VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
STUDIJŲ PROGRAMA: INFORMATIKA

Programinio kodo generavimas naudojant LLM
Software code generation using LLM

Baigiamasis bakalauro darbas

Atliko: Aleksandras Juzumas
VU el. p.: aleksandras.juzumas@mif.stud.vu.lt

Vadovas: J. Asist., Dr. Igor Katin

Vilnius
2025

Santrauka

Pastaruoju metu vis daugiau dėmesio sulaukia nuolat tobulėjantys ir kuriami didieji kalbos modeliai, kurie iš esmės keičia mūsų kasdienybę. Viena dažniausiai sutinkamų sričių, kurioje yra plačiai naudojama ši technologija yra – programinio kodo generavimas. Vis dėlto, siekiant šias technologijas išnaudoti tinkamai, taip kad jos nepakenktų, o supaprastintų užduočių sprendimą, reikia pasirinkti tinkamus metodus. Šiuo darbu yra siekiama nustatyti geriausią kodo generavimo metodą, kuris yra pagrįstas didžiojo kalbos modelio naudojimu. Pirmiausia yra atliekama metodų analizė, nustatant kokie yra galimi metodų pasirinkimai. Tuomet pasirinkti metodai yra pritaikomi generuoti Java Spring Boot karkaso komponentus ir atliekamas bandymas juos generuojant. Galiausiai yra atliekamas vertinimas, kurio metu iš gautų rezultatų yra nustatoma, kuris iš pasirinktų metodų generuoja taisyklingiausią programinį kodą.

Raktiniai žodžiai: didieji kalbos modeliai, kodo generavimas, Java Spring Boot, dirbtinis intelektas, kodo generavimo metodai.

Summary

Recently, there has been an increasing focus on the continuous development and creation of large language models, which are fundamentally changing our daily lives. One of the most common areas where this technology is widely used is in the generation of software code. However, to make good use of these technologies, so that they do not harm and simplify tasks, the right methods must be chosen. This work aims to identify the best method of code generation, which is based on the use of the large language model. Firstly, an analysis of the methods is carried out, identifying what are the possible choices of methods. The selected methods are then applied to generate the components of Java Spring Boot framework, and an experiment is preformed to generate them. Finally, an evaluation is carried out to determine from the results which of the selected methods generates the most correct software code.

Keywords: large language models, code generation, Java Spring Boot, artificial intelligence, code generation methods.

Turinys

Santrauka 2
Summary 3
Įvadas 6
Teorinė dalis 8
1. Didieji kalbos modeliai 8
1.1. Kalbos modelių raida 8
1.2. Didžiųjų kalbos modelių apžvalga 9
2. Kodo generavimas naudojant didžiuosius kalbos modelius 10
2.1. Kodo generavimas naudojant LLM ir tarpine domeno kalba 10
2.2. Kodo generavimas naudojant LLM koordinuota paiešką 12
2.3. Kodo generavimas naudojant LLM ir užklausų inžinerija 14
3. Literatūros analizės apibendrinimas 15
Tiriamoji dalis 17
4. Metodika 17
4.1. Metodų kategorijos 18
4.2. Didžiojo kalbos modelio pasirinkimas 19
5. Metodų įgyvendinimas 19
5.1. Užklausų siuntimas didžiajam kalbos modeliui 19
5.2. Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas 20
5.3. Užklausų inžinerija pagrįstas metodas 22
5.4. Agentais pagrįstas autonominis metodas 23
6. Bandymas 25
6.1. Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas 25
6.2. Užklausų inžinerija pagrįstas metodas 26
6.3. Agentais pagrįstas autonominis metodas 28
7. Rezultatai 30

Išvados 31

Literatūros sąrašas.. 32

Iliustracijų sąrašas 33

Priedai 34

Įvadas

Tobulėjant technologijoms daugėja įvairių skirtingų įrankių, skirtų palengvinti kiekvieną dieną atliekamas užduotis ir darbus. Pastaruoju metu daug dėmesio yra skiriama įrankiams bei technologijoms, kurios naudoja didįjį kalbos modelį arba kitaip – dirbtinį intelektą. Viena iš sričių, kuriose plačiai yra naudojamas dirbtinis intelektas, yra programų kūrimas. Ištobulėjus didiesiems kalbų modeliams ši technologija buvo plačiai pradėta naudoti kodo generavime, kodo klaidų taisyme, bei dokumentacijos susisteminimui ir jos greitai paieškai. Ši technologija labai prisideda prie efektyvesnio programuotojų darbo. Tai yra, panaudojant dirbtinį intelektą kodo generavimui yra sutaupomas programuotojų darbo laikas, dėl to programuotojas gali atlikti daugiau užduočių arba jas įgyvendinti greičiau. Vien dėl šių priežasčių labai svarbu išanalizuoti, kaip didžiuosius kalbos modelius pritaikyti korektiškam ir optimaliam kodui generuoti. Nes taip pat kaip dirbtinis intelektas gali padėti ir pagreitinti visą darbo procesą, taip pat jis gali jį sulėtinti, generuodamas blogą, nekokybišką kodą, kurį vėliau gali reikėti taisyti.

Kuriant programas, naudojančias Java Spring Boot karkasą, dažnai pasitaiko daug pasikartojančių komponentų. Šių komponentų automatinis generavimas galėtų žymiai palengvinti ir pagreitinti programų kūrimo procesą, dėl to tai yra puiki vieta taikyti programinio kodo generavimo metodus, kurie naudoja didžiuosius kalbos modelius. Tačiau dažnu atveju generuojant programinį kodą su didžiaisiais kalbos modeliais yra susiduriama su problemomis. Viena pagrindinių problemų yra užtikrinimas, kad sugeneruotas kodas atliktų tai, kas yra prašoma – jog jis būtų optimalus ir grąžintų norimą rezultatą. Dažniausiai su šiomis užduotimis didieji kalbos modeliai susitvarko be jokių sunkumų, tačiau problema kyla, kai užklausa yra interpretuojama kitaip nei užklausėjas turėjo omenyje. Todėl sugeneruotas kodas neatitinka norimo rezultato ar neatlieka to, kas yra prašoma. Taip pat kyla problemų, kai sugeneruotas kodas būna bendrinis, kuris neatitinka projekto struktūros ar normų, todėl programuotojui vis tiek tenka kodą koreguoti.

Šio darbo tikslas yra nustatyti, kuris kodo generavimo metodas, naudojantis didžiuosius kalbos modelius, geba sugeneruoti taisyklingiausią Java Spring Boot karkaso komponentų programinį kodą.

Tikslui pasiekti reikia įgyvendinti šiuos tris uždavinius:

- išanalizuoti egzistuojančius metodus naudojančius didžiuosius kalbos modelius, skirtus generuoti programinį kodą;
- iš analizės pasirinkti tris programinio kodo generavimo metodus naudojančius didžiuosius kalbos modelius;
- identifikuoti ir suklasifikuoti tris skirtingus kodo generavimo būdus naudojančius didžiuosius kalbos modelius;

- atlikti tyrimą generuojant Java Spring Boot karkaso komponentus ir palyginti išrinktų metodų generavimo taisyklingumą.

Atliekant tyrimą generuojami Java Spring Boot dažniausiai naudojami komponentai, tokie kaip esybės (angl. *entity*), kontroleriai (angl. *controller*) ir atvaizdai (angl. *view*). Šie komponentai yra vieni pagrindinių Spring Boot karkaso elementų, kuriuos sujungus sukuriama žiniatinklio programa. Sugeneruotas kodas buvo lyginamas atsižvelgiant į sugeneruoto kodo aiškumą, struktūros korektiškumą ir gauto rezultato tikslumą. Tyrime buvo atsižvelgiama į tai, ar sugeneruotas kodas yra pilnai vykdomas be papildomo programuotojo įsikišimo ar jo pataisymo. Taip pat generuotas kodas buvo palyginimas su realiame projekte naudojamu kodu. Iš šio tyrimo buvo nustatyta, kuris metodas geriausiai atitinka keliamus lūkesčius, generuojant programinį kodą.

Teorinė dalis

1. Didieji kalbos modeliai

Dirbtinis intelektas vis labiau tampa kasdienės rutinos dalimi, keisdamas įvairių sričių įpročius. Didieji kalbos modeliai nuolatos tobulėja, taip atverdami naujas galimybes tekstų generavime ar jų vertime, supaprastindami programuotojų ar kitų sričių specialistų darbą, informacijos paieškoje ir daugybėje kitų sričių. Norint tikslingai panaudoti šias galimybes reikia būti susipažinus su šių technologijų istorija, veikimo principais ir perspektyvomis. Šiame skyriuje trumpai apžvelgsime kaip vystėsi kalbos modeliai nuo pirmojo kalbos modelio iki dabartinių didžiųjų kalbos modelių.

1.1. Kalbos modelių raida

Kalbos modelių raida vyko palaipsniui, pradedant nuo paprastų statistikos metodais remiančiais kalbos modelių ir baigiant pažangiais dabar plačiai naudojamais didžiaisiais kalbos modeliais, kurie geba apdoroti didžiulius kiekius duomenų ir suprasti kalbos kontekstą.

1 pav. Kalbos modelių raida chronologine tvarka

Pateiktoje sekoje (1 pav.) yra atvaizduojama kalbos modelių vystymosi raida. Remiantis šaltiniu [WCD24] vieni pirmųjų kalbos modelių buvo statistiniai kalbos modeliai. Šie modeliai buvo pradėti kurti dar 1990-taisiais metais. Pirmieji modeliai rėmėsi tikimybių taisyklėmis ir statistika, kurios leisdavo apskaičiuoti atitinkamų žodžių parinkimo tikimybę. Vienas dažniausiai naudotas statistinio kalbos modelio tipas buvo n-gram‘ų modelis, kuris skaičiuodavo kokia tikimybė kad žodis X eina po žodžio Y. Nors statistiniai kalbos modeliai gebėjo interpretuoti tekstą ir prognozuoti sekantį žodį, jų galimybės buvo ribotos – jie nesuprasdavo platesnio konteksto, o ilgos frazės nebesudarydavo logiškos žodžių sekos.

Po statistinių kalbos modelių sekė pažangesni neuroniniai kalbos modeliai. Šie modeliai naudojo dirbtinius neuroninius tinklus, kurie galėjo mokytis žodžių kontekstinius ryšius. Priešingai, nei statistiniai kalbos modeliai, šie modeliai gebėjo analizuoti sudėtingesnes struktūras, ir formuluoti teisingesnius sakinius. Vienas didžiausių šio etapo pasiekimų buvo Word2Vec algoritmas, kuris paversdavo žodžius į vektorius, taip leisdamas nustatyti jų semantinius ryšius.

Laikui bėgant buvo pradėti kurti iš anksto apmokyti kalbos modeliai. Tai buvo didelis žingsnis į priekį, kadangi šie modeliai buvo iš anksto apmokomi su dideliais duomenų kiekiais ir pritaikomi konkrečiose srityse, konkrečioms užduotims spręsti. Vienas pirmųjų tokių sėkmingų modelių buvo BERT modelis, kuris naudojo naują Transformer architektūrą. Ši architektūra leido modeliams analizuoti vienu metu visą sakinio struktūra, o ne atskirus žodžius.

Galiausiai buvo prieita prie didžiųjų kalbos modelių, kurie yra naudojami šiomis dienomis. Vieni geriausiai žinomų tokių modelių yra plačiai naudojami GPT-3 ar GPT-4. Didieji kalbos modeliai pasižymėjo gebėjimu generuoti aukštos kokybės tekstus, generuoti programinį kodą, analizuoti pateiktus duomenis be specialaus apmokymo. Šie modeliai pagrinde naudoja didelius duomenų kiekius, kurie gali siekti net kelis šimtus terabaitų, ir dideliu kiekiu neuroninio tinklo parametrų. Šių kalbos modelių panaudojimo sritys yra beribės.

1.2. Didžiųjų kalbos modelių apžvalga

Didieji kalbos modeliai tapo neatsiejama kiekvieno gyvenimo dalis. Jų naudojimas atveria naujas galimybes ieškant idėjų, atliekant įvairaus tipo darbus, bei palengvina skirtas užduotis. Nors jų panaudojimas yra labai paprastas, svarbu suprasti kas tai yra iš esmės ir koks yra jų veikimo principas ir kuo išsiskiria šios technologijos.

