UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

TEZĂ DE DOCTORAT - REZUMAT

Asistarea mentenabilității, fiabilității și securității software-ului prin analiză statică

Conducător științific
Prof. Dr. Simona Motogna

Student doctorand
Liviu Marian Berciu

2026

Mulţumiri

În sfârșit, după un drum lung, plin de provocări și imprevizibilitate, această lucrare poate fi considerată încheiată. Și, deși acesta este un moment de mare bucurie, este, de asemenea, un moment pentru a fi recunoscător celor care au făcut posibilă această lucrare.

Doresc să îmi exprim recunoștința sinceră față de coordonatoarea mea, doamna profesor Simona Motogna, pentru dedicarea sa și contribuția inestimabilă la această lucrare. A fost o sursă constantă de îndrumare și sprijin, în special în momentele critice de-a lungul acestui proces. A știut cum să mă consoleze când o lucrare a fost respinsă și cum să mă repună pe drumul cel bun când mă simțeam pierdut. Etica ei de muncă și disciplina au fost cu adevărat admirabile și un exemplu de durată pentru mine, determinându-mă să fiu mai profesionist și mai muncitor, atât în viața mea academică, cât și în cea profesională.

Sunt recunoscător comisiei mele de îndrumare (Doamna Prof. Laura Dioșan, Doamna Conf. Camelia Șerban și Domnul Conf. Arthur Molnar) pentru ghidajul și feedback-ul lor, care au ajutat la conturarea cercetării mele și mi-au oferit noi direcții științifice pe care să lucrez. Un alt pilon important care m-a ajutat să termin această teză sunt colegii mei de doctorat (Vasilica Moldovan, Patcaș Rareș, Oskar Picuș), care au fost o sursă de inspirație și încredere și mi-au arătat cum se simte adevărata colaborare. De asemenea, mulțumesc comisiei externe pentru feedback-ul și contribuția lor, care m-au ajutat să adaug ultimele retușuri acestei teze.

Persoanele care m-au ghidat pe calea mediului academic sunt părinții mei, Liviu și Nicoleta, cărora le sunt extrem de recunoscător. Ei m-au învățat disciplina, cum să merg mai departe până ajung la linia de sosire și cum să nu renunț niciodată la obiectivele mele. Pentru acest lucru, le sunt veșnic recunoscător. Vreau, de asemenea, să o menționez pe sora mea, care a fost un pilon de sprijin în călătoria mea.

Pentru cea mai importantă persoană din viața mea, soția mea Alice, care m-a încurajat când a fost greu, mi-a ascultat plângerile în tăcere și m-a pus cu "capul pe umeri" când a fost momentul să mă adun și să fiu bărbat: îți mulțumesc din toată inima pentru dragostea și încurajarea ta, pentru răbdarea și încrederea ta.

În cele din urmă, pentru băiețelul meu Liam, care mi-a oferit determinarea de a termina această teză înainte ca el să se nască.

Rezumat

Instrumentele de analiză statică a codului, precum SonarQube, sunt utilizate pe scară largă pentru a identifica problemele de mentenabilitate, fiabilitate și securitate, inclusiv problemele de tip code smell, erorile (bugs) și vulnerabilitățile. Cu toate acestea, caracterizarea modului în care aceste probleme evoluează în timp în sistemele open-source, a modului în care acestea se raportează la activitatea dezvoltatorilor și a modului în care pot fi detectate și prioritizate mai eficient în practică rămâne o provocare. Această teză investighează aceste întrebări, cu un accent principal pe proiectele open-source în Python.

Pentru a facilita studiile empirice, sunt introduse un instrument de extragere a datelor din re-

pozitoare (PyDs Builder) și un set de date open-source (Python Software Quality Dataset) pentru a colecta și integra artefacte legate de calitate din instrumentele de control al versiunilor și de analiză statică. Utilizând aceste resurse, este condusă o analiză longitudinală a problemelor raportate de SonarQube, arătând că problemele de tip code smell reprezintă majoritatea problemelor raportate și persistă de-a lungul lansărilor (releases), în timp ce erorile și vulnerabilitățile sunt mai puțin frecvente și distribuite mult mai neuniform între proiecte.

Comportamentul dezvoltatorilor este examinat prin analize axate pe commit-uri. Mesajele commit-urilor sunt clasificate în categoriile Specificației Convenționale a Commit-urilor (Conventional Commit Specification) utilizând modele de limbaj de mari dimensiuni, susținând analizele modului în care diferitele tipuri de activități de dezvoltare se corelează cu problemele observate. În plus, este analizată frecvența în rafală a commit-urilor (commit burstiness) pentru a evalua dacă perioadele de dezvoltare rapidă sunt asociate cu schimbări în prevalența problemelor. Transferabilitatea interlingvistică este explorată prin compararea Arborilor Sintactici Abstracți (*Abstract Syntax Trees* - AST) din Python și JavaScript pentru zece reguli comune SonarQube, folosind atât măsuri de similaritate bazate pe procesarea limbajului natural (NLP), cât și distanța de editare a arborilor (tree-edit distance); rezultatele indică faptul că o similaritate semnificativă se limitează în mare măsură la reguli simple, independente de limbaj, și că adesea AST-urile nu sunt suficiente prin ele însele pentru a susține transferul regulilor de la un limbaj la altul.

În cele din urmă, teza sintetizează lucrările anterioare printr-o recenzie sistematică a literaturii privind inteligența artificială în refactorizare (și viceversa), investighează predicția refactorizării pe baza problemelor de datorie tehnică (*technical debt*) și evaluează modelele de învățare automată pentru detectarea problemelor de tip code smell în Python, unde datele extrem de dezechilibrate joacă un rol central în performanța modelelor.

Per ansamblu, teza contribuie cu seturi de date, metode și dovezi empirice pentru evaluarea pe termen lung, axată pe commit-uri, și pentru automatizarea analizei calității software-ului în proiectele open-source, lucruri de care beneficiază atât cercetătorii, cât și dezvoltatorii.

Cuprins

Lista publicaţiilor

1. **L.-M. Berciu**, *"PyDsBuilder - A Dataset Builder Written in Python Django"*, Studia Universitatis Babeș-Bolyai Informatica, vol. 69, no. 2, pp. 5–22, Feb. 2025. ISSN: 2065-9601, DOI: 10.24193/subbi.2024.2.01. (*categ D, **1** point, MathSciNet indexed*) [Ber25]

2. **L.-M. Berciu**, V.-A. Moldovan, *"Software Maintainability and Refactorings Prediction Based on Technical Debt Issues"*, Studia Universitatis Babeș-Bolyai Informatica, vol. 68, no. 2, pp. 22–40, Oct. 2023. ISSN: 2065-9601, DOI: 10.24193/subbi.2023.2.02. (*categ D, **1** point, MathSciNet indexed*) [BM23]

3. M. Petrescu, S. Motogna, **L.-M. Berciu**, *"Women in Scrum Master Role: Challenges and Opportunities*"*, 2023 IEEE/ACM 4th Workshop on Gender Equity, Diversity, and Inclusion in Software Engineering (GEICSE), Melbourne, Australia, pp. 49–55, 2023. DOI: 10.1109/GEICSE59319.2023.00011. (*categ A, **6** points, Scopus indexed*) [PMB23]

4. S. Motogna, **L.-M. Berciu**, V.-A. Moldovan, *"Artificial Intelligence Methods in Software Refactoring: A Systematic Literature Review"*, 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Paris, France, pp. 309–316, 2024. DOI: 10.1109/SEAA64295.2024.00055. (*categ B, **4** points, Scopus indexed*) [MBM24]

5. V.-A. Moldovan, **L.-M. Berciu**, R.-D. Patcas, *"The Python Software Quality Dataset"*, 2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Paris, France, pp. 395–398, 2024. DOI: 10.1109/SEAA64295.2024.00066. (*categ B, **4** points, Scopus indexed*) [MBP24]

6. **L.-M. Berciu**, S. Motogna, A. Molnar, *"A long-term exploratory study of source code quality issues in open-source Python projects"*, Software Quality Journal, vol. 34, no. 9, 2026. DOI: 10.1007/s11219-026-09740-z. (*categ C, **2** points, Scopus indexed*) [BMM26b]

7. **L.-M. Berciu**, S. Motogna, O. Pikus, A. Molnar, *"Exploring the relation between source code commit information and SonarQube issues"*, 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025), vol. 270, pp. 841–850, 2025. ISSN: 1877-0509, DOI: 10.1016/j.procs.2025.09.204. (*categ B, **2** points, Scopus indexed*) [BMPM25]

8. **L.-M. Berciu**, V.-A. Moldovan, S. Motogna, *"Exploring the Impact of Commit Burstiness on Software Quality"*, 18th International Conference on the Quality of Information and Communications Technology (QUATIC 2025), 2025. (*categ C, **2** points, Scopus indexed*) [LMBM]

9. V.-A. Moldovan, S. Motogna, **L.-M. Berciu**, *"A Comparative Evaluation of Intelligent Methods for Code Smell Detection in Python"*, in publicare. (*categ C, **0** points, Scopus indexed*) [VAMM]

10. R.-D. Patcas, O. Pikus, **L.-M. Berciu**, S. Motogna, *"AST Similarity of Code Quality Issues in Python and JavaScript"*, in publicare. (*categ X, **0** points, Scopus indexed*) [RDPM]

Scor total publicaţii: **22** points*

Ierarhizarea publicațiilor a fost realizată conform listelor de reviste și conferințe publicate de CNATDCU (Consiliul Național de Atestare a Titlurilor, Diplomelor și Certificatelor Universitare) aplicabile studenților doctoranzi înmatriculați între anii 2020-2024.

