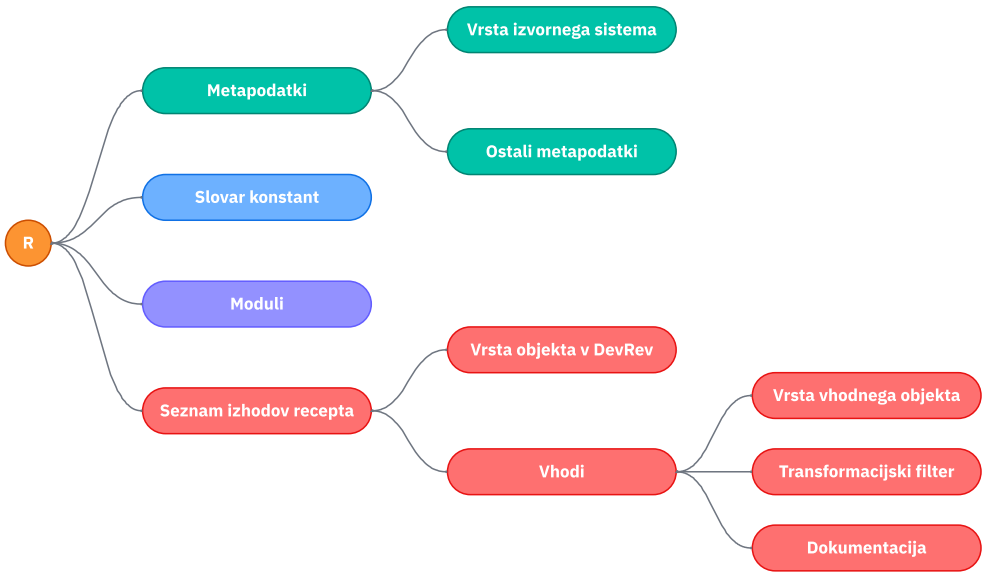
Struktura recepta

Recept se začne s ključi za metapodatke (od katerih je za nas pomembno le ime oz. vrsta izvornega sistema). Metapodatkom sledi **slovar konstant**, ki se v nadaljevanju uporabljajo v transformacijskih filtrih jq. Sledijo **definicije modulov jq** z vnaprej določenimi filtri, ki predstavljajo pogoste operacije, kot so odstranjevanje praznih znakov iz nizov, pretvarjanje sekund v nanosekunde, oblikovanje datumov itd.

[2]Owen Ou, Wadman Mattias, David Tolnay, Emanuele Torre, *jq Manual.* URL: `https://jqlang.github.io/jq/manual/`.

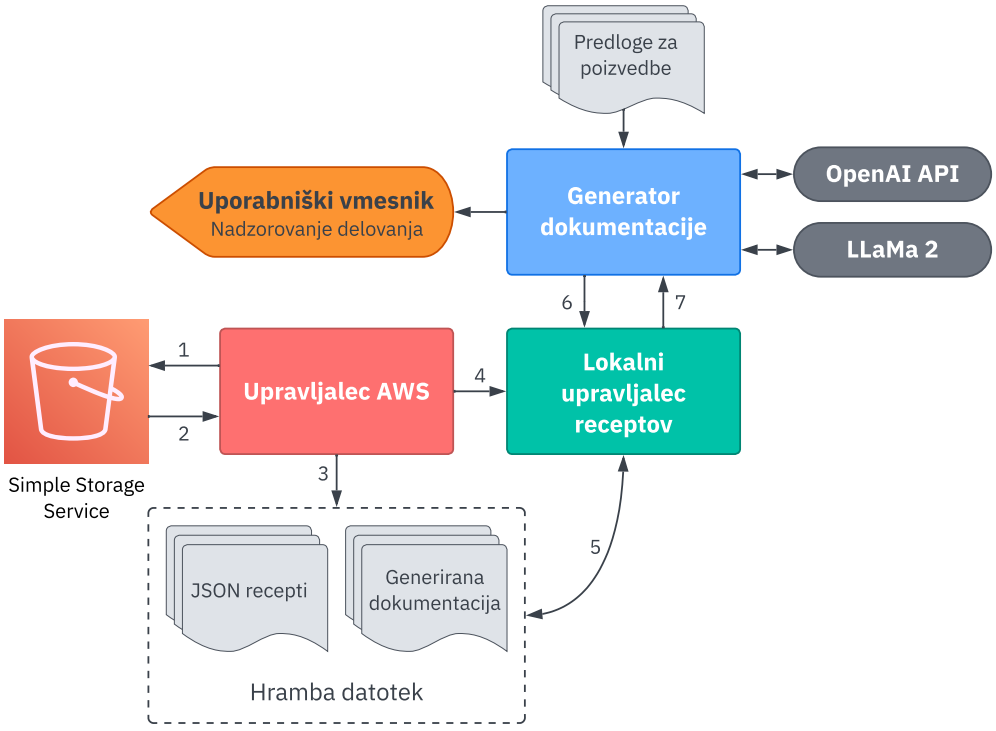
Največji in hkrati najpomembnejši del recepta pa je **seznam izhodov recepta**, kjer je vsak element seznama sestavljen iz:

- vrste objekta na platformi DevRev, v katerega se transformirajo objekti iz izvornega sistema,
- seznama vhodov, kjer je vsak vhod sestavljen iz:
  - vrste vhodnega objekta v izvornem sistemu,
  - transformacijskega filtra v programskem jeziku jq, ki lahko uporablja konstante in module iz predhodno omenjenih delov recepta,
  - dokumentacije, ki jo zgenerira upravljalec receptov pri odkrivanju preslikav.



Slika 3.2: Drevesna struktura recepta, kjer oznaka R predstavlja izhodišče recepta (prvi nivo dokumenta JSON). Slovar konstant in definicije modulov so pomembne za delovanje transformacijskih funkcij, ki so vsebovane v posameznem elementu seznama izhodov recepta.

3.2 Arhitektura



Slika 3.3: Aplikacija pridobi recepte iz AWS S3, pri čemer komunikacijo s S3 upravlja ločen modul. Lokalni upravljalec receptov deluje kot posrednik med hrambo datotek in generatorjem dokumentacije, ki je naš fokus. Generator dokumentacije sestavlja poizvedbe, z njimi poizveduje na velikih jezikovnih modelih, in vrne generirane odgovore, t.j. dokumentacijo. Uporabniški vmesnik deluje kot pripomoček za nadzor delovanja generatorja.

Aplikacijo smo zasnovali kot samostojno nanostoritev, ki dopolnjuje funkcionalnost sistema Airdrop, njeno arhitekturo pa smo orisali na sliki 3.3. Pri tem smo se odločili generiranje dokumentacije ponuditi na zahtevo (angl. on-demand), ker predvidevamo, da določenih uporabnikov platforme DevRev ne bo zanimala generirana dokumentacija. Upravljalec receptov znotraj Airdropa sicer že ponuja generirano dokumentacijo, ki pa ni primerna za uporabnike in je namenjena razvijalcem.

Aplikacija za delovanje potrebuje dostop do zunanjih storitev, kot sta AWS S3 in OpenAI API, zato smo jo arhitekturno zasnovali kot 3 manjše module, ki sledijo konceptu ločevanja nalog (angl. separation of concerns): **upravljalec AWS**, **lokalni upravljalec receptov** in **generator dokumentacije**. Pretok podatkov v aplikaciji poteka od upravljalca AWS, preko lokalnega upravljalca receptov do generatorja dokumentacije, ki upravljalcu vrne generirano dokumentacijo. Z lokalnimi podatki (t.j. prenesenimi recepti in generirano dokumentacijo) upravlja lokalni upravljalec receptov, ki ima vlogo edinega vira resnice (angl. single source of truth) za lokalne podatke.

V nadaljevanju bomo podrobneje predstavili module in uporabniški vmesnik, ki je namenjen nadzorovanju delovanja aplikacije.

3.3 Upravljalec AWS

Upravljalec AWS je modul, ki skrbi za komunikacijo med aplikacijo in AWS S3, ki je naš vir podatkov. V AWS S3 se shranjujejo recepti, vključno z izvornimi podatki, ki pa za nas niso pomembni.