Didieji kalbos modeliai yra pažangūs dirbtinio intelekto modeliai, kurie geba suprasti, interpretuoti ir generuoti natūralų tekstą, kuris prilygsta žmogaus kalbai. Šie modeliai išsiskiria būtent tuo, kad jie geba mokytis iš didžiulių duomenų rinkinių, kurie apima įvairias sritis ir kontekstą. Būtent dėl šios savybės jie turi platų ir universalų žinių spektrą, kuris gali būti pritaikomas įvairioms sritims. Remiantis šaltiniu [MM25] didieji kalbos modeliai yra iš anksto apmokyti ir priklauso iš anksto apmokytų modelių grupei. Tokių modelių pavyzdžiai būtų GPT-3 ir GPT-4, kurie demonstruoja aukšto lygio kalbos supratimą ir tekstų generavimą.

Pagrindinis didžiųjų kalbos modelių veikimo principas remiasi transformerių architektūra (angl. Transformer architecture). Kaip straipsnyje [WCD24] yra aiškinama, transformerių architektūra remiasi savi-dėmesio (angl. self-attention) mechanizmu, kuris leidžia efektyviai fiksuoti priklausomybes tarp žodžių ir sakinių, lygiagrečiai atliekant skaičiavimus, tokiu būdu užtikrindamas ganėtinai tikslų konteksto supratimą ir kokybišką teksto generavimą. Ši architektūra pasižymi tuo, kad modeliams leidžia efektyviai ir greitai apdoroti informaciją, taip suteikdamas galimybe generuoti prasmingą tekstą atsižvelgiant į visą pateiktą informaciją.

Didieji kalbos modeliai pasižymi keliomis pagrindinėmis savybėmis, dėl kurių išsiskiria iš kitų dirbtinio intelekto sistemų. Viena pagrindinių savybių kuria pasižymi didieji kalbos modeliai yra gebėjimas greitai prisitaikyti prie įvairaus sudėtingumo užduočių, be papildomo specialaus

apmokymo. Šios savybes yra įvardijamos [MM25] kaip zero-shot arba few-shot, kurios atitinkamai reiškia be jokio papildomo apmokymo ir su minimaliu papildomu apmokymu, mokymai. Taip pat svarbi savybė kuria pasižymi šie modeliai yra išorinių žinių šaltinių naudojimas, kuris leidžia dar labiau prisitaikyti prie užduočių, taip didinant pritaikomumą ir efektyvumą.

Dabar turint bazinį supratimą, kas yra didieji kalbos modeliai ir kokie yra jų veikimo principai galime detaliau nagrinėti šių modelių panaudojimo atvejus, kuris šiuo atveju šiame darbe yra programinio kodo generavimas.

2. Kodo generavimas naudojant didžiuosius kalbos modelius

Didieji kalbos modeliai geba atlikti įvairias užduotis, tokias kaip kodo generavimas ar tekstų rašymas. Šis gebėjimas atrodo kaip didelis privalumas, tačiau šis universalumas neretu atveju tampa iššūkiu. Dėl šio universalumo, didieji kalbos modeliai geba sugeneruoti daug skirtingų variantų ir tai kelia daug klausimų, tokių kaip: kuris iš variantų tinkamiausias, kuo jie skiriasi ir ar visi yra teisingi. Taip pat dar viena svarbi problema yra didžiųjų kalbos modelių gebėjimas dirbti su mažiau paplitusiomis technologijomis. Nors šie modeliai yra apmokomi su milžiniškais duomenų rinkiniais, bet dažniausiai tai būna duomenys apie plačiai paplitusias ir naudojamas technologijas. Dėl to, kai užklausa yra užduodama apie tokias technologijas, modelis gali pateikti netikslius arba minimalistinius atsakymus.

Su šiomis problemomis susiduriama ir atliekant kodo generavimo užduotis. Kiekvienas projektas pasižymi unikaliomis detalėmis, naudojamomis technologijomis ir kodo rašymo struktūros susitarimais. Taigi didieji kalbos modeliai, neturėdami konkretaus konteksto apie projekto savybes, negali tiksliai sugeneruoti kodo, kuris atitiktų standartus. Dėl šios priežasties programuotojas turi papildomai skirti laiko rankiniu būdų pritaikant sugeneruotą kodą, kad jį būtų galimą panaudoti, taip pat taisyti kylančias problemas, kas sumažiną pagrindinę kodo generavimo automatizavimo esmę.

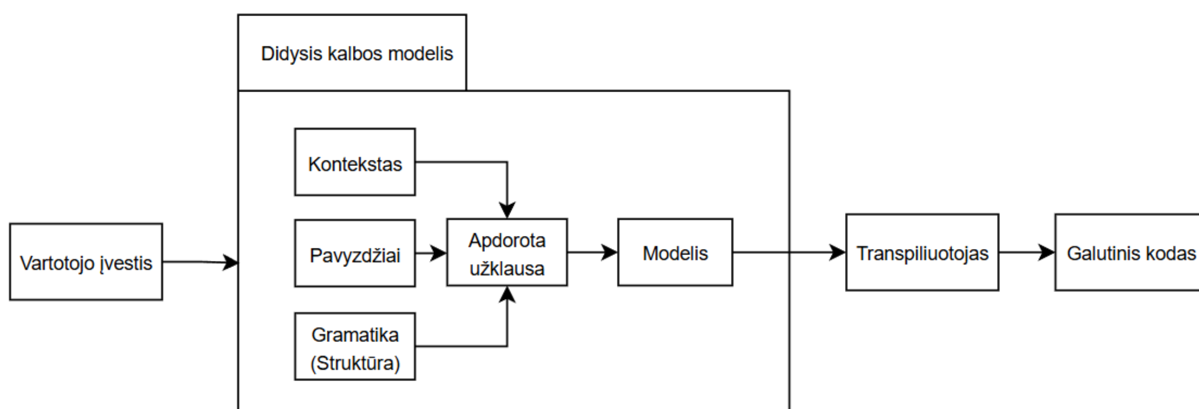
Šiame skyriuje tyrinėsime kaip kiti mokslininkai bando spręsti šias problemas ir kokie egzistuoja metodai pagerinantys kodo generavimo automatizavimo kokybę.

2.1. Kodo generavimas naudojant LLM ir tarpine domeno kalba

Kaip ir buvo minėta anksčiau viena iš dažniausių problemų su kuria yra susiduriama generuojant efektyvų ir teisingą programinį kodą yra didžiųjų kalbos modelių žinių trūkumas su mažiau paplitusiomis programavimo kalbomis ar technologijomis ir konkrečios užduoties konteksto trūkumas. Straipsnyje [KCW25] autorius labai aiškiai įvardija šią kylančia problemą ir jos priežastį, teigdamas, kad didieji kalbos modeliai yra apmokomi naudojant didžiulius kodavimo duomenų rinkinius, kuriuose yra gausu informacijos apie plačiai paplitusias ir dažnai naudojamas technologijomis. Tačiau kuo užduotis yra specifiškesnė ar naudojama nepopuliari programavimo

kalba, tuo kodo generavimas naudojant didžiuosius kalbos modelius yra prastesnis. Dėl šios priežasties, net didžiausi kalbos modeliai tokie kaip GPT-4 ar CodeLlama sunkiai susitvarko su tokiomis užduotimis.

Šiai problemai spręsti autorius straipsnyje [KCW25] siūlo metodą, kuris remiais tarpine domenui specifine kalba, tai yra JSON-DSL, kurio atlieka tarpininko vaidmenį, tarp pradinės vartotojo įvesties ir galutinio kodo. Pagrindinė šio JSON-DSL kaip tarpininko esmė yra, paversti vartotojo užklausą į griežtai apibrėžta struktūra, kuri vėliau yra transpiliuojamas, kitaip tariant vienos kalbos kodas, yra paverčiamas į kitos kalbos kodą. Transpiliavimui yra naudojamas paprastas šabloninis transpiliuotojas, tai yra gautą JSON kodą, transpiliuotojas pritaiko ir užpildo šabloną.



2 pav. Supaprastinta JSON-DSL generavimo schema naudojant didįjį kalbos modelį

Schemoje (2 pav.) pavaizduota supaprastinta tarpinio domenui specifinio modelio generavimo schema. Joje matome kad vartotojas pateikia įvesti, kuri yra apdorojama pridedant kontekstą, norimo gauti rezultato pavyzdžius ir kokia struktūra turėtu būti pateiktas atsakymas. Taip pat schemoje nėra pavaizduota, tačiau šiame procese dalyvauja ir validatorius, kuris validiuoja ar didžiojo kalbos modelio sugeneruotas JSON atitinka reikalavimus. Gautas JSON kuris atitinka reikalavimus yra perduodamas transpiliatoriui, kuris tada jau JSON kodą paverčia į reikiamos programavimo kalbos programinį kodą.

JSON kodo generavimas šiame metode yra pasirinktas ne atsitiktiniu būdų. Kaip autorius straipsnyje [KCW25] aiškina, JSON kodas yra plačiai naudojamas ir lengvai suprantamas didžiųjų kalbos modelių, todėl nereikia papildomo apmokymo. Taip pat dėl paprastos JSON struktūros, nesunku aprašyti visus norimus elementus, ir patikrinti JSON‘o korektiškumą.

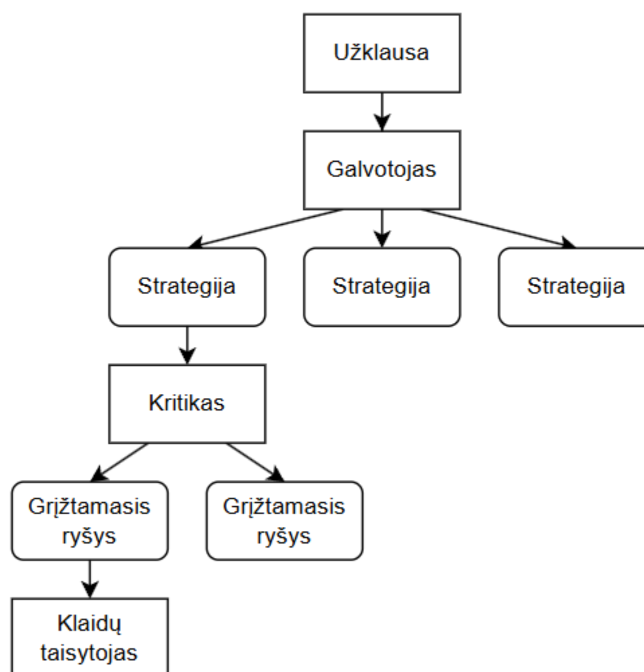
Šio metodo efektyvumą autorius straipsnyje [KCW25] pademonstruoja atlikdamas porą bandymų. Pirmasis bandymas buvo atliktas bandant sugeneruoti kodą su MiniZinc programavimo kalba, kuri yra skirta optimizavimo užduotims spręsti. Šio bandymo metu autoriaus teigimu, kodo generavimo tikslumas naudojant JSON-DSL kaip tarpininką išaugo nuo 48% iki 64%. Antrasis

bandymas buvo atliktas su specifine geležinkelių infrastruktūros testavimo kalbas – BTTL. Šio bandymo metu, gauti rezultatai dar labiau džiugino. Kodo tikslumas išaugo nuo 46% iki 72%. Šių bandymų rezultatai rodo, kad naudojant tarpinį sluoksnį, kodo generavimo tikslumas ir efektyvumas išauga daugiau nei 20%.

Apibendrinant straipsnyje [KCW25] siūlomą metodą matome kad visa esmė slypi tarpininke, kurį sugeneruoja didysis kalbos modelis. Metodas pasižymi savo paprastumu, kadangi naudoją paprastai suprantamą JSON kodą, kurį nesunku interpretuoti tiek didiesiems kalbos modeliams, tiek žmogui. Taip pat šio metodo demonstruojamas tikslumo prieaugis verčia stebėtis.

2.2. Kodo generavimas naudojant LLM koordinuota paiešką

Išpopuliarėjus didiesiems kalbos modeliams, mokslininkai pradėjo nagrinėti būdus kaip šiuos modelius išnaudoti kuo efektyviau. Prieš tai esamame skyriuje apžvelgėme jau vieną iš galimų metodų. Nors šis metodas ir parodė, kad tikslumas išaugo ženkliai, tačiau kad jis veiktų reikia papildomų dalių, tokių kaip transpiliatorių. Didžiųjų kalbos modelių koordinuota paieška būtent išsprendžia šią problemą, kadangi nereikia papildomų modulių, norint sugeneruoti programinį kodą, kadangi kiekvienas agentas atlieka skirtinga savo funkcija. Apie šį metodą straipsnyje [RWK+23] buvo rašoma kaip vizija, kurią būtu galimą įgyvendinti. Autoriaus teigimu, toks daugiaagentinių sistemų naudojamas tik padidinta didžiųjų kalbos modelių galimybes sprendžiant užduotis.