Lista artefactelor

1. **L. -M. Berciu**, V. -A. Moldovan, R. -D. Patcas - *The Python Software Quality Dataset* - [BMP24]
2. **L. -M. Berciu** - *PyDs Builder* - [Ber24]
3. **L. -M. Berciu**, S. Motogna, V. -A. Moldovan - *Software Refactoring SLR Dataset Resource* - [BMM24]
4. **L. -M. Berciu**, S. Motogna, A. Molnar - *Dataset for "A long-term exploratory study of source code quality issues in open-source Python projects" article* - [BMM26a]
5. **L. -M. Berciu**, A. Molnar, S. Motogna, O. Pikus - *Replication package for Exploring the relation between source code commit information and SonarQube issues* - [BMMP25]
6. **L. -M. Berciu**, V. -A. Moldovan, S. Motogna - *Dataset for "Exploring the Impact of Commit Burstiness on Software Quality"* - [BMM25]

Introducere

Context

Calitatea software-ului explică cât de bine funcționează un software, cât de bine se potrivește scopului său și cum este întreținut. În contextul acestei teze, *dimensiunea calității software* este împărțită în trei direcții principale [fSC21]: mentenabilitate, fiabilitate și securitate.

Calitatea software-ului trebuie măsurată [Ber16] folosind metrici. Cel mai bun mod de a o măsura automat este prin intermediul *instrumentelor de analiză statică (SATs)* precum SonarQube [Son24a] și PyLint [PyL23]. SonarQube este explorat pe larg în lucrările științifice ([LLTH19, LST19a, LLHT20]) și utilizat în studiile noastre asupra software-ului cu sursă deschisă (OSS).

Pe baza acestor măsurători, programatorii modifică codul prin *refactorizare*, schimbând structura internă fără a altera comportamentul, pentru a-l face mai ușor de înțeles și mai puțin costisitor de modificat [Fow99].

Intenția dezvoltatorului este un alt aspect important, împărtășit prin intermediul commit-urilor Git [git]. Clasificarea intenției dezvoltatorului a evoluat de la cele trei categorii ale lui Swanson [Swa76] la cele zece categorii granulare ale specificației convenționale a commit-urilor (CCS) [ZZQL25].

Parcursul de cercetare al acestei teze pornește de la construirea unui instrument de extragere a datelor și a unui set de date open-source, până la utilizarea Inteligenței Artificiale pentru clasificarea mesajelor de commit. În cele din urmă, rulăm modele de învățare automată (ML) pentru a prezice problemele de tip Code Smells și identificăm diferențele și asemănările dintre Arborii Sintactici Abstracți (AST) din diferite limbaje de programare.

Motivaţie

Ascensiunea software-ului colaborativ ([KS07]) a condus la numeroase proiecte OSS care acoperă domenii multidisciplinare și utilizează diferite limbaje de programare. Această diversitate se confruntă cu o amenințare semnificativă: vulnerabilitățile și erorile. Disciplina care poate preveni daunele ulterioare este calitatea software-ului [SGK+09].

Deși există multe experimente pentru Java ([LLTH19], [LST19a], [ALRT19]) și pentru capacitățile SonarQube ([LLHT20]), am observat un decalaj în ceea ce privește calitatea software-ului în Python. În Python, majoritatea defectelor sunt probleme de tip *code smells* [CCMX16]. Mai mult, regulile SonarQube pentru Python [Son24a] reprezentau aproape jumătate din numărul regulilor pentru Java, iar seturile de date disponibile erau limitate. Am decis să creăm un set de date vast pentru Python pentru a analiza modul în care funcționează SonarQube. Acest lucru ne conduce la primul nostru obiectiv, ***Obiectivul 1****: analiza atributelor calității software-ului și a evoluției lor pe termen lung în proiectele software.*

A doua motivație provine din apariția LLM-urilor (modele de limbaj de mari dimensiuni) și a intenției dezvoltatorului. Pentru Github [git], am ales să ne concentrăm pe commit-uri, deoarece acestea sunt unele dintre cele mai granulare unități ale intenției. Diferite lucrări categorizează mesajele de commit în mai multe categorii [Swa76], în timp ce altele utilizează o taxonomie mai granulară [ZZQL25]. Spațiul open-source permite analizarea unor perioade lungi de timp și crearea de studii longitudinale [MM20b]. Așa cum este descris în [ZZQL25], LLM-urile pot clasifica mesajele de commit cu un randament mai mare decât clasificarea manuală, obținând o precizie considerabil de mare (76%). Acest lucru ne conduce la al doilea obiectiv, **Obiectivul 2: utilizarea tehnologiilor AI pentru evaluarea longitudinală a problemelor de calitate a software-ului de-a lungul timpului**.

Contribuții

Scopul acestei teze este de a face progrese în domeniul Ingineriei Software, în special în ceea ce privește Calitatea Software-ului, combinată cu Inteligența Artificială. Contribuțiile sunt împărțite în trei părți principale, după cum sunt prezentate în următoarea enumerare:

- **Partea 1: Fundamente**
  - **PyDs Builder și dezvoltarea setului de date [Ber25]**: Prezentăm PyDs Builder, un instrument bazat pe Python pentru extragerea și analizarea artefactelor legate de calitate. Teza detaliază procesul său de inginerie și implementarea pentru a produce setul de date public Python Software Quality Dataset [MBP24], care cuprinde probleme SonarQube, metrici de cod, istorice de refactorizare și perechi de commit-uri pentru peste 90 de proiecte open-source.
- **Partea 2: Code Issues**
  - **Analiza longitudinală a problemelor codului sursă în proiectele Python [BMM26b]:** Analizarea a 57 de proiecte Python open-source de-a lungul unui deceniu dezvăluie că problemele de analiză statică își menține o prezență stabilă în timp. Problemele de tip *code smell*, erorile și vulnerabilitățile tind să se grupeze în anumite faze ale proiectului, subliniind importanța evaluării continue a calității.
  - **Clasificarea commit-urilor la nivel de cod sursă în Python utilizând modele de limbaj de mari dimensiuni [BMPM25]:** Folosind un model de limbaj de mari dimensiuni ajustat

fin (*fine-tuned*), am clasificat peste 90.000 de commit-uri cu o precizie ridicată. Contribuția a scos la iveală faptul că documentația, remedierea erorilor și adăugarea de funcționalități domină, facilitând o analiză precisă a modului în care tipurile de commit-uri se corelează cu problemele de analiză statică.

  - **Frecvența în rafală a commit-urilor și impactul acesteia asupra calității codului sursă [LMBM]:** Perioadele de rafală (*bursts*) cu commit-uri rapide și succesive sunt, în general, asociate cu o incidență mai scăzută a problemelor de analiză statică. Dezvoltarea concentrată în perioadele de rafală favorizează o calitate mai bună a codului, în timp ce fazele fără rafală prezintă o rată mai mare a problemelor, subliniind importanța tiparelor de commit disciplinate.
  - **Similaritatea AST a problemelor de calitate a codului în Python și JavaScript [RDPM]:** Am comparat Arborii Sintactici Abstracți (AST) ai fragmentelor de cod care induc reguli comune SonarQube în Python și JavaScript. Rezultatele arată că doar regulile simple, independente de limbaj, prezintă o similaritate AST interlingvistică ridicată. Regulile bazate pe complexitate prezintă constant o similaritate scăzută, demonstrând că AST-urile nu sunt suficiente prin ele însele pentru transferul regulilor de la un limbaj la altul.

- **Partea 3: Code Smells**

  - **Recenzie sistematică a literaturii privind inteligența artificială în refactorizare [MBM24]**: Realizăm o recenzie sistematică a metodelor de inteligență artificială aplicate în refactorizarea software, cartografiind peisajul cercetării pentru a identifica tendințele dominante și provocările insuficient explorate.
  - **Modele predictive pentru mentenabilitate și refactorizare: [BM23]** Propunem abordări bazate pe învățare automată supervizată pentru prognozarea cerințelor de mentenabilitate și refactorizare. Clasificatorul ExtraTrees produce o acuratețe ridicată, oferind informații acționabile pentru mentenanța direcționată și gestionarea datoriei tehnice.
  - **Evaluarea modelelor inteligente pentru detectarea problemelor de tip Code Smell [VAMM]**: Această contribuție evaluează șapte modele de învățare automată și un algoritm genetic pentru detectarea a zece probleme majore de tip *code smell* în Python. Studiul demonstrează că, frecvent, clasificatorii tradiționali, precum kNN și Naive Bayes, depășesc performanța modelelor de învățare profundă (*deep learning*) pentru anumite *code smells*.

