Kauno technologijos universitetas

Informatikos fakultetas

Statinės kodo analizės tobulinimas taikant konsensusu grįstus didžiuosius kalbos modelius

Baigiamasis magistro projektas

Augustinas Labutis

Projekto autorius

Doc. Jonas Čeponis

Vadovas

Kaunas, 2026

Kauno technologijos universitetas

Informatikos fakultetas

Statinės kodo analizės tobulinimas taikant konsensusu grįstus didžiuosius kalbos modelius

Baigiamasis magistro projektas

Informacijos ir informacinių technologijų sauga (6211BX008)

Augustinas Labutis

Projekto autorius

Doc. Jonas Čeponis

Vadovas

dr. Jurgita Zencevičienė

Recenzentė

Kaunas, 2026

Kauno technologijos universitetas

Informatikos fakultetas

Augustinas Labutis

Statinės kodo analizės tobulinimas taikant konsensusu grįstus didžiuosius kalbos modelius

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdama(s) kitų asmenų autoriaus ar kitų teisių, laikydamasi(s) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;

2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;

3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs (-usi);

4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalinta(s) iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Augustinas Labutis

Patvirtinta elektroniniu būdu

Labutis, Augustinas. Statinės kodo analizės tobulinimas taikant konsensusu grįstus didžiuosius kalbos modelius. Magistro baigiamasis projektas / vadovas doc. Jonas Čeponis; Kauno technologijos universitetas, informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): informatikos inžinerija.

Reikšminiai žodžiai: statinė kodo analizė; didieji kalbos modeliai; konsensusas; automatizuotas programų taisymas; kompiliacijos pagrindu atliekamas patvirtinimas; istorinis pagrindimas.

Kaunas, 2026. 63 p.

Santrauka

Šiame magistro darbe nagrinėjama statinės kodo analizės tobulinimo problema, kylanti dėl didėjančio programinės įrangos sistemų sudėtingumo, tradicinių analizės įrankių ribotumo ir poreikio patikimiau taisyti diagnostinių pranešimų nurodomas problemas. Darbe siūlomas metodas, grindžiamas kelių didžiųjų kalbos modelių sugeneruotų pataisų vertinimu ir konsensuso principu pagrįstu galutinio sprendimo pasirinkimu. Metodo pagrindą sudaro istorinio pagrindimo procento skaičiavimas iš ankstesnių pataisymų, C# / .NET programinio kodo saugyklų analizė, diagnostinių pranešimų išgavimas kompiliacijos metu, kelių LLM kandidatinių pataisų generavimas ir jų patvirtinimas pagal objektyvius kriterijus.

Sukurtas prototipas leidžia apskaičiuoti būdingą diagnostinio atvejo pakeitimo mastą, atsisiųsti ir kompiliuoti programinio kodo saugyklas, išgauti klaidas ar įspėjimus, pritaikyti LLM sugeneruotas pataisas ir įvertinti jų tinkamumą. Pataisos vertinamos pagal kompiliacijos sėkmę, tikslinio diagnostinio pranešimo pašalinimą, naujų diagnostinių pranešimų nebuvimą ir nuokrypį nuo istorinio pagrindimo procento. Atliktas tyrimas parodė, kad siūlomas metodas yra praktiškai įgyvendinamas ir gali padėti objektyviau atrinkti patikimesnes pataisas. Rezultatai taip pat parodė, kad pažangesni modeliai pateikia stabilesnius ir dažniau kompiliuojamus sprendimus, o konsensuso metodo veiksmingumas priklauso nuo pasirinktų modelių kokybės.

Labutis, Augustinas. Improving Static Code Analysis Using a Consensus-Based Multi-LLM. Master's Final Degree Project / supervisor Assoc. Prof. Jonas Čeponis; Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Informatics Engineering.

Keywords: static code analysis; large language models; consensus; automated program repair; compilation-based validation; historical grounding.

Kaunas,2026. 63 p.

Summary

This master's thesis explores the enhancement of static code analysis in the context of increasingly intricate software systems, the constraints imposed by conventional analysis tools, and the necessity for more dependable remediation of issues identified by diagnostic messages. The proposed methodology entails the evaluation of code fixes generated by multiple large language models, with the ultimate solution being determined through a consensus-based approach. The proposed methodology integrates historical grounding, repository analysis, compilation-based diagnostic extraction, the generation of candidate fixes by multiple large language models (LLMs), and objective validation of the proposed changes.

A prototype was developed to calculate the typical scope of code changes for a diagnostic case, retrieve and compile C#/NET repositories, extract compiler warnings and errors, apply LLM-generated fixes, and evaluate their suitability. The evaluation of candidate fixes is conducted through the application of a multifaceted set of criteria, encompassing the successful compilation of the fix, the removal of the target diagnostic message, the absence of newly introduced diagnostics, and the deviation from the historical grounding percentage. The experimental evaluation demonstrated that the proposed method can be practically implemented and can support a more objective selection of reliable code fixes. The findings further suggest that more advanced models tend to yield more stable and compilable solutions, while the effectiveness of the consensus-based approach is contingent upon the quality of the selected models.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:

AI (angl. *Artificial Intelligence*) – dirbtinis intelektas;

API (angl. *Application Programming Interface*)– aplikacijų programavimo sąsaja;

APR (angl. *Automated Program Repair*) – automatizuotas programų taisymas;

AST (angl. *Abstract Syntax Tree*) – abstraktus sintaksės medis;

CFG (angl. *Control Flow Graph*) – valdymo srauto grafas;

CLI (angl. *Command Line Interface*) – komandinės eilutės sąsaja;

CWE (angl. *Common Weakness Enumeration*) – bendras programinės įrangos silpnybių sąrašas;

DSL (angl. *Domain-Specific Language*) – sričiai specifinė kalba;

GPT (angl. *Generative Pre-trained Transformer*) – generatyvus iš anksto apmokytas transformeris;

HTTP (angl. *Hypertext Transfer Protocol*) – hiperteksto perdavimo protokolas;

IoT (angl. *Internet of Things*) – daiktų internetas;

LINQ (angl. *Language Integrated Query*) – .NET užklausų technologija;

LLM (angl. *Large Language Model*) – didelis kalbos modelis;

MAS (angl. *Multi-Agent System*) – daugiaagentė sistema;

ML (angl. *Machine Learning*) – mašininis mokymasis;

PBFT (angl. *Practical Byzantine Fault Tolerance*) – praktinis Bizantijos gedimų toleravimo algoritmas;

SAST (angl. *Static Application Security Testing*) – statinis programų saugumo testavimas;

SDK (angl. *Software Development Kit*) – programinės įrangos kūrimo rinkinys;

SMT (angl. *Satisfiability Modulo Theories*) – tenkinamumo pagal teorijas sprendimas;

SQL (angl. *Structured Query Language*) – struktūrizuota užklausų kalba;

URL (angl. *Uniform Resource Locator*) – universalusis ištekliaus adresas.

Terminai:

C# – objektinė, statinio tipizavimo programavimo kalba, veikianti .NET ekosistemoje;

Daugiaagentė sistema – sistema, kurioje keli nepriklausomi agentai arba modeliai bendradarbiauja spręsdami bendrą užduotį.

Diagnostinis kodas – unikalus klaidos arba įspėjimo identifikatorius, leidžiantis susieti problemą su konkrečia analizės taisykle.

Diagnostinis pranešimas – kompiliatoriaus arba statinės analizės įrankio pateiktas pranešimas apie galimą programinio kodo problemą.

Didelis kalbos modelis – dirbtinio intelekto modelis, gebantis apdoroti ir generuoti natūralią kalbą bei programinį kodą.

Dinaminė kodo analizė – programinio kodo analizės metodas, kai programa vykdoma su konkrečiais įvesties duomenimis ir stebima jos elgsena vykdymo metu.

Istorinis pagrindimas – iš ankstesnių pataisymų gauta atskaitos reikšmė, naudojama vertinti, koks kodo pakeitimo mastas yra būdingas konkrečiam diagnostiniam atvejui.

Kandidatinė pataisa – vieno didelio kalbos modelio sugeneruotas galimas programinio kodo pataisymo variantas.

Kompiliacija – programinio kodo tikrinimo ir vertimo procesas, kurio metu gali būti nustatomos klaidos ir įspėjimai.

Kompiliacijos pagrindu atliekamas patvirtinimas – sugeneruotos pataisos tikrinimas pakartotinai kompiliuojant projektą ir vertinant, ar pataisa yra techniškai tinkama.

Konsensusas – kelių modelių rezultatų suderinamumas, leidžiantis nustatyti patikimesnį galutinį pataisymo sprendimą.

Konsensuso kriterijai – sąlygos, pagal kurias vertinama, ar pataisa tinkama: sėkminga kompiliacija, tikslinės diagnostikos pašalinimas, naujų problemų nebuvimas ir priimtinas pakeitimo mastas.

Nauji diagnostiniai pranešimai – papildomos klaidos ar įspėjimai, atsiradę po sugeneruotos pataisos pritaikymo.

Nuokrypis nuo pagrindimo procento – skirtumas tarp LLM sugeneruotos pataisos pakeitimo procento ir istorinio pagrindimo procento.

Pagrindimo procentas – procentinė reikšmė, rodanti, kokia originalaus programinio kodo dalis buvo pakeista ankstesnio pataisymo metu.

Pakeitimo procentas – procentinė reikšmė, rodanti, kokią programinio kodo dalį pakeitė LLM sugeneruota pataisa.

PLBART – programinio kodo apdorojimui taikomas transformerių architektūros modelis;

PMD – statinės kodo analizės įrankis, naudojamas programinio kodo kokybės problemoms aptikti;

Refaktoringas – programinio kodo struktūros keitimas nekeičiant jo išorinio funkcionalumo.

Repozitorija – programinio kodo saugykla, naudojama projektui atsisiųsti, kompiliuoti ir analizuoti.

Statinė kodo analizė – programinio kodo analizės metodas, kai kodas tiriamas jo nevykdant, siekiant aptikti klaidas, įspėjimus, saugumo spragas ar kokybės trūkumus.

Sutaisyti diagnostiniai pranešimai – diagnostiniai pranešimai, kurie po pataisos pritaikymo nebepasikartoja.

Techninė skola – programinio kodo ar architektūros trūkumai, kurie ilgainiui apsunkina sistemos palaikymą ir plėtrą.

.NET – „Microsoft“ programinės įrangos kūrimo platforma ir vykdymo aplinka;

Įvadas

Modernios programinės sistemos yra didelės apimties, sudėtingos ir nuolat kintančios, todėl jų kokybės, saugumo ir palaikomumo užtikrinimas tampa vis sudėtingesnis. Šioms problemoms spręsti programinės įrangos kūrimo procese plačiai taikoma statinė kodo analizė, leidžianti tirti programinį kodą jo nevykdant ir anksti nustatyti galimas klaidas, saugumo spragas ar kokybės trūkumus.

Nors statinės kodo analizės įrankiai padeda automatizuoti kodo peržiūrą ir sumažinti klaidų taisymo sąnaudas, dideliuose programinės įrangos projektuose išryškėja jų apribojimai. Tokie įrankiai gali pateikti klaidingai teigiamus pranešimus, nepakankamai įvertinti platesnį kodo kontekstą arba praleisti problemas, kurioms reikalingas gilesnis semantinis supratimas. Dėl to kūrėjams tenka rankiniu būdu vertinti diagnostinius pranešimus ir priimti sprendimus dėl jų taisymo.

Pastaraisiais metais šiuos procesus vis dažniau papildo didieji kalbos modeliai, gebantys generuoti ir taisyti programinį kodą. Tačiau vieno modelio pasiūlytas sprendimas ne visada yra teisingas ar kompiliuojamas, todėl aktualu tirti konsensusu grindžiamą metodą, kuriame keli modeliai generuoja sprendimus tai pačiai problemai, o galutinis pataisymas parenkamas pagal objektyvius vertinimo kriterijus.

Dėl šios priežasties darbo tikslas – pasiūlyti ir ištirti statinės kodo analizės tobulinimo metodą, kuris, naudojant kelių didžiųjų kalbos modelių sugeneruotus pataisymus ir konsensuso principu pagrįstą vertinimą, padėtų patikimiau taisyti statinės analizės metu aptiktas programinio kodo problemas. Šiam tikslui pasiekti suformuluoti šie uždaviniai:

1. Išnagrinėti programiniame kode atsirandančiomis problemomis ir jų priežastis;
2. Išanalizuoti statinės kodo analizės technikas, metodus ir naudojamus įrankius;
3. Išnagrinėti problemas, su kuriomis susiduria statinės kodo analizės įrankiai;
4. Išanalizuoti daugiaagenčių ir konsensuso sistemų taikymo galimybes programinio kodo taisymo procese;
5. Pasiūlyti statinės kodo analizės tobulinimo metodą, pagrįstą kelių didžiųjų kalbos modelių sugeneruotų pataisymų vertinimu;
6. Įgyvendinti pasiūlyto metodo prototipą ir atlikti jo testavimą bei veiksmingumo tyrimą.

Pirmajame darbo skyriuje analizuojamos programinio kodo problemos, programavimo kalbų evoliucija, kodo sudėtingumo didėjimas, dažniausios klaidos, statinės ir dinaminės analizės technikos bei daugiaagentės ir konsensuso sistemos. Antrajame skyriuje pristatomas siūlomas konsensusu pagrįstas statinės kodo analizės tobulinimo metodas, apimantis sistemos architektūrą, istorinio pagrindimo modelį, diagnostinių pranešimų gavimą, kelių LLM generuojamų pataisymų vertinimą ir galutinio sprendimo pasirinkimą. Trečiajame skyriuje aprašomas metodo prototipas, naudotos technologijos, pagrindiniai komponentai, kompiliacijos procesas ir konsensuso rezultatų formavimas. Ketvirtajame skyriuje atliekamas metodo testavimas, vertinant kompiliacijos sėkmę, naujai įvestas klaidas, diagnostinių pranešimų pašalinimą, nuokrypį nuo istorinių taisymų ir konsensuso sąlygų atitikimą skirtinguose modeliuose. Darbo pabaigoje pateikiamos išvados, kuriose apibendrinami tyrimo rezultatai, įvertinamas uždavinių įgyvendinimas, aptariami metodo ribotumai ir numatomos tolesnio tyrimo kryptys.

1. Problemos programiniame kode

Šiame skyriuje išsamiai nagrinėjamos pagrindinės problemos, kylančios kuriant programinę įrangą, bei metodikos, naudojamos joms nustatyti. Skyriuje gilinamasi į programavimo kalbų raidą, vis sudėtingesnes programinės įrangos sistemas, dažniausiai pasitaikančias programavimo klaidas bei statinės ir dinaminės kodo analizės svarbą užtikrinant programinės įrangos kokybę. Be to, skyriuje aptariamos statinės kodo analizės technikos, dažniausiai naudojami analizės įrankiai, daugiaagentės ir konsensuso pagrindu veikiančios sistemos bei pagrindiniai esamų kodo analizės metodų trūkumai.

1.1. Programavimo kalbų evoliucija

Programavimo kalbų istorinę trajektoriją galima apibūdinti kaip perėjimą nuo žemo lygio, mašinai artimų žymėjimų prie aukšto lygio, išraiškingų ir sričiai optimizuotų abstrakcijų. Ankstyvosios bendrosios paskirties kalbos, pavyzdžiui, „Fortran“, suteikė simbolinį žymėjimą skaitmeniniams skaičiavimams, tačiau išliko glaudžiai susijusios su aparatinės įrangos apribojimais ir buvo ribotos modulinio bei didelio masto programinės įrangos projektavimo atžvilgiu. Vėlesnių kartų kalbos įdiegė turtingesnes tipų sistemas, struktūrizuotą valdymo srautą ir didesnį programos semantikos bei jos realizavimo mašinos lygmeniu atskyrimą – tendenciją, apibūdinamą kaip nuolatinį perėjimą „nuo primityvaus mašinos kodo prie sudėtingų aukšto lygio kalbų“, kuris pakeitė programinės įrangos kūrimo praktiką. Šis ilgalaikis pokytis leido programuotojams sutelkti dėmesį į problemų sritis, o ne į žemo lygio detales, pavyzdžiui, registrų paskirstymą ar rankinį atminties valdymą [1].

```python
import math

def triangle_area(a: float, b: float, c: float) -> float:
    if a <= 0 or b <= 0 or c <= 0:
        raise ValueError("All sides must be > 0.")
    if a + b <= c or a + c <= b or b + c <= a:
        raise ValueError("Triangle inequality violated.")
    s = (a + b + c) / 2.0
    area_sq = max(0.0, s * (s - a) * (s - b) * (s - c))
    return math.sqrt(area_sq)

def main() -> None:
    raw = input("Enter A B C: ").strip()
    a_str, b_str, c_str = raw.split()
    a, b, c = map(float, (a_str, b_str, c_str))
    area = triangle_area(a, b, c)
    print(f"A={a:g}  B={b:g}  C={c:g}  AREA={area:.2f} square units")

if __name__ == "__main__":
    main()
```

```
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
          IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  777 STOP 1
  703 S = (IA + IB + IC) / 2.0
          AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
          WRITE(6,801) IA,IB,IC,AREA
  801 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.
         $13H SQUARE UNITS)
          STOP
          END
```

1 pav. „Fortran“ ir „Python“ programavimo kalbų palyginimas

Pateiktas „Fortran“ ir „Python“ pavyzdys (žr. 1 pav.) parodo ne tik sintaksinius, bet ir platesnius programavimo kalbų abstrakcijos skirtumus. Abu fragmentai sprendžia tą pačią užduotį – apskaičiuoja trikampio plotą pagal kraštines, tačiau „Python“ kode naudojamos funkcijos, aiškūs kintamųjų pavadinimai, išimtys klaidoms apdoroti ir standartinės bibliotekos funkcijos. Tuo tarpu „Fortran“ pavyzdyje matyti labiau procedūrinis, ankstesnėms kalbų kartoms būdingas stilius: naudojamos numeruotos eilutės, *FORMAT* sakiniai, *STOP* komandos ir glaudesnis ryšys su įvesties bei išvesties valdymu. Tai iliustruoja, kaip šiuolaikinės kalbos daugiau dėmesio skiria skaitomumui, moduliarumui ir klaidų valdymui, o ankstyvesnės kalbos buvo labiau orientuotos į skaičiavimų aprašymą ir vykdymo kontrolę.

Šiuolaikinės kalbos, tokios kaip „Python“, „C#“ ir „JavaScript“, iliustruoja šią evoliuciją, integruodamos kelias paradigmas – imperatyvią, objektų orientuotą ir funkcinę – į vieną ekosistemą. Šaltinyje [1] pabrėžiama šiuolaikinė aplinka, „kurioje dominuoja tokios kalbos kaip „Python“ ir

JavaScript“, kurių projektavimo tikslai akcentuoja skaitomumą, greitą prototipų kūrimą ir plačią taikymo sritį žiniatinklyje, duomenų moksle ir programų kūrime. Tokios funkcijos kaip automatinis atminties valdymas, išsamios standartinės bibliotekos ir dinamiškos arba valdomos vykdymo aplinkos yra sąmoninga abstrakcija nuo aparatinės įrangos, gerinanti kūrėjų produktyvumą, kodo priežiūrą ir perkeliamumą [1]. „C#“ ypač iliustruoja statinio tipizavimo ir objektų orientuotų kalbų brandą, nes ši kalba siejama su pažangiais įrankiais ir vykdymo aplinkos saugumo garantijomis per „.NET“ platformą.

Kalbų evoliucija taip pat apima diversifikaciją į specifines srities kalbas (*DSL – domain specific languages)* ir specializuotas ekosistemas. Šaltinyje [1] minimos kalbos, tokios kaip „SQL“, skirta reliacinėms duomenų bazėms, ir „Swift“, skirta „iOS“ kūrimui, rodo, kaip kalbos dabar yra kuriamos siekiant optimizuoti išraiškingumą ir efektyvumą konkrečiose taikymo srityse. Šią diversifikaciją lydi atsinaujinęs susidomėjimas funkcinėmis kalbomis, tokiomis kaip „Haskell“ ir „Scala“, kurios skatina nekintamumą ir aukštesnio lygio abstrakcijas. Šie stiliai daro įtaką pagrindinėms kalboms per tokias funkcijas kaip lambda išraiškos, „LINQ C#“ kalboje ir funkcinės bibliotekos „Python“ kalboje. Tuo pat metu evoliucinių skaičiavimų ir didelių kalbos modelių tyrimai nagrinėja automatinį kodo generavimą bei optimizavimą. Tai keičia programuotojų darbą su programavimo kalbomis, nes vis daugiau užduočių atliekama aukštesniu abstrakcijos lygiu [2].

Šiuolaikiniai tyrimai, susiję su kodo generavimu ir programų sinteze, pabrėžia, kaip programavimo kalbos vaidmuo plečiasi nuo vien žmonėms skirtos notacijos iki bendros žmonių ir DI įrankių sąsajos. Šaltinyje [2] apžvelgiamos sistemos, kuriose didieji kalbos modeliai generuoja pradinius programų kandidatus tokiomis kalbomis kaip „Python“, o šie kandidatai vėliau tobulinami evoliuciniais algoritmais, naudojančiais abstrakčius sintaksės medžius ir gramatikos pagrįstus apribojimus. Taip galiausiai gaunamas labiau optimizuotas arba teisingesnis kodas. Kiti metodai naudoja evoliucinę paiešką, kad pritaikytų užklausas ar kodo struktūras, todėl kalbos modeliai sukuria funkcionalesnes ar saugesnes programas [2]. Ši sinergija iliustruoja naują programavimo praktikos evoliucijos etapą, kuriame „kalba“ apima ne tik sintaksę ir semantiką, bet ir jos pritaikomumą automatizuotam samprotavimui, transformavimui ir tikrinimui.

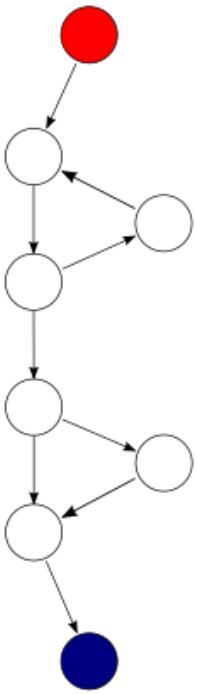
Ateities programavimo kalbų tyrimų kryptys rodo, kad bus siekiama stipresnių integruotų patikimumo ir saugumo garantijų. Šaltinyje [1] teigiama, kad naujos kalbos turėtų būti labiau integruotos su naujomis technologijomis, pavyzdžiui, kvantiniu skaičiavimu, ir įtraukti etinius projektavimo principus, o tai reiškia, kad teisingumas, patikimumas ir socialinis poveikis taps svarbiais projektavimo kriterijais [1]. Tuo pačiu metu šaltinyje [2] aptariamas į saugumą orientuotas kodo generavimas atskleidžia dabartinių kalbų ir įrankių grandinių pažeidžiamumą sunkiai pastebimiems defektams ir priešiškiems manipuliavimams, o tai skatina gramatika apribotą sintezę ir griežtą sintaksinių bei semantinių invariantų taikymą programų evoliucijos metu. Kartu šios tyrimų kryptys rodo, kad ateities programavimo kalbos greičiausiai bus labiau optimizuotos saugumo požiūriu, derinant turtingesnes statines garantijas, formaliai apibrėžtą semantiką ir DI padedamą patikrinimą. Tai galėtų sumažinti vykdymo klaidų ir saugumo trūkumų, kurie vis dar yra paplitę šiuolaikinėse bendrosios paskirties kalbose, skaičių.

1.2. Programavimo sudėtingumas didėjant programinio kodo kiekiui

Vis daugiau naujausių atvirosios prieigos tyrimų rodo, kad programinės įrangos sistemos tampa didesnės ir sudėtingesnės, todėl tyrimų ir praktikos bendruomenė yra skatinama tobulinti matavimo,

testavimo ir kokybės užtikrinimo metodus, kad galėtų susidoroti su šiuo augimu. Šis sudėtingumas vis labiau susijęs su naujomis paradigmomis, pavyzdžiui, objektiniu projektavimu, mikropaslaugų architektūra ir mašininio mokymosi ar didžiųjų kalbos modelių pagrindu veikiančiomis sistemomis, kurios suteikia naujų galimybių, bet kartu kelia papildomų struktūrinių ir saugumo rizikų.

Šiuolaikinės sistemos laikui bėgant kaupia vis daugiau modulių, atributų ir logikos, o tai didina struktūrinį ir loginį sudėtingumą bei mažina jų prižiūrimumą ir suprantamumą. Maina ir kt. dokumentuoja, kaip tradiciniai rodikliai, tokie kaip kodo eilučių skaičius, Halstead rodikliai ir ciklomatinis sudėtingumas, turi būti nuolat tobulinami, nes programinės įrangos dydis ir architektūros sudėtingumas nuolat didėja. Dėl to sudėtingumas tampa nuolatine ir vis didėjančia problema [3]. Chen ir Babar taip pat rodo, kad mašininio mokymosi pagrindu veikiančios sistemos per visą gyvavimo ciklą įtraukia papildomus komponentus, duomenų srautus ir modelių artefaktus, taip didindamos programinės įrangos sistemų struktūrinę apimtį [4].



2 pav. Paprastos programos ciklomatinio sunkumo grafas

2 pav. pateiktas grafas vaizduoja paprastą programą, kurios vykdymas prasideda įėjimo mazge, tęsiasi per kilpos struktūrą, o tada susiduria su sąlygine šaka prieš pasiekiant išėjimo mazgą. Grafike yra 8 mazgai, 9 kraštai ir 1 sujungtas komponentas. Naudojant „McCabe“ ciklomatinio sudėtingumo formulę:

$$M = E - N + 2P \quad (1)$$

kur E – kraštų skaičius, N – mazgų skaičius, o P – sujungtų komponentų skaičius. Įstačius reikšmes į formulę gaunama M = 9 - 8 + 2 · 1 = 3, todėl programos ciklomatinis sudėtingumas yra 3. Tai reiškia, kad programa turi tris nepriklausomus vykdymo kelius.

Naujausi tyrimai rodo, kad LLM daugiausia naudojami programinės įrangos analizės ar testavimo užduotims automatizuoti, pavyzdžiui, programų taisymui, testų generavimui ar statinės analizės pagalbai. Tokiu būdu didinama sukurto kodo ir artefaktų apimtis, tačiau savaime neužtikrinama geresnė architektūra ar paprastesnis dizainas. Maina ir kt. pabrėžia, kad didelis sudėtingumas kyla dėl

daugybės atributų ir sudėtingų loginių ryšių ir lieka „nepageidaujamas“, nes kenkia priežiūrai, neatsižvelgiant į tai, kaip kodas buvo sukurtas [4]. Be aiškios sudėtingumo kontrolės, LLM sugeneruoti pataisymai, testai ar jungiamasis kodas gali paspartinti kodo apimties, techninės skolos ir sudėtingų valdymo kelių augimą, o ne sistemingai gerinti struktūrinę kokybę [3], [4].

Objektinis projektavimas, komponentizavimas ir mašininio mokymosi principais grindžiamos architektūros įveda naujus abstrakcijos sluoksnius, sąveikos modelius ir priklausomybes, kurias klasikinės metrikos tik iš dalies atspindi. Maina ir kt. pažymi, kad objektinio programavimo sistemoms reikalingos paradigmai būdingos sudėtingumo metrikos, pavyzdžiui, klasių, paveldėjimo hierarchijų ir projektavimo diagramų vertinimui, būtent todėl, kad šios paradigmos prideda naujas sudėtingumo dimensijas, kurios turi įtakos suprantamumui ir testavimo pastangoms [3]. Chen ir Babar teigia, kad ML pagrįstos programinės įrangos sistemos taip pat turi integruoti duomenų srautus, modelių valdymą ir saugumo mechanizmus, kurie išplečia atakos paviršių ir apsunkina sistemos kaip visumos vertinimą [4]. Taigi šis paradigmų ir infrastruktūrų gausėjimas didina sisteminį sudėtingumą net ir tada, kai atskiri komponentai atrodo gerai struktūrizuoti.

Tyrimuose taip pat sutariama, kad didesnis sudėtingumas ir didesnės kodo bazės kenkia programinės įrangos saugumui, didindamos klaidų tikimybę ir plėsdamos atakos paviršių. Maina ir kt. aiškiai apibūdina padidėjusį sudėtingumą kaip nepageidaujamą ir sieja jį su prastu dizainu, priežiūros problemomis ir padidėjusiu testavimo sudėtingumu – sąlygomis, kurios, kaip žinoma, didina gedimų ir defektų tikimybę [3]. Chen ir Babar tiria mašininio mokymosi pagrindu veikiančias programinės įrangos sistemas ir rodo, kad skubotas kūrimas, techninė skola ir sudėtingi, nepakankamai apsaugoti ML komponentai sukelia pažeidžiamumus ir privatumo problemas, kurias sunku aptikti ir sumažinti didelėse, sudėtingose sistemose [4]. Šie rezultatai kartu patvirtina išvadą, kad nekontroliuojamas kodo eilučių skaičiaus ir architektūros sudėtingumo augimas mažina saugumą, jei tai nekompensuojama sistemingais sudėtingumo rodikliais, refaktoringu ir saugumo projektavimo praktika.

1.3. Dažniausiai atliekamos klaidos programiniame kode

Šiuolaikinės programavimo kalbos ir programinės įrangos platformos vis dažniau įtraukia saugumo ir patikimumo mechanizmus, kurie automatizuoja kritinių klaidų prevenciją. Dikici ir Bilgin pabrėžia, kad šiuolaikinės kalbos lygio saugumo funkcijos, pavyzdžiui, stipresnės tipų sistemos, atminties saugumo konstrukcijos ir lygiagretaus veikimo abstrakcijos, veiksmingai užkerta kelią daugeliui sintaksinių ir žemo lygio defektų dar prieš jiems pasireiškiant programos vykdymo metu Tuo pačiu metu modernios APR (angl. *automated program repair*) sistemos – šablonų pagrindu veikiančios, mašininio mokymosi ir giluminio mokymosi metodais pagrįstos priemonės – pasiekia aukštus taisymo rodiklius pasikartojančių modelių, pavyzdžiui, dažnų aritmetinių klaidų, *null* reikšmių patikrinimų ir paprastų valdymo srauto pataisymų, atveju. Kai kuriose kategorijose pranešama apie 80–90 % sėkmės rodiklius [5]. Kartu šios tendencijos rodo, kad techninė ekosistema jau siūlo galingas automatizuotas apsaugos priemones nuo daugelio didelio poveikio, bet struktūriškai paprastų defektų.