Upravljalec iz S3 prenaša datoteke z recepti in jih shrani v vnaprej določeno lokacijo v hrambi datotek. Po končanem prenosu upravljalec S3 lokalnega upravljalca receptov obvesti o novo prenesenih receptih. Modul torej služi kot pripomoček za prenos podatkov, s čimer se znebimo ročnega prenašanja z orodji, kot je AWS CLI[3].

[3] `https://aws.amazon.com/cli/`

3.4 Lokalni upravljalec receptov

Lokalni upravljalec receptov upravlja lokalne recepte in že generirano dokumentacijo, ki je shranjena v hrambi datotek.

Od upravljalca AWS sprejema obvestila o novo naloženih receptih. Ko sprejme obvestilo, preveri stanje v hrambi datotek – prebere nove recepte in posodobi svoje interno stanje. Interno stanje je sestavljeno iz zadnjega časa posodobitve in seznama poti do receptov ter pripadajoče že generirane dokumentacije. Ponuja tudi možnost izbrisa lokalnih receptov (in dokumentacije), lahko pa tudi posodobi svoje stanje po ročnih spremembah v hrambi datotek.

Druga naloga lokalnega upravljalca receptov pa je zagotavljanje receptov generatorju dokumentacije. Po končanem generiranju dokumentacije generator lokalnemu upravljalcu receptov vrne dokumentacijo v naravnem jeziku. Lokalni upravljalec nato dokumentacijo shrani v hrambo datotek kot besedilno datoteko, ki se uporabi za prikaz uporabnikom.

3.5 Generator dokumentacije

Generator dokumentacije je primarni cilj naše naloge in glavna komponenta aplikacije, zato mu bomo posvetili največ pozornosti.

Generatorjev vir podatkov so lokalni recepti, ki jih pridobi od lokalnega upravljalca receptov. Če bi recepte v njihovi izvorni obliki brez kakršnegakoli konteksta posredovali neposredno v LLM-je, bi generirani odgovori variirali, in s tem ne bi dosegli svojega cilja. Tipično bi odgovori pri takem pristopu bili videti kot visokonivojski opis strukture recepta: “Videti je, da ste posredovali strukturo JSON, ki vsebuje informacije, povezane z receptom za preoblikovanje podatkov med dvema izvornima sistemoma.” Primer takega odgovora v angleščini smo predstavili na sliki 3.4, kjer opazimo predhodno omenjen visokonivojski opis, ki pa za namene naše naloge ni zadovoljiv. Zato smo poleg vira podatkov generatorju dokumentacije zagotovili tudi predloge za poizvedbe (angl. prompt templates). Predloge za poizvedbe so del širšega

področja inženiringa poizvedb (angl. prompt engineering) in nam pomagajo pri lažjem sestavljanju poizvedb. S predlogami za poizvedbe in podatki iz recepta ustvarimo strukturirana navodila za LLM, ki v primerjavi z poizvedbo brez konteksta usmerjajo LLM k boljšim odgovorom in posledično boljšim rezultatom.

```
It looks like you've shared a JSON document containing
information related to a recipe, specifically for transforming
Jira data.  The recipe seems to define transformations using an
Advanced Document Format (ADF) for comments.
If you have specific questions or tasks related to this recipe
or if you need assistance with anything specific, feel free to
provide more details!
```

Slika 3.4: Tipičen visokonivojski opis recepta za Jiro, ki ga generira GPT-3.5, če mu ne podamo dodatnega konteksta (navodil).

Predloge za poizvedbe so v osnovi besedilni dokumenti s fiksnimi navodili in rezerviranimi mesti za dinamične vhodne podatke (angl. placeholder). Rezervirana mesta za dinamične vhodne podatke se pri konstruiranju poizvedbe (angl. prompt) zamenjajo z dejanskimi podatki, v našem primeru to pomeni s podatki iz recepta. Ker pa lahko poizvedbe konstruiramo le iz že pripravljenih podatkov, smo pred korakom konstruiranja uvedli še predhodni korak urejanja podatkov. V tem koraku filtriramo podatke, ki jih ne želimo posredovati v poizvedbo in po potrebi uredimo njihovo obliko.

Za delovanje generatorja pa potrebujemo integracijo z velikimi jezikovnimi modeli, v našem primeru so to zunanje storitve, ki ponujajo dostop do LLM-jev – OpenAI API, Llama API. Seveda bi lahko generator povezali tudi z lokalnimi postavitvami jezikovnih modelov kot so LLaMa 2. Integracija z LLM-ji je ključen del generatorja in v našem primeru tudi **kritična točka odpovedi** (angl. single point of failure, SPOF) v aplikaciji. Če jezikovni model preneha delovati, dokumentacije ne bomo mogli generirati in posledično ne bo na voljo uporabnikom.

3.6 Uporabniški vmesnik

Za lažje nadzorovanje in razhroščevanje (angl. debugging) generatorja dokumentacije smo se odločili v aplikacijo vključiti tudi uporabniški vmesnik, ki bo omogočal pregled nad delovanjem in vizualno primerjavo med različnimi poizvedbami, rezultati in modeli. Pri delovanju generatorja pa nas sicer zanimajo naslednji parametri:

- vsebina poizvedb,
- rezultati (generirana dokumentacija),
- poraba časa pri generiranju in
- denarni stroški generiranja.

Poglavje 4

Implementacija rešitve

V tem poglavju bomo glede na zasnovo rešitve iz prejšnjega poglavja predstavili izzive, težave in potek implementacije naše rešitve. Za implementacijo smo zaradi enostavnosti in razširjenosti zunanjih knjižnic, ki nam bodo pomagale pri implementaciji, izbrali programski jezik Python.

4.1 Prenos receptov iz AWS S3

Prvi korak pri implementaciji naše rešitve je vzpostavitev povezave s storitvijo AWS S3. Pri tem smo uporabili paket za razvoj programske opreme (angl. software development kit, SDK) Boto3 oz. AWS SDK for Python[1], ki ga vzdržuje in objavlja AWS.

Za uspešno povezavo na S3 mora Boto3 imeti dostop do naših poverilnic (angl. credentials). Poverilnice lahko pridobimo preko prijavnega portala AWS, nastaviti pa moramo sledeče okoljske spremenljivke [1]:

- `AWS_ACCESS_KEY_ID`: določa dostopni ključ za uporabnikov račun,
- `AWS_SECRET_ACCESS_KEY`: določa tajni ključ, povezan z dostopnim ključem,
- `AWS_SESSION_TOKEN`: določa žeton uporabnikove seje.

[1] `https://aws.amazon.com/sdk-for-python/`

Prijavo in osveževanje poverilnic smo v našem primeru olajšali z uporabo AWS Single-Sign On (SSO). Orodje AWS CLI omogoča, da se z enim ukazom (`aws sso login`) prijavimo v želeni profil (lahko jih imamo več: enega za produkcijsko okolje, drugega npr. za testno okolje) in poskrbi, da se vse zahtevane okoljske spremenljivke nastavijo pravilno. Boto3 lahko nato preko poverilnic aktivnega profila SSO dostopa do AWS S3.

4.1.1 Filtriranje objektov v vedru

Pri inicializaciji odjemalca Boto3 za S3 moramo podati ime vedra, iz katerega želimo brati objekte. Po uspešni inicializaciji odjemalca s poizvedbo v vedru poiščemo recepte. Ker S3 ne omogoča filtriranja po priponi (angl. suffix), iteriramo preko vseh objektov v vedru in poiščemo tiste, ki se končajo z `recipe.json` (ti predstavljajo recepte).

Zaradi razvoja sistema Airdrop se je struktura ključa, pod katerim se shranjujejo objekti v S3, čez čas spreminjala. Stari recepti, ki uporabljajo staro strukturo ključa, nas ne zanimajo, zato smo jih odstranili iz rezultatov in rezultate razvrstili po datumu nastanka padajoče (od najnovejših do najstarejših).