Partea I

Fundamente

I.1 Fundament Teoretic

Ca în cazul oricărei cercetări, este important să înțelegem aspectele teoretice care o înconjoară. În acest capitol, vom încerca să subliniem câteva dintre cele mai importante concepte ale calității software-ului, care vor introduce conceptele și noțiunile de bază utilizate în această teză. Vom explora o perspectivă mai tehnică a fundamentelor, constând în *analiza statică* și scopul acesteia, *instrumentele* care sunt construite pe baza conceptului de analiză statică, ce sunt *seturile de date* și cum ajută acestea cercetătorii din întreaga lume, modul în care intenția dezvoltatorului și practicile de dezvoltare pot fi deduse prin *clasificarea commit-urilor*, cum graba în scrierea codului poate introduce probleme software prin *frecvența în rafală a commit-urilor* (*commit burstiness*) și ce este *refactorizarea software* și de ce este indispensabilă procesului de dezvoltare software.

I.1.1 Analiza Statică

Analiza statică este o metodă de inginerie software care permite dezvoltatorilor să identifice potențiale probleme prin scanarea automată a bazei de cod pentru structura, sintaxa și semantica acesteia. Oferă o perspectivă asupra calității software-ului fără a fi nevoie de execuția efectivă a codului. Această metodă este implementată prin crearea unui instrument software care utilizează arbori sintactici abstracți (AST) și euristici.

Din punct de vedere istoric, analiza statică a apărut în anii 1970. Verificările de bază ale corectitudinii au fost separate în instrumente dedicate prin crearea instrumentului *lint* de către Stephen C. Johnson [Tho21], unul dintre primele utilitare folosite pe scară largă pentru detectarea tiparelor în codul C. În paralel, informatica teoretică a dezvoltat cadre generale, cum ar fi interpretarea abstractă [CC77], care a oferit o bază riguroasă pentru aproximarea comportamentului programelor.

Analiza statică a evoluat de la simple verificări la nivel de sintaxă la un ecosistem divers de instrumente care analizează securitatea, fiabilitatea și mentenabilitatea. Au început să apară diferite tipuri de nereguli, pe care le vom numi *probleme* (*issues*). Pentru securitate, problemele sunt numite *vulnerabilități*, pe care un actor rău intenționat le poate folosi pentru a dăuna proprietarului. Pentru fiabilitate, problemele sunt numite *erori* (*bugs*), reprezentând un mod anormal de funcționare a unui sistem. Din perspectiva mentenabilității, problemele sunt numite *code smells*, care corespund unui cod sursă ce funcționează, dar este greu de înțeles și de modificat. Este important de clarificat faptul că problemele *code smells* definite de instrumente nu sunt aceleași cu cele definite de Fowler [Fow99]. Problemele *code smells* ale lui Fowler indică o problemă de design mai profundă, în timp ce instrumentele de analiză statică definesc verificări mai granulare care dăunează nivelului de înțelegere, dar nu sunt neapărat erori sau vulnerabilități.

Contrapartida este analiza dinamică, care implică scanarea software-ului în timp ce acesta rulează. Analiza dinamică nu face obiectul tezei noastre, deoarece ne concentrăm exclusiv pe analiza statică. Vom continua prin examinarea unor instrumente care au contribuit masiv la ecosistemul calității software.

I.1.2 Instrumente de Analiză Statică

Un instrument de analiză statică examinează codul sursă al altui program pentru a detecta probleme. Ne vom concentra în principal pe cele trei categorii: mentenabilitate, fiabilitate și securitate.

Instrumentul parsează codul într-un arbore sintactic abstract (AST) și aplică euristici pentru verificări de flux de control, flux de date și verificări semantice [Dat25]. Rezultatele sunt raportate ca probleme și afișate sub diferite forme, cum ar fi widget-uri IDE, platforme desktop [ST24] și platforme web [Son24a].

O componentă importantă a problemelor de cod o reprezintă metricile codului. Metricile denotă măsuri cantitative calculate folosind formule statice pe codul parsat. Pe baza unor praguri stabilite, o combinație de valori ale metricilor constituie o încălcare. Să luăm exemplul complexității cognitive [Sonnd]:

$$(V(G) = e - n + 2p)$$

Dacă rezultatul este > 15, va declanșa o regulă de tip *code smell*.

Există multiple instrumente care se încadrează în categoria SAT (Instrumente de Analiză Statică):

- Sonarqube: un instrument popular cu suport multi-limbaj pentru detectarea problemelor de cod și calcularea metricilor [Son25a].
- Understand: un instrument comercial care suportă grafuri de dependență și metrici OOP [ST24].
- PyLint: un linter specific Python care respectă convențiile PEP8 și detectează *code smells* [PyL23].
- PySZZ: un instrument Python open-source pentru identificarea commit-urilor care introduc erori (*bug-inducing*).
- Radon: un instrument Python open-source care calculează metrici precum complexitatea ciclomatică [Rub24].

SonarQube îmbunătățește standardele de calitate software [Son24a] prin detectarea problemelor printr-un flux de lucru bazat pe reguli (*rule-based pipeline*) și printr-o tranziție către principiile "Clean Code". Understand [ST24] oferă o vedere detaliată a structurii codului și extrage metrici orientate pe obiecte pentru a oferi perspective asupra potențialelor defecte de design [CK94]. PyLint verifică codul sursă Python comparându-l cu standardele de programare și generează mesaje categorizate cu niveluri de severitate pentru a ajuta dezvoltatorul. PySZZ [RPS+23] produce metrici de urmărire a defectelor pentru a determina ce commit a introdus cel mai probabil o eroare. Radon [Rub24] folosește verificatoare personalizate pentru a extrage metrici pentru detectarea de *code smells*.

Aceste instrumente constituie baza tehnică ce ne permite să redactăm studiile noastre. Pe parcursul acestei teze vom folosi în mod special instrumente pentru Python, unele studii concentrându-se, de asemenea, pe Java [BM23] și JavaScript [RDPM].

Programatorii trebuie să acorde o atenție specială rezultatelor fals pozitive. Un *fals pozitiv* este o problemă raportată care se potrivește sintactic cu o regulă, dar care este de fapt corectă în contextul real al codului și nu necesită o modificare [Son25b].

I.1.3 Studiul Commit-urilor

Un commit reprezintă o unitate logică și atomică de modificare a sursei unei baze de cod. Un dezvoltator modifică fișiere pentru a atinge un scop specific, cum ar fi implementarea unei funcționalități

sau remedierea unei erori. Commit-urile Git acționează ca un instantaneu (*snapshot*) al bazei de cod la un moment dat, permițând comutarea și urmărirea ușoară a versiunilor.

Părțile componente ale unui commit sunt următoarele:

- Hash ID: identificatorul unic al commit-ului.
- Object Tree (Arborele de obiecte): reprezintă conținutul commit-ului, conținând blocurile de date ale fișierelor (*file blobs*) și structura arborescentă a directoarelor. Git stochează întregul sistem de fișiere.
- Metadata: contextul commit-ului, incluzând autorul, persoana care face commit-ul (*committer*) și marcajul de timp (*timestamp*).
- Parent Pointer (Indicatorul către părinte): referința către commit-ul precedent.
- Message (Mesajul): descrierea lizibilă pentru om a modificărilor. Este adesea trecută cu vederea, ducând la descrieri neclare.

Clasificarea commit-urilor reprezintă acțiunea de a atribui o taxonomie specifică unui *mesaj de commit* pentru a interpreta activitățile de dezvoltare și a îmbunătăți calitatea codului. Acest lucru ajută la mentenanța software-ului, care modifică un program după livrare pentru a se asigura că funcționează corect.

Lucrarea lui Swanson [Swa76] reprezintă categorizarea tradițională a mentenanței software: *corectivă* (abordarea defectelor), *adaptivă* (răspunsul la medii în schimbare) și *perfectivă* (îmbunătățirea funcționalității).

Deși au fost investigate de multe studii [Hs23, HBS22, MV00], acestor categorii tradiționale le lipsește granularitatea necesară pentru o analiză mai profundă [CHK+01]. Cercetările recente [ZZQL25] pun accentul pe o granularitate mai fină în clasificările commit-urilor.

Specificația Convențională a Commit-urilor (CCS) [Con20] oferă un cadru mai structurat pentru a clasifica commit-urile. Aceasta facilitează o înțelegere mai profundă a scopului unui commit și ajută echipele să asocieze tipurile de commit-uri cu metricile calității software-ului.

Rafalele de commit-uri (*commit bursts*) sunt perioade de commit-uri rapide și succesive, concentrate pe anumite componente. Ele sunt indicatori ai unor potențiale defecte și sunt asociate cu o probabilitate crescută de apariție a defectelor ([NMB10, XF14, WNLB22]). Rafalele neregulate provoacă modificări pripite ale codului, rezultând în reducerea calității sau acumularea datoriei tehnice.

Clasificarea commit-urilor și frecvența în rafală a acestora se completează reciproc pentru a forma un portret mai apropiat de realitate al modului în care evoluează calitatea software-ului. Intenția dezvoltatorului și ferestrele de timp ale încărcărilor efective de cod oferă o imagine de ansamblu asupra modului în care scrierea în grabă a codului influențează apariția defectelor în diferite categorii de commit-uri.

I.1.4 Refactorizarea Software-ului

Refactorizarea software-ului restructurează structura internă a unui sistem software fără a-i altera comportamentul extern [Fow99]. Se concentrează pe reducerea datoriei tehnice, îmbunătățirea mentenabilității și optimizarea eficienței.

Programării timpurii îi lipsea îmbunătățirea deliberată a codului, riscând o logică încâlcită. În anii 1970, *Legile lui Lehman privind Evoluția Software-ului* [Leh80] au arătat că sistemele trebuie să se adapteze continuu (*Schimbare Continuă*) și că complexitatea internă crește dacă dezvoltatorii nu o reduc în mod deliberat (*Complexitate în Creștere*).