Tačiau literatūroje taip pat nurodoma, kad daugybė problemų lieka neišspręsta. Net tose srityse, kuriose APR įrankiai yra ištobulinti, sėkmės rodikliai reikšmingai sumažėja, kai susiduriama su sudėtingesnėmis klaidomis, pavyzdžiui, sudėtingais valdymo srauto defektais, API naudojimo klaidomis, kurioms išspręsti reikalingos srities žinios, ir lygiagretaus veikimo klaidomis, kurių ištaisymo rodikliai yra gerokai žemesni nei paprastesnių kategorijų klaidų [5]. Panašiai Zhang ir kt.

teigia, kad nors „ChatGPT“ gali automatiškai ištaisyti didelę dalį etaloninių klaidų, jo rezultatai priklauso nuo užklausų formulavimo, ir nemažas skaičius klaidų lieka neištaisytas net po kelių sąveikos iteracijų [6]. Abu darbai pabrėžia, kad klaidų lokalizavimas, semantinis programos tikslo supratimas ir patikimas pataisų patvirtinimas vis dar yra neišspręstos problemos, todėl šiuolaikinės sistemos ir įrankiai tik iš dalies aprėpia faktines klaidas įdiegtose programose [5], [6].

Sparčiai integruojant didžiuosius kalbos modelius (LLM) į programinės įrangos kūrimo procesus, situacija dar labiau komplikuojasi. LLM gali generuoti ir taisyti kodą dideliu mastu, o ChatGPT jau pranoksta ankstesnius į kodą orientuotus modelius, tokius kaip CodeT5 ir PLBART, konkurencinio programavimo klaidų testuose, ištaisydamas daugiau nei 100 iš 151 klaidingų programų pagal pagrindinius nurodymus [6]. Tuo pačiu metu tiek APR tyrimas, tiek ChatGPT tyrimas pabrėžia LLM generuojamų pataisų ir kodo patvirtinimo sąnaudas bei sudėtingumą, nes funkcinis teisingumas ir nefunkcinių apribojimų laikymasis nėra garantuotas [5], [6]. Kadangi LLM didina sukurto ir modifikuoto kodo apimtį, neišspręsti apribojimai klaidų lokalizavimo, semantinio samprotavimo ir automatizuotų testų tinkamumo srityse kelia pavojų, kad kodų bazėse atsiras daugiau paprastų, bet nepatikrintų defektų.

APR vertinimų duomenys rodo, kad būtent tos paprastos defektų klasės, kurias automatizuotos priemonės tvarko geriausiai, pavyzdžiui, dažni valdymo srauto nukrypimai, trūkstami patikrinimai ir stereotipiniai API netinkamo naudojimo modeliai, taip pat yra vieni iš dažniausiai pasitaikančių klaidų tipų šiuolaikiniuose etalonuose [5]. Zhang ir kt. taip pat rodo, kad daugelis klaidų neseniai pateiktose konkurencinėse programose priklauso palyginti nedidelėms, pasikartojančioms kategorijoms, kurias LLM dažnai gali ištaisyti, tačiau ne pakankamai patikimai, kad būtų galima atsisakyti sistemingo patvirtinimo [6]. Atsižvelgiant į automatizacijos skatinamą kodo apimties didėjimą, šie rezultatai rodo, kad tokių iš pažiūros paprastų problemų dažnumas realaus pasaulio kodo bazėse nemažėja. Priešingai, jos kartojasi dideliu mastu, todėl būtent tos klaidos, kurias šiuolaikinės sistemos ir APR / LLM įrankiai gali aptikti ir ištaisyti, tampa vis dažnesnės. Dėl to likusi programinės įrangos kokybės rizika išlieka nuolat didelė [5], [6].

1.4. Statinės kodo analizės technikos

Statinė kodo analizė yra technika, skirta programinio kodo arba baitų kodo tyrimui, siekiant nustatyti semantines savybes, pavyzdžiui, galimas klaidas ar pažeidžiamumus, faktiškai nevykdant programos. Analizatoriai sukuria formalius kodo modelius, tada taiko algoritminį samprotavimą, kad apytiksliai įvertintų visus galimus vykdymo scenarijus, todėl jie gali anksti aptikti defektų klases kūrimo ciklo metu. Kadangi realūs įvesties duomenys nevykdomi, šios priemonės remiasi valdymo srauto grafikais, vertės srauto modeliais ir simboliniais apribojimais, o ne vykdymo stebėjimais.

Šiuolaikinė statinė analizė grindžiama sistemingu tarpinės reprezentacijos, kuri patikimai, bet konservatyviai aproksimuoja visus galimus programos elgesio modelius, kūrimu ir tyrinėjimu. Įrankiai paprastai paverčia kodą abstrakčiais sintaksės medžiais ir tada išveda turtingesnes struktūras, pavyzdžiui, vertės srauto ar priklausomybės grafikus, kurie atspindi, kaip vertės plinta per sakinius ir visą programą. Analizės veikia kaip fiksuotų taškų skaičiavimai šiose struktūrose, kartotinai skleidžiant abstrakčias būsenas, kol gaunamas stabilus visų pasiekiamų konfigūracijų modelis, pagal kurį tikrinamos saugos savybės, pavyzdžiui, tam tikrų pažeidžiamumo modelių nebuvimas [7]. Šis konservatyvus apytikslis apskaičiavimas yra būtinas didelėms sistemoms analizuoti, tačiau neišvengiamai sukelia klaidingų teigiamų rezultatų. Todėl naujausi darbai sutelkia dėmesį į

architektūras, kurios glaudžiai integruoja greitą vertės srauto analizę su simboliniais sprendėjais, kad išlaikytų aukštą tikslumą, kartu išlikdamos praktiškai pritaikomos realaus pasaulio išmaniosioms sutartims ir sudėtingai programinei įrangai [7], [8].

Simbolinis vykdymas yra keliams jautrus statinės analizės metodas, kuris interpretuoja kodą naudodamas simbolinius įvesties duomenis, o ne konkrečias vertes, išlaikydamas logines kelio sąlygas, kurios apibrėžia kiekvienam tiriamam vykdymo keliui reikalingus apribojimus. Kiekvienai šakai analizė išsišakoja, sujungdama šakos predikatus su esama kelio sąlyga, SMT ar kitam simboliniam sprendėjui pateikiamos patenkinamumo užklausos, kurios nustato kelio įgyvendinamumą ir leidžia išgauti konkrečius priešingus pavyzdžius, kai pažeidžiama saugos sąlyga. Naujausi darbai rodo, kad simbolinio samprotavimo ir vertės srauto analizės derinimas leidžia kurti simbolinės vertės srauto analizės architektūras, kurios modeliuoja programos elgesį beveik pilnai atsižvelgdamos į kelio jautrumą, reikšmingai padidindamos sakinių aprėptį ir sumažindamos klaidingus teigiamus rezultatus, palyginti su tradiciniais simbolinio vykdymo įrankiais saugumo požiūriu kritinėse srityse, pavyzdžiui, „Ethereum“ išmaniosiose sutartyse [7], [8]. Tokios platformos kaip „Desyan“ apibendrina šią integraciją, įterpdamos simbolinį sprendimą priimantį mechanizmą į našų „Datalog“ pagrįstą analizės variklį, leidžiantį analizės projektuotojams sklandžiai derinti lengvą simbolinį vertinimą ir visišką SMT sprendimą priimantį procesą, kaip to reikalauja tiriamoji savybė [8].

Valdymo srauto grafikai (CFG) yra statinės analizės pagrindas, nes jie koduoja visus galimus valdymo perdavimus tarp pagrindinių blokų, suteikdami struktūrinį pagrindą, kuriuo remiasi aukštesnio lygio analizės. Kiekvienas mazgas atitinka tiesinį kodo segmentą, o kraštai žymi šakas, kilpas, funkcijų iškvietimus ir išimtinius srautus, taip apibrėždami potencialių vykdymo kelių rinkinį, į kurį turi atsižvelgti simbolinė arba vertės srauto analizė. Analizuojant žemo lygio taikinius, pavyzdžiui, „Ethereum“ baitkodą, tikslus CFG sudarymas yra sudėtingas dėl netiesioginių perėjimų ir apskaičiuotų perėjimo tikslų, netikslūs CFG tiesiogiai sumažina pažeidžiamumo aptikimo veiksmingumą. Naujausi tyrimai rodo, kad statinio samprotavimo papildymas simboliniu operandų steko modeliavimu leidžia atlikti labai tikslų CFG atkūrimą, o tai savo ruožtu pagerina tolesnių pažeidžiamumo detektorių, pavyzdžiui, pakartotinio įėjimo ir prieigos kontrolės trūkumų, tikslumą, palyginti su ankstesniais įrankiais [7]. Tokie CFG orientuoti metodai iliustruoja, kaip valdymo srauto modeliavimo tikslumo gerinimas gali pagerinti visą statinės analizės procesą.

Statinio programų saugumo testavimo (SAST) įrankiai įgyvendina šiuos principus analizuodami patį šaltinio kodą, transformuodami jį į vidinius modelius, pavyzdžiui, AST, CFG ir vertės srauto grafikus, ir tada taikydami taisyklėmis pagrįstą modelių atpažinimą, užteršimo analizę ir simbolinį samprotavimą, kad nustatytų galimas pažeidžiamas vietas. Šiuolaikiniai SAST ir pažangių statinių analizatorių tyrimai pabrėžia, kad veiksmingumas labai priklauso nuo pagrindinių reprezentacijų, ypač valdymo ir vertės srauto grafikų, turtingumo ir simbolinių sprendimų, galinčių pašalinti neįmanomus kelius, integracijos. Taip sumažinamas klaidingų teigiamų rezultatų skaičius ir išlaikoma didelė aprėptis didelėse, saugumui kritiškai svarbiose programinio kodo saugyklose [7], [8].

1.5. Statinės kodo analizės įrankiai

Statinės analizės įrankiai yra automatizuotos priemonės, skirtos programinės įrangos artefaktams tirti, siekiant nustatyti defektus ir pažeidžiamumus prieš programos vykdymą. Jie taikomi kokybės

užtikrinimo procesuose įvairiomis kalbomis ir platformomis, padėdami anksti aptikti saugumo trūkumus, priežiūros problemas ir kodo kokybės problemas. Nepaisant šių įrankių brandos, praktinis jų naudojimas atskleidžia svarbius apribojimus, įskaitant neišsamią pažeidžiamumų aprėptį, didelį klaidingų teigiamų ar klaidingų neigiamų rezultatų skaičių ir sunkumus pritaikant juos skirtongomis pramoninio masto sistemoms. Šie apribojimai skatina sistemingus empirinius vertinimus ir hibridinių metodų, derinančių taisyklėmis pagrįstą analizę su gilesniu semantiniu vertinimu ir mokymosi modeliais, tyrimą.

Empirinis plačiai naudojamų statinės analizės įrankių, pavyzdžiui, FindBugs, PMD, ESC/Java ir Java Pathfinder, vertinimas, palyginti su standartizuotais testų rinkiniais, tokiais kaip Juliet, rodo didelius rezultatų skirtumus ir tai, kad nėra vieno dominuojančio įrankio. Įrankiai paprastai pasiekia aukštą tikslumą, dažnai viršijantį 80 %, tačiau palieka daug neaptiktų pažeidžiamumų net ir palyginti mažuose, dirbtiniuose testuose [9]. Atlikus platesnį kelių SAST įrankių palyginimą nustatyta, kad realiame Java kode pranešama tik apie nedidelę dalį žinomų pažeidžiamumų, o įrankių derinimas padidina aprėptį, tačiau vis tiek palieka neaptiktą daugumą trūkumų [10]. Šie rezultatai rodo, kad praktikoje dažnai reikia įrankių rinkinių ir papildomų technikų, kad būtų galima užtikrinti priimtiną saugumo lygį.

Vertinami SAST įrankiai daugiausia orientuoti į saugumo pažeidžiamumus, pavyzdžiui, įterpimo atakas, išteklių kontrolės problemas ir nepakankamą įvesties tikrinimą, taip pat į tam tikrą teisingumo ir patikimumo trūkumų pogrupį [9], [10]. Daugelis analizatorių naudoja taisyklėmis pagrįstus detektorius, kurie aptinka sintaksinius modelius arba paviršutiniškas duomenų srauto konfigūracijas, susietas su CWE tipo kategorijomis [9]. Kiti įtraukia daugiau semantinio vertinimo, pavyzdžiui, tarpprocedūrinį duomenų srautą ir kelio jautrumą, tačiau lieka apriboti taisyklių rinkiniais ir neišsamiais aplinkos modeliais [10]. Abiejų tyrimų rezultatai rodo, kad net ir esant pažeidžiamumo tipo taisyklėms, realūs atvejai, ypač tie, kuriuose pasireiškia sudėtingos duomenų ir kontrolės srauto sąveikos, dažnai yra praleidžiami. Tai pabrėžia daugiausia taisyklėmis grindžiamų metodų ribotumą.

Pereinant nuo sintetinių testų rinkinių prie realių projektų, tampa akivaizdūs mastelio, tikslumo ir atgaminimo kompromisai. Didelio masto septynių atvirojo kodo Java SAST įrankių vertinimas rodo, kad jie gerai veikia sintetiniuose testuose, bet realaus pasaulio testuose aptinka tik apie 12,7 % pažeidžiamumų, o maždaug 70,9 % problemų lieka nepastebėtos net ir sujungus visų įrankių rezultatus [10]. Siekiant išlaikyti priimtiną analizės trukmę didelėse kodo bazėse, įrankiai riboja kelio ir konteksto jautrumą. Tai pagerina veikimo laiką, tačiau padidina klaidingų neigiamų rezultatų skaičių tais atvejais, kai pažeidžiamumams nustatyti reikia visos programos analizės [9], [10]. Apskritai dabartiniai statiniai analizatoriai yra pritaikyti didelėms sistemoms, tačiau tai pasiekiama sumažinus semantinį tikslumą ir nevisiškai aprėpiant kritines defektų klases.

1.6. Dinaminė kodo analizė

Dinaminė kodo analizė reiškia metodų grupę, kuri tiria programinę įrangą stebėdama jos veikimą vykdymo metu. Vietoj to, kad būtų vertinamos tik sintaksinės ar struktūrinės šaltinio ar dvejetainio kodo savybės, dinaminiai metodai vykdo programas su konkrečiais įvesties duomenimis ir renka vykdymo informaciją, pavyzdžiui, kontrolės srauto sekas, atminties prieigą ir sąveiką su operacine sistema. Ši vykdymu ir elgsena grindžiama perspektyva leidžia aptikti defektus ir pažeidžiamumus, kurie pasireiškia tik tam tikromis vykdymo sąlygomis, įskaitant subtilią sąveiką su vykdymo aplinka, išorinėmis paslaugomis ir aparatinės įrangos ištekliais.

Praktikoje dinaminė kodo analizė atliekama naudojant įvairias vykdymo stebėjimo ir instrumentavimo priemones, kurios integruojamos į eksploatacinę ar bandymo aplinką. Didelės apimties sistemose vykdymo stebėjimo sistemos ir paskirstytojo sekimo infrastruktūros perima užklausas, registruoja vykdymo įvykius ir generuoja išsamius sekimus, kad būtų galima diagnozuoti gedimus, optimizuoti našumą ir pagrįsti architektūrinius sprendimus. Pavyzdžiui, „Tigris“ yra sistema, kuri apima dviejų etapų stebėjimo metodą: pirmasis apibendrintas etapas identifikuoja atitinkamas vykdymo sritis su mažomis papildomomis išlaidomis, po jo taikomas detalusis etapas, kurio metu renkami išsamūs vykdymo duomenys tik pasirinktiems sekimams, taip suderinant stebėjimą ir našumą [11]. Tokios priemonės remiasi konfigūruojamomis domeno specifinėmis kalbomis ir aktualumo metrikomis, kad instrumentavimas būtų sutelktas į kodo sritis, kurios yra svarbiausios sistemos taikymui ar paslaugų kokybės tikslams.

Dinaminė analizė yra svarbi norint nustatyti vykdymo išimtis, atminties klaidas ir lygiagretaus veikimo problemas, tačiau ji turi būti kruopščiai suprojektuota, kad būtų išvengta nepriimtinų papildomų išlaidų ir mastelio problemų. „Tigris“ tyrimas rodo, kad nekontroliuojamas išsamus stebėjimas kai kuriais atvejais gali padidinti vykdymo laiką net kelis kartus, todėl jis netinka realiam naudojimui ir yra ribojamas tik autonominiais bandymais [11]. Siekiant šią problemą sušvelninti, šiuolaikinės technologijos naudoja atrankos ir filtravimo strategijas, taip pat etapinį stebėjimą, kad sumažintų sekimo apimtį, bet vis tiek užfiksuotų gedimus, pavyzdžiui, periodines vykdymo išimtis, nuo aplinkos priklausančias atminties klaidas ir su lygiagretiškumu susijusias veikimo anomalijas. Nepaisant to, nė vienas iš pagrindiniame „Tigris“ tyrime nagrinėtų metodų negarantuoja, kad atrinkti sekimai yra visiškai reprezentatyvūs, o tai reiškia, kad lieka likutinė rizika, jog tam tikros retos lygiagretaus veikimo klaidos ar atminties gedimai gali likti nepastebėti vykdymo metu [11].

Apibendrinant, statinės ir dinaminės technikos suteikia viena kitą papildančių perspektyvų programos elgsenai. Statinė analizė veikia nevykdant programos ir leidžia didelėje apimtyje anksti aptikti galimus defektus, analizuojant kodo struktūrą ir semantiką, tačiau susiduria su aplinkos ir įvesties priklausomos elgsenos vertinimo sunkumais. Dinaminė analizė, priešingai, reikalauja programos vykdymo su stebėjimu ar instrumentavimu. Tai leidžia tiesiogiai stebėti vykdymo išimtis, atminties klaidas ir lygiagretaus veikimo problemas realiomis darbo apkrovomis, tačiau lemia vykdymo sąnaudas, dalinę aprėptį ir potencialius stebėjimo trūkumus, kai naudojamas atrankos ar filtravimo metodas [11].

1.7. Statinė kodo analizė ir dinaminė kodo analizė

Hibridinė statinė ir dinaminė analizė derina kodo tikrinimą kompiliavimo metu su elgsenos stebėjimu vykdymo metu, siekiant pagerinti pažeidžiamumų aptikimą. Statinė analizė užtikrina plačią aprėptį ir ankstyvą aptikimą, tačiau jai būdingi klaidingi teigiamieji rezultatai, o dinaminė analizė teikia tikslius, į kontekstą atsižvelgiančius rezultatus, tačiau jos aprėptis yra ribota, o išteklių sąnaudos didesnės. Integravus šias dvi analizes, pasinaudojama jų viena kitą papildančiais privalumais, siekiant pagerinti tikslumą, aprėptį ir patikimumą.

Hibridiniuose modeliuose statinė analizė naudojama dinaminiam testavimui prioritetizuoti ir nukreipti, nustatant didelės rizikos kodo sritis, valdymo srauto maršrutus arba užterštumo srautus. Tai sumažina paieškos erdvę tokioms technikoms kaip sąlyginis ar simbolinis vykdymas, taip pagerinant mastelį ir aptikimo efektyvumą [12]. Be to, statinės savybės, pavyzdžiui, valdymo srauto struktūros

ir API taškai, gali nukreipti dinaminę instrumentaciją, užtikrinant, kad būtų išbandyti kritiniai keliai, ir leidžiant aptikti kitaip paslėptas pažeidžiamumo vietas.

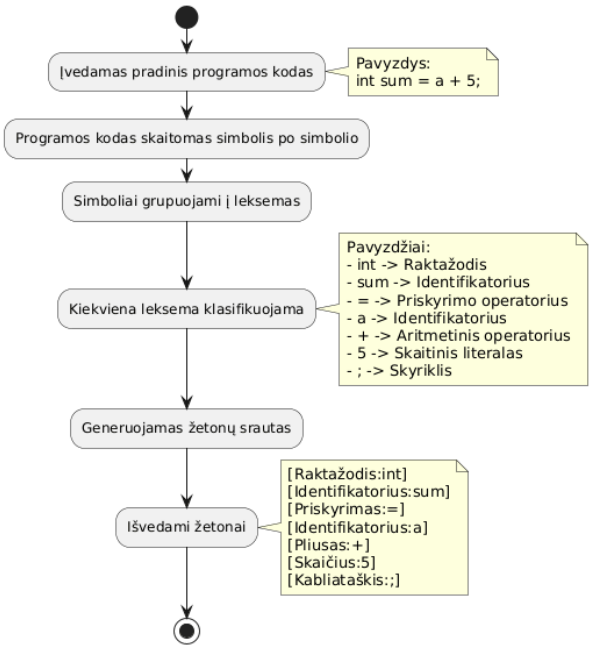
Hibridiniai metodai taip pat tikrina statinius įspėjimus vykdymo metu, taip sumažindami klaidingų teigiamų rezultatų skaičių, nes patikrina, ar nurodytos problemos yra pasiekiamos ir ar jas galima išnaudoti. Tai padeda tiksliau nustatyti rizikos prioritetus, ypač didelio masto sistemose [12]. Be to, sujungus statinius ir dinaminius duomenis į vientisus aptikimo modelius, padidėja atsparumas kodo užmaskavimui ir apgaulėms, o tai pagerina tiek žinomų, tiek nežinomų grėsmių nustatymą.

Hibridinė analizė derina plačią statinę aprėptį su tikslia vykdymo metu atliekama patikra, taip reikšmingai padidindama aptikimo tikslumą ir sumažindama klaidingų teigiamų rezultatų skaičių. Tačiau dėl įrankių koordinavimo ir padidėjusių duomenų apdorojimo reikalavimų ji lemia papildomą sudėtingumą ir išteklių sąnaudas. Todėl norint ją veiksmingai įdiegti, reikia suderinti geresnius saugumo rezultatus su skaičiavimo ir inžineriniais sąnaudų apribojimais.

1.8. Statinės kodo analizės technikos ir metodai

1.8.1. Leksinė analizė

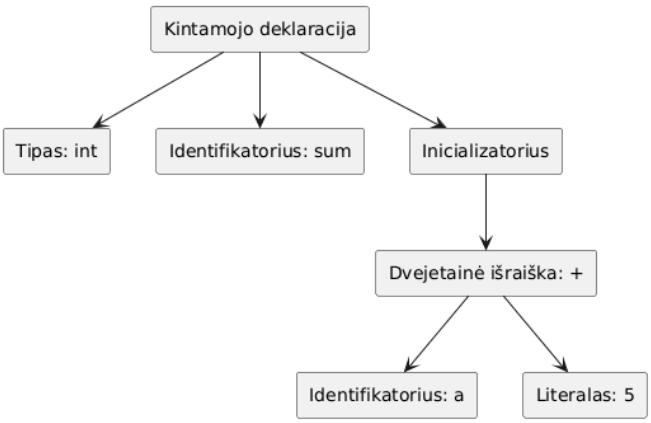
Leksinė analizė yra pirmasis statinės kodo analizės etapas, kurio metu šaltinio kodas suskaidomas į leksemas – atominius vienetus, pavyzdžiui, raktinius žodžius, identifikatorius, konstantas ir operatorius. Šis procesas sudaro sąlygas tolesniam sintaksiniam analizavimui, pateikdamas struktūrizuotą kodo tekstinių elementų atvaizdą. Šio proceso darbo eiga pateikta 3 pav. Leksiniai analizatoriai nustato pagrindines sintaksines klaidas, pavyzdžiui, neleistinus simbolius, ir užtikrina kalbos specifinių taisyklių laikymąsi [13]. Išplėstiniuose statiniuose analizatoriuose, skirtuose automobilių ar daiktų interneto (IoT) programinei įrangai, leksinė analizė dažnai integruojama su taisyklių tikrinimo varikliais, siekiant užtikrinti atitiktį pramonės standartams [14].



3 pav. Leksinės analizės darbo eiga

1.8.2. Sintaksės analizė

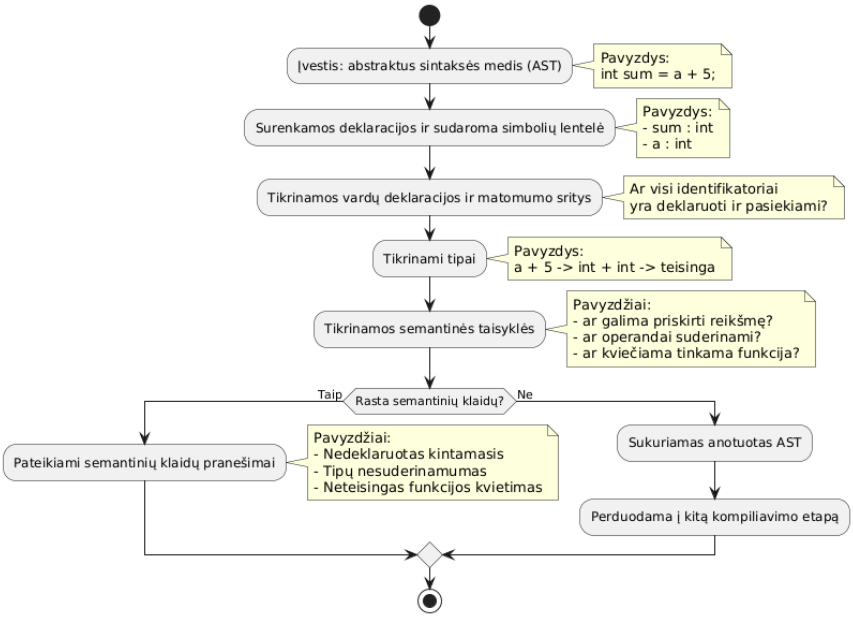
Sintaksės analizė, dar vadinama sintaksės išskaidymu, yra procesas, kurio metu iš leksinės analizės metu gautų leksemų srautų sudaromos hierarchinės struktūros, pavyzdžiui, abstrakčiosios sintaksės medžiai (AST). AST yra programos struktūros gramatinis atvaizdas, apibrėžtas pagal kalbos taisykles, tokio medžio pavyzdys pateiktas 4 pav.. Sintaksės analizatoriai yra programinės įrangos įrankiai, kurie aptinka programavimo kalbų struktūrines klaidas. Šios klaidos apima trūkstamas skliaustelius ar neteisingą teiginių tvarką. Nustatydami šias klaidas, sintaksės analizatoriai sudaro sąlygas tolesnei analizei, pateikdami organizuotą programos konstrukcijų vaizdą. Šis vaizdas leidžia efektyviau analizuoti programos struktūrą [15], [13]. Kaip rodo [15], šiuolaikiniai statiniai analizatoriai naudoja abstrakčiuosius sintaksės medžius (AST), siekdami palengvinti daugiakalbinį palaikymą ir aukšto lygio komponentų išgavimą sudėtingose sistemose.



4 pav. Abstraktaus sintaksės medžio (AST) pavyzdys

1.8.3. Semantinė analizė

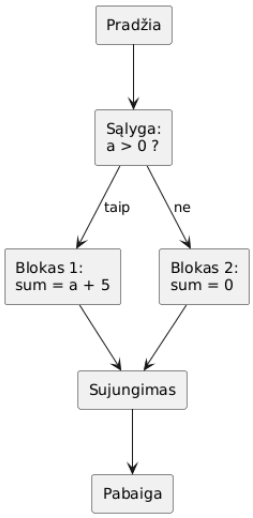
Semantinė analizė – tai sintaksinių konstrukcijų reikšmės aiškinimo procesas, kurio metu taikomos tipų taisyklės, apibrėžiama galiojimo sritis ir nustatomi kintamųjų priskyrimai. Kaip parodyta [13] darbe, ši analizė papildo AST mazgus semantine informacija, įskaitant duomenų tipus ir simbolių nuorodas (žr. 5 pav.). Šis žingsnis skirtas nustatyti gilesnes logines klaidas, pavyzdžiui, tipų nesutapimus ar nedeklaruotus kintamuosius, kurie nėra lengvai pastebimi vien tik išnagrinėjus sintaksę. Srityse, kuriose saugumas yra ypač svarbus, pavyzdžiui, automobilių pramonėje, semantiniai patikrinimai yra esminiai siekiant patikrinti atitiktį šioms sritims būdingiems standartams [14]. Naujausi tyrimai pabrėžia iššūkius, susijusius su visapusiškos semantinės aprėpties pasiekimu skirtingose arba sparčiai besikeičiančiose sistemose [14], [15].



5 pav. Semantinės analizės darbo eiga

1.8.4. Valdymo srauto analizė

Valdymo srauto analizė – tai procesas, kurio metu, sudarant valdymo srauto grafikus (CFG), tiriamos galimos programos vykdymo trajektorijos. CFG – tai duomenų struktūra, modeliuojanti, kaip valdymas juda tarp teiginių ar blokų, remiantis sąlygomis ir kilpomis [13], ši struktūra pavaizduota 6 pav. Šios technikos naudojimas palengvina nepasiekiamo kodo, kitaip vadinamo „negyvu kodu“, begalinių ciklų, netinkamų šakojimų ir potencialių vykdymo klaidų, įskaitant „naudojimo prieš inicijavimą“ klaidas, identifikavimą [16]. Duomenų srauto analizės metodai, įskaitant konstantų sklaidą ir pseudonimų analizę, dažnai įgyvendinami kaip sluoksnis ant CFG, siekiant nustatyti kintamųjų vertes skirtinguose keliuose.



6 pav. Valdymo srauto grafas (CFG)

1.8.5. Duomenų srauto analizė

Duomenų srauto analizė yra išplėstinis statinės analizės etapas, grindžiamas valdymo srauto grafiku (CFG), skirtas stebėti verčių kūrimą, perdavimą ir transformavimą per kintamuosius bei atminties

vietas. Naudojant tokius atvaizdavimus kaip verčių srauto grafikai ir retieji duomenų priklausomybių grafikai, šiuolaikinės metodikos leidžia išsiaiškinti, kaip kintamųjų apibrėžimai naudojami skirtinguose programos keliuose [17], [18]. Ši galimybė leidžia identifikuoti neinicijuotus skaitymus, nepakankamai išnaudojamus skaičiavimus ir saugumo pažeidžiamumus, kylančius dėl netinkamo duomenų plitimo.