4.1.2 Shranjevanje receptov

Prenos receptov izvedemo po skupinah, za vsako vrsto izvornega sistema posebej; recepte razvrstimo v skupine po 10 receptov glede na vrsto izvornega sistema. Za vsako skupino receptov na lokalnem upravljalcu receptov zaženemo transakcijo (4. vrstica na sliki 4.1), prenesemo datoteke JSON z recepti in zaključimo transakcijo. Brez uporabe transakcije bi se interno stanje upravljalca posodabljalo ob vsakem prenosu enega samega recepta. Z uporabo transakcije se izognemo nenehnemu posodabljanju internega stanja, ki se posodobi le ob končanem prenosu vsake skupine receptov.

```
def fetch_recipes(self):
    recipe_groups = self._get_recipe_groups(10)
    for system, group in recipe_groups.items():
        lrm.start_transaction()

        for path in group:
            if lrm.get_recipe(path) is None:
                fp = path.get_file_path()
                # Create the recipe's directory if needed
                fp.parent.mkdir(parents=True, exist_ok=True)
                # Download the recipe JSON
                self.bucket.download_file(path.path, str(fp))
                lrm.notify_new(path)

        lrm.end_transaction()
```

Slika 4.1: Prenos receptov iz S3 z uporabo transakcije na lokalnem upravljalcu receptov (v programski kodi: `lrm`).

4.2 Lokalni upravljalec receptov

Ker želimo v aplikaciji dostop do lokalnih receptov in generirane dokumentacije nadzorovati na enem mestu, smo se pri lokalnem upravljalcu receptov odločili za uporabo vzorca kreiranja Singleton.

Singleton je načrtovalski vzorec, ki zagotavlja, da ima določen razred v aplikaciji samo eno instanco[2]. S tem se izognemo konfliktom in neskladnostim, ki lahko nastanejo pri dostopanju do deljenih virov – v našem primeru do datotek na disku. Z uporabo Singletona dosežemo globalno dostopnost instance znotraj aplikacije in preprečimo prepisovanje instance, ki se lahko zgodi pri uporabi globalnih spremenljivk.

[2]Refactoring.Guru, *Singleton*. URL: `https://refactoring.guru/design-patterns/singleton`.

Za implementacijo načrtovalskega vzorca smo izkoristili Pythonov način delovanja, ki nam uporabo vzorca enostavno omogoča z moduli. Prvi primerek modulov v Pythonu se ustvari ob prvem uvozu, t.j. ob prvem stavku `import`, ki uvozi izbrani modul. Vse metode in spremenljivke znotraj izbranega modula se ob prvem uvozu ustvarijo v globalnem imenskem prostoru modula, vsak nadaljnji uvoz modula pa jih ne ustvari ponovno, temveč vrne referenco na že obstoječe.

4.2.1 Interno stanje

Interno stanje lokalnega upravljalca smo implementirali kot datoteko JSON. Format JSON smo izbrali zaradi enostavne berljivosti človeku in enostavne uporabe v programski kodi.

Datoteka je shranjena v vnaprej določenem imeniku, ki vsebuje tudi lokalne recepte. Za branje in pisanje internega stanja smo uporabili knjižnico `json`, za zagotavljanje celovitosti in preprečevanje napak v delovanju aplikacije pri branju stanja pa smo določili fiksno shemo JSON. Preverjanje veljavnosti sheme (angl. schema validation) izvedemo s knjižnico `jsonschema` ob vsakem zagonu aplikacije in vsaki posodobitvi internega stanja. Če preverjanje veljavnosti sheme ne uspe, se aplikacija ustavi. Datoteko lahko popravimo ročno, ali pa jo izbrišemo in aplikaciji prepustimo, da jo ponovno ustvari.

Posodobitve internega stanja smo znotraj aplikacije programsko omogočili v sklopu enega recepta ali v sklopu transakcije, kjer prenesemo več receptov naenkrat (transakcije smo že omenili v razdelku 4.1.2). V sklopu prenosa enega recepta upravljalec receptov prebere novo preneseni recept in takoj posodobi interno stanje. V sklopu transakcije pa se interno stanje ne posodobi takoj, temveč le, ko transakcijo eksplicitno zaključimo. Stanje transakcije nadziramo z globalno spremenljivko `_is_transaction` znotraj modula. Ker Python ne pozna koncepta zasebnih metod in spremenljivk, smo pri implementaciji sledili slogovnemu vodniku PEP 8[3], ki določa, naj imajo zasebne metode in spremenljivke predpono _.

[3] `https://peps.python.org/pep-0008`

4.2.2 Posredovanje receptov generatorju

Generator dokumentacije recepte od lokalnega upravljalca prejme preko metod `get_recipes(system: str)` in `get_recipe(s3_path: S3Path)`, ki sta kot javni metodi izpostavljeni za uporabo v ostalih modulih aplikacije.

Ob klicu metode `get_recipes(system: str)` lokalni upravljalec iz internega stanja pridobi recepte za podano vrsto zunanjega sistema (parameter `system`). V primeru, da za izbrani zunanji sistem ne obstaja noben recept, vrne prazen seznam.

Z metodo `get_recipe(s3_path: S3Path)` pa lahko za podano interno predstavitev recepta v S3 pridobimo točno določen recept. To metodo uporabljamo samo pri prenosu receptov iz S3, generator dokumentacije je ne uporablja.

4.3 Generiranje dokumentacije

Generator dokumentacije smo implementirali z razredom `Pipeline`, ki obsega celoten cevovod generiranja dokumentacije za izbran recept, vključno z integracijo z uporabniškim vmesnikom za sledenje delovanju. Za implementacijo generatorja smo uporabili ogrodje LangChain.

LangChain je odprtokodno ogrodje za razvoj aplikacij, ki jih poganjajo veliki jezikovni modeli[4]. Ponuja širok nabor knjižnic z modularnimi gradniki, komponentami, orodji in integracijami z zunanjimi storitvami, ki pospešijo in olajšajo razvoj takih aplikacij. S pomočjo LangChaina lahko zgradimo klepetalne robote, robote za odgovarjanje na vprašanja, aplikacije za ekstrahiranje strukturiranih podatkov in ostale aplikacije, ki uporabljajo velike jezikovne modele.

LangChain nam implementacijo cevovodov poenostavi s konceptom verig (angl. chain). Verige so zaporedja klicev, v katerih lahko kličemo velike jezikovne modele, orodja in korake za predprocesiranje ali postprocesiranje

[4]LangChain, Inc., *LangChain Docs*. URL: `https://python.langchain.com/`

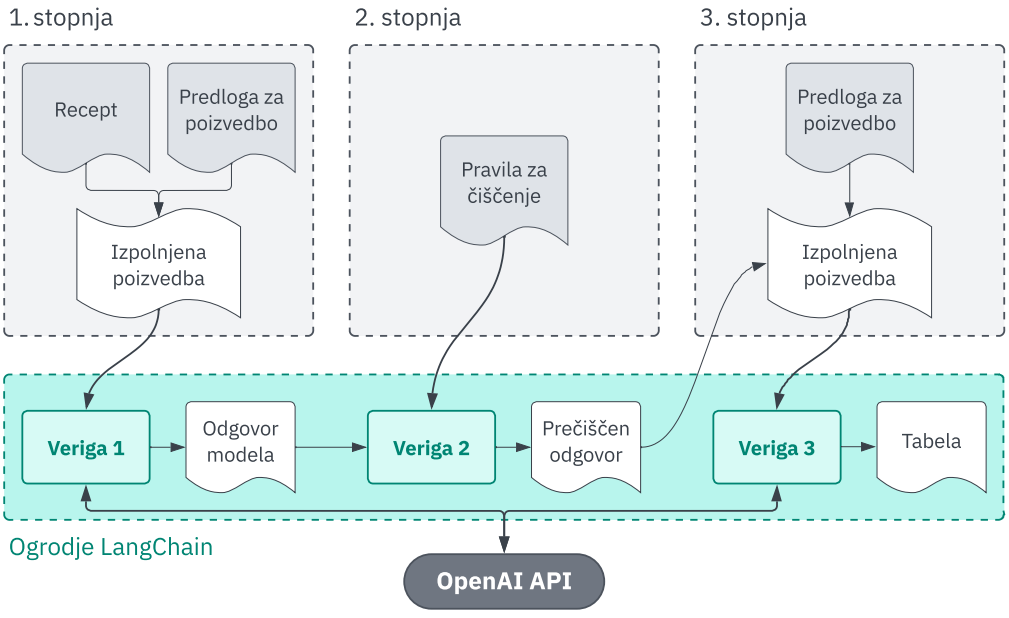
podatkov. Verige lahko med sabo povežemo tudi v daljše verige, kjer izhodni podatki ene verige postanejo vhodni podatki druge verige.