Termenul *refactorizare* a apărut pentru prima dată în 1990 [OJ90] și în teza lui Opdyke din 1992 [Opd92], și a fost abordat independent de Griswold [Gri91]. A câștigat o recunoaștere pe scară largă în 1999 prin Martin Fowler [Fow99] și Kent Beck [Bec99].

Suportul automatizat a început cu *Smalltalk Refactoring Browser* în 1997 [RBJ97]. Aceste instrumente operează pe Arbori Sintactici Abstracți (AST) pentru a efectua transformări sigure. Mediile de dezvoltare integrate (IDE) și instrumentele moderne combină acum perfect analiza statică și inteligența artificială. PyRef [ALT+21] este un instrument automatizat folosit în această teză pentru a extrage date de refactorizare din proiecte Python. Acesta analizează istoricul git și transformă fișierele modificate în AST-uri pentru a identifica operațiunile. Alte instrumente care identifică refactorizarea includ RefactoringMiner [1], RefactoringCrawler [2] și RefDiff [3]. Analizarea refactorizării în proiectele OSS le permite cercetătorilor să valideze dacă aceste modificări îmbunătățesc într-adevăr calitatea codului.

I.2 PyDs Builder: Un constructor de seturi de date și un instrument de extragere

Acest capitol descrie rezultatele publicate în [Ber25], iar contribuțiile științifice constau în crearea artefactului aplicației [Ber24] și publicarea acestuia pentru comunitatea științifică.

I.2.1 Motivație

Proiectele de software open-source (OSS) au cunoscut o creștere constantă în popularitate [MP12]. Făcute accesibile gratuit pe platforme precum GitHub, acestea invită un public larg să utilizeze software-ul. Integrarea sistemelor de versionare facilitează dezvoltarea colaborativă și păstrează un istoric cuprinzător al evoluției proiectului, oferind perspective asupra practicilor de dezvoltare și îmbogățind peisajul cercetării [KS07].

Cercetătorii s-au concentrat pe evaluarea calității software-ului, care se referă la măsura în care un produs software îndeplinește cerințele într-un mod fiabil, eficient și ușor de întreținut. Numărul mare de proiecte OSS a permis cercetătorilor să revizuiască în mod transparent calitatea software-ului [SGK+09].

Evaluarea calității software-ului necesită volume mari de informații. Aici intervine extragerea datelor (*data mining*) [CSS13], implicând instrumente care extrag date pentru experimente. Exemplele includ analizarea datoriei tehnice [LST19b] și studierea mentenabilității [MM21]. Aplicarea inteligenței artificiale necesită, de asemenea, cantități mari de date extrase [Wan18], cum ar fi extragerea mesajelor de commit din Github [KCA21] și automatizarea detectării erorilor [ESK18]. Extragerea din repozitoare presupune soluții software personalizate. Unele extrag date din sistemele de urmărire a problemelor (*issue trackers*) [JSM21] și [FMNL21], în timp ce altele oferă o gamă mai largă de metode de extragere [SAB18].

Acest subcapitol prezintă un raport de experiență pentru proiectarea unui instrument de extragere a datelor din repozitoare, creat pentru a extrage seturi de date din sisteme precum GitHub. Obiectivul principal este de a prezenta o propunere de design și de a oferi un artefact open source

[1] https://github.com/tsantalis/RefactoringMiner
[2] http://dig.cs.illinois.edu/tools/RefactoringCrawler/
[3] https://github.com/aserg-ufmg/RefDiff

[Ber24] numit PyDs Builder. Oferim un scenariu de experiment pentru a-i demonstra eficacitatea și sugestii pentru studii de cercetare empirică. Perspectivele obținute oferă lecții valoroase privind proiectarea instrumentelor scalabile de extragere a datelor.

Instrumentul constă din două părți principale. Prima implică extragerea datelor de urmărire a problemelor și a datelor generale ale proiectului din Github. A doua implică rularea instrumentelor de calitate software precum SonarQube [Son24a] și SZZ [RPS+23] pentru a rafina setul de date și a oferi perspective asupra atributelor calității software-ului. Ambele părți sunt esențiale pentru construirea cu succes a unui set de date.

I.2.2 Designul Instrumentului

Extragerea datelor în mod consecvent pentru experimente duce adesea la nevoia de a crea o soluție software. Structurarea și normalizarea unor cantități mari de date implică utilizarea diferiților algoritmi pentru a scana, extrage și agrega informații. În loc să folosim instrumente software diferite, ceea ce necesită mai mult efort, raportăm experiența de creare a unei soluții care agregă date din fluxuri diferite într-un singur format uniform, care poate fi extins după cum consideră cercetătorii de cuviință.

Un aspect important este **să se decidă ce formate de date sunt suportate**. Datele extrase prin cereri API sau fișiere (JSON, CSV, YAML) sunt procesate într-un standard unic, uniform. Vedem necesitatea de a suporta conversia datelor într-un format standardizat, cum ar fi SQL, care permite interogări complexe și se poate scala pe măsură ce cantitatea de date crește.

O altă problemă importantă este **automatizarea extragerii datelor**. Implementarea automatizării prin cozi de sarcini (*task queues*) permite procesarea continuă a datelor fără supraveghere constantă. Mai mult, soluția ar trebui implementată într-un limbaj de programare popular pentru a asigura adoptarea și extinderea rapidă.

Obținerea datelor în formatul dorit ar trebui să permită procesarea ulterioară prin introducerea lor într-un flux de date (*data pipe*) al instrumentelor personalizate. Datele obținute în urma execuției instrumentelor vor fi apoi salvate în aceeași structură uniformă. În concluzie, **decizia cu privire la flux este importantă**.

I.2.3 Soluția PyDs Builder

Introducem un instrument de extragere a datelor concentrându-ne pe liniile directoare subliniate mai sus: **automatizarea datelor, specificarea formatului** și **decizia privind fluxul**.

PyDs Builder este o soluție web, bazată pe API, construită folosind Python și framework-ul Django, care își propune să ofere o modalitate prin care cercetătorii să creeze seturi de date experimentale.

Obiectivul său principal este extragerea datelor despre repozitoare din sistemele de versionare, oferind inițial suport pentru Github. Datele sunt procesate în forma dorită și inserate într-o bază de date personalizată, urmând o **schemă SQL prestabilită**. Datele pot fi procesate în continuare de instrumente personalizate pentru a atinge obiectivele de achiziție de date ale unui experiment, și apoi utilizate pentru publicarea unui nou set de date sau pentru a alimenta cu date o soluție de inteligență artificială.

O prezentare generală a funcționalităților soluției este enumerată după cum urmează:

- Permite interacțiunea cu instrumentul printr-o interfață API

- Extrage date din repozitoare Github, cum ar fi problemele (*issues*), cronologiile problemelor, detaliile commit-urilor și codul sursă
- Execută instrumente de calitate software precum SonarQube [Son24a], SZZ [RPS+23] și PyRef [ALT+21] și asigură persistența datelor
- Oferă automatizare pentru procesarea rapidă a datelor
- Permite contribuția și extinderea codului prin modularitatea proiectului și o interfață de programare intuitivă.

I.2.4 Concluzii

Extragerea datelor și analiza proiectelor open source au fost subiecte importante în mediul academic datorită disponibilității datelor. Acest subcapitol a început cu motivația de a crea PyDs, o soluție Python cu framework-ul Django, care permite cercetătorilor să genereze seturi de date concentrându-se în principal pe experimente privind atributele calității software-ului. Am acoperit designul conceptual, pașii de implementare și am oferit instrucțiuni detaliate pentru configurarea și utilizarea aplicației. În cele din urmă, am explorat posibile direcții pentru extinderea aplicației și ne-am împărtășit experiența.

Etapele ulterioare implică implementarea sa practică în proiecte de cercetare. PyDs permite cercetătorilor să o utilizeze pentru extragerea datelor din instrumente populare de calitate software, precum SonarQube și SZZ. În prezent, PyDs suportă doar Github, dar există planuri viitoare de integrare cu Jira și de extragere a datelor CI/CD din Jenkins. În cele din urmă, PyDs Builder a fost utilizat cu succes pentru a crea un set de date axat pe proiecte Python open-source [MBP24], subliniindu-i flexibilitatea și utilitatea practică.

I.3 Setul de Date privind Calitatea Software-ului în Python

Seturile de date reprezintă unul dintre cei mai importanți piloni ai unui studiu empiric de succes. Deoarece am dorit să ne concentrăm eforturile pe studierea modului în care calitatea software-ului evoluează în proiectele OSS în Python, având în vedere că am construit instrumentul din chapter I.2 și că seturile de date pentru Python erau destul de rare în ecosistemul literaturii de specialitate, am considerat necesar să construim propriul nostru set de date, care va fi utilizat de noi și de alți cercetători care l-ar putea considera util pentru scopurile lor. Acest capitol prezintă studiul [MBP24] care a fost realizat în colaborare cu studenții doctoranzi Vasilica-Andreea Moldovan și Rareș Patcaș. Contribuțiile mele au constat în stabilirea criteriilor de selecție a proiectelor, rularea instrumentului PyDs Builder din chapter I.2 pentru extragerea datelor din diferitele instrumente, configurarea arhitecturii bazei de date și crearea per ansamblu a setului de date.