Naujausi tyrimai padidino duomenų srauto analizės tikslumą ir mastelį, integruodami konteksto, srauto ir lauko jautrumą į integruotas sistemas, pavyzdžiui, sinchronizuotas „pushdown“ sistemas arba sujungtas kelių jautrias priklausomybės analizes [18], [19]. Srityse, kurioms būdingas aukštas saugumo jautrumo lygis, pvz., išmaniosiose sutartyse, paplitusia praktika tapo praturtintų grafų, koduojančių tiek valdymo srauto, tiek duomenų srauto semantiką, naudojimas. Šie grafai naudojami kaip įvestys mokymosi pagrįstiems detektoriams – strategijai, kuri, kaip įrodyta, padidina pažeidžiamumų aptikimo tikslumą [20], [21].

1.8.6. Simbolių vykdymu grindžiama analizė

Simbolinis vykdymas – tai papildomas statinės analizės metodas, leidžiantis įvertinti programos elgseną, kai įvestys traktuojamos kaip simbolinės reikšmės, o ne konkrečios reikšmės. Simboliniu būdu vykdant programą, kiekviena šaka sukuria kelio sąlygą – tai loginis apribojimas, apibūdinantis įvestis, reikalingas tam, kad būtų galima eiti tuo keliu, bei simbolinę būseną, atspindinčią kintamųjų reikšmes. Šių apribojimų išsprendimas padeda nustatyti, ar galima pasiekti klaidų būsenas, teiginių pažeidimus ar saugumo savybes, pvz., pastovaus laiko vykdymą ar slaptų duomenų ištrynimą [22].

Šiuolaikiniai tyrimai lėmė reikšmingą simbolinio vykdymo mastelio ir tikslumo pažangą. Ši pažanga buvo pasiekta plėtojant tokias technikas kaip reliacinis simbolinis vykdymas dvejetainiu lygiu, terminų perrašymas ir normalizavimas apribojimų supaprastinimui. Be to, buvo sukurti specializuoti metodai informacijos srauto savybėms analizuoti [22]. Dėl šios savybės simbolinis vykdymas tampa ypač veiksminga priemone tais atvejais, kai šaltinio kodas nėra prieinamas arba kai kompiliatoriai gali sukelti pažeidžiamumą. Tokių scenarijų pavyzdžiai apima kriptografinių dvejetainių failų arba „Ethereum“ baitkodo analizę [23].

1.8.7. Šablonais grindžiama analizė

Šablonu pagrįsta analizė – tai metodas, skirtas nustatyti pasikartojančias struktūras ar antišablonus, susijusius su defektais ar pažeidžiamumais. Atliekant šią analizę, šablonams nustatyti naudojami taisyklių rinkiniai arba mašininio mokymosi modeliai. Programinės įrangos saugumo kontekste naudojami įrankiai, skirti nustatyti potencialias pažeidžiamąsias vietas, ieškodami žinomų nesaugių funkcijų, buferio perpildymų, įterpimo modelių ir architektūros problemų. Šie įrankiai naudoja iš anksto apibrėžtus šablonus, su kuriais lygina kodo fragmentus, taip nustatydami bet kokius neatitikimus, kurie gali rodyti saugumo pažeidimą [14], [24] [25]. Įrodyta, kad šablonais pagrįsti metodai yra labai veiksmingi greitam pažeidžiamumų aptikimui, tačiau jie gali susidurti su sunkumais sprendžiant naujas grėsmes, jei nėra nuolat atnaujinami. Naujausi šios srities pasiekimai parodė, kad dirbtinio intelekto ir mašininio mokymosi metodų integravimas reikšmingai pagerino sistemų prisitaikymo gebėjimus, sudarydamas sąlygas joms atpažinti sudėtingus modelius, kurie peržengia aiškių taisyklių rinkinių ribas [14], [25].

1.9. Daugiaagentės ir konsensuso sistemos

Programavimo srityje daugiaagentės ir konsensuso sistemos yra skirtos tirti, kaip keli programiniai agentai koordinuoja savo veiksmus, derasi ir susitaria dėl bendrų sprendimų ar būsenų paskirstytose aplinkose. Šie mechanizmai yra itin svarbūs šiuolaikinių programinės įrangos sistemų patikimumui, plečiamumui ir pasitikėjimui užtikrinti.

Programinės įrangos inžinerijoje daugiaagentės sistemos (MAS) apima autonominius programinės įrangos objektus, kurie bendradarbiauja vykdydami kūrimo, testavimo ar priežiūros užduotis per visą programinės įrangos gyvavimo ciklą. Dideliais kalbos modeliais pagrįstos MAS naudojamos kodui generuoti, klaidoms taisyti ir reikalavimams analizuoti, o jų agentai specializuojasi tam tikrose funkcijose, pavyzdžiui, projektuotojo, programuotojo ar testuotojo, ir, bendradarbiaudami bei naudodamiesi struktūrizuotais protokolais, sprendžia sudėtingas programavimo užduotis. Tokios sistemos padidina patikimumą ir plečiamumą, paskirstydamos darbą ir užtikrindamos argumentavimo bei problemų sprendimo dubliavimą. Koordinavimo logika, įgyvendinama per orkestravimo sistemas ir bendros atminties struktūras, yra esminė siekiant užtikrinti nuoseklią pažangą ir išvengti prieštaringų kodo pakeitimų [26].

Daugumos balsavimo algoritmai plačiai naudojami kaip sprendimų priėmimo protokolai tarp programavimo agentų, siekiant pasirinkti labiausiai tikėtiną atsakymą, pataisą ar kodo pakeitimą. Daugiaagentėse diskusijose tarp LLM pagrįstų programuotojų daugumos balsavimas, palyginti su kitais protokolais, reikšmingai padidina užduočių tikslumą sprendimų priėmimo reikalaujančiose programinės įrangos užduotyse. Balsavimas dėl kelių nepriklausomai sukurtų kodo sprendimų ar paaiškinimų sumažina atskirų modelių šališkumą ir nesuderinamumą, veiksmingai sujungdamas agentus sprendimų priėmimo lygmenyje [27]. Semantinis arba pasitikėjimu svertinis balsavimas gali dar labiau patobulinti atranką, teikiant pirmenybę agentams ar kodo variantams, kurių patikimumas įvertintas kaip didesnis [27], [28].

Konsensuso pagrindu priimami sprendimai į programavimą orientuotose daugiaagentėse sistemose (MAS) apima ne tik paprastą balsavimą, bet ir protokolus, reikalaujančius tvirtesnio susitarimo, pavyzdžiui, vienbalsiškumo ar Bizantijos gedimų tolerancijos konsensuso. Tyrimai, kuriuose lyginami sprendimų priėmimo protokolai daugiaagentėse diskusijose, rodo, kad vienbalsiškumo tipo konsensusas gali pagerinti našumą atliekant žinių reikalaujančias užduotis, tačiau kartu gali padidinti vėlavimą ir jautrumą išskirtiniams agentams [27]. Žinių valdymo MAS sistemose PBFT pagrįstas konsensusas naudojamas susitarimams dėl atminties valymo ar būsenos atnaujinimų, toleruojant ribotą dalį piktavališkų ar gedimų turinčių programinės įrangos agentų ir kartu išlaikant teisingumo garantijas [28]. Hierarchinės konsensuso struktūros sumažina komunikacijos sudėtingumą, sujungdamos vietinių klasterių sprendimus prieš visuotinį arbitražą, taip užtikrindamos plečiamumą dideliuose į programavimą orientuotuose agentų tinkluose.

Apskritai konsensuso sistemos į programavimą orientuotose daugiaagentėse architektūrose, derinančios daugumos balsavimą, griežtesnius konsensuso protokolus ir hierarchinį koordinavimą, reikšmingai padidina patikimumą ir pasitikėjimą. Jos sumažina atskirų agentų klaidų poveikį, padeda susidoroti su priešišku ar netinkamu elgesiu ir užtikrina atkuriamus bei patikrinamus sprendimų priėmimo procesus, skirtus kodo ir žinių atnaujinimams. Todėl daugiaagentės ir konsensuso strategijos tampa pagrindiniais mechanizmais kuriant plečiamas, patikimas ir autonomiškas programinės įrangos inžinerijos sistemas [26], [27], [28].

1.10. Problemos su kuriomis susiduria programinio kodo analizės įrankiai

Šiuolaikinė programinės įrangos inžinerija, siekdama nustatyti ir sumažinti kodo problemas, remiasi automatizuotomis metodikomis, tokiomis kaip statinė analizė, dinaminė analizė ir trūkumų prognozavimas. Nepaisant to, kad šie metodai yra plačiai taikomi, jie susiduria su dideliais apribojimais tikslumo, plečiamumo ir prisitaikymo prie sudėtingų ar besikeičiančių kodo bazių atžvilgiu.

Statinės analizės įrankiai tikrina šaltinio kodą jo nevykdydami, siekdami anksti, dar kūrimo proceso pradžioje, aptikti galimas klaidas ir pažeidžiamumą. Tačiau nuolatinė problema yra jų polinkis generuoti daug klaidingų teigiamų rezultatų – įspėjimų, kurie neatitinka tikrųjų trūkumų. Ši problema kyla dėl konservatyvaus statinės analizės pobūdžio, kai siekiant užtikrinti, kad nebūtų praleista nė viena tikra klaida, dažnai pernelyg apytikriai įvertinami galimi vykdymo keliai. Dėl to kūrėjai turi rankiniu būdu patikrinti daugybę įspėjimų, o tai didina darbo krūvį ir gali sukelti įspėjimų nuovargį [29]. Naujausi tyrimai rodo, kad net pažangūs statiniai analizatoriai gali generuoti pernelyg daug klaidingų pavojaus signalų dėl netikslaus programos elgsenos modeliavimo ir riboto konteksto suvokimo [16], [30]. Pastangos sumažinti klaidingų teigiamų rezultatų skaičių naudojant mašininį mokymąsi ar hibridinius metodus yra daug žadančios, tačiau šios problemos visiškai dar neišsprendžia.

Didėjant programinės įrangos sistemų apimčiai ir sudėtingumui, statinės analizės įrankiai susiduria su apimties ir tikslumo problemomis. Didelės kodo bazės su sudėtingais valdymo srautais, plačiu išorinių bibliotekų naudojimu ar dinaminėmis funkcijomis, pavyzdžiui, refleksija, kelia didelių iššūkių tradiciniams statiniams analizatoriams. Susidūrusios su sudėtingomis programų struktūromis, šios priemonės gali nesugebėti išanalizuoti visų reikšmingų kodo kelių arba gali pateikti neišsamius rezultatus [16], [30]. Be to, poreikis pritaikyti ir adaptuoti įrankius prie konkrečių programavimo kalbų ar sistemų dar labiau riboja jų pritaikomumą heterogeninėse aplinkose.

Tradicinė statinė analizė remiasi sintaksiniais modeliais ir iš anksto nustatytomis taisyklėmis, o tai riboja jos gebėjimą nustatyti gilius semantinius ryšius kodo viduje. Dėl šio apribojimo gali būti nepastebimos klaidos, kurių aptikimui reikalingas programos tikslo supratimas arba aukštesnio lygio abstrakcijos [16], [30]. Nors pastaruoju metu integruojami dideli kalbos modeliai (LLM) pagerino semantinio samprotavimo gebėjimus, dabartiniai jų įgyvendinimai vis dar susiduria su sunkumais suprantant ilgalaikes priklausomybes ir sudėtingą logiką, būdingą realiems projektams [16].

Dinaminės analizės metodai leidžia vykdyti programas su konkrečiais įvesties duomenimis, siekiant stebėti elgseną vykdymo metu ir aptikti trūkumus, kurie statiniu būdu gali būti nepastebimi. Tačiau jų veiksmingumas iš esmės ribojamas turimų testavimo atvejų kokybės ir aprėpties. Pasiekti išsamią kodo aprėptį yra sudėtinga – tam tikri vykdymo keliai gali likti neišbandyti dėl nepakankamos įvesties įvairovės ar aplinkos apribojimų. Be to, dinaminė analizė negali garantuoti paslėptų klaidų, kurios pasireiškia tik retomis sąlygomis ar esant konkrečioms konfigūracijoms, aptikimo.

Defektų prognozavimo modeliai naudoja istorinius duomenis, pavyzdžiui, ankstesnes klaidų ataskaitas ar kodo metrikas, siekdami numatyti būsimus defektus. Šie modeliai dažnai labai priklauso nuo ankstesnių kūrimo ciklų paženklintų duomenų rinkinių prieinamumo ir kokybės [29]. Todėl jų prognozavimo tikslumas sumažėja, kai jie taikomi naujiems projektams, kuriems trūksta pakankamai istorinių duomenų, arba perkeliant modelius iš vienos srities į kitą dėl duomenų rinkinių

specifiškumo. Be to, daugelis defektų prognozavimo metodų traktuoja visas klaidas vienodai, tinkamai neįvertindami jų sunkumo ar nesuteikdami pirmenybės kritinėms problemoms [29].

Apibendrinant galima teigti, kad nors statinė analizė užtikrina plačią aprėptį, jai būdingas didelis klaidingų teigiamų rezultatų skaičius ir ribotas semantinis gylis, ypač sudėtingose kodo bazėse, o dinaminė analizė susiduria su apribojimais dėl neišsamios testavimo aprėpties. Defektų prognozavimo metodai labai priklauso nuo istorinių duomenų ir dažnai negali būti taikomi įvairiems projektams. Šių trūkumų šalinimas išlieka aktyvia mokslinių tyrimų sritimi, nes programinės įrangos sistemos toliau auga ir tampa vis sudėtingesnės.

1.11. Analizės apibendrinimas ir išvados

Šioje literatūros apžvalgoje aptarti naujausi tyrimai rodo, kad programinės įrangos kokybės ir saugumo analizė vis dažniau vykdoma keliose tarpusavyje susijusiose srityse, įskaitant statinę kodo analizę, dinaminę kodo analizę, hibridinę analizę, automatizuotą programų taisymą ir didelių kalbos modelių pagalba atliekamus kūrimo procesus. Apžvalgoje statinė analizė pateikiama kaip plečiamas ir ankstyvojo etapo metodas, kurio pagrindu gaunami formalūs atvaizdai, tokie kaip abstrakčiosios sintaksės medžiai, valdymo srauto grafikai ir vertės srauto modeliai, tačiau taip pat pabrėžiama, kad šis privalumas yra neatsiejamas nuo konservatyvių apytikrių vertinimų, dėl kurių dažnai gaunami dideli klaidingų teigiamų rezultatų rodikliai. Dinaminė analizė, priešingai, pateikiama kaip į vykdymo laiką orientuota perspektyva, leidžianti atskleisti nuo vykdymo priklausančias klaidas, su atmintimi susijusias klaidas ir aplinkai būdingą elgseną, kuri lieka nematoma taikant grynai statinį samprotavimą. Literatūroje taip pat nurodoma, kad hibridiniai metodai bando suderinti šiuos kompromisus, leisdami statiniams įrodymams teikti pirmenybę vykdymo metu atliekamiems testams, o vykdymo įrodymus naudodami statiniu būdu nustatytų rezultatų patvirtinimui arba atmetimui.

Tuo pačiu metu apžvelgti tyrimai apie automatizuotą programų taisymą ir LLM pagrįstą kodo taisymą rodo, kad šiuolaikiniai modeliai gali sėkmingai išspręsti reikšmingą dalį pasikartojančių ir struktūriškai paprastų trūkumų, ypač pasikartojančių klaidų kategorijose ir ribotose testavimo aplinkose. Tačiau tie patys tyrimai taip pat rodo, kad šios sistemos išlieka mažiau patikimos, susidūrusios su semantiškai sudėtingais trūkumais, ilgalaikėmis priklausomybėmis, srities specifinių API netinkamu naudojimu arba sudėtingomis valdymo srauto sąveikomis, kurioms reikia gilesnio konteksto supratimo. Šis apribojimas yra ypač reikšmingas, nes apžvalgoje taip pat teigiama, kad automatizavimas didina sukuriamo ir modifikuojamo kodo apimtį greičiau, nei užtikrina semantinį teisingumą, todėl išlieka nemaža likutinė programinės įrangos kokybės rizika. Šiame kontekste esami tyrimai rodo, kad dabartiniai pažangieji įrankiai teikia naudingą pagalbą kūrėjams, tačiau jie nepilnai išsprendžia pagrindinių teisingumo užtikrinimo, interpretuojamumo ir patikimo defektų patvirtinimo iššūkių.

Tuo pačiu metu skyriuje apie daugiaagentines ir konsensuso sistemas parodoma, kad koordinuotos agentų architektūros, naudojančios daugumos balsavimą, svertinį atrankos metodą, vienbalsiškumu grindžiamą susitarimą ir griežtesnius konsensuso protokolus, gali padidinti patikimumą, sumažinti atskirų modelių šališkumą ir pagerinti sprendimų atkuriamumą programavimo užduotyse. Vis dėlto literatūros apžvalga dabartine forma nenurodo mokslinių tyrimų krypties, kuri konsensusu pagrįstą LLM bendradarbiavimą taikytų konkrečiai kodo analizės veiklai, pavyzdžiui, įspėjimų interpretavimui, analizatoriaus rezultatų suderinimui, defektų reitingavimui ar įtartinų rezultatų semantiniam patvirtinimui prieš taisymą. Todėl tarp esamų tyrimų apie kodo analizės metodus ir

esamų tyrimų apie daugelio agentų konsensusą išlieka aiški akademinė spraga, o tai patvirtina teiginį, kad konsensuso pagrįsti LLM įrankiai kodo analizei yra aktuali ir pagrįsta tyrimų kryptis, o ne išsamiai ištirta sritis.

Apibendrinant analizės metu surinktą informaciją galima teigti, kad:

1. Naudojant kelis metodus ar kelis didelius kalbos modelius (LLM) būtų galima patobulinti statinę kodo analizę, derinant skirtingas samprotavimo strategijas, o tai galėtų sumažinti priklausomybę nuo vieno modelio šališkumo, ribotumų ar klaidų modelių.
2. Daugiamodelis arba konsensuso pagrįstas požiūris galėtų sustiprinti rezultatų patikimumą, lyginant to paties įspėjimo interpretacijas, patvirtinant įtartinus radinius keliuose agentuose ir atmetant silpnesnes arba nepagrįstas išvadas, taip potencialiai sumažinant klaidingų teigiamų rezultatų skaičių.
3. Toks požiūris taip pat galėtų pagerinti semantinį supratimą sudėtingais atvejais, nes skirtingi LLM arba analizės metodai gali užfiksuoti skirtingus programos logikos, konteksto ir ketinimų aspektus, todėl galutinis vertinimas taptų išsamesnis nei tradicinė statinė analizė, atliekama naudojant vieną įrankį.
4. Remiantis atlikta literatūros analize, statinės kodo analizės ribotumus galima spręsti tobulinant esamus analizės metodus ir juos papildant konsensusu pagrįstu kelių LLM vertinimu, kuris leistų patikimiau interpretuoti diagnostinius pranešimus, patvirtinti siūlomų pataisymų tinkamumą ir sumažinti priklausomybę nuo vieno įrankio ar modelio rezultatų.

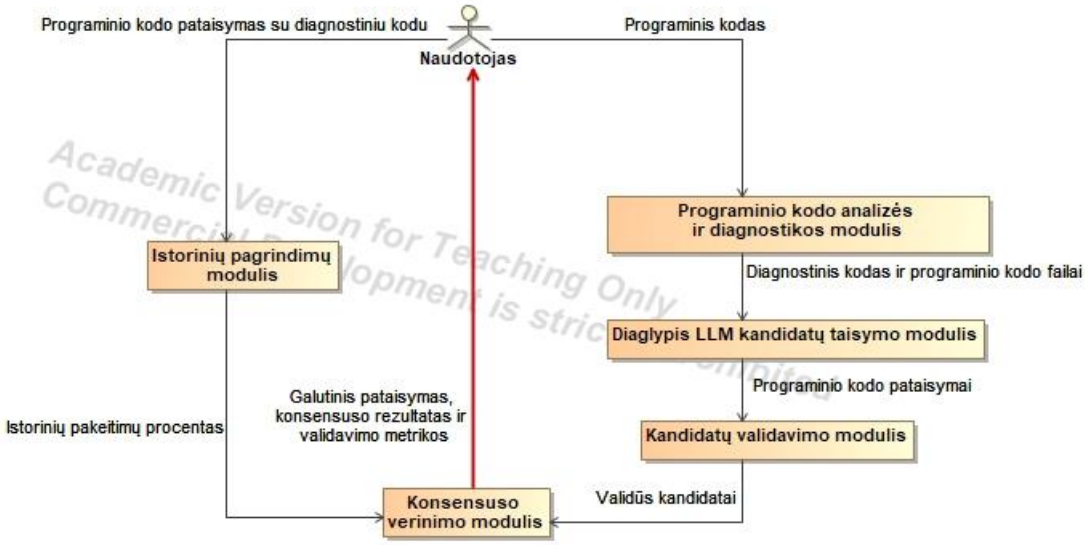
2. Statinės kodo analizės tobulinimas naudojant konsensuso metodo projektas

Šiame skyriuje pateikiama siūlomo metodo, skirto statinei kodo analizei tobulinti naudojant daugelio didelių kalbos modelių konsensuso principu pagrįstą metodiką ir struktūra. Skyriuje aprašoma sistemos architektūra, istorinių duomenų pagrindimo modelis, programinio kodo saugyklos analizė, diagnostinių duomenų gavimas bei galimų pataisų generavimas ir patvirtinimas naudojant kelis didelius kalbos modelius. Skyriuje taip pat paaiškinamas konsensuso principu pagrįstas procesas, taikomas galutinių kodo pataisų reitingavimui ir atrankai. Skyriuje apibrėžiami pagrindiniai metodo komponentai ir paaiškinama, kaip jie sąveikauja.

2.1. Bendra sistemos architektūra

Siūlomas metodas sukurtas kaip daugiapakopis procesas, skirtas automatizuotai analizuoti ir taisyti šaltinio kodo klaidas, taikant konsensuso principu pagrįstą didelių kalbos modelių metodiką. Pagrindinis šio proceso tikslas – sujungti istorinius taisymų duomenis, diagnostikos išgavimą saugyklos lygmeniu, daugelio modelių kandidatinių pataisų generavimą ir patvirtinimu pagrįstą reitingavimą į vieningą procesą, skirtą tinkamiausiam kodo pataisymui pasirinkti. Užuot rėmusis vieno modelio atsakymu, sistemoje taisymų generavimas traktuojamas kaip lyginamasis procesas. Šiame procese kelios nepriklausomai sukurtos pataisos vertinamos atsižvelgiant tiek į istorines taisymų charakteristikas, tiek į objektyvius patvirtinimo rezultatus.

Sistema sudaryta iš penkių tarpusavyje susijusių komponentų. Pirmasis komponentas yra istorinių duomenų modulis, antrasis – saugyklos analizės ir diagnostikos išgavimo modulis, trečiasis – daugialypio LLM kandidatinių pataisų generavimo modulis, ketvirtasis – kandidatų patvirtinimo modulis, o penktasis – konsensuso ir reitingavimo modulis. Istorinių duomenų modulis nustato numatomų kodo pakeitimų bazinį atvaizdą, analizuodamas anksčiau pastebėtas klaidingų ir ištaisytų kodo fragmentų poras, susijusias su konkrečiu diagnostikos ar klaidų tipu. Saugyklos analizės ir diagnostikos išgavimo modulis parengia tikslinę taisymo užduotį: apdoroja naudotojo pateiktą saugyklą, identifikuoja kompiliatoriaus ar kūrimo metu nustatytus diagnostinius duomenis ir išgauna atitinkamą šaltinio kodo regioną kartu su susijusiais diagnostiniais metaduomenimis. Bendra sistemos architektūra pateikta 7 pav.



7 pav. Bendra prototipo architektūra

Kandidatinių pataisų generavimo modulis pateikia tą pačią taisymo užduotį keliems dideliems kalbos modeliams arba keliems taisymo agentams, sukonfigūruotiems taikant skirtingas užklausų strategijas. Kiekvienas modelis sukuria nepriklausomą kandidatinę pataisą, skirtą aptiktai problemai išspręsti. Šis lygiagretus generavimo procesas yra viena iš pagrindinių sistemos savybių, nes leidžia surinkti įvairias taisymo alternatyvas, o ne priklausyti nuo vieno sugeneruoto sprendimo. Gautos kandidatinės pataisos vėliau perduodamos į patvirtinimo etapą, kuriame kiekviena pataisa taikoma ir vertinama atskirai pagal praktinį taisymo veiksmingumą.

Po patvirtinimo likusios kandidatinės pataisos apdorojamos konsensuso ir reitingavimo modulyje. Šis komponentas atlieka sėkmingų arba iš dalies sėkmingų pataisymų lyginamąją analizę su istoriniu duomenų etalonu ir, jei taikytina, tarpusavyje. Šio etapo tikslas yra ne tik nustatyti labiausiai panašią pataisą, bet ir įvertinti, kuris kandidatas geriausiai atitinka bendrus kriterijus: pataisymo sėkmę, atitiktį istoriniam pagrindui ir kandidatų tarpusavio sutapimą. Todėl bendra architektūra apibrėžia nuoseklų perėjimą nuo istorinių žinių prie konkrečios programinio kodo diagnozės, nuo kandidatinių pataisų generavimo prie empirinio patvirtinimo ir nuo patvirtinimo prie konsensusu pagrįsto galutinio atrinkimo. Šis projektas atitinka pagrindinę tyrimo prielaidą, kad automatizuoto kodo taisymo veiksmingumą galima padidinti kolektyviai vertinant didelių kalbos modelių rezultatus ir atsižvelgiant į istoriniu pagrindu paremtus taisymo bruožus.

2.2. Istorinio pagrindimo modelis

Istorinis pagrindimo modelis sukurtas siekiant suteikti atskaitos tašką, pagal kurį būtų galima įvertinti automatinio taisymo metu atliekamų kodo pakeitimų mastą. Siūlomoje sistemoje pagrindimas apibrėžiamas kaip laipsnis, kuriuo ištaisyta šaltinio kodo versija skiriasi nuo klaidingos pradinės versijos, atsižvelgiant į konkretų diagnostikos atvejį. Ši sąvoka grindžiama prielaida, kad istorinės klaidų taisymo poros suteikia naudingos informacijos apie tai, kiek paprastai pasikeičia kodas, kai išsprendžiama konkreti problema. Todėl pagrindimas veikia kaip taisymui orientuota bazinė linija, kuri vėliau gali būti naudojama vertinant, ar LLM sukurtos pataisos išlieka priimtiname pakeitimų intervale.

Pagrindimo procesui reikalingi trys pagrindiniai įvesties duomenys: su problema susijęs diagnostinis arba klaidos kodas, šaltinio failas, kuriame yra klaidingas įgyvendinimas, ir to paties failo pataisyta versija po sėkmingo žmogaus atlikto taisymo. Atliekamas klaidingos ir pataisytos versijų palyginimas, siekiant nustatyti šaltinio kodo dalį, kuri buvo pakeista taisymo proceso metu. Esamoje sistemoje ši pakeitimų dalis žymima kaip „grounding“ procentas, nurodantis istorinio pataisymo santykinį mastą. Pagrindinė hipotezė yra ta, kad veiksmingi analogiškų diagnostinių problemų sprendimai gali pasižymėti panašiu pakeitimų mastu, nepaisant konkrečių kodo pakeitimų turinio ar struktūrinių savybių skirtumų.

Siekiant gauti pagrindinę vertę, sistemoje atliekamas simbolinis originalios ir pataisytos failo versijų palyginimas. Šaltinio kodas pateikiamas kaip simbolių arba žodžių seka. Skirtumas tarp dviejų versijų apskaičiuojamas kaip pakeistų elementų dalis, palyginti su bendru elementų skaičiumi originalioje kodo versijoje. Šis procesas suteikia normalizuotą pakeitimų rodiklį, kuris gali būti naudojamas lyginant įvairius galimus pataisymus. Sistema išsiskiria tuo, kad taisymo mastui išreikšti naudoja procentais pagrįstą rodiklį, o ne absoliutų skaičių. Šis metodologinis sprendimas grindžiamas siekiu išvengti netyčinio palankumo labai mažiems arba labai dideliems failams, kuris galėtų atsirasti, jei vienintelis vertinimo rodiklis būtų absoliutus pakeistų elementų skaičius.