3 pav. Supaprastinta koordinuotų didžiųjų kalbos modelių paieškos metodo schema

Straipsnyje [RWK+23] pristatomo metodo vizijos esmė yra kelių agentų tarpusavio komunikacija bandant generuoti kodą, o schemoje (3 pav.) pateikiama supaprastinta šio metodo veikimo schema.

Autorius pristato idėja, kurioje kiekvienas agentas yra atsakingas už savo funkciją, pavyzdžiui: strategijų kūrimą, vertinimą ar klaidų aptikimą. Tai reiškia kad tokio tipo sistema susideda iš keliolikos skirtingų agentų kurie atsakingi už skirtingus programinio kodo generavimo etapus, taip pašalinant papildomą žmogaus įsikišimą generuojant efektyvų kodą. Taigi naudojant tokias sistemas tikimasi, kad sistema sugeneruos aukšto lygio programinį kodą efektyviau, taip sutaupydama programavimo laiką.

Taip pat straipsnyje [RWK+23] buvo atliktas trumpas bandymas. Bandymo metu buvo sukonstruota sistema, kurioje dalyvauja dvylika agentų. Tokiai sistemai tada buvo duota paprasta užklausa – „Write a code for a snake game". Bandymas buvo atliktas tris kartus su skirtingais sistemoje naudojamais didžiaisiais kalbų modeliais. Pirmo bandymo metu buvo naudojamas GPT-3 modelis. Uždavus užklausą ir gavus rezultatą, kad žaidimas veiktų reikėjo žmogaus įsikišimo. Antruoju bandymu buvo naudotas modelis GPT-4. Šio bandymo rezultate buvo pateiktas kodas kurį buvo galima vykdyti iškarto po paleidimo, tačiau neturėjo vartotojo sąsajos. Trečiuoju bandymu buvo naudojamas tas pats GPT-4 modelis, tik su truputį pakoreguota sąlyga – „develop a snakegame with GUI". Atlikus šį bandymą, sistema pateikė pilnai veikiantį žaidimą, kurio nereikėjo taisyti.

Atsižvelgiant į gautus bandymo rezultatus, matome kad tokia daugiaagentine sistema nesunkiai susitvarko su programinio kodo kūrimo užduotimis be jokio papildomo žmogaus įsikišimo. Žinoma kuo didesnis yra didysis kalbos modelis, tuo rezultatas yra tikslesnis. Taip pat pateikus kuo tikslesnę užklausą, galima tikėtis tikslesnio rezultato.

Kaip akcentavau anksčiau kad straipsnyje [RWK+23] koordinuota didžiųjų kalbos modelių paieška pristatoma kaip vizija, tačiau yra mokslininkų, kurie šį metodą yra įgyvendinę. Vienas iš įgyvendinimų yra pristatoma straipsnyje [LLZ+24], kuriame kalbama apie kelių agentų panaudojimą generuojant tikslų programinį kodą. Šis straipsnyje [LLZ+24] pristatomas metodas yra patobulintas ir išsiskiria nuo prieš tai straipsnyje [RWK+23] minėtos vizijos tuo, kad sprendimo paieškai metodas naudoja medžio struktūra. Tai yra originali užklausa yra medžio šaknis ir iš jos išeina keletas skirtingų sprendimo strategijų, iš kurių seka kodo vertinimas ir jo tobulinimas, taip atrenkant geriausią ir tiksliausia rezultatą.

Šis straipsnyje [LLZ+24] pristatomas patobulintas metodas pasižymi tuo, kad jis nenaudoja didelio kiekio agentų, priešingai nei buvo pristatoma vizijoje. Tai yra, metode yra naudojami tik 4 esminiai agentai, kurie iš jų yra „Galvotojas" (angl. Thinker), „Sprendėjas" (angl. Solver), „Klaidų taisytojas" (angl. Debuger) ir galiausiai „Kritikas" (angl. Critic), Šie agentai komunikuodami tarpusavyje sprendžia problema. Pirmiausiai „Galvotojas" sugeneruoja strategija, kuri yra pateikta anglų kalba kaip tekstas, po to „Sprendėjas" sugeneruoja programinį kodą, tada prisijungia kritikas, kuris vertina ar sugeneruotas kodas atitinka standartus, ar galimi teisingi rezultatai ir duoda grįžtamąjį

ryšį, „Klaidų taisytojui“ ar strategija vystyti toliau, atšaukti ar tobulinti. Taip pat prie visa to, galima prijungti ir programą, kuri kodą gali ištestuoti realiu laiku, tai yra paleisti ir patikrinti testų scenarijus.

Straipsnyje [LLZ+24] autoriai pateikia gautus rezultatus atlikus generuoto programinio kodo vertinimus. Imant vidurkį buvo pastebėta, kad nuo prasčiausiai pasirodžiusio metodo, kodo generavimas naudojant medžio struktūrą ir koordinuota agentų paieška skyrėsi apie 20%. Tai reiškia, kad imant 100 užduočių, šis siūlomas metodas, pateiktų 20 teisingų atsakymų daugiau, nei kitas metodas.

Bendrai tariant, koordinuoti didžiųjų kalbos modelių paieškos metodai yra itin perspektyvūs. Šie metodai parodo, kokia galia slypi tarp kelių tarpusavyje komunikuojančiu agentų. Nors šis metodas reikalauja daugiau resursų ir yra sunkiau įgyvendinamas, tačiau jo demonstruojami gebėjimai pranoksta lūkesčius.

2.3. Kodo generavimas naudojant LLM ir užklausų inžinerija

Vienas dažniausiai sutinkamų metodų tai yra užklausų inžinerija arba kitaip specifinių užklausų konstravimas. Praktiškai žiūrint net prieš tai aptarti ir nagrinėti metodai daugiau ar mažiau naudoją šią metodiką. Nors iš pirmo žvilgsnio užklausų inžinerija gali atrodyti kaip gana paprastas metodas, tačiau jis gali turėti gausybę skirtingų įgyvendinimo, bei pritaikymo būdų, bei turėti daug įtakos generuojant tiksliam ir teisingam kodui. Šiuo metodu yra sprendžiama viena didžiausių spragų kylančių generuojant programinį kodą su didžiaisiais kalbos modeliais, tai yra konteksto trūkumas, ar papildomų duomenų suteikimas didžiajam kalbos modeliui. Vien dėl šių galimybių užklausų inžinerija yra laikomas kaip atskiras metodas ir yra nagrinėjamas atskirai.

Kaip jau minėjau anksčiau, viena dažniausiai pasitaikančių problemų yra didžiųjų kalbos modelių informacijos trūkumas apie projekto specifiką ir jame jau esamas klases, funkcijas ar kintamuosius. Straipsnyje [ZFS+24] autoriai siūlo šios problemos sprendimo būdą, pristatydami savo sistemą CREAM. Autorių teigimu, šios sistemos pagrindinė idėja yra praplėsti pradinę užklausą papildomi kontekstu, kuris yra gaunamas pasitelkus panašaus kodo paieška tarp pritaikytų duomenų rinkinių ir testavimo rezultatais. Tokiu būdų yra bandoma atkartoti įprastą programuotojo elgseną kuriant sistemą, tai yra problemos supratimą, panašaus kodo paiešką ir testavimą. Dar viename straipsnyje [PHX+24] taip pat yra siūloma panaši sistema, kurios pagrindinė idėja paremta užklausų inžinerija. Šios sistemos pagrindinis skirtumas yra tas, kad vietoje to kaip straipsnyje [ZFS+24] siūlomoje sistemoje yra naudojami platūs pritaikyti programinio kodo rinkiniai, šio straipsnio [PHX+24] siūlomoje sistemoje, kodo paieška yra vykdoma tarp pačio projekto failų ir struktūros.

Kaip jau supratome iš straipsnių [ZFS+24, PHX+24] užklausų inžinerijos metode visa esmė slypi už specifiškai transformuotų užklausų sudarymo. Pateiktame paveiksliuke (4 pav.) yra pateikiamas vienas iš tokių užklausų pavyzdžių. Joje matome pradinę užduota užklausą, kuri gali būti dalinai arba visiškai transformuota. Prie jos yra prijungiamas surenkamas kontekstas iš duomenų rinkinių, ar tai būtų paieška internete ar tarp pačio projekto failų. Kontekste gali būti pateikiamos klasės, klasių laukai ir metodai. Po papildomo konteksto yra pateikiamas generuojamos funkcijos ar klasės aprašymas naudojamos programavimo kalbos sintakse.

```
# Request
You are given a blank Java function with context, similar code snippets
and function declaration. Your goal is to create function that maps view
to entity

# Context
public class Entity {
    # FIELDS
    ...
    # METHODS
    ...
}
    # OTHER CLASSES
    # SIMILAR METHODS

# Method declaration
public Payment populateEntity(Entity entity, EntityView view) { }
```

4 pav. Sukonstruotos užklausos pavyzdys

Šis metodas iš pažiūros gali pasirodyti vienas paprasčiausių. Tačiau yra atliekama daug bandymų ir yra sukurta daug šaltinių, kuriuose jau yra aprašytos išbandytos užklausos, kurios gali pagerinti didžiųjų kalbos modelių generavimo tikslingumą. Kaip ir minėjau anksčiau, šis metodas dalinai yra naudojamas visose metoduose, kadangi visi procesai prasideda nuo gerai suformuluotos užklausos. Taip pat vien dėl geresnio užklausos transformavimo ir sudarymo yra kuriamos įvairios sistemos, tokios kaip straipsniuose [ZFS+24, PHX+24], kurios sumažina rankinį darbą ieškant tinkamo konteksto.

3. Literatūros analizės apibendrinimas

Išrinkus šiuos tris metodus galime juos apibendrinti. Visi šie metodai atlieka tą pačią funkciją ir pasižymi viena bendra savybe – naudojant šiuos metodus, didžiojo kalbos modelio nereikia

papildomai specialiai apmokyti. Apačioje pateiktoje lentelėje (1 lentelė) glaustai yra pateikiami kiekvieno metodo privalumai, trūkumai, įgyvendinimo sudėtingumas ir tikslumas.

1 lentelė. Metodų savybės

	Privalumai	Trūkumai	Įgyvendinimo sudėtingumas	Generuojamo kodo tikslumas
LLM su tarpine domeno kalba	Generuoja griežtai apibrėžtos struktūros tarpinį JSON, pagal kurį konstruojamas galutinis kodas, mažindamas klaidų tikimybę ir gerindamas kodo kokybę	Sugeneruoto tarpinio JSON konvertavimui į galutinį kodą reikalingi papildomi moduliai, kurie konvertuoja JSON į galutinio kodo komponentą	Vidutinis – užtenka dviejų komponentų: tarpinio JSON generavimo ir JSON konvertavimo į galutinį programinį kodą	Taikant šį metodą, pagal straipsnyje [KCW25] atliktą tyrimą, nustatyta, kad kodo teisingumas išauga daugiau nei 20%
LLM koordinuota paieška	Koordinuotų agentų paieška padidina sprendimo galią, atskiriant kiekvieno agento darbą taip suteikiant galimybe spręsti sudėtingesnes užduotis	Kompleksiška sistema, reikalaujanti daug priežiūros skirtingų agentų suderinimui ir koordinavimui	Sudėtingas – reikia daug skirtingų komponentų: kiekvienas agentas įgyvendinamas kaip atskiras komponentas, taip pat reikia komunikacijos komponento	Taikant šį metodą, pagal straipsnyje [LLZ+24] atliktą tyrimą, nustatyta, kad šis metodas pateikia 78,9% teisingų atsakymų.
LLM užklausų inžinerija	Konstruojant specialias užklausas, pateikiant papildomą kontekstą yra užtikrinama kad sugeneruotas kodas atitiks projekte naudojama struktūrą	Didelė priklausomybė nuo pateikiamo konteksto, blogas kontekstas generuoja blogą kodą	Vidutinis – užtenka dviejų komponentų: kontekstinio kodo ištraukimo ir užklausos konstravimo komponentų	Taikant šį metodą, pagal straipsnyje [PHX+24] atliktą tyrimą, nustatyta, kad generuojamo kodo teisingumas padidėja iki 17% generuojant Java kodą.