I.3.1 Motivație

Ingineria software a cunoscut o creștere semnificativă, transformând Calitatea Software-ului într-un domeniu de un interes deosebit. Aceasta ridică nivelul general al calității software-ului prin îmbunătățirea funcționalității și corectitudinii, ajutând la reducerea riscurilor asociate cu erorile (*bugs*).

Pentru a obține o înțelegere mai profundă a codului, analiza statică identifică problemele potențiale fără a necesita execuție. Un rezultat pe cale de consecință este identificarea datoriei tehnice, care se referă la costul acumulat al deciziilor de dezvoltare suboptime ce conduc la defecte și complexitate. Mai mult, refactorizarea software-ului servește ca o abordare metodică pentru curățarea codului, modificând structura internă fără a afecta comportamentul extern, pentru a diminua erorile.

Numeroase studii și seturi de date, precum [LST19b], [SK21] și [SSM05], au fost compilate, dar acestea se concentrează în principal pe Java. Deoarece Python a devenit un concurent puternic [Sri17], acest lucru determină luarea în considerare a studierii proiectelor Python într-o manieră similară.

Crearea unui set de date cuprinzător pentru Python este importantă pentru a oferi o perspectivă asupra practicilor de dezvoltare software și pentru a facilita noi studii privind predicția datoriei tehnice, identificarea oportunităților de refactorizare și evaluarea calității software-ului.

Obiectivul acestui studiu este de a utiliza instrumentul de extragere din repozitoare din Capitolul I.2 pentru a crea un set de date cuprinzător [LB24] pentru proiecte Python. Setul de date conține informații pentru 92 de proiecte Python, cuprinzând 51.805.677 de probleme SonarQube, 324.969 de metrici de calitate a codului SonarQube, 12.301 de înregistrări de refactorizare software, 16.177 de perechi de commit-uri care introduc erori (*bug-inducing*) și care repară erori (*bug-fixing*) și 863.931 de intrări din sistemul de urmărire a problemelor de pe GitHub.

I.3.2 Metodologie

Metodologia utilizată pentru construirea setului de date prezentat în acest capitol se inspiră din diverse abordări documentate în domeniul Ingineriei Software, așa cum s-a discutat în lucrări anterioare precum [RTLR18] și [LS08]. Mai mult, metodologii similare au fost aplicate în mai multe studii documentate în literatura de specialitate, inclusiv cele menționate în [ODA+15] și [VSM13].

I.3.3 Criterii de includere/excludere

Pentru a identifica proiectele pentru analiză și stocarea rezultatelor în baza de date, au fost evaluate sistematic mai multe criterii. Doar proiectele open-source au fost examinate pentru a asigura accesibilitatea în vederea validării. Am ales proiecte care au Python ca bază de cod dominantă, deoarece ne concentrăm doar pe proiecte Python. Am considerat că un minim de trei lansări (*releases*) asigură stabilitate structurală și puncte de comparație. Proiectele trebuiau să aibă cel puțin 50 de probleme (*issues*) pentru a facilita algoritmul SZZ și a surprinde complexitatea proiectului. Pragul de commit-uri a fost setat la 500, pentru a asigura o activitate de dezvoltare suficientă. În cele din urmă, a fost necesară o maturitate minimă de cel puțin 3 ani, pentru a exclude proiectele instabile, aflate în stadii incipiente. Criteriile luate în considerare sunt prezentate concis în Tabelul I.3.1.

Criteriu	**Prag**
Utilizarea codului Python	$\geq 50\%$
Lansări ale proiectului (*Releases*)	≥ 3
Număr de probleme (*Issues*)	≥ 50
Număr de commit-uri	≥ 500
Vechimea proiectului	≥ 3 ani

Tabela I.3.1: Criterii de selecție pentru proiectele Python

I.3.4 Prezentare generală a instrumentelor și a setului de date

Principalul instrument utilizat pentru a crea setul de date este PyDs Builder I.2, care utilizează module de aplicație separate și PostgreSQL [Pos23] pentru performanța sa ridicată și gestionarea conexiunilor paralele. Am extras date folosind SonarQube datorită popularității sale în industrie, PyRef pentru performanța sa remarcabilă în extragerea refactorizărilor [ALT+21] și PySZZ pentru a găsi commit-urile care introduc erori. Acest set de date cuprinde o analiză a 92 de proiecte Python open-source, unde SonarQube a scos la iveală 51.805.677 de probleme tehnice și 324.969 de metrici de calitate a software-ului. În plus, PySZZ a generat 16.177 de perechi de commit-uri care introduc erori și care repară erori în 50 de proiecte, în timp ce PyRef a identificat 12.301 de instanțe de refactorizare software în 52 de proiecte. Setul de date final este disponibil public la [MBP24].

I.3.5 Concluzii

Obiectivul acestui capitol a fost de a stabili un set de date cuprinzător, care să includă factorii ce influențează mentenabilitatea codului. Ne-am concentrat pe proiecte Python datorită importanței acestui limbaj. Setul de date include probleme de datorie tehnică, metrici de calitate a software-ului, perechi de commit-uri *bug-inducing* și *bug-fixing*, precum și procese de refactorizare software executate.

Datele au fost extrase cu instrumente specializate: SonarQube pentru analiza statică, PyRef pentru refactorizările executate și PySZZ pentru a colecta informații despre perechile de commit-uri care introduc și care repară erori. Aceste instrumente au fost integrate într-un cadru de lucru cu o bază de date PostgreSQL.

Proiectele cu erori ambigue în timpul analizei au fost omise. Obiectivul nostru principal a fost să compilăm date diverse, mai degrabă decât să încercăm să rectificăm problemele de analiză.

Rezultatele vor oferi cercetătorilor posibilitatea de a efectua diverse investigații utilizând un set de date comun. Cercetările viitoare pot explora diverse dimensiuni ale calității, cum ar fi datoria tehnică și *code smells*, alături de potențialele relații cauzale dintre refactorizările software, commit-urile cu erori și problemele de datorie tehnică.

Partea II

Code Issues

II.1 Un studiu exploratoriu pe termen lung al problemelor de calitate a codului sursă în proiectele Python open-source

Cercetarea empirică vizând calitatea software-ului a dus la un volum consistent de lucrări, în special cu ajutorul instrumentelor automate care permit cercetătorilor să extragă cantități mari de date. Cu toate acestea, constatăm că multe dintre aceste eforturi oferă analize transversale sau longitudinale doar pe termen scurt. Mai mult, ele sunt adesea concentrate pe un singur limbaj, și în majoritatea cazurilor, un limbaj cu tipizare statică precum Java. În acest subcapitol, ne propunem să lărgim orizontul eforturilor existente prin explorarea compoziției, distribuției și evoluției problemelor de calitate a codului sursă în proiecte Python open-source complexe. Utilizăm instrumentul de analiză statică SonarQube pe un set de date care cuprinde 3.656 de lansări (*releases*) individuale ale 57 de proiecte Python. Explorăm impactul acestor probleme asupra mentenabilității, fiabilității și securității software-ului, investigăm evoluția acestor probleme pe termen lung și comparăm rezultatele noastre cu cele din literatura de specialitate. Descoperirile noastre sunt astfel comparate cu cercetările existente care vizează atât Python, cât și Java. Datele rezultate sunt publicate și open-source pentru a ajuta la reproducerea investigației și a contribui la construirea de seturi de date deschise și la scară largă pentru cercetare.

Acest capitol prezintă studiul [BMM26b] care a fost realizat în colaborare cu coordonatoarea mea, doamna profesor Simona Motogna, și domnul conferențiar Arthur Molnar. Contribuțiile mele au constat în curățarea datelor (*data curation*), mai precis extragerea și procesarea datelor, concentrându-mă pe analiza datelor și pe răspunsul la întrebarea de cercetare **RQ3** a studiului prin rularea corelației Spearman, precum și crearea artefactului setului de date [BMM26a] pentru comunitatea științifică.

II.1.1 Motivație

Platformele colaborative precum GitHub au fost adoptate pe scară largă pentru dezvoltarea de software. În acest proces, atenția noastră se concentrează pe calitatea software-ului. Analiza codului sursă utilizează metrici tradiționale și orientate pe obiect [AAM16, GFS05], precum și analiza statică [dPB+23, Mol20, DACA22, MM22]. SonarQube este cel mai utilizat instrument [Ao21] datorită analizelor sale cuprinzătoare. Aplicarea sa pe lansări succesive oferă date istorice privind diferiți indicatori de calitate care pot servi la identificarea diverselor practici deficitare.

Motivația acestui subcapitol constă în acoperirea unui decalaj în cercetarea empirică. În timp ce literatura de specialitate acoperă pe larg proiectele Java [LST19a, MM20a, DACA22, NCK+19], puține studii abordează Python [TFAL21, TFA20]. Corelăm modificările codului sursă cu problemele SonarQube, concentrându-ne pe distribuția generală a categoriilor de calitate a software-ului. Java este un limbaj cu tipizare statică și orientat pe obiect, în timp ce Python are o tipizare dinamică. Acest lucru are un impact direct asupra aplicabilității regulilor SonarQube, ceea ce înseamnă că regulile comune ambelor limbaje se manifestă diferit. Acest aspect motivează necesitatea unui studiu longitudinal dedicat bazelor de cod Python pentru a înțelege mai bine evoluția calității codului lor.