Istorinio pagrindimo procentinė reikšmė apskaičiuojama lyginant klaidingą ir pataisytą to paties failo versiją. Ši reikšmė parodo, kokia pradinio kodo dalis buvo pakeista žmogaus atlikto taisymo metu::

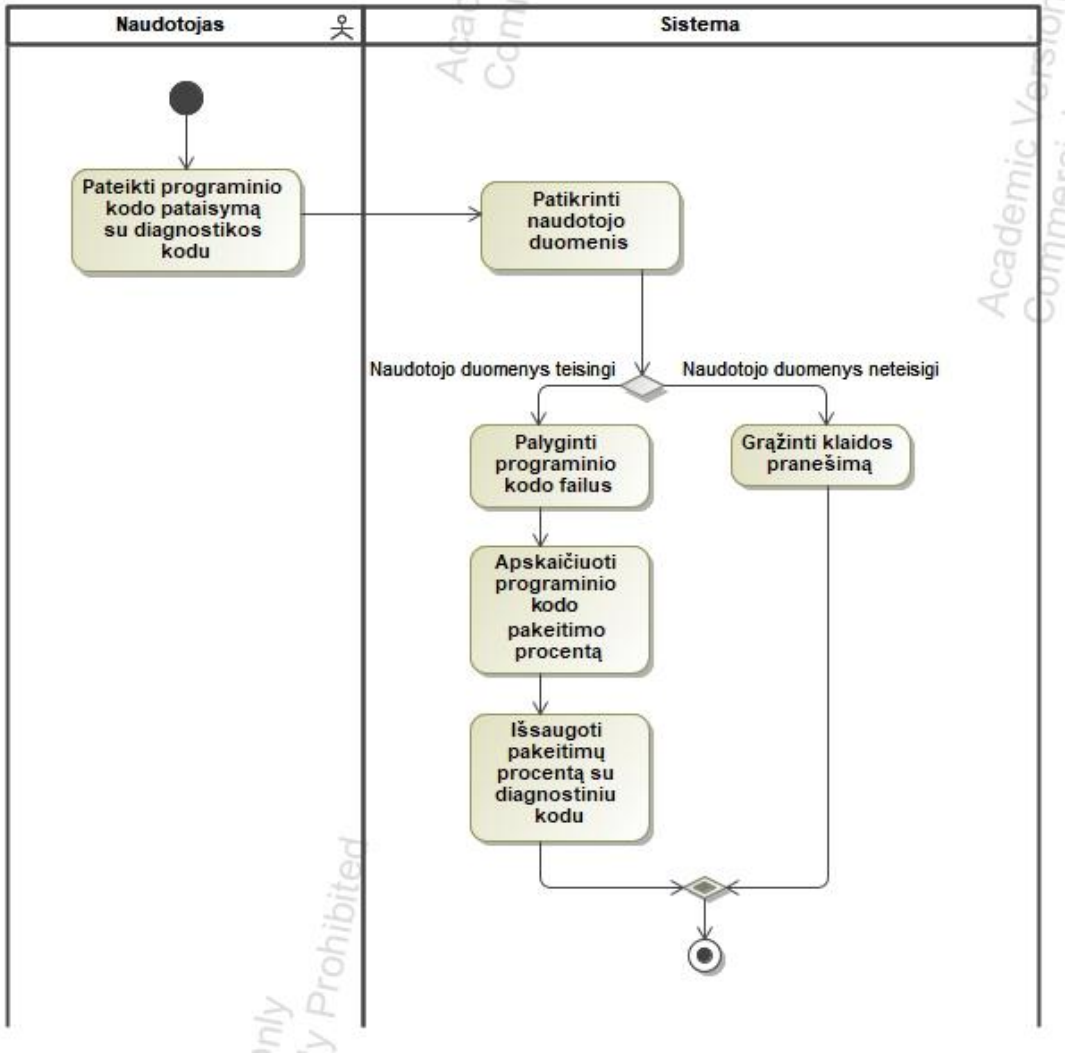
$$G_d = \frac{C_{changed}}{C_{total}} \times 100 \quad (2)$$

Kai tam pačiam diagnostiniam kodui yra pateikiami keli istoriniai taisymo pavyzdžiai, naudojama vidutinė pagrindimo reikšmė:

$$\overline{G_d} = \frac{1}{n}\sum_{i=1}^{n} G_{d,i} \quad (3)$$

Gautas pagrindimo procentas nėra skirtas pataisos semantiniam teisingumui apibūdinti, jis veikiau nusako jos struktūrinį mastą. Kitaip tariant, tikslas yra ne nustatyti, ar kodo pakeitimas buvo logiškai optimalus, o įvertinti, kokiu mastu kodas buvo modifikuotas. Todėl pagrindimas nenaudojamas kaip nepriklausomas kriterijus galutiniam sprendimui pasirinkti. Vietoj to jis veikia kaip pagalbinė nuoroda, papildanti vėlesnius patvirtinimo ir konsensuso etapus. Tais atvejais, kai keli LLM sugeneruoti pataisymai veiksmingai sprendžia tą pačią problemą, pagrindimas gali padėti atskirti kandidatus, kurių modifikavimo mastas labiau atitinka istoriškai stebėtus taisymo elgsenos modelius.

Istorinis pagrindimo modelis suteikia sistemą, kurioje taisymo kontekstas kyla iš ankstesnių pavyzdžių, o ne vien iš dabartinio programinio kodo. Jo funkcija – įtvirtinti pataisų vertinimo procesą empiriškai stebėtuose pakeitimų modeliuose ir sumažinti tikimybę, kad bus pasirinkti pernelyg plačiai apimantys arba nerealistiškai minimalūs sprendimai. Bendroje architektūros struktūroje „grounding“ veikia kaip tarpinis interpretacinis sluoksnis tarp istorinių taisymo duomenų ir naujai sukurtų galimų pataisų. Šis metodas skatina sistemingesnį ir pagrįstesnį požiūrį į automatizuotą kodo taisymą. Istorinio pagrindimo modulio darbo eiga pateikta 8 pav.



8 pav. Istorinio pagrindimo modulio darbo eiga

2.3. Repozitorijos analizė ir diagnostikos gavimas

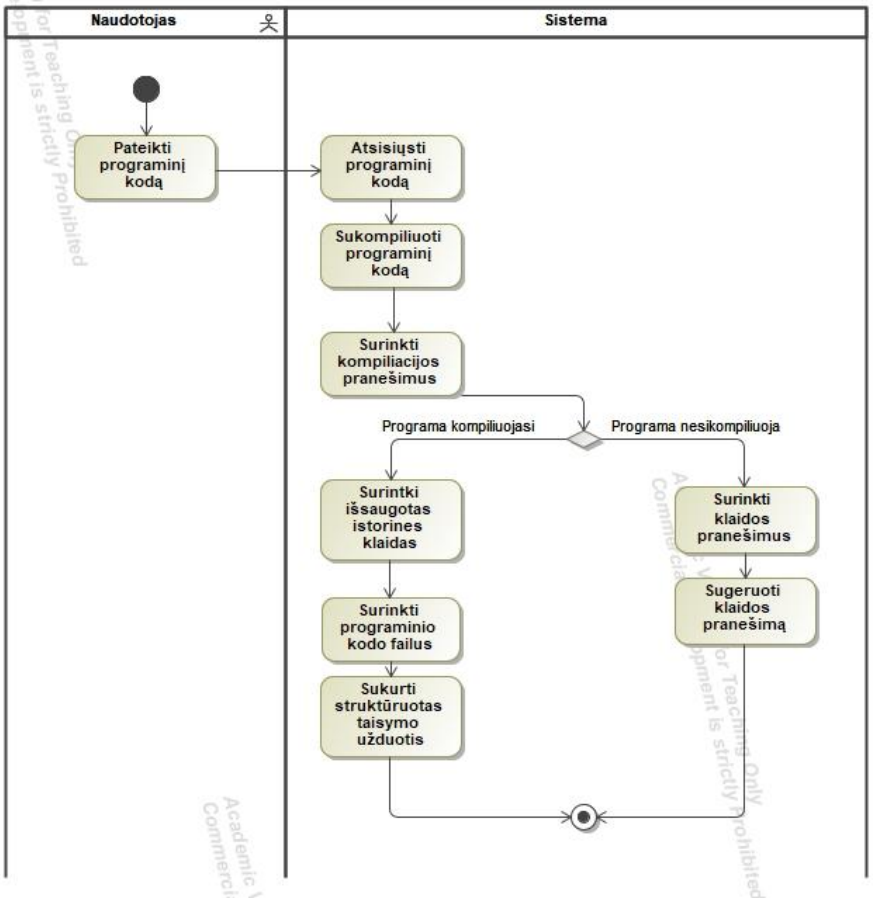
Analizės ir diagnostikos išgavimo etapas yra skirtas naudotojo pateiktai programinei įrangai transformuoti į struktūrizuotą taisymo užduotį, tinkamą automatizuotam apdorojimui. Nors istorinis pagrindimo modelis teikia informaciją apie anksčiau pastebėtą taisymo mastą, šiame etape dėmesys sutelkiamas į konkretaus problemos atvejo identifikavimą tikslinėje kodo bazėje. Šio proceso tikslas – nustatyti konkrečią diagnostinę problemą, jos vietą saugykloje ir tikslų šaltinio kodo segmentą, kuris turėtų būti pateiktas dideliems kalbos modeliams taisymo generavimo tikslais.

Procesas pradedamas, kai naudotojas pateikia programinį kodą analizei. Tuomet sistema parengia programinį kodą kontroliuojamoje vykdymo aplinkoje ir inicijuoja kompiliacijos procedūrą, naudodama atitinkamą projekto konfigūraciją bei priklausomybes. Šiame etape sistema sukonfigūruojama fiksuoti įvairius susijusius duomenis, įskaitant kompiliatoriaus pranešimus ir kūrimo įspėjimus. Ši informacija bendrai vadinama diagnostika. Diagnostika yra pagrindinis informacijos šaltinis taisymo tikslams nustatyti, nes ji nurodo tiek problemos buvimą, tiek jos techninę klasifikaciją. Remdamasi kompiliacijos metu gauta diagnostika, sistema taisymo užduotį grindžia stebimomis programinės įrangos kokybės problemomis, o ne vien rankiniu būdu atrinktais pavyzdžiais.

Surinkus kompiliacijos išvestį, sistema analizuoja diagnostinę informaciją, siekdama išgauti su kiekviena problema susijusius metaduomenis. Šie metaduomenys gali apimti diagnostinį kodą, tekstinį pranešimą, failo kelią ir, jei įmanoma, konkrečią eilutę ar kodo sritį, susijusią su įspėjimu arba klaida. Išgauta informacija leidžia sistemai nustatyti ryšį tarp praneštos problemos ir konkrečios vietos saugykloje. Šis ryšys yra būtinas, nes vėlesniame taisymų generavimo etape reikalingas ne tik klaidos aprašymas, bet ir atitinkamas šaltinio kodo kontekstas, kuriame problema pasireiškia.

Nustačius atitinkamą diagnostiką, sistema izoliuoja kodo artefaktą, kuris bus naudojamas kaip įvestis taisymų generavimui. Šio artefakto sudėtis priklauso nuo pasirinktos įgyvendinimo strategijos. Jis gali apimti visą šaltinio failą, lokalizuotą kodo fragmentą arba apribotą sritį, papildytą aplinkinėmis kontekstinėmis eilutėmis. Pasirinkta reprezentacija turi būti pakankamai informatyvi, kad LLM galėtų suprasti nurodytą problemą, tačiau kartu pakankamai apribota, kad būtų galima generuoti tikslius pataisymus. Šis balansas yra ypač svarbus, nes pernelyg plačios saugyklos lygmens kodo įvestys gali įtraukti nereikšmingą kontekstą, o pernelyg siauras išgavimas gali praleisti priklausomybes, būtinas tinkamam pataisymui.

Šio etapo rezultatas yra struktūrizuota taisymo užduotis, sujungianti diagnostinius metaduomenis su susijusiu šaltinio kodo kontekstu. Ši taisymo užduotis veikia kaip sąsaja tarp saugyklos lygmens analizės ir kelių LLM kandidatų generavimo. Saugyklos analizės ir diagnostikos išgavimo etapas yra svarbus siekiant užtikrinti, kad tolesnis taisymo procesas būtų grindžiamas aiškiai apibrėžtu ir atkartojamu problemos pavyzdžiu. Bendroje sistemoje jis nustato ryšį tarp neapdorotų projekto artefaktų ir automatizuoto taisymo darbo eigos, kuri vėliau taikoma kandidatinių pataisų generavimo ir vertinimo etapuose. Modulio veikimas pavaizduotas 9 pav.



9 pav. Programinio kodo kompiliacijos ir diagnostikos modulio darbo eiga

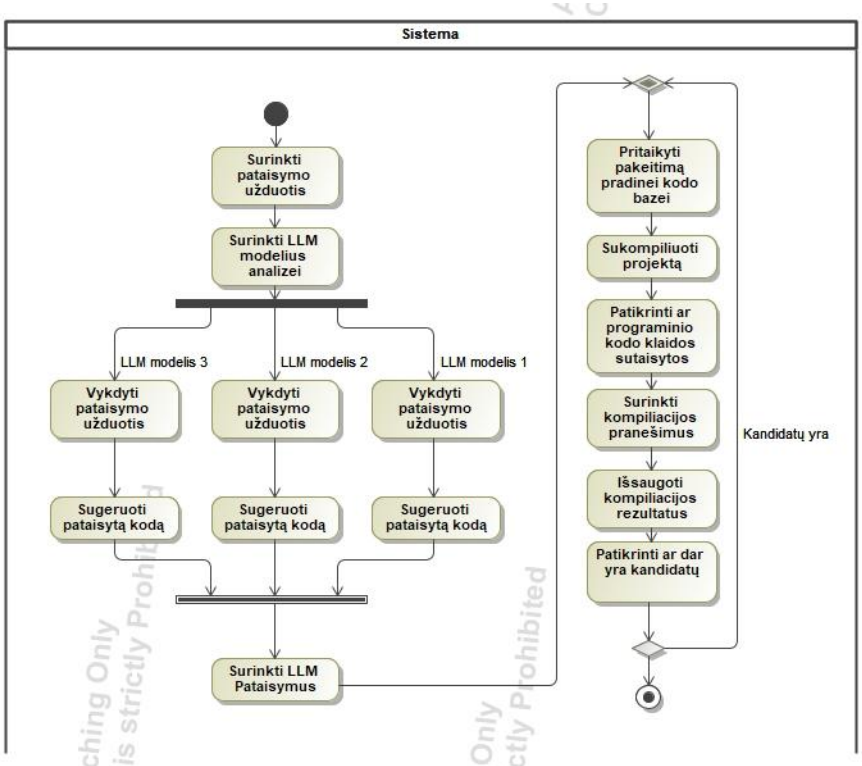
2.4. Kelių didelių kalbos modelių (LLM) siūlomų pataisų generavimas ir vertinimas

Šis sistemos etapas, kuris pavaizduotas 10 pav. skirtas siūlomoms kodo pataisoms generuoti ir vertinti, koordinuotai naudojant kelis didelius kalbos modelius. Šis metodas sujungia vienalaikį siūlomų pataisų generavimą su vėlesniu jų vertinimu, užtikrinant, kad sukurti sprendimų variantai būtų įvairūs ir funkcionaliai prasmingi. Šių procesų integravimas sistemoje padeda sumažinti priklausomybę nuo vieno rezultato, o pataisų generavimas traktuojamas kaip lyginamasis procesas, apimantis kelis nepriklausomai sukurtus variantus.

Procesas prasideda nuo struktūrizuotos taisymo užduoties pateikimo keliems dideliems kalbos modeliams arba taisymo agentams. Kiekvienas modelis gauna tą patį įvesties duomenų rinkinį, apimantį diagnostinį kodą, diagnostinį pranešimą ir susijusį šaltinio kodo kontekstą. Tada modeliai savarankiškai sukuria kandidatines pataisas nustatytai problemai išspręsti. Kadangi pataisos gali skirtis struktūra, apimtimi ar įgyvendinimo strategija, tam pačiam diagnostiniam atvejui gaunami keli alternatyvūs sprendimai.

Toliau kiekviena kandidatinė pataisa tikrinama taikant specialiai parengtą patvirtinimo procesą. Sistema pritaiko sukurtą modifikaciją originaliam programiniam kodui ir iš naujo kompiliuoja projektą kontroliuojamoje aplinkoje. Patvirtinimo metu vertinama, ar pataisytas kodas sėkmingai kompiliuojamas, ar tikslinė diagnostika išspręsta ir ar neatsirado naujų įspėjimų arba klaidų. Taip kandidatų kokybė vertinama pagal faktinį taisymo veiksmingumą, o ne vien tekstinį panašumą.

Kiekvienas kandidatas susiejamas su patvirtinimo rezultatų rinkiniu, apimančiu kompiliacijos sėkmingumą, diagnostikos išsprendimą ir galimus šalutinius poveikius. Šie rezultatai suteikia objektyvų pagrindą pataisoms palyginti ir leidžia sistemai atmesti arba žemesniu balu įvertinti nesėkmingus sprendimus. Taigi šiame etape gaunamas empiriškai patikrintų kandidatinių pataisų rinkinys, sudarantis pagrindą tolesniam konsensuso ir reitingavimo procesui.



10 pav. Daugialypio LLM kandidatų taisymo modulio darbo eiga

2.5. Konsensusas, reitingavimas ir galutinis pataisų pasirinkimas

Ši sistemos modulis (žr. 11 pav.) skirtas tinkamiausiai pataisai iš patvirtintų kandidatinių pataisų rinkinio nustatyti. Šis procesas sujungia konsensuso įvertinimą, pagrindimo suderinimą ir galutinį pasirinkimą į vieningą sprendimų priėmimo sistemą. Siūloma sistema nuo tradicinių metodikų skiriasi tuo, kad nėra remiamasi vieno modelio išvestimi ar paprastu balsavimo mechanizmu. Vietoj to taikoma vertinimo sistema, integruojanti empirinius patvirtinimo rezultatus ir jų atitikimą istoriškai stebėtoms pataisų charakteristikoms.

Procesas prasideda nuo kandidatinių pataisų, kurios sėkmingai praėjo patvirtinimo etapą, vertinimo. Kandidatai, kurių kompiliacija nepavyksta arba kurie neišsprendžia tikslinės diagnostikos, pašalinami arba jiems suteikiamas gerokai mažesnis prioritetas. Tai užtikrina, kad reitingavimo procesas pirmiausia būtų taikomas funkciškai reikšmingiems sprendimams. Vertinant likusius kandidatus, taikomas daugialypis požiūris, apimantis papildomus kriterijus jų santykinei kokybei ir patikimumui įvertinti.

Pagrindinis šio etapo elementas yra pagrindimo suderinimo vertinimas. Šiame etape kiekvienas patvirtintas kandidatas lyginamas su istorine pagrindimo baze, gauta iš ankstesnių klaidų taisymo porų. Šis palyginimas padeda įvertinti, kiek kandidato pakeitimų mastas atitinka istorinius to paties diagnostinio tipo taisymo modelius. Kandidatai, kurių pakeitimai patenka į tikėtiną intervalą, laikomi labiau atitinkančiais numatomą taisymo elgseną. Priešingai, kandidatams, kurie reikšmingai nukrypsta nuo šios bazės, gali būti taikomos baudos arba mažesnis prioritetas.

Kiekvieno LLM sugeneruoto kandidato nuokrypis nuo istoriškai pagrįsto taisymo masto apskaičiuojamas taip:

$$D_i = \left|G_i - \overline{G_d}\right| \tag{4}$$

Kandidatas laikomas atitinkančiu pagrindinę konsensuso sąlygą tik tada, kai pataisymas sėkmingai kompiliuojamas, pašalina tikslinę diagnostiką ir neviršija didžiausio leistino pakeitimų procento:

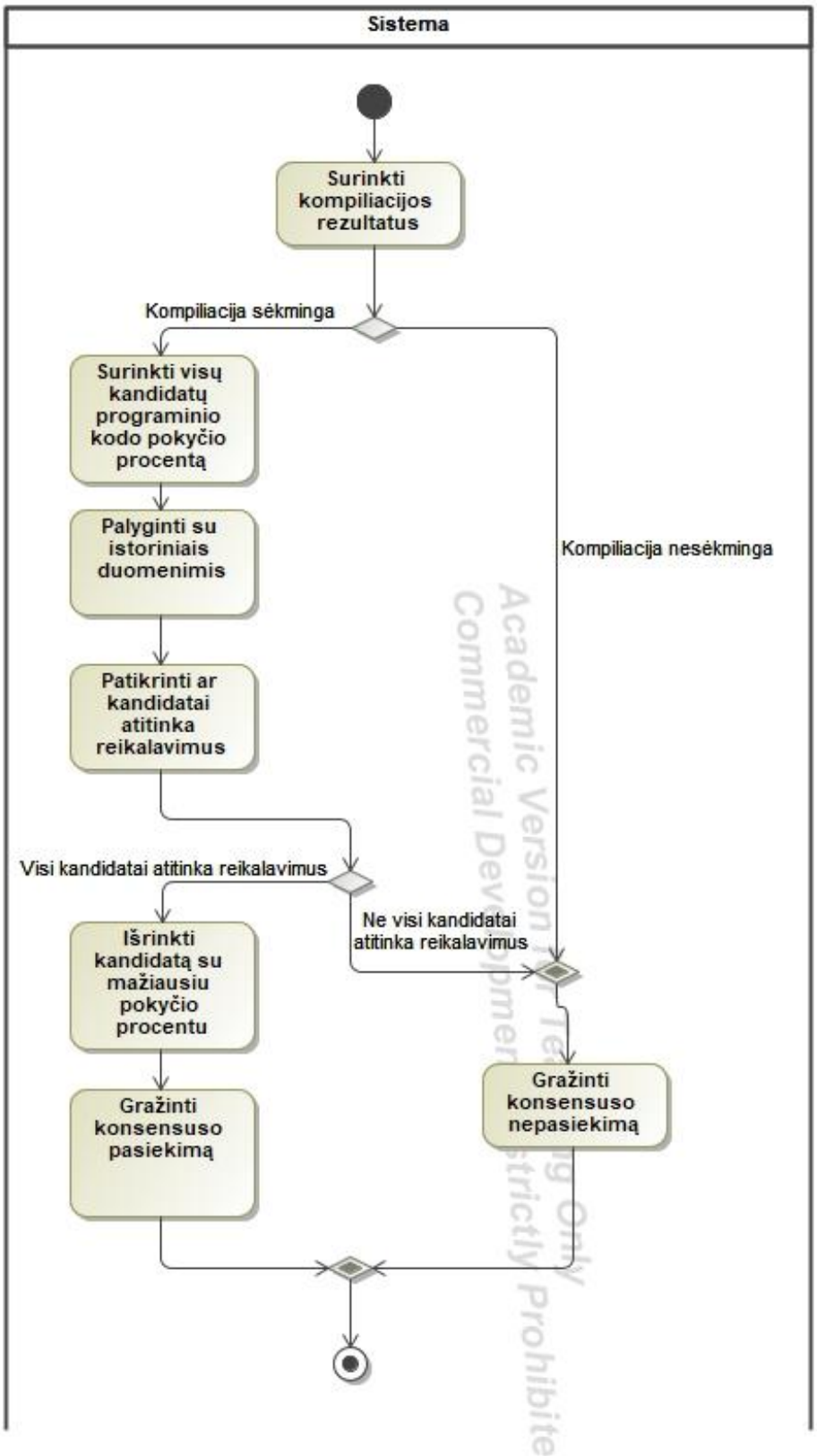
$$C_i = \mathbb{1}(B_i = 1 \wedge R_i = 1 \wedge G_i \leq G_{max}) \tag{5}$$

Čia $C_i = 1$ reiškia, kad kandidatas sėkmingai kompiliuojamas, pašalina diagnostinę klaidą ir neviršija didžiausio leistino pakeitimų procento. Priešingu atveju $C_i = 0$.

Be pagrindimo atitikimo, sistema apima kandidatinių pataisų konsensuso sąvoką. Užuot reikalavus identiškų rezultatų, konsensusas aiškinamas kaip nepriklausomai sukurtų kandidatinių pataisų charakteristikų sutapimas. Tai gali apimti panašumą modifikuotose kodo srityse, struktūriniuose modeliuose ar bendrose pataisymo strategijose. Kandidatai, rodantys aukštesnį sutapimo lygį su kitomis patvirtintomis pataisomis, laikomi patikimesniais, nes jie atspindi kelių nepriklausomų modelių sukurtus sutampančius sprendimus.

Galutinis kandidatų reitingas nustatomas sujungiant patvirtinimo rezultatus, pagrindimo suderinimą ir kandidatų tarpusavio sutapimą į vieningą balų skyrimo arba prioritetų nustatymo schemą. Nors tiksli šių komponentų svarba gali skirtis priklausomai nuo įgyvendinimo, patvirtinimo sėkmė išlieka pagrindiniu kriterijumi, po kurio vertinamas pagrindimo nuoseklumas ir konsensuso stiprumas.

Aukščiausią bendrą reitingą turintis kandidatas pasirenkamas kaip galutinis sistemos pateikiamas taisymo sprendimas.



11 pav. Konsensuso modulio darbo eiga

2.6. Konsensuso vertinimo kriterijai

Pradinis istorinių duomenų pagrindo procentinis dydis nebūtinai turi būti grindžiamas tik vienu konkrečiu istorinių duomenų tipu. Galima naudoti įvairias ankstesnių taisymų įrodymų formas, jei jos padeda įvertinti tipinį taisymo mastą konkrečiam diagnostiniam atvejui ar defektų modelio tipui. Tokie duomenys gali apimti originalių ir pataisytų failų poras, ankstesnius žmogaus atliktus pataisymus arba modeliais pagrįstus pavyzdžius, apibūdinančius pasikartojančias taisymo struktūras. Svarbiausias šios analizės elementas yra ne tiksli istorinių duomenų forma, o gebėjimas nustatyti palyginamą pokyčio dydžio vertę. Ši vertė vėliau naudojama antrojoje analizės dalyje, siekiant įvertinti LLM sukurtų pataisų proporcingumą.

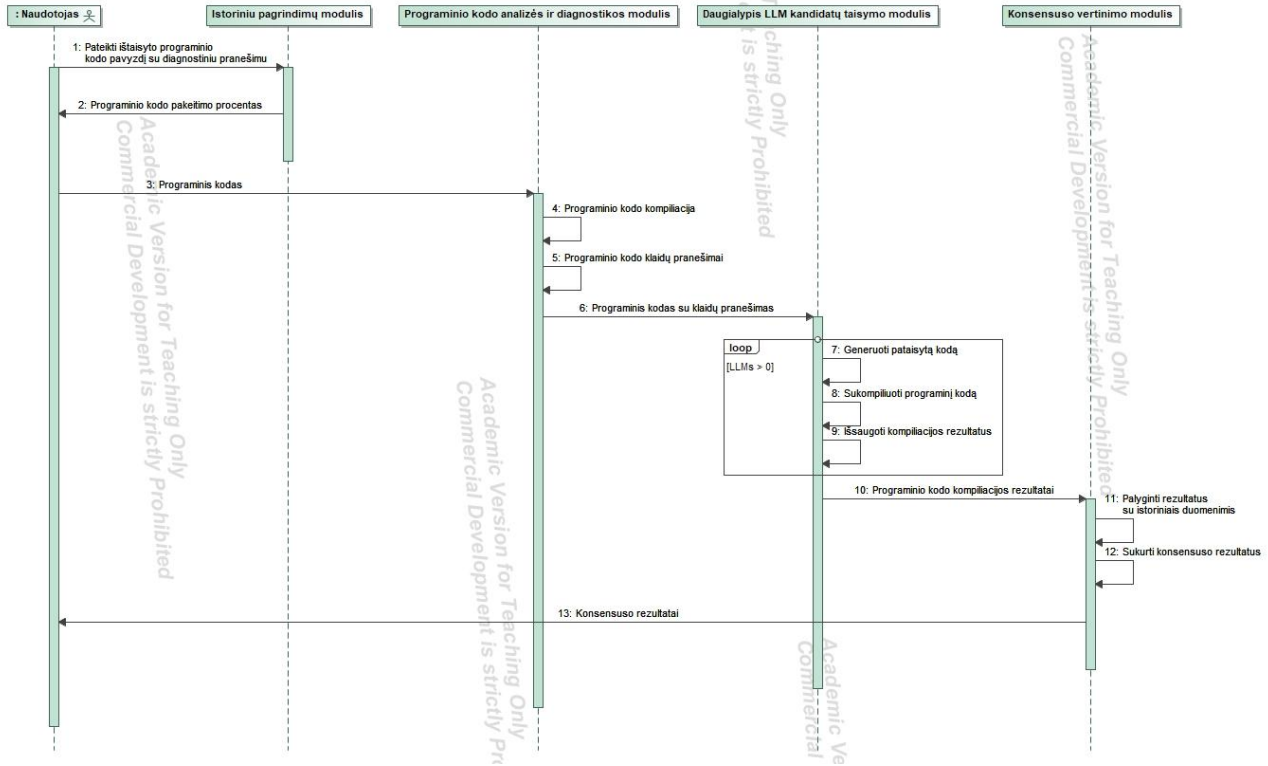
Konsensuso etape kandidatinių pataisų vertinimas turi būti atliekamas pagal aiškiai apibrėžtą kriterijų seką. Pirmasis privalomas kriterijus yra sėkmingas kompiliavimas. Jei kandidatinė pataisa nesukompiliuojama sėkmingai, ji negali būti laikoma galiojančia konsensuso proceso dalyve, nes toks rezultatas techniškai negali būti taikomas kodo bazei. Antrasis kriterijus yra tikslinės diagnostinės problemos išsprendimas. Net jei modifikuotas kodas sėkmingai kompiliuojamas, kandidatas nelaikomas galiojančiu, jei originali kompiliatoriaus ar statinės analizės problema lieka neišspręsta. Šis metodas užtikrina, kad kiekvienas kandidatas būtų vertinamas ne tik pagal sintaksinį teisingumą, bet ir pagal praktinį gebėjimą išspręsti nurodytą problemą.

Tik užbaigus šiuos du patvirtinimo etapus galima tinkamai įvertinti pakeitimo mastą. Šiame etape sistema nustato, ar LLM sugeneruotas pakeitimas nėra pernelyg didelis, palyginti su istoriniu pagrindimo procentu arba iš anksto nustatyta maksimalia leistina pakeitimo riba. Šis kriterijus padeda atmesti pataisas, kurios, nors ir gali būti kompiliuojamos bei pašalina tikslinę diagnostiką, vis dėlto modifikuoja pernelyg didelę kodo dalį, įveda nereikalingą refaktoringą arba nukrypsta nuo vyraujančio taisymo modelio, skirto nurodytai problemai. Todėl pakeitimo mastas veikia kaip proporcingumo kontrolė, atskirianti tikslinę pataisą nuo pernelyg didelių modifikacijų.

Paskutiniame etape kiekviena kandidatinė pataisa vertinama pagal visus nustatytus kriterijus. Konsensusas laikomas pasiektu tik tada, kai visi įvertinti kandidatai atitinka pagrindinius reikalavimus: sėkmingą kompiliavimą, tikslinės diagnostikos pašalinimą ir leistino pakeitimų masto neviršijimą. Jei bent vienas kandidatas pažeidžia šiuos kriterijus, laikoma, kad vyraujantis konsensusas nebuvo pasiektas. Tokiais atvejais sistema vis tiek gali pateikti atskirus modelių rezultatus, tačiau jų neturėtų laikyti galutiniu konsensusu pagrįstu sprendimu.