Tolimesniuose šio darbo etapuose, suskirstysime šiuos tris pasirinktus metodus, skirtus programinio kodo generavimui į atitinkamas kategorijas. Po to bus atliekamas tyrimas, kuriame generuojami Java Spring Boot karkaso komponentai, ir nustatoma, kuris iš šių metodų grąžina tiksliausius, mažiausiai klaidų turinčius rezultatus. Ši literatūros analizė buvo naudojama kaip pagrindas nustatyti, kokie egzistuoja metodai.

Tiriamoji dalis

4. Metodika

Kaip jau buvo užsiminta darbo pradžioje, pagrindinis šio darbo tikslas yra palyginti skirtingus programinio kodo generavimo metodus, naudojančius didžiuosius kalbos modelius, generuojant Java Spring Boot karkaso komponentus. Siekiant nenukrypti nuo šio tikslo tiriamojoje dalyje, buvo suformuluotas pagrindinis tiriamosios dalies uždavinys – atrinkti keletą skirtingų, įvairiose sistemose sutinkamų, objektų ir kiekvienam iš jų sugeneruoti atitinkamus komponentus, pasitelkiant išsirinktus metodus.

Nors iš literatūros analizės pasirinkti metodai yra orientuoti į įvairių programavimo kalbų programinio kodo generavimą, toliau šiame darbe pasirinkta dėmesį skirti konkrečių – Java Spring Boot karkaso – komponentų generavimui.

Rezultatams pasiekti buvo generuojami tokie komponentai:

- esybė (angl. *entity*);
- specifinis esybės atvaizdas, naudojantis projekcijas (angl. *view with projections*);
- kontroleris (angl. *controller*).

Šie komponentai pasirinkti todėl, kad jie sudaro pagrindą kuriant WEB aplikacijas, naudojančias Java Spring Boot karkasą. Kiekvienas sugeneruotas komponentas atspindi tam tikrus objektus, pavyzdžiui: naudotojas (angl. *user*), apmokėjimas (angl. *payment*), paraiška (angl. *application*) ar autentifikavimas (angl. *authentication*). Kiekvienu metodu buvo generuojama po 100 aukščiau minėtų komponentų, tai yra 300 komponentų su vienu metodu, o bendrai buvo sugeneruota 900 komponentų.

Sugeneruoto kodo taisyklingumas buvo vertinamas peržiūrint sugeneruotą kodą ir atrenkant blogus atvejus. Blogu atveju buvo skaitomas toks sugeneruotas kodas, kuris:

- yra neveikiantis – panaudojus kodą, programa nesikompiliuoja arba iššaukia klaidas;
- turi netikslumų – kodas neišpildo užklausoje esančių reikalavimų, arba kodas kompiliuojasi, tačiau yra netikslumų, pavyzdžiui: netinkamas kintamojo ar metodo grąžinamos reikšmės tipas, nenaudojamos anotacijos, kode yra „haliucinacijų“.

Taip pat buvo vertinamas ir generavimo greitis. Jis buvo vertinimas kokybiškai, atsižvelgiant į tai, kiek laiko trunka sugeneruoti programinį kodą. Buvo sudaryta tokia vertinimo skalė:

- greitas – kodas sugeneruojamas beveik iš karto;
- vidutinis – sugeneruoto kodo reikia palaukti, tačiau laukimo laikas yra priimtinas;
- lėtas – sugeneruoto kodo reikia laukti ilgai, jaučiasi laukimo laikas.

4.1. Metodų kategorijos

Sekantis žingsnis yra kiekvieną metodą suskirstyti į kategorijas. Visi išrinkti metodai gali pasirodyti panašūs vienas į kitą, tačiau visi pasižymi bent viena detale kuri išskiria metodą į atitinkamą kategorija. Pavyzdžiui užklausų inžinerija yra naudojama praktiškai kiekviename metode, tačiau tokie metodai turi savo atskirą kategoriją.

Iš literatūros analizės buvo išsirinkta trys skirtingi metodai, šie metodai bus skirstomi į tokias kategorijas:

- Tarpinių šablonų generavimu pagrįsti kodo generavimo metodai;
- Agentais pagrįsti autonominiai kodo generavimo metodai;
- Užklausų inžinerija pagrįsti kodo generavimo metodai.

Šios kategorijos buvo parinktos atsižvelgiant į kiekvieno nagrinėto metodo išskirtinę savybę. Pirmasis nagrinėtas metodas, kuriame yra naudojama tarpinis JSON kalbos šablono generavimas yra priskiriamas pirmajai kategorijai - tarpinių šablonų generavimu pagrįsti kodo generavimo metodai, kadangi šio metodo išskirtinė savybė yra pateikto šabloninio JSON sugeneravimas ir jo perdarymas į kitos programavimo kalbos kodą. Antrasis metodas, kuriame naudojami didžiųjų kalbos modelių agentai, kurie tarpusavyje komunikuodami generuoja programinį kodą, yra priskiriamas antrajai kategorijai - agentais pagrįsti autonominiai kodo generavimo metodai. Ir galiausiai paskutinis nagrinėtas metodas, kuris remiasi specifiškai sukonstruotomis užklausomis, kuriuo kaip ir minėjau remiasi daugelis kitų metodų, tačiau yra išskiriamas į atskirą savo kategoriją - užklausų inžinerija pagrįsti kodo generavimo metodai.

2 lentelė. Kategorijų palyginimo lentelė

Kategorija	Aprašymas	Privalumai	Trūkumai
Tarpinių šablonų generavimu pagrįsti metodai	Didieji kalbos modeliai užpildo šabloną ir jis konvertuojamas į norimą programavimo kalbą	• Nesunku validuoti • Suprantama struktūra • Užtikrinama generuojamo kodo struktūra	• Reikia papildomo modulio • Mažai lankstumo
Agentais pagrįsti autonominiai metodai	Keli didžiųjų kalbos modelių agentai bendradarbiauja tarpusavyje generuodami kodą.	• Daug savarankiškumo • Geba generuoti sudėtingas užduotis • Lankstus	• Sudėtinga sistema • Reikalingi dideli resursai įgyvendinant sistemą
Užklausų inžinerija pagrįsti metodai	Didieji kalbos modeliai generuoja kodą pateikiant parengtas specifines užklausas.	• Greitai įgyvendinamas • Lengvai pritaikomas • Nebūtinas papildomas programavimas	• Daug priklausomybės nuo užklausos kokybės • Nenuspėjamas rezultatas

Aukščiau pateikta palyginimų lentelė (2 lentelė), kurioje yra pateiktas trumpas aprašymas apie kategoriją, jos trūkumus ir privalumus. Kaip matome pirmosios ir trečiosios kategorijos metodai pasižymi savo paprastumu, priešingai nei antrosios kategorijos metodai. Tačiau antrosios kategorijos metodai pasižymi savo lankstumu, ko trūksta pirmosios kategorijos metodams. Būtent šios detalės kiekvieną kategorija padaro unikalia.

Suskirsčius metodus į kategorijas, buvo pradėta įgyvendinti šiuos metodus. Tolimesniuose skyreliuose yra trumpai aprašoma kai buvo įgyvendinti kiekvienas iš metodų ir kokie komponentai buvo sukurti įgyvendinant metodus.

4.2. Didžiojo kalbos modelio pasirinkimas

Įgyvendinant šiame darbe pasirinktus metodus, buvo pasirinktas naudoti GPT-4o-mini didysis kalbos modelis, sukurtas bendrovės „OpenAI“. Šis modelis pasižymi aukštu tikslumu, maža paklaida ir galimybė kurti ne tik natūraliosios kalbos atsakymus, bet ir struktūrizuotus atsakymus. Taip pat naudojamas modelis nebuvo specialiai apmokytas spręsti programavimo uždavinius.

3 lentelė. GPT-4o-mini modelio apžvalgos lentelė

Modelis	GPT-4o-mini
Kūrėjas	„OpenAI“
Maksimalus konteksto ilgis	128 tūkst. ženklų
Panaudos sritys	• Kodų generavimas • Analizavimas
Privalumai	• Atsakymų tikslumas • Didelis konteksto ilgis

5. Metodų įgyvendinimas

Toliau esančiuose skyreliuose yra aprašomas detalus metodų įgyvendinimo procesas. Prie kiekvieno metodo yra pateikiamas atitinkamų dalių programinis kodas. Taip pat yra pabrėžiamos problemos, kurios buvo pastebėtos įgyvendinant metodus.

5.1. Užklausų siuntimas didžiajam kalbos modeliui

Komunikavimui su didžiuoju kalbos modeliu buvo naudojama Python programavimo kalbos biblioteka – „openai“. Jos pagalba buvo galima siųsti užklausas bei gauti atsakymus iš didžiojo kalbos modelio. Taip pat šios bibliotekos pagalba galima kontroliuoti modelio veikimą keičiant parametrus, tokius kaip atsitiktinumą, ar atsakymo ilgį.

Pateiktame kodo fragmente yra pavaizduojamas užklausos siuntimo didžiajam kalbos modeliui pavyzdys (5 pav.). Užklausoje yra nurodomas naudojamas modelis, siunčiamas pranešimas, kuris susideda iš dviejų dalių (sisteminės ir naudotojo) bei modelio nustatymai.

```python
import openai

openai.api_key = "YOUR_API_KEY"

def call_llm(user_prompt: str, system_prompt: str):
    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.2,
        max_tokens=1024
    )
    return response["choices"][0]["message"]["content"]
```

5 pav. Užklausos siuntimo didžiajam kalbos modeliui kodo pavyzdys.

Tokia struktūra buvo taikoma kreipiantis į didįjį kalbos modelį, įgyvendinant kiekvieną iš pasirinktų metodų. Vieninteliai skirtumai buvo siunčiamų pranešimų turinys ir atsitiktinumo rodiklis, atsižvelgiant į tai, kiek atsitiktinumo buvo reikalaujama prie kiekvieno metodo.

5.2. Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas

Pirmasis metodas, kuris buvo įgyvendintas, yra metodas, kuris generuoja JSON kalbos užpildytą šabloną ir jį konvertuoja į norimos programavimo kalbos kodą. Pradėkime nuo to, kad šis metodas susideda iš dviejų dalių, tai yra JSON kalbos šablono generavimo ir konvertavimo į norimą programavimo kalbą.

Pirmasis žingsnis, kuris buvo atliktas, buvo šablono sukonstravimas, kuris bus pateikiamas užklausoje didžiajam kalbos modeliui. Ši struktūra nėra sudėtinga, ir susideda iš laukų, kurie yra naudojami tame komponente. Iš viso buvo sukurta trys skirtingos tokios struktūros, skirtos generuoti esybę, esybės atvaizdą ir kontrolerį.

Pateiktame pavyzdyje (6 pav.) yra pateiktas JSON kalbos šablonas, kuris yra skirtas generuoti esybės atvaizdą (angl. *view*). Šablone yra pateikiami visi reikiami metaduomenys, kad būtų galima generuoti komponentą. Šablonas, kaip matoma, susideda iš bazinių laukų, tokių kaip komponento tipas, ar klasės pavadinimas, ir kitų laukų, kurie yra aktualūs komponentui. Turint tokį šabloną, jį galima pridėti prie siunčiamos užklausos, kad jis būtų užpildytas.

```json
{
  "type": "LIST_DTO",
  "name": "string",
  "annotations": ["string"],
  "fields": [
    {
      "name": "string",
      "type": "string",
      "annotations": ["string"],
      "source": "string"
    }
  ],
  "projectionProvider": {
    "projections": ["string"],
    "resultClass": "string"
  }
}
```

6 pav. JSON šablono pavyzdys.

Konstruojant šį šabloną buvo pastebėtas pirmasis šio metodo trūkumas – lankstumas. Norint generuoti skirtingus komponentus, reikalingi keletas skirtingų šablonų, skirtų atskiriems komponentams. Žinoma yra komponentų, kurie gali būti tarpusavyje labai panašūs. Tokių komponentų šablonus galima apjungti į vieną bendrą šabloną. Tačiau iš kitos pusės, dėl skirtingų šablonų bei konverterių naudojimo, šį metodą galima nesunkiai praplėsti ir palaikyti.

