VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

Didelio kalbos modelio generuojamų kodo testų su struktūrine validacija kokybės vertinimo metodas

A Method for Quality Evaluation of Code Tests Generated by a Large Language Model with Structural Validation

Magistro baigiamasis darbas

Atliko:	Dovydas Marius Zapkus	(parašas)
Darbo vadovas:	doc. dr. Asta Slotkienė	(parašas)
Recenzentas:	asist. dr. Vytautas Valaitis	(parašas)

Vilnius – 2026

Santrauka

Šiame darbe yra nagrinėjamas didelių kalbos modelių generuojamų kodo vieneto testų kokybės vertinimas, taikant griežtą įvesties užklausos struktūrą bei rezultato struktūrinę validaciją. Literatūros apžvalgoje yra pristatomi kodo vieneto testavimo principai, kodo vieneto testų kokybės vertinimo metrikos, didelių kalbos modelių veikimo ypatumai bei kodo vieneto testų kompiliavimo, vykdymo ir mutacijų testavimo procesai. Tolesnėje darbo dalyje pateikiamas septynių pasirinktų didelių kalbos modelių palyginimas, remiantis eksperimentu su C# programavimo kalbos duomenų rinkiniu, naudojant PydanticAI biblioteką rezultato validacijai, dotnet-coverage įrankį kodo padengiamumo vertinimui bei Stryker.NET įrankį mutacijų testavimui. Atlikus eksperimentą pateikiami modelių kompiliavimo, vykdymo rodikliai, kodo padengiamumo bei mutacijų testavimo metrikos. Modeliai aptariami išskiriant jų gebėjimą generuoti sintaksiškai bei semantiškai taisyklingus kodo vieneto testus, atsparius kodo klaidoms. Remiantis sudarytais palyginimo kriterijais, gauti rezultatai palyginti su kitų autorių atliktais tyrimais. Tyrimas atskleidžia, kad griežtos įvesties struktūros bei rezultato validavimo taikymas gali pagerinti sugeneruotų kodo vieneto testų kompiliavimosi rodiklį, kodo padengiamumą bei mutacijų įvertį. Modeliai grok-4 ir gemini-2.5-flash yra įvertinti kaip pažangiausi modeliai kodo vieneto testų generavimo srityje.

Raktiniai žodžiai: dideli kalbos modeliai, kodo vieneto testai, kodo kokybės metrikos, mutacijų testavimas, griežta įvesties užklausos struktūra, PydanticAI

Summary

This paper examines the evaluation of the quality of unit tests generated by large language models, using a strict prompt structure and structural validation of the results. The literature review presents the principles of unit testing, metrics for evaluating the quality of unit tests, the operational characteristics of large language models and the processes of compilation, execution, and mutation testing of unit tests. The subsequent section of the paper presents a comparison of seven selected large language models based on an experiment using a C# dataset, using PydanticAI library for result validation, dotnet-coverage tool for code coverage assessment, and Stryker.NET tool for mutation testing. Following the experiment, the paper presents the compilation rates of generated tests, execution performance, code coverage, and mutation testing. The models are discussed with a focus on their ability to generate syntactically and semantically correct unit tests that are resistant to code faults. Based on the established comparison, the results are compared with studies conducted by other authors. The study reveals that the application of strict prompt structure and result validation can improve the compilation performance of generated unit tests, code coverage, and mutation coverage metrics. The grok-4 and gemini-2.5-flash models are evaluated as the most advanced models in the field of unit test generation.

Keywords: large language models, unit tests, code quality metrics, mutation testing, strict prompt structure, PydanticAI

Iliustracijų sąrašas

Lentelių sąrašas

Turinys

Įvadas

Programų sistemų testavimas yra veikla, kuri gali pagerinti programų sistemos (toliau - PS) patikimumą, leisti išvengti ar sumažinti kritinių klaidų skaičių bei padėti programų sistemų kūrėjams sukurti kokybiškesnį ir patikimesnį galutinį produktą [GPD+19].

Kodo vieneto testavimas yra vienas iš svarbiausių PS testavimo etapų, kuris, jei atliktas teisingai, gali pagerinti galutinio produkto kokybę ir suteikti produkto kūrėjams darbo našumo [ATA23, Job21]. Kodo vieneto testais yra testuojamos mažiausios PS programos dalys, t. y. funkcijos, metodai, procedūros ir t. t. [Nev10, FG22, Dau20]. Dėl to didelių PS, turinčių daug atskirų komponentų, padengimas kodo vieneto testais yra gana sudėtingas [VRA+22].

Šiuo metu programų sistemų testavimui taikomi tokie sprendimai kaip paieška pagrįstas PS testavimas [FG22] ir atsitiktinis testų generavimas [Has23, KPB+09]. Šie sprendimai geba sugeneruoti kodo vieneto testus, tačiau dažniausiai gautas sugeneruotas testas yra neįskaitomas ir sunkiai suprantamas [FG22, Has23, GPD+19]. Dėl to pasiūlyta naudoti didelius kalbos modelius, kurie ne tik efektyviai padengtų sistemų funkcionalumą, bet ir būtų lengvai interpretuojami. Šiuo metu programos, kurios pasitelkia didelius kalbos modelius, geba sėkmingai sugeneruoti kodą iš komentarų arba sugeneruoti likusią dalį pradėto kodo segmento [SST+24, Gri22]. Dėl to manoma, kad didelius kalbos modelius galima naudoti ir kodo vieneto testavime.

Apžvelgtuose tyrimuose DKM sugeneruoti kodo vieneto testai dažniausiai vertinami pagal kodo padengiamumo bei sugeneruoto kodo teisingumo metrikas [ATA23, TLZ+23, Has23, SST+24, GV23, HCC+24, NCP+23, WHC+23, SNE+23, BSD+22, LFN+22], rečiau vertinami pagal mutacijų įvertį [Has23, GV23, Cat25] ar testų skaitomumą [TLZ+23]. Kokybiniai kodo kriterijai yra nagrinėjami dar rečiau.

Taip pat atlikus tyrimų apžvalgą pastebėti ir kiti trūkumai. Daugelis tyrimų [ATA23, TLZ+23, Has23, SST+24, GV23, LLT+23] yra atlikti naudojant Java arba Python programavimo kalbas, C# programavimo kalba nėra ištirta. Taip pat naujesni, bendro naudojimo ir didesnį parametrų skaičių turintys DKM nėra išsamiai išnagrinėti kodo vieneto testų generavimo srityje [SST+24, SNE+23, LFN+22]. Tik keli tyrimai [SST+24, WXB+26] taiko griežtą įvesties užklausos struktūrą kartu su rezultato struktūriniu validavimu, bet šie tyrimai naudoja autorių sukurtus sprendimus, kurie susitelkia tik į specifinius klaidų tipus. Plačiai naudojamų rezultato validavimo bibliotekų, tokių kaip PydanticAI, pritaikymas kodo vieneto testų generavimo srityje dar nėra ištirtas.

Šis tyrimas siekia ištirti septynių pasirinktų DKM, įskaitant naujesnius didesnio parametrų skaičiaus modelius, generuojamų C# kodo vieneto testų kokybę, taikant griežtą struktūros validaciją,

vertinant testų kompiliavimo, vykdymo rodiklius bei jų klaidas, kodo eilučių padengimą, mutacijų įverčius, išgyvenusių mutacijų operatorių tipus bei vykdymo rodiklių ir mutacijų įverčio priklausomybę nuo ciklomatinio sudėtingumo.

Tyrimo tikslas

Ištirti struktūrinės validacijos taikymo įtaką didelio kalbos modelio generuojamų kodo vieneto testų kokybei.

Tyrimo objektas

Didelių kalbos modelių sugeneruotų C# kodo vieneto testų kokybė, taikant struktūrinę validaciją.

Tyrimo uždaviniai

1. Išanalizuoti kodo vieneto testų sudarymo ir testavimo metodus.
2. Išanalizuoti didelių kalbos modelių gebėjimo generuoti kodo vieneto testus tyrimus ir jų rezultatus.
3. Suprojektuoti tyrimo metodiką ir struktūrinę kodo vieneto testo validaciją.
4. Atlikti eksperimentus, įvertinančius sugeneruotų kodo vieneto testų kokybės metrikas.
5. Įvertinti struktūrinės validacijos įtaką didelių kalbos modelių sugeneruotų kodo vieneto testų kokybei.

Tikėtini rezultatai

Tyrimas sieks išnagrinėti automatizuoto kodo vieneto testų kūrimo metodus ir didelius kalbos modelius, kurie turi galimybę generuoti kodo vieneto testus [Has23]. Svarbi tyrimo dalis bus apibrėžti kodo vieneto testų kokybės vertinimo kriterijus ir jų vertinimo metodus, ribas [ATA23, TLZ+23, FG22, SST+24].

Vienas iš svarbiausių tyrimo uždavinių yra išsamiai ištirti kodo vieneto testų generavimą, taikant didelius kalbos modelius, ir įvertinti gautus testus pagal išmatuojamus kodo parametrus, tokius kaip kodo kompiliavimo, vykdymo rodikliai, kodo padengiamumas bei mutacijų testavimas [ATA23, FG22, SST+24].

Atlikus tyrimą, bus siekiama nustatyti, kokią įtaką griežta įvesties užklausos struktūra kartu su rezultato struktūriniu validavimu, taikant PydanticAI biblioteką, daro didelių kalbos modelių sugeneruotų C# kodo vieneto testų kokybei.

Publikuoti straipsniai

Remiantis tyrimo rezultatais parengtos mokslinės publikacijos nacionalinėms bei tarptautinėms konferencijoms „Lietuvos magistrantų informatikos ir IT tyrimai“ bei „eStream“. Konferencijos darbų leidinyje publikuoti straipsniai:

- Zapkus, D. M., & Slotkienė, A. (2026, balandžio mėn.) Impact Assessment of Structured Results for the Reliability of LLM-generated Tests. In 2026 Open Conference of Electrical, Electronic and Information Sciences (eStream) (pp. 1-7). IEEE.
- Zapkus, D. M., & Slotkienė, A. (2025). Quality evaluation of large language models generated unit tests: influence of structured output. Lietuvos magistrantų informatikos ir IT tyrimai: konferencijos darbai, 2025 m. gegužės 13 d., 281-288.
- Zapkus, D. M., & Slotkienė, A. (2024). Unit test generation using large language models: A systematic literature review. Lietuvos magistrantų informatikos ir IT tyrimai: konferencijos darbai, 2024 m. gegužės 10 d., 136-144.

Dalyvavimas konferencijose

Tyrimo literatūros analizė pristatyta stendiniu pranešimu nacionalinėje konferencijoje „Lietuvos magistrantų informatikos ir IT tyrimai“. Dovydas Marius Zapkus, Asta Slotkienė. Unit test generation using large language models: A systematic literature review. 2024 m. gegužės 10 d., Vilnius.

Tyrimo rezultatai pristatyti stendiniu pranešimu nacionalinėje konferencijoje „Lietuvos magistrantų informatikos ir IT tyrimai“. Dovydas Marius Zapkus, Asta Slotkienė. Quality evaluation of large language models generated unit tests: influence of structured output. 2025 m. gegužės 13 d., Vilnius.

Tyrimo rezultatai pristatyti žodiniu pranešimu tarptautinėje konferencijoje „eStream“. Dovydas Marius Zapkus, Asta Slotkienė. Impact Assessment of Structured Results for the Reliability of LLM-generated Tests. 2026 m. balandžio 23 d., Vilnius.

1. Literatūros apžvalga

Šiame skyriuje pateikiama literatūros apžvalga, tirianti didelius kalbos modelius ir jų gebėjimą generuoti kodo vieneto testus bei sugeneruotų testų vertinimą kokybės metrikomis.

1.1. Kodo vieneto testavimas

Kodo vieneto testavimas yra programinės įrangos kūrimo veikla, kurios tikslas patikrinti, ar atskiri programos kodo vienetai (įskaitant funkcijas, metodus, procedūras) veikia teisingai [Nev10, FG22, Dau20, Ani22]. Programinės įrangos kūrimo procesuose kodo vieneto testavimas yra viena iš standartinių praktikų [KSK+23, DF14], kuri padeda išvengti arba sumažinti klaidų skaičių galutiniame produkte. Sukurti kodo vieneto testai gali būti įvertinami pagal jų kokybės kriterijus, kurie gali nurodyti testo padengiamumą, įskaitomumą, atitikimą tam tikriems standartams.

Kodo vieneto testai gali būti kuriami rankiniu būdu paties programuotojo, tačiau egzistuoja ir automatiniai kodo vieneto testų generavimo sprendimai. Automatiniai kodo vieneto testų generavimo sprendimai yra svarbi sritis programų sistemų kūrimo procese [Pet23]. Šie sprendimai automatiškai generuoja kodo vieneto testus, taip palengvindami programuotojų darbą ir sumažinant programinės įrangos kūrimo trukmę. Vienas iš plačiausiai naudojamų automatinių kodo vieneto testų generavimo sprendimų yra paieška pagrįstas testų generavimas [TLZ+23]. Taip pat yra ir kitų sprendimų, tokių kaip atsitiktinis testų generavimas bei neseniai išpopuliarėjęs kodo vieneto testų generavimas naudojantis dideliais kalbos modeliais [Pet23].

1.2. Kodo vieneto testų kokybės metrikos

Kodo vieneto testų vertinimas gali būti atliekamas naudojantis tam tikromis testavimo programomis arba platformomis, kurios įvertina kodo vieneto testus pagal išsikeltas metrikas, t. y. skirtingos testavimo platformos gali būti naudojamos skirtingoms kodo vieneto testų metrikoms įvertinti priklausomai nuo šių programų galimybių. Šios metrikos gali būti naudojamos skirtingoms kodo vieneto testų kokybės charakteristikoms įvertinti. Kodo vieneto testų charakteristikos gali būti naudojamos vertinti kodo vieneto testų gebėjimą atpažinti klaidas, skaitomumą, atitikimą tam tikriems standartams, principams ir t. t. Dėl to yra labai svarbu ištirti ir nustatyti, kokios kodo vieneto testų metrikos yra naudojamos atliktuose tyrimuose ir apžvelgti jų tinkamumą automatinio kodo vieneto testų generavimo užduotyse.

1 lentelė. Kodo vieneto testų vertinimo metrikos ir jų apžvalga

Nr.	Kodo vieneto testų vertinimo metrika	Apžvalga	Matavimo vienetai
1.	Kodo padengiamumas	Apibrėžiamas kaip metrika, parodanti, kokia programinio kodo dalis yra padengta kodo vieneto testais [Pet23].	%, procentai, padengto programinio kodo
2.	Mutacijų testavimas	Mutacinis testavimas yra baltosios dėžės testavimo metodas, galintis atskleisti modelio sugeneruotų kodo vieneto testų rinkinio efektyvumą, t. y. atsparumą klaidoms. Mutacinis testavimas naudojamas, kai neužtenka kitų naudojamų įrankių, pvz., kodo padengimo [Pet23]. Mutacinis testavimas yra brangus laiko ir atminties atžvilgiu [GG23].	%, procentai, testo išspręstų mutantų skaičius lyginant su visu mutantų skaičiumi.
3.	BLEU reikšmė	Išmatuoja modelio sugeneruoto kodo panašumą su teisingu kodo segmentu, naudojant n gretimų simbolių sekos persidengimą [Has23]. Neatsižvelgia į sintaksinį ir semantinį sugeneruoto kodo teisingumą [RGL+20].	Skaičius, ribose nuo 0 iki 1.
4.	CodeBLEU reikšmė	Patobulinta BLEU versija naudoja svertinį n gretimų simbolių sekos persidengimą, kad prioritizuotų esminių raktažodžių svarbą (t. y. public, int, return ir t. t.) tarp sugeneruoto teksto ir teisingo tekstinio rezultato [Has23].	Skaičius, ribose nuo 0 iki 1.
5.	Kodo tikėtinos klaidos (syn. kodo kvapai)	Kodo tikėtinos klaidos nurodo, kad kodo vieneto teste yra galimų klaidų, tai gali būti programinio kodo standartų nesilaikymas, neužbaigtumas bei kitos klaidos [SST+24].	Natūralus skaičius arba 0.
6.	Ciklomatinis sudėtingumas	Nurodo kodo sudėtingumą. Pagrįsta tuo, kad kuo daugiau įdėtinių eilučių yra programiniame kode, tuo programa gali būti sudėtingesnė [EJK+23].	Natūralus skaičius arba 0.
7.	Kognityvinis sudėtingumas	Gali nusakyti kodo suprantamumą [EJK+23].	Natūralus skaičius arba 0.
8.	Atitikimas kodo standartams	Statinio kodo analizės įrankiai, turintys tam tikros programinės kalbos stilių taisyklių rinkinius. Šie įrankiai geba identifikuoti programinio kodo neatitikimą kodo standartams [TLZ+23].	Skaičius ir/arba sąrašas standartų, kurie neatitinka standartų

Kodo vieneto testų vertinimo metrikos pateikiamos ir apžvelgiamos 1 lentelėje. Viena iš dažniausiai tyrimuose naudojamų metrikų įvertinti didelių kalbos modelių sugeneruotiems kodo vieneto testams yra kodo padengiamumas [ATA23, DNM+23]. Išanalizavus tyrimus, pastebėta, kad rečiausiai automatiškai sugeneruotiems kodo vieneto testams vertinti naudojamos ciklomatinio, kognityvinio sudėtingumo bei atitikimo kodo standartams metrikos.

1.3. Dideli kalbos modeliai

Dideli kalbos modeliai yra modeliai, apmokomi dideliais tekstinių duomenų kiekiais ir gebantys generuoti žmogui suprantamus tekstinius atsakymus į klausimus bei atlikti kitas su kalba susijusias užduotis [SST+24, KSK+23]. Šie modeliai taip pat geba suprasti natūralią kalbą ir spręsti sudėtingas užduotis [BD23]. Per pastaruosius metus dėl pažangos natūralios kalbos apdorojimo srityje, susidomėjimas dideliais kalbos modeliais smarkiai išaugo. Šiuo metu DKM viešai gali būti prieinami per tokias sąsajas kaip „ChatGPT“, „Gemini“ ir t. t.

Programinio kodo generavimo gebėjimas atsirado, nes daugelis šiuolaikinių didelių kalbos modelių apmokymo metu yra mokomi ne tik natūralios kalbos, bet ir viešo kodo saugyklų duomenimis [DMN+22, LWG+22]. Tokiu būdu modelis įsisavina programavimo kalbų sintaksę, kodo struktūras, bibliotekų pavadinimus bei programinės įrangos kūrėjų naudojamus programavimo stilius. Specializuoti kodo modeliai, tokie kaip Codex, StarCoder ar CodeLlama, yra papildomai apmokomi išskirtinai kodo duomenų rinkiniais [ZZL+23], dėl to jie pasiekia geresnių rezultatų kodo generavimo užduotyse nei bendros paskirties modeliai [SST+24, LWG+22].

Kodo vieneto testų generavimo užduočiai DKM pasitelkiami, nes modeliai geba nuspėti iš testuojamojo metodo gaunamas reikšmes ir suformuoti atitinkamus teiginius šioms reikšmėms patikrinti [ATA23, YLL+23]. DKM geba generuoti sintaksiškai taisyklingą kodą pasirinkta programavimo kalba bei atsižvelgia į konkrečių testavimo karkasų taisykles ir naudojimą [SST+24, SNE+23]. Modelių sugeneruoti kodo vieneto testai paprastai yra lengviau skaitomi ir aiškesni nei paieška pagrįstų įrankių generuojami testai, kurie dažnai sukuria testus su sudėtingesne struktūra ir neprasmingais kintamųjų pavadinimais [TLZ+23, FG22].

Vis dėlto DKM turi silpnybių kodo vieneto testų generavimo užduotyje. Modeliai gali sugeneruoti sintaksiškai taisyklingą, tačiau semantiškai neteisingą kodą. Dėl DKM vadinamų „haliucinacijų“ sugeneruoti testai gali iškviesti neegzistuojančius metodus, klaidingai numatyti grąžinamas reikšmes ar netinkamai naudoti bibliotekas [SST+24, YLL+23]. Ta pati užklausa gali grąžinti skirtingus rezultatus, dėl to reikia taikyti papildomus mechanizmus kaip griežtą įvesties užklausos struktūrą, rezultato validavimą ar iteratyvius grįžtamojo ryšio metodus [SST+24, YLD+24, WXB+26, PT24].

1.4. Didelio kalbos modelio rezultato validavimas pagal apibrėžtą struktūrą

Dideli kalbos modeliai šiuo metu geba generuoti atsakymus, kurie gali būti lengvai suprantami žmogui. Tačiau norint panaudoti iš didelių kalbos modelių gautus atsakymus automatizuojant tam tikras sritis gali būti sudėtinga, nes dideli kalbos modeliai ne visada geba laikytis tam tikros struktūros, užsibrėžtos įvesties užklausoje (angl. prompt). Norint išlaikyti tam tikrą struktūrą, kai kurie tyrimai papildomai kuria tam tikrus algoritmus, kurie pašalina iš didelio kalbos modelio grąžinto rezultato nereikalingus paaiškinimus, netikslumus, pasikartojimus, taip gaunant tikslesnį rezultatą bei pasiekiant geresnius kokybės kriterijus [SST+24, YLD+24]. Dėl to manoma, kad naudojant jau egzistuojančius įrankius, kurie atlieka automatinį gauto rezultato iš didelio kalbos modelio validavimą, būtų galima pasiekti dar geresnių rezultatų.

Šiame tyrime pasirinkta naudoti plačiai naudojamą PydanticAI biblioteką, kuri atlieka rezultato gauto iš didelio kalbos modelio validavimą, taip patikrinant, ar rezultatas atitinka tam tikrą užsibrėžtą struktūrą. PydanticAI turi kelis mechanizmus, kurie patikrina modelio grąžinamą išvestį. Pirmiausia, naudojant iš anksto apibrėžtus duomenų modelius (angl. schema), biblioteka automatiškai tikrina, ar modelio sugeneruotas atsakymas atitinka numatytą duomenų tipą ir struktūrą, pavyzdžiui, ar grąžinamas objektas turi reikiamus laukus, ar jų tipai yra teisingi. Jeigu grąžintas rezultatas neatitinka apibrėžtos struktūros, PydanticAI gautą rezultatą bando apdoroti ir pašalinti nereikalingus duomenis iš gauto rezultato. Jei po šio apdorojimo rezultatas vis tiek negali būti apdorotas, PydanticAI automatiškai atlieka pakartotinį užklausos siuntimą modeliui, kartu pateikdamas klaidų aprašymą, kad modelis galėtų pataisyti savo atsakymą. Toks validavimo procesas leidžia padidinti gautų rezultatų tikslumą ir patikimumą be papildomų pastangų.

1.5. Kodo vieneto testų kompiliavimas

Kodo kompiliavimas yra aukšto lygio programavimo kalba parašytos programos vertimas į kompiuterinę kalbą arba jai artimą, įtraukiant į programą jos veikimui reikalingus pagalbinius komponentus [DGJnd]. Dėl to, kodo vieneto testų pritaikymui, naudojimui bei įvertinimui labai svarbu atlikti kodo vieneto testų kompiliaciją, kuri parodo, ar sugeneruotas kodas gali būti paverstas į mašininį kodą ir įvykdytas.

Kompiliavimo etapas gali nusakyti didelių kalbos modelių sugeneruotų kodo vieneto testų kokybę. Sugeneruotas kodo vieneto testas vizualiai gali atrodyti teisingas, tačiau kompiliavimo metu gali būti aptiktos įvairios klaidos: neteisingi kintamųjų tipai, neegzistuojančių metodų ar klasių naudojimas, trūkstamos priklausomybės arba netinkamas testavimo karkaso naudojimas. Kodo vieneto testas su kompiliavimo klaidomis negali būti įvykdytas neatlikus papildomų veiksmų šioms

klaidoms išspręsti.

Didelių kalbos modelių kompiliavimo sėkmės rodiklis leidžia įvertinti, kiek modelio sugeneruotas kodas yra praktiškai panaudojamas. Modelis gali sugeneruoti didelį kiekį kodo vieneto testų, tačiau jei reikšminga jų dalis nekompiliuojama, tolimesnis testavimo procesas gali būti nenaudingas. Reikia suprasti, kad nesikompiliuojantys testai negali būti naudojami kituose testavimo procesuose, dėl to jie yra ignoruojami arba atmetami. Sugeneruotų kodo vieneto testų kompiliavimas taip pat gali parodyti modelio gebėjimą suprasti ne tik testuojamo kodo logiką, bet ir konkrečios programavimo kalbos sintaksės bei naudojamų bibliotekų specifiką, žinias.