Pasiekus konsensusą ir pripažinus kelis kandidatus tinkamais, galutinis pataisos variantas pasirenkamas remiantis mažiausiu pokyčio dydžiu. Ši atrankos taisyklė grindžiama prielaida, kad tarp techniškai teisingų, kompiliuojamų ir diagnostiškai sėkmingų pataisų pirmenybė turėtų būti teikiama sprendimui, kuris mažiausiai keičia esamą kodo struktūrą. Laikoma, kad toks požiūris sumažina neigiamų šalutinių poveikių riziką, išlaiko originalaus kodo konteksto vientisumą ir leidžia pasirinkti proporcingiausią iš galimų pataisų.

2.7. Apibendrintas konsensuso algoritmas



12 pav. Apibendrintas metodo veikimas

12 pav. pavaizduota siūlomo, konsensusu grindžiamo statinės kodo analizės tobulinimo metodo integruota veikimo eiga. Diagrama iliustruoja anksčiau aprašytų komponentų sąveiką kaip procesą nuo pradžios iki pabaigos: nuo istorinių pagrindimo duomenų paruošimo iki saugyklos analizės, galimų pataisų generavimo, patvirtinimo ir galutinio konsensuso nustatymo. Todėl šis metodas nelaikomas atskirų komponentų rinkiniu, o veikiau nuoseklia operacijų seka, kurioje kiekvieno etapo rezultatas naudojamas kaip įvestis kitam etapui.

Įvykių seka prasideda, kai naudotojas pateikia pataisytą kodo iškarpą kartu su atitinkamu diagnostiniu pranešimu. Ši informacija perduodama istorinių pagrindimo duomenų moduliui, kuriame matuojamas santykinis žmogaus atlikto pataisymo mastas ir paverčiamas programos kodo pokyčio procentine dalimi. Šis procentas veikia kaip istorinė atskaitos vertė, atspindinti tikėtiną pataisymo dydį pasirinktam diagnostikos tipui. Po šio parengiamojo etapo vartotojas pateikia analizuoti tikslinį programos kodą arba saugyklą. Tuomet programos kodo analizės ir diagnostikos modulis kompiliuoja pateiktą kodą ir išgauna gautus kompiliatoriaus įspėjimus arba klaidų pranešimus. Taip metodas gauna diagnostinę informaciją ir kodo kontekstą, reikalingus taisymo užduočiai sukurti.

Nustačius diagnostinį kodo fragmentą, taisymo užduotis perduodama daugialypės LLM kandidatinių pataisų generavimo moduliui. Šiame etape keli pasirinkti dideli kalbos modeliai iškviečiami pakartotiniame cikle, kuriame kiekvienas modelis savarankiškai generuoja kandidatinį pataisos variantą tai pačiai problemai spręsti. Po kiekvieno kandidatinio pataisos varianto sukūrimo modifikuotas kodas vėl kompiliuojamas, o šios kompiliacijos rezultatas išsaugomas. Šis iteracinis patvirtinimo ciklas yra esminis metodikos elementas, nes užtikrina, kad kiekvienas kandidatas būtų vertinamas ne tik kaip tekstinis pasiūlymas, bet ir kaip techniškai patikrinamas taisymo bandymas.

Todėl metodas sukuria taisymo alternatyvų rinkinį kartu su objektyviais patvirtinimo rezultatais, o ne remiasi vienu nepatikrintu modelio atsakymu.

Pasibaigus kandidatų generavimo etapui, kompiliacijos ir patvirtinimo procesų rezultatai perduodami konsensuso vertinimo moduliui. Šiame etape gauti rezultatai lyginami su anksčiau apskaičiuotais istoriniais pagrindimo duomenimis. Šio palyginimo tikslas – nustatyti, ar kalbos modelių siūlomi pakeitimai išlieka pakankamai suderinti su istoriškai stebėta taisymo elgsena toje pačioje ar panašioje diagnostikos kategorijoje. Tuo pačiu metu sistema vertina sėkmingų kandidatų atliktų pataisų sutapimo laipsnį. Taigi galutinis sprendimas grindžiamas trimis vienas kitą papildančiais aspektais: ar kandidatas sėkmingai kompiliuojamas, ar jis išsprendžia arba pagerina tikslinę diagnostinę problemą ir ar jo pakeitimo mastas bei pataisos charakteristikos išlieka suderinamos su istoriškai pagrįstais lūkesčiais bei kitais tinkamais kandidatais.

Metodo rezultatas yra konsensuso išvada, kuri vėliau pateikiama naudotojui. Tais atvejais, kai tarp patvirtintų kandidatinių pataisų yra pakankamas sutapimas, o siūlomi pakeitimai išlieka priimtinai suderinti su istoriniu pagrindu, metodas pateikia galutinį konsensusu pagrįstą pataisos rezultatą. Jei tokio konsensuso nėra, metodas nurodo, kad patikimo konsensuso nustatyti nepavyko. 12 pav. pavaizduotas integruotas darbo srautas parodo, kaip siūlomas metodas sujungia istorines taisymo žinias, diagnostikos išgavimą saugyklos lygmeniu, pakartotinį daugiamodelių pataisų generavimą, patvirtinimą remiantis perkompiliavimu ir konsensusu grindžiamą sprendimų priėmimą į vientisą procesą, siekiant padidinti automatizuoto statinės kodo analizės taisymo patikimumą.

2.8. Konsensuso programinio kodo tobulinimo projekto išvados

Apžvelgus siūlomą statinės kodo analizės tobulinimo metodo projektą galima daryti išvadą, kad:

1. Įrodyta, kad ankstesnių klaidų taisymų poros sudaro tvirtą pagrindą numatomiems kodo pakeitimams.
2. Kodo saugyklos kompiliacijos išskiria diagnostinius duomenis ir sukuria taisymo užduotis.
3. Suprojektuota, kad keli didelio masto kalbos modeliai (LLM) generuotų pataisas, kurios vėliau būtų patvirtinamos atliekant kompiliaciją ir klaidų patikrinimus.
4. Šių pataisų patvirtinimas vyksta taikant konsensuso principu pagrįstą metodą, pagal kurį veiksmingiausias sprendimas nustatomas per nuolatinį tobulinimo procesą.
5. Suprojektuotas metodas apibrėžia pagrindinius sistemos komponentus ir jų sąveiką, todėl sudaro pagrindą prototipo įgyvendinimui bei tolesniam eksperimentiniam vertinimui.

3. Statinės kodo analizės tobulinimo metodo prototipas

Šiame skyriuje aprašomas siūlomo metodo prototipas. Prototipas buvo įgyvendintas ir paleistas lokaliai kompiuteryje su „Windows 11 Home 25H2“ operacine sistema, naudojant „Python 3.14.3“ kaip pagrindinę kūrimo aplinką. 3.1 skyriuje aprašomos prototipe panaudotos technologijos ir priemonės.

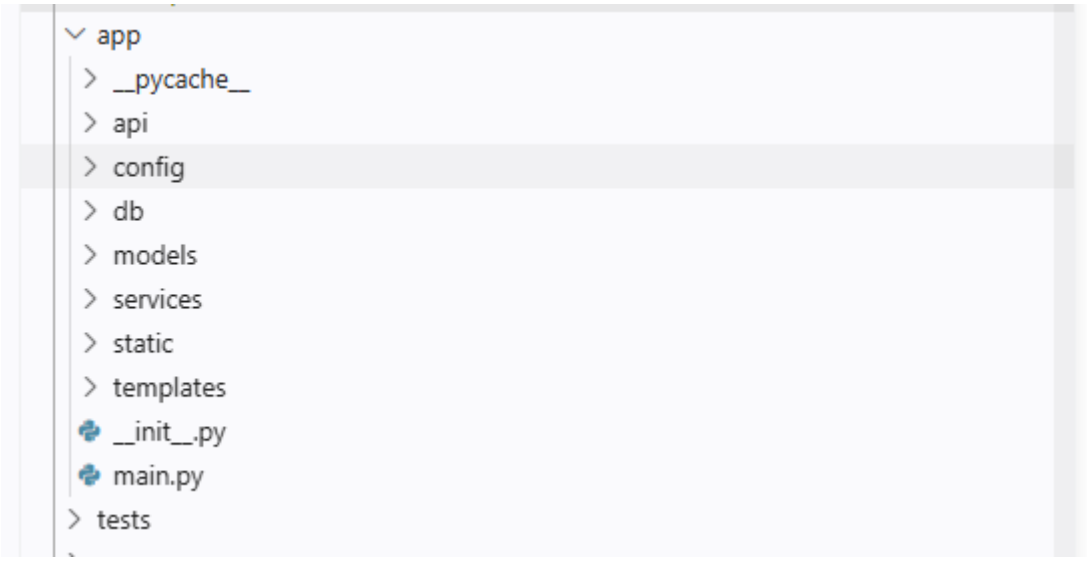
3.1. Prototipo technologijos

Siūlomos sistemos prototipas įgyvendintas kaip lokaliai veikianti aplinka, susidedanti iš projekto vykdytojo paslaugos, išplėstinių LLM teikėjų integracijų ir projekto kompiliatoriaus komponento. Projekto vykdytojo paslauga, atsakinga už klaidų surinkimą, pagrįstumo procento skaičiavimą ir skirtingų LLM teikėjų iškvietimą, realizuota naudojant „Python 3.14.3“, „Git 2.53.0.windows.2“ ir „SQLite 3.50.4“ per „Python sqlite3“ biblioteką. „Python“ pasirinktas dėl greito prototipavimo, patogaus darbo su failų sistema, išoriniais procesais ir HTTP integracijomis, kurios yra svarbios vykdant programinio kodo analizę, formuojant LLM užklausas ir apdorojant jų atsakymus. „Git“ leidžia automatiškai atsisiųsti programinio kodo saugyklas, atkurti jų pradinę būseną ir užtikrinti, kad kiekvienas LLM sugeneruotas taisymo variantas būtų tikrinamas vienodomis sąlygomis. „SQLite“ naudojama kaip lengva lokali duomenų saugykla tarpiniams rezultatams, klaidų kodams ir analizės metaduomenims kaupti, nereikalaujant papildomos serverinės infrastruktūros.

LLM pagrįsta saugyklos analizė atliekama per išorinių teikėjų integracijas, įgyvendintas naudojant „OpenAI Python SDK“ 1.99.9 ir „Anthropic Python SDK“ 0.62.0. Šie įrankiai leidžia vieningai formuoti užklausas skirtingiems modeliams ir nuosekliai apdoroti jų atsakymus, o tai svarbu kelių LLM pasiūlymų palyginimui ir sutarimo nustatymui. Projekto kompiliavimą valdo atskiras komponentas, naudojantis „.NET SDK / CLI“ 10.0.201, kuris leidžia automatizuotai atkurti projektą, vykdyti kompiliavimą ir surinkti diagnostinius pranešimus. Toks sprendimas užtikrina, kad LLM sugeneruoti pakeitimai būtų vertinami realiomis vykdymo sąlygomis, atsižvelgiant ne tik į tikslinės problemos pašalinimą, bet ir į naujai atsiradusias klaidas ar įspėjimus. Pasirinkta technologijų struktūra leidžia kurti lankstų, vietiniams eksperimentams tinkamą prototipą, o modulinė komponentų architektūra sudaro prielaidas sistemą ateityje plėsti papildomais modeliais ar kitomis technologijomis.

3.2. Prototipo sandara

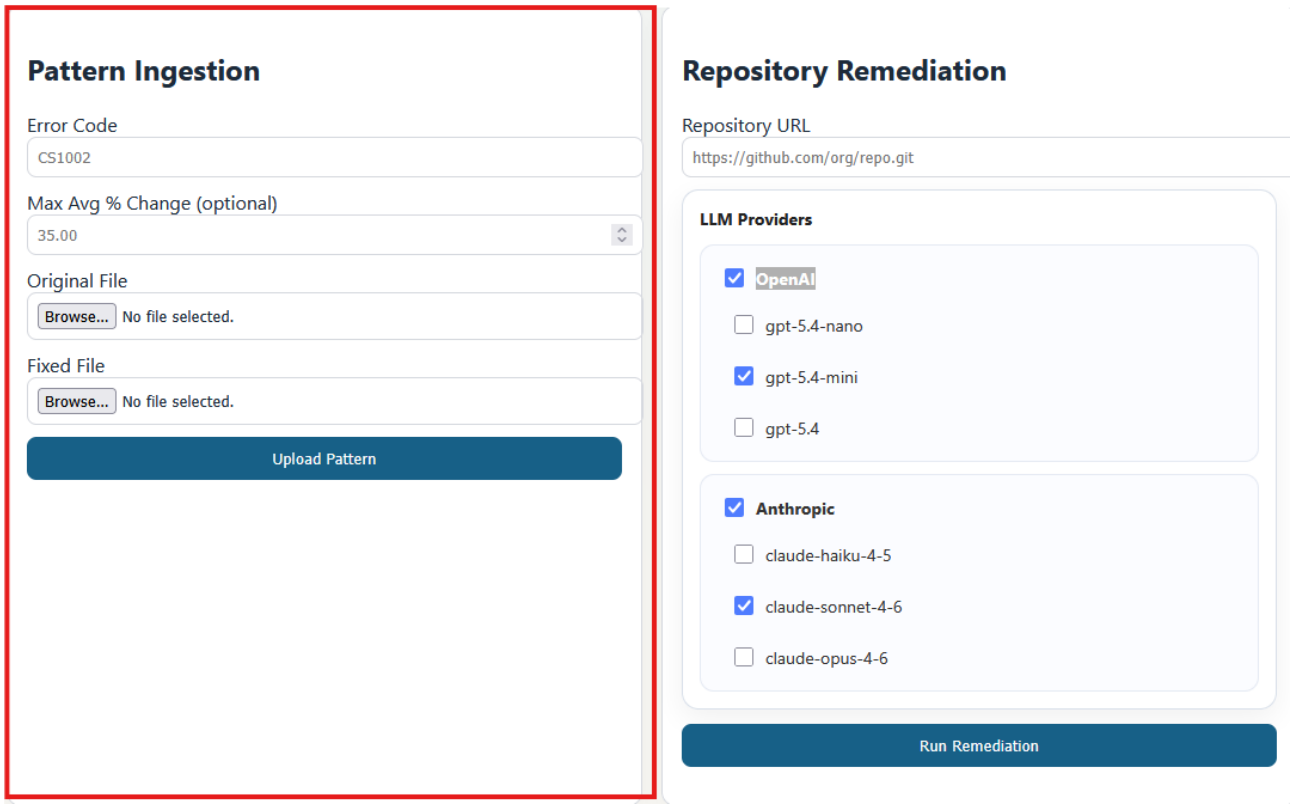
Prototipo konfigūracija pavaizduota 13 paveiksle, kuriame parodyta paslaugų komponento struktūra. Kaip matyti iš paveikslo, prototipas suskirstytas į keletą pagrindinių dalių, įskaitant API, konfigūraciją, duomenų bazę, modelius, paslaugas, statinius išteklius, šablonus, pagrindinius programos paleidimo failus ir testus. Ši struktūra buvo pasirinkta dėl būtinybės atskirti skirtingas funkcines atsakomybes, taip užtikrinant aiškesnį sistemos organizavimą, paprastesnę priežiūrą ir geresnes tolesnio plėtimo galimybes. Toliau pateiktuose skyriuose aptariamas kiekvienas prototipo struktūrinis elementas ir jo funkcija bendroje įgyvendinimo sistemoje.



13 pav. Prototipo aplankų medis

3.2.1. API sąsaja

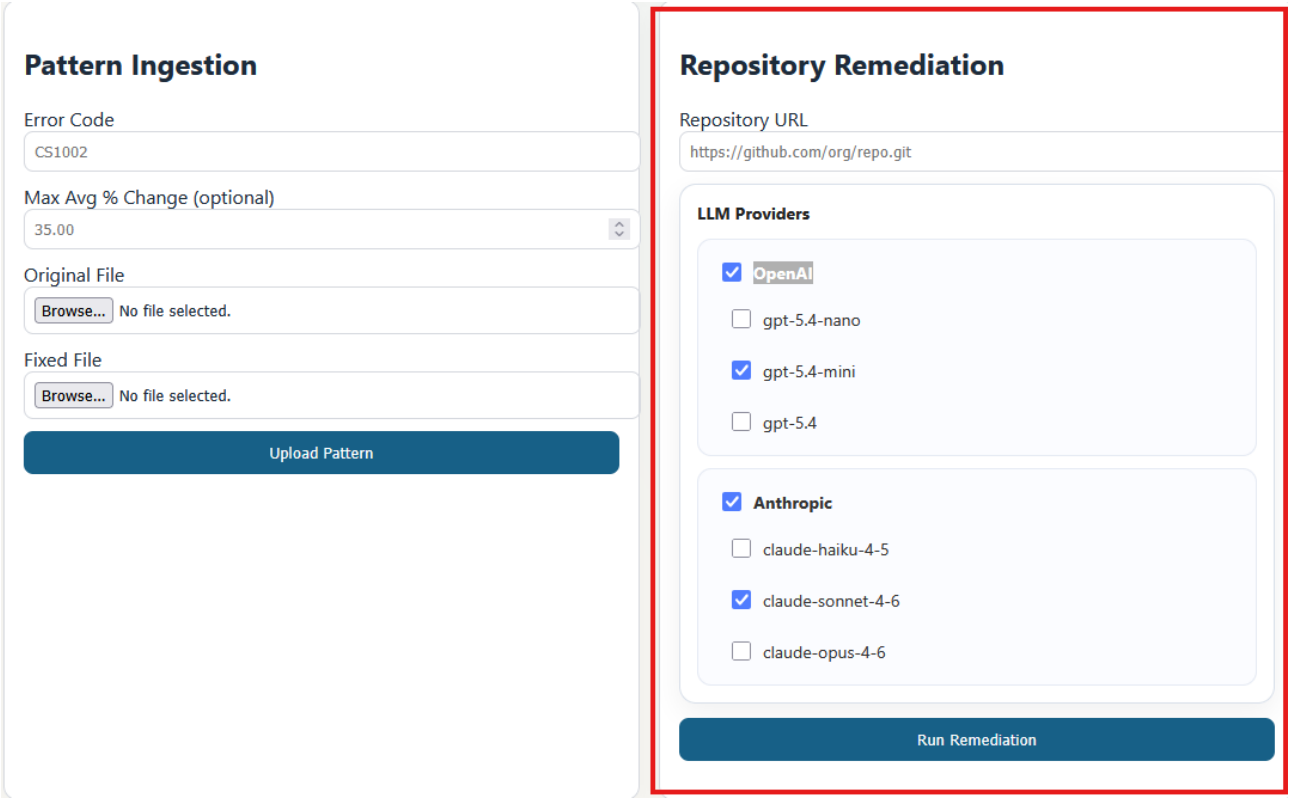
Prototipo API lygis susideda iš dviejų pagrindinių komponentų, kurie užtikrina pagrindinį sistemos darbo srautą. Pirmasis segmentas skirtas priimti klaidos kodą kartu su šaltinio kodo failu prieš pataisos įgyvendinimą ir atitinkamu failu po pataisos įgyvendinimo. Antroji dalis skirta priimti saugyklos nuorodas, kurios vėliau naudojamos projektų paieškai, kompiliavimui ir tolesnei analizei. Šis suskirstymas leidžia prototipui atskirti bazinių duomenų paruošimą nuo realių projektų saugyklų analizės.



14 pav. Pirmasis API sąsajos prieigos taškas

Pirmasis API prieigos taškas pavaizduotas 14 pav. yra atsakingas už pateiktų kodo pakeitimų pagrįstumo procentinės dalies apskaičiavimą. Palygindama šaltinio failą prieš pataisos įgyvendinimą

su to paties failo pataisyta versija, sistema nustato kodo modifikacijos, susijusios su žinomu sprendimu, proporciją. Šis rezultatas naudojamas kaip atskaitos bazė, kuri vėliau taikoma vertinant, kiek LLM sukurtos pataisos atitinka numatytą kodo pakeitimų mastą konkrečiai klaidai.



15 pav. Antrasis API sąsajos prieigos taškas

Antrasis sąsajos taškas, pavaizduotas 15 pav. yra atsakingas už pateiktos kodo saugyklos paiešką ir jos analizės darbo eigos inicijavimą. Gavus programinio kodo saugyklą, sistema atlieka parengiamuosius veiksmus, įskaitant kompiliaciją ir įspėjimų išgavimą, prieš perduodama atitinkamą projekto informaciją integruotiems LLM teikėjams tolesnei analizei. Tokiu būdu galinis taškas integruoja kodo saugyklos gavimą, kompiliacija pagrįstą tikrinimą ir LLM palaikomą vertinimą į vieningą apdorojimo grandinę.

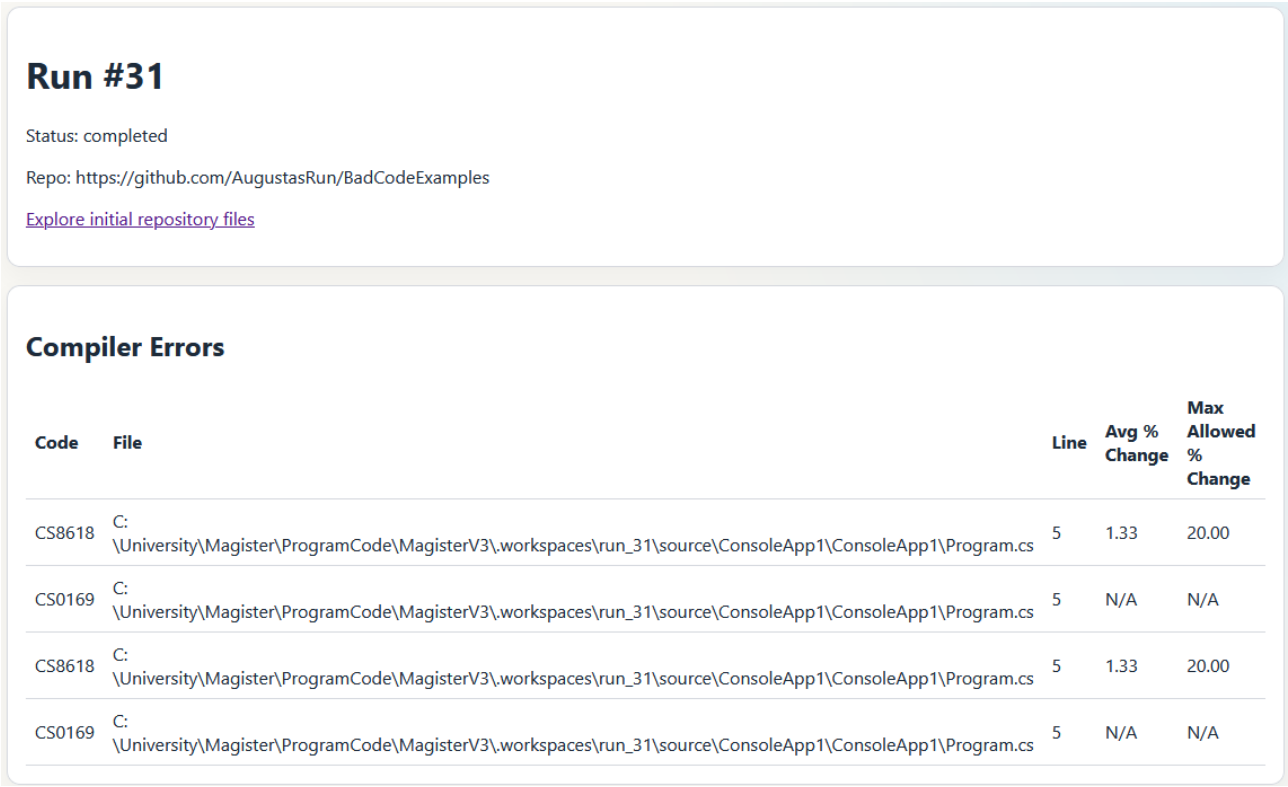
3.2.2. Programinio kodo kompiliacija

Antrasis API sąsajos taškas yra atsakingas už naudotojo pateiktos saugyklos apdorojimą ir pagrindinės prototipo analizės darbo eigos inicijavimą. Gavusi saugyklos nuorodą, sistema atsisiunčia programinį kodą ir parengia jį tolesniam apdorojimui vietinėje analizės aplinkoje. Šis žingsnis yra būtinas siekiant užtikrinti, kad projektas būtų valdomas nuosekliai, nepriklausomai nuo konkrečios kodo saugyklos, pateiktos vertinimui.

Atsisiuntus saugyklą, nurodytas galinis taškas paleidžia projekto kompiliatoriaus komponentą, taip pradėdamas pateikto .NET projekto kompiliavimą. Kompiliavimo proceso metu sistema fiksuoja kompiliacijos išvestį ir išskiria kompiliatoriaus sukurtus įspėjimus bei klaidas. Šie rezultatai sudaro pagrindą tolesnei analizei, nes jie identifikuoja kodo problemas, kurias vėliau reikia ištirti ir išspręsti prototipo darbo eigoje.

Jei sistemoje jau yra nustatyti klaidos atitikimo procentiniai duomenys, galinis taškas šią vertę priskiria aptiktai problemai. Todėl kompiliatoriaus sugeneruoti įspėjimai ir klaidos papildomi

atskaitos tašku, kuris nurodo numatomą kodo modifikavimo mastą. Šis papildymas leidžia prototipui ne tik aptikti su kompiliavimu susijusias problemas, bet ir susieti jas su anksčiau apskaičiuota atitikimo informacija. Ši informacija vėliau gali būti naudojama lyginant LLM sugeneruotus pataisymus. Sistemos rezultato pavyzdys pateikiamas 16 pav.

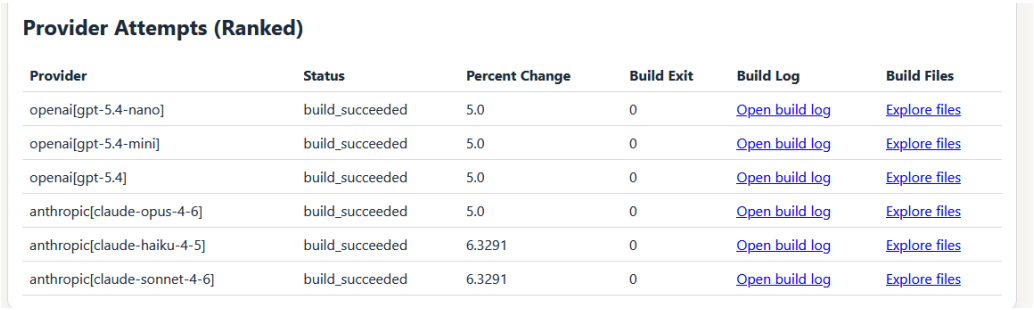


16 pav. Programinio kodo kompiliacijos langas

3.2.3. Kelių didelių kalbos modelių rezultatai

Atsisiuntus kodo saugyklą ir nustačius atitinkamus įspėjimus ar klaidų kodus, ši informacija perduodama integruotų didelių kalbos modelių (LLM) teikėjams tolesniam apdorojimui. Atsižvelgdamas į pranešimą apie kompiliatoriaus problemą ir susijusį šaltinio kodo kontekstą, kiekvienas pasirinktas modelis sukuria siūlomą kodo pakeitimą, kuriuo siekiama išspręsti nustatytą problemą. Tokiu būdu prototipas papildo kompiliatoriaus atliekamą analizę automatizuotu kodo pataisų generavimu, kurį atlieka keli dideli kalbos modeliai.

Kiekvienam pasirinktam LLM sistema saugo sukurtą išvestį atskirame failų kataloge, taip leidžiant nepriklausomai valdyti ir peržiūrėti sukurtus kodo variantus. Ši katalogais pagrįsta organizacija palengvina kiekvieno kodo pasiūlymo kilmės sekimą, taip pat atitinkamų pakeistų failų išsaugojimą vėlesnei patikrai, palyginimui ir vertinimui. Šis atskyrimas ypač svarbus daugelio teikėjų kontekste, kur įvairūs modeliai gali pateikti skirtingus sprendimus tai pačiai problemai.



Provider Attempts (Ranked)

Provider	Status	Percent Change	Build Exit	Build Log	Build Files
openai[gpt-5.4-nano]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4-mini]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-opus-4-6]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-haiku-4-5]	build_succeeded	6.3291	0	Open build log	Explore files
anthropic[claude-sonnet-4-6]	build_succeeded	6.3291	0	Open build log	Explore files

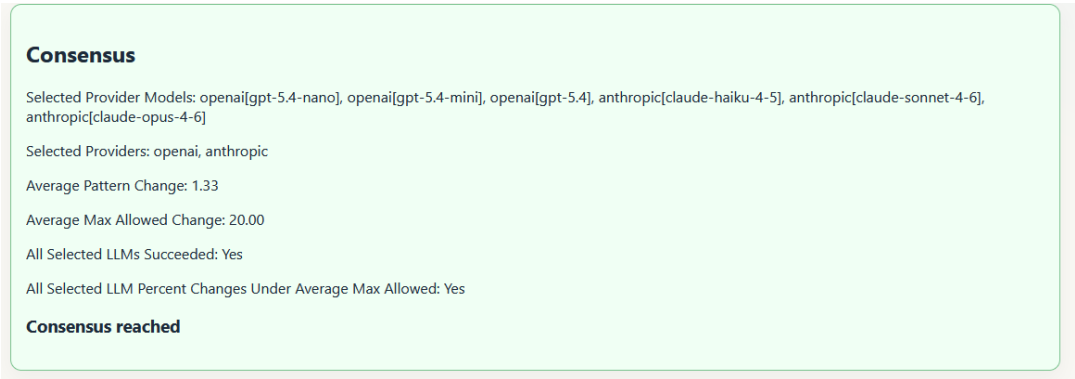
17 pav. Programinio kodo taisymo rezultatai

Kaip parodyta 17 paveiksle, prototipas taip pat įrašo su kiekvienu LLM sukurtu bandymu susijusių kodo pakeitimų procentinę dalį kartu su jo kompiliavimo būsena ir susijusiais išvesties duomenimis. Tai leidžia palyginti sukurtus sprendimus ne tik pagal jų sėkmingą kompiliaciją, bet ir pagal jų modifikavimo apimties atitikimą numatytam pagrindimo procentui. Sąsajoje pademonstruotas failų naršymas ir prieiga prie kompiliavimo žurnalo dar labiau palengvina skaidrų kiekvieno atskiro teikėjo bandymo tyrimą.