Antrasis žingsnis, kuris yra šiek tiek sunkesnis, yra sukonstruoti konverterį, kuris gali transformuoti JSON kodą, atitinkantį Java Spring Boot karkaso komponentą. Šiuo atveju buvo sukonstruotas šabloninis konverteris, kuris nuskaito sugeneruotą JSON struktūrą, ir pagal iš anksto apibrėžtą struktūrą sukuria reikiamą komponentą.

```python
def transpile_to_java_code(dsl):
    name = dsl["name"]
    fields = dsl["fields"]
    field_decls = ",\n\t".join(f"{f['type']} {f['name']}" for f in fields)
    result_class = dsl["projectionProvider"]["resultClass"]
    projections = ",\n\t\t\t\t\t\t".join(dsl["projectionProvider"]["projections"])

    return f"""public record {name}(\n\t{field_decls}\n) {{\n
    public static ProjectionProvider<{result_class}> getProjection() {{
        return ProjectionProvider.<{result_class}>builder()
                .resultClass({result_class}.class)
                .projections(List.of(
                        {projections}
                ))
                .build();
    }}
    """
```

7 pav. Užpildyto JSON kodo konvertavimo į Java Spring Boot komponentą kodas.

Pateiktame paveikslėlyje (7 pav.) yra pateiktas metodo įgyvendinimas, kuriame konvertuojamas gautas JSON kodas į Java Spring Boot komponento kodą. Iš esmės procesas atrodo paprastas – iš gauto JSON kodo parenkami reikalingi laukai, kurie vėliau yra įstatomi į tam tikras kodo vietas. Iš pateikto pavyzdžio matome, kad šablonas turi būti griežtai apibrėžtas, priešingu atveju – įsivėlus klaidai kodas bus nekorektiškas.

5.3. Užklausų inžinerija pagrįstas metodas

Antrasis metodas, kuris buvo įgyvendintas šiame darbe, yra labiau žinomas ir gali būti taikomas be papildomo programavimo. Tačiau šiame darbe buvo sukonstruotas mažas įrankis, kuris pagal pateiktą vartotojo užklausą automatiškai išgauna aktualių komponentų kodo pavyzdžius iš esamo Java Spring Boot projekto, ir juos kaip papildomą kontekstą pateikia didžiajam kalbos modeliui.

Šio metodo įgyvendinimui buvo sukurti du atskiri moduliai. Pirmasis modulis buvo sukurtas naudojant Python programavimo kalbą. Šio modulio paskirtis yra apdoroti vartotojo užklausą, nustatyti, kokį komponentą reikia generuoti, ir užklausti atitinkamų kodo pavyzdžių. Generuojamo komponento nustatymas vyko pagal vartotojo pateiktą griežtai apibrėžtą užklausą, kurioje yra aiškiai nurodyta vieta, kurioje turi būti nurodomas komponentas. Šis modulis naudojo biblioteką, „Py4J“, kurios pagalba naudojant klasę JavaGateway buvo galima komunikuoti su Java programa, tai yra kviesti jos metodus.

```
from py4j.java_gateway import JavaGateway

gateway = JavaGateway()
parser = gateway.entry_point

def extract_similar_code_by_component_name(project_path: str, description: str):
    component = extract_component_from_description(description)
    examplesString = parser.extractByComponentAsString(project_path, component)
    examples = examplesString.split("\n\n") if examplesString else []
    return examples
```

8 pav. Kreipimosi į Java modulyje esantį metodą kodas.

Paveikslėlyje (8 pav.) pateikiamas tokio proceso pavyzdys. Pirmiausia yra gaunamas komponento pavadinimas, kurį norima generuoti. Tada yra kreipiamasi į Java modulį, kuris priima užklausas iš Python modulio, ir kviečiamas metodas *extractByComponentAsString*, kuriam yra paduodamas projekto kelias ir komponento pavadinimas kaip parametrai. Galiausiai, pavyzdžiai yra apdorojami išskleidžiant juos į sąrašą.

Antrasis modulis sukurtas naudojant Java programavimo kalbą ir dvi bibliotekas, iš kurių viena geba atlikti Java programavimo kalbos kodo analizę, kita – atlikti komunikaciją tarp Java ir Python programų. Šio modulio esmė yra atrinkti norimo generuoti komponento panašių komponentų kodo pavyzdžius, kurie yra naudojami realiame projekte. Paveikslėlyje (9 pav.) yra pateiktas Java modulyje esantis metodo kodas, kuris randa norimo generuoti komponento kodo pavyzdžius ir juos grąžina atgal Python moduliui.

```
public String extractByComponentAsString(String projectPath, String suffix) {
        List<String> matches = new ArrayList<>();
        try {
            ParserConfiguration config = new ParserConfiguration();
            config.setLanguageLevel(LanguageLevel.JAVA_17);
            SourceRoot sourceRoot = new SourceRoot(Paths.get(projectPath));
            sourceRoot.setParserConfiguration(config);

            List<ParseResult<CompilationUnit>> parsedResults = sourceRoot.tryToParse();
            for (ParseResult<CompilationUnit> result : parsedResults) {
                if (result.isSuccessful() && result.getResult().isPresent()) {
                    CompilationUnit compilationUnit = result.getResult().get();
                    Path filePath = result.getResult()
                        .flatMap(CompilationUnit::getStorage)
                        .map(s -> s.getPath())
                        .orElse(null);
                    if (filePath != null && filePath.getFileName().toString().contains(suffix)) {
                        matches.add(compilationUnit.toString());
                    }
                }
            }
        } catch (Exception e) {
            matches.add("ERROR: " + e.getMessage());
        }
        return String.join("\n\n", matches);
    }
```

9 pav. Panašių komponentų radimo ir apdorojimo Java modulyje kodas.

Galiausiai Python modulis apjungia gautus kodo pavyzdžius prie sisteminės užklausos, skirtos didžiajam kalbos modeliui, taip pagerindamas sugeneruoto komponento kodo kokybę.

Nors metodo įgyvendinimas nesukėlė didesnių problemų, tačiau vienas iš pagrindinių šio modulio ribojimų yra griežtos struktūros vartotojo užklausa. Vartotojas pateikdamas užklausą turi tiksliai ir aiškiai nusakyti generuojamo komponento pavadinimą tinkamoje vietoje. Priešingu atveju, sistema negalės išgauti kontekstui svarbių kodo pavyzdžių ir užtikrinti didelio kodo tikslumo.

5.4. Agentais pagrįstas autonominis metodas

Galiausiai buvo įgyvendintas paskutinis metodas, kuris pasirodė esąs sudėtingiausias. Šis metodas yra pagrįstas generavimo modeliu, kuriame dalyvauja keli skirtingi agentai, turintys savo rolę. Įgyvendintas metodas buvo sudarytas iš keturių skirtingų agentų: galvotojas, programuotojas, vertintojas ir taisytojas. Kiekvienas agentas buvo įgyvendintas kaip atskiras kreipinys į didįjį kalbos modelį, turintis savo specifinę sisteminę užklausą. Tokio kreipinio pavyzdys buvo pateiktas aukščiau esančiame paveikslėlyje (5 pav.).

Pirmiausia buvo realizuotas galvotojo agentas, nuo kurio prasideda visas generavimo procesas. Šiam agentui buvo paduodama vartotojo įvesta informacija, tai yra, kokį komponentą yra siekiama sugeneruoti, ir šio komponento detalus aprašymas su reikalavimais. Uždavus tokią užklausą, agentas sugeneruodavo keletą skirtingų strategijų, kaip šį komponentą būtų galimą realizuoti.

Toliau sekė programuotojo agento realizavimas, kuris atsakingas už komponento programinio kodo generavimą. Šiam agentui buvo pateikiama kiekviena iš strategijų, kurios buvo sugeneruotos galvotojo agento.

Trečiasis agentas, kuris buvo realizuotas, yra vertintojo agentas. Jo atsakomybė yra įvertinti programuotojo agento sugeneruotą komponento programinį kodą, kad jis būtų teisingas, tvarkingas ir atitiktų pradinius vartotojo reikalavimus. Taip pat patikrinti papildomus nurodytus reikalavimus sisteminėje agento užklausoje. Buvo prašoma, kad agentas sugeneruotą programinį kodą įvertintų balu nuo 0 iki 10, taip pat pateiktų grįžtamąjį ryšį, kuriame pateiktos tobulinamos vietos.

Galiausiai paskutinis agentas, kuris buvo įgyvendintas, yra taisytojo agentas. Jo pagrindinis darbas yra blogai įvertintą komponento programinį kodą pataisyti pagal iš vertintojo agento gautą grįžtamąjį ryšį. Toks pataisytas kodas buvo laikomas galutiniu rezultatu.

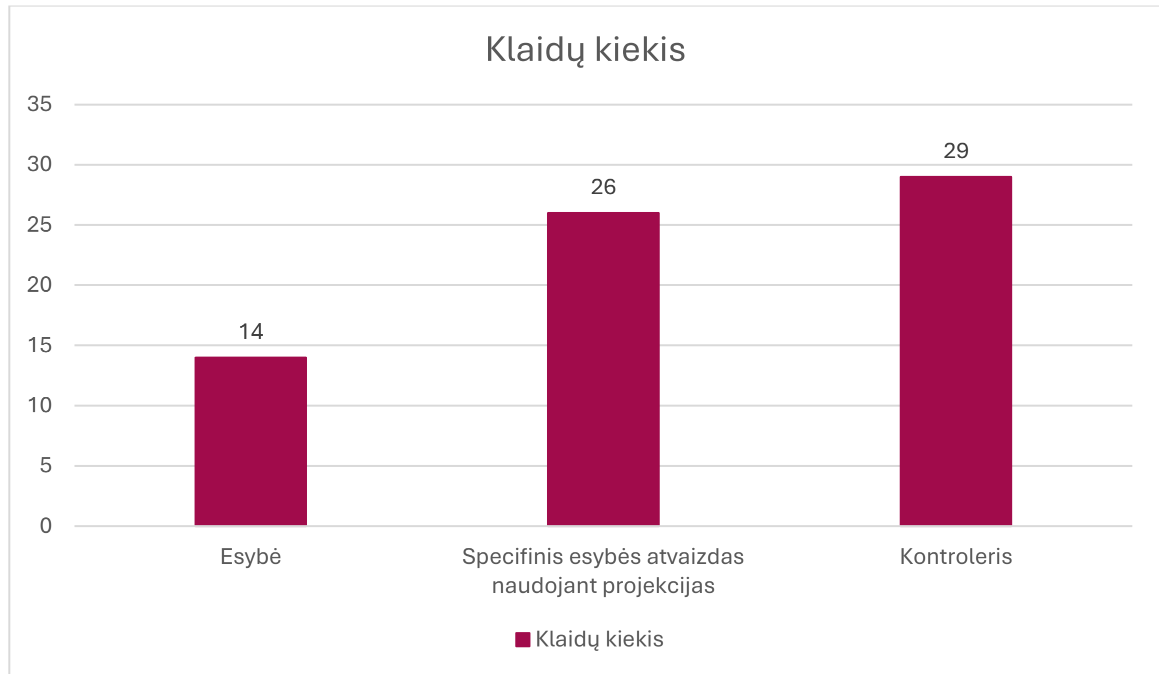
Siekiant užtikrinti sklandžią komunikaciją tarp agentų, kiekvieno agento pateikiamas rezultatas buvo apdorojamas, skaidomas ir transformuojamas, kad užtikrintų sklandžią ir suprantamą komunikaciją.

Įgyvendinant šį metodą buvo susidurta su keletą iššūkių. Pirmiausia, tai metodo kompleksiškumas. Nors atskirų agentų įgyvendinimas daug laiko neatėmė, tačiau daug laiko pareikalavo jų koordinavimas, siekiant užtikrinti taisyklingiausius rezultatus. Taip pat iššūkį kėlė ir gaunamo atsakymo iš didžiojo kalbos modelio apdorojimas, kadangi sunku garantuoti, kad kiekvieną generavimo iteraciją didysis kalbos modelis pateiks atsakymą tokia pačia struktūra.