Sėkmingas kompiliavimas patvirtina, kad kodas yra sintaksiškai ir struktūriškai tinkamas vykdymui, tačiau nepatvirtina testo prasmingumo bei atsparumo klaidoms. Todėl kompiliavimo rezultatai šiame tyrime nagrinėjami kartu su kodo vieneto testų vykdymo ir mutacinio testavimo rezultatais, siekiant gauti išsamesnį vaizdą apie kiekvieno didelio kalbos modelio gebėjimą generuoti kokybiškus kodo vieneto testus.

1.6. Kodo vieneto testų vykdymas

Nesėkmingai įvykdyti kodo vieneto testai nurodo priežastį arba klaidą, dėl kurios kodo vieneto testas negali būti sėkmingai įvykdytas. Kiekviena klaida suteikia detalią informaciją bei priežastį, kodėl testas negalėjo būti įvykdytas. Siekiant klasifikuoti aptiktas klaidas, kiekviena klaida buvo priskirta tam tikrai kategorijai pagal jos kilmę ir pobūdį. Šios klaidos ir jų paaiškinimai yra aprašomi žemiau pateiktoje lentelėje:

2 lentelė. C# programavimo kalbos kodo vykdymo klaidos

Klaida	**Aprašymas**
Assert [Netnd, Fel24]	Klaidos skirstomos į kelis tipus: • *Assert.Equal* - reikšmių lyginimas nesutapo. • *Assert.Throws* - gautas klaidos kodas nesutapo su apibrėžta klaida. • *Assert.True* - negrąžino *bool: True* tipo rezultato. • *Assert.False* - negrąžino *bool: False* tipo rezultato. • *Assert.Null* - negrąžino *null* tipo rezultato. • *Assert.Single* - negrąžino sąrašo, kuriame yra vienas įrašas. • *Assert.Contains* - grąžino kintamąjį arba sąrašą, kuriame nebuvo sąlygoje nurodyto kintamojo/įrašo. • *Assert.Empty* - negrąžino tuščio kintamojo arba sąrašo.

IndexOutOfRange [Micnda]	Kodas bando pasiekti masyvo arba sąrašo elementą naudodamas indeksą, kuris neegzistuoja duomenų struktūroje.
InvalidOperation [Micndb]	Metodas iškviečiamas tinkamu argumentu, tačiau objekto būsena neleidžia sėkmingai įvykdyti metodo.
ArgumentOutOfRange [Micndc]	Metodas yra iškviečiamas su argumentu, kurio reikšmė nepriklauso leistinų reikšmių aibei.
NullReference [Micndd]	Kodas bando pasiekti objektą arba jo dalį per kintamąjį, kurio reikšmė yra null, kintamasis nerodo į jokį objektą atmintyje, yra neinicijuotas.

Programavimo kalbos, įskaitant C# kalbą, turi didelį klaidų pranešimų skaičių, dėl to, klaidos, aprašytos 2 lentelėje, buvo pasirinktos dėl jų dominavimo šiame tyrime. Klaida *Assert* yra dalis XUnit karkaso, kuris yra vienas iš dažniausiai pasirenkamų C# programavimo kalbos testavimo įrankių, naudojamų kodo vieneto testams kurti ir vykdyti [Sab24]. Dėl XUnit populiarumo, šį karkasą buvo pasirinkta naudoti šiame tyrime. Dideli kalbos modeliai kaip reikalavimą turėjo sukurti kodo vieneto testus būtent pagal XUnit apibrėžtį.

Assert klaidos XUnit kontekste atsiranda tuomet, kai testuojamo kodo grąžinama reikšmė neatitinka testo metu nurodytos reikšmės. XUnit karkase *Assert* metodai, tokie kaip *Assert.Equal(), Assert.True()* ar *Assert.NotNull()*, yra pagrindinė priemonė tikrinti, ar testuojamas kodas grąžina teisingas, programuotojo apibrėžtas reikšmes. Jei didelis kalbos modelis sugeneruoja testą su neteisinga reikšme, vykdymo metu yra išmetama *Assert* klaida, testas pažymimas kaip nepavykęs. Tai nurodo, kad modelis arba neteisingai interpretavo testuojamos funkcijos logiką, arba negalėjo tiksliai numatyti, kokią reikšmę funkcija grąžins konkrečiu įvesties atveju. Šių klaidų analizė parodo didelių kalbos modelių gebėjimą generuoti teisingus testus, kurie pasižymi loginiu teisingumu.

1.7. Kodo vieneto testų mutacijų testavimas

Mutacinis testavimas yra programinės įrangos testavimo metodika, kuria siekiama įvertinti testų rinkinio atsparumą klaidoms, keičiant atsitiktines testuojamos programinės įrangos dalis, vadinamas mutacijomis [Has23, Pet23, PKZ+19]. Atlikti tyrimai atskleidžia, kad mutacijų testavimas yra veiksmingas metodas, leidžiantis įvertinti kodo vieneto testų kokybę ir nustatyti perteklinius kodo vieneto testus [SDM+22].

Mutacijų įvertis atspindi testavimo atvejų efektyvumą, apskaičiuojant sunaikintų (angl. killed) mutacijų skaičiaus santykį su visų mutacijų skaičiumi [DNM+23, JH10, DMN+22]. Mutacijos yra sunaikintos, jei testavimo atvejis baigiasi nesėkme arba negrąžina jokio rezultato, tokiu atveju yra

daroma prielaida, kad kodo mutacija buvo aptikta testu. Priešingu atveju, jei kodo vieneto testai neaptiko kodo mutacijos, laikoma, kad mutacija išgyveno (angl. survived). Mutacijų įvertis skaičiuojamas formule:

$$M_{\text{įvertis}} = \frac{Mut_{sunaik}}{Mut_{išgyv}+Mut_{sunaik}} \times 100 \quad (1)$$

Mut$_{sunaik}$ - sunaikintų mutacijų skaičius. Mutacija laikoma sunaikinta, kai bent vienas kodo vieneto testas baigiasi nesėkme arba negrąžina rezultato vykdant mutavusį kodą.

Mut$_{išgyv}$ - išgyvenusių mutacijų skaičius. Mutacija laikoma išgyvenusia, kai visi kodo vieneto testai sėkmingai praeina, nepaisant kode atliktos modifikacijos, t. y. testai neaptiko įvestos klaidos.

Šiame tyrime naudojama išsamesnė Stryker.NET įrankio mutacijų įverčio lygtis, kuri buvo naudojama visiems sugeneruotiems kodo vieneto testų bei testuojamojo kodo mutacijų įverčiams skaičiuoti:

$$M_{\text{įvertis}} = \frac{Mut_{sunaik}+Mut_{laikas}}{(Mut_{išgyv}+Mut_{sunaik}+Mut_{laikas}+ Mut_{nepad})} \times 100 \quad (2)$$

Mut$_{laikas}$ - laiko limitą viršijusių mutacijų skaičius. Mutacija priskiriama šiai kategorijai, kai kodo vieneto testo vykdymas su mutavusiu kodu užtrunka ilgiau nei nustatytas laiko limitas.

Mut$_{nepad}$ - kodo nepadengtų mutacijų skaičius. Mutacija priskiriama šiai kategorijai, kai mutavusi kodo eilutė nėra padengta nei vieno kodo vieneto testo.

Mutacijos yra sudaromos iš operatorių. Kiekvienas mutacijos operatorius apibrėžia konkrečią kodo pakeitimo taisyklę [JH10], t. y. nurodo, kokiu būdu originalus kodo fragmentas yra pakeičiamas, siekiant imituoti galimą programuotojo klaidą. Skirtingi operatorių tipai leidžia testuoti įvairias kodo klaidos formas: nuo aritmetinių ir loginių operatorių pakeitimų iki sąlyginių išraiškų ir grąžinamų reikšmių modifikavimų [MO05]. 3 lentelėje pateikiami šiame tyrime dažniausiai pasikartoję išgyvenusiose mutacijose operatorių tipai bei jų aprašai:

3 lentelė. Mutacijų testavimo mutantų operatorių tipai ir jų aprašai

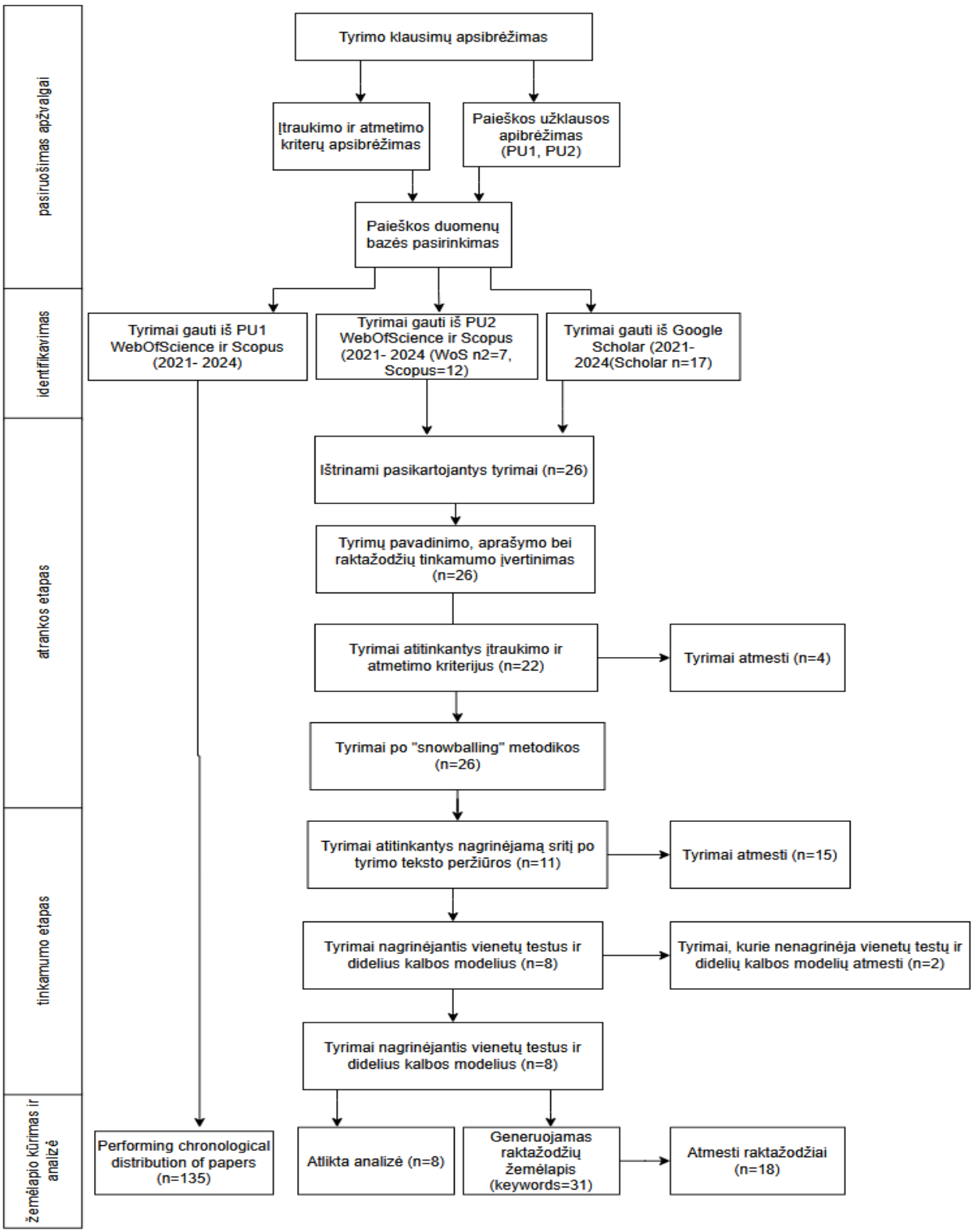
Operatorius	**Aprašymas**
Aritmetinis (angl. arithmetic)	Pakeičia kode esančių aritmetinių veiksmų operatorius. *+, -, /, *, %*
Lygybės (angl. equality)	Pakeičia kode esančius lygybės operatorius tokius kaip *=, <, >, !=, ==.*

Loginių operatorių (angl. logical)	Pakeičia kode esančius loginius operatorius į tokius kaip *||, &&, ^, == ir t. t.*
Būsenos sąlygų (angl. boolean)	Pakeičia kode esančias būsenų sąlygas. *True, False*
Priskyrimo (angl. assignment)	Pakeičia kode esančius priskyrimo veiksmus. *+=, *=, -=, /= ir t. t.*
Tekstinis (angl. string)	Pakeičia kode esančius tekstinius kintamuosius į kitas reikšmes arba funkcijas.
Bitwise	Pakeičia *bitwise* operacijas į tokias kaip <<, >>, &, | ir t. t.

Pagrindiniai operatoriai, naudojami Stryker.NET įrankyje ir dominuojantys šiame tyrime, atvaizduoti 3 lentelėje. Mutantai testuojamajame kode sukuriami naudojant šių operatorių aprašytas taisykles, t. y. kaip pavyzdys, aritmetinis operatorius kodo segmente naudos aritmetinių operatorių varijavimą, keis aritmetinių veiksmų simbolius ir stebės, ar kodo vieneto testai geba pamatyti šį pokytį ir sukelti klaidą.

1.8. Didelių kalbos modelių naudojimas kodo vieneto testavimo užduotims

Komponentų testams generuoti vis dažniau pasirenkami naudoti dideli kalbos modeliai, tačiau dėl riboto tyrimų skaičiaus šioje srityje buvo nuspręsta išnagrinėti, kokie dideli kalbos modeliai yra naudojami kodo generavimo užduotims ir kodo vieneto testų generavimo užduotims. Abstraktus terminas „code generation“ buvo pasirinktas, kad būtų gauta didesnė apimtis tyrimų, tokiu būdu pagerinant šio tyrimo tikslumą. „Kodo vieneto testų generavimo“ sritis yra dalis „kodo generavimo užduočių“ [Kho20], dėl to, galime daryti prielaidą, kad didelių kalbos modelių galimybė sugeneruoti kodo segmentus leidžia taip pat sugeneruoti ir kodo vieneto testus. Tyrimams, apimantiems kodo generavimą ir didelius kalbos modelius gauti, buvo pasitelkta sistematinės apžvalgos metodika, kuri buvo sukurta ir įvykdyta pasitelkiant gaires, pasiūlytas Kitchenham ir Brereton [KPB+09, KB13].



1 pav. Sistematinės literatūros apžvalgos procesas

Metodikos struktūra (žr. 1 pav.) sudaryta iš penkių etapų: (1) pasiruošimas apžvalgai, (2) identifikavimas, (3) atrankos etapas, (4) tinkamumo etapas, (5) žemėlapio kūrimas ir analizė (žr. 1 paveikslą).

Tyrimo rezultato tikslumui padidinti buvo apibrėžti įtraukimo ir atmetimo kriterijai pateikiami žemiau:

4 lentelė. Įtraukimo/atmetimo kriterijai

Tyrimų įtraukimo kriterijai (IK)	Tyrimų atmetimo kriterijai (AK)
IK1. Tyrimai buvo paskelbti po 2021 metų. **IK2.** Tyrimai turi būti parašyti anglų kalba. **IK3.** Tyrimas yra pirminė literatūros analizė. **IK4.** Tyrimai turi nagrinėti didelių kalbos modelių gebėjimą generuoti kodo vieneto testus. **IK5.** Tyrimai priklauso programų sistemų arba informatikos sritims. **IK6.** Tyrimai yra empirinio tipo.	**AK1**. Tyrimai, atkartojantys kitus tyrimus, jeigu jie priklauso tam pačiam autoriui. **AK2**. Mokslinis darbas, kuris nesusijęs su dideliais kalbos modeliais ir kodo vieneto testais arba didelių kalbos modelių gebėjimas generuoti kodo vieneto testus nėra aptariamas mokslinio darbo turinyje. **AK3**. Tyrimas, kurio turinys neparašytas anglų kalba. **AK4**. Moksliniai darbai, nesuteikiantys prieigos prie pilno darbo turinio.

Paieškos strategija įtraukia du pagrindinius terminus, kurie padeda apsibrėžti paieškos užklausas. Galutinės paieškos užklausos yra apibrėžiamos naudojantis reikalavimais iš WoS, Scopus ir Google Scholar duomenų bazių ir pateikiamos žemiau:

PU1 (Paieškos užklausa 1) yra sukurta ir pritaikyta pagal: pirminių paieškos raktažodžių išskyrimą atsižvelgiant į pagrindinį tyrimo klausimą; Didelių kalbos modelių alternatyvų paiešką (DKMAlternatyvos) tokių kaip GPT-4, Codex ir t. t.

PU1: *DKMAlternatyvos AND code AND (generation OR automation)*

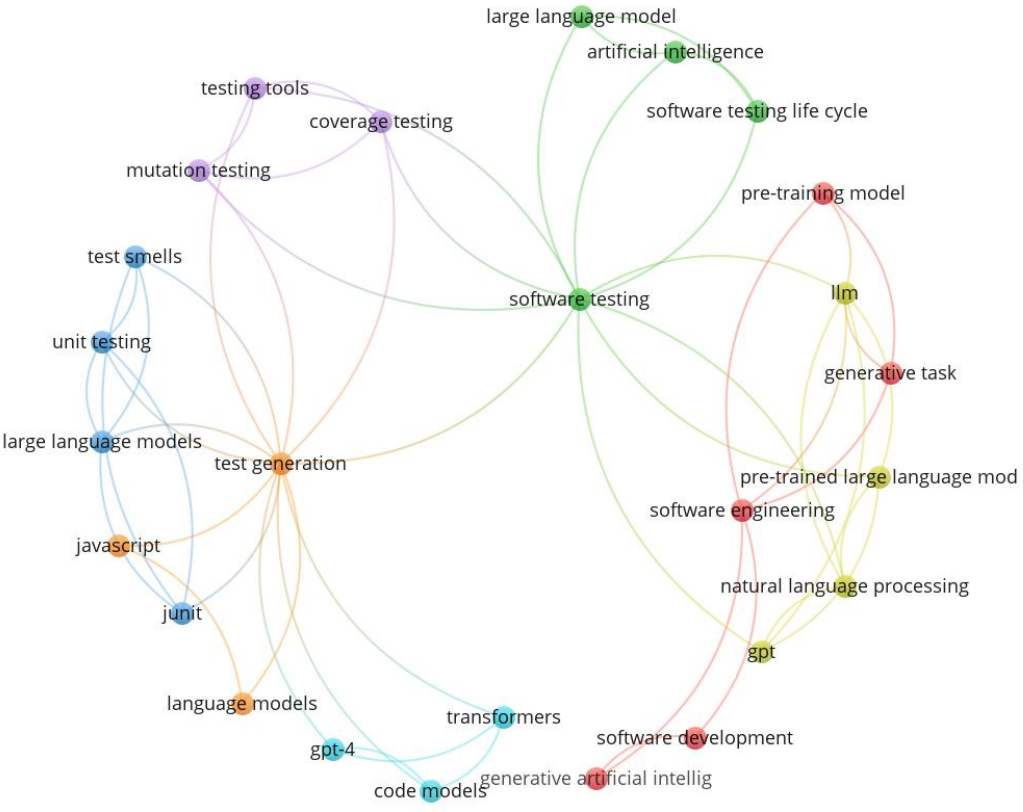
PU2 (Paieškos užklausa 2) išrenka ir padengia terminus, susijusius su kodo vieneto testais: kodo vieneto testai ir komponentų testai ir terminus, susijusius su dideliais kalbos modeliais: didelis kalbos modelis (angl. LLM).

PU2: *(“unit test*” OR “component test*”)*
AND (“large language model” OR LLM OR LLM's)*

Identifikavimo etape buvo atlikta tyrimų paieška, naudojantis apibrėžta paieškos užklausa, trijose mokslinėse duomenų bazėse: WoS, Scopus ir Google Scholar, naudojantis šiais tyrimų įtraukimo ir atmetimo kriterijais. Atlikus paiešką, naudojantis išsikeltomis paieškos užklausomis, buvo gauti 36 tyrimai nuo 2021 iki 2024 metų. Dėl trijų skirtingų mokslinių tyrimų duomenų bazių naudojimo, reikia atsižvelgti į galimą tyrimų pasikartojimą. Dėl to, pasikartojantys tyrimai buvo pašalinti (pašalinta 10 pasikartojančių tyrimų). Taip pat, pašalinti tyrimai, neatitikę įtraukimo ir

atmetimo kriterijų (4 tyrimai atmesti). Atlikus išsamesnę gautų mokslinių tyrimų analizę, buvo pastebėta, kad kai kurie mums žinomi tyrimai, gauti rankiniu būdu, nėra įtraukti į paieškos rezultatus. Dėl to, šiame tyrime naudojama „snowball sampling“ metodika ir buvo pridėti keli aktualūs moksliniai tyrimai (4 tyrimai), kurie įtraukia apibrėžtus terminus ir pagrindinį tyrimo klausimą. Raktažodžių žemėlapio analizė, sudaryta iš 8 mokslinių straipsnių, parodė, kad yra poreikis pritaikyti aktualumo analizę pilnos sudėties dokumentams. Kiekvieno tyrimo turinio tinkamumas buvo įvertintas (15 atmesta dėl neatitikimo) ir kiekvienas tyrimas buvo patikrintas atsižvelgiant, ar mokslinio darbo metu atliekamas tyrimas yra didelių kalbos modelių kodo vieneto testų generavimo srityje (2 atmesti).

Šiame darbe sugeneruotas raktažodžių žemėlapis, naudojantis straipsniais, gautais iš 1 paveikslo atvaizduoto proceso. Raktažodžių žemėlapis buvo pasirinktas dėl gebėjimo generuoti raktažodžius į klasterius, tokiu būdu palengvinant analizės procesą. Pradinis rezultatas turėjo 49 unikalius raktažodžius. Buvo pastebėta, kad kai kurie raktažodžiai neturėjo sąsajos su didelių kalbos modeliais arba kodo vieneto testais, kelių šių raktažodžių pavyzdžiai: apibūdinantys tyrimo tipą („review“, „literature survey“, „literature review“), abstraktūs raktažodžiai („agenda“, „group“, „focus“, „codes”). Pašalinus neaktualius raktažodžius, raktažodžių žemėlapis, atvaizduojantis 31 raktažodį, buvo sugeneruotas.



2 pav. Didelių kalbos modelių ir kodo vieneto testų generavimo raktažodžių žemėlapis

Sugeneruotas raktažodžių žemėlapis (žr. 2 paveikslą) siekia paaiškinti, su kuo siejami dideli kalbos modeliai ir kodo vieneto testų generavimas. Iš raktažodžių žemėlapio galima pastebėti, kad susiformavo 6 klasteriai. Dažniausiai pasikartojantys raktažodžiai yra „test generation“ ir „software testing“.

Daugiausiai asociacijų su kitais klasteriais turėjo “software testing” raktažodis. “Test generation” raktažodis yra susijęs su „gpt-4”, „large language models“, „code models“ raktažodžiais, kurie pažymi tyrimuose naudotų įrankių arba modelių gebančių sugeneruoti kodo vieneto testus tipą. „Test generation“ raktažodis taip pat yra susijęs su kodo vieneto testų kokybės vertinimo metodikomis, kurios yra violetinės spalvos klasteryje. „Software testing“ turi ryšį su „LLM“ raktažodžiu, kuris nurodo, kad didelė dalis tyrimų naudojo didelius kalbos modelius testuoti programinei įrangai.

Remiantis istorine apžvalga, galima teigti, kad didelių kalbos modelių gebėjimas generuoti kodo vieneto testus yra nauja sritis, kuri neturi pakankamo skaičiaus tyrimų, kad būtų išsamiai išnagrinėta, tačiau tai yra vis populiarėjanti sritis, kuri nuo 2023 metų turėjo didžiausią tyrimų skaičių nagrinėjančių DKM ir kodo vieneto testus. Daugelyje tyrimų buvo naudojami Codex ir GPT-3 modeliai.