3.2.4. Konsensuso rezultatai

Pasibaigus atrinktų didelių kalbos modelių (LLM) bandymams generuoti kodą ir sėkmingai apdorojus gautus sprendimus, prototipas atlieka papildomą vertinimo etapą, kuriuo siekiama nustatyti, ar galima pasiekti bendrą sutarimą. Šiame etape sistema nagrinėja visų dalyvaujančių modelių pateiktus rezultatus, įskaitant jų sėkmės statusą ir su kiekvienu sukurtu sprendimu susijusių kodo pakeitimų procentinę dalį. Šis metodas leidžia įvertinti du pagrindinius aspektus: pirma, modelių gebėjimą generuoti kompiliuojamus rezultatus, ir, antra, jų pakeitimų atitikimą priimtinam diapazonui, apibrėžtam remiantis pagrindimo duomenimis grindžiamais palyginimo kriterijais.

Jei įvykdomos būtinos sąlygos, sistema nustato, kad konsensusas pasiektas, ir atitinkamus rezultatus rodo naudotojo sąsajoje. Kaip parodyta 18 paveiksle, rezultatai apima pasirinktus modelius, dalyvaujančius teikėjus, vidutinį modelio pakeitimą, didžiausią leidžiamą pakeitimą ir rodiklius, nurodančius, ar visi pasirinkti LLM buvo sėkmingi ir neviršijo leidžiamos pakeitimų ribos. Šis paskutinis žingsnis suteikia išsamią bendrų rezultatų apžvalgą ir leidžia naudotojui įsitikinti, ar sugeneruoti sprendimai atitinka konsensuso pagrindu apibrėžtus kriterijus.



Consensus

Selected Provider Models: openai[gpt-5.4-nano], openai[gpt-5.4-mini], openai[gpt-5.4], anthropic[claude-haiku-4-5], anthropic[claude-sonnet-4-6], anthropic[claude-opus-4-6]

Selected Providers: openai, anthropic

Average Pattern Change: 1.33

Average Max Allowed Change: 20.00

All Selected LLMs Succeeded: Yes

All Selected LLM Percent Changes Under Average Max Allowed: Yes

Consensus reached

18 pav. Konsensuso rezultatas

3.2.5. Konsensuso patvirtinimo įgyvendinimo detalės

Prototipe konsensuso patvirtinimas įgyvendinamas kaip taisyklėmis pagrįstas palyginimo lygmuo, įterptas po atskirų LLM vykdymo ir perkompiliavimo etapų. Šio tyrimo tikslas nėra atkurti ar iš naujo interpretuoti siūlomus sprendimus, o standartizuoti visų modelių bandymų rezultatus į vienodą vertinimo formatą. Ši galimybė leidžia sistemai sistemingai vertinti įvairius LLM rezultatus, taikant vienodus kriterijus kiekvienam sprendimui, nepriklausomai nuo teikėjo ar modelio, kuris jį sukūrė.

Pagrindimo skaičiavimą atlieka prototipas, kuris susieja kiekvieną saugomą istorinį modelį su konkrečiu diagnostiniu ar klaidos kodu. Todėl pagrindimo vertė nelaikoma universaliu slenksčiu visiems taisymams. ji naudojama kaip konkrečiai klaidai būdingas atskaitos taškas. Jei vėliau, analizuojant saugyklą, nustatoma diagnostika, prototipas gali palyginti sukurtus pakeitimus tik su to paties klaidos kodo pagrindimo duomenimis. Tais atvejais, kai istorinių verčių konkrečiai diagnostikai nėra, konsensuso palyginimas vis tiek gali remtis kompiliacija ir diagnostikos išsprendimu, tačiau proporcingumo vertinimas yra ribotas.

Pakeitimo procentas gaunamas apskaičiuojant eilutėmis pagrįstus skirtumus tarp originalaus ir modifikuoto failo. Šis metodas buvo pasirinktas, nes palyginimas simbolių lygiu gali iškreipti rezultatą, kai modifikacija įvyksta netoli failo pradžios. Tokiais atvejais net nedidelis įterpimas ar šalinimas gali sukelti reikšmingą vėlesnių simbolių poslinkį, sukurdamas dirbtinį ir žymų skirtumą. Eilutėmis pagrįstas palyginimas suteikia stabilesnį faktinio modifikacijos masto apytikslį įvertinimą ir yra lengviau interpretuojamas nustatant, ar generuotas pataisymas yra pernelyg platus.

Sutarimo lygmuo skirtas atskirti techninį teisingumą nuo proporcingumo. Kompiliacijos ir tikslinės diagnostikos išsprendimo procesai laikomi privalomais, o pagrįstas pakeitimų slenkstis naudojamas kaip pagalbinis reguliavimo mechanizmas, skirtas pernelyg dideliems pakeitimams riboti. Šis skirtumas yra svarbus, nes pataisa gali būti techniškai teisinga, bet vis tiek pernelyg plati numatytam pataisymui. Šių patikrinimų įgyvendinimas atskirai leidžia prototipui nustatyti, ar kandidato nesėkmė gali būti priskirta šioms priežastims: kandidatas nebuvo sėkmingai apdorotas kompiliavimo metu, nepavyko išspręsti pradinės problemos arba viršijo numatytą modifikacijos dydį.

Jei keletas kandidatų atitinka reikiamus patvirtinimo kriterijus, prototipas teikia pirmenybę pataisai, turinčiai mažiausią galiojančių pakeitimų procentą. Ši taisyklė užtikrina, kad galutinis pasirinkimas būtų deterministinis, taip sumažinant tikimybę pasirinkti sprendimą, kuris atlieka nereikalingą refaktoringą. Todėl konsensusas prototipe įgyvendinamas ne kaip paprastos daugumos balsavimas, o kaip filtruotas atrankos procesas, kuriame galutiniam pasirinkimui tinka tik kompiliuojamos, problemą sprendžiančios ir proporcingos pataisos.

3.3. Prototipo išvados ir rezultatai

Prototipas sukurtas ir paleistas lokaliai kompiuteryje su „Windows 11 Home 25H2“ operacine sistema, naudojant „Python 3.14.3“ kaip pagrindinę kūrimo aplinką, „Git 2.53.0.windows.2“ saugyklų tvarkymui, „SQLite 3.50.4“ vietiniam duomenų saugojimui, „OpenAI Python SDK 1.99.9“ ir „Anthropic Python SDK 0.62.0“ didelių kalbos modelių (LLM) integracijai bei „.NET SDK“ / CLI 10.0.201 – projekto kompiliacijai. Prototipą sudaro keli pagrindiniai komponentai, įskaitant API sluoksnį, konfigūracijos modulį, duomenų bazės sluoksnį, modelius, paslaugas, statinius išteklius, šablonus, paleidimo failus ir testus. Įgyvendinta sistema šioje struktūroje apima du pagrindinius API galinius taškus. Pirmasis galinis taškas skirtas pagrįstumo procentui skaičiuoti remiantis šaltinio kodu

prieš ir po žinomo pataisymo. Antrasis galinis taškas leidžia gauti saugyklą, kompiliuoti projektą, išgauti įspėjimus, atlikti LLM pagrįstą taisymą ir konsensuso vertinimą.

Iš prototipo aprašymo ir jo įgyvendinimo rezultatų galima padaryti keletą išvadų:

1. Siūlomas statinės kodo analizės tobulinimo metodas gali būti praktiškai įgyvendintas kaip veikiantis prototipas, integruojantis saugyklos analizę, kompiliacijos pagrįstą diagnostiką, LLM generuotus pataisymus ir konsensuso vertinimą.
2. Prototipas užtikrina visapusišką darbo eigą, kurioje pagrįstumo procentinės dalies apskaičiavimo ir saugyklos taisymo procesai yra suskaidyti į specializuotus API galinius taškus, taip užtikrinant aiškų ir sistemingą apdorojimo procesą.
3. Integravus kelis didelius kalbos modelius, sistema gali generuoti alternatyvius pataisymų variantus, juos saugoti atskirai ir vertinti pagal kompiliacijos sėkmę bei kodo pakeitimo procentą.
4. Konsensuso etapas leidžia sistemai apibendrinti visų atrinktų modelių rezultatus ir nustatyti, ar generuoti pataisymai atitinka nustatytus priimtinumo kriterijus. Šis procesas padidina galutinio rezultato skaidrumą ir patikimumą.
5. Įgyvendintas prototipas parodo, kad pasiūlytą metodą galima praktiškai realizuoti, integruojant saugyklos analizę, kompiliacijos patikrinimą, kelių LLM pataisų generavimą ir konsensuso rezultatų pateikimą naudotojui.

4. Statinės kodo analizės tobulinimo metodo tyrimas

Šiame skyriuje nagrinėjamas statinės kodo analizės tobulinimo metodo, grindžiamo konsensuso principu ir kelių didelių kalbos modelių naudojimu, veiksmingumas. Metodas vertinamas atliekant eksperimentinius bandymus su C# programiniu kodu. Prototipo vertinimo tikslas – įvertinti pasirinktų kodo taisymo ir sprendimų priėmimo mechanizmų funkcionalumą bei veiksmingumą apdorojant kompiliatoriaus įspėjimus ir klaidas. Tyrime taip pat vertinamas siūlomo metodo veiksmingumas kompiliacijos sėkmės, tikslinės diagnostikos išsprendimo, naujų problemų atsiradimo ir sugeneruotų kodo pakeitimų atitikties istoriškai nustatytam pagrindimo procentui aspektais. Siūlomo metodo veiksmingumas vertinamas lyginant gautus rezultatus su rezultatais, pasiektais naudojant atskirus didelius kalbos modelius. Tai leidžia nustatyti, ar konsensusu pagrįstas požiūris leidžia gauti patikimesnius ir tikslesnius statinės kodo analizės išvadų taisymo rezultatus.

4.1. Statinės kodo analizės tobulinimo metodo testavimas

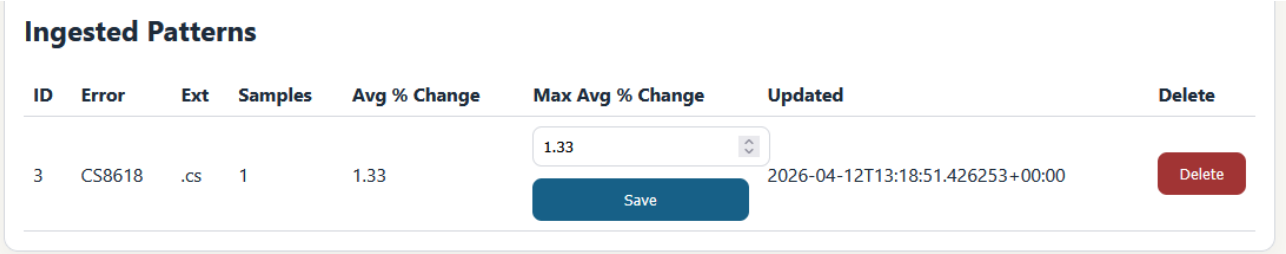
Siūlomas metodas buvo išbandytas atliekant seriją rankinių bandymų naudojant prototipinę naudotojo sąsają.1 lentelė pateikta atliktų bandymų santrauka, kurioje išsamiai aprašytos tikrintos funkcijos, numatyti rezultatai ir rezultatai, gauti po kiekvieno bandymo atlikimo.

1 lentelė. Prototipo testavimo atvejai

Testas	**Tikrinimas**	**Laukiamas rezultatas**	**Rezultatas**
„Grounding“ procento apskaičiavimas	Tikrinama, ar sistema, pateikus galiojantį klaidos kodą, pradinį šaltinio failą ir pataisytą failą, teisingai inicijuoja „grounding“ procento skaičiavimą	Sistema sėkmingai priima užklausą ir apskaičiuoja „grounding“ procentinę reikšmę	Sėkmingai priimta užklausta ir apskaičiuota procentinė reikšmė
Klaidos kodo validavimas	Tikrinama sistemos elgsena, kai „grounding“ procento skaičiavimo užklausoje nepateikiamas klaidos kodas	Sistema atmeta užklausą ir pateikia pranešimą apie privalomą klaidos kodo lauką	Užklausa atmesta, pateiktas klaidos pranešimas apie privalomą klaidos kodo lauką.
Privalomų failų validavimas	Tikrinama sistemos elgsena, kai „grounding“ procento skaičiavimo metu nepateikiamas vienas iš būtinų failų	Sistema atmeta užklausą ir nurodo, kad turi būti pateikti abu failai	Užklausa atmesta, pateiktas pranešimas, kad „grounding“ procento skaičiavimui būtina pateikti pradinį ir pataisytą failus.
Vienodų failų palyginimas	Tikrinama sistemos elgsena, kai tas pats failas pateikiamas kaip pradinis ir pataisytas	Sistema sėkmingai apdoroja užklausą, o apskaičiuotas „grounding“ procentas yra lygus 0 % arba artimas 0 %	Užklausa sėkmingai apdorota, apskaičiuotas „grounding“ procentas lygus 0 %.
Repozitorijos analizės inicijavimas	Tikrinama, ar sistema, pateikus galiojančią .NET repozitorijos nuorodą ir pasirinkus bent vieną LLM, pradeda analizės darbo eigą	Sistema sėkmingai parsiunčia repozitoriją ir inicijuoja analizės procesą	Repozitorija sėkmingai parsiųsta, analizės darbo eiga inicijuota ir pradėtas diagnostinių pranešimų apdorojimas.
Repozitorijos nuorodos validavimas	Tikrinama sistemos elgsena, kai repozitorijos analizės užklausoje nepateikiama repozitorijos nuoroda	Sistema atmeta užklausą ir pateikia pranešimą apie privalomą repozitorijos URL lauką	Užklausa atmesta, pateiktas klaidos pranešimas apie privalomą repozitorijos URL lauką.

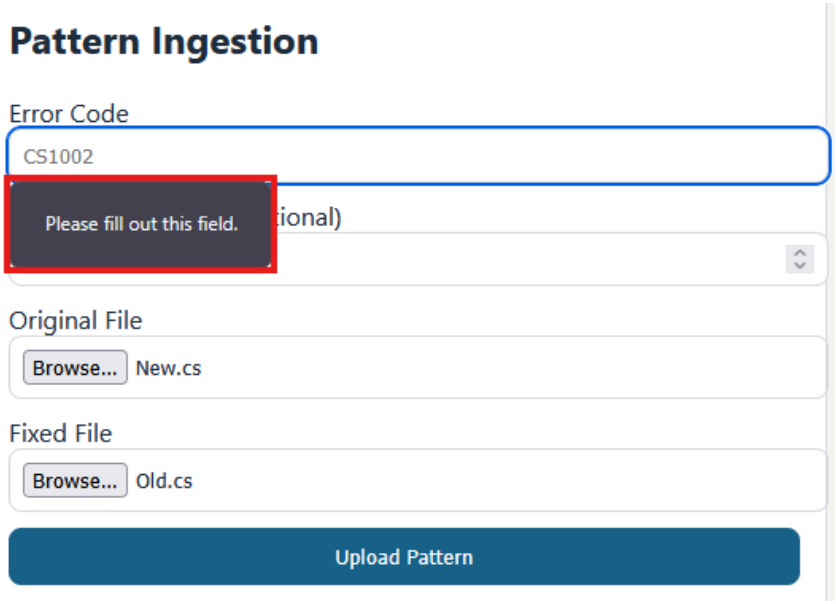
Testas	Tikrinimas	Laukiamas rezultatas	Rezultatas
Nepasiekiamos repozitorijos apdorojimas	Tikrinama sistemos elgsena, kai pateikiama nepasiekiama arba privati repozitorija	Nepavykus gauti repozitorijos duomenų, sistema pateikia klaidos pranešimą ir neužbaigia analizės.	Repozitorijos gauti nepavyko, pateiktas klaidos pranešimas, o analizės procesas nebuvo tęsiamas.
Diagnostinių pranešimų išgavimas	Tikrinama, ar iš .NET repozitorijos, turinčios kompiliatoriaus įspėjimų arba klaidų, sistema išgauna diagnostinius duomenis	Sistema pateikia diagnostinius kodus, failų vietas ir kitą susijusią analizės informaciją	Sistema sėkmingai išgavo diagnostinius kodus, susijusias failų vietas ir papildomą analizės informaciją.
Kelių LLM pataisymų generavimas	Tikrinama, ar pasirinkus kelis LLM modelius tam pačiam diagnostikos atvejui sistema sugeneruoja atskirus pataisymo variantus	Kiekvienas pasirinktas modelis sugeneruoja nepriklausomą pataisymo variantą, kuris išsaugomas atskirai	Kiekvienas pasirinktas LLM modelis sugeneravo atskirą kandidatinį pataisymą, o rezultatai buvo išsaugoti atskirai.
Sugeneruoto pataisymo validavimas	Tikrinama, ar kiekvieno sugeneruoto kandidatinio pataisymo tinkamumas vertinamas pakartotinės kompiliacijos būdu	Sistema pritaiko sugeneruotą pataisymą ir iš naujo sukompiliuoja projektą	Sugeneruotas kandidatinis pataisymas buvo pritaikytas, projektas pakartotinai sukompiliuotas ir gautas validavimo rezultatas.
Pakeitimo procento nustatymas	Tikrinama, ar kiekvienam LLM sugeneruotam rezultatui sistema apskaičiuoja kodo pakeitimo procentinę reikšmę	Sistema prie kiekvieno sugeneruoto rezultato pateikia pakeitimo procentą	Kiekvienam LLM sugeneruotam rezultatui apskaičiuota ir pateikta kodo pakeitimo procentinė reikšmė.
Konsensuso nustatymas	Tikrinama, ar sistema, įvertinusi visų pasirinktų LLM rezultatus, nustato galutinę konsensuso būseną	Sistema pateikia išvadą, ar konsensusas buvo pasiektas, ir parodo galutinę rezultatų suvestinę	Sistema įvertino pasirinktų LLM rezultatus, pateikė konsensuso būseną ir galutinę rezultatų suvestinę.

Kaip parodyta 19 pav., pateikiamas sėkmingo grounding procento apskaičiavimo ir išsaugojimo pavyzdys. Šis apskaičiavimas buvo atliktas vartotojui pateikus kompiliatoriaus klaidos kodą bei pradinius ir pataisytus šaltinio failus. Sistema užregistravo .cs failų klaidą CS8618, užfiksavo vieną pavyzdžių porą ir apskaičiavo vidutinį kodo pakeitimą, kuris sudarė 1,33 %. Šis rodiklis taip pat buvo išsaugotas kaip didžiausias vidutinis procentinis pokytis.



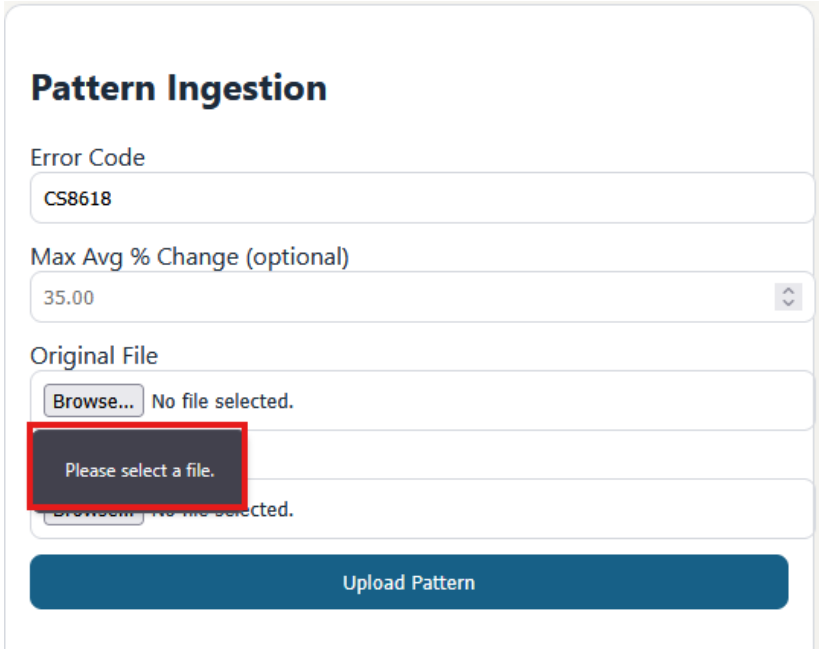
19 pav. „Grounding“ procento apskaičiavimo rezultatas

20 pav. pavaizduotas šablono įkėlimo formos tikrinimo veikimas, kai reikalaujamas klaidos kodas nėra tinkamai nurodytas. Nors vartotojo sąsaja leidžia pateikti klaidos kodą kartu su originaliais ir pataisytais failais, sistema rodo tikrinimo pranešimą „**Please fill out this field.**“, nurodydama, kad klaidos kodo laukas yra privalomas ir turi būti užpildytas prieš įkeliant šabloną. Tai patvirtina, kad forma užtikrina įvesties tikrinimą ir neleidžia pateikti neišsamių duomenų. Taip užtikrinama, kad „grounding“ procento skaičiavimai būtų atliekami tik tada, kai pateikiami visi reikalingi duomenys.



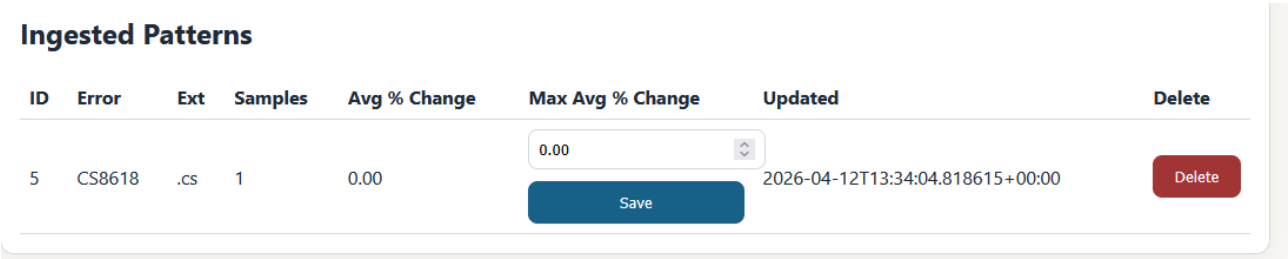
20 pav. Klaidos kodo validavimo rezultatas

21 pav. matyti, kad, be privalomojo klaidos kodo lauko tikrinimo, šablonų įkėlimo forma taip pat užtikrina failų įvesties tikrinimą. Nors buvo įvestas klaidos kodas **CS8618** ir neprivaloma maksimali vidutinė procentinio pokyčio vertė, sistema rodo pranešimą „**Please select a file.**“, kai naudotojas bando tęsti neįkėlęs reikiamų šaltinio failų. Tai patvirtina, kad forma neleidžia pateikti neišsamių duomenų ir užtikrina, jog prieš užbaigiant šablonų įkėlimo procesą būtų pateiktas klaidos kodas bei reikalingi originalūs arba pataisyti kodo failai..



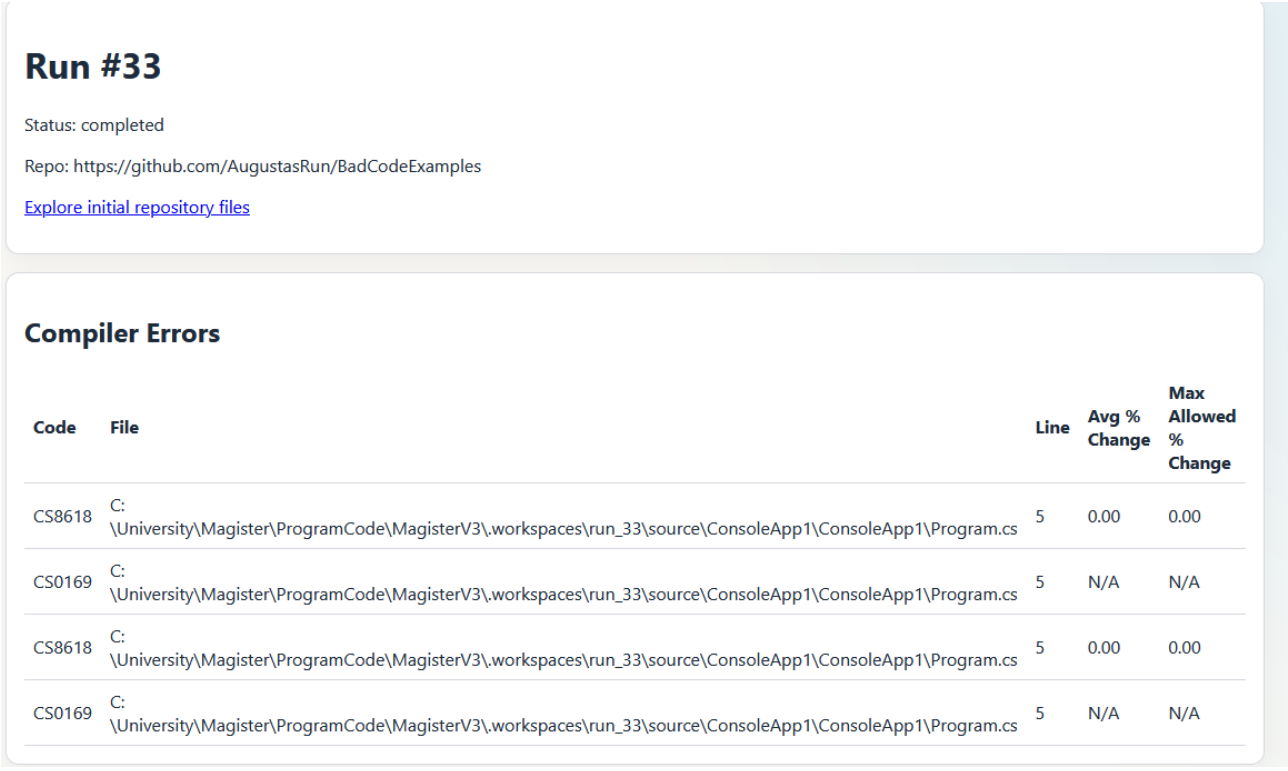
21 pav. Failo validavimo rezultatas

22 pav. matyti, kad kai tas pats failas pateikiamas ir kaip originalas, ir kaip pataisyta versija, sistema apskaičiuoja „grounding“ procentą, lygų **0,00 %**, nes tarp šių dviejų įvesties failų nenustatoma jokių kodo skirtumų. Vidutinis procentinis pokytis ir didžiausias vidutinis procentinis pokytis išlieka **0,00 %**, o tai patvirtina, kad „grounding“ procento skaičiavimas teisingai atspindi pateiktų failų tapatumą.



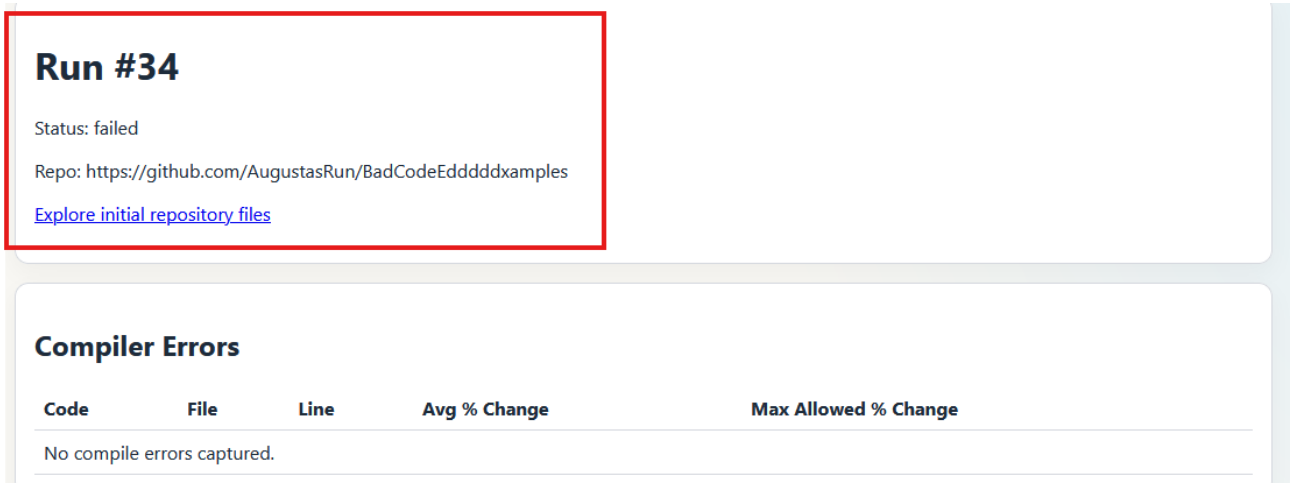
22 pav. Identiškų failų validacijos rezultatas

23 pav. matyti, kad pateikus bent vieną saugyklą analizei atlikti, analizės procesas sėkmingai pradedamas, o jo vykdymui stebėti sukuriamas atskiras vykdymo puslapis. Šiame pavyzdyje rodomas „**Run #33**“, kurio būsena pažymėta kaip „**completed**“, taip pat pateikiamas saugyklos adresas. Rezultatų puslapyje suteikiama tiesioginė prieiga prie pradinių saugyklos failų ir aptiktų kompiliatoriaus klaidų. Tai patvirtina, kad sistema teisingai priima saugyklos įvestį, pradeda analizės darbo eigą ir pateikia vykdymo peržiūrą, kurioje vartotojas gali patikrinti diagnostikos rezultatus bei susijusią pagrindinę informaciją.