6. Bandymas

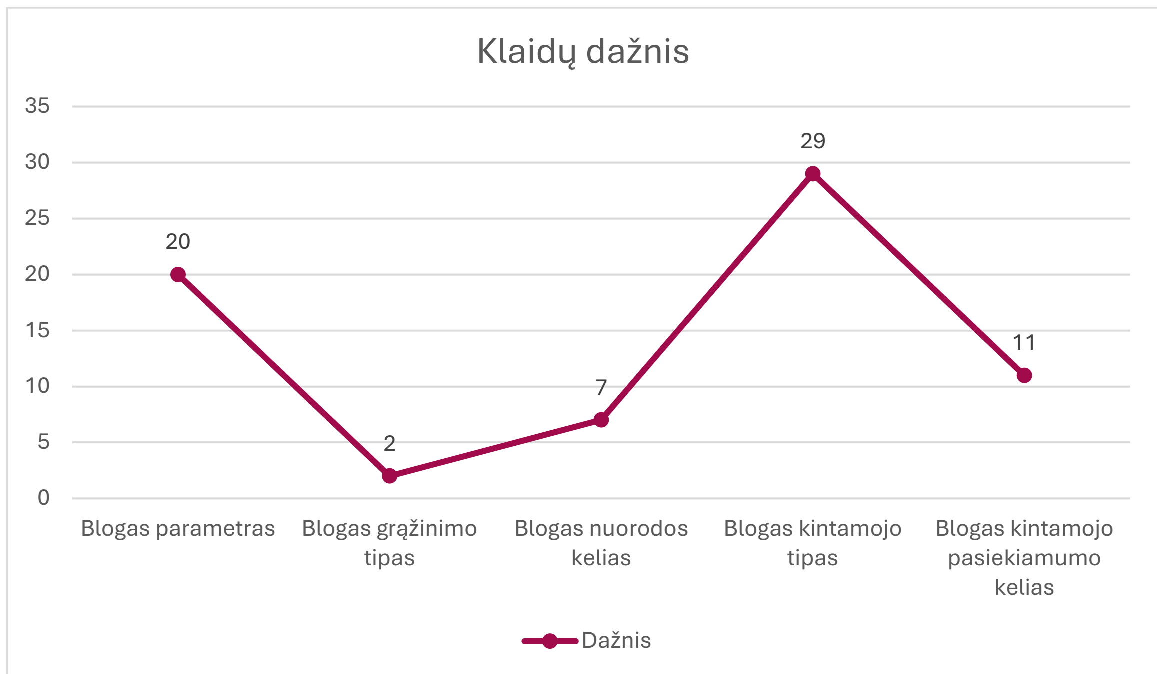
Įgyvendinus visu pasirinktus metodus, buvo atliktas bandymas, kuris aprašytas metodikos skyriuje. Žemiau pateiktuose skyreliuose pateikiami bandymo procese gauti rezultatai ir pastebėjimai.

6.1. Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas



1 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant tarpinių šablonų generavimo metodu.

Iš pateikto grafiko (1 figūra) matome, kad vidutiniškai tarp 100 sugeneruotų atvejų dažniausiai pasitaiko 23 klaidingi atvejai. Taip galima apskaičiuoti, kad tarp visų sugeneruotų kodų atvejų, vidutiniškai 77% atvejų yra pateikiami teisingi iš pirmo generavimo karto. Daugiausia blogų atvejų pasitaikė generuojant kontrolerius, tačiau taip ir buvo tikėtasi, kadangi kontroleris yra kur kas sudėtingesnis komponentas, lyginant su esybėmis ar atvaizdais. Taip pat dideliu blogų atvejų kiekiu pasižymėjo ir esybės atvaizdo komponentas. Tačiau šiame darbe buvo bandoma sugeneruoti ne paprastą, o specifiškai projekte naudojamo sąrašams atvaizduoti esybės atvaizdo komponentą, kuris pasižymi neįprastine struktūra ir naudoja papildomas bibliotekas. Keletas blogų atvejų pavyzdžių pateikiami 1 priede.



2 figūra. Dažniausiai pasikartojančių klaidų dažnis.

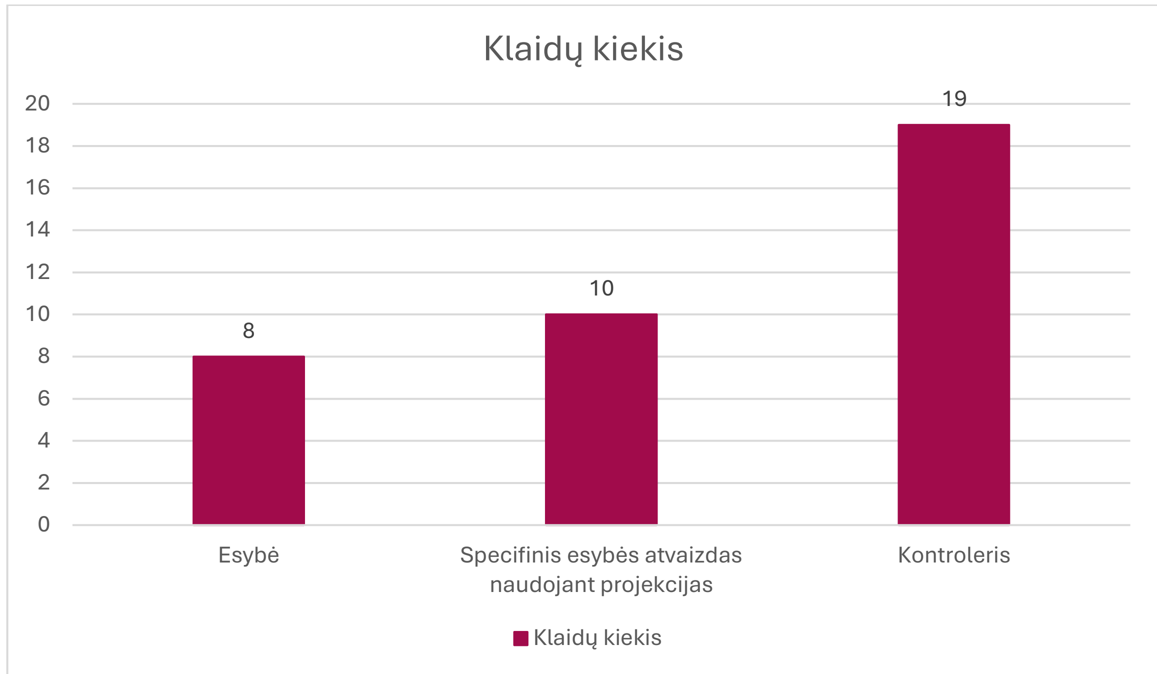
Aukščiau pateiktame grafike (2 figūra) pateikiamas dažniausiai sutinkamos klaidos. Kaip matome dažniausiai yra naudojamas neteisingas parametras arba blogas kintamojo tipas. Tai yra suprantama, kadangi didysis kalbos modelis neturi visos informacijos apie įgyvendinimo detales. Kitos klaidos vėlgi labiau susijusios su atvaizdu, kuris nėra plačiai naudojamas ir yra pritaikytas projekto kontekste. Tačiau šių klaidų nebuvo pastebėta daug. Tai tokios klaidos, kur nenaudojamas tinkamas kintamojo pasiekiamumo kelias. Nors klaidų buvo aptikta ganėtinai nemažai, tačiau jos yra greitai išsprendžiamos ir nereikalauja didelio laiko sugaišimo.

Atlikus šį bandymą, buvo pastebėti du svarbūs dalykai, kurie galėtu patobulinti šio metodo efektyvumą. Pirmiausia, papildomas didžiojo kalbos modelio apmokymas. Tai yra, šie bandymai buvo atlikti ant didžiojo kalbos modelio, kuris nėra papildomai apmokytas. Taigi, jeigu modelis būtų papildomai apmokytas su duomenų rinkiniais, kurie yra aktualūs projekto struktūrai, yra tikėtina, kad rezultatai būtų dar geresni. Dar vienas dalykas, kuris padeda pagerinti šio metodo generuojamo kodo tikslumą, yra sisteminės užklausos dalies praplėtimas platesniu kontekstu ir iš vartotojo gaunamas detalus uždavinio aprašymas.

6.2. Užklausų inžinerija pagrįstas metodas

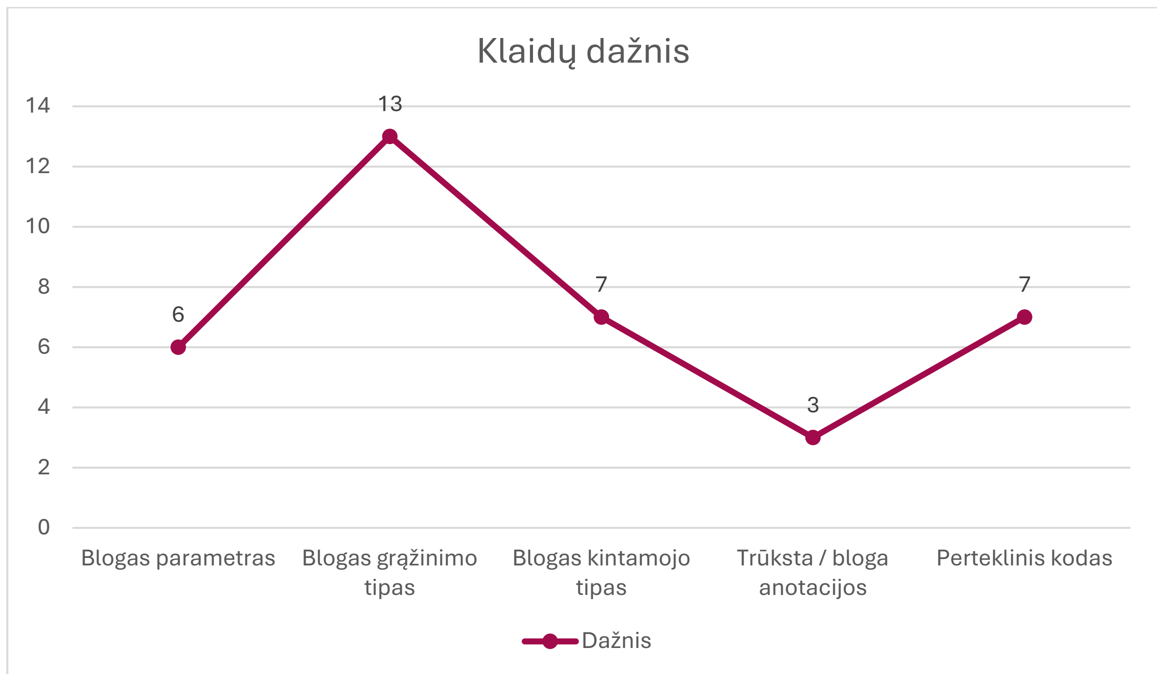
Atlikus antrąjį bandymą blogų atvejų kiekis pateiktas žemiau esančiame grafike (3 figūra). Jau iš pirmo žvilgsnio yra matomas didelis patobulėjimas tarp teisingų sugeneruotų kodų atvejų. Buvo apskaičiuota, kad šiuo metodu vidutiniškai 87,67% buvo pateikiamas teisingas atvejis. Taigi, šis metodas lyginant su pirmuoju metodu parodė 13,86% patobulėjimą pateikiant teisingus atvejus. Visgi

daugiausia blogų atvejų vėlgi pasitaikė tarp sugeneruotų kontrolerio komponentų. Tačiau generuojant specifiškai projekto kontekste naudojamas esybių sąrašų atvaizdus, teisingų atvejų kiekis ženkliai išaugo. Keletas blogų atvejų pavyzdžių pateikiami 2 priede.



3 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant užklausų inžinerija pagrįstu metodu.