Pentru a ne atinge obiectivul, am folosit un set de date extins de 57 de proiecte Python pentru a

extrage metrici ale codului sursă și informații despre calitatea codului folosind SonarQube.

Contribuțiile noastre originale pot fi rezumate astfel:

- **Un studiu longitudinal amplu al problemelor de calitate în Python** cuprinzând 57 de proiecte și 3.656 de lansări de-a lungul a 10 ani;
- **Analiza corelației la nivel de commit** permițând urmărirea evoluției pe termen lung a problemelor de calitate și a relației acestora cu modificările de bază;
- **Contribuția la înțelegerea instrumentului SonarQube în sine** prin analizarea modului în care schimbările recente din taxonomia sa influențează detectarea problemelor.

II.1.2 Designul Studiului

Obiectiv și Întrebări de Cercetare

Definim acest capitol ca un studiu de caz cantitativ longitudinal [Ral21, KK15, MMD11] pentru a observa, descrie și explica problemele SonarQube. Acesta cuprinde 57 de proiecte Python, fiecare având cel puțin 6 lansări investigate [KK15], iar peste 70% dintre ele acoperind o perioadă de cel puțin 3 ani.

Utilizând abordarea Goal-Question-Metric [BCR94], obiectivul nostru este de a investiga compoziția, distribuția și evoluția problemelor de analiză statică în proiectele Python open-source, și de a le compara cu proiectele Java. Operaționalizăm acest lucru prin:

- **RQ1:** *Care este compoziția problemelor de analiză statică?*
- **RQ2:** *Cum sunt distribuite problemele de analiză statică în codul sursă al aplicației?*
- **RQ3:** *Care este evoluția codului sursă pe termen lung în ceea ce privește problemele de analiză statică?*
- **RQ4:** *Cum se compară compoziția problemelor de analiză statică din proiectele studiate cu rezultatele existente în literatura de specialitate?*

Prin **RQ1**, am vizat înțelegerea modului în care au fost distribuite problemele de tip *code smells*, erorile (*bugs*) și vulnerabilitățile, precum și impactul lor asupra mentenabilității, fiabilității și securității [Son24a]. Cu **RQ2**, am analizat distribuția problemelor SonarQube pentru a oferi recomandări de prioritizare, explicând care reguli au produs cel mai mare număr de probleme. Prin **RQ3**, ne-am propus să examinăm evoluția problemelor pe termen lung alături de metricile proiectului. În cele din urmă, **RQ4** compară compoziția problemelor descoperite cu lucrările conexe, examinând constatările frecvente și discutând amenințările la adresa validității.

II.1.3 Rezultate și Discuții

RQ1: *Care este compoziția problemelor de analiză statică?*

Am început prin a analiza tipurile de probleme returnate de SonarQube. Problemele de tip *code smells* au reprezentat, în medie, 96,67% din toate problemele raportate, urmate de erori (*bugs*) cu 3,08% și vulnerabilități cu 0,25%. Majoritatea proiectelor au un număr moderat spre mic de *code smells*, indicând o calitate bună a codului. Există o variație considerabilă în ceea ce privește numărul mediu de erori per proiect. Vulnerabilitățile au fost rare, cu excepția câtorva proiecte: *ansible*, *sqlalchemy*, *scrapy* și *plex-trakt-scrobbler*.

Constatarea 1: *Problemele de tip code smells reprezintă marea majoritate a problemelor de cod detectate în proiectele Python. Incidențele mai mari de erori sau vulnerabilități reprezintă cazuri izolate de calitate scăzută, nu o tendință.*

Începând cu versiunea 8.6, SonarQube a introdus atributele unui cod curat (*clean code*). Problemele de mentenabilitate domină (89,80%), în timp ce fiabilitatea (7,42%) și securitatea (2,76%) reprezintă proporții mai mici. Diferențe în distribuția problemelor de securitate au apărut în cazuri atipice precum *matplotlib* și *keras* din cauza regulilor clasificate ca *code smells* care au, suplimentar, un impact scăzut asupra securității.

Am explorat dacă problemele de calitate depind de dimensiunea proiectului utilizând liniile de cod (LOC). Deși inspecția vizuală a sugerat că nivelul de calitate al unui proiect nu este direct legat de dimensiune, o corelație a rangurilor Spearman per proiect ($p < 0,05$) a arătat că pe măsură ce dimensiunea crește, numărul de probleme crește, de asemenea, pentru numărătorile legate de mentenabilitate, fiabilitate și securitate.

Constatarea 2: *Metricile de numărare ale SonarQube oferă perspective precise asupra mentenabilității, fiabilității și securității. Corelațiile Spearman per proiect arată că LOC este, în general, asociat pozitiv cu numărul de probleme, iar proiectele cu o calitate mai scăzută prezintă adesea mai multe probleme de calitate simultan.*

Punct Cheie de Acțiune 1: Problemele de tip *code smells* domină problemele de calitate, dar erorile și vulnerabilitățile ar trebui prioritizate primele. Problemele de tip *code smells* recurente în anumite proiecte sugerează că problemele de calitate ar putea fi moștenite de la componente terțe (*third-party*).

RQ2: *Cum sunt distribuite problemele de analiză statică în codul sursă al aplicației?*

Ne propunem să investigăm modul în care problemele SonarQube sunt răspândite în cadrul proiectelor. Studii anterioare au sugerat [LST19a, MM22, WM18] că un subset mic de reguli este responsabil pentru majoritatea problemelor. Pentru fiecare aplicație, am luat în considerare primele 10 reguli care au generat probleme, obținând astfel o listă de 73 de reguli.

Trei reguli care generează probleme de tip *code smell* au apărut în marea majoritate a celor 57 de proiecte analizate: *S3776: Cognitive Complexity of functions should not be too high* (severitate critică), *S1192: String literals should not be duplicated* (severitate critică) și *S1481: Unused local variables should be removed* (severitate minoră). Mai mult, regulile *S117* și *S1192* au generat un număr considerabil din totalul de probleme.

Combinând distribuția și severitatea acestor reguli, un set mic necesită o atenție specială: *S1192*, care ascunde decizii proaste de design, și *S3776*, care face ca porțiunile de cod afectate să fie greu de înțeles.

Constatarea 3: *Există un număr mic de reguli SonarQube care tind să apară în majoritatea proiectelor Python și care sunt responsabile pentru majoritatea problemelor de cod, având un impact ridicat asupra mentenabilității.*

Punct Cheie de Acțiune 2: Majoritatea problemelor decurg dintr-un set mic de reguli, indicând complexitatea cognitivă și un design deficitar. Aceste rezultate pot fi folosite pentru a prioritiza soluționarea problemelor.

RQ3: *Care este evoluția codului sursă pe termen lung în ceea ce privește problemele de analiză statică?*

În această subsecțiune, explorăm relația dintre calitatea software-ului (SQ) și modificările din commituri de-a lungul timpului. Toate proiectele selectate au fost analizate pe parcursul întregii lor durate de viață, 90% dintre ele întinzându-se pe o perioadă cuprinsă între 3 și 10 ani. Scopul principal a

fost de a investiga corelația dintre modificările aduse de commit-uri și evoluția problemelor de calitate a software-ului, informând dacă modificările frecvente duc la o îmbunătățire sau o degradare [BASB17, KRS13, TEHG23].

Am calculat coeficientul Spearman între metricile commit-urilor (modificări de cod și modificări de fișiere) și metricile de calitate a software-ului pentru fiecare repozitoriu. Din totalul de **342** de coeficienți calculați, ne-am concentrat pe rezultatele semnificative statistic ($p < 0,05$). După filtrare, am reținut **77** de coeficienți din **23** de repozitoare, bazându-ne pe un eșantion solid de 3.202 intrări [Buj24].

Constatarea 4: *Peste* 70% *dintre proiecte prezintă o corelație între numărul de fișiere sau modificările de cod și problemele SonarQube asociate cu atributele de calitate a software-ului. Atributele de calitate asociate cu mentenabilitatea, fiabilitatea și securitatea prezintă o variație medie în timpul perioadei de dezvoltare a proiectelor.*

Punct Cheie de Acțiune 3: Iterarea regulată asupra designului și implementării codului sursă este corelată cu îmbunătățiri ale calității codului sursă, în special pentru aspectele legate de mentenabilitate și securitate.

RQ4: *Cum se compară compoziția problemelor de analiză statică din proiectele studiate cu rezultatele existente în literatura de specialitate?*

Pentru a aborda această întrebare de cercetare, am comparat problemele SonarQube din setul nostru de date cu studiile empirice existente. O comparație riguroasă este constrânsă de diferențele de metodologie, de versiunile SonarQube, de configurațiile regulilor și de proiectele selectate. Acești factori, alături de evoluția regulilor SonarQube și diferențele specifice limbajului, limitează comparabilitatea între studii. Cu toate acestea, analiza noastră confirmă că regulile legate de mentenabilitate domină numărul de probleme în toate studiile. Cel mai notabil, *S3776* (complexitate cognitivă ridicată) și *S125* (cod comentat) apar în mod repetat printre cele mai frecvent încălcate reguli atât în sistemele Java, cât și în cele Python.