1.9. Didelių kalbos modelių kodo vieneto testų generavimo tyrimų apžvalga

Tyrimuose pastebima (žr. 4 priedą), kad didelių kalbos modelių gebėjimas generuoti kodo vieneto testus gali būti reikšmingai pagerintas pateikiant modeliui papildomą kontekstą bei taikant grįžtamojo ryšio mechanizmus. Schäfer ir kt. [SNE+23] tyrime modeliui pateikiamas ne tik testuojamo metodo aprašas ir integravimas, bet ir jo panaudojimo pavyzdžiai iš dokumentacijos. Nepavykus įvykdyti sugeneruoto testo, užklausa modeliui kartojama kartu su klaidos pranešimu. Toks metodas leido pasiekti didesnes vidutines teiginių bei medžio šakų aprėptis ir pranoko pažangiausius testų generavimo metodus. Lahiri ir kt. [LFN+22] pristato interaktyvų testais pagrįstą kodo generavimą, kuris pasitelkia naudotojo atsiliepimus kodo pasiūlymams filtruoti ir prioritizuoti. Taikant šį metodą su OpenAI Codex modeliu, pass@1 kodo generavimo tikslumas išaugo nuo 22.49 % iki 37.71 % MBPP testo atveju ir nuo 24.79 % iki 53.98 % HumanEval duomenų rinkinio atveju.

Tyrimuose pastebėtas ir modelių adaptavimas konkrečiai sričiai (angl. fine-tuning). Hashtroudi [Has23] tyrime siūlomas testavimo karkasas parodė, kad DKM pritaikymas tam tikrai sričiai vidutiniškai 49.9 % pagerina kodo eilučių aprėptį. Alagarsamy ir kt. [ATA23] tyrimas pristatė A3Test metodą, kuris papildytas teiginių (angl. assertion) žiniomis, pavadinimų nuoseklumo ir testų antraščių tikrinimo mechanizmu. Šis tyrimas įvertintas su 5278 metodais iš Defects4j duomenų rinkinio ir

sugeneravo 147 % daugiau teisingų testų bei padengė 15 % daugiau kodo nei kiti testavimo karkasai (AthenaTest).

Reikšmingą tyrimų dalį sudaro DKM ir tradicinių paieška pagrįstų testų generavimo įrankių, tokių kaip EvoSuite, palyginimai. Tang ir kt. [TLZ+23] tyrimas atskleidė, kad nors 69.6 % ChatGPT sugeneruotų komponentų testų gali būti sėkmingai sukompiliuoti ir įvykdyti, EvoSuite vis dar geba sugeneruoti testus, padengiančius 19 % daugiau kodo. Be to, didžioji dalis ChatGPT sugeneruotų testų pasižymėjo mažu sudėtingumu. Guilherme ir Vincenzi [GV23] tyrime, vertinant Java programavimo kalba sugeneruotus testus naudojant gpt-3.5-turbo modelį, nustatė, kad DKM sugeneruotų testų rinkinys prilygsta tradiciniams Java testų generavimo įrankiams. Siddiq ir kt. [SST+24] išnagrinėjo trijų modelių (Codex, GPT-3.5-Turbo ir StarCoder) galimybes ir nustatė, kad Codex modelis pasiekė daugiau nei 80 % kodo padengiamumo su HumanEval duomenų rinkiniu, tačiau nė vienas modelis nesugebėjo pasiekti net 2 % padengiamumo su EvoSuite SF110 testu. Be to, sugeneruoti testai turėjo pasikartojančių teiginių ir tuščių testų klaidų.

Tik keli tyrimai naudoja griežtą įvesties užklausos (angl. prompt) struktūrą kartu su automatiniu rezultato struktūriniu validavimu [SST+24, YLD+24, WXB+26]. Šiuose tyrimuose rezultato validavimas atliekamas autorių sukurtais sprendimais, susitelkiant į labai specifinį iš didelių kalbos modelių gaunamuose vieneto testuose pasitaikančių klaidų tipą ir neatsižvelgiant į platesnį klaidų spektrą. Plačiai naudojamų rezultato validavimo bibliotekų, tokių kaip PydanticAI, pritaikymas kodo vieneto testų generavimo srityje dar nėra ištirtas.

Analizuojant duomenų šaltinius, dažniausiai pasitelkiami HumanEval [SST+24, LFN+22] bei Defects4J [ATA23, TLZ+23, Has23] duomenų rinkiniai. Kiti šaltiniai apima Methods2test [Has23], GCPY ir APPS Benchmark [LWG+22], MBPP našumo testą [LFN+22] bei specializuotus rinkinius: 25 npm paketus su 1 684 API funkcijomis [SNE+23] ar 33 Java programų rinkinį [GV23].

Analizuojant tyrimuose naudojamas programavimo kalbas, pastebima, kad dažnai naudojamos Java [ATA23, TLZ+23, Has23, SST+24, GV23] bei Python [LWG+22, LFN+22] programavimo kalbos. Kitos programavimo kalbos yra apžvelgiamos rečiau [SNE+23]. Daugelis duomenų šaltinių yra sukurti naudojant populiaresnes programavimo kalbas (pvz., Defects4J Java atveju ir HumanEval bei MBPP Python atveju), dėl to galimai kitos programavimo kalbos, įskaitant C#, lieka mažiau ištirtos.

Remiantis analizuotais tyrimais galima teigti, kad paieška grįsti kodo vieneto testų generavimo algoritmai, tokie kaip EvoSuite, gali būti pranašesni už DKM, tačiau kombinuojant šiuos metodus su dideliais kalbos modeliais ir papildomai apmokant modelius galima reikšmingai pagerinti kodo padengiamumo ir mutacijų įverčio reikšmes [Has23]. Literatūros analizė taip pat atskleidžia spragų:

griežtos įvesties užklausos struktūros bei automatinio rezultato validavimo, naudojant plačiai prieinamas bibliotekas, taikymas yra neištirtas, taip pat programavimo kalbos, tokios kaip C#, nėra apžvelgtos.

1.10.Sugeneruotų testų vertinimo kokybės metrikų apžvalga

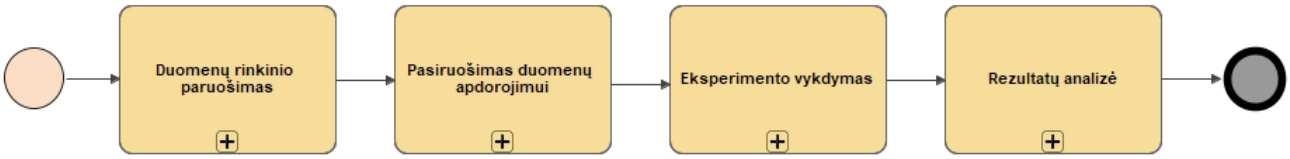
Išanalizavus tyrimuose taikytas kokybės vertinimo metrikas (žr. 5 priedą), pastebima, kad dažniausiai nagrinėjama kokybės metrika yra kodo padengiamumas, kuris nagrinėtas daugumoje analizuotų tyrimų [ATA23, TLZ+23, Has23, SST+24, GV23]. Schäfer ir kt. [SNE+23] vertinimui pasirinko detalesnes metrikas: teiginių aprėptį bei medžio šakų aprėptį, kurios suteikia išsamesnį vaizdą apie sugeneruotų testų kokybę nei kodo eilučių padengimas. Hashtroudi [Has23] tyrime, be kodo eilučių padengiamumo, naudotos BLEU ir CodeBLEU reikšmės sugeneruoto kodo panašumui vertinti.

Vis dėlto, sugeneruoto kodo skaitomumas ir atitikimas programavimo kalbos standartams nagrinėjamas itin retai. Tik vienas iš apžvelgtų tyrimų [TLZ+23] vertino sugeneruotų testų skaitomumą naudojant Checkstyle-Google ir Checkstyle-SUN įrankius, taip pat skaičiavo klaidų skaičių testuose. Taip pat tik keli tyrimai analizavo sugeneruotų kodo vieneto testų mutacijų reikšmę [Has23, GV23]. Siddiq ir kt. [SST+24] tyrime papildomai vertintos kodo tikėtinos klaidos.

Apibendrinant galima teigti, kad kodo padengiamumas išlieka dažniausiai naudojama kodo vieneto testų vertinimo metrika. Kitos kodo vieneto testų vertinimo metrikos, tokios kaip skaitomumas, atitikimas kodo standartams, mutacijų įvertis, ciklomatinis bei kognityvinis sudėtingumas, yra naudojamos retai. Dėl to galima teigti, kad daugelis tyrimų visapusiškai neįvertina sugeneruotų kodo vieneto testų kokybės.

2. Tyrimo metodika

Šiame skyriuje pateikiama tyrimo metodika ir pateikiamos kokybės metrikos, gautos iš sugeneruotų kodo vieneto testų bei išvados. Tyrimo metodika suskirstyta į keturis pagrindinius procesus:



3 pav. Bendra tyrimo metodologijos schema, atvaizduojanti tyrimo eigą, jos žingsnius

Tyrimo metodika skirstoma į 4 pagrindinius procesus (žr. 3 paveikslą). Pirmasis procesas aprašo duomenų rinkinio paruošimą. Duomenų rinkinio paruošimas apima testuojamojo kodo gavimą, kompiliavimą bei pateikimą tolimesniems procesams. Antrasis procesas *Pasiruošimas duomenų apdorojimui* aprašo procesus, kurie turi būti įvykdyti prieš eksperimento įvykdymą. Tolesnis procesas aprašo eksperimento vykdymą, įskaitant kodo vieneto testų generavimą. Ketvirtasis procesas yra orientuotas į rezultatų išsaugojimą bei analizę.

2.1. Duomenų rinkinio paruošimas

Pirmasis tyrimo žingsnis buvo sudarytas iš duomenų rinkinio paruošimo. Duomenų rinkinys buvo gautas iš tyrimo, nagrinėjančio C# kodo segmentus. Naudotas duomenų rinkinys buvo sudarytas iš 310 skirtingų *.cs* (C# formato) kodo failų. Kiekvienas kodo failas talpino po vieną C# kodo klasę, įskaitant ir testuojamąjį metodą arba kitaip kodo vienetą, esantį klasėje.

```
public string Sort(string stringToSort) {
            char[] charactersToSort = stringToSort.ToCharArray();
            for (int i = 0; i < charactersToSort.Length - 1; i++) {
                for (int j = 0; j < charactersToSort.Length - i - 1; j++) {
                    if (charactersToSort[j] > charactersToSort[j + 1])
                    {
                        char temp = charactersToSort[j];
                        charactersToSort[j] = charactersToSort[j + 1];
                        charactersToSort[j + 1] = temp;
                    }
                }
            }
            return new string(charactersToSort);
        }
```

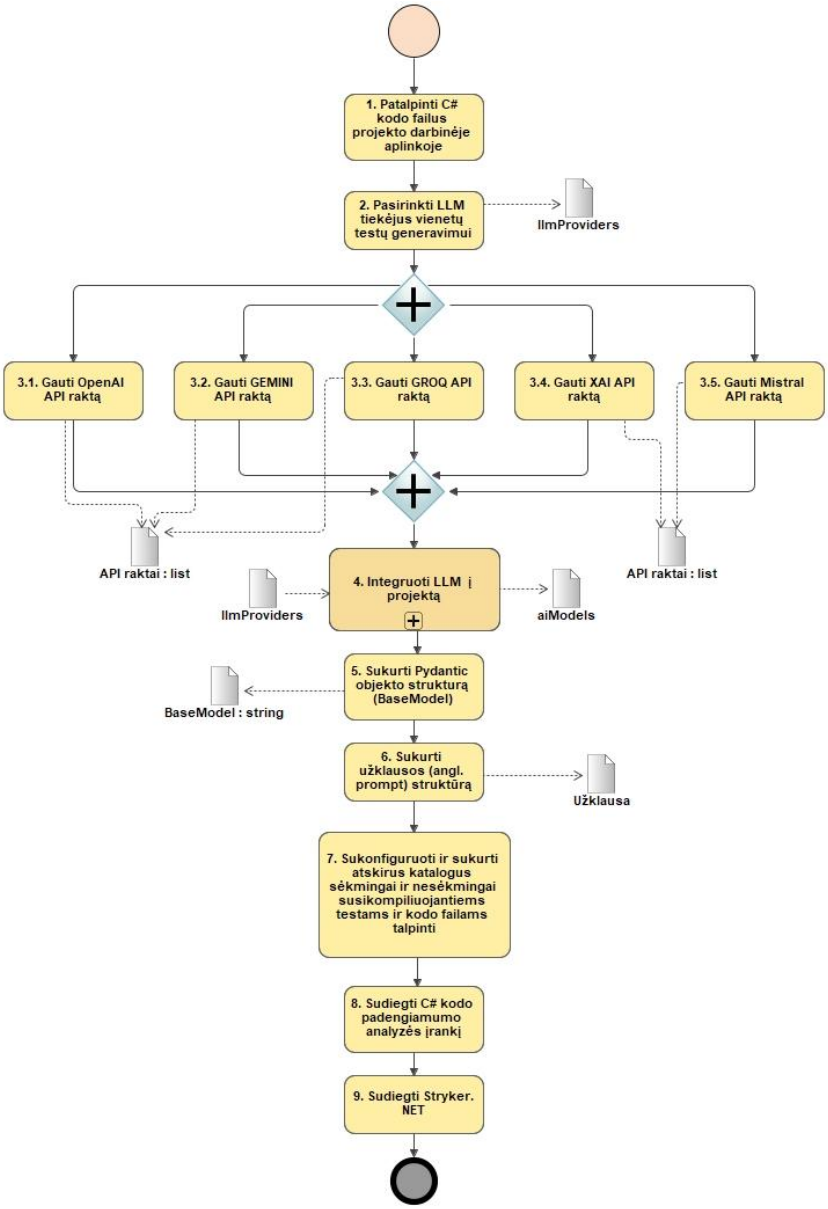
4 pav. Testuojamojo kodo vieneto pavyzdys

Eksperimento metu naudotas kodo vienetas kaip pavyzdys atvaizduotas 4 paveiksle. Iš viso buvo 310 kodo vienetų, kuriems buvo kuriami kodo vieneto testai. Kodo failas sudarytas iš naudojamų

bibliotekų, klasės ir klasėje esančio kodo vieneto. Visais atvejais kodas buvo sudarytas tik iš vienos klasės ir vieno testuojamojo metodo, esančio šioje klasėje. Kiekvieno iš šių kodo failų kompiliavimas buvo patikrintas, naudojantis dotnet 8.0 kompiliatoriumi, tokiu būdu siekiant pašalinti klases ir metodus, kurie turi sintaksinių klaidų. Visi metodai iš duomenų rinkinio susikompiliavo sėkmingai, dėl to, nebuvo imtasi papildomų veiksmų siekiant pašalinti kodo failus, kurie turi klaidų.

2.2. Pasiruošimas duomenų apdorojimui

Pasiruošimo duomenų apdorojimui procesas siekia paruošti visus svarbiausius duomenis prieš eksperimento vykdymą. Žemiau pateikiama diagrama su išsamiau atvaizduotais pasiruošimo duomenų apdorojimui proceso žingsniais:



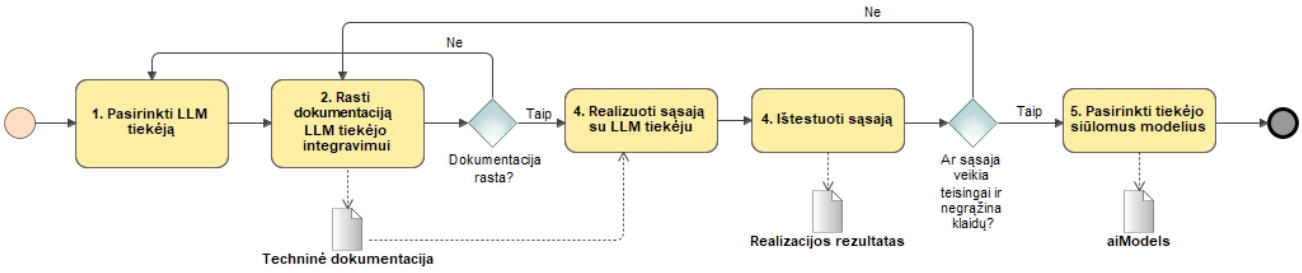
5 pav. Pasiruošimo duomenų apdorojimui proceso diagrama

Pirmasis procesas, 5 paveiksle, nurodo, kad susikompiliuojantys C# kodo failai yra perkeliami į darbinį projekto/eksperimento katalogą, iš kurio eksperimento procesas pasiims kodo failus tolimesniam apdorojimui.

Antrasis procesas nurodo, kokie didelių kalbos modelių tiekėjai bus naudojami eksperimento metu. Šio eksperimento metu buvo pasirinkta naudoti išorinius API tokių tiekėjų, kaip Google, OpenAI, Mistral, Groq, xAI. Šie tiekėjai suteikia vartotojui prieigą naudotis jau apmokytais dideliais kalbos modeliais, veikiančiais duomenų centruose, už tam tikrą mokestį. Išoriniai tiekėjai pasirinkti siekiant geresnių rezultatų ir išvengti resursų trūkumų. Iš šių tiekėjų iš viso buvo pasirinkti 7 modeliai: gemini-2.0-flash, gemini-2.5-flash, devstral-small-2505, gpt-4.1, gpt-4o-mini, grok-4, llama-3.3-70b-versatile.

Procesai 3.x aprašo API raktų gavimą iš DKM API tiekėjų. Didelių kalbos modelių tiekėjai kiekvienam iš vartotojų suteikia individualų API raktą, kuris yra būtinas, norint naudotis tiekėjų paslaugomis. Tyrinėjant skirtingus DKM, dažniausiai gaunami keli API raktai iš skirtingų API tiekėjų. Šis procesas yra atliekamas rankiniu būdu pačio vartotojo. Gavus raktus, jie yra patalpinami į API raktų sąrašą.

Procesas 4 aprašo DKM integravimą į eksperimentą. DKM integracija suteikia galimybę sąveikauti su modeliu naudojantis *Application Programming Interface* (API) arba bendrauti su lokaliai veikiančiais dideliais kalbos modeliais.



6 pav. DKM integravimo į projektą diagrama

DKM integravimo į eksperimentą eiga atvaizduota 6 paveiksle. Integracija pradedama nuo dokumentacijos paieškos. Didelių kalbos modelių tiekėjai nurodo informaciją, kaip yra vykdomas prisijungimas prie jų sąsajos. Suradus šią informaciją ir naudojantis dokumentacija, yra realizuojama sąsaja tarp eksperimentinio projekto ir DKM tiekėjo. Jei realizacija veikia teisingai, be klaidų, yra pasirenkami tiekėjo siūlomi modeliai, kurie bus naudojami eksperimentui vykdyti.

Procesas 5 naudojamas sukurti PydanticAI objekto struktūrą. Ši struktūra yra sukuriama rankiniu būdu ir suteikia galimybę validuoti iš DKM gautą rezultatą, t. y. ar rezultatas atitinka

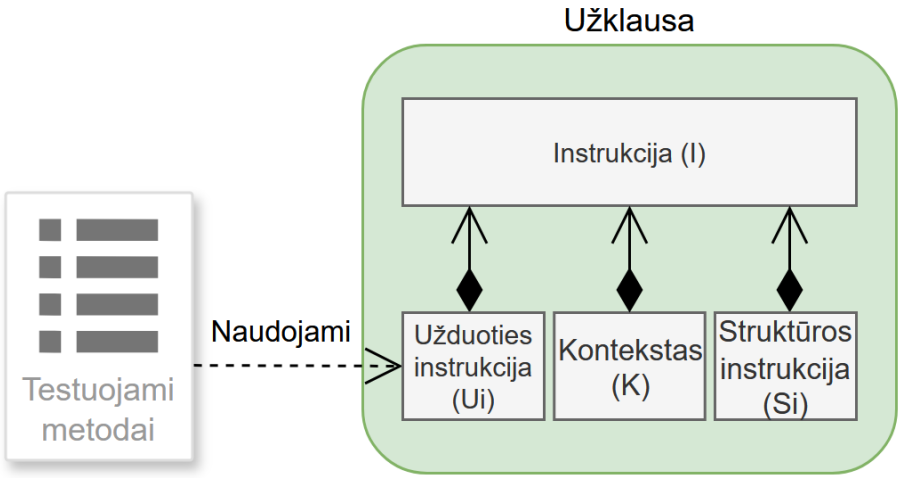
apibrėžtą struktūrą. Žemiau pateikiama eksperimento metu naudota PydanticAI objekto struktūra.

```
class UnitTest(BaseModel):
    generatedUnitTest: str = Field(..., description="A fully generated unit test that matches its description.")
class TestCase(BaseModel):
    unitTests: List[UnitTest] = Field(..., description="List of generated unit tests for the focal method.")
```

7 pav. PydanticAI objekto struktūra naudojama validuoti DKM grąžinamus duomenis

Paveiksle 7 pavaizduotas kodo segmentas buvo naudojamas DKM grąžinamam atsakymui validuoti. Grąžinamo atsakymo validavimas naudoja algoritmą, kuris patikrina, ar grąžinti duomenys iš didelio kalbos modelio atitinka apibrėžtą objekto struktūrą ir gautus duomenis grąžina kaip apibrėžtąjį objektą, šiuo atveju, grąžinamas TestCase objektas su kodo vieneto testų sąrašu.

Šeštajame procese sukuriama ir dideliam kalbos modeliui išsiunčiama užklausa. Žemiau pateikta šios užklausos struktūra:



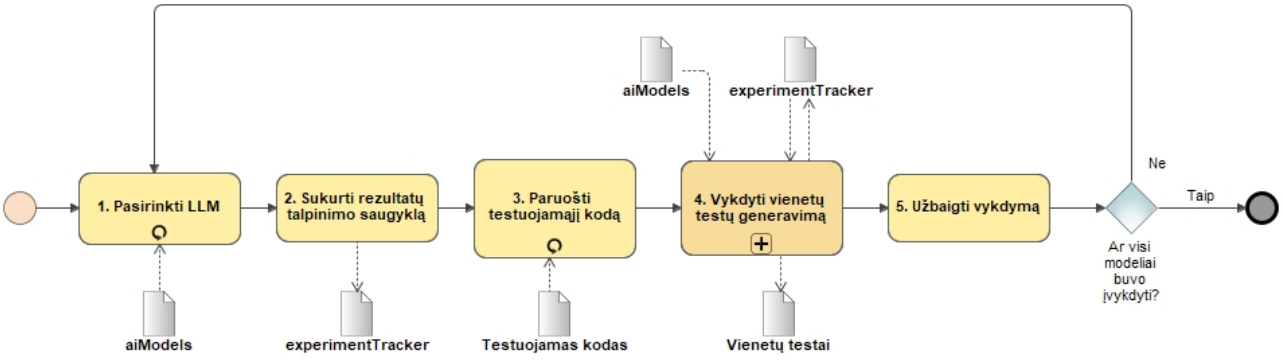
8 pav. Užklausos (angl. prompt) sudarymo diagrama

Užklausa yra sudaroma iš užduoties, struktūros instrukcijų, konteksto bei testuojamųjų metodų (žr. 8 paveikslą) Testuojamieji metodai yra pateikiami kaip visas kodo failas, taip dideliam kalbos modeliui yra suteikiama papildoma informacija apie nagrinėjamą kodo failą, t. y. kokios bibliotekos naudojamos bei klasė. Testuojamieji metodai yra naudojami užduoties instrukcijos. Užduoties instrukcija aprašo užduotį bei reikalavimus, kuriuos didelis kalbos modelis privalo atlikti ir kartu prisega testuojamojo kodo failą. Kontekste yra nurodomas teisingai sugeneruotų kodo vieneto testų rezultatas kaip pavyzdys. Struktūros instrukcija nurodo dideliam kalbos modeliui, kokiu formatu bei struktūra turi būti grąžintas rezultatas, kad atitiktų validavimo reikalavimus. 7 procese pasirenkami atskiri katalogai, kuriuose bus saugomi sėkmingai susikompiliuojantys ir nesusikompiliuojantys kodo vieneto testai kartu su jų testuojamuoju kodu, kuris buvo naudojamas sukurti kodo vieneto testams. 8

procese yra sudiegiamas kodo padengiamumo įrankis (angl. code coverage tool). Šis įrankis suteikia galimybę analizuoti sugeneruotų kodo vieneto testų kodo padengiamumo metrikas. Eksperimento metu buvo naudojamas dotnet-coverage įrankis. 9 procese sudiegiamas Stryker.NET mutacijų testavimo įrankis, kuris leidžia atlikti sugeneruotų kodo vieneto testų kokybės bei atsparumo klaidoms atvejų vertinimą.