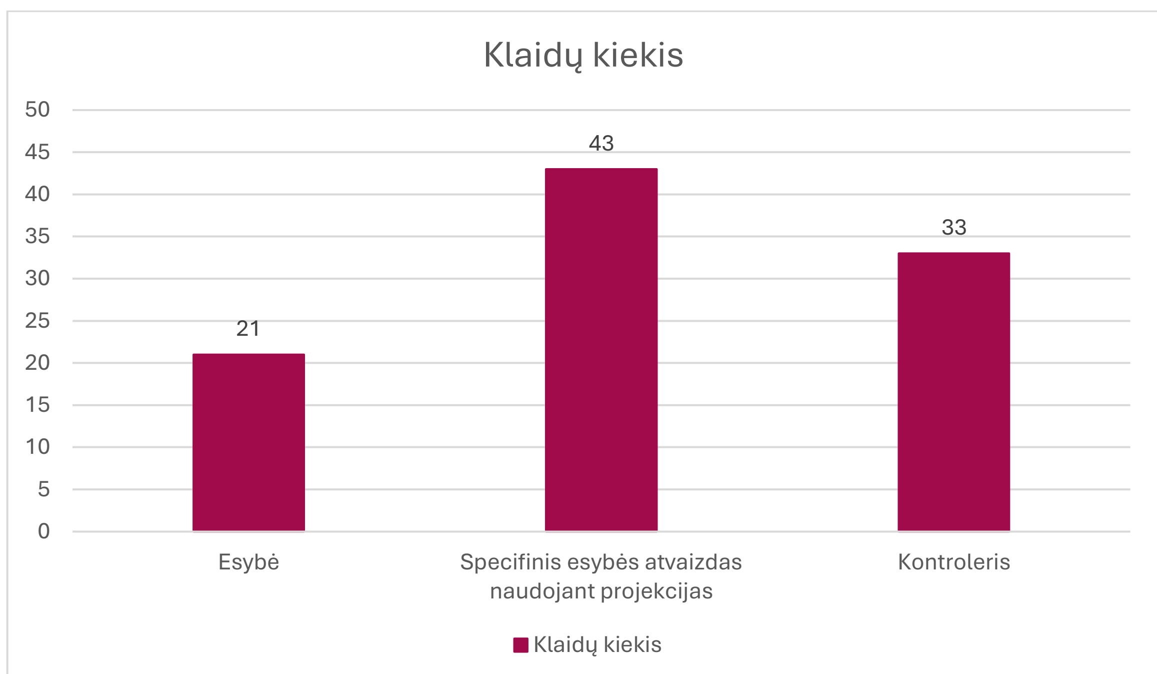
Iš pateikto grafiko (4 figūra), kuriame pateikiamos dažniausiai sutinkamos klaidos, matome, kad klaidos išliko panašios, tačiau jų kiekis ženkliai sumažėjo. Pagrindinės klaidos išliko tokios pat, kad blogas kontrolerio metodų grąžinimo tipas. Taip pat šiuo metodu generuojant komponentus buvo pastebėtos keletas naujų klaidų atvejų, tai yra pateikiamas perteklinis kodas. Generuojant atvaizdus, kurie naudojami konkretaus projekto kontekste, klaidų buvo aptinkama kur kas mažiau.



4 figūra. Dažniausiai pasikartojančių klaidų dažnis.

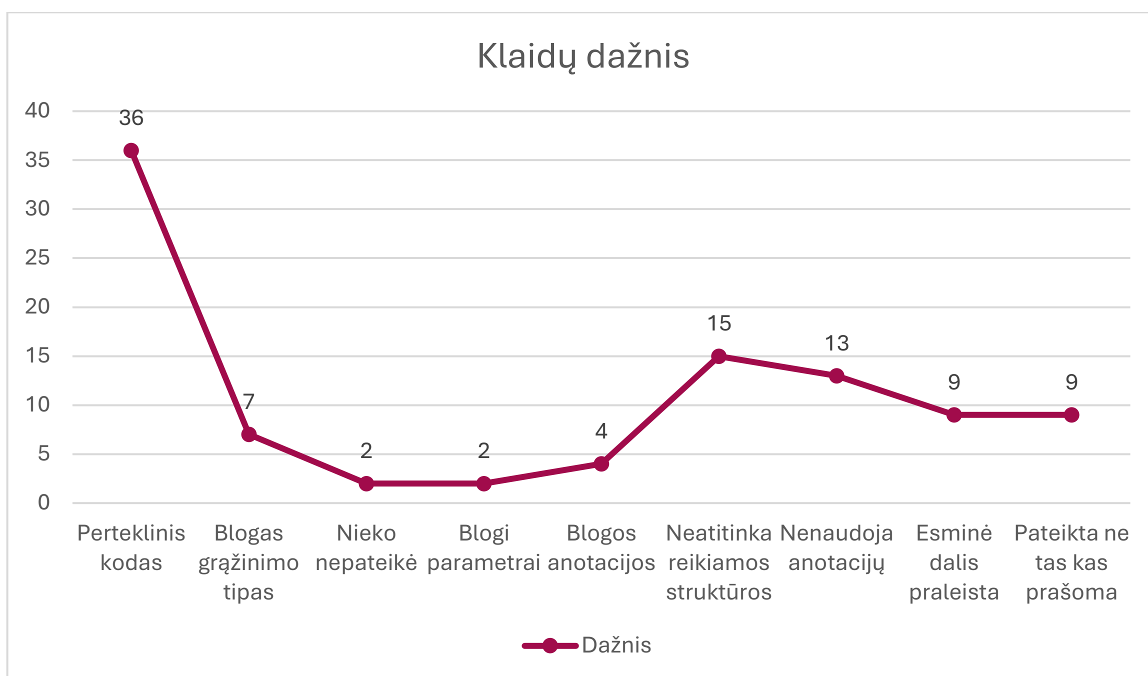
Atlikus bandymus su šiuo metodu, didelių trūkumų ar pastabų šiam metodui nebuvo atrasta. Vienintelis trūkumas būtų priklausomybė nuo griežtos užklausos struktūros. Tačiau ši problema gali būti nesunkiai išsprendžiama pritaikant sintaksės analizavimo algoritmą analizuojant vartotojo užklausą, ar sukonkretinant paduodamą kontekstą. Vis dėlto, šio metodo įgyvendinimas daug laiko neužėmė ir sunkumų nesukėlė.

6.3. Agentais pagrįstas autonominis metodas



5 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant agentais pagrįstu autonominiu metodu.

Paskutinio bandymo, generuojant komponentus naudojant agentais pagrįstą autonominį metodą, rezultatai nuvylė. Iš grafiko (5 figūra) yra matoma, kad šis metodas parodė prasčiausią rezultatą lyginant su prieš tai naudotais metodais. Vidutiniškai tik 63,67% pateikiamas teisingas atvejis iš pirmo karto. Lyginant su pirmuoju metodu, teisingų atvejų sumažėjo 12,12%, o su antruoju net 22,81%. Generuojant šiuo metodu, priešingai nei su kitais dvejais metodais, daugiausia blogų atvejų pasitaikė generuojant specifinius esybės atvaizdus. Mažiausiai klaidų buvo aptikta generuojant labai paprastus komponentus – esybes. Keletas blogų atvejų pavyzdžių pateikiami 3 priede.



6 figūra. Dažniausiai pasikartojančių klaidų dažnis.

Taikant šį metodą buvo pastebėta naujų, dar prieš tai nesutiktų, klaidų atsiradimas. Iš grafiko (6 figūra) matoma, kad dažniausiai pasitaikanti klaida buvo perteklinio kodo pateikimas. Tokių klaidų buvo aptikta generuojant visus komponentus. Generuojant specifinius esybės atvaizdus dažniausiai aptinkamos klaidos buvo struktūros neatitikimas arba pateikimas ne to, kas buvo užduota. Generuojant paprastus komponentus, tai yra esybės, dažniausiai pasitaikanti klaida buvo anotacijų nenaudojimas. Taip pat iš visų trijų metodų generuojant komponentus, šiuo metodu generuojant pasitaikė atvejų, kai rezultatas iš viso nebuvo pateiktas.

Atliekant bandymus su šiuo metodu buvo pastebėta keletas problemų. Pirmiausia generavimo procesas užima daug laiko. Dėl visos trunkančios komunikacijos ir duomenų apdorojimo, generavimo laikas labai išauga. Taip pat gaunami rezultatai nestebina. Nors šis metodas turėtų pateikti rezultatus žymiai taisyklingiau, tačiau atliekant bandymą, to nebuvo pastebėta. Metodas pasilieka daug laisvės sprendimams priimti, dėl to kyla daug „haliucinacijų“ generuojant programinį kodą. Taisyklingumas buvo bandomas gerinti pridedant pavyzdžių prie užklausos, tačiau rezultatas patobulėjo tik dalinai.

7. Rezultatai

Sėkmingai buvo išanalizuoti trys metodai, naudojantys didžiuosius kalbos modelius kodo generavimui. Šie metodai buvo suskirstyti į atitinkamas kategorijas:

- tarpinių šablonų generavimu pagrįsti kodo generavimo metodai – tinka generuojant programinį kodą, kuris yra parašytas rečiau sutinkamomis arba mažiau naudojamomis technologijomis;
- užklausų inžinerija pagrįsti kodo generavimo metodai – tinka generuojant įvairaus tipo programinį kodą;
- agentais pagrįsti autonominiai kodo generavimo metodai – tinka generuojant didelės apimties programinį kodą arba konstruojant sudėtingas sistemas.

Pasirinkti ir išanalizuoti metodai buvo sėkmingai pritaikyti generuoti Java Spring Boot karkaso komponentus. Pritaikius juos buvo atliktas bandymas kurio metu buvo sugeneruoti 900 komponentų ir surinkti reikiami duomenys šių metodų įvertinimui.

4 lentelė. Rezultatų palyginimų lentelė.

Metodas	Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas	Užklausų inžinerija pagrįstas kodo generavimo metodas	Agentais pagrįstas autonominis kodo generavimo metodas
Bendras blogų atvejų skaičius	69 blogi atvejai	37 blogi atvejai	97 blogi atvejai
Pateikiamų teisingų atvejų procentas	77% teisingų atvejų	87,67% teisingų atvejų	63,67% teisingų atvejų
Generavimo greitis (kokybiniu įverčiu)	Vidutinis	Greitas	Lėtas

Kaip matoma lentelėje (4 lentelė), taisyklingiausia kodą generuoja užklausų inžinerija pagrįstas metodas, kuris sugeneravo mažiausiai blogų atvejų. Blogiausią generavimo taisyklingumą parodė agentais pagrįstas autonominis kodo generavimo metodas. Trečdalis jo sugeneruotų kodo atvejų buvo neteisingi.

Vertinant metodų generavimo greitį, užklausų inžinerija pagrįstas metodas buvo greičiausias iš visų metodų. Pateikus užklausą rezultatas buvo gaunamas beveik iš karto. Tarpinių šablonų generavimu pagrįstas kodo generavimo metodas taip pat pasižymėjo generavimo greičiu, tačiau būdavo atveju, kai rezultato tekdavo palaukti. Ir lėčiausia generavimo greitį parodė agentais pagrįstas autonominis kodo generavimo metodas, dėl savo kompleksiškumo komunikuojant su agentais tarpusavyje.

Išvados

Atlikus bandymus generuojant Java Spring Boot karkaso komponentus metodais, kurie naudoja didžiuosius kalbos modelius, buvo nustatyta, kad taisyklingiausia programinį kodą galima sugeneruoti naudojant užklausų inžinerija pagrįstus metodus. Šiam metodui pavyko tai padaryti dėl pateikiamo papildymo tikslaus konteksto didžiajam kalbos modeliui, kas leisdavo sugeneruoti kodą išvengiant klaidų ar „haliucinacijų“.

Taip pat buvo nustatyta, kad agentais pagrįsti autonominiai metodai programinį kodą generuoja prasčiausiai, įtraukiant daugiausiai klaidų. Nors išanalizavus literatūros šaltinius buvo manoma, kad šis metodas parodys geriausią rezultatą, tačiau literatūros analizėje pateikti rezultatai nesutapo su šio darbo gautais rezultatais.

Tarp visų metodų buvo nustatyta kad dažniausiai pasitaikiusios klaidos programiniame kode buvo blogas kintamojo ar metodo grąžinimo tipas ir perteklinio kodo sugeneravimas. Tai buvo galima numatyti, kadangi didysis kalbos modelis neturi papildomo konteksto. Tačiau tarpinių šablonų generavimu pagrįstam metodui pavyko sumažinti perteklinio kodo generavimo, dėl griežtų generavimo taisyklių, o užklausų inžinerija pagrįstam metodui pavyko sumažinti blogų tipų klaidas, dėl pateikiamo konteksto.

Apibendrinant išvadas galima teigti, kad geriausias metodas generuoti Java Spring Boot karkaso komponentus yra užklausų inžinerija pagrįstas metodas. Kiti du metodai gali pademonstruoti pranašumą juos taikant kitose srityse. Taip pat išvados skirtųsi, jei bandymai būtų atlikti su specialiai apmokytais didžiaisiais kalbos modeliais.