4.3.1 Struktura cevovoda

Cevovod smo sestavili iz treh stopenj:

1. generiranje dokumentacije iz recepta,
2. čiščenje generirane dokumentacije,
3. preoblikovanje prečiščene dokumentacije v tabelo.

LangChain nam izvedbo vsake stopnje omogoča s posamezno verigo, vse 3 verige pa nato za popolno izvedbo združimo v daljšo verigo. Cevovod smo za lažjo predstavo s shemo ponazorili na sliki 4.2.



Slika 4.2: Predlogo za poizvedbo izpolnimo s podatki iz recepta, z LLM-jem zgeneriramo dokumentacijo, jo prečistimo s pravili in generirano dokumentacijo z LLM-jem preoblikujemo v tabelo.

4.3.2 Generiranje dokumentacije iz recepta

Prva stopnja cevovoda je **generiranje dokumentacije iz recepta**. Kot smo omenili v razdelku 3.5, posredovanje recepta v jezikovni model brez dodatnega konteksta za našo nalogo ne zadostuje. V ta namen smo izbrali, katere podatke iz recepta bomo posredovali kot kontekst, in napisali pripadajoče predloge za poizvedbe z rezerviranimi mesti za izbrane podatke.

Ker celoten recept običajno presega največjo dovoljeno dolžino konteksta, smo za vsak izhod recepta iz seznama izhodov recepta (glej razdelek 3.1.1 o strukturi recepta) ustvarili posamezno poizvedbo. Če ima izbrani recept recimo 3 izhode, to pomeni, da bomo ustvarili 3 poizvedbe, in vsako posebej posredovali v jezikovni model. Za jezikovni model smo izbrali GPT-3.5 Turbo, ki ga uporabljamo preko storitve OpenAI API.

Vsako poizvedbo smo sestavili iz 3 sporočil, ki so osnovne enote v OpenAI Chat Completions API[5]. Sporočila so sestavljena iz vsebine in vloge, ki je lahko **sistemska** (angl. system role), **uporabniška** (angl. user role) ali **pomočniška** (angl. assistant role). Sistemska sporočila predstavljajo navodila jezikovnemu modelu, ki določajo, kako naj model odgovarja na poizvedbe. Tu lahko določimo želeno natančnost, ton in stil odgovorov. Uporabniška sporočila so naša sporočila, komentarji, vprašanja in poizvedbe. Sporočila, ki jih generira model, pa imajo pomočniško vlogo.

Sistemsko sporočilo

V poizvedbi najprej podamo sistemsko sporočilo, s katerim določimo vlogo modela kot pisca dokumentacije, ki ga poganja umetna inteligenca. Povemo, da bo njegova naloga opisovanje preslikav polj med objekti v DevRevu in objekti v zunanjem sistemu. Navedemo nekaj navodil, kako naj model opiše preslikave, kakšne informacije naj vključi in izpusti, ter povemo, naj se izogiba kakršnemukoli drugemu besedilu, ki ne sledi našim navodilom. Izsek iz

[5]OpenAI, *Text generation - OpenAI API*. URL: `https://platform.openai.com/docs/guides/text-generation`

predloge smo prikazali na sliki 4.3, kjer pa smo bolj specifična navodila, ki se glede na izvorni sistem razlikujejo, izpustili.

```
You are an AI documentation writer. You are writing
documentation for mappings between fields of DevRev and
{external_system} (hereinafter: source) objects.
...
Each DevRev field should appear in your response once only.
Minimize any other prose.
```

Slika 4.3: Predloga za sistemsko sporočilo z zahtevano vlogo, opisom naloge in zahtevano obliko

Moduli

Ker želimo, da bo generirana dokumentacija čim bolj natančna, moramo jezikovnemu modelu podati čim več konteksta, zato drugo sporočilo v poizvedbi vsebuje pare imen in definicij modulov jq iz recepta (glej sliko 4.4). Transformacijski filtri uporabljajo vnaprej določene funkcije iz modulov, moduli pa so zato pomembni za razlago filtrov. Recept ima lahko več modulov, zato smo pri sestavi poizvedbe uporabili načela tehnike *few-shot prompting*.

```
The following are jq modules defined by their names and jq
code. Remember the modules as they contain functions that are
used within transformation queries between objects.

Module name: {name}
jq code: {definition}
```

Slika 4.4: Predloga za poizvedbo za module, kjer module navedemo kot pare imen modula in pripadajoče kode

Few-shot prompting je tehnika, kjer z navajanjem primerov v poizvedbi jezikovnemu modelu omogočimo učenje znotraj konteksta (angl. in-context learning)[6]. Primeri so sestavljeni iz vhodov in izhodov, na koncu poizvedbe pa navedemo nov vhod brez izhoda, za katerega naj model generira izhod. Z navajanjem primerov ponazorimo, kakšne izhode naj model generira, hkrati pa želimo izboljšati zmogljivost modela in s tem doseči boljše rezultate. Prednosti tehnike so veliko zmanjšanje števila podatkov, ki bi jih sicer potrebovali za natančno nastavljanje modela, in manjša verjetnost prekomernega prileganja podatkom iz množice za natančno nastavljanje [3]. Slabosti tehnike pa so kljub temu slabša zmogljivost kot na modelih, ki so natančno nastavljeni za izbrano nalogo.

V našem primeru omenjene tehnike nismo uporabili, smo pa uporabili njena načela. Za vsak podan primer definiramo predlogo, ki se nato izpolni z izbranimi podatki iz vsakega primera (za našo nalogo sta to ime in definicija modula). Dodamo lahko tudi predpono in pripono, ki se dodata na začetek oz. konec poizvedbe.

Izhodi recepta in transformacijski filtri

Zadnji in najpomembnejši del poizvedbe so izhodi recepta, iz katerih bomo generirali dokumentacijo. Kot smo že omenili, bomo za vsak izhod recepta sestavili po eno poizvedbo.

Vsak izhod recepta je sestavljen iz vrste objekta na platformi DevRev in seznama pripadajočih vhodov, ki so sestavljeni iz vrste vhodnega objekta v izvornem sistemu, transformacijskega filtra in dokumentacije. Dokumentacije, ki je vsebovana v vhodih, ne vključimo v poizvedbi, ker je preobsežna, v večini primerov nerazumljiva in ne ponuja nobene dodane vrednosti, ki bi lahko pomagala pri razumevanju recepta. Ostale podatke vključimo in v ta namen, tako kot pri sestavljanju poizvedbe oz. sporočila za module, uporabimo načela tehnike *few-shot prompting*. Vhodnih objektov, ki se preslikajo

[6]Elvis Saravia, *Prompt Engineering Guide*. URL: `https://github.com/dair-ai/Prompt-Engineering-Guide`.

v eno vrsto objekta na platformi DevRev, je namreč lahko več.

V predponi predloge povemo, da bomo v nadaljevanju navedli vrste vhodnih objektov v izvornem sistemu in pripadajoče transformacijske filtre, ki določajo preslikave v določeno vrsto objekta na platformi DevRev. Nato za vsako vrsto vhodnega objekta podamo prej omenjene pare vrste vhodnih objektov in pripadajočega filtra, v priponi pa podamo dodatna navodila in zahtevani format izhoda.

Pripono smo glede na preizkuse delovanja in pridobljene rezultate razvili v več iteracijah, ki je zaradi naših zahtev postala precej obsežna. V priponi navedemo navodila za želeno natančnost generiranega izhoda:

1. Filtri lahko vsebujejo spremenljivke, v izhodu pa naj se spremenljivke zamenjajo z dejanskimi imeni polj.

2. Objekti lahko vsebujejo tudi vgnezdena polja, ki naj bodo vsa prisotna v izhodu.

3. Če se polje preslika v konstantno vrednost, naj bo ta vrednost ovita v dvojne narekovaje.