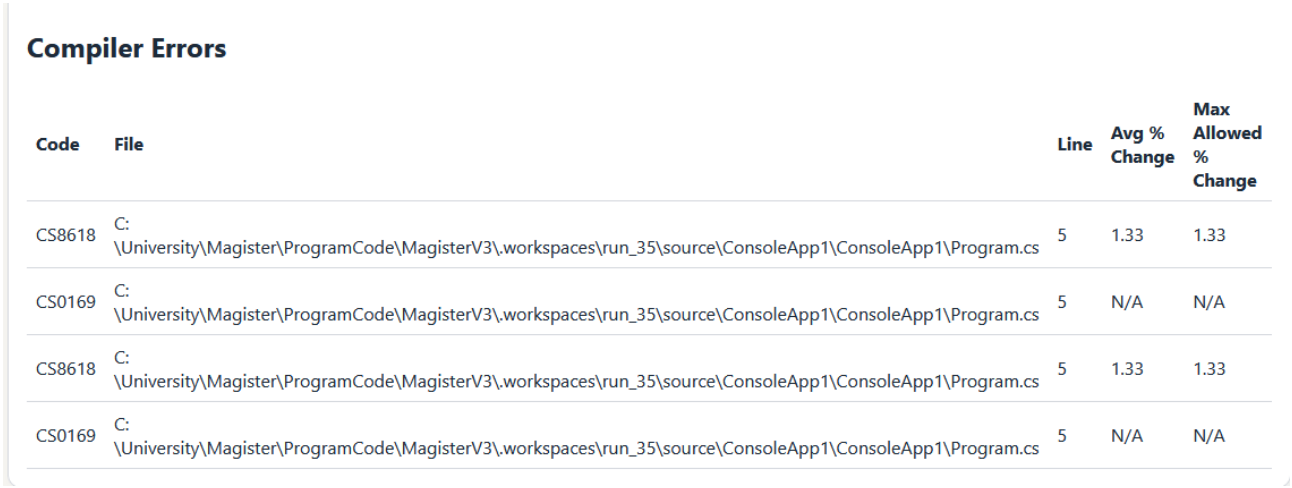
23 pav. Repositorijos analizės testas

24 pav. matyti, kad pateikus neegzistuojančią repozitoriją analizei atlikti, sistema vis tiek sukuria atskirą vykdymo puslapį ir užregistruoja vykdymo bandymą, tačiau vykdymo būsena rodoma kaip „**failed**“. Šiame pavyzdyje „**Run #34**“ pateikiama nurodyta repozitorijos nuoroda, rodoma nesėkminga vykdymo būsena ir tuščia kompiliatoriaus klaidų skiltis su pranešimu „**No compile errors captured.**“ Tai reiškia, kad analizės darbo eiga buvo pradėta, tačiau saugyklos nepavyko gauti arba apdoroti, todėl diagnostiniai duomenys nebuvo sugeneruoti.



24 pav. Nepasiekiamos repozitorijos apdorojimo rezultatas

25 pav. matyti, kad, atlikus saugyklos analizę, sistema sėkmingai surenka ir parodo visus aptiktus kompiliatoriaus klaidų kodus kartu su susijusiais failų keliais, eilučių numeriais ir „grounding“ reikšmėmis. Šiame pavyzdyje analizės rezultatai apima kelis aptiktus diagnostinius pranešimus, pavyzdžiui, **CS8618** ir **CS0169**, susijusius su analizuojamu šaltinio failu. Atitinkamas vidutinis procentinis pokytis ir didžiausias leidžiamas procentinis pokytis rodomi tais atvejais, kai yra prieinami „grounding“ duomenys. Tai patvirtina, kad saugykla buvo sėkmingai sukompiliuota, diagnostinė informacija išgauta teisingai, o nustatytos kompiliatoriaus klaidos pateiktos vartotojui tolesniam LLM pagrįstam apdorojimui.



Compiler Errors

Code	File	Line	Avg % Change	Max Allowed % Change
CS8618	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_35\source\ConsoleApp1\ConsoleApp1\Program.cs	5	1.33	1.33
CS0169	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_35\source\ConsoleApp1\ConsoleApp1\Program.cs	5	N/A	N/A
CS8618	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_35\source\ConsoleApp1\ConsoleApp1\Program.cs	5	1.33	1.33
CS0169	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_35\source\ConsoleApp1\ConsoleApp1\Program.cs	5	N/A	N/A

25 pav. Repositorijos diagnostinių pranešimų gavimas

26 pav. matyti, kad, gavus užklausą dėl saugyklos analizės, keli dideli kalbos modeliai (LLM) nepriklausomai vienas nuo kito sukuria galimus to paties kompiliatoriaus diagnostinio atvejo sprendimus, o jų rezultatai pateikiami reitinguotoje palyginimo lentelėje. Šiame pavyzdyje keli modeliai, įskaitant Claude Sonnet 4.6, GPT-5.4 Nano, GPT-5.4 Mini, GPT-5.4, Claude Opus 4.6 ir Claude Haiku 4.5, sukūrė savo kodo modifikacijas, kurios atsispindi skirtingose pokyčio procentų reikšmėse. Nors visi parodyti bandymai pasiekė sėkmingą kompiliacijos rezultatą, modifikavimo procentinių dydžių skirtumai patvirtina, kad modeliai pasiūlė skirtingus įgyvendinimo būdus tai pačiai problemai išspręsti. Tai sudaro pagrindą tolesniam rezultatų palyginimui ir konsensuso pasirinkimui.

Provider Attempts (Ranked)

Provider	Status	Percent Change	Build Exit	Build Log	Build Files
anthropic[claude-sonnet-4-6]	build_succeeded	2.6316	0	Open build log	Explore files
openai[gpt-5.4-nano]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4-mini]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-opus-4-6]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-haiku-4-5]	build_succeeded	6.3291	0	Open build log	Explore files

26 pav. Kelių LLM pataisymų generavimo užklausos rezultatas

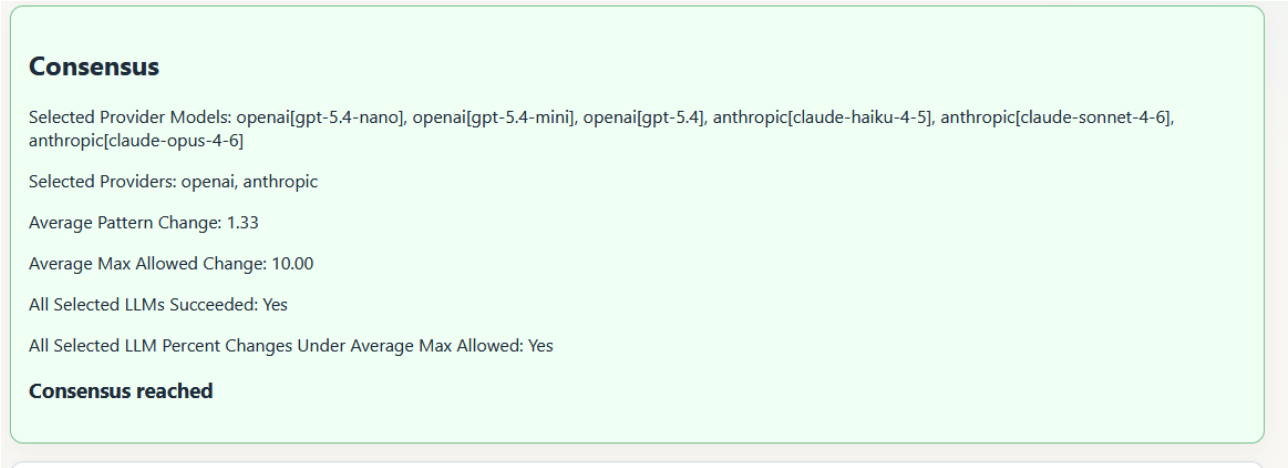
27 pav. matyti, kad kiekvienas LLM sukurtas kodo pasiūlymas atskirai perduodamas kompiliacijos procesui, siekiant patikrinti, ar siūlomas pataisymas leidžia sėkmingai sukompiliuoti projektą. Kiekvieną kartą, kai modelis pateikia pasiūlymą, sistema įrašo kompiliacijos būseną, kompiliacijos baigimo kodą, kompiliacijos žurnalą ir sukurtą failų rinkinį. Tai leidžia kiekvieną galimą sprendimą patikrinti tomis pačiomis techninėmis sąlygomis. Šis procesas patvirtina, kad LLM išvestys nėra priimamos vien kaip tekstiniai pasiūlymai, bet yra automatiškai kompiliuojamos ir vertinamos, siekiant nustatyti, ar jos išsprendžia aptiktą problemą nepažeisdamos kompiliacijos.

Provider Attempts (Ranked)

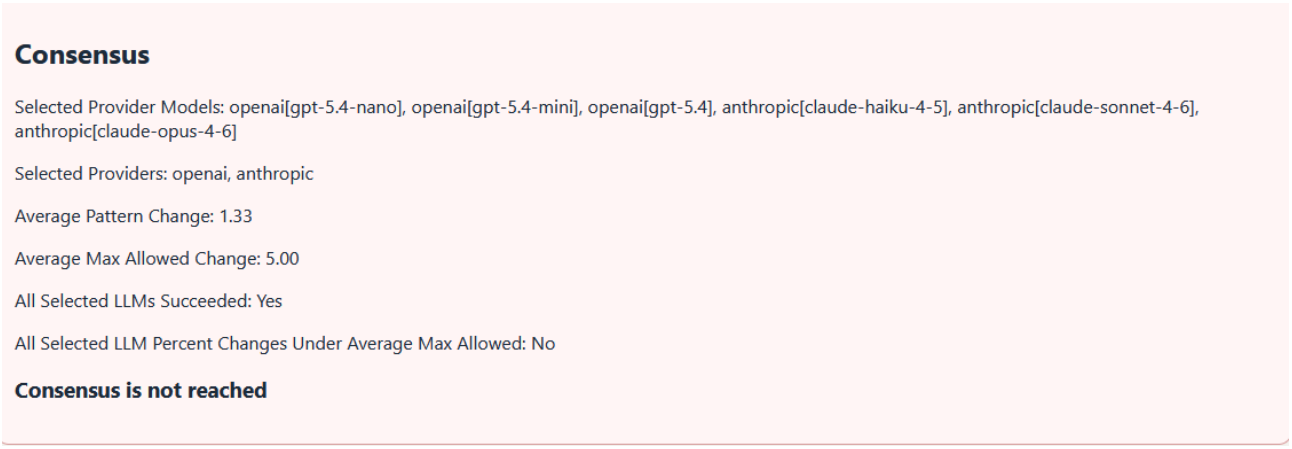
Provider	Status	Percent Change	Build Exit	Build Log	Build Files
anthropic[claude-sonnet-4-6]	build_succeeded	2.6316	0	Open build log	Explore files
openai[gpt-5.4-nano]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4-mini]	build_succeeded	5.0	0	Open build log	Explore files
openai[gpt-5.4]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-opus-4-6]	build_succeeded	5.0	0	Open build log	Explore files
anthropic[claude-haiku-4-5]	build_succeeded	6.3291	0	Open build log	Explore files

27 pav. LLM pasiūlymų rezultatų patikrinimas

Be to, kiekvienas LLM sukurtas pasiūlymas vertinamas pagal tą patį „grounding“ procentą, susijusį su aptikta kompiliatoriaus klaida, taip užtikrinant nuoseklų visų modelių bandymų palyginimo pagrindą. Kadangi „grounding“ reikšmė gaunama iš anksčiau įtraukto istorinio taisymo šablono, skirto konkrečiam klaidos kodui, kiekvienas galimas sprendimas vertinamas ne tik pagal kompiliacijos rezultatą, bet ir pagal tai, kiek jo kodo pakeitimų apimtis atitinka numatomą pakeitimų mastą. Tai užtikrina vertinimo proceso vienodumą ir leidžia lyginti skirtingus LLM rezultatus pagal tą pačią klaidai būdingą koregavimo ribą. Sėkmingo ir nesėkmingo konsensuso rezultatai pateikiami 28 ir 29 paveikslėliuose.



28 pav. Konsensuso pasiekimo rezultatas



29 pav. Konsensuso nepasiekimo rezultatas

4.2. Statinės kodo analizės tobulinimo metodo tyrimas

Šiame poskyryje pateikiamas siūlomo metodo, skirto statinei kodo analizei tobulinti naudojant konsensusu pagrįstą kelių didelių kalbos modelių (LLM) metodiką, vertinimas. Tyrime daugiausia dėmesio skiriama prototipo veiksmingumui apdorojant su saugumu susijusius diagnostinius duomenis, generuojant galimus pataisymus, juos tikrinant perkompiliavimo būdu ir nustatant, ar galima pasiekti patikimą konsensusą. Gauti rezultatai naudojami siekiant įvertinti siūlomo metodo praktinį pritaikomumą, nuoseklumą ir ribotumus automatizuotos statinės kodo analizės tobulinimo kontekste.

4.2.1. Tyrimo metodika

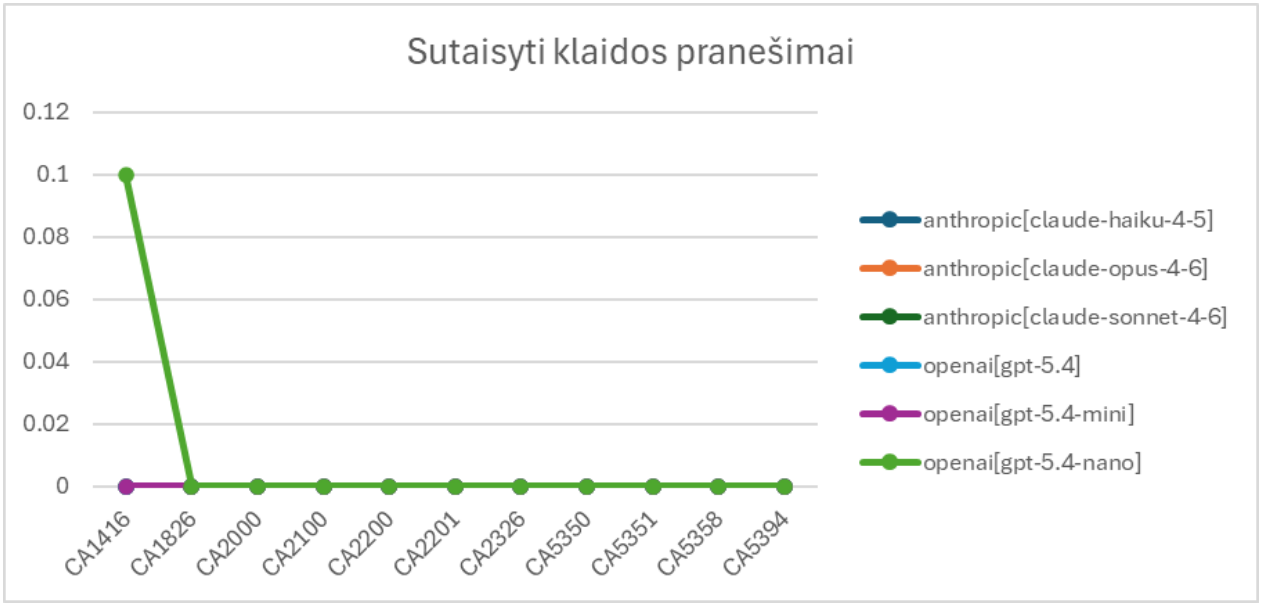
Šiame tyrime kiekvienas atrinktas statinės analizės diagnostinis kodas (žr. 1 Priedas) buvo vertinamas naudojant atskirą nedidelės apimties programinio kodo pavyzdį (žr. 2 Priedas, 3 Priedas), specialiai parengtą atitinkamam įspėjimui sukelti. Toks metodas užtikrino, kad visi dideli kalbos modeliai (LLM) būtų išbandyti palyginamomis ir kontroliuojamomis sąlygomis, kartu sumažinant nesusijusio įgyvendinimo sudėtingumo įtaką gautiems rezultatams. Kompaktiškų, į diagnostiką orientuotų pavyzdžių naudojimas palengvino kiekvieno modelio veiksmingumo vertinimą reaguojant į aiškiai apibrėžtą statinės analizės problemą.

Siekiant padidinti vertinimo patikimumą, kiekvienas LLM kiekvienam diagnostiniam atvejui buvo paleistas dešimt kartų. Pakartotinis vykdymas buvo būtinas, nes LLM generuojami rezultatai gali skirtis skirtinguose vykdymuose net ir tada, kai pateikiami tie patys įvesties duomenys. Todėl galutiniai rezultatai buvo gauti ne iš vieno sugeneruoto pataisymo, o iš vidutinių verčių, apskaičiuotų iš dešimties nepriklausomų kiekvieno modelio iteracijų. Ši vertinimo strategija suteikia stabilesnį pagrindą modelių veiksmingumui palyginti ir leidžia stebėtoms tendencijoms atspindėti tipišką modelio elgseną, o ne pavienius rezultatus.

4.2.2. Sutaisyti klaidų pranešimai

30 pav. diagramoje matyti, kad visi vertinti modeliai pasiekė beveik tobulą diagnostinių problemų išsprendimo lygį visų išbandytų C# diagnostinių kodų atžvilgiu. Beveik kiekvieno diagnostinio kodo atveju vertė išlieka lygi 0, o tai rodo, kad įgyvendinus LLM sukurtus pataisymus pagal šį rodiklį nebuvo pastebėta neišspręstų problemų, susijusių su taisomais pranešimais. Šis rezultatas atitinka darbe pateiktą prototipo vertinimo logiką, pagal kurią LLM išvestys neturėtų būti vertinamos vien kaip tekstiniai pasiūlymai. Vietoj to, jos turi būti pritaikomos, perkompiliuojamos ir patikrinamos kompiliacija pagrįsto patvirtinimo metu. Darbe sukurtas prototipas skirtas patvirtinti sėkmingą kiekvieno sugeneruoto pataisymo kompiliaciją ir nustatyti, ar aptikta diagnostika yra išspręsta nepakenkiant projekto vientisumui. Šis rezultatas yra svarbus įrodant įgyvendintų pataisymų praktinį veiksmingumą.

Vienintelis ryškus nukrypimas susijęs su CA1416, kai openai[gpt-5.4-nano] rodo nedidelę 0,10 vertę, o visi kiti modeliai išlieka ties 0. Šis pastebėjimas rodo, kad net mažiausias iš nagrinėtų modelių pademonstravo bendrą gebėjimą spręsti diagnostines problemas, nors ir pasižymėjo vienu nestabilumo atveju. Grafikas patvirtina išvadą, kad išbandyti LLM buvo labai veiksmingi sprendžiant statinės analizės diagnostikos uždavinius, o modelių skirtumai pagal šį rodiklį buvo minimalūs. Šio darbo kontekste tai rodo, kad vien tikslinės diagnostikos išsprendimo rodiklis yra beveik tobulas. Todėl reikšmingesniam modelių palyginimui daugiau dėmesio turi būti skiriama tokiems veiksniams kaip kompiliacijos sėkmė, naujų problemų atsiradimas, nuokrypis nuo pagrindimo procento ir tai, kiek sugeneruoti pakeitimai atitinka bendruosius konsensuso kriterijus.



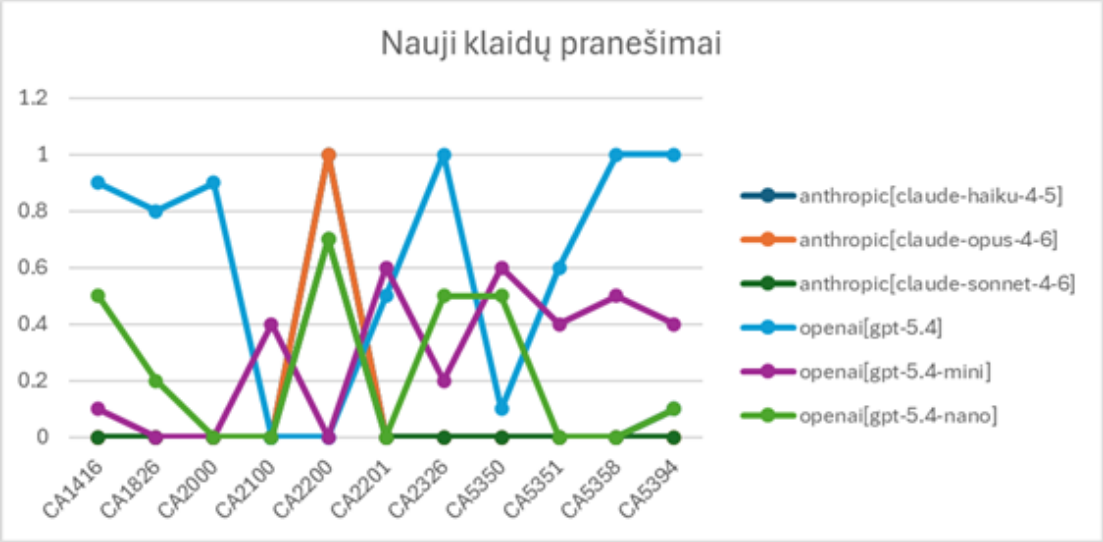
30 pav. Sutaisyti klaidų pranešimai

4.2.3. Nauji klaidų pranešimai

31 pav. diagramoje pavaizduotas naujų diagnostinių pranešimų skaičius, atsiradęs įdiegus kiekvieno modelio sukurtus pataisymus. Palyginti su ankstesniu grafiku, kuriame dauguma modelių veiksmingai išsprendė pradines diagnostines problemas, šiame grafike matyti ryškesni modelių skirtumai. Tai yra svarbi šio darbo vertinimo metodo dalis, nes prototipas ne tik patikrina, ar tikslinė diagnostika buvo ištaisyta, bet ir nustato, ar įgyvendintas pakeitimas lėmė naujų įspėjimų arba klaidų atsiradimą patvirtinimo metu.

Labiausiai išsiskiriantis rezultatas yra tai, kad openai[gpt-5.4] dažniausiai sukelia naujus diagnostinius pranešimus, kurių reikšmės padidėja daugelio diagnostinių kodų atvejais, pavyzdžiui, CA1416, CA1826, CA3526, CA5358 ir CA5394. Šis rezultatas rodo, kad nors modelis veiksmingai išsprendžia tikslinę problemą, jo pakeitimai kartais gali sukelti nenumatytų pasekmių. Priešingai, anthropic[claude-haiku-4-5] reikšmė visų pateiktų diagnostinių kodų atveju išlieka lygi 0, o anthropic[claude-sonnet-4-6] taip pat išlieka stabilus, tik keliais atvejais pastebimas nežymus padidėjimas.

Grafikas rodo, kad vien diagnostinio atvejo išsprendimo nepakanka pataisymo kokybei įvertinti. Nors modelis gali pasiūlyti pradinės problemos sprendimą, jo įgyvendinimas gali sukelti kodo kokybės pablogėjimą dėl naujų diagnostinių problemų atsiradimo. Todėl šie rezultatai patvirtina darbe taikomą konsensusu pagrįstą patvirtinimo metodą, pagal kurį modeliai turėtų būti vertinami ne tik pagal jų gebėjimą spręsti tikslinę problemą, bet ir pagal tokius veiksnius kaip kompiliacijos sėkmė, naujų diagnostinių kodų atsiradimas, nuokrypis nuo pagrindimo procento ir bendras tinkamumas konsensusui.



31 pav. Nauji klaidų pranešimai

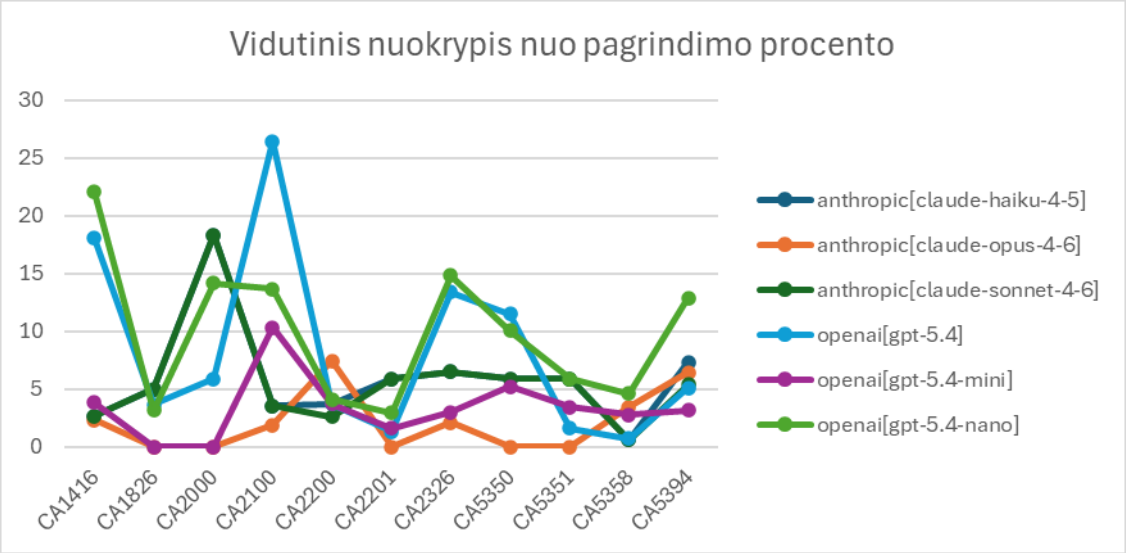
4.2.4. Vidutinis nuokrypis nuo pagrindimo procento

32 pav. diagramoje pateikiamas vidutinis nuokrypis nuo pagrindimo procento kiekvienam modeliui ir diagnostiniam kodui. Šio darbo kontekste šis rodiklis yra svarbus, nes pagrindimo procentas naudojamas kaip istorinis atskaitos taškas, leidžiantis įvertinti tipinę konkrečiai diagnostikai skirto pataisymo apimtį. Mažesnis nuokrypis rodo, kad sugeneruotas pataisymas yra artimesnis istoriniu

požiūriu tikėtinam pokyčio dydžiui, o didesnis nuokrypis reiškia, kad modelis pakeitė per didelę arba per mažą kodo dalį, palyginti su ankstesniais tinkamais pataisymais.

Rezultatai rodo, kad nuokrypiai pasižymi didesniu kintamumu nei diagnostikos išsprendimo rodiklis. Pažymėtina, kad openai[gpt-5.4] ir openai[gpt-5.4-nano] demonstruoja didžiausius svyravimus, ypač CA1416, CA2100, CA3526 ir CA5394 diagnostinių kodų atvejais. Šis rezultatas rodo, kad šie modeliai kartais pateikė sprendimus, kurie išsprendė problemą, tačiau pasižymėjo dideliais nuokrypiais nuo numatyto pagrindimo modelio. Tarp nagrinėjamų modelių anthropic[claude-opus-4-6] pasižymi didžiausiu stabilumu, nes rodo minimalius nuokrypius ir tik nedidelį padidėjimą keliais atvejais. anthropic[claude-sonnet-4-6], anthropic[claude-haiku-4-5]ir openai[gpt-5.4-mini] demonstruoja vidutinį stabilumo lygį: jų rezultatuose matyti tam tikra variacija, tačiau mažiau ekstremalių šuolių.

Grafikas rodo pagrindimo nuokrypio naudą kaip papildomo vertinimo rodiklio. Kadangi dauguma modelių geba išspręsti tikslinę diagnostiką, reikšmingesniu skirtumu tampa tai, ar jų pakeitimai yra proporcingi ir suderinami su ankstesniais pataisymais. Siūlomame konsensuso metode modeliai, kurių pagrindimo nuokrypis yra mažas, laikomi tinkamesniais, nes jie labiau linkę pateikti tikslius pataisymus, o ne pernelyg plačius ar nesuderinamus kodo pakeitimus.



32 pav. Vidutinis nuokrypis nuo pagrindimo procento

4.2.5. Konsensuso atitikimo procentas

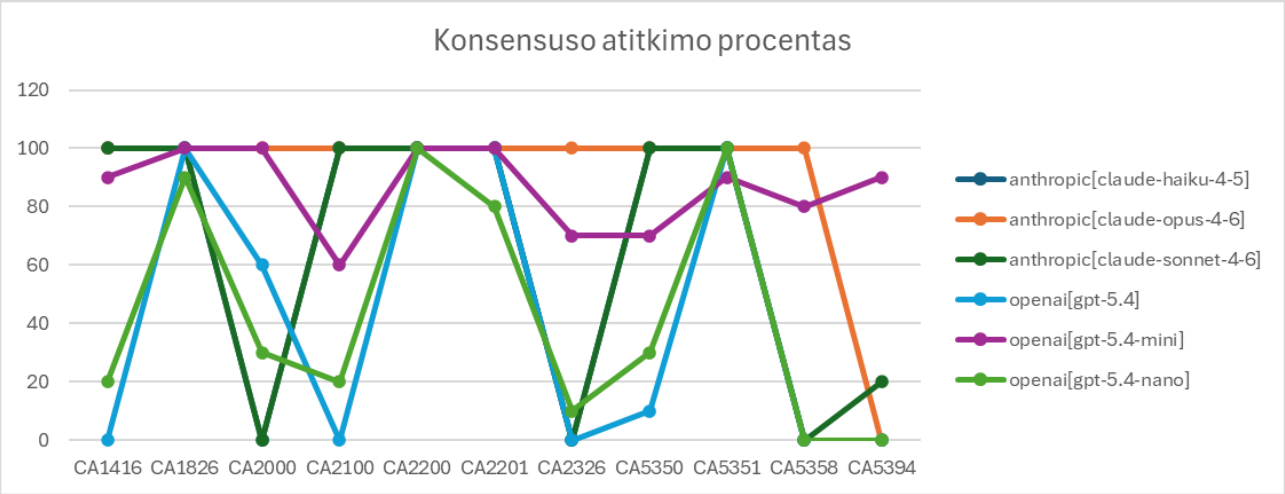
33 pav. Diagrama iliustruoja kiekvieno modelio atitikimo konsensusui procentinę dalį visų tiriamų diagnostinių kodų atžvilgiu. Šis rodiklis rodo, kaip dažnai kiekvieno modelio sukurtas sprendimas atitiko konsensuso kriterijus. Kitaip tariant, sprendimas buvo ne tik techniškai teisingas, bet ir pakankamai artimas numatytam pagrindimo intervalui. Šio darbo kontekste tai yra vienas iš svarbiausių galutinių rodiklių, nes siūlomas metodas sprendimus atrenka remdamasis kompiliacijos patikrinimu, pagrindimo suderinimu ir modelių rezultatų sutapimu.