Literatūros sąrašas

[KCW25] P. Kogler, W. Chen, S. Wallner. „Code Generation for Niche Programming Languages with Large Language Models“, 2025

[LLZ+24] J. Li, H. Le, Y. Zhou, C. Xiong, S. Savarese, D. Saho. „CodeTree: Agent-guided Tree Search for Code Generation with Large Language Models“, 2024

[MM25] A. Mumuni, F. Mumuni. „Large language models for artificial general inteligence (AGI): A survey of foundational principles and approcahes“, 2025

[PHX+24] Z. Pan, X. Hu, X. Xia, X. Yang. „Enhancing Repository-Level Code Generation with Integrated Contextual Information“, 2024

[RWK+23] Z. Rasheed, M. Waseem, K. Kemell, W. Xiaofeng, A. N. Duc, K. Systa, P. Abrahamsson. „Autonomous Agents in Software Development: A Vision Paper“, 2023

[WCD24] Z. Rasheed, M. Waseem, M. A. Sami, K. K. Kemell, A. Ahmad, A. N. Duc, P. Abrahamsson. „Autonomous agents in software development: A vision paper“. Rinkinyje: *International Conference on Agile Software Development*. Springer Nature Switzerland, Cham, p. 15–23, 2024

[ZFS+24] Q. Zhang, C. Fang, Y. Shang, T. Zhang, S. Yu, Z. Chen. „NoManisanIsland: Towards Fully Automatic Programming by CodeSearch, Code Generation and Program Repair“, 2024

Iliustracijų sąrašas

1 pav. Kalbos modelių raida chronologine tvarka 8
2 pav. Supaprastinta JSON-DSL generavimo schema naudojant didįjį kalbos modelį 11
3 pav. Supaprastinta koordinuotų didžiųjų kalbos modelių paieškos metodo schema 12
4 pav. Sukonstruotos užklausos pavyzdys 15
5 pav. Užklausos siuntimo didžiajam kalbos modeliui kodo pavyzdys. 20
6 pav. JSON šablono pavyzdys. 21
7 pav. Užpildyto JSON kodo konvertavimo į Java Spring Boot komponentą kodas. 21
8 pav. Kreipimosi į Java modulyje esantį metodą kodas. 22
9 pav. Panašių komponentų radimo ir apdorojimo Java modulyje kodas. 23

1 lentelė. Metodų savybės 16
2 lentelė. Kategorijų palyginimo lentelė 18
3 lentelė. GPT-4o-mini modelio apžvalgos lentelė 19
4 lentelė. Rezultatų palyginimų lentelė. 30

1 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant tarpinių šablonų generavimo metodu. 25
2 figūra. Dažniausiai pasikartojančių klaidų dažnis. 26
3 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant užklausų inžinerija pagrįstu metodu. 27
4 figūra. Dažniausiai pasikartojančių klaidų dažnis. 28
5 figūra. Blogai sugeneruotų komponentų atvejai generuojant komponentus naudojant agentais pagrįstu autonominiu metodu. 28
6 figūra. Dažniausiai pasikartojančių klaidų dažnis. 29

Priedai

1 priedas. Blogų atvejų pavyzdžiai generuojant tarpinių šablonų generavimu pagrįstu metodu.

Blogas kintamųjų tipas:

```
public record ApplicationListView(
    string id,
    string status,
    string submissionDate,
    string individualName,
    string legalPersonName,
    string jobPlacesCount,
    string totalAmountOfSubsidies
) {

  public static ProjectionProvider<ApplicationListView> getProjection() {
    return ProjectionProvider.<ApplicationListView>builder()
        .resultClass(ApplicationListView.class)
        .projections(List.of(
            ApplicationSpecification.root.id,
            ApplicationSpecification.root.status,
            ApplicationSpecification.root.submissionDate,
            ApplicationSpecification.root.individualName,
            ApplicationSpecification.root.legalPersonName,
            ApplicationSpecification.root.jobPlacesCount,
            ApplicationSpecification.root.totalAmountOfSubsidies
        ))
        .build();
  }
}
```

Blogas pametras:

```
@RestController
@RequestMapping("/files")
public class FileController {

   @PostMapping("/upload")
   public void uploadFile(@RequestBody multipart/form-data request) {
      // TODO: implement uploadFile
      return new void();
   }

   @GetMapping("/{id}")
   public byte[] downloadFile() {
      // TODO: implement downloadFile
      return new byte[]();
   }
}
```

Blogas kelias į kintamąjį:

```
public record PaymentListView(
      Long id,
      PaymentStatus status,
      LocalDate paymentDate,
      BigDecimal paymentAmount
) {

   public static ProjectionProvider< PaymentListView > getProjection() {
      return ProjectionProvider.< PaymentListView >builder()
            .resultClass(PaymentListView.class)
            .projections(List.of(
                  id,
                  status,
                  paymentDate,
                  paymentAmount
            ))
            .build();
   }
}
```

2 priedas. Blogų atvejų pavyzdžiai generuojant užklausų inžinerija pagrįstu metodu.

Perteklinis kodas:

```
@Entity
@Getter
@Setter
@Table(name = "INVOICE")
public class InvoiceEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "NAME")
    private String name;

    @Column(name = "CREATED_DATE")
    private LocalDateTime createdDate;

    @Column(name = "STATUS")
    @Enumerated(EnumType.STRING)
    private InvoiceStatus status;

    public enum InvoiceStatus {
        PENDING,
        PAID,
        CANCELLED
    }
}
```

Blogas grąžinimo tipas:

```
@RestController
@RequestMapping("/internal/project")
@RequiredArgsConstructor
@Validated
public class ProjectController {

    private final ProjectService projectService;

    @PostMapping("/create-or-update")
    public void createOrUpdateProject(@Valid @RequestBody ProjectDataView projectDataView) {
        projectService.createOrUpdate(projectDataView);
    }

    @GetMapping("/{id}")
    public void findById(@PathVariable Long id) {
        return projectService.findById(id);
    }

    @DeleteMapping("/{id}")
    public void deleteById(@PathVariable Long id) {
        projectService.deleteById(id);
    }
}
```

3 priedas. Blogųogų atvejų pavyzdžiai generuojant agentais pagrįstu autonominiu metodu.

Perteklinis kodas:

```
@RestController
@RequestMapping("/products")
public class ProductController {

    private final ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }


    @PostMapping
    public ResponseEntity<Void> addProduct(@RequestBody Product product) {
        productService.addProduct(product);
        return ResponseEntity.status(201).build(); // Return 201 Created status
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Product product = productService.getProductById(id);
        if (product != null) {
            return ResponseEntity.ok(product); // Return 200 OK status with product
        } else {
            return ResponseEntity.notFound().build(); // Return 404 Not Found status
        }
    }
}

interface ProductService {
    void addProduct(Product product);
    Product getProductById(Long id);
}

class Product {
    // Product fields, getters, and setters would be defined here
}
```

Nenaudojamos anotacijos:

```
@Entity
@Table(name = "audit")
public class AuditEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "entity_name")
    private String entityName;

    @Column(name = "entity_id")
    private Long entityId;

    @Enumerated(EnumType.STRING)
    @Column(name = "operation_type")
    private OperationType operationType;

    @Column(name = "timestamp")
    private LocalDateTime timestamp;

    @Column(name = "performed_by")
    private String performedBy;

    @Column(name = "description")
    private String description; // Added essential field for description

    // Getters and Setters using Lombok annotations
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getEntityName() {
        return entityName;
    }
```

```
public void setEntityName(String entityName) {
    this.entityName = entityName;
}

public Long getEntityId() {
    return entityId;
}

public void setEntityId(Long entityId) {
    this.entityId = entityId;
}

public OperationType getOperationType() {
    return operationType;
}

public void setOperationType(OperationType operationType) {
    this.operationType = operationType;
}

public LocalDateTime getTimestamp() {
    return timestamp;
}

public void setTimestamp(LocalDateTime timestamp) {
    this.timestamp = timestamp;
}

public String getPerformedBy() {
    return performedBy;
}

public void setPerformedBy(String performedBy) {
    this.performedBy = performedBy;
}

public String getDescription() {
    return description;
}
```

```
    public void setDescription(String description) {
        this.description = description;
    }
}
```

Sugeneruota ne tai kas buvo prašoma (Buvo generuojamas PaymentListView):

```
@Repository
public interface PaymentRepository extends JpaRepository<Payment, Long> {

    @Query("SELECT new com.example.PaymentListView(p.id, p.paymentStatus, p.paymentAmount, p.currency, p.paymentDate, p.payerName, p.paymentType, p.lastModifiedBy, p.lastModifiedDate) FROM Payment p JOIN p.relatedEntity r")
    List<PaymentListView> getProjection();
}
```

4 priedas. Nustatyto geriausio metodo kodas

Java modulio programinis kodas:

pom.xml

```
<dependencies>
  <dependency>
    <groupId>com.github.javaparser</groupId>
    <artifactId>javaparser-core</artifactId>
    <version>3.25.4</version>
  </dependency>
  <dependency>
    <groupId>net.sf.py4j</groupId>
    <artifactId>py4j</artifactId>
    <version>0.10.9.5</version>
  </dependency>
</dependencies>
```

JavaParserServer.java

```
public class JavaParserServer {
  public String extractByComponentAsString(String projectPath, String suffix) {
    List<String> matches = new ArrayList<>();
    try {
      ParserConfiguration config = new ParserConfiguration();
      config.setLanguageLevel(LanguageLevel.JAVA_17);
      SourceRoot sourceRoot = new SourceRoot(Paths.get(projectPath));
      sourceRoot.setParserConfiguration(config);

      List<ParseResult<CompilationUnit>> parsedResults = sourceRoot.tryToParse();
      for (ParseResult<CompilationUnit> result : parsedResults) {
        if (result.isSuccessful() && result.getResult().isPresent()) {
          CompilationUnit compilationUnit = result.getResult().get();
          Path filePath = result.getResult()
              .flatMap(CompilationUnit::getStorage)
              .map(CompilationUnit.Storage::getPath)
              .orElse(null);
          if (filePath != null && filePath.getFileName().toString().contains(suffix)) {
            matches.add(compilationUnit.toString());
          }
        }
      }
```

```
        } catch (Exception e) {
            matches.add("ERROR: " + e.getMessage());
        }
        return String.join("\n\n", matches);
    }

    public static void main(String[] args) {
        GatewayServer server = new GatewayServer(new JavaParserServer());
        server.start();
        System.out.println("JavaParserServer started");
    }
}
```

Python modulio kodas:

```
from py4j.java_gateway import JavaGateway
import openai
import re
import time

openai.api_key = "YOUR_OPENAI_API_KEY"
gateway = JavaGateway()
parser = gateway.entry_point

def extract_component_from_description(description: str) -> str:
    match = re.search(r"generate (\w+)", description, re.IGNORECASE)
    if not match:
        return ""
    name = match.group(1)
    match = re.search(r"([A-Z][a-z0-9]+)$", name)
    return match.group(1) if match else ""

def extract_similar_code_by_component_name(project_path: str, description: str):
    component = extract_component_from_description(description)
    examplesString = parser.extractByComponentAsString(project_path, component)
    examples = examplesString.split("\n\n") if examplesString else []
    return examples

def build_prompt(examples: list[str], description: str) -> str:
    prompt = "Here are similar Spring Boot components from the existing codebase:\n\n"
    for code in examples:
```

```
        prompt += f"```java\n{code.strip()}\n```\n\n"
    prompt += f"Now generate a similar component based on this task:\n{description}\n\nReturn Java code only."
    return prompt

def call_llm(system_prompt: str, user_prompt: str, retries: int = 3) -> str:
    for attempt in range(retries):
        try:
            response = openai.ChatCompletion.create(
                model="gpt-4o-mini",
                messages=[
                    {"role": "system", "content": system_prompt},
                    {"role": "user", "content": user_prompt}
                ],
                temperature=0.2,
                max_tokens=1024
            )
            return response["choices"][0]["message"]["content"]
        except Exception as e:
            print(f"[Attempt {attempt+1}] Error: {e}")
            if attempt < retries - 1:
                time.sleep(2 + attempt * 2)
            else:
                return f"// ERROR: {e}"

def generate_component(project_path: str, task_description: str):
    examples = extract_similar_code_by_component_name(project_path, task_description)
    prompt = build_prompt(examples, task_description)
    result = call_llm("You are a senior Java backend developer.", prompt)
    return result.replace("```java", "").replace("```", "").strip()

if __name__ == "__main__":
    project_path = "PROJECT_PATH"
    description = "I need to generate PaymentEntity that contains fields id, paymentStatus, paymentReason,
paymentDate, paymentAmount."
    result = generate_component(project_path, description)
    print(result)
```