Priponi glede na vrsto izvornega sistema recepta dodamo še specifična navodila, ki smo jih razvili iz rezultatov obnašanja modela ob posredovanju receptov za ta izvorni sistem. Specifična navodila dodamo za izvorne sisteme GitHub, HubSpot, Jira, Linear in Zendesk. Zahtevamo tudi specifičen format izhoda, ki naj vsebuje vrsto vhodnega objekta v izvornem sistemu, vrsto objekta na platformi DevRev in preslikave med polji teh objektov v obliki seznama alinej.

4.3.3 Čiščenje generirane dokumentacije

Stopnjo čiščenja generirane dokumentacije smo dodali po opazovanjih, da izhodi modela vsebujejo tudi podatke, ki jih ne želimo prikazati. Glede na opazovanja smo določili dve pravili, po katerih filtriramo izhod modela in odstranimo neželene vzorce.

Prvi vzorec, ki ga odstranimo, se pojavlja pri objektih, ki lahko vsebujejo polja po meri. V tem primeru polje `custom_fields` v objektih obstaja, vendar ne vsebuje vgnezdenih polj (recimo `custom_fields.nested_field`). Take preslikave odstranimo iz izhoda tako, da poiščemo polje `custom_fields`. Če je to edino polje v izhodu in vgnezdenih polj ni v izhodu, preverimo, ali preslikava vsebuje besedilo `empty`, prazen objekt `{}`, ali pa če se za to polje samo kliče funkcija iz modulov, ki odstranjuje prazna polja.

Drugi vzorec je vključevanje metapodatkov ali kode filtrov jq v izhodne preslikave. Če izhodna preslikava vsebuje `migration_metadata` ali znak za cevovod `|`, po katerem prepoznamo filter, to preslikavo odstranimo iz izhoda.

4.3.4 Preoblikovanje v tabelo

Zadnja stopnja cevovoda je izbirna, omogoča pa preoblikovanje prečiščenega izhoda (dokumentacije) v tabelo Markdown. Za to stopnjo cevovoda smo tudi uporabili jezikovni model, bi jo pa lahko implementirali tudi ročno s samostojno metodo, ki razčleni izhod in oblikuje tabelo.

V tej stopnji modelu določimo vlogo pisca tabel Markdown, ki ga poganja umetna inteligenca. Navedemo, da bo ustvarjal tabele za preslikave med polji objektov na platformi DevRev in zunanjimi sistemi, kot vhod pa bo dobil dokumentacijo v vnaprej določenem formatu, ki smo ga opisali v prejšnjih dveh poglavjih. Uporabimo različico tehnike *few-shot prompting*, imenovano *one-shot prompting*, kjer podamo samo en primer vhoda in izhoda. Zahtevamo, naj glede na naše zahteve zgenerira tabelo Markdown, pri čemer naj tabeli ne dodaja nepotrebnih praznih znakov (presledkov in novih vrstic). S tem zmanjšamo porabo žetonov in posledično ceno generiranja tabele.

4.3.5 Integracija z uporabniškim vmesnikom

Za integracijo z uporabniškim vmesnikom smo uporabili orodje Aim[7], odprtokodno orodje, ki omogoča enostavno sledenje eksperimentom. Namenjeno

[7] `https://aimstack.io/`

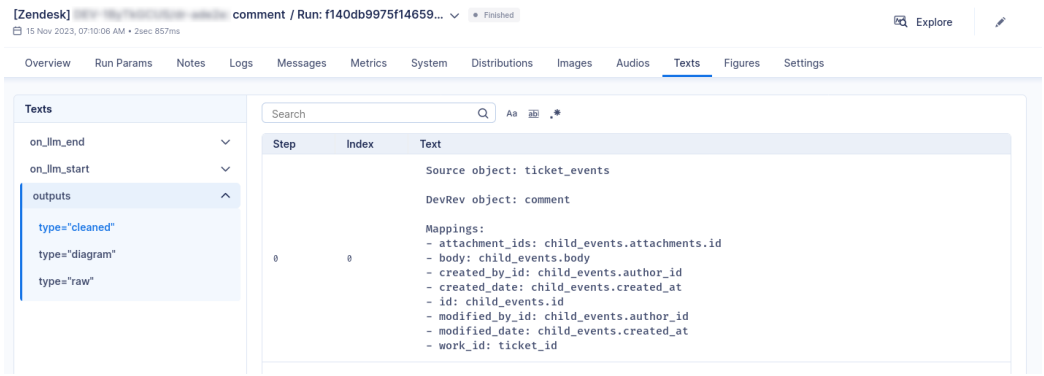
je predvsem sledenju učenja modelov strojnega učenja, vendar ga lahko uporabljamo tudi za sledenje napovedanju, med drugim tudi preko zunanjih storitev kot so OpenAI API. Orodje temelji na konceptu izvajanj (angl. run), omogoča pa enostavno beleženje in primerjavo metapodatkov med izvajanji, boljšo organizacijo eksperimentov, pregled z uporabniškim vmesnikom in API za programsko poizvedovanje.

LangChain omogoča metode povratnih klicev, ki jih lahko vključimo v različne stopnje delovanja naše aplikacije. Obstoječa integracija orodja Aim v LangChain nam olajša delo in že brez dodatne konfiguracije sledi vhodom, izhodom in parametrom modela in metrikam sistema kot so uporaba procesorja, pomnilnika itd. Za organizacijo in lažje poizvedovanje določimo predpono eksperimenta, ki je sestavljena iz vrste zunanjega sistema, enoličnega identifikatorja organizacije in enote.

```
# Track the filtered completion from GPT
aim_clean = Text(input["documentation"])
self.AIM_HANDLER._run.track(
    aim_clean, name="outputs", context={"type": "cleaned"}
)
```

Slika 4.5: Z Aimom shranimo izhod 2. stopnje cevovoda, prečiščeno dokumentacijo.

Izvirno, prečiščeno in dokumentacijo v obliki tabele shranimo z uporabo metode, ki jo ponuja Aim. Orodje sicer že shranjuje vhode in izhode v model preko privzetih povratnih klicev, vendar smo za boljšo preglednost shranjevanje implementirali tudi sami. Dokumentacija je v obliki nizov, zato jo najprej shranimo v Aimov ovojni razred `Text`. V parametrih metode `track` definiramo ime zbirke, v katero naj se dokumentacija shrani, in dodamo slovar metapodatkov oz. konteksta shranjenega izhoda (glej sliko 4.5). Tako shranjeno dokumentacijo lahko nato pregledujemo znotraj Aimovega uporabniškega vmesnika (glej sliko 4.6, občutljive podatke smo zameglili).



Slika 4.6: Na primeru izvornega sistema Zendesk pregledamo shranjeno prečiščeno dokumentacijo za komentar na platformi DevRev `comment`. Na levi strani lahko izberemo tudi tabelo ("diagram") ali izvorni izhod modela ("raw"). Pregledamo lahko tudi shranjene vhode in izhode privzetih povratnih klicev kot sta `on_llm_end` in `on_llm_start`. Vidimo, da se npr. vgnezdeno polje `child_events.body` objekta `ticket_events` na platformi Zendesk preslika v polje `body` komentarja na platformi DevRev.

Poglavje 5

Rezultati in evalvacija

V tem poglavju bomo predstavili rezultate implementacije naše rešitve. Ovrednotili bomo zmogljivost jezikovnega modela GPT-3.5 Turbo na naši nalogi in rezultate primerjali z izhodiščem – obstoječo dokumentacijo, oz. če ta ne obstaja, s transformacijskim filtrom v receptih.

5.1 Izbira kriterijev vrednotenja

Ker cilj naše naloge in posledično evalvacija temeljita na **berljivosti dokumentacije**, ki je subjektivna glede na bralca, smo pri izbiri kriterijev vrednotenja naleteli na težavo. Standardni kriteriji kot so specifičnost (angl. precision), občutljivost (angl. recall) in mera F1 so tipično uporabljeni v klasifikacijskih nalogah za evalvacijo binarnih ali večrazrednih modelov. Ti kriteriji niso primerni za našo nalogo, ki je generiranje berljive dokumentacije; osredotočamo se na kvaliteto in celovitost generirane dokumentacije. V ta namen smo izbrali naslednja 2 kriterija, ki najbolje zajameta naše cilje in zahteve, pri tem pa hkrati dobro opišeta kakovost naših rezultatov.