2.3. Eksperimento vykdymas

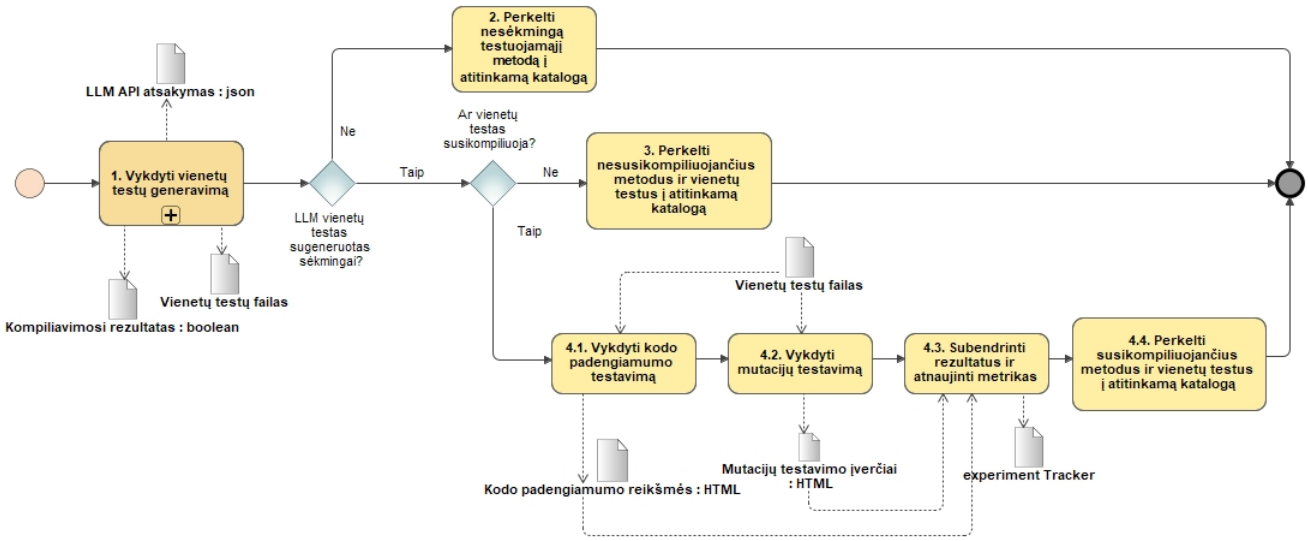
Eksperimento vykdymo proceso pagrindinis tikslas yra atlikti užklausos formavimo, kodo vieneto testų generavimo ir sugeneruotų kodo vieneto testų vertinimo užduotis. Žemiau pateikiama aukščiausio lygio eksperimento vykdymo diagrama.



9 pav. Eksperimento vykdymo diagrama

Procese 1 (žr. 9 paveikslą), pasirenkamas didelis kalbos modelis, kuris bus vykdomas kodo vieneto testams generuoti. Didelis kalbos modelis pasirenkamas iš *aiModels* objekto sąrašo, kuriame yra apsibrėžti visi eksperimento metu naudojami dideli kalbos modeliai. Eksperimento vykdymas tęsiasi iki tol, kol visi modeliai iš aiModels sąrašo yra panaudoti kodo vieneto testams generuoti. Procesas 2 atmintyje sukuria rezultatų talpyklą, kurioje bus talpinama informacija apie modelio susikompiliavusių testų skaičių ir apdorotus testuojamus metodus. Procesas 3, iteratyviai pasirenka kodo failą, kuris bus testuojamas ir naudojamas DKM užklausoje, siekiant sugeneruoti šio pasirinkto kodo failo kodo vieneto testus. Procesas 4, aprašo kodo vieneto testų generavimo vykdymą. Procesas 5, kodo vieneto testų generavimo vykdymo užbaigimas.

2.4. Kodo vieneto testų skirstymas pagal API atsakymą bei kompiliavimo rezultatą



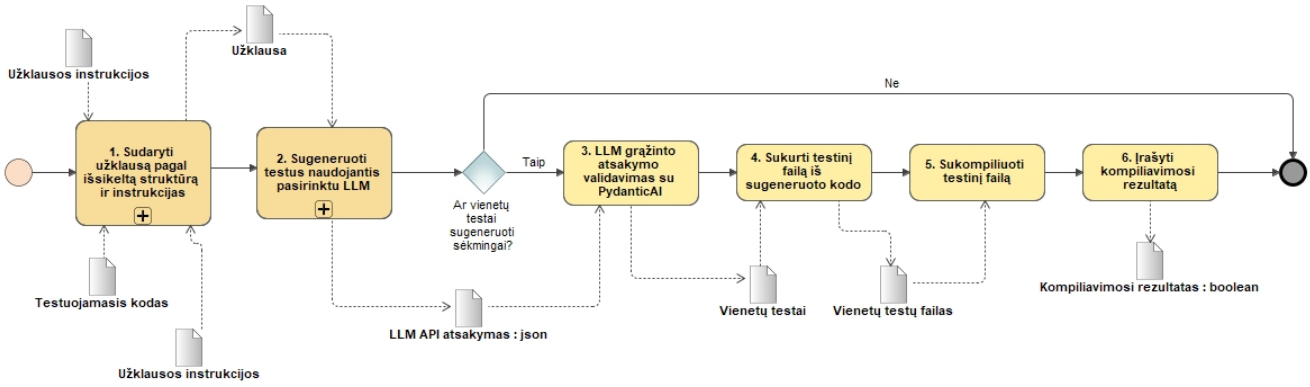
10 pav. Sugeneruotų kodo vieneto testų skirstymo pagal API atsakymą bei kompiliavimo rezultatą diagrama

Sugeneruotų kodo vieneto testų skirstymo pagal API atsakymą ir kompiliavimosi rezultatą diagrama (žr. 10 paveikslą) yra sudaryta iš 4 procesų:

Pirmasis procesas aprašo kodo vieneto testų gavimą. Procesas grąžina kompiliavimosi rezultatą, t. y. ar sugeneruotų kodo vieneto testų kompiliavimas buvo sėkminga. Taip pat grąžina sugeneruotų kodo vieneto testų failą ir DKM tiekėjo API atsakymą. Iš API atsakymo yra nustatoma, ar kodo vieneto testas buvo sukompiliuotas sėkmingai, ir ar neįvyko klaidų testų generavimo etape. Antrasis procesas aprašo atvejį, kai iš DKM API atsakymo gaunama indikacija, kad įvyko klaida tarp kodo vieneto testų generavimo rašmenos ir DKM tiekėjo API. Tokiu būdu kodo failas, kuris iššaukė šią klaidą, yra perkeliamas į atitinkamą katalogą, peržiūrai vėliau rankiniu būdu. Trečiasis procesas aprašo atvejį, kai sugeneruotų kodo vieneto testų kompiliavimas yra nesėkmingas. Tokiu atveju, sugeneruoti kodo vieneto testai yra perkeliami į atitinkamą katalogą su modelio pavadinimu ir kodo failu, kuriam buvo sugeneruoti nesikompiliuojantys kodo vieneto testai. 4 procesas aprašo atvejį, kai didelio kalbos modelio tiekėjas sugrąžina sėkmingai sugeneruotus kodo vieneto testus ir šių kodo vieneto testų kompiliavimas yra sėkmingas. Šiuo atveju kodo vieneto testams yra atliekamas kodo padengiamumo vertinimas, mutacijų testavimas bei atnaujinamos kompiliavimosi statistikos reikšmės rezultatų talpinimo saugykloje. Taip pat sėkmingai validuoti kodo vieneto testai yra perkeliami į atitinkamą katalogą su modelio pavadinimu bei kodo failu ir kodo vieneto testais.

2.5. Didelių kalbos modelių sugeneruotų kodo vieneto testų validavimas su PydanticAI

Procesas, *Vykdyti vieneto testų generavimą* (žr. 10 paveikslą), yra tęsiamas šiame skyriuje. Žemiau yra pateikiamas šio proceso plėtinys:

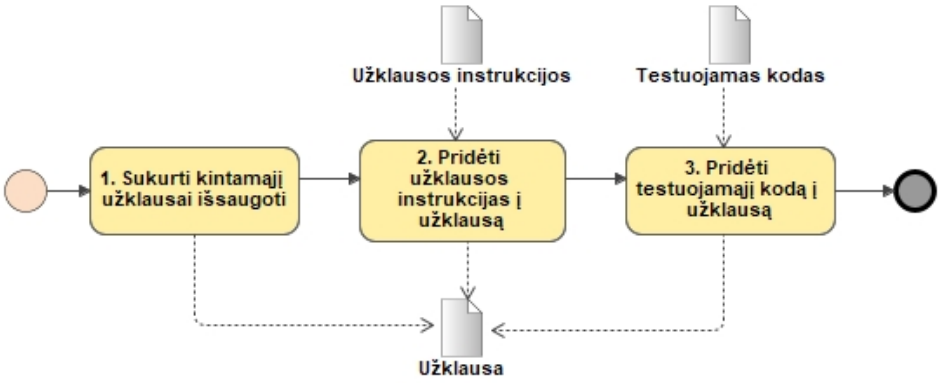


11 pav. Išsami kodo vieneto testų generavimo diagrama

Pirmasis procesas (žr. 11 paveikslą) sudaro užklausą (angl. prompt) pagal struktūrą, apibrėžtą 8 paveiksle. Antrasis procesas atlieka kodo vieneto testų generavimą kreipiantis į išorinius DKM API tiekėjus ir grąžina DKM API atsakymą. Procesas nr. 3 yra vykdomas tik, jei atsakymas iš didelių kalbos modelių tiekėjų buvo sėkmingas, t. y. atsakyme nebuvo aptikta klaidų. Šis procesas naudoja PydanticAI biblioteką, siekiant validuoti JSON formatu gautus kodo vieneto testus API atsakyme ir paversti į objektą, pavaizduotą 7 paveiksle. Tolesnis procesas nr. 4, sukuria ir prideda kodo vieneto testus į testinį failą, kuris bus naudojamas patikrinti, ar sugeneruoti kodo vieneto testai kompiliuojasi. Procesas nr. 5 aprašo testinio failo kompiliavimą naudojant dotnet 8.0 versiją. Procesas nr. 6 įrašo gautą kompiliavimosi rezultatą į kintamąjį *Kompiliavimosi rezultatas*, kuris bus naudojamas aukštesnio lygio procesuose.

2.6. Užklausos (angl. prompt) sudarymas

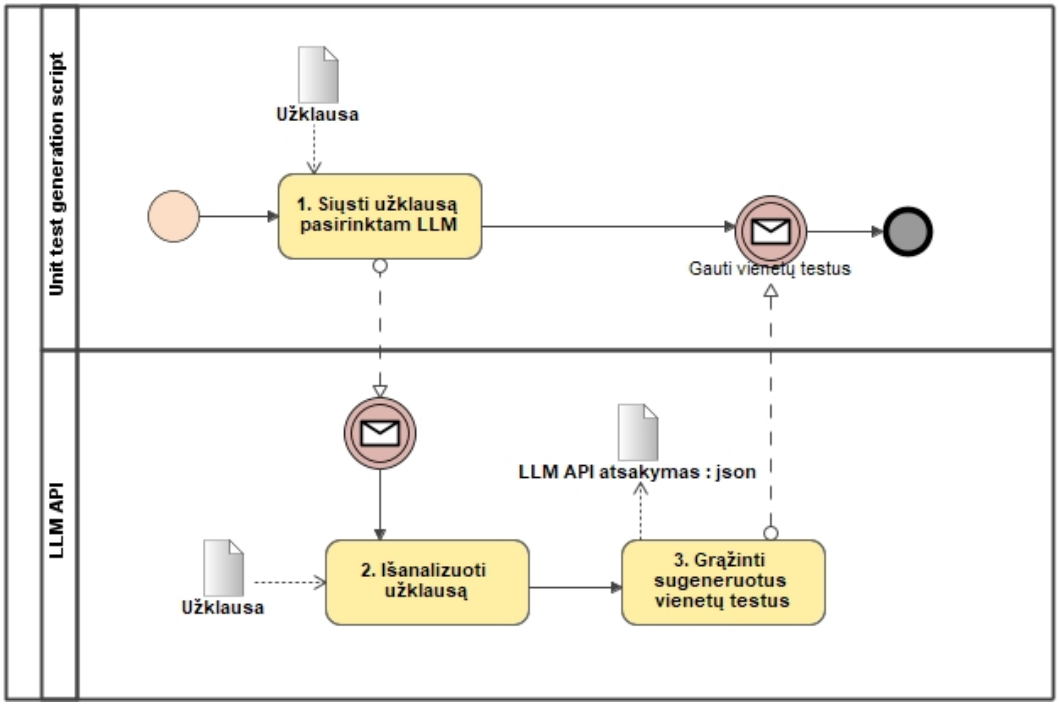
Užklausos sudėtis yra pateikta 8 paveiksle. Užklausos sudarymo procesas yra atvaizduotas 11 paveikslo pirmajame procese. Užklausos sudarymo procesas yra sudarytas iš trijų pagrindinių procesų:



12 pav. Užklausos (angl. prompt) sudarymo procesų diagrama

Pirmasis procesas (žr. 12 paveikslą) sukuria globalų užklausos kintamąjį, kuriame bus saugoma užklausa. Tolesnis procesas nr. 2, prideda instrukcijas į užklausą. Užklausos instrukcijos sudarytos iš užduoties, reikalavimų, konteksto bei struktūros taisyklių. Procesas nr. 3 įterpia testuojamąjį kodą į užklausą, tokiu būdu yra priskiriamas testuojamasis kodo failas.

2.7. Kodo vieneto testų generavimas naudojant DKM tiekėjų API



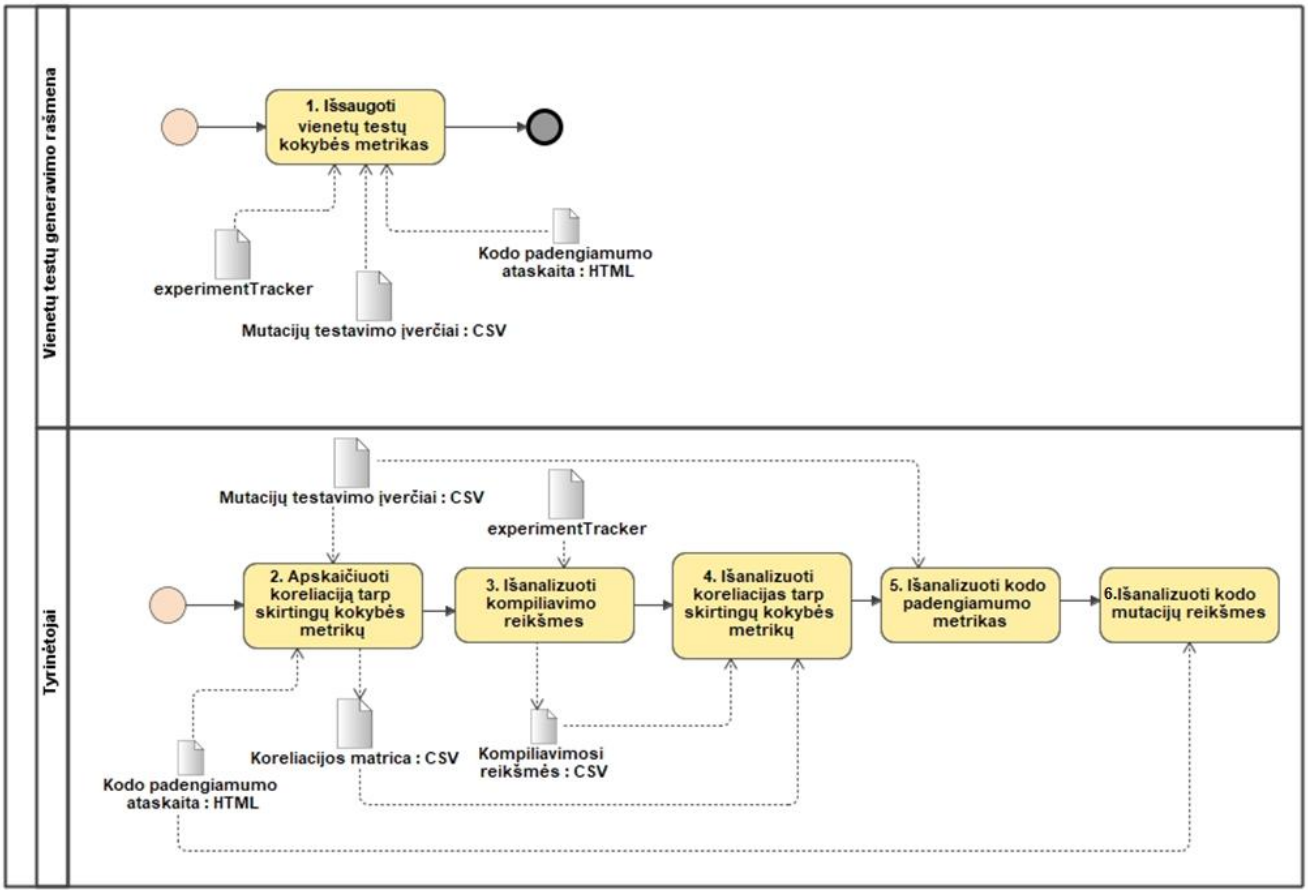
13 pav. Kodo vieneto testų generavimo naudojantis DKM API diagrama

Pirmasis procesas aprašo suformuotos užklausos siuntimą į pasirinktą didelio kalbos modelio tiekėją. Siunčiant užklausą tiekėjui, didelių kalbos modelių parametrai nebuvo keičiami, t. y. modelis buvo naudojamas toks, koks numatytas tiekėjo. Numatytieji parametrai nėra atskleidžiami tiekėjo, dėl

to, kokius parametrus didelis kalbos modelis naudoja, nėra žinoma. Antrasis procesas yra juodosios dėžės procesas, kurio vidinių procesų detaliai aprašyti negalima dėl informacijos trūkumo apie didelių kalbos modelių tiekėjų vidinius procesus. Dėl to manoma, kad vidiniai procesai gauna vartotojo išsiųstą užklausą per API sąsają ir ją apdoroja. Trečiasis procesas suformuotus kodo vieneto testus JSON formatu grąžina vartotojui kaip API atsakymą.

2.8. Rezultatų analizė

Rezultatų analizės proceso tikslas yra išsaugoti ir apdoroti duomenis, gautus iš kodo vieneto testų generavimo bei kokybės metrikų įrankių. Gautus duomenis paversti informacija, kurią tyrėjas galėtų panaudoti, kad įvertintų didelių kalbos modelių kodo vieneto testų generavimo galimybes bei kokybę.



14 pav. Rezultatų analizės procesas su pavaizduotais skirtingais aktoriais: kodo vieneto testų generavimo rašmena, tyrinėtojai

Pirmasis procesas aprašo gautų duomenų iš kodo vieneto testų generavimo proceso išsaugojimą. Procesas išsaugo trijų tipų duomenis: kompiliavimosi reikšmes (*experimentTracker* objektas), mutacijų testavimo įverčius/ataskaitas bei kodo padengiamumo ataskaitą. Duomenų išsaugojimas

suteikia tyrinėtojams galimybę gauti šiuos duomenis ir atlikti tolimesnę jų analizę. Antrasis procesas apskaičiuoja koreliacijas tarp skirtingų kokybės metrikų reikšmių kiekvienam pasirinktam dideliam kalbos modeliui. Trečiasis procesas išanalizuoja kompiliavimosi reikšmes, gautas iš kodo vieneto testų generavimo proceso ir paverčia jas į CSV formatą, taip yra išfiltruojami nesėkmingai susikompiliuojantys kodo vieneto testai. Ketvirtame procese tyrėjai analizuoja koreliaciją tarp skirtingų kokybės metrikų. Penktame ir šeštame procesuose analizuojamos kodo padengiamumo bei iš kodo mutacijų testavimo gautos reikšmės.

3. Tyrimo rezultatai

Šiame skyriuje pateikiami tyrimo rezultatai, gauti atlikus eksperimentą pagal metodologiją aprašytą 2 skyriuje, iš pasirinkto testuojamojo kodo buvo sugeneruoti kodo vieneto testai su pasirinktais dideliais kalbos modeliais. Dideli kalbos modeliai buvo ištirti naudojantis kodo padengiamumo ir kodo mutacijų testavimo įrankiu.

3.1. Kodo vieneto testų generavimas

Siekiant išsamiai įvertinti didelių kalbos modelių gebėjimą generuoti kodo vieneto testus, išanalizuojama, kokiai daliai pateiktų kodo failų kiekvienas modelis sėkmingai sugeneravo testus ir kiekvieno modelio sugeneruotų kodo vieneto testų skaičius. Didesnis testų skaičius gali nulemti platesnį kodo padengimą, tačiau nebūtinai įtakoja aukštesnę testų kokybę. Žemiau pateikiamas kiekvieno modelio sugeneruotų kodo vieneto testų lyginant su kodo failais santykis.

5 lentelė. Didelių kalbos modelių bendras sugeneruotų kodo vieneto testų skaičius ir sugeneruotų testų dalis kodo failams

Modelis	**Kodo failų skaičius, kuriems sugeneruoti kodo vieneto testai, %**	**Visų sugeneruotų kodo vieneto testų skaičius**
gemini-2.0-flash	100 % (310)	2229
gemini-2.5-flash	100 % (310)	3133
gpt-4.1	100 % (310)	2005
gpt-4o-mini	100 % (310)	2297
grok-4	100 % (310)	2789
devstral-small-2505	99.7 % (309)	2419
llama-3.3-70b-versatile	54.8 % (170)	1316

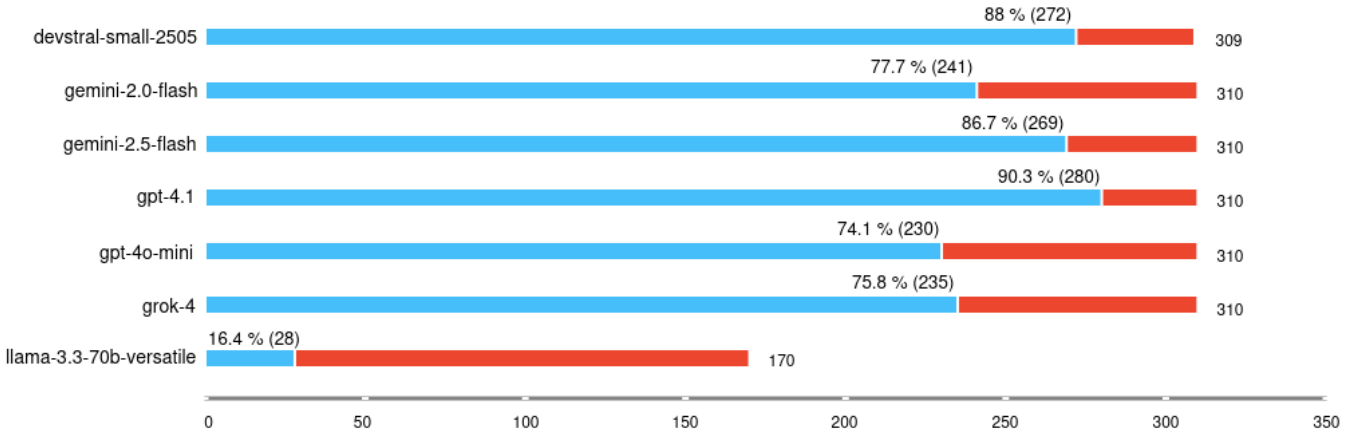
Tyrimo metu pastebėta, kad ne visiems pateiktiems kodo failams buvo sėkmingai sugeneruoti kodo vieneto testai. Mažiausiai kodo failų sėkmingai sugeneravo llama-3.3-70b-versatile modelis. Iš 310 kodo failų, llama-3.3-70b-versatile modelis kodo vieneto testus sugeneravo tik 170 kodo failams. Devstral-small-2505 modelis nesugeneravo kodo vieneto testų tik vienam kodo failui.

Atlikus sugeneruotų kodo vieneto testų statistikos analizę, pastebėta, kad daugiausiai kodo vieneto testų sugeneravo gemini-2.5-flash modelis. Mažiausiai kodo vieneto testų sugeneravo llama-3.3-70b-versatile modelis, tačiau iš 5 lentelės pastebima, kad llama-3.3-70b-versatile modelis nesugeneravo kodo vieneto testų visiems kodo failams, dėl to, galima teigti, kad kodo vieneto testų

skaičius šiam modeliui būtų didesnis, jei modelis būtų sugeneravęs visus arba panašų skaičių kodo vieneto testų kodo failams kaip kiti tyrinėjami modeliai. Didelių kalbos modelių sugeneruotų kodo vieneto testų skaičius neparodo kodo vieneto testų kokybės, todėl ne visada daugiau testų sugeneravęs modelis yra pranašesnis už mažesnį skaičių, bet kokybiškesnius testus sugeneravusį modelį.