Constatarea 5: *Există o suprapunere semnificativă între regulile care generează cele mai multe probleme în studiile existente. Complexitatea cognitivă excesivă (S3776) și secțiunile de cod comentat (S125) par prevalente atât în proiectele Java, cât și în cele Python.*

Punct Cheie de Acțiune 4: Anumite probleme detectabile prin utilizarea instrumentelor de analiză statică sunt comune în diferite proiecte și limbaje. Dezvoltatorii pot îmbunătăți calitatea codului configurând regulile aplicate și prioritizând eforturile de soluționare a problemelor cu importanță ridicată.

II.1.4 Concluzii

Constatările oferă o perspectivă asupra caracterizării și evoluției problemelor de calitate a codului sursă în proiectele Python open source. Explorarea longitudinală la scară largă și analiza comparativă cu Java contribuie la înțelegerea dinamicii specifice Python în ceea ce privește calitatea codului.

O observație cheie este dominanța problemelor de tip *code smells*, erorile și vulnerabilitățile fiind mult mai puțin frecvente. Deoarece problemele *code smells* sunt o cauză majoră a acumulării datoriei tehnice [DACA22, MM20a], acest studiu subliniază cele mai comune reguli încălcate și poate fi folosit ca un ghid atunci când se prioritizează soluționarea problemelor în timpul dezvoltării software și al refactorizării codului.

În cele din urmă, am descoperit că modificările frecvente ale codului au fost asociate cu metrici de calitate îmbunătățite într-un subset de proiecte (≈40%), în timp ce majoritatea nu au arătat o relație

semnificativă. Acest lucru evidențiază faptul că efectul activității de commit este specific proiectului, ceea ce înseamnă că monitorizarea continuă a indicatorilor de calitate rămâne importantă.

Pentru cercetători, rezultatele oferă dovezi empirice privind compoziția și evoluția pe termen lung a problemelor SonarQube pentru Python, alături de o comparație cu Java. Pentru practicieni, studiul crește nivelul de conștientizare cu privire la cele mai frecvente *code smells*, erori și vulnerabilități, permițând o acțiune proactivă pentru a prioritiza cele mai importante probleme în funcție de severitatea și impactul lor.

II.2 Explorarea relației dintre informațiile despre commit-urile codului sursă și problemele SonarQube

Acest capitol reprezintă a doua parte a analizei problemelor de cod, în care explorăm modul în care categoriile din specificația convențională a commit-urilor sunt distribuite între lansările (*releases*) de software. Am analizat 2.600 de perechi de lansări de software din 57 de proiecte open-source în Python și am utilizat modele de limbaj de mari dimensiuni pentru a eticheta 90.318 commit-uri. Am identificat ce tipuri de commit-uri sunt predominant responsabile pentru introducerea problemelor SonarQube și am explorat relația dintre tipurile de commit-uri și atributele de tip „clean-code" afectate de problemele de calitate. De asemenea, am publicat setul de date rezultat pentru a putea fi utilizat de alți cercetători.

Acest studiu a fost realizat în colaborare cu coordonatoarea mea, prof. Simona Motogna, studentul doctorand Oskar Picuș și conferențiarul Arthur Molnar. Contribuțiile mele au constat în metodologie, colectarea și analiza datelor, investigarea întrebărilor de cercetare unu și doi, concluziile și crearea artefactului setului de date [BMMP25]. Articolul a fost acceptat pentru publicare la KES 2025 [BMPM25].

II.2.1 Motivație

Dezvoltarea colaborativă de software se bazează pe gestionarea eficientă a contribuțiilor de cod prin intermediul commit-urilor structurate. Clasificarea commit-urilor oferă o modalitate sistematică de interpretare a activităților de dezvoltare, de măsurare a impactului acestora și de îmbunătățire a calității generale a software-ului.

În mod tradițional, activitățile de mentenanță software au fost clasificate în trei tipuri principale: *corective*, *adaptive* și *perfective* [Swa76]. Deși au fost investigate de mai multe studii [Hs23, HBS22, MV00], acestor categorii le lipsește granularitatea necesară pentru o analiză mai profundă [CHK+01]. În consecință, cercetările recente [ZZQL25] au subliniat necesitatea unei granularități mai fine în clasificările commit-urilor pentru a înțelege mai bine procesul de dezvoltare.

A fost introdusă Specificația Convențională a Commit-urilor (CCS) [Con20], oferind un cadru structurat pentru a clasifica commit-urile dincolo de categoriile tradiționale de mentenanță. CCS facilitează o înțelegere mai profundă a activităților de dezvoltare, permițând echipelor să asocieze tipuri specifice de commit-uri cu metricile calității software-ului.

În acest capitol, explorăm modul în care clasificările commit-urilor se corelează cu problemele de calitate a software-ului identificate de instrumentul de analiză statică SonarQube [Son24a], ales datorită prevalenței sale [TFA23, DLA+18, TFAL21]. Ne-am propus să investigăm clasificările commit-urilor și impactul acestora asupra calității software-ului prin examinarea distribuției categoriilor

CCS. Am identificat ce tipuri de commit-uri au fost predominant responsabile pentru introducerea problemelor SonarQube și am explorat relația dintre tipurile de commit-uri și categoriile de tip „clean code" [Son24b]. Emitem ipoteza că înțelegerea tipurilor de modificări asociate cu problemele de calitate îi poate ajuta pe dezvoltatori să gestioneze mai bine datoria tehnică.

II.2.2 Designul Studiului

Ne rafinăm obiectivul utilizând următoarele întrebări de cercetare:

- **RQ5:** *Care este distribuția tipurilor de commit-uri în rândul commit-urilor dintre lansări (releases)?* Deoarece clasificarea CCS a câștigat o atenție sporită [ZZQL25], ne concentrăm pe determinarea celor mai prevalente tipuri de commit-uri între lansări consecutive.
- **RQ6:** *Ce tip de commit-uri introduc cele mai multe probleme SonarQube?* Adăugăm la categoriile de commit-uri dimensiunea aspectelor legate de calitatea software-ului, așa cum sunt exprimate prin problemele SonarQube.
- **RQ7:** *Există o asociere între categoriile „clean code" din SonarQube și tipurile de commit-uri CCS?* Fiecare problemă de calitate a software-ului afectează diverse atribute de tip „clean code", SonarQube definind 4 categorii principale [Son24b]. „Clean code" servește ca fundament pentru dezvoltarea unui software ușor de întreținut și de înaltă calitate; prin urmare, acest lucru permite evaluarea aspectelor de calitate dintr-o perspectivă diferită. În cadrul acestui RQ, ne propunem să investigăm modul în care această nouă dimensiune este asociată commit-urilor.

II.2.3 Metodologie

Studiul de față analizează datele commit-urilor din proiecte open source în Python, concentrându-se pe commit-urile dintre lansări GitHub consecutive [Git24]. Am ales lansările (*releases*) deoarece acestea marchează etape importante în dezvoltarea codului. Am vizat Python datorită popularității sale și lipsei relative de lucrări exploratorii în comparație cu Java. SonarQube (versiunea 10.3) este analizatorul nostru static preferat datorită popularității sale în industrie.

Am utilizat setul de date din [MBP24], acoperind perioada 2013-2023, luând în considerare parcursurile directe de commit (**R1** $\rightarrow$ **R2**) și eliminând progresiile neliniare pentru a include **2.648** de perechi de lansări și **90.318** de commit-uri, publicând datele [BMMP25]. Am clasificat aceste commit-uri în 10 categorii CCS [Con20] utilizând un model CodeLLama condus de LLM [ZZQL25] cu un scor F1 acceptabil de 76,41%, eliminând commit-urile non-englezești utilizând biblioteca langdetect, alături de cele clasificate incorect. Pentru a identifica commit-ul care a introdus o problemă SonarQube, am conceput un algoritm specific[1] care combină datele diff ale commit-ului, datele de context AST și gestionarea unicității din SonarQube [Son25a] pentru a lega o problemă din **R1** sau **R2** de un anumit commit. Algoritmul potrivește Git diff-urile cu problemele, validează faptul că problema nu există în lansările adiacente și elimină duplicatele. L-am validat utilizând un eșantion aleatoriu de 100 de probleme, ignorând commit-urile cu erori de parsare.

[1] O vedere cuprinzătoare a algoritmului și a pașilor săi poate fi găsită în pachetul de replicare [BMMP25], în fișierul **matching_algorithm_explained.MD**.

II.2.4 Rezultatele Analizei

RQ5: *Care este distribuția tipurilor de commit-uri în rândul commit-urilor dintre lansări (releases)?*

Categoriile convenționale de commit includ 10 tipuri [Con20]. Distribuția arată o concentrare puternică pe *docs*, urmată îndeaproape de *fix*. Un procent de 9% dintre commit-uri sunt etichetate ca *feat*. Testarea, refactorizarea și stilul au o distribuție generală decentă, în timp ce *CI*, *build* și *perf* apar mai rar. Am explorat distribuția între lansările luate în considerare și pe parcursul duratei de viață a proiectului. În majoritatea cazurilor, au existat una sau două lansări în care numărul de commit-uri a depășit semnificativ restul.

Constatarea 6: Deși am detectat o variație semnificativă în utilizarea clasificării commit-urilor între lansări, există un echilibru clar între diferitele tipuri de commit-uri, documentarea și dezvoltarea de funcționalități fiind prevalente.

RQ6: *Ce tip de commit-uri introduc cele mai multe probleme SonarQube?*

Pentru a aborda a doua întrebare de cercetare, am adoptat o metodă în trei pași: prezentăm distribuția generală a problemelor în diferite categorii CCS, examinăm primele 10 etichete (*tags*) și evidențiem primele 10 reguli încălcate.