- **Točnost** (angl. accuracy): rezultat primerjamo z izhodiščem (dokumentacijo ali transformacijskim filtrom). Kriterij številčno predstavimo kot delež pravilno dokumentiranih preslikav izmed vseh preslikav, vključenih v rezultatu (generirani dokumentaciji).

- **Celovitost** (angl. completeness): celovitost predstavlja delež preslikav, ki obstajajo v receptu in so tudi vključene v rezultatu. Vrednost 100 % pomeni, da so v rezultatu vključene vse preslikave iz recepta.

Točnost bomo torej izračunali tako, da bomo prešteli število vseh pravilno dokumentiranih preslikav v dokumentaciji, in to številko delili s številom vseh preslikav v dokumentaciji. Če dokumentacija vsebuje 15 preslikav in je od tega 12 pravilnih, bo točnost znašala $\frac{12}{15} = \frac{4}{5} = 80$ %.

Celovitost pa bomo izračunali tako, da bomo prešteli število vseh preslikav v dokumentaciji, ki obstajajo tudi v receptu samem, in to številko delili s številom vseh preslikav v receptu. Če recept vsebuje 10 preslikav, v dokumentaciji pa imamo dokumentiranih tudi 10 preslikav, ampak jih od tega v receptu obstaja samo 7, bo celovitost znašala $\frac{7}{10} = 70$ %.

Našo evalvacijo bomo torej izvajali ročno, ker moramo za vsako preslikavo preveriti, ali sploh obstaja v receptu, in preveriti, ali se ujema z dejanskim transformacijskim filtrom iz recepta.

5.2 Vrednotenje rezultatov

Vrednotenje rezultatov smo razvrstili v skupine glede na vrsto izvornega sistema v receptu. Ovrednotili bomo izvirni izhod modela pred stopnjami postprocesiranja (t.j. čiščenja in preoblikovanja v tabelo), ker nas pri vrednotenju zanimajo tudi preslikave, ki jih sicer ne prikažemo.

Pri vrednotenju smo opazili, da se v rezultatih pojavljajo preslikave z manjkajočimi predponami (npr. `property` namesto `properties.property`), z napačnim ločilom pri vgnezdenih poljih (npr. `properties_property` namesto `properties.property`) ali z delnim izhodom (ne vsebuje vseh možnih polj, ki se lahko preslikajo v eno polje na platformi DevRev), sicer pa so imena polj v teh preslikavah pravilna. Take preslikave smo pri računanju točnosti upoštevali kot polovico pravilne preslikave.

V nadaljevanju bomo število pravilno dokumentiranih preslikav označili z $N_{pravilni}$, število vseh preslikav v rezultatu z N_{izhod} in število vseh preslikav

v receptu z N_{recept}. Točnost in celovitost izračunamo s formulama 5.1 in 5.2.

$$\text{Točnost} = \frac{N_{pravilni}}{N_{izhod}} \qquad (5.1) \qquad \text{Celovitost} = \frac{N_{izhod}}{N_{recept}} \qquad (5.2)$$

5.2.1 Parametri modela

Ovrednotili bomo dokumentacijo, ki je bila generirana z iteracijo modela GPT-3.5 Turbo `gpt-3.5-turbo-1106`. Parameter modela, imenovan temperatura (angl. temperature), nam omogoča nadzorovanje kreativnosti oz. naključnosti generiranega besedila in se giblje med vrednostima 0 in 1[1]. Višja temperatura omogoča bolj raznolike in nepredvidljive izhode, saj dopušča večjo verjetnost za izbiro manj verjetnih besed ali fraz. Nižja temperatura zmanjšuje stopnjo naključnosti in spodbuja generiranje bolj deterministčnih in natančnih izhodov. V našem primeru smo izbrali vrednost 0.1, ker želimo procesu generiranja vseeno pustiti nekaj naključnosti oz. ustvarjalnosti, kljub temu pa bo generirana dokumentacija še zmeraj sledila navodilom in zahtevani obliki.

5.2.2 Dokumentirani objekti

V tem razdelku bomo za lažje razumevanje ovrednotenja dokumentacije opisali, kaj v nadaljevanju omenjeni objekti predstavljajo na platformi DevRev. Temeljni objekt, ki združuje in povezuje vse spodaj naštete objekte, je organizacija, ki predstavlja osnovno enoto na platformi.

- `account` predstavlja podjetje stranke in vsebuje informacije o podjetju kot so ime, naslov, industrija, domena in spletni naslov;
- `dev_user` je član organizacije, ki je del razvojne ali podporne ekipe (razvijalec, oblikovalec, podporni agent ipd.). Razvija, gradi ali zagotavlja produkte strankam;

[1] OpenAI Developer Forum, *Cheat Sheet: Mastering Temperature and Top_p in ChatGPT API*. URL: `https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683`

- `rev_user` je član organizacije, ki predstavlja stranko in pripada objektu `account`;
- `sys_user` je sistemski uporabnik, ki deluje avtomatsko in se tipično aktivira ob določenih dogodkih. Primer so boti;
- `work` je katerikoli objekt na platformi DevRev, ki ima lastnika in zahteva nekaj truda (dela), da se dokonča. Med sabo so lahko povezani v različnih razmerjih (starš-otrok ali npr. odvisnost, kjer je en `work` predpogoj za začetek dela na drugem. To je lahko `conversation`, ki predstavlja pogovor s stranko; lahko je `ticket`, ki predstavlja zahtevo s strani stranke; lahko je `issue`, ki predstavlja zaključeno enoto dela za razvijalca; lahko je `task`, ki predstavlja manjšo enoto dela in je ponavadi vezan na `issue`;
- `link` je povezava med objekti tipa `work`, s čimer na platformi DevRev ohranjamo kontekst (definicijo problema, zahteve, potrebno delo);
- `tag` je oznaka, tipično beseda, s katero lahko različnim objektom (`work`, `user` itd.) dodamo več konteksta, ki nam jih pomaga identificirati. Z ustvarjanjem asociacij pomagajo pri organizaciji in kategorizaciji teh objektov.

5.2.3 HubSpot

HubSpot je platforma za trženje, prodajo in podporo strankam, ki podjetjem omogoča optimizacijo poslovnih procesov. Vsebuje orodja za CRM, različne vrste trženja in analitiko. Namenjena je izboljšanju komunikacije s strankami in povečanju prodaje.

Izbrali smo 3 recepte z različnimi izvornimi enotami (to pomeni, da pripadajo različnim projektom na platformi HubSpot) in posledično različnimi podatki, polji po meri in preslikavami.

Pri objektih `account` in `rev_user` opazimo izpuščanje predpon, ki smo ga omenili v razdelku 5.2 (posledično vidimo nizko točnost za `account` v

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
account	68,5	74	75	92,56	98,66
comment	3,5	6	9	58,33	66,67
dev_user	7	8	8	87,50	100,00
rev_user	40,5	92	93	44,02	98,92
work	62,5	64	64	97,65	100,00
Skupno	182	244	249	74,59	97,99

Tabela 5.1: Rezultati vrednotenja 1. recepta za HubSpot

tabeli 5.3 in nizko točnost za `rev_user` v tabeli 5.1). Pri objektu `comment` v tabeli 5.1 točnost zmanjšuje uporaba napačnega ločila pri vgnezdenih poljih (glej sliko 5.1). V splošnem so preslikave kljub omenjenim pomanjkljivostim razumljive; večjih napak, ki bi vplivale na razumljivost dokumentacije nismo zaznali.

```
Source object: notes
DevRev object: comment

Mappings:
- body: properties.hs_note_body
- created_by_id: properties_hubspot_owner_id
- created_date: properties_hs_createdate
- id: properties_hs_object_id
- modified_by_id: properties_hubspot_owner_id
- modified_date: properties_hs_lastmodifieddate
```

Slika 5.1: Generirana dokumentacija za objekt `comment` pri 1. receptu za HubSpot, kjer je namesto pike med `properties` in imenom polja uporabljen podčrtaj

Imamo nekaj osamelcev z nizko točnostjo ali nizko celovitostjo ali obojim (`comment` in `rev_user` v tabeli 5.1 ter `account` v tabeli 5.3). Ostali objekti v

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
account	83	88	89	94,31	98,87
dev_user	7	8	8	87,50	100,00
work	91	92	92	98,91	100,00
Skupno	181	188	189	96,27	99,47

Tabela 5.2: Rezultati vrednotenja 2. recepta za HubSpot

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
account	27,5	51	51	53,92	100,00
dev_user	7	8	8	87,50	100,00
rev_user	66,5	68	69	97,79	98,55
Skupno	101	127	128	79,52	99,21

Tabela 5.3: Rezultati vrednotenja 3. recepta za HubSpot

splošnem dosegajo visoko točnost in celovitost nad 85 %. Visoke točnosti in celovitosti smo dosegli pri 2. receptu (glej tabelo 5.2). Opazili smo, da lahko rezultati variirajo med različnimi izvajanji generiranja dokumentacije za isti recept; nekatere napake se v takih primerih popravijo, nastanejo pa nove.