3.2. Kodo vieneto testų kompiliavimosi rodiklis ir klaidos

Šiame skyriuje analizuojamas didelių kalbos modelių sugeneruotų kodo vieneto testų kompiliavimosi rodiklis bei dažniausiai pasitaikančios kompiliavimo klaidos. Kompiliavimosi rodiklis parodo, kokia dalis modelio sugeneruotų kodo vieneto testų gali būti sėkmingai sukompiliuota ir paruošta vykdymui. Tai yra esminis kokybės kriterijus, nes nesusikompiliuojantys testai negali būti įvykdyti ir nesuteikia jokios vertės testavimo procesui. Šio rodiklio analizė leidžia įvertinti, ar modelio sugeneruotas kodas yra sintaksiškai taisyklingas, ar teisingai naudojamos bibliotekos ir priklausomybės bei ar laikomasi XUnit karkaso apibrėžties. 15 paveiksle pateikiami kiekvieno modelio kompiliavimosi rodikliai kodo failo lygmeniu.



15 pav. Kodo vieneto testų kompiliavimosi rodikliai per kodo failą. Mėlyna spalva - kompiliavimas sėkmingas, raudona - nesėkmingas

Išnagrinėjus kompiliavimosi rodiklius (žr. 15 paveikslą), pastebima, kad gpt-4.1 modelis pirmauja pagal kompiliavimosi rodiklį. Pastebima, kad llama-3.3-70b-versatile modelis grąžino daugiausiai nesusikompiliavusių testų. Dėl modelio llama-3.3-70b-versatile sugeneruotų susikompiliuojančių kodo vieneto testų trūkumo, modelio kokybės metrikos neturėtų būti vertinamos lygiaverčiai su kitais modeliais, kurie sugeneravo didesnį kompiliuojančių testų kiekį.

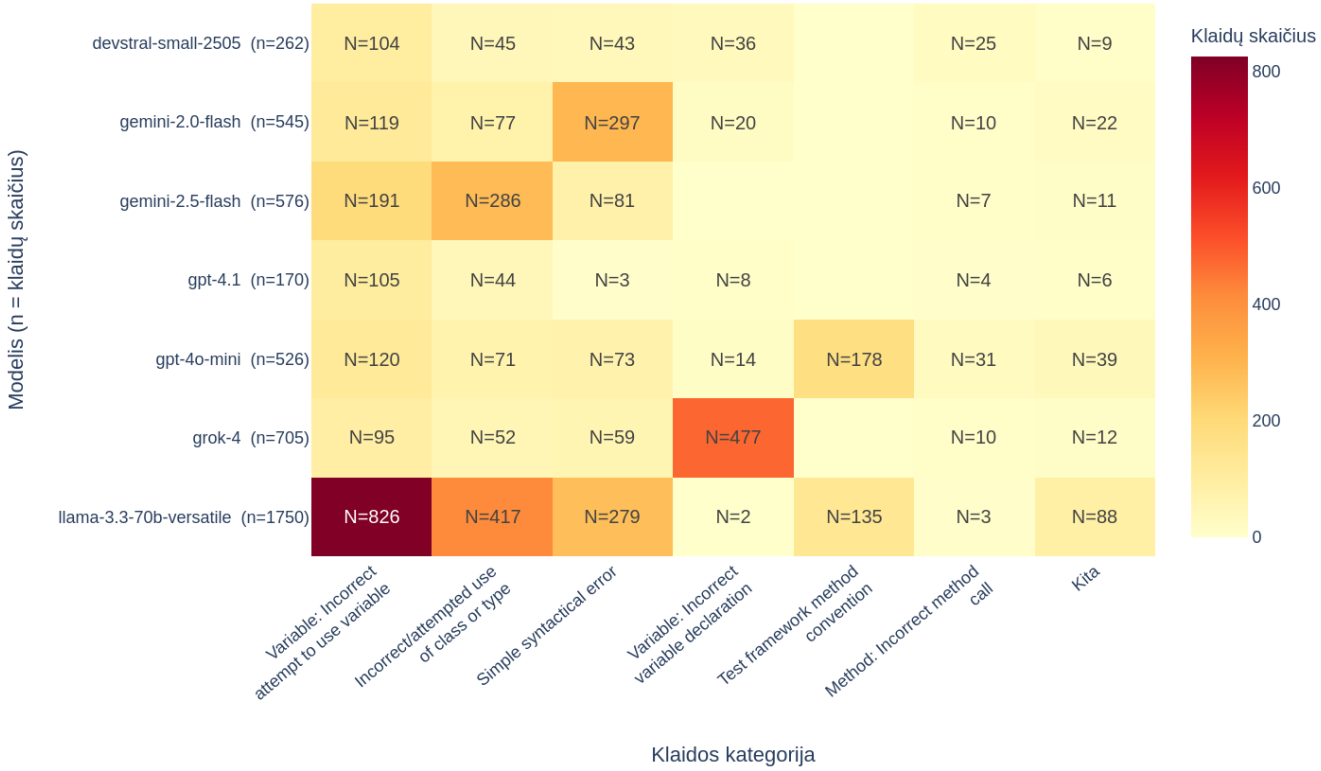
Toliau buvo nagrinėjamos kodo vieneto testų kompiliavimo klaidos. Iš viso gauti 88 skirtingi kompiliavimo klaidų kodai. Dėl per didelio skaičiaus klaidų tipų, buvo pasitelkti kiti tyrimai, kurie

atliko klaidų klasifikavimą pagal jų loginę prasmę, o ne pagal kompiliatoriaus pranešimą ar kodą, taip užtikrinant tikslesnį ir nuoseklesnį klaidų vertinimą [MK14, MK19]. Norint išsamiau išnagrinėti gautas kompiliavimo klaidas, buvo nutarta klaidas klasifikuoti naudojantis klasifikatoriais iš kitų tyrimų [MK14, MK19]. Nagrinėtuose tyrimuose klaidos buvo suskirstytos į 11 klasifikatorių, kurie pateikiami žemiau:

6 lentelė. Kompiliavimo klaidų klasifikatoriai [MK14, MK19]

Klasifikatorius	**Komponentas**	**Apibrėžimas**
Incorrect attempt to use variable	Kintamasis	Klaidos, kai kintamasis naudojamas netinkamai, yra neapibrėžtas, neteisingai parašytas, panaudotas iš kitos klasės.
Incorrect variable declaration	Kintamasis	Klaidos deklaruojant kintamuosius - neleistini modifikatoriai, neteisinga deklaracijos tvarka, pasikartojantys ar trūkstami pavadinimai.
Incorrect method call	Metodas	Klaidos iškviečiant metodą - neteisingas pavadinimas, neatitinkantys parametrai, praleisti skliaustai ar iškvietimas neapibrėžtam metodui.
Incorrect method declaration	Metodas	Klaidos deklaruojant metodą - praleistas kūnas, grąžinimo tipas, neteisingi parametrai arba pasikartojantys metodai.
Incorrect constructor call	Konstruktorius	Klaidos iškviečiant konstruktorių.
Incorrect constructor declaration	Konstruktorius	Klaidos deklaruojant konstruktorių.
Incorrect attempted use of class or type	-	Klaidos naudojant klases ar tipus - neapibrėžta klasė, neteisingas pavadinimas arba praleistas import.
Semantic error	-	Semantinės klaidos, kai kodas sintaksiškai teisingas, bet neatitinka kalbos taisyklių: tipų neatitikimai, praleistas return, neapdorotos išimtys.
Simple syntactical error	-	Paprastos sintaksės klaidos: praleisti kabliataškiai, nesutampantys skliaustai, neteisingai parašyti raktiniai žodžiai.
Statement outside method/block	-	Klaida, kai sakinys parašytas už metodo ar bloko ribų.
Uncategorized	-	Klaidos, kurios neatitinka jokios apibrėžtos kategorijos ir negali būti aiškiai priskirtos.

Atlikus kompiliavimo klaidų klasifikatorių analizę (žr. 6 lentelę), klaidų klasifikatoriai buvo panaudoti klasifikuoti klaidoms, gautoms iš didelių kalbos modelių sugeneruotų kodo vieneto testų. 88 klaidų kodai buvo sugrupuoti į 11 klasifikatorių. Žemiau pateikiami tyrime reikšmingiausi suklasifikuotų klaidų tipai:



16 pav. Kodo vieneto testų kompiliavimosi klaidos, suklasifikuotos pagal klaidų klasifikatorius (išrikiuotos pagal klaidų skaičių/reikšmingumą)

Išnagrinėjus klaidų pasiskirstymą pagal modelį (žr. 16 paveikslą), pastebima, kad kiekvienas modelis pasižymi skirtinga dominuojančia klaidų kategorija. Klaida „Variable: Incorrect attempt to use variable“ buvo dažniausiai pasitaikanti tarp visų modelių (34.4 % nuo bendro klaidų skaičiaus) ir labiausiai dominavo gpt-4.1 (61.8 %), llama-3.3-70b-versatile (47.2 %) bei devstral-small-2505 (39.7 %) modeliuose. Tai galimai parodo, kad dideli kalbos modeliai dažnai bando naudoti kintamuosius, kurie nėra apibrėžti arba yra nepasiekiami.

Klaida „Simple syntactical error“ dažniausiai pasireiškė gemini-2.0-flash modelyje (54.5 %), o kituose modeliuose ji sudarė mažesnę dalį. Klaida parodo, kad gemini-2.0-flash modelis yra dažniau linkęs praleisti kabliataškius ar netinkamai uždaryti skliaustus bei į kitas sintaksines klaidas. Klaida „Incorrect/attempted use of class or type“ pastebėta gemini-2.5-flash modelyje (49.7 %), tai parodo, kad modelis bando naudoti neegzistuojančias arba neteisingai nurodytas klases ir tipus.

Klaida „Variable: Incorrect variable declaration“ sudarė grok-4 modelio 67.7 % visų šio modelio klaidų, o kituose modeliuose ši kategorija buvo nežymi, arba visiškai neaptikta. Tai rodo, kad grok-4 modelis sistemingai neteisingai deklaruoja kintamuosius testuose.

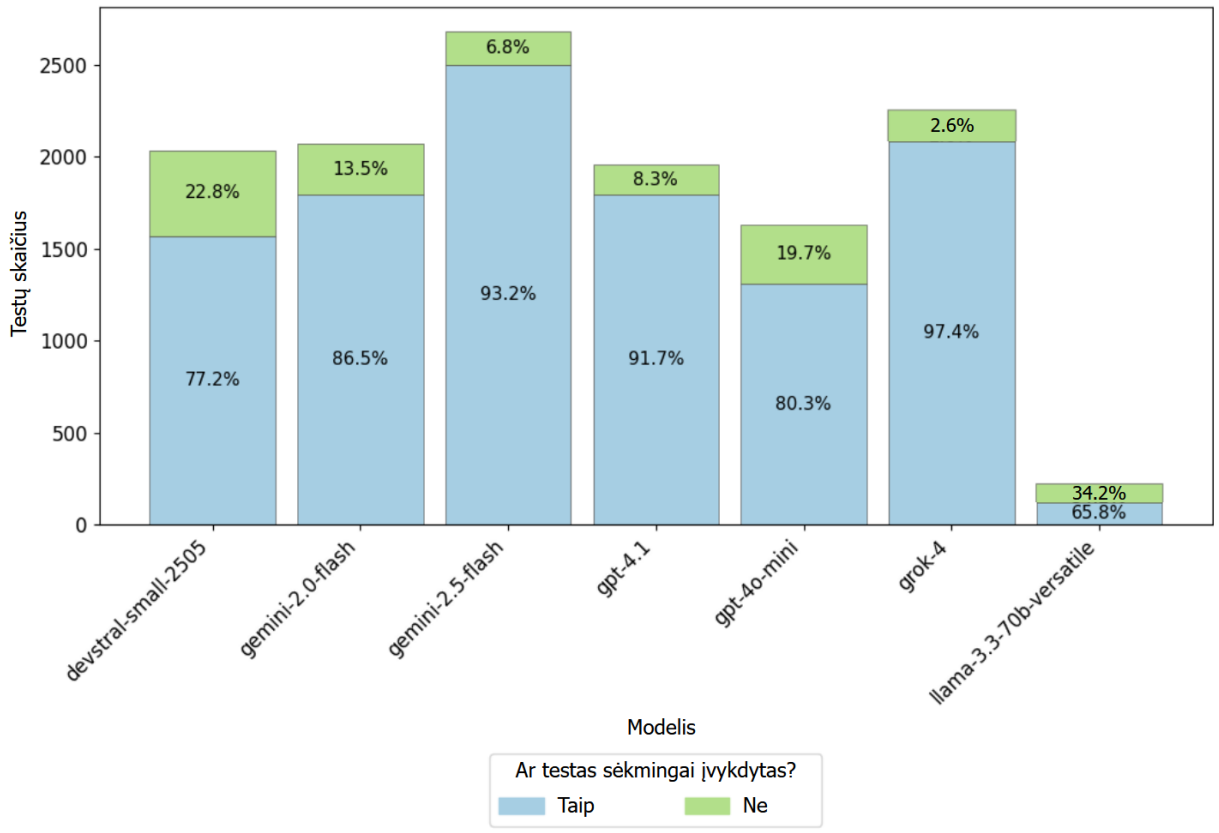
Klaida „Test framework method convention“ buvo aptikta tik gpt-4o-mini (33.8 %) ir llama-3.3-70b-versatile (7.7 %) modeliuose, o kituose modeliuose ji nepasireiškė. Tai gali rodyti, kad šie du

modeliai negeba išlaikyti XUnit karkaso metodų konvencijų.

Bendras klaidų skaičius reikšmingai skiriasi tarp modelių. Modelis gpt-4.1 sugeneravo mažiausiai klaidų (170), tuo tarpu llama-3.3-70b-versatile daugiausiai (1750). Tai parodo, kad gpt-4.1 geba generuoti testus, kurie yra sintaksiškai taisyklingi, tačiau neparodo, ar šio modelio sugeneruoti testai yra visapusiškai kokybiški.

3.3. Kodo vieneto testų vykdymo rodiklis ir klaidos

Tolesniame etape atliekama kiekybinė analizė, kurioje iš kiekvieno ištirto didelio kalbos modelio įvertinama, kokia dalis sugeneruotų testų yra semantiškai teisinga, t. y. įvykdyta be klaidų.



17 pav. Sėkmingai įvykdytų ir neįvykdytų kodo vieneto testų santykis

Iš 17 paveikslo pastebima, kad daugiausiai sėkmingai įvykdytų kodo vieneto testų sugeneravo grok-4 modelis, net 97.4 % šio modelio testų sėkmingai buvo įvykdyti. Pagal rezultatus po grok-4 modelio seka gemini-2.5-flash modelis, kuris sugeneravo mažesnę dalį, t. y. 93.2 % sėkmingai įvykdytų testų iš visų savo sugeneruotų testų. Didžiausią dalį kodo vieneto testų su vykdymo klaidomis sugeneravo devstral-small-2505 ir llama-3.3-70b-versatile modeliai, atitinkamai 22.8 % ir 34.2 %. Iš 17 paveikslo pastebima, kad lyderiais išlieka grok-4, gemini-2.5-flash ir gpt-4.1 modeliai. Galima teigti, kad grok-4 modelis geba grąžinti kodo vieneto testus, kurių kodo logika yra teisingesnė,

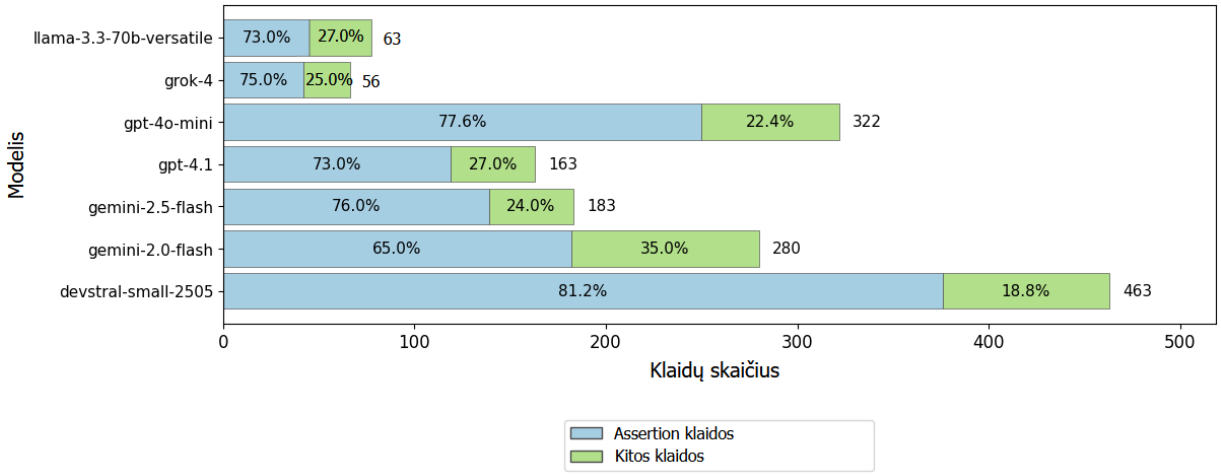
kai naudojama apibrėžta įvesties struktūra palyginus su kitais modeliais.

18 pav. Kodo vieneto testų vykdymo metu sėkmingai įvykdyti testai lyginant su bendru testų skaičiumi (n – kodo vieneto testų skaičius)

Toliau analizuojamas sėkmingai įvykdytų kodo vieneto testų santykis pagal metodo ciklomatinį sudėtingumą (žr. 18 paveikslą). Daugiausiai sėkmingai įvykdytų testų sugeneravo grok-4 modelis. Antrą vietą užėmė gemini-2.5-flash, kurio rezultatai svyravo tarp 90-96 %.

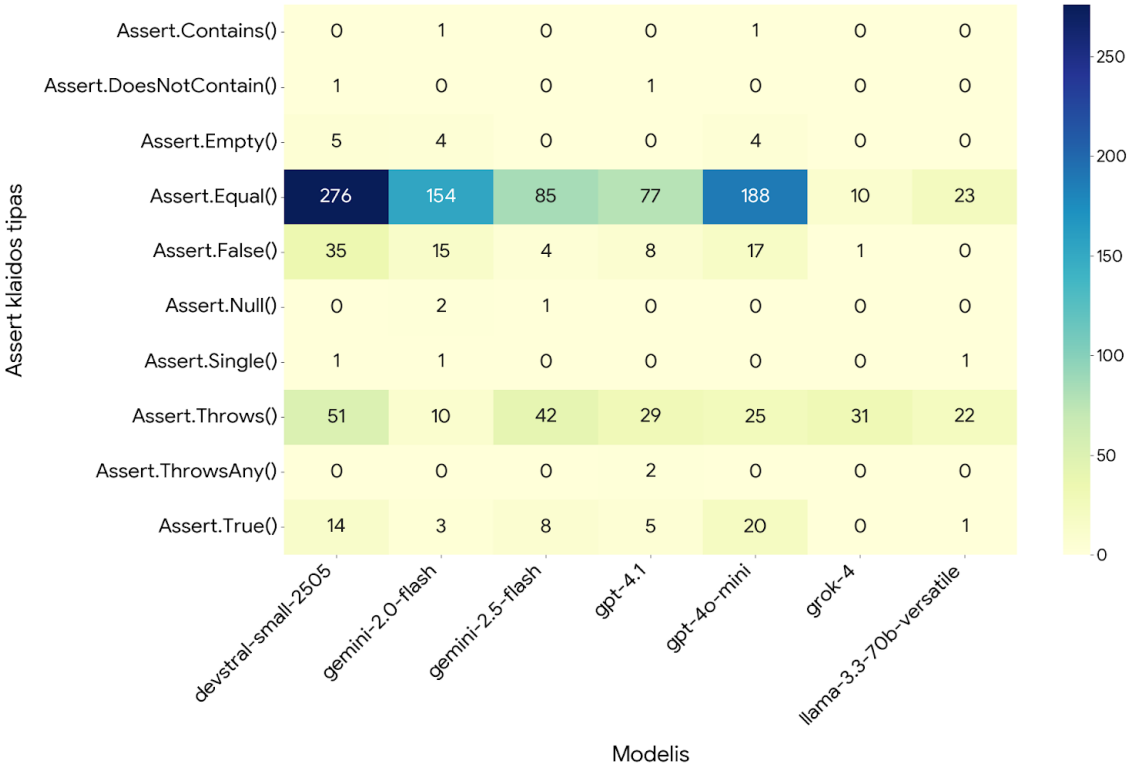
Blogiausius rezultatus parodė llama-3.3-70b-versatile modelis. Modelis devstral-small-2505 taip pat demonstravo žemesnius rezultatus, prie CC=1 sėkmingo vykdymo santykis siekė tik 55.6 %, o aukščiausia reikšmė neviršijo 81.4 %. Aukšto sudėtingumo atvejais, kai CC≥16, imčių dydžiai yra maži, todėl šie rezultatai negali būti visapusiškai įvertinami.

Toliau buvo atlikta išsami vykdymo klaidų analizė, siekiant nustatyti, kokio tipo klaidos dažniausiai pasitaiko modelių sugeneruotuose kodo vieneto testuose ir kokios yra pagrindinės jų atsiradimo priežastys. Ši analizė apima visų tyrime dalyvavusių didelių kalbos modelių vykdymo klaidų pasiskirstymą, leidžiantį palyginti modelių elgseną ir identifikuoti sistemines silpnybes.



19 pav. Kodo vieneto testų vykdymo metu gautos Assert klaidos dalis

Dažniausiai pasitaikiusi vykdymo klaida kiekvieno didelio kalbos modelio sugeneruotuose testuose buvo *Assert* klaida (žr. 19 paveikslą). Tai galimai parodo, kad dideli kalbos modeliai nesugeba suprasti, kokį rezultatą kodo vieneto testas grąžins vykdymo metu. Didžiausią dalį ir skaičių *Assert* klaidų turėjo devstral-small-2505 ir gpt-4o-mini, atitinkamai 81.2 % ir 77.6 %.



20 pav. Kodo vieneto testų vykdymo metu gautos Assert klaidų tipai ir jų skaičius pagal modelį

Išnagrinėjus *Assert* klaidų tipų pasiskirstymą pagal modelį (žr. 20 paveikslą), pastebima, kad

dažniausiai pasitaikiusi *Assert* klaida visuose modeliuose buvo *Assert.Equal()*. Daugiausia šio tipo klaidų užfiksuota devstral-small-2505 (276) ir gpt-4o-mini (188) modeliuose, o mažiausiai - grok-4 (10) ir llama-3.3-70b-versatile (23) modeliuose. Tai parodo, kad kai kurie dideli kalbos modeliai dažniausiai netiksliai numato laukiamą metodo grąžinamą reikšmę, o lygybės tikrinimas yra plačiausiai naudojamas tikrinimo metodas XUnit karkase. Antroji pagal dažnumą *Assert* klaida buvo *Assert.Throws()*, kurios skaičius pasiskirstė tolygiai tarp modelių (nuo 10 iki 51). Tai rodo, kad modeliai dažnai neteisingai numato, ar tam tikras kodo fragmentas išmes išimtį, arba klaidingai nurodo numatomos išimties tipą. Kiti *Assert* tipai: *Assert.Contains(), Assert.DoesNotContain(), Assert.Null(), Assert.Single()* ir *Assert.ThrowsAny()* pasitaikė retai. Tai galimai parodo, kad šių tipų tikrinimai rečiau naudojami modelių sugeneruotuose testuose, todėl ir jų sukeliamų klaidų yra mažiau.

Toliau nagrinėjamos kitos klaidos, gautos sugeneruotų kodo vieneto testų vykdymo metu.

7 lentelė. Kodo vieneto testų vykdymo kitų klaidų dalis lyginant su bendru klaidų skaičiumi pagal modelį

Modelis	IndexOutOfRange	InvalidOperation	ArgumentOutOfRange	NullReference	Kita
devstral-small-2505	41.4% (36)	6.9% (6)	14.9% (13)	6.9% (6)	29.9% (26)
gemini-2.0-flash	21.4% (21)	12.2% (12)	5.1% (5)	9.2% (9)	52.0% (51)
gemini-2.5-flash	11.4% (5)	25.0% (11)	4.5% (2)	15.9% (7)	43.2% (19)
gpt-4.1	22.7% (10)	13.6% (6)	20.5% (9)	2.3% (1)	40.9% (18)
gpt-4o-mini	27.8% (20)	6.9% (5)	9.7% (7)	13.9% (10)	41.7% (30)
grok-4	21.4% (3)	35.7% (5)	0.0% (0)	0.0% (0)	42.9% (6)
llama-3.3-70b-versatile	35.3% (6)	0.0% (0)	0.0% (0)	0.0% (0)	64.7% (11)

Toliau nagrinėjamos kitos klaidos, kurios užėmė mažesnę dalį bendro klaidų skaičiaus (žr. 7 lentelę). Klaida IndexOutOfRange buvo rečiausiai pastebima gemini-2.5-flash modelyje (11.4 %), o dažniausiai devstral-small-2505 modelyje (41.4 %). Tai gali rodyti, kad devstral-small-2505 modelis dažniau generuoja testus, kuriuose naudojami neteisingi indeksai, kurie nėra sąrašo arba masyvo ribose.