Rezultatai rodo, kad „Anthropic“ ir „OpenAI“ modeliai demonstruoja didžiausią nuoseklumą konsensuso atitikimo atžvilgiu. Nustatyta, kad „Claude Opus“ daugumos diagnostinių kodų atveju pasiekia 100 % atitikimą ir tik CA5394 atveju rodo žymų sumažėjimą. openai[gpt-5.4-mini] taip pat išlaiko aukštą veiksmingumą beveik visų diagnostinių kodų atveju, paprastai svyruodamas nuo 70 %

iki 100 %. Be to, pastebėta, kad anthropic[claude-sonnet-4-6] modelis daugeliu atvejų rodo gerus rezultatus, tačiau jo veikimas yra mažiau nuoseklus, o CA2000 ir CA5358 atvejais sumažėja iki 0 %.

openai[gpt-5.4] ir openai[gpt-5.4-nano] modeliai pasižymi didesniu rezultatų kintamumu. Tam tikruose diagnostiniuose testuose, pavyzdžiui, CA1826, CA2200, CA2201 ir CA5351, jų konsensuso procentai pasiekė aukštą sutapimo lygį. Tačiau kitų diagnostinių kodų, įskaitant CA1416, CA2100, CA2326 ir CA5358, atvejais konsensuso atitikimas reikšmingai sumažėjo. Šis rezultatas rodo, kad šie modeliai gali generuoti konsensusą atitinkančius sprendimus, tačiau jų veiksmingumas priklauso nuo konkretaus diagnostinio kodo ypatumų.

Grafikas rodo, kad konsensuso vertinimas užtikrina aiškesnį modelių atskyrimą nei vien tik diagnostinio atvejo išsprendimo rodiklis. Nors dauguma modelių gali išspręsti tikslinį įspėjimą, ne visi sprendimai vienodai atitinka numatomą pakeitimo dydį ir konsensuso taisykles. Todėl veiksmingiausi modeliai yra tie, kurie sėkmingą taisymą suderina su stabiliu pagrindimo atitikimu. Tokie modeliai tampa patikimesniais kandidatais galutiniam automatiniam pataisymo pasirinkimo procesui.



33 pav. Konsensuso atitikimo procentas

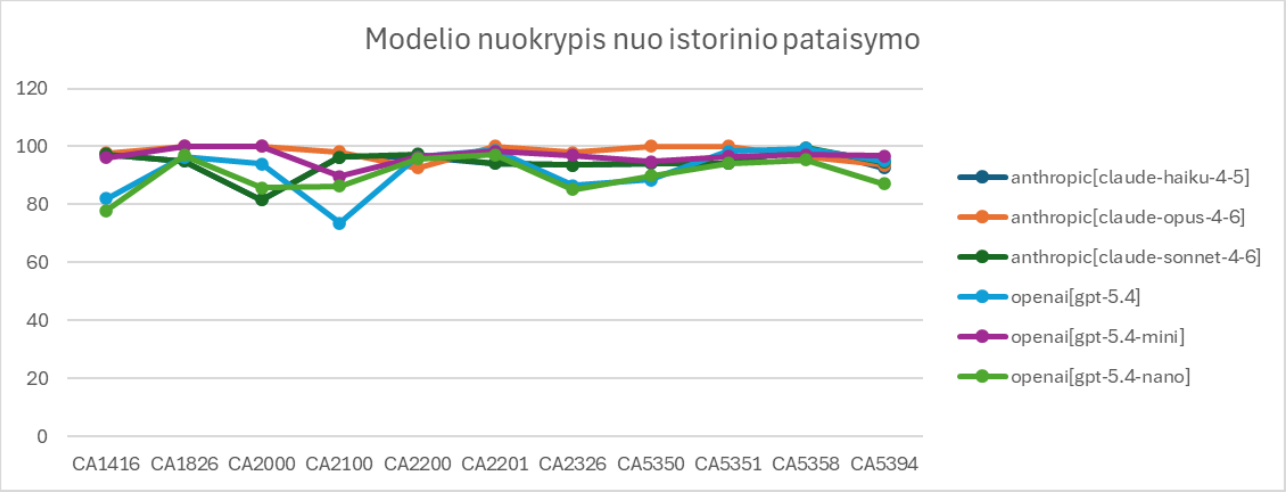
4.2.6. Modelio nuokrypis nuo istorinio pataisymo

34 pav. diagramoje iliustruojamas modelio nukrypimas nuo istorinių taisymų pagal tiriamus diagnostinius kodus. Palyginus šią diagramą su ankstesne nukrypimų diagrama matyti, kad visi modeliai išlieka artimi numatytam istorinių taisymų modeliui. Dauguma verčių sutelkta maždaug nuo 85 % iki 100 %, o tai rodo, kad generuojami taisymai iš esmės gerai atitinka istorinius pagrindinius duomenis, naudotus darbe aprašytoje metodikoje.

Ryškiausias nuoseklumas pastebimas anthropic[claude-opus-4-6] ir openai[gpt-5.4-mini] modelių atveju. Šie modeliai daugumoje diagnostinių kodų išlaiko artumą viršutiniam intervalui. anthropic[claude-haiku-4-5], anthropic[claude-sonnet-4-6] ir openai[gpt-5.4-nano] taip pat rodo nuoseklius rezultatus, nors kai kuriais atvejais jų reikšmių sumažėjimas yra šiek tiek ryškesnis. Labiausiai pastebimas žemiausias taškas matomas openai[gpt-5.4] modelio atveju, kai CA2100 diagnostiniam kodui jo rezultatas rodo didesnį sumažėjimą, palyginti su kitais modeliais.

Grafikas patvirtina hipotezę, kad sugeneruoti pataisymai nėra atsitiktiniai ar pernelyg nutolę nuo istorinių pavyzdžių. Nors modeliai taiko skirtingas įgyvendinimo strategijas, jų atliekami pakeitimai,

palyginti su ankstesniais galiojančiais pataisymais, daugiausia išlieka panašiose ribose. Siūlomo konsensuso metodo kontekste tai yra teigiamas rezultatas, nes jis parodo, kad istorinę bazę galima naudoti kaip praktinį atskaitos tašką lyginant LLM sukurtus kodo pakeitimus ir atrenkant pataisas, kurios labiau atitinka tikėtinus žmogiškojo koregavimo modelius.



34 pav. Modelio nuokrypis nuo istorinio pataisymo

4.2.7. Apibendrinti modelių rezultatai

Toliau pateiktoje 2 lentelė pateikiama vertintų didelių kalbos modelių (LLM) lyginamoji analizė, atsižvelgiant į kompiliacijos sėkmę, naujų problemų skaičių, išspręstas problemas, istorinius nukrypimus ir suderinamumą su konsensusu. Išsamus turimų duomenų vertinimas rodo, kad didesni pažangiausi modeliai pasižymi geriausiais ir stabiliausiais veiklos rezultatais. openai[gpt-5.4] modelis, anthropic[claude-sonnet-4-6] ir anthropic[claude-opus-4-6] demonstruoja aukštą kompiliacijos sėkmės lygį, veiksmingai išsprendžia nemažą problemų skaičių, sukelia tik ribotą naujų problemų skaičių ir išlaiko artumą istoriniams koregavimo modeliams. Todėl šie modeliai pasižymi aukštu konsensuso lygiu, o tai rodo, kad jų generuojami pataisymai yra nuoseklesni, techniškai pagrįsti ir suderinti su numatytu pakeitimų mastu.

Mažesni modeliai pasižymi didesniu rezultatų kintamumu. Tiek openai[gpt-5.4-nano], tiek openai[gpt-5.4-mini] parodė gebėjimą spręsti daugelį problemų, tačiau jų vidutinis kompiliacijos sėkmės lygis ir mažesnis konsensuso atitikimas rodo mažiau stabilų elgesį pakartotinių vykdymų metu. openai[gpt-5.4-mini] yra veiksmingesnis už openai[gpt-5.4-nano], nes rečiau sukelia naujas problemas ir pasižymi mažesniu istoriniu nuokrypiu. Dėl šių savybių jo konsensuso atitikimas gali būti vertinamas kaip vidutinis. anthropic[claude-haiku-4-5] modelis apskritai rodo mažiausiai patikimą veiksmingumą. Nors jis geba išspręsti nemažą dalį problemų, modelis pasižymi žemu arba vidutiniu kompiliacijos sėkmės lygiu, sukelia daugiau naujų problemų ir labiau nukrypsta nuo istorinių koregavimo modelių. Šis rezultatas rodo, kad nors mažesni ar lengvesni modeliai gali pasiūlyti perspektyvius sprendimus, jų rezultatai turi būti griežčiau patvirtinami prieš juos laikant patikimais pataisymais.

2 lentelė. Prototipe naudotų modelių apibendrinimas

Modelis	**Kompiliacijos sėkmė**	**Naujų diagnostinių pranešimų kiekis**	**Sutaisytų diagnostinių pranešimų kiekis**	**Vidutinis nuokrypis nuo pagrindimo procento**	**Konsensuso atitikimo procentas**
gpt-5.4-nano	vidutinis	vidutinis	aukštas	vidutinis	žemas
gpt-5.4-mini	vidutinis	žemas	aukštas	žemas	vidutinis
gpt-5.4	aukštas	vidutinis	aukštas	vidutinis	vidutinis
claude-haiku-4.5	aukštas	žemas	aukštas	žemas	žemas
claude-sonnet-4.6	aukštas	žemas	aukštas	žemas	aukštas
claude-opus-4.6	aukštas	žemas	aukštas	žemas	aukštas

4.3. Statinės kodo analizės tobulinimo prototipo metodo tyrimo apibendrinimas ir išvados

Siūlomas statinės kodo analizės tobulinimo metodas buvo įvertintas prototipinėje aplinkoje. Gauti rezultatai parodė, kad metodas sėkmingai įgyvendintas ir tinkamas taikyti statinės kodo analizės taisymo užduotims. Atliktas išsamus empirinių duomenų ir esamų tyrimų vertinimas, kurio pagrindu daroma šios išvados:

1. Prototipas buvo sėkmingai įgyvendintas ir eksperimentų metu veikė tinkamai.
2. Kelių didelių kalbos modelių (LLM) naudojimas sudarė sąlygas alternatyvių sprendimų generavimą ir palyginimą sprendžiant tą pačią diagnostinę problemą.
3. Pažangiausi modeliai nuosekliai pateikė patikimesnius rezultatus, o mažesni modeliai dažniau patirdavo kompiliacijos nesėkmių.
4. Kompiliacija pagrįsto patvirtinimo ir pagrindimo palyginimo metodų naudojimas palengvino objektyvų patikimesnių ir mažiau patikimų taisymo variantų nustatymą.
5. Konsensuso pagrįstas metodas parodė potencialą pagerinti statinę kodo analizę, nors jo veiksmingumas priklauso nuo pasirinktų modelių kokybės.

Išvados

1. Išsami programinės įrangos kodo problemų ir jų priežasčių analizė parodė, kad šiuolaikinės programinės įrangos sistemos tampa vis sudėtingesnės dėl didėjančių kodų bazių, įvairių programavimo paradigmų taikymo, išorinių priklausomybių naudojimo ir dirbtinio intelekto pagalba generuojamo kodo integravimo. Šis sudėtingumas didina defektų, saugumo pažeidžiamumų ir priežiūros problemų tikimybę. Be to, daugelis dažnų klaidų vis dar reikalauja sistemingo aptikimo, patikrinimo ir ištaisymo.
2. Statinės kodo analizės technikų, metodų ir įrankių analizė parodė, kad statinė analizė yra svarbus kokybės užtikrinimo metodas ankstyvojoje programinės įrangos kūrimo stadijoje. Nustatyta, kad leksinės, sintaksinės, semantinės, valdymo srauto ir šablonais pagrįstos analizės technikos padeda identifikuoti defektus nevykdant programos. Tačiau esamų įrankių veiksmingumas priklauso nuo jų taisyklių suderinimo su žinomais defektų šablonais. Sudėtingesnių semantinių ar nuo konteksto priklausančių problemų identifikavimas išlieka sudėtingas ir ne visada patikimas.
3. Problemų, su kuriomis susiduria statinės kodo analizės įrankiai, tyrimas parodė, kad šiuos įrankius riboja keli veiksniai: klaidingi teigiami rezultatai, klaidingi neigiami rezultatai, neišsamus semantinis supratimas, mastelio keitimo problemos ir sunkumai analizuojant didelius ar heterogeniškus kodų rinkinius. Todėl statinės analizės rezultatai dažnai reikalauja tolesnio patvirtinimo, nes vien diagnostinio pranešimo buvimas neužtikrina siūlomo pataisymo teisingumo, proporcingumo ar kompiliuojamumo.
4. Daugiaagentinių ir konsensusu pagrįstų sistemų analizė parodė, kad kelių nepriklausomų agentų arba didelių kalbos modelių naudojimas gali padidinti automatizuoto kodo taisymo patikimumą. Nustatyta, kad konsensusu pagrįstas metodas sumažina priklausomybę nuo vieno modelio, sudaro sąlygas palyginti alternatyvius sprendimus tai pačiai diagnostinei problemai spręsti ir padeda atmesti silpnesnius, pernelyg plačius arba techniškai neteisingus pasiūlymus.
5. Šiame darbe pasiūlytas statinės kodo analizės tobulinimo metodas, pagrįstas keliais dideliais kalbos modeliais, istoriniu pagrindimo procentu, kompiliacijos pagrindu atliekamu patvirtinimu ir konsensuso vertinimu. Siūlomas metodas galimus pataisymus vertina ne tik pagal sugeneruotą tekstą, bet ir pagal objektyvius kriterijus, įskaitant sėkmingą kompiliaciją, tikslinės diagnostikos išsprendimą, naujų problemų nebuvimą ir nuokrypį nuo istoriškai stebimos pataisymo apimties.
6. Įgyvendintas prototipas ir jo eksperimentinis vertinimas parodė siūlomo metodo praktinį pritaikomumą statinės kodo analizės tobulinimo užduotims. Prototipas veiksmingai rinko diagnostinius duomenis, generavo alternatyvius pataisymus naudojant kelis LLM, perkompiliavo galimus sprendimus ir vertino konsensuso sąlygų įvykdymą. Rezultatai parodė, kad pažangiausi modeliai pateikė stabilesnius, kompiliuojamus ir istoriniu požiūriu suderintus sprendimus, o mažesni modeliai dažniau sukeldavo kompiliacijos nesėkmes arba papildomų problemų atsiradimą. Taigi siūlomas metodas sprendžia pagrindinę darbe nustatytą problemą, nors jo veiksmingumas priklauso nuo pasirinktų modelių kokybės ir patvirtinimo kriterijų griežtumo.
7. Tolesniuose tyrimuose metodą būtų tikslinga išbandyti su didesnėmis kodų bazėmis, įvairesniais diagnostiniais pranešimais ir platesniu modelių rinkiniu. Tai leistų tiksliau įvertinti metodo plečiamumą, stabilumą bei pritaikomumą didesnės apimties programinės įrangos projektuose. Taip pat būtų naudinga toliau tobulinti konsensuso kriterijus, kad galutinių pataisymų atranka būtų dar patikimesnė.

Literatūros sąrašas

1. J. Salet, K. Rakholia, O. Rahul, K. Jignesh, and K. Jay, “Navigating the Evolution: Current Trends and Future Directions in Programming Languages,” *Int. J. Innov. Res. Comput. Sci. Technol.*, vol. 12, no. 4, pp. 43–46, Jul. 2024, doi: 10.55524/ijircst.2024.12.4.7.
2. X. Wu, S. Wu, J. Wu, L. Feng, and K. C. Tan, “Evolutionary Computation in the Era of Large Language Model: Survey and Roadmap,” May 29, 2024, *arXiv*: arXiv:2401.10034. doi: 10.48550/arXiv.2401.10034.
3. “(PDF) A Literature Survey of Complexity Metrics for Object-Oriented Programs,” *ResearchGate*, doi: 10.7753/IJSEA1105.1003.
4. H. Chen and M. A. Babar, “Security for Machine Learning-based Software Systems: A Survey of Threats, Practices, and Challenges,” *ACM Comput Surv*, vol. 56, no. 6, p. 151:1-151:38, Feb. 2024, doi: 10.1145/3638531.
5. S. Dikici and T. T. Bilgin, “Advancements in automated program repair: a comprehensive review,” *Knowl. Inf. Syst.*, vol. 67, no. 6, pp. 4737–4783, Jun. 2025, doi: 10.1007/s10115-025-02383-9.
6. Q. Zhang *et al.*, “A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair,” Apr. 17, 2024, *arXiv*: arXiv:2310.08879. doi: 10.48550/arXiv.2310.08879.
7. Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, “Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts,” *Symb. Value-Flow Static Anal. Deep Precise Complete Model. Ethereum Smart Contracts Artifact*, vol. 5, no. OOPSLA, p. 163:1-163:30, Oct. 2021, doi: 10.1145/3485540.
8. P. Diamantakis, T. Avgerinos, and Y. Smaragdakis, “Desyan: A Platform for Seamless Value-Flow and Symbolic Analysis,” Aug. 01, 2025, *arXiv*: arXiv:2508.00508. doi: 10.48550/arXiv.2508.00508.
9. “Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites - Amankwah - 2023 - Software: Practice and Experience - Wiley Online Library.” Accessed: Mar. 10, 2026. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/spe.3181
10. “Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java | Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.” Accessed: Mar. 10, 2026. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616262
11. J. Mertz and I. Nunes, “Tigris: A DSL and framework for monitoring software systems at runtime,” *J. Syst. Softw.*, vol. 177, p. 110963, Jul. 2021, doi: 10.1016/j.jss.2021.110963.
12. F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, and A. A. H. Elnour, “Malware Detection Issues, Challenges, and Future Directions: A Survey,” *Appl. Sci.*, vol. 12, no. 17, p. 8482, Jan. 2022, doi: 10.3390/app12178482.
13. G. Horvath, R. Kovacs, R. Szalay, and Z. Porkolab, “Implementing and Executing Static Analysis Using LLVM and CodeChecker,” Aug. 10, 2024, *arXiv*: arXiv:2408.05657. doi: 10.48550/arXiv.2408.05657.
14. D. Gomes, E. Felix, F. Aires, and M. Vieira, “Static Code Analysis for IoT Security: A Systematic Literature Review,” *ACM Comput Surv*, vol. 58, no. 3, p. 65:1-65:47, Sep. 2025, doi: 10.1145/3745019.
15. M. Schiewe, J. Curtis, V. Bushong, and T. Cerny, “Advancing Static Code Analysis With Language-Agnostic Component Identification,” *IEEE Access*, vol. 10, pp. 30743–30761, 2022, doi: 10.1109/ACCESS.2022.3160485.
16. H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach,” *Enhancing Static Anal. Pract. Bug Detect. LLM-Integr.*

Approach Artifact, vol. 8, no. OOPSLA1, p. 111:474-111:499, Apr. 2024, doi: 10.1145/3649828.

17. D. Guo *et al.*, “GRAPHCODEBERT: PRE-TRAINING CODE REPRESEN- TATIONS WITH DATA FLOW,” 2021.
18. P. Yao, J. Zhou, X. Xiao, Q. Shi, R. Wu, and C. Zhang, “Falcon: A Fused Approach to Path-Sensitive Sparse Data Dependence Analysis,” *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 567–592, Jun. 2024, doi: 10.1145/3656400.
19. J. Späth, K. Ali, and E. Bodden, “Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 1–29, Jan. 2019, doi: 10.1145/3290361.
20. Y. Zhang and D. Liu, “Toward Vulnerability Detection for Ethereum Smart Contracts Using Graph-Matching Network,” *Future Internet*, vol. 14, no. 11, p. 326, Nov. 2022, doi: 10.3390/fi14110326.
21. Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, “Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection,” *IEEE Trans. Knowl. Data Eng.*, pp. 1–1, 2021, doi: 10.1109/TKDE.2021.3095196.
22. L.-A. Daniel, S. Bardin, and T. Rezk, “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level,” in *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2020, pp. 1021–1038. doi: 10.1109/SP40000.2020.00074.
23. F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, “EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, Madrid, Spain: IEEE, May 2021, pp. 127–137. doi: 10.1109/ICPC52881.2021.00021.
24. K. Kuszczyński and M. Walkowski, “Comparative Analysis of Open-Source Tools for Conducting Static Code Analysis,” *Sensors*, vol. 23, no. 18, p. 7978, Jan. 2023, doi: 10.3390/s23187978.
25. D. Gnieciak and T. Szandala, “Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection,” *IEEE Access*, vol. 13, pp. 198410–198422, 2025, doi: 10.1109/ACCESS.2025.3635168.
26. [J. He, C. Treude, and D. Lo, “LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead,” *ACM Trans Softw Eng Methodol*, vol. 34, no. 5, p. 124:1-124:30, May 2025, doi: 10.1145/3712003.
27. L. B. Kaesberg, J. Becker, J. P. Wahle, T. Ruas, and B. Gipp, “Voting or Consensus? Decision-Making in Multi-Agent Debate,” in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds., Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 11640–11671. doi: 10.18653/v1/2025.findings-acl.606.
28. D. Bach, “PBFT-Backed Semantic Voting for Multi-Agent Memory Pruning,” Jun. 24, 2025, *arXiv*: arXiv:2506.17338. doi: 10.48550/arXiv.2506.17338.
29. E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati, and G. Uddin, “An Empirical Study on Bug Severity Estimation using Source Code Metrics and Static Analysis,” Aug. 02, 2024, *arXiv*: arXiv:2206.12927. doi: 10.48550/arXiv.2206.12927.
30. H. Wang *et al.*, “Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, in ICSE ’24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. doi: 10.1145/3597503.3639212.

Priedai

1 priedas. Diagnostiniai kodai

CA1416 – .NET analizatoriaus diagnostinis kodas, susijęs su platformos suderinamumo patikromis;
CA1826 – .NET analizatoriaus diagnostinis kodas, susijęs su kolekcijų metodų efektyvesniu naudojimu;
CA2000 – .NET analizatoriaus diagnostinis kodas, susijęs su objektų, naudojančių resursus, tinkamu atlaisvinimu;
CA2100 – .NET analizatoriaus diagnostinis kodas, susijęs su SQL užklausų saugumo rizikomis;
CA2200 – .NET analizatoriaus diagnostinis kodas, susijęs su išimčių pakartotiniu išmetimu;
CA2201 – .NET analizatoriaus diagnostinis kodas, susijęs su rezervuotų išimčių tipų naudojimu;
CA2326 – .NET analizatoriaus diagnostinis kodas, susijęs su nesaugaus JSON serializavimo konfigūracija;
CA5350 – .NET analizatoriaus diagnostinis kodas, susijęs su silpnų kriptografinių algoritmų naudojimu;
CA5351 – .NET analizatoriaus diagnostinis kodas, susijęs su silpnų kriptografinių algoritmų naudojimu;
CA5358 – .NET analizatoriaus diagnostinis kodas, susijęs su nesaugių kriptografinių algoritmų naudojimu;
CA5394 – .NET analizatoriaus diagnostinis kodas, susijęs su nesaugiu atsitiktinių reikšmių generavimu;
CS0169 – C# kompiliatoriaus diagnostinis kodas, žymintis deklaruotą, bet nenaudojamą lauką;
CS8618 – C# kompiliatoriaus diagnostinis kodas, susijęs su neinicijuotu ne „null“ reikšmės lauku arba savybe.

2 priedas. Programinio kodo pavyzdžiai generuojantys pranešimus naudoti tyrime

CA1416:

```
using Microsoft.Win32;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }

      public static string? ReadRegistry()
      {
         return Registry.GetValue(
            @"HKEY_LOCAL_MACHINE\Software\MyApp",
            "InstallPath",
            null)?.ToString();
      }
   }
}
```

CA1826:

```
namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
         var temp = new C();
      }
   }

   internal sealed class C
   {
      public static void M(IReadOnlyList<string> list)
      {
         Console.Write(list.First());
         Console.Write(list.Last());
         Console.Write(list.Count());
      }
   }
}
```

CA2000:

```
namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }

      public static string Read(string path)
      {
         var reader = new StreamReader(path);
         return reader.ReadToEnd();
      }
   }
}
```

CA2100:

```
using Microsoft.Data.SqlClient;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
```

```
        UnsafeQuery("test", "test", "test");
    }

    public static object UnsafeQuery(string connection, string name, string password)
    {
        SqlConnection someConnection = new SqlConnection(connection);
        using SqlCommand someCommand = new SqlCommand();
        someCommand.Connection = someConnection;

        someCommand.CommandText = "SELECT AccountNumber FROM Users " +
          "WHERE Username='" + name +
          "' AND Password='" + password + "'";

        someConnection.Open();
        object accountNumber = someCommand.ExecuteScalar();
        someConnection.Close();
        return accountNumber;
    }
  }
}
```

CA2200:

```
namespace ConsoleApp1
{
  internal sealed class Program
  {
    static async Task Main(string[] args)
    {
      var temp = new ExceptionExample();
    }

    internal sealed class ExceptionExample
    {
      public static void Run()
      {
        try
        {
          DoWork();
        }
        catch (Exception ex)
        {
          throw ex;
        }
      }

      private static void DoWork() => throw new InvalidOperationException();
    }
```

```
    }
}
```

CA2201:

```
namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
        }

        public static void Import(string path)
        {
            if (string.IsNullOrWhiteSpace(path))
            {
                throw new Exception("Path is required.");
            }
        }
    }
}
```

CA2326:

```
using Newtonsoft.Json;

namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
            var temp = new ExampleClass();
        }
    }

    internal sealed class ExampleClass
    {
        public JsonSerializerSettings Settings { get; }

        public ExampleClass()
        {
            Settings = new JsonSerializerSettings();
            Settings.TypeNameHandling = TypeNameHandling.All;
        }
    }
}
```

CA5350:

```
using System.Security.Cryptography;
```

```
namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
         UnsafeQuery("test", "test", "test");
      }

      public static void UnsafeQuery(string connection, string name, string password)
      {
         using var hashAlg = SHA1.Create();
      }
   }
}
```

CA5351:

```
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }

      public static byte[] Hash(string input)
      {
         return MD5.HashData(Encoding.UTF8.GetBytes(input));
      }
   }
}
```

CA5358:

```
using System.Security.Cryptography;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }

      public static Aes Create()
```

```
        {
            var aes = Aes.Create();
            aes.Mode = CipherMode.ECB;
            return aes;
        }
    }
}
```

CA5394:

```
namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
        }

        public static int GenerateCode()
        {
            var random = new Random();
            return random.Next(100000, 999999);
        }
    }
}
```

3 priedas. Programinio kodo pavyzdžiai su sutaisytais klaidos pranešimais naudoti tyrime

CA1416:

```
using Microsoft.Win32;

namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
        }

        public static string? ReadRegistry()
        {
            if (!OperatingSystem.IsWindows())
                return null;

            return Registry.GetValue(
                @"HKEY_LOCAL_MACHINE\Software\MyApp",
                "InstallPath",
                null)?.ToString();
        }
```

```
    }
}
```
CA1826:
```
namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
            var temp = new C();
        }
    }

    internal sealed class C
    {
        public static void M(IReadOnlyList<string> list)
        {
            Console.Write(list[0]);
            Console.Write(list[list.Count - 1]);
            Console.Write(list.Count);
        }
    }
}
```
CA2000:
```
namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
        }

        public static string Read(string path)
        {
            using var reader = new StreamReader(path);
            return reader.ReadToEnd();
        }
    }
}
```
CA2100:
```
using Microsoft.Data.SqlClient;

namespace ConsoleApp1
{
    internal sealed class Program
    {
```

```
        static async Task Main(string[] args)
        {
            UnsafeQuery("test", "test", "test");
        }

        public static object UnsafeQuery(string connection, string name, string password)
        {
            SqlConnection someConnection = new SqlConnection(connection);
            using SqlCommand someCommand = new SqlCommand();
            someCommand.Connection = someConnection;

                someCommand.CommandText = "SELECT AccountNumber FROM Users WHERE
Username=@name AND Password=@password";
            someCommand.Parameters.AddWithValue("@name", name);
            someCommand.Parameters.AddWithValue("@password", password);

            someConnection.Open();
            object accountNumber = someCommand.ExecuteScalar();
            someConnection.Close();
            return accountNumber;
        }
    }
}
```

CA2200:

```
namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
            var temp = new ExceptionExample();
        }

        internal sealed class ExceptionExample
        {
            public static void Run()
            {
                try
                {
                    DoWork();
                }
                catch
                {
                    throw;
                }
            }
```

```
        private static void DoWork() => throw new InvalidOperationException();
    }
  }
}
```

CA2201:

```
namespace ConsoleApp1
{
  internal sealed class Program
  {
    static async Task Main(string[] args)
    {
    }

    public static void Import(string path)
    {
      if (string.IsNullOrWhiteSpace(path))
      {
        throw new ArgumentException("Path is required.", nameof(path));
      }
    }
  }
}
```

CA2326:

```
using Newtonsoft.Json;

namespace ConsoleApp1
{
  internal sealed class Program
  {
    static async Task Main(string[] args)
    {
      var temp = new ExampleClass();
    }
  }

  internal sealed class ExampleClass
  {
    public JsonSerializerSettings Settings { get; }

    public ExampleClass()
    {
      Settings = new JsonSerializerSettings();
    }
  }
}
```

CA5350:

```
using System.Security.Cryptography;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
         UnsafeQuery("test", "test", "test");
      }

      public static void UnsafeQuery(string connection, string name, string password)
      {
         using var hashAlg = SHA256.Create();
      }
   }
}
```

CA5351:

```
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }

      public static byte[] Hash(string input)
      {
         return SHA256.HashData(Encoding.UTF8.GetBytes(input));
      }
   }
}
```

CA5358:

```
using System.Security.Cryptography;

namespace ConsoleApp1
{
   internal sealed class Program
   {
      static async Task Main(string[] args)
      {
      }
```

```
        public static Aes Create()
        {
            var aes = Aes.Create();
            aes.Mode = CipherMode.CBC;
            aes.GenerateIV();
            aes.GenerateKey();
            return aes;
        }
    }
}
```

CA5394:

```
using System.Security.Cryptography;

namespace ConsoleApp1
{
    internal sealed class Program
    {
        static async Task Main(string[] args)
        {
        }

        public static int GenerateCode()
        {
            return RandomNumberGenerator.GetInt32(100000, 1000000);
        }
    }
}
```