5.2.4 Jira

Jira je platforma za upravljanje projektov in sledenje napak, ki jo razvija Atlassian. Primarno se uporablja za načrtovanje in sledenje razvoju programske opreme ter učinkovito sodelovanje med ekipami razvijalcev.

Pri receptih za Jiro smo opazili, da imajo objekti `comment`, `dev_user`, `link`, `sys_user` in `tag` vedno enako točnost in celovitost (kot je razvidno iz tabel 5.4 in 5.5), tudi v receptih za različne izvorne enote (projekte v Jiri). Ob podrobnejšem pregledu receptov smo zaključili, da so transformacijski filtri za te objekte določeni vnaprej, t.j. vedno enaki, ne glede na izvorno enoto v Jiri, ki je bila uvožena.

S tem smo odkrili, da so napake jezikovnega modela GPT-3.5 Turbo na naši nalogi za Jiro vedno iste vrste. Napake se pojavljajo pri preslikavah za

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
comment	7	7	7	100,00	100,00
dev_user	8,5	9	9	94,44	100,00
link	6,5	7	7	92,85	100,00
sys_user	4	4	4	100,00	100,00
tag	6	6	6	100,00	100,00
work	30	30	30	100,00	100,00
Skupno	62	63	63	98,41	100,00

Tabela 5.4: Rezultati vrednotenja 1. recepta za Jiro

objekta `dev_user`, kjer preslikava za polje `state` vsebuje preveč polj iz Jire, in `link`, kjer preslikavi za polje `link_type` manjka predpona `fields` (zato dosegata ta objekta tudi enako točnost in celovitost v tabelah 5.4 in 5.5). Sklepamo, da gre tu za napake zaradi variacij v strukturi transformacijskih filtrov med različnimi izvornimi sistemi – filtri za Jiro so kompleksnejši kot npr. za HubSpot. Napake so tako kot pri HubSpotu manjše in na razumljivost dokumentacije ne vplivajo. Ker smo našo aplikacijo primarno razvijali in testirali z recepti za Jiro, smo tu pričakovali visoko točnost in celovitost (točnost nikoli ne pade pod 92,85 %, celovitost pa je povsod 100,00 %).

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
comment	7	7	7	100,00	100,00
dev_user	8,5	9	9	94,44	100,00
link	6,5	7	7	92,85	100,00
sys_user	4	4	4	100,00	100,00
tag	6	6	6	100,00	100,00
work	33,5	34	34	98,52	100,00
Skupno	65,5	67	67	97,76	100,00

Tabela 5.5: Rezultati vrednotenja 2. recepta za Jiro

5.2.5 Linear

Linear je novejša platforma za upravljanje projektov, konceptualno podobna Jiri, ki omogoča učinkovito sledenje nalogam. Ponuja integracijo z različnimi orodji za razvoj programske opreme in sodelovanje med ekipami.

Podobno kot pri Jiri so transformacijski filtri za objekte vrste `link` določeni vnaprej, kar razloži enake vrste napak, in posledično enake rezultate pri različnih receptih (glej tabeli 5.6 in 5.7, kjer ima `link` v obeh primerih enako točnost in celovitost).

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
comment	7	7	7	100,00	100,00
dev_user	9	9	9	100,00	100,00
link	5,5	6	6	91,66	100,00
tag	5	5	5	100,00	100,00
work	12	13	15	92,30	86,66
Skupno	38,5	40	42	96,25	95,23

Tabela 5.6: Rezultati vrednotenja 1. recepta za Linear

```
"tags": (."labels" | (."nodes" |
    map((."name" | { "tag_id": . }))
))
```

Slika 5.2: Filter za ekstrakcijo oznak iz sistema Linear, ki ga jezikovni model najverjetneje zaradi vgnezdenosti in mapiranja ne razume najbolje.

Pri objektih vrste `work` so napake minimalne: preslikava za `tag_id` znotraj seznama oznak `tags` je ponavadi izpisana samo kot `tags`. Predvidevamo, da se ta napaka zgodi zaradi oblike filtra, predstavljenega na sliki 5.2, ki najprej dostopa do vgnezdenih polj objekta in nato vsakega preslika v posamezen objekt. Včasih manjka polje `work_type`, ki pa je lahko ob ponovnem generiranju dokumentacije prisotno v dokumentaciji.

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
comment	7	7	7	100,00	100,00
dev_user	8	8	8	100,00	100,00
link	5,5	6	6	91,66	100,00
tag	5	5	5	100,00	100,00
work	15	16	16	93,75	100,00
Skupno	40,5	42	42	96,42	100,00

Tabela 5.7: Rezultati vrednotenja 2. recepta za Linear

5.2.6 Zendesk

Zendesk je platforma za podporo strankam, ki ponuja upravljanje izkušnje strank preko več kanalov (e-pošta, telefon itd.). Vsebuje orodja za pomoč uporabnikom, avtomatizacijo podpore in analitiko.

Objekt	$N_{pravilni}$	N_{izhod}	N_{recept}	Točnost [%]	Celovitost [%]
account	9	10	11	90,00	90,90
comment	9	9	9	100,00	100,00
dev_user	8	8	8	100,00	100,00
link	6	6	6	100,00	100,00
rev_user	12	13	13	92,30	100,00
tag	4,5	6	6	75,00	100,00
work	19	20	20	95,00	100,00
Skupno	67,5	72	73	93,75	98,63

Tabela 5.8: Rezultati vrednotenja recepta za Zendesk

Pri dokumentaciji objekta `work` pri receptih za Zendesk se pojavlja enaka vrsta napake z oznakami kot pri receptih za Linear – napačen izpis preslikave za `tags.tag_id`. Poleg te napake se v oznakah (objekt `tag`) pojavlja tudi napaka, kjer model v izhodu za polja `description`, `id` in `name` izpiše samo piko (glej sliko 5.3). Oznake v Zendesku so namreč samo nizi, filter pa tako ta 3 polja nastavi na ime oznake. Izhod modela torej ni napačen, vendar bi

lahko tak primer obravnavali posebej in v poizvedbo dodali navodilo, ki bi modelu nakazalo, s čim naj zamenja piko.

```
Source object: organizations
DevRev object: tag

Mappings:
- access_level: "internal"
- created_by_id: "SYSU-1"
- description: .
- id: .
- name: .
- style.color.code: "#3333FF"
```

Slika 5.3: Generirana dokumentacija za objekt `tag` pri receptu za Zendesk, kjer je za polja `description`, `id` in `name` izpisana samo pika.

Točnost in celovitost sta tu sicer visoka pri vseh objektih razen pri objektu `tag` (glej tabelo 5.8), kar predvidevamo, da izhaja iz preprostosti filtrov in majhnega števila polj v objektih v primerjavi z drugimi izvornimi sistemi.