InvalidOperation klaida pastebima grok-4 (35.7 %) ir gemini-2.5-flash (25.0 %) modeliuose, llama-3.3-70b-versatile bei gpt-4o-mini modeliuose ji sudarė mažesnę dalį, atitinkamai 0.0 % ir 6.9 %. InvalidOperation klaida atsiranda, kai objektas yra panaudojamas netinkamai. Tai parodo, kad modeliai ne visada sugeba numatyti, kokiomis sąlygomis tam tikros operacijos gali būti sėkmingai

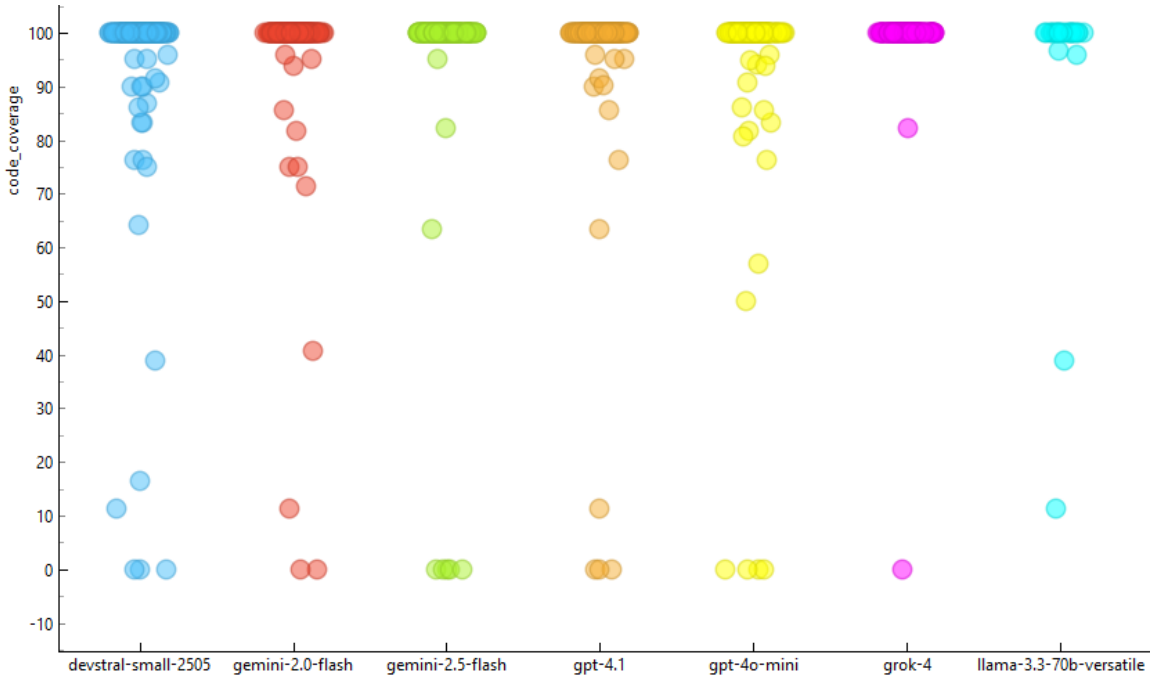
vykdomos.

ArgumentOutOfRange klaida buvo pastebima gpt-4.1 (20.5 %) ir devstral-small-2505 (14.9 %) modeliuose, grok-4 ir llama-3.3-70b-versatile modeliuose ši klaida nebuvo aptikta. NullReference klaida pastebėta gemini-2.5-flash (15.9 %) ir gpt-4o-mini (13.9 %) modeliuose, o grok-4 ir llama-3.3-70b-versatile modeliuose ji nebuvo aptikta.

Kategorija „Kita“ visuose modeliuose užėmė reikšmingą dalį. Tai galimai rodo, kad didelių kalbos modelių sugeneruoti kodo vieneto testai sukelia klaidas, kurių negalima priskirti dažniausiai pasitaikančioms kategorijoms.

3.4. Kodo padengiamumo rodikliai

Šiame skyriuje analizuojami didelių kalbos modelių sugeneruotų kodo vieneto testų kodo padengiamumo (angl. code coverage) rodikliai. Kodo padengiamumas yra vienas iš plačiausiai naudojamų kodo vieneto testų kokybės vertinimo kriterijų, parodantis, kokia testuojamo kodo dalis yra padengta kodo vieneto testais. Žemiau pateikiami kiekvieno modelio sugeneruotų kodo vieneto testų padengiamumo rodikliai.



21 pav. Kodo vieneto testų kodo padengiamumo reikšmės

Kodo vieneto testų padengiamumo reikšmės atskleidžia, kad didžiausias ir pastoviausias padengiamumo reikšmes turėjo grok-4 modelis (žiūrėti 1 priedą). Modeliai gemini-2.5-flash ir gpt-4.1 taip pat turėjo padengiamumo reikšmes su standartiniu padengiamumo nuokrypiu atitinkamai ±2.52 ir ±6.09. Modelių gpt-4.1 ir gpt-4o-mini padengiamumo reikšmės neparodo didelio skirtumo.

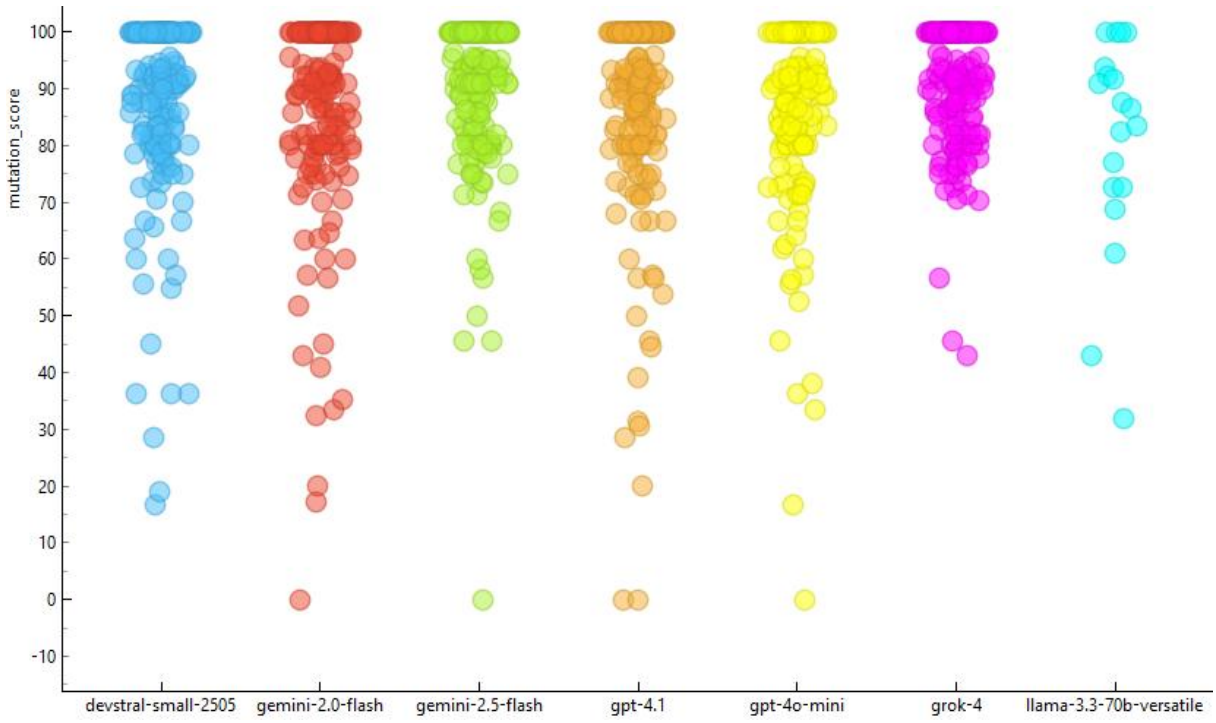
3.5. Kodo mutacijų įverčiai

Mutacijų įvertis, dar kitaip mutacijų atitikmens įvertis, yra aprašoma kaip sunaikintų ir visų neekvivalentinių mutantų santykis [Pet23]. Mutacijų testavime buvo naudotas Stryker.NET įrankis testuoti C# kodo vieneto testams ir testuojamajam kodui. Stryker.NET įrankis mutacijoms sukurti naudojo mutacijų operatorius pateiktus žemiau:

8 lentelė. Naudotų Stryker.NET mutacijų operatoriai [Strnd]

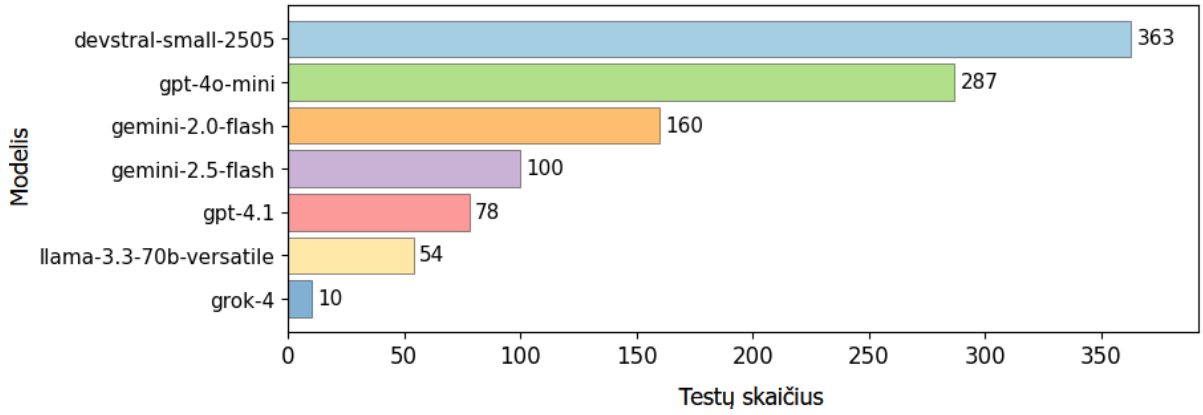
Operatorius	**Pradinio kodo eilutės pavyzdys**	**Mutantas į kurį pakeičiama kodo eilutė**
aritmetinis	a + b	a - b
priskyrimo	x += 1	x -= 1
bitwise	a & b	a | b
loginių reikšmių	if (person.IsAdult())	if (!person.IsAdult())
checked	checked(2 + 4)	2 + 4
masyvo arba sąrašo	[1, 2, 3]	[]
sąlyginis	x ? a : b	true ? a : b
lygybės	a == b	a != b
inicijavimo	new int[] { 1, 2 };	new int[] { };
linq	list.First()	list.FirstOrDefault()
loginių operatorių	a && b	a || b
Math klasės metodų	Math.Ceiling(x)	Math.Floor(x)
nullcoalescing	a ?? b	b ?? a
reguliariosios išraiškos	new Regex("^abc")	new Regex("abc")
sakinių/blokų pašalinimas	MyMethodCall();	(pašalinta)
tekstinis	„foo“	„“
tekstinių metodų	s.ToLower()	s.ToUpper()
unary	-variable	+variable
didinimo/mažinimo operatorių	variable++	variable--

Atliekant mutacijų testavimą buvo pasitelkiami operatoriai iš 8 lentelės, pradinio kodo eilutės buvo keičiamos pasitelkiant šiuos operatorius taip gaunant mutavusį kodą. Atlikus mutacijų testavimą bei jo analizę, buvo nagrinėjami rezultatai, gauti iš didelių kalbos modelių sugeneruotų kodo vieneto testų.



22 pav. Kodo vieneto testų kodo mutacijų testavimo reikšmės

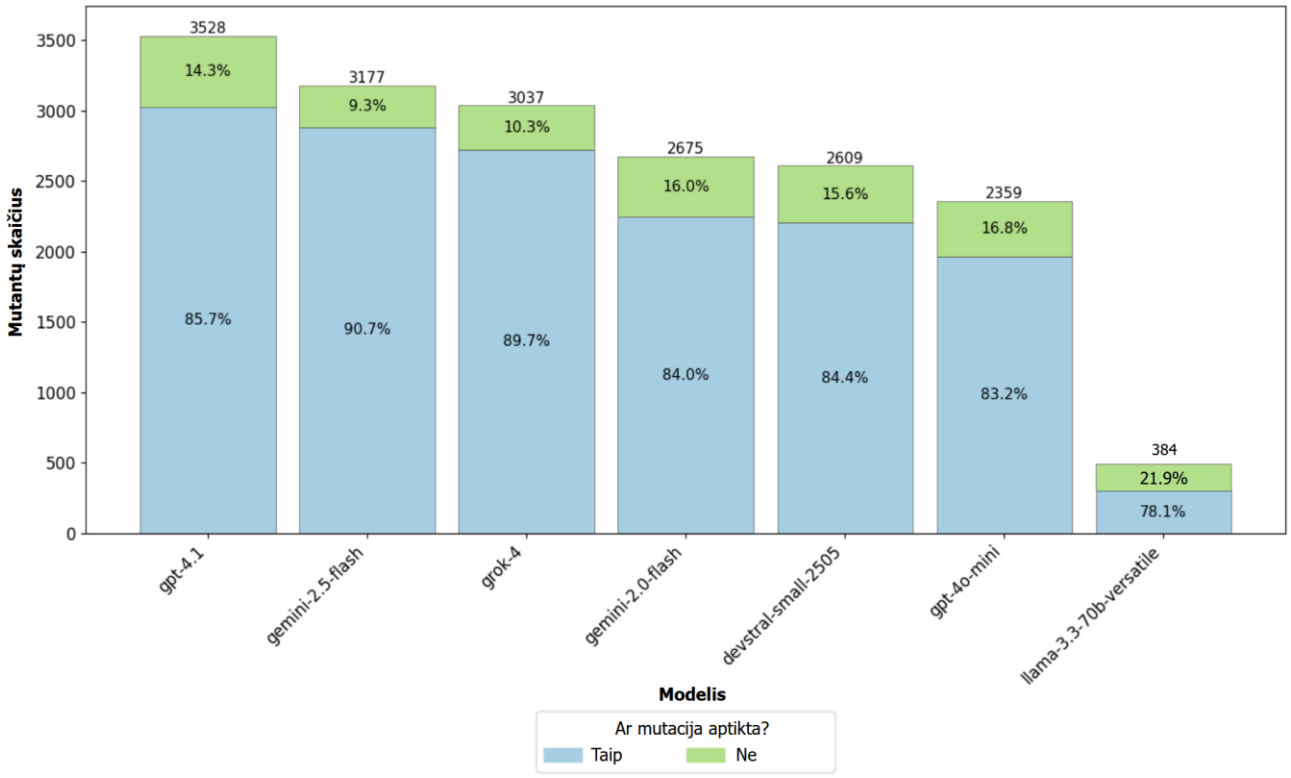
Iš 22 paveikslo ir 2 priedo pastebima, kad didžiausią įverčių vidurkį turi grok-4 modelis, po jo seka gemini-2.5-flash modelis. Abu šie modeliai demonstruoja aukštą testų kokybę, nes jų vidurkiai turi aukščiausias vertes. Modeliai gpt-4.1, gemini-2.0-flash ir devstral-small-2505 pasižymi panašiais rezultatais, tačiau pasižymi didesniu standartiniu nuokrypiu, kuris siekia ± 16.67, tai gali nulemti svyruojančią testų kokybę priklausomai nuo pateiktos užduoties sudėtingumo.



23 pav. Ignoruotų kodo vieneto testų skaičius

Mutacijų testavimo karkasas, Stryker.NET, nesėkmingai įvykusius kodo vieneto testus ignoruoja ir neįtraukia į galutinį mutacijų įvertį, tokiu būdu neiškreipiant įverčio reikšmės. Palyginus 17 ir 23 paveikslus galima pastebėti tendenciją tarp modelių sugeneruotų kodo vieneto testų vykdymo

klaidų skaičiaus ir Stryker.NET įrankio ignoruotų kodo vieneto testų. Šis ryšys parodo, kad Stryker.NET įrankis identifikavo ir ignoravo didelę dalį kodo vieneto testų, kurių vykdymas buvo nesėkmingas ir neįtraukė šių testų į bendrą statistiką.



24 pav. Kodo vieneto testų mutacijų išgyvenimo/sunaikinimo dalis lyginant su bendru mutacijų skaičiumi pagal modelį

Išanalizavus kiekvieno didelio kalbos modelio sugeneruotus kodo vieneto testus, kurie išgyveno arba buvo sunaikinti, pastebima, kad gemini-2.5-flash ir grok-4 modeliai sugeneravo didžiausią dalį testų, kurie aptiko kodo mutacijas. Tai parodo, kad šie modeliai kuria testus, kurie ne tik geba padengti kodo eilutes, bet ir identifikuoti pakeitimus bei klaidas kodo metoduose.

9 lentelė. Išgyvenusių mutacijų operatorių tipai ir dalis nuo bendro išgyvenusių mutacijų skaičiaus pagal modelį

Modelis\Mutacijos	Equality	Assignment	String	Arithmetic	Bitwise	Other	Total
devstral-small-2505	44.2%	16.0%	14.3%	7.9%	1.2%	16.5%	407
gemini-2.0-flash	32.9%	13.3%	16.3%	16.8%	3.3%	17.5%	428
gemini-2.5-flash	41.4%	16.9%	11.5%	10.8%	8.1%	11.2%	296
gpt-4.1	34.9%	14.3%	13.3%	14.9%	5.2%	17.5%	505
gpt-4o-mini	34.3%	13.9%	14.9%	8.3%	9.6%	19.1%	397
grok-4	35.4%	13.7%	12.4%	12.4%	4.8%	21.3%	314
llama-3.3-70b-versatile	32.1%	16.7%	19.0%	14.3%	0%	17.9%	84

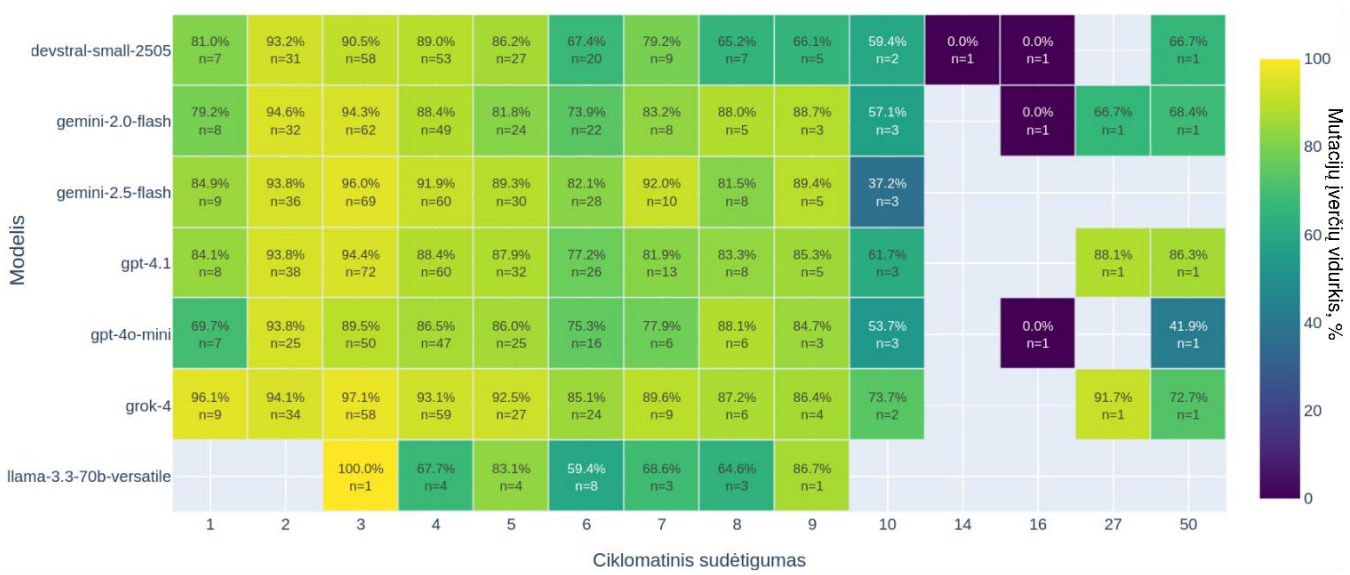
Toliau tyrimas analizuoja išgyvenusias mutacijas (žr. 9 lentelė), kurių dideli kalbos modeliai sugeneruotais kodo vieneto testais neidentifikavo. Galima pastebėti, kad didžiausia dalis išgyvenusių mutacijų buvo lygybės (angl. Equality) tipo - šis operatorius dominuoja visuose modeliuose ir svyruoja nuo 32.1 % (llama-3.3-70b-versatile) iki 44.2 % (devstral-small-2505). Tai rodo, kad modelių sugeneruoti testai dažniausiai neatsižvelgia į lygybės sąlygas, pavyzdžiui, palyginimo operatorių pakeitimus (== į !=, < į <= ir t. t.).

Antrą pagal dydį grupę daugumai modelių sudaro priskyrimo (angl. assignment) tipo mutacijos, kurių dalis svyruoja nuo 13.3 % iki 16.9 %. Tai reiškia, kad sugeneruoti kodo vieneto testai ne visada padengia visus kodo sakinius, todėl tam tikri pakeitimai išlieka neaptikti. Šios kategorijos mutacijos pasiskirsto pakankamai tolygiai tarp visų modelių, o tai rodo, kad priskyrimo mutacijos tipo padengimo trūkumas yra bendra didelių kalbos modelių problema.

Tekstinio (angl. String) tipo mutacijos taip pat sudarė reikšmingą dalį išgyvenusių mutacijų, t. y. nuo 11.5 % (gemini-2.5-flash) iki 19.0 % (llama-3.3-70b-versatile). Tai parodo, kad modelių sugeneruoti testai dažnai netikrina tekstinių reikšmių pakeitimų.

Aritmetinės (angl. Arithmetic) tipo mutacijos gemini-2.0-flash (16.8 %), gpt-4.1 (14.9 %) ir llama-3.3-70b-versatile (14.3%) modeliuose išgyvena dažniau. Tai rodo, kad kai kurie modeliai geriau aptinka aritmetinių operatorių pakeitimus (pvz., + į -, * į /), tačiau kitų modelių testai į šiuos pakeitimus atkreipia mažiau dėmesio. Palyginus su kitų mutacijų operatorių dalimis, aritmetinės operacijos neužima didelės dalies bendro išgyvenusių mutacijų skaičiaus.

Bendras išgyvenusių mutacijų skaičius taip pat reikšmingai skiriasi tarp modelių. Modelis gpt-4.1 turėjo daugiausiai išgyvenusių mutacijų (505), o llama-3.3-70b-versatile mažiausiai (84).



25 pav. Mutacijų įverčių vidurkis lyginant su ciklomatiniu sudėtingumu pagal modelį

Išnagrinėjus mutacijų įverčių vidurkį pagal ciklomatinį sudėtingumą (žr. 25 paveikslą), pastebima, kad geriausius rezultatus visose sudėtingumo kategorijose pasiekė grok-4 modelis. Šio modelio mutacijų įverčio vidurkis 1<=CC<=9 ribose viršijo 85.1 %, o aukščiausia reikšmė (97.1 %) užfiksuota prie CC=3. Modelis gemini-2.5-flash, kurio mutacijų įverčio vidurkis svyravo tarp 81 % ir 96 %, taip pat demonstravo panašius rezultatus ir mutacijų įverčio reikšmėmis neatsiliko nuo grok-4 modelio.

3.6. Koreliacijos tarp modelių ir jų kokybės metrikų įverčių

Tyrimo metu buvo apskaičiuojamos koreliacijos (naudojant Spirmeno ranginės koreliacijos koeficientą) tarp didelių kalbos modelių sugeneruotų testų kokybės metrikų. Tokiu būdu buvo norima įžvelgti, kaip dideli kalbos modeliai suvokia ryšį tarp kodo struktūros ir testavimo efektyvumo. Taip pat ar jų sugeneruoti testai pasižymi loginiu vientisumu, o ne mechanišku kodo eilučių padengimu.