5.2.7 Zaključki

Skupno smo na vseh ovrednotenih receptih dosegli povprečno točnost 87,54 % in povprečno celovitost 98,82 %. Za našo nalogo sta pomembna oba kriterija: celovitost želimo maksimizirati, hkrati pa želimo pri maksimalni celovitosti maksimizirati tudi točnost. Z drugimi besedami: če je model v generirani dokumentaciji vključil N_{izhod} preslikav (kjer želimo, da je N_{izhod} čim bližje številu preslikav v dejanskem receptu N_{recept}), želimo, da bo čim več teh preslikav tudi pravilnih (torej da bo število pravilnih preslikav $N_{pravilni}$ čim bližje N_{izhod}). Kar se tiče teh dveh kriterijev, smo malce nižjo povprečno točnost pričakovali, celovitost pa se je izkazala za zelo dobro; pri večini dokumentiranih objektov je znašala kar 100,00 %. S subjektivnega vidika vrednotenja so

nam doseženi rezultati omogočili lažji vpogled v preslikave, saj smo se znebili prebijanja skozi dolge neberljive filtre v programskem jeziku jq.

Po drugi strani pa vidimo, da je metoda lahko tudi nezanesljiva. Pri receptih za vse izvorne sisteme (posebej pa pri platformah HubSpot, glej razdelek 5.2.3, in Linear, glej razdelek 5.2.5) smo opazili, da lahko ponovni zagon generatorja dokumentacije spremeni generirano dokumentacijo in posledično točnost in celovitost. Pri prvem izvajanju lahko dosežemo dobre rezultate pri obeh kriterijih, pri naslednjih izvajanjih za isti recept pa lahko že en ali oba drastično padeta zaradi narave delovanja jezikovnih modelov.

Druga težava je, da smo metodo zasnovali z enim samim modelom, GPT-3.5-Turbo, s čimer smo navodila v poizvedbah prilagodili delovanju in dobrim praksam za uporabo tega modela. Metodo smo tudi preizkusili na modelu GPT-4-Turbo in ugotovili, da se že med modeloma iz iste družine pojavljajo razlike v razumevanju poizvedb in stilu generirane dokumentacije. Metoda je torej občutljiva na izbiro modela in jo moramo glede na model prilagoditi. Če bi uporabili kakšen drug model, recimo LLaMa 2, bi navodila v poizvedbah skoraj zagotovo morali prilagoditi, kar bi ponovno zahtevalo preizkuševanje in popravljanje – torej dodatno porabo časa.

Zadnja težava, ki jo lahko predvidimo, se lahko zgodi ko platforma DevRev začne podpirati nov izvorni sistem, ali pa če se oblika receptov popolnoma spremeni. V primeru podpore novega izvornega sistema, za katerega še nimamo specifičnih navodil za poizvedbe, bi lahko generirana dokumentacija vključevala podatke, ki jih ne želimo, ali pa bi bila popolnoma neuporabna. V primeru spremembe oblike receptov (recimo konec uporabe programskega jezika jq ali strukture JSON recepta) pa bi morali celoten proces generiranja dokumentacije prilagoditi na novo obliko, kar bi ponovno zahtevalo razvijalsko delo.

Poglavje 6

Zaključek

V diplomskem delu smo uporabili velike jezikovne modele za generiranje dokumentacije iz izvorne kode. Osredotočili smo se na model GPT-3.5 Turbo, pri tem pa smo raziskali nov pristop s tehnikami inženiringa poizvedb (angl. prompt engineering). Razvili smo prototip aplikacije za generiranje dokumentacije, ga preizkusili in pridobljene rezultate ročno ovrednotili s kriterijema točnosti in celovitosti. Pri vrednotenju smo odkrili, da so veliki jezikovni modeli za našo nalogo zelo zmogljivi, čeprav smo pri tem naleteli na nekaj težav, kot so pomanjkljivi izhodi in halucinacije.

V raziskavi smo sodelovali s podjetjem DevRev, ki je izrazilo zadovoljstvo z našim delom in pridobitvami. Pridobitve diplomskega dela so za podjetje dragocene, saj omogočajo lažje razumevanje transformacij z recepti in posledično razhroščevanja napak ter interpretacij delovanja storitve Airdrop.

Z raziskavo smo tudi pridobili znanje in izkušnje na področjih inženiringa poizvedb in obnašanja velikih jezikovnih modelov pri različnih načinih podajanja poizvedb in parametrih. Pridobljene izkušnje in znanje nam bodo zagotovo koristile pri nadaljnjem delu, raziskavah in razvoju rešitev na tem področju.

Čeprav smo z rezultati raziskave zadovoljni, smo identificirali področja za izboljšave in nadaljnje delo. Prototip aplikacije lahko v prihodnosti integriramo v storitev Airdrop in s tem tudi uporabnikom storitve omogočimo razu-

mevanje notranjega delovanja podsistema za transformacijo med podatkovnimi modeli platforme DevRev in zunanjimi sistemi. V prihodnosti bi lahko raziskali tudi zmogljivosti drugih jezikovnih modelov izven družine GPT, kot so LLaMa, Claude in Gemini.

Literatura

[1] *Amazon Simple Storage Service: User Guide*. API Version 2006-03-01. [Pridobljeno 14. jan. 2024]. Amazon Web Services, Inc. Nov. 2023, str. 1, 5–8. URL: https://docs.aws.amazon.com/pdfs/AmazonS3/latest/userguide/s3-userguide.pdf.

[2] Rishi Bommasani in sod. *On the Opportunities and Risks of Foundation Models*. To delo je ponujeno pod licenco Creative Commons Attribution 4.0 International License. Podrobnosti licence so dostopne na spletni strani http://creativecommons.org/licenses/by/4.0/. 2022. arXiv: 2108.07258 [cs.LG].

[3] Tom B. Brown in sod. "Language models are few-shot learners". V: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS '20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[4] Uri Dekel in James D. Herbsleb. "Improving API documentation usability with knowledge pushing". V: *2009 IEEE 31st International Conference on Software Engineering*. 2009, str. 320–330. DOI: 10.1109/ICSE.2009.5070532.

[5] Samuel Gehman in sod. "RealToxicityPrompts: Evaluating Neural Toxic Degeneration in Language Models". V: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ur. Trevor Cohn, Yulan He in Yang Liu. Online: Association for

Computational Linguistics, nov. 2020, str. 3356–3369. DOI: 10.18653/v1/2020.findings-emnlp.301. URL: https://aclanthology.org/2020.findings-emnlp.301.

[6] Daniel Jurafsky in James H. Martin. *Speech and Language Processing (3rd ed. draft)*. [Pridobljeno 3. jan. 2024]. 2023. URL: https://web.stanford.edu/~jurafsky/slp3/.

[7] Ralph Kimball in Margy Ross. "The Data Warehouse Toolkit, 3rd Edition". V: 10475 Crosspoint Boulevard, Indianapolis, IN: John Wiley & Sons, Inc, 2013. Pogl. 19, str. 450–496. ISBN: 978-1-118-53080-1.

[8] Jenny T. Liang, Thomas Zimmermann in Denae Ford. "Understanding skills for OSS communities on GitHub". V: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. ¡conf-loc¿, ¡city¿Singapore¡/city¿, ¡country¿Singapore¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2022, str. 170–182. ISBN: 9781450394130. DOI: 10.1145/3540250.3549082. URL: https://doi.org/10.1145/3540250.3549082.

[9] Nelson F. Liu in sod. "Lost in the Middle: How Language Models Use Long Contexts". V: *Transactions of the Association for Computational Linguistics* 12 (feb. 2024), str. 157–173. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00638. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00638/2336043/tacl_a_00638.pdf. URL: https://doi.org/10.1162/tacl%5C_a%5C_00638.

[10] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[11] Alec Radford in sod. *Improving Language Understanding by Generative Pre-Training*. 11. jun. 2018. URL:

https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[12] Alec Radford in sod. *Language Models are Unsupervised Multitask Learners*. 14. feb. 2019. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[13] Freda Shi in sod. *Large Language Models Can Be Easily Distracted by Irrelevant Context*. 2023. arXiv: 2302.00093 [cs.CL].

[14] Ashish Vaswani in sod. "Attention is All you Need". V: *Advances in Neural Information Processing Systems*. Ur. I. Guyon in sod. Zv. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[15] *What is generative AI and what are its applications?* [Pridobljeno 3. jan. 2024]. 7. jun. 2023. URL: https://cloud.google.com/use-cases/generative-ai.