Išnagrinėjus 3 priedą, pastebima, kad skirtingi modeliai demonstruoja nevienodą gebėjimą generuoti kokybiškus testus. Atlikus tolimesnę analizę, pastebėta, kad devstral-small-2505, gemini-2.5-flash, gemini-2.0-flash parodė stipriausią teigiamą koreliaciją tarp kodo padengiamumo ir mutacijų įverčio. Stipri koreliacija tarp kodo padengiamumo ir mutacijų reikšmės parodo, kad dideli kalbos modeliai generuoja kokybišką kodą, t. y. didėjant kodo padengiamumui didėja ir mutacijų įvertis. Taip pat pastebėta, kad prasčiausias kodo padengimo ir mutacijų reikšmes turėjo llama-3.3-70b-versatile modelis, tačiau šis modelis negali būti vertinamas lygiaverčiai su kitais modeliais dėl sugeneruotų ir susikompiliuojančių kodo vieneto testų trūkumo. Modelio llama-3.3-70b-versatile koreliacijos matricos rezultatas gali būti išskiriamas kaip anomalija dėl neigiamų koreliacijos

reikšmių.

3.7. Rezultatų palyginimas su kitais tyrimais

Gauti rezultatai palyginti su kitais tyrimais (žr. 6 priedą), nagrinėjančiais didelių kalbos modelių gebėjimą generuoti kodo vieneto testus, norint įvertinti šio tyrimo rezultatų reikšmingumą moksliniame kontekste bei nustatyti, kokie metodai labiausiai prisideda prie sugeneruotų testų kokybės gerinimo. Palyginimui pasirinkti tyrimai, kurie taip pat analizavo didelių kalbos modelių sugeneruotų kodo vieneto testų kompiliavimo, vykdymo, kodo padengimo bei mutacinio testavimo rodiklius.

Šiame tyrime, lyginant su kitų autorių atliktais tyrimais, pastebėta, kad gautos kodo eilučių padengimo reikšmės yra gerokai didesnės, t. y. kodo eilučių padengimo reikšmės padidėjo iki 26.7 %. Taip pat pastebėta, kad nagrinėtuose tyrimuose dažniausia vykdymo klaida buvo *Assert* tipo, tai sutampa ir su šiame tyrime gautais rezultatais. Kompiliavimo rodiklis tyrimuose, kuriuose nebuvo naudojamas rezultato validavimas arba griežta įvesties struktūra, pasiekė vidutiniškai 42-43% sėkmingo testų kompiliavimo, palyginimui, šiame darbe gautos kompiliavimo reikšmės (laikant llama-3.3-70b-versatile modelį kaip išimtį) pasiekė gerokai didesnį kompiliavimo rodiklį nuo 74.1 % (gpt-4o-mini) iki 90.3 % (gpt-4.1) taikant griežtą įvesties struktūrą ir rezultato validavimą.

Šie rezultatai rodo, kad griežtos įvesties užklausos struktūros bei rezultato validavimo taikymas galimai pagerina didelių kalbos modelių sugeneruotų kodo vieneto testų kokybę. Nereikėtų atmesti galimybės, kad naujesnių didelių kalbos modelių ar skirtingo duomenų rinkinio naudojimas šiame darbe taip pat galėjo turėti įtakos geresniems rezultatams, tačiau, panašią tendenciją savo tyrime aprašė ir Siddiq ir kt. [SST+24], kurie autoriaus sukurtais validacijos algoritmais kompiliavimo rodiklį padidino nuo 43.1 % iki 100 %. Tai parodo, kad papildomi validavimo mechanizmai, gali būti naudinga priemonė, siekiant užtikrinti aukštesnę modelių sugeneruoto kodo kokybę.

Wang ir kt. [WXB+26] tyrime, kuriame buvo naudojamas iteratyvus metodas, mutacijų įvertis svyravo nuo 69.9-77.9 % iki 89.1-89.5 %. Šiame darbe gauti panašaus dydžio mutacijų įverčiai, tačiau be papildomo iteratyvaus mutacijų grąžinimo, t. y. tik taikant struktūrizuotą įvestį ir rezultato validavimą. Dėl to, galima teigti, kad griežta įvesties struktūra teigiamai daro įtaką sugeneruotų kodo vieneto testų kokybei. Manoma, kad integravus griežtą įvesties struktūrą kartu su iteratyviais metodais, būtų galima pasiekti dar geresnių rezultatų.

Rezultatai

Iš septynių ištirtų didelių kalbos modelių aukščiausią kodo vieneto testų kompiliavimosi rodiklį pasiekė gpt-4.1 (90.3 %), devstral-small-2505 (88.0 %) ir gemini-2.5-flash (86.7 %) modeliai, o žemiausią llama-3.3-70b-versatile (16.4 %), kuris kodo vieneto testus sėkmingai sugeneravo tik 170 iš 310 kodo failų (54.8 %). Iš viso užfiksuoti 88 skirtingi kompiliavimo klaidų kodai, suklasifikuoti į 11 kategorijų pagal McCall ir Kölling klasifikatorius.

Vykdymo metu daugiausiai sėkmingai įvykdytų testų sugeneravo grok-4 (97.4 %), gemini-2.5-flash (93.2 %) ir gpt-4.1 (91.7 %) modeliai. Dažniausiai pasitaikiusi vykdymo klaida visuose modeliuose buvo Assert tipo (nuo 65.0 % iki 81.2 % bendro vykdymo klaidų skaičiaus). Tarp Assert klaidų dominavo Assert.Equal() klaidos tipas.

Aukščiausią ir pastoviausią kodo eilučių padengiamumo vidurkį pasiekė grok-4 (99.924 ± 1.16 %) ir gemini-2.5-flash (99.773 ± 2.52 %) modeliai. Visi modeliai, išskyrus llama-3.3-70b-versatile (93.724 ± 20.61 %), pasiekė didesnį nei 98 % vidutinį kodo padengiamumą.

Aukščiausią mutacijų įverčio vidurkį pasiekė grok-4 (93.48 ± 9.80 %) ir gemini-2.5-flash (93.23 ± 11.74 %) modeliai. Tarp išgyvenusių mutacijų visuose modeliuose dominavo lygybės (Equality) tipo mutacijos, sudarančios nuo 32.1 % (llama-3.3-70b-versatile) iki 44.2 % (devstral-small-2505) bendro išgyvenusių mutacijų skaičiaus.

Lyginant su kitų autorių tyrimais, kuriuose nebuvo taikyta griežta įvesties užklausos struktūra bei rezultato validavimas (vidutinis kompiliavimosi rodiklis 42-43 %), šiame tyrime kompiliavimosi rodiklis pasiekė nuo 74.1 % (gpt-4o-mini) iki 90.3 % (gpt-4.1), kodo eilučių padengimo reikšmės padidėjo iki 26.7 %. Šiame tyrime mutacijų įverčio vidurkis siekė 89-93 % ir prilygo Wang ir kt. [WXB+26] (89.1-89.5 %) tyrimui.

Rekomendacijos

Kodo vieneto testų generavimo užduotims rekomenduojama rinktis grok-4 arba gemini-2.5-flash modelius. Modelio llama-3.3-70b-versatile naudoti nerekomenduojama, nes jis sėkmingai sugeneravo testus tik 54.8 % kodo failų ir parodė žemiausius kokybės rodiklius.

Rekomenduojama kodo vieneto testų generavimo sistemose taikyti griežtą įvesties užklausos struktūrą kartu su rezultato struktūrine validacija, naudojant jau egzistuojančias bibliotekas. Tyrimas parodė, kad toks sprendimas gali reikšmingai padidinti kompiliavimosi rodiklį bei mutacijų įverčius.

Tyrimas parodė, kad kiekvienas DKM pasižymi skirtingomis dominuojančiomis klaidų kategorijomis. Atliekant modelio integraciją kodo vieneto testų generavimo užduočiai rekomenduojama atlikti modeliui būdingų klaidų analizę ir pritaikyti validavimo mechanizmus, taikytus konkrečioms šio modelio silpnybėms.

Vertinant DKM generuojamų kodo vieneto testų kokybę rekomenduojama kartu naudoti kompiliavimo, vykdymo, kodo padengimo ir mutacijų testavimo metrikas, nes vien tik kodo padengiamumas nesuteikia pakankamai informacijos apie sugeneruotų testų kokybę.

Remiantis tyrimo rezultatais, užklausą rekomenduojama sudaryti iš trijų atskirtų užklausos dalių: užduoties instrukcijos (kurioje įterptas testuojamasis metodas), struktūros instrukcijos, konteksto su teisingo rezultato pavyzdžiu.

Išvados

Atlikus septynių modelių palyginimą, galima teigti, kad rezultato struktūrinio validavimo bibliotekos integravimas su griežta įvesties užklausos struktūra leidžia pasiekti panašaus dydžio mutacijų testavimo rezultatus kaip ir sudėtingesni iteratyvūs grįžtamojo ryšio metodai. Wang ir kt. [WXB+26] su Llama-3.3 70B modeliu, pakartotinai siunčiant išgyvenusias mutacijas, pasiekė 89.1-89.5 % mutacijų įvertį. Šiame tyrime grok-4 ir gemini-2.5-flash modeliai nenaudojant iteratyvaus grįžtamojo ryšio mechanizmo pasiekė nuo 93.23 % iki 93.48 % mutacijų įvertį. Tai galimai parodo, kad kodo vieneto testų generavimo sistemose panašaus efektyvumo galima pasiekti integruojant jau egzistuojančias rezultato validavimo bibliotekas nekuriant papildomų algoritmų.

Išanalizavus 88 unikalias kompiliavimo klaidas septyniuose modeliuose, pastebėta, kad kiekvienas didelis kalbos modelis pasižymi skirtingomis dominuojančiomis kompiliavimo klaidų kategorijomis. Toks pasiskirstymas literatūroje neaprašytas, t. y. egzistuojantys tyrimai neatlieka tarpmodelinio klaidų palyginimo analizės. Tai gali rodyti, kad renkantis didelį kalbos modelį kodo vieneto testų generavimo užduotims, reikalinga papildoma modeliui būdingų klaidų tipų analizė, kuri leistų pritaikyti validavimo mechanizmus, kurie leistų išvengti dažniausių modelio daromų klaidų, taip pagerinant modelio generuojamų kodo vieneto testų kokybę.

Atlikus mutacijų testavimą ir išanalizavus išgyvenusias mutacijas pagal operatorių tipus, nustatyta, kad nepaisant aukšto kodo padengiamumo ir mutacijų įverčio, visuose septyniuose modeliuose dominuojantis išgyvenusių mutacijų tipas yra lygybės („Equality“) operatorių mutacijos. Lygybės operatorių mutacijų išgyvenamumo dominavimas visuose modeliuose parodo, jog tai yra bendra didelių kalbos modelių silpnybė. Tai parodo, kad dideliems kalbos modeliams integravus papildomus mechanizmus, kurie užtikrintų lygybės tipo mutacijų operatorių sunaikinimą, galėtų ženkliai pagerinti sugeneruotų kodo vieneto testų atsparumą klaidoms ir bendrą testų kokybę.

Šaltiniai

[Ani22] Aniche, M. (2022). Effective software testing: A developer's guide. Simon and Schuster.

[ATA23] Alagarsamy, S., Tantithamthavorn, C., & Aleti, A. (2023). A3Test: Assertion-Augmented Automated Test Case Generation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2302.10352

[BD23] Bayrı, V., & Demirel, E. (2023, December). AI-Powered Software Testing: The Impact of Large Language Models on Testing Methodologies. 2023 4th International Informatics and Software Engineering Conference (IISEC), 1-4. doi:10.1109/iisec59749.2023.10391027

[BSD+22] Bareiß, P., Souza, B., d'Amorim, M., & Pradel, M. (2022). Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. arXiv [Cs.SE]. doi:10.48550/ARXIV.2206.01335

[Cat25] Catir, E. (2025). C# Unit Test Generation Using Google Gemini Code Assist: An Empirical Study.

[Dau20] Daukšys, M. (2020). Modeliais paremtas testavimas: testavimo įrankių analizė (Doctoral dissertation). [žiūrėta 2026-05-25]. Prieiga per internetą: <https://epublications.vu.lt/object/elaba:69467905/>

[DF14] Daka, E., & Fraser, G. (2014, November). A Survey on Unit Testing Practices and Problems. 2014 IEEE 25th International Symposium on Software Reliability Engineering, 201-211. doi:10.1109/issre.2014.11

[DGJnd] Dagienė, V., Grigas, G., & Jevsikova, T. Enciklopedinis kompiuterijos žodynas (4th ed.). Vilniaus universiteto Matematikos ir informatikos institutas. [žiūrėta 2026-05-25]. Prieiga per internetą: <http://www.ims.mii.lt/KF/index.html>

[DMN+22] Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2022). GitHub Copilot AI pair programmer: Asset or Liability? arXiv [Cs.SE]. doi:10.48550/ARXIV.2206.15331

[DNM+23] Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., & Desmarais, M. C. (2023). Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. arXiv [Cs.SE]. doi:10.48550/ARXIV.2308.16557

[EJK+23] Esposito, M., Janes, A., Kilamo, T., & Lenarduzzi, V. (2023). Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception. arXiv [Cs.SE]. doi:10.48550/ARXIV.2303.07722

[Fel24] Feldhausen, R. (2024, June 27). xUnit assertions. In CIS 400 textbook. Kansas State University. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://textbooks.cs.ksu.edu/cis400/1-object-orientation/04-testing/05-xunit-assertions/index.html>

[FG22] Fontes, A., & Gay, G. (2022). The Integration of Machine Learning into Automated Test Generation: A Systematic Mapping Study. arXiv [Cs.SE]. doi:10.48550/ARXIV.2206.10210

[GG23] Golla, M. R., & Godboley, S. (2023, February). gMutant: A gCov based Mutation Testing Analyser. 16th Innovations in Software Engineering Conference, 1-5. doi:10.1145/3578527.3578546

[GPD+19] Grano, G., Palomba, F., Di Nucci, D., De Lucia, A., & Gall, H. C. (2019). Scented since the beginning: On the diffuseness of test smells in automatically generated test code. Journal of Systems and Software, 156, 312-327. doi:10.1016/j.jss.2019.07.016

[Gri22] Grigas, O. (2022). Dirbtinio intelekto metodų taikymas grafinio turinio generavime (Doctoral dissertation). [žiūrėta 2026-05-25]. Prieiga per internetą: <https://epubl.ktu.edu/object/elaba:132149266/>

[GV23] Guilherme, V., & Vincenzi, A. (2023, September). An initial investigation of ChatGPT unit test generation capability. 8th Brazilian Symposium on Systematic and Automated Software Testing, 15-24. doi:10.1145/3624032.3624035

[Has23] Hashtroudi, S. P. (2023). Automated Test Case Generation Using Transformers and Domain Adaptation. [žiūrėta 2026-05-25]. Prieiga per internetą: <http://hdl.handle.net/1880/116140>

[HCC+24] Huang, Y., Chen, Y., Chen, X., Chen, J., Peng, R., Tang, Z., … Zheng, Z. (2024). Generative Software Engineering. arXiv [Cs.SE]. doi:10.48550/ARXIV.2403.02583

[JH10] Jia, Y., & Harman, M. (2010). An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering, 37(5), 649-678.

[Job21] Job, M. A. (2021). Automating and Optimizing Software Testing using Artificial Intelligence Techniques. International Journal of Advanced Computer Science and Applications, 12(5). doi:10.14569/ijacs0a.2021.0120571

[KB13] Kitchenham, B., & Brereton, P. (2013). A systematic review of systematic review process research in software engineering. Information and Software Technology, 55(12), 2049-2075. doi:10.1016/j.infsof.2013.07.010

[Kho20] Khorikov, V. (2020). Unit Testing Principles, Practices, and Patterns. Manning Publications. ISBN: 9781617296277.

[KPB+09] Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering - A systematic literature review. Information and Software Technology, 51(1), 7-15. doi:10.1016/j.infsof.2008.09.009

[KSK+23] Kasneci, E., Sessler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., … Kasneci, G. (2023). ChatGPT for good? On opportunities and challenges of large language models for education. Learning and Individual Differences, 103, 102274. doi:10.1016/j.lindif.2023.102274

[LFN+22] Lahiri, S. K., Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Musuvathi, M., … Gao, J. (2022). Interactive Code Generation via Test-Driven User-Intent Formalization. arXiv [Cs.SE]. doi:10.48550/ARXIV.2208.05950

[LLT+23] Li, J., Li, G., Tao, C., Li, J., Zhang, H., Liu, F., & Jin, Z. (2023). Large Language Model-Aware In-Context Learning for Code Generation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2310.09748

[LWG+22] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv [Cs.LG]. doi:10.48550/ARXIV.2207.01780

[Micnda] Microsoft. (n.d.). IndexOutOfRangeException Class. Microsoft Learn. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://learn.microsoft.com/en-us/dotnet/api/system.indexoutofrangeexception>

[Micndb] Microsoft. (n.d.). InvalidOperationException Class. Microsoft Learn. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://learn.microsoft.com/en-us/dotnet/api/system.invalidoperationexception>

[Micndc] Microsoft. (n.d.). ArgumentOutOfRangeException Class. Microsoft Learn. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://learn.microsoft.com/en-us/dotnet/api/system.argumentoutofrangeexception>

[Micndd] Microsoft. (n.d.). NullReferenceException Class. Microsoft Learn. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://learn.microsoft.com/en-us/dotnet/api/system.nullreferenceexception>

[MK14] McCall, D., & Kölling, M. (2014). Meaningful categorisation of novice programmer errors.

[MK19] McCall, D., & Kölling, M. (2019). A new look at novice programmer errors. ACM Transactions on Computing Education (TOCE), 19(4), 1-30.

[MO05] Ma, Y. S., & Offutt, J. (2005). Description of class mutation mutation operators for Java. Electronics and Telecommunications Research Institute, Korea.

[MZ16] Masri, W., & Zaraket, F. A. (2016). Coverage-Based Software Testing. In Advances in Computers (pp. 79-142). doi:10.1016/bs.adcom.2016.04.003

[NCP+23] Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., … Abrahamsson, P. (2023). Generative Artificial Intelligence for Software Engineering -- A Research Agenda. arXiv [Cs.SE]. doi:10.48550/ARXIV.2310.18648

[Netnd] .NET Foundation. (n.d.). xUnit.net: Home. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://xunit.net/>

[Nev10] Neverdauskas, T. (2010). Statinė CIL kodo analizė, remiantis simboliniu vykdymu. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://epubl.ktu.edu/object/elaba:1875094/>

[Pet23] Petrėtis, A. (2023). Automatinis programinio kodo testavimas (Doctoral dissertation). [žiūrėta 2026-05-25]. Prieiga per internetą: <https://epublications.vu.lt/object/elaba:192827851/>

[Petnd] Petkevičius, L. (n.d.). Mašininio ir giliojo mokymosi sąvokų žodynėlis [GitHub repository]. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://github.com/linas-p/ML-AI-2-LT>

[PKZ+19] Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2019). Mutation testing advances: an analysis and survey. In Advances in Computers (Vol. 112, pp. 275-378). Elsevier.

[PT24] Pornprasit, C., & Tantithamthavorn, C. (2024). Fine-Tuning and Prompt Engineering for Large Language Models-based Code Review Automation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2402.00905

[RGL+20] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., … Ma, S. (2020). CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv [Cs.SE]. doi:10.48550/ARXIV.2009.10297

[Sab24] Sabbag Filho, N. (2024). Improving Test Coverage in Csharp Using Moq and xUnit. In Leaders.Tec.Br (Vol. 1, Number 20, pp. 1-5). Zenodo. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://doi.org/10.5281/zenodo.14164752>

[SDM+22] Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., & Segura, S. (2022). Mutation testing in the wild: findings from GitHub. Empirical Software Engineering, 27(6), 132.

[SNE+23] Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2302.06527

[SST+24] Siddiq, M. L., Santos, J. C. S., Tanvir, R. H., Ulfat, N., Rifat, F. A., & Lopes, V. C. (2024). Using Large Language Models to Generate JUnit Tests: An Empirical Study. The 28th International Conference on Evaluation and Assessment in Software Engineering (EASE), 2024, 313-322. doi:10.1145/3661167.3661216

[Strnd] Stryker Mutator. (n.d.). Mutations. Stryker Mutator. [žiūrėta 2026-05-25]. Prieiga per internetą: <https://stryker-mutator.io/docs/stryker-net/mutations/>

[TLZ+23] Tang, Y., Liu, Z., Zhou, Z., & Luo, X. (2023). ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2307.00588

[VRA+22] Vijayasree, D., Roopa, N. S., Arun, A., et al. (2022). A Review on the Process of Automated Software Testing. arXiv [Cs.SE]. doi:10.48550/ARXIV.2209.03069

[WHC+23] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2023). Software Testing with Large Language Models: Survey, Landscape, and Vision. arXiv [Cs.SE]. doi:10.48550/ARXIV.2307.07221

[WXB+26] Wang, G., Xu, Q., Briand, L., & Liu, K. (2026). Mutation-guided unit test generation with a large language model. IEEE Transactions on Software Engineering.

[YLD+24] Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., & Lou, Y. (2024). Evaluating and improving ChatGPT for unit test generation. Proceedings of the ACM on Software Engineering, 1(FSE), 1703-1726.

[YLL+23] Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., & Peng, X. (2023). No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv [Cs.SE]. doi:10.48550/ARXIV.2305.04207

[ZZL+23] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., … Wen, J.-R. (2023). A Survey of Large Language Models. arXiv [Cs.CL]. doi:10.48550/ARXIV.2303.18223

Sąvokų apibrėžimai

Kodo vieneto testavimas (angl. Unit testing) - procesas, kurio metu testuojami atskiri programos moduliai ir funkcijos, nepriklausomai nuo kitų sistemos modulių. Kodo vieneto testavimas užtikrina individualių funkcijų teisingumą ir stabilumą [Gri22].

Kodo vieneto testas (angl. Unit test) - mažiausia testuojama taikomosios programos dalis [Nev10].

Struktūrinė validacija - metodika, kuri taiko griežtą įvesties užklausos struktūrą bei rezultato validavimą.

Paieška pagrįstas testų generavimas (angl. Search Based Software Testing) - testų generavimo metodika, naudojanti genetinius algoritmus, siekiant sukurti testavimo atvejus [Dau20].

Atsitiktinis testų generavimas (angl. Random testing) - metodas, naudojantis atgalinį atsitiktinį testavimą testavimo atvejams generuoti. Algoritmo idėja yra generuoti atsitiktines metodų kvietimų ir įvedimo sekas, kurios įvairiais būdais testuoja programą [Dau20].

Dideli kalbos modeliai (angl. Large Language Models) - teksto generavimo modeliai, gebantys suprasti ir generuoti tekstą [SST+24].

Kodo padengiamumas (angl. Code coverage) - metrika, nurodanti santykį tarp kodo eilučių, kurias įvykdo testas/-ai ir visų kodo eilučių skaičiaus [Kho20].

Kodo/Testo teisingumo įvertis (angl. Code/Test Correctness) - nurodo, ar suprogramuotas funkcionalumas teisingai atlieka jam priskirtą funkciją [Kho20].

Kodo sudėtingumas (angl. Code complexity) - nustatomas pagal išsišakojimų skaičių programiniame kode. Kuo didesnis išsišakojimų skaičius, tuo kodas yra sudėtingesnis [Kho20].

Išankstinis apmokymas (angl. Pre-training) - modelio apmokymo tipas, kuris nurodo, kad modelis yra iš anksto apmokytas su dideliu duomenų kiekiu, norint gauti geresnius rezultatus.

Adaptavimas (angl. Fine-tuning) - iš anksto apmokyto modelio pritaikymas sričiai, naudojantis nagrinėjamos srities duomenimis [Petnd].

Medžio šakų aprėptis (angl. Branch coverage) - aprėpties/padengiamumo metrika, suteikia tikslesnius rezultatus nei kodo padengiamumo metrika, nes padeda įveikti kodo padengiamumo metrikos trūkumus. Vietoj kodo eilučių skaičiaus naudojimo, ši metrika atsižvelgia į tokias sąlygas kaip *if* ir *switch*. Metrika parodo, kiek kodo teiginių atšakų testas aprėpia [Kho20, MZ16].

Teiginių aprėptis (angl. Statement coverage) - nurodo, kiek programinio kodo segmento teiginių testas aprėpia [MZ16].

Pasikartojantys teiginiai (angl. Duplicated Asserts) - kodo kvapas, nurodantis, kad testinis kodas turi daugiau nei vieną pasikartojantį teiginį, tokiu būdu apsunkinamas kodo skaitomumas ir

suprantamumas.

pass@k - metrika, kuri nurodo procentinę dalį klaidų išspręstų naudojantis k sugeneruotomis programomis per klaidą, [LWG+22, LFN+22].

n@k - metrika, kuri vertina tik n kandidatus iš k sugeneruotų programų kiekvienai problemai [LWG+22].

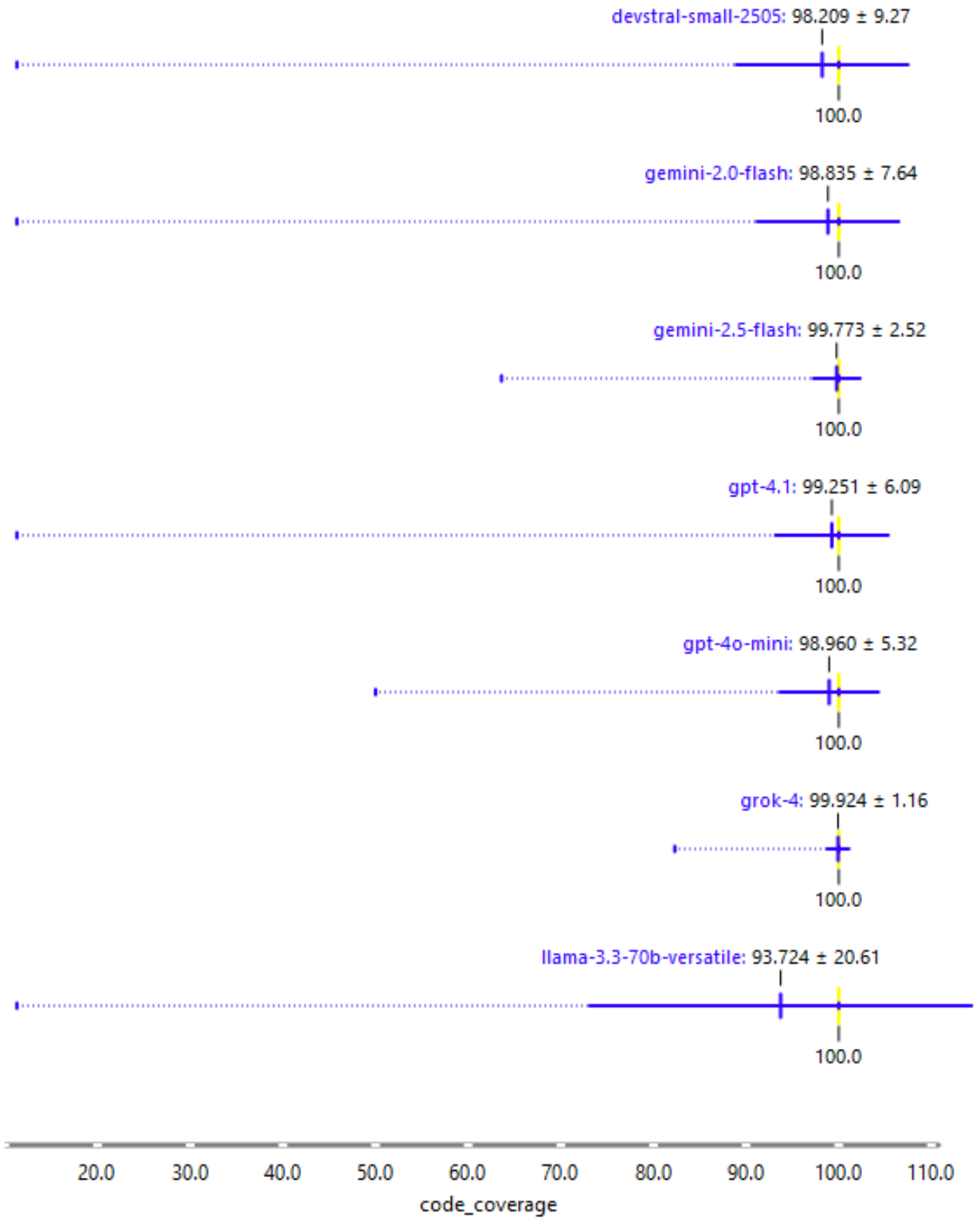
accept@m - metrika, naudojama sugeneruotų testų teisingumui vertinti [LFN+22].

Santrumpos

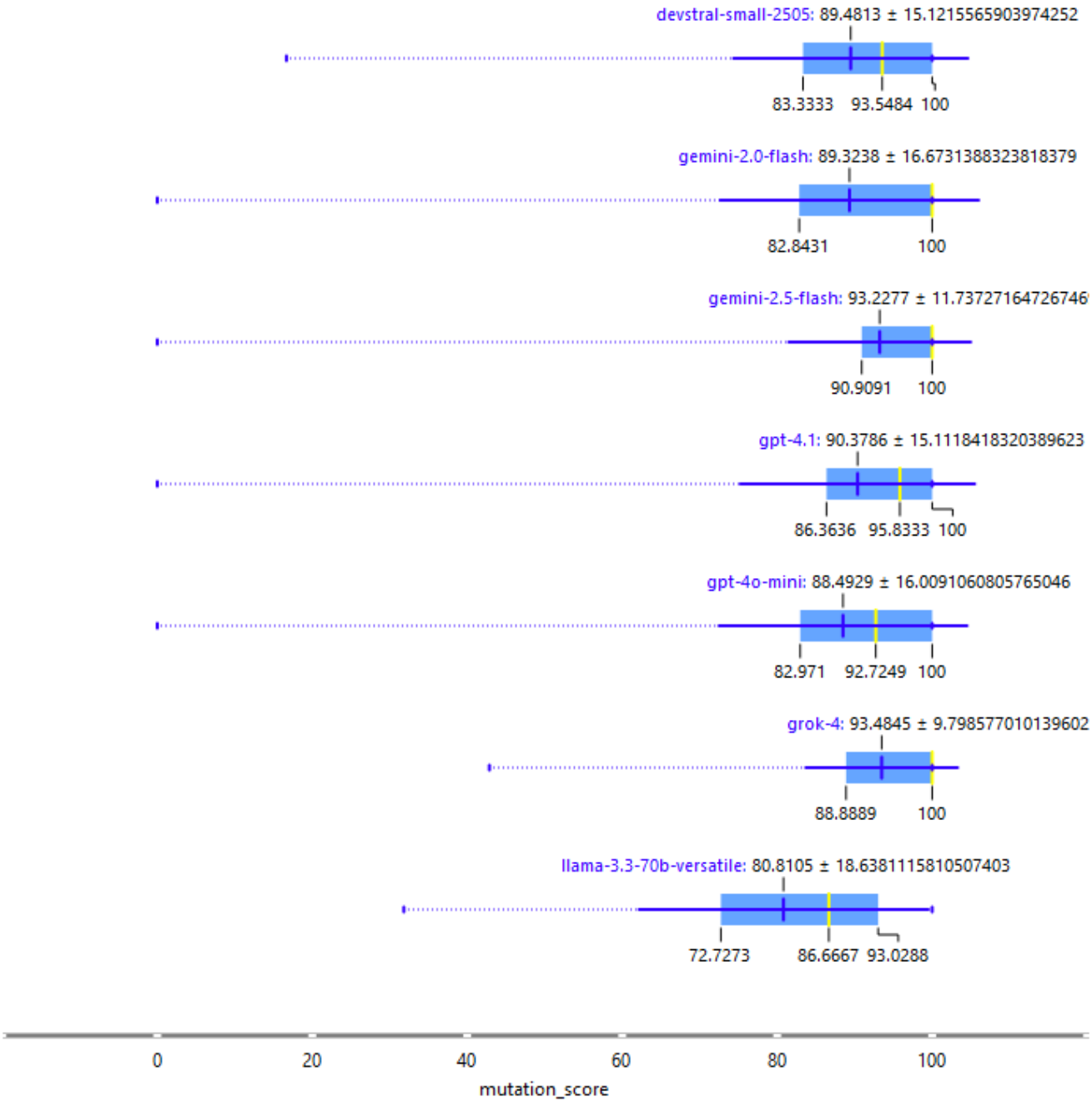
PS	programų sistema (angl. Software system).
DKM	dideli kalbos modeliai (angl. Large Language Models).
API	programų sistemos prieigos sąsaja (angl. Application Programming Interface).
CC	ciklomatinis sudėtingumas (angl. Cyclomatic complexity).
WoS	Web of Science, mokslinių tyrimų duomenų bazė.
IK	įtraukimo kriterijai [KPB+09, KB13].
AK	atmetimo kriterijai [KPB+09, KB13].
PU	paieškos užklausa [KPB+09, KB13].

Priedai

1 priedas. Kodo padengiamumo reikšmės



2 priedas. Kodo vieneto testų mutacijų reikšmės



3 priedas. Sugeneruotų kodo vieneto testų kodo kokybės metrikų koreliacijų reikšmės

Koreliacijos matrica (devstral)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.9966741	0.50395263	0.6555175	0.70152061
Branch coverage	0.99667408	1	0.5019568	0.6576249	0.7023363
Cyclomatic complexi	0.50395263	0.5019568	1	0.4617193	0.13643456
Class coupling	0.65551747	0.6576249	0.46171928	1	0.3933006
Mutation score	0.70152061	0.7023363	0.13643456	0.3933006	1

Koreliacijos matrica (gemini2.0f)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.996014	0.4450489	0.71593317	0.665891
Branch coverage	0.996014	1	0.4473352	0.71876014	0.66758
Cyclomatic complexity	0.445049	0.447335	1	0.43560089	0.104443
Class coupling	0.715933	0.71876	0.4356009	1	0.385614
Mutation score	0.665891	0.66758	0.1044434	0.38561384	1

Koreliacijos matrica (gemini2.5f)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.996941	0.608487	0.72011918	0.686279
Branch coverage	0.996941	1	0.6050821	0.72027342	0.687419
Cyclomatic complexity	0.608487	0.605082	1	0.45540471	0.227282
Class coupling	0.720119	0.720273	0.4554047	1	0.461403
Mutation score	0.686279	0.687419	0.2272824	0.46140267	1

Koreliacijos matrica (gpt4.1)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.965482	0.6066959	0.65797771	0.629673
Branch coverage	0.965482	1	0.5512728	0.6623668	0.644355
Cyclomatic complexity	0.606696	0.551273	1	0.55275371	0.116561
Class coupling	0.657978	0.662367	0.5527537	1	0.287418
Mutation score	0.629673	0.644355	0.1165613	0.28741805	1

Koreliacijos matrica (gpt4o-mini)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.932487	0.6046504	0.6503883	0.619067
Branch coverage	0.932487	1	0.4971736	0.65862644	0.648996
Cyclomatic complexity	0.60465	0.497174	1	0.4957334	0.093232
Class coupling	0.650388	0.658626	0.4957334	1	0.312637
Mutation score	0.619067	0.648996	0.0932315	0.31263662	1

Koreliacijos matrica (grok4)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.989975	0.6740217	0.68549592	0.634448
Branch coverage	0.989975	1	0.6588005	0.68637354	0.640215
Cyclomatic complexity	0.674022	0.658801	1	0.50143518	0.221504
Class coupling	0.685496	0.686374	0.5014352	1	0.378593
Mutation score	0.634448	0.640215	0.2215037	0.37859271	1

Koreliacijos matrica (llama-3.3-70b-versatile)

	Code coverage	Branch coverage	Cyclomatic complexity	Class coupling	Mutation score
Code coverage	1	0.836101	0.8512346	0.71127399	0.21573
Branch coverage	0.836101	1	0.6470096	0.79565987	0.090485
Cyclomatic complexity	0.851235	0.64701	1	0.74084837	-0.002175
Class coupling	0.711274	0.79566	0.7408484	1	-0.097014
Mutation score	0.21573	0.090485	-0.0021746	-0.097014	1

4 priedas. Tyrimų nagrinėjančių DKM sugeneruotų kodo vieneto testų kokybę literatūros analizės lentelė

Autoriai	**Metodika**	**Rezultatas**
Max Schäfer, Sarah Nadi, Aryaz Eghbali ir Frank Tip [SNE+23].	Autoriai siūlo sprendimą, kuris sugeneruoja kodo vieneto testus, pateikiant DKM modeliui testuojamojo metodo aprašą ir implementaciją, taip pat kartu įtraukiant metodo panaudojimo pavyzdžius iš dokumentacijos. Be to, jei sugeneruoto testo įvykdyti nepavyksta, sprendimas bando sugeneruoti naują testą, kuris išspęstų problemą, iš naujo užklausdamas modelį su nepavykusiu testu ir klaidos pranešimu.	Sprendimas pasiekė didesnį vidutinį teiginių aprėpties ir medžio šakų padengiamumą. Kodo vieneto testų generavimas naudojantis didelių kalbų modeliais yra pranašesnis už pažangiausius testų generavimo metodus, tokius kaip Nessie pagal esmines metrikas. Code-cushman-002 modelio sugeneruotų kodo vieneto testų padengiamumo metrikos yra prilyginamos gpt-3.5-turbo modeliui, tačiau gpt-3.5-turbo yra pranašesnis
Saranya Alagarsamy, Chakkrit Tantithamthavorn ir Aldeida Aleti [ATA23].	Autoriai supažindina su A3Test - giliojo mokymosi pagrindu paremtu testų atvejų generavimo metodu, kuris yra papildytas teiginių (assertion) žiniomis ir turi mechanizmą, skirtą tikrinti pavadinimų nuoseklumą ir testų antraštes.	Autoriai išnagrinėjo 5278 pagrindinius metodus iš Defects4j duomenų rinkinio ir nustatė, kad A3Test: (1) sugeneruoja 147% daugiau teisingų testų ir padengia 15% daugiau kodo tuo pačiu sugeneruodamas mažiau testų nei AthenaTest (2) Pranoksta esamus iš anksto apmokytus modelius testų atvejų generavimo užduotyje (3) Teiginių (assertion) apmokymas ir mechanizmas skirtas tikrinti pavadinimų nuoseklumą ir testų antraštes žymiai prisideda prie A3Test tikslumo. (4) A3Test yra 97,2% greitesnis ir tikslesnis nei AthenaTest.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, Steven C.H. Hoi [LWG+22].	Siūlo „CodeRL“ - naują karkasą programų sintezės užduotims atlikti, naudojant iš anksto apmokytus kalbos modelius ir gilųjį pastiprinamąjį mokymąsi (RL). Tikslas supažindinti su apmokymo strategija, kuri optimizuoja iš anksto apmokytus kalbos modelius programų sintezės užduotims, naudojant veikėjo-kritiko metodą.	CodeRL naudoja CodeT5 modelį ir pasiekia geresnius rezultatus nei kiti dideli modeliai, turintys didesnį parametrų skaičių. Pasiekti rezultatai: SOTA 2.57% 6.21% pass@5 ir 19.36% pass@1000. Ištestuotas modelio adaptavimas naujai užduočiai, nepateikiant pavyzdžių naudojant “MBPP benchmark“ našumo testą. Pasiektas SOTA rezultatas 63% pass@80 lyginant su papildomai išmokytu modeliu GPT-137B, kuris pasiekė 61.4%. Išleistas patobulintas CodeT5-large (770M) modelis, kuris pasiekia geresnius rezultatus nei daugelis kitų didelių modelių, turinčių didesnį parametrų skaičių.
Hashtroudi, Sepehr Pourabolfath [Has23].	Pagrindinis šio tyrimo tikslas yra pasiūlyti visiškai automatizuotą testavimo karkasą, kuris remdamasis programuotojų parašytais testais ir turimais kodo modeliais, sugebėtų sugeneruoti kompiliuojamus kodo vieneto testus.	Rezultatai parodo, kad didelių kalbos modelių pritaikymas tam tikrai sričiai, pagerina kodo eilučių aprėptį vidutiniškai 49.9% ir 54% medianos atžvilgiu, lyginant su modeliu be srities adaptavimo. Nustatyta, kad autorių pasiūlytas sprendimas gali būti taikomas su paieška pagrįstu automatiniu testų generavimu padidinant kodo padengiamumo metrikas vidutiniškai 25.3% ir 6.3 medianos atžvilgiu. Taip pat tokiu būdu pagerinamos ir mutacijų reikšmės vidutiniškai 8.6% ir 54% medianos atžvilgiu.
Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas Saikat Chakraborty, Madanlal Musuvathi, Jeevana Priya Inala, Piali Choudhury	Autoriai supažindina su interaktyviu testais pagrįstu kodo generavimu, kuris pasitelkia lengvu naudotojo atsiliepimu tam, kad:	Naudojant OpenAI Codex DKM: geriausias autoriaus pasiūlytas algoritmas pagerina pass@1 kodo generavimo tikslumą nuo 22,49% iki 37,71% MBPP našumo testo atveju ir nuo 24,79% iki 53,98%

Curtis von Veh, Chenglong Wang, Jianfeng Gao [LFN+22].	• Atsižvelgtų į naudotojo ketinimus naudojant sugeneruotus testus. • Pateiktų patobulintą kodo pasiūlymų rinkinį, išfiltruojant ir prioritizuojant kodo pasiūlymus.	HumanEval duomenų rinkinio atveju, naudojant nuo 1 iki 5 simuliuotų naudotojo užklausų.
Yutian Tang, Zhijie Liu, Zhichao Zhou ir Xiapu Luo [TLZ+23].	Tyrimas siekia ištirti didelius kalbos modelius, įvertinti šių modelių pranašumus, kad galėtų būti pagerintos pažangiausios paieška pagrįsto kodo vieneto testų generavimo technikos. Siekiama atskleisti esamų didelių kalbos modelių potencialius trūkumus generuojant testų rinkinius.	Tyrimas atskleidė: (1) 69.6% ChatGPT sugeneruotų komponentų testų gali būti sėkmingai sukompiliuojami ir įvykdomi. (2) Dažniausiai pasikartojanti sugeneruotų testų stiliaus klaida buvo neteisingai panaudotos įtraukos. Taip pat didžioji dalis testų turėjo mažą sudėtingumą. (3) Tyrimas nustatė, kad paieška grįstas kodo vieneto testų generavimo įrankis EvoSuite geba generuoti testus, kurie padengia 19% daugiau kodo nei ChatGPT.
Vitor H. Guilherme, Auri M. R. Vincenzi [GV23].	Tyrimo tikslas įvertinti Java programavimo kalba sugeneruotus kodo vieneto testus naudojantis OpenAI DKM.	Tyrimas atskleidė, kad OpenAI DKM sugeneruotų testų rinkinys geba sugeneruoti kodo vieneto testus, kurie vertinant pagal našumą, prilygsta tradiciniams kodo vieneto testų generavimo įrankiams naudojamiems Java programavimo kalbai.
Siddiq, M. L., Santos, J., Tanvir, R. H., Ulfat, N., Rifat, F. A., & Lopes, V. C. [SST+24]	Autoriai ištiria, trijų didelių kalbos modelių (Codex, GPT-3.5-Turbo ir StarCoder) kodo vieneto testų generavimo galimybes Java programavimo kalbai.	Nustatyta, kad Codex modelis pasiekė daugiau nei 80% kodo padengiamumą su HumanEval duomenų rinkiniu, tačiau nė vienas modelis nesugebėjo pasiekti 2% kodo padengiamumo su EvoSuite SF110 našumo testu. Sugeneruoti testai taip pat turėjo tikėtinų klaidų, tokių kaip pasikartojantys teiginiai ir tušti testai.

5 priedas. Sugeneruotų testų vertinimo kokybės metrikų tyrimų analizė

Autoriai	**Naudoti dideli kalbos modeliai**	**Duomenų šaltiniai**	**Kokybės metrikos**
Max Schäfer , Sarah Nadi , Member, IEEE, Aryaz Eghbali ir Frank Tip.	TESTPILOT (Javascript kodo vieneto testus gebantis generuoti įrankis, paremtas dideliais kalbos modeliais) • gpt3.5-turbo • code-cushman-002 • StarCoder	25 npm paketai sudarantys 1,684 API funkcijų.	• Sugeneruotų testų skaičius • Sėkmingai įvykdytų testų skaičius • Teiginių aprėptis • Medžio šakų aprėptis
Saranya Alagarsamy, Chakkrit Tantithamthavorn ir Aldeida Aleti.	A3Test Autoriaus siūlomas papildomai apmokytas DKM modelis paremtas PLBART architektūra su masked language mode.	Defects4j duomenų rinkinys	• Teisingų testų skaičius • Kodo padengiamumas
Hung Le , Yue Wang , Akhilesh Deepak Gotmare, Silvio Savarese, Steven C.H. Hoi.	CodeRL. Autorių sukurtas sprendimas, išplečiantis encoder-decoder CodeT5 modelio architektūrą su geresnėmis apmokymo strategijomis.	Apmokymui naudojama: GCPY Testavimui: APPS Benchmark	pass@k n@k
Hashtroudi, Sepehr Pourabolfath.	CodeT5 prieš EvoSuite	Methods2test Duomenų šaltinis naudotas papildomai apmokyti CodeT5 modeliui. Defects4j duomenų šaltinis naudotas adaptavimui domenui ir įvertinimui	• BLEU ir CodeBLEU reikšmės • Kodo eilučių padengiamumas • Mutacijų reikšmė
Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas Saikat Chakraborty, Madanlal Musuvathi, Jeevana Priya Inala,	Open AI’s Codex code-davinci-002 model	MBPP našumo testas HumanEval	• pass@k • accept@m

Piali Choudhury Curtis von Veh, Chenglong Wang , Jianfeng Gao.			
Yutian Tang, Zhijie Liu, Zhichao Zhou ir Xiapu Luo.	ChatGPT prieš EvoSuite	Defects4J	• Teisingumas • Skaitomumas (Checkstyle-Google, Checkstyle-SUN) • Kodo padengiamumas • Klaidų skaičius testuose.
Vitor H. Guilherme, Auri M. R. Vincenzi.	OpenAI API gpt-3.5-turbo prieš EvoSuite	33 Java programų rinkinys	• Kodo padengiamumas • Mutacijų reikšmė • Sėkmingai sukompiliuotų ir įvykdytų testų skaičius
Siddiq, M. L., Santos, J., Tanvir, R. H., Ulfat, N., Rifat, F. A., & Lopes, V. C.	Codex GPT-3.5-Turbo StarCoder	HumanEval Evosuite SF110	• Sėkmingai sukompiliuotų testų skaičius • Testų teisingumo įvertis • Kodo padengiamumas • Kodo tikėtinos klaidos

6 priedas. Tyrimų nagrinėjančių DKM gebėjimą generuoti kodo vieneto testus kokybės metrikų rezultatai taikant ir netaikant struktūrinės validacijos

Autoriai	**Įvesties užklausos (angl. prompt) tipas**	**Dideli kalbos modeliai**	**Griežtos įvesties struktūros / rezultato validavimas ar kiti taikyti metodai**	**Rezultatas (be griežtos įvesties struktūros / rezultato validavimo)**	**Rezultatas (taikant griežtos įvesties struktūros / rezultato validavimą, ar kitus metodus)**
Siddiq, M. L., Santos, J. C. S., Tanvir, R. H., Ulfat, N., Rifat, F. A., & Lopes, V. C. [SST+24]	Zero-shot	Codex (2K, 4K), StarCoder, GPT-3.5-Turbo	Taikytas. Rezultatas validuotas autorių sukurtais algoritmais.	Kompiliavimo rodiklis: 43.1 % (GPT-3.5-Turbo) iki 70 % (StarCoder).	Kompiliavimo rodiklis: 76.9 % (StarCoder) iki 100 % (Codex). Kodo eilučių padengimo rodiklis: nuo 67 % iki 87.7 %
Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., & Lou, Y. [YLD+24]	Zero-shot	GPT-3.5-turbo	Netaikytas	Kompiliavimo rodiklis: 42.1 % Sėkmingai įvykdytų testų dalis: 24.8% Dažniausiai pasitaikiusi vykdymo klaida: *Assert* klaida	-
Wang, G., Xu, Q., Briand, L., & Liu, K. [WXB+26]	Zero-shot, Few-shot	Llama-3.3 70B	Taikytas. Iš mutacijų testavimo gautos išgyvenusios mutacijos pakartotinai siunčiamos dideliam kalbos modeliui	Kodo eilučių padengimo rodiklis: nuo 96.2 % iki 96.3 % Mutacijų įvertis: 69.9 % iki 77.9 %	Kodo eilučių padengimo rodiklis: nuo 98.3 % iki 98.4 % Mutacijų įvertis: 89.1 % iki 89.5 %