Un număr mare de probleme au fost introduse de categoria *feat*. În cazul categoriei *refactor*, am observat numere aproximativ egale de probleme adăugate și eliminate.

Datele dezvăluie că etichetele *convention*, *design* și *numpy* au ocupat constant primele poziții. Acest lucru indică faptul că majoritatea problemelor nu sunt critice, subliniind prevalența *code smells*. Subliniem că cele mai frecvent introduse și remediate probleme sunt legate de convențiile de denumire din Python (*python:S117*[2]), literalii șir de caractere duplicați (*python:S1192*), funcțiile cu o complexitate cognitivă ridicată (*python:S3776*) și parametrii neutilizați (*python:S1172*).

Constatarea 7: Indiferent de categoria CCS, majoritatea problemelor sunt *code smells* și afectează în principal mentenabilitatea și lizibilitatea codului sursă. Ele se pot acumula în timp, făcând baza de cod mai greu de gestionat.

RQ7: *Există o asociere între categoriile „clean code" din SonarQube și tipurile de commit-uri CCS?*

Pentru a studia această asociere, am utilizat testul de independență χ^2 [dTF+19] cu un nivel de semnificație de 0,10 [TAA+18]. Am efectuat teste pe perechi grupate după tipul de commit și am aplicat corecțiile Benjamini-Hochberg. Pentru asocierile semnificative, am calculat coeficientul V al lui Cramér pentru a evalua puterea efectului [Lee16]. Din cele 40 de teste efectuate, doar 7 asocieri s-au dovedit a fi semnificative statistic, toate prezentând mărimi ale efectului neglijabile. Acest lucru indică absența unei relații semnificative între tipul de commit și proprietățile de tip „clean code" afectate de problemele de calitate asociate.

Constatarea 8: Nu am putut identifica o asociere semnificativă statistic între atributele de tip „clean code" definite de SonarQube și tipurile de commit-uri CCS.

[2]Toate regulile SonarQube sunt detaliate la `[https://next.sonarqube.com/sonarqube/coding_rules?languages=py&selected=python](https://next.sonarqube.com/sonarqube/coding_rules?languages=py&selected=python)%3AInequalityUsage`

II.2.5 Concluzii

Acest studiu exploratoriu a examinat modul în care diferite tipuri de commit-uri se raportează la problemele de calitate a software-ului detectate de SonarQube în proiectele Python. Am analizat peste 90.000 de commit-uri de-a lungul lansărilor de software, utilizând un model de limbaj de mari dimensiuni pentru a clasifica mesajele de commit și un algoritm personalizat pentru a asocia problemele cu commit-urile individuale.

Rezultatele au arătat că *docs*, *feat* și *fix* au fost cele mai utilizate categorii de commit-uri. Majoritatea problemelor introduse au fost legate de *code smells*, care afectează mentenabilitatea și lizibilitatea codului mai degrabă decât să provoace defecte funcționale. Aceste probleme au apărut frecvent în adăugările de funcționalități și în refactorizare, sugerând necesitatea unor revizuiri mai stricte ale codului și a unor practici de refactorizare mai bune.

De asemenea, nu am găsit o asociere statistică semnificativă între tipurile de commit-uri și atributele de tip „clean code". Acest lucru a indicat că tipul de commit în sine nu este un predictor de încredere al problemelor de calitate, deoarece alte elemente pot juca un rol mai semnificativ.

II.3 Explorarea Impactului Frecvenței în Rafală a Commit-urilor asupra Calității Software-ului

Frecvența în rafală a commit-urilor (*commit burstiness*) reprezintă perioade de dezvoltare rapidă în care commit-uri succesive sunt trimise (pushed) în repozitoriu și poate fi un indicator al comportamentului dezvoltatorului și al impactului acestuia asupra calității software-ului. Acest capitol reprezintă finalul părții referitoare la „Probleme la Nivel de Cod", unde am trecut de la analizarea longitudinală a proiectelor open-source în Python cu privire la modul în care calitatea software-ului evoluează în timp, la aplicarea categoriilor CCS pentru a vedea care este mai exact scopul relativ al acelor commit-uri, iar în acest capitol vom vedea dacă fenomenul de *commit burstiness* este prevalent în setul nostru de date și ce concluzii putem extrage din el.

Acest studiu [LMBM] a fost realizat în colaborare cu studenta doctorandă Vasilica Moldovan și coordonatoarea prof. Simona Motogna, iar contribuțiile mele au constat în metodologie, execuția experimentului, discuția rezultatelor, analiza și crearea artefactului setului de date [BMM25].

II.3.1 Motivație

Gestionarea eficientă a contribuțiilor de cod este crucială, în special în proiectele open-source. Menținerea calității devine din ce în ce mai complexă pe măsură ce sistemele evoluează, subliniind necesitatea identificării timpurii a problemelor de calitate.

Frecvența în rafală a commit-urilor, caracterizată prin commit-uri rapide și succesive, a apărut ca un factor care influențează calitatea software-ului. Studiile anterioare ([NMB10, XF14, WNLB22]) au introdus rafalele de commit-uri (*commit bursts*) ca predictori ai defectelor și au subliniat legătura lor cu o probabilitate crescută de apariție a defectelor. În timp ce practicile constante de commit reflectă o dezvoltare sănătoasă, rafalele neregulate ar putea declanșa modificări pripite ale codului, ducând la degradarea calității sau la acumularea datoriei tehnice.

Clasificarea semantică a commit-urilor completează analiza rafalelor oferind o perspectivă asupra intențiilor din spatele modificărilor individuale. Specificația Convențională a Commit-urilor (CCS)

oferă etichete standardizate precum *feat*, *fix* și *refactor*, abordând limitările clasificărilor tradiționale ([Swa76, Con20, ZZQL25]).

În acest capitol, ne propunem să reducem acest decalaj prin combinarea perspectivelor din analiza rafalelor de commit-uri, etichetarea structurată a commit-urilor și metricile statice de calitate. Investigăm dacă momentul și intenția commit-urilor pot servi ca predictori pentru rezultatele de calitate raportate de SonarQube. Obiectivul nostru este triplu: să validăm dacă rafalele de commit-uri sunt predictori ai creșterii numărului de probleme, să explorăm dacă diferite categorii de commit-uri sunt mai predispuse la rafale și să verificăm dacă anumite categorii sunt mai predispuse la introducerea de probleme în timpul perioadelor de rafală comparativ cu cele fără rafală.

II.3.2 Designul Studiului

Lucrarea noastră este un studiu exploratoriu cantitativ, care urmează practicile de Inginerie Software din [Ral21, KK15]. Dorim să verificăm dacă există o relație între rafalele de commit-uri, problemele SonarQube și categoriile CCS. Spre deosebire de un studiu explicativ, care explică ceea ce se știe deja, un studiu exploratoriu verifică dacă merită urmată o nouă cale de cercetare.

Studiul este cantitativ deoarece rulăm analiza pe un set de date suficient de mare și un studiu de caz deoarece vizăm repozitoare open-source împreună cu date SonarQube. Această abordare a mai fost utilizată în Ingineria Software [AdM+23, Jm12].

Ne împărțim obiectivul în trei întrebări de cercetare:

- **RQ8**: Se corelează frecvența în rafală a commit-urilor cu o șansă mai mare de a introduce probleme de calitate a software-ului?
- **RQ9**: Sunt anumite tipuri din Specificația Convențională a Commit-urilor (CCS) mai prevalente în timpul perioadelor de rafală?
- **RQ10**: Există o corelație între anumite tipuri de commit-uri CCS și introducerea a mai multe probleme SonarQube în commit-urile de tip Rafală (*Burst*) versus Non-Rafală (*Non-Burst*)?

Selecția și Procesarea Datelor

Utilizăm setul de date Python Software Quality[1], care conține 90.318 commit-uri clasificate conform CCS din 57 de proiecte Python open-source. Am păstrat doar commit-urile care induc probleme SonarQube, ajungând la **4.784**. Am ales Python deoarece este utilizat pe scară largă în industrie, în special în domeniul AI. Commit-urile acoperă perioada 2013-2023. Wen și colab. [WNLB22] au eșantionat 500 de commit-uri în lucrarea lor, așa că noi considerăm cele 4.784 de commit-uri ca fiind suficiente pentru o analiză statistică exploratorie. Setul de date conține date despre commit-uri (mesaj, dată, clasificare CCS) și probleme SonarQube.

Detectarea ferestrei de rafală

Pentru a defini o fereastră de rafală (perioada dintre două commit-uri), am analizat lucrările anterioare. Xuan și colab. [XF14] folosesc o fereastră între 1 și 10 zile. Nagappan și colab. [NMB10] o definesc în *zile de build* (*build days*), de la una la trei zile, cu 1 până la 4 commit-uri per rafală. Wen și colab. [WNLB22] folosesc incremente de 5 minute. Noi am încercat, de asemenea, ferestre mai largi (2, 3, 6, 24, 48 și 72 de ore) și numărători de commit-uri (2 și 3 commit-uri). În cele din urmă,

[1]Disponibil la `[https://doi.org/10.6084/m9.figshare.25008653.v4](https://doi.org/10.6084/m9.figshare.25008653.v4)`

folosim cel puțin **trei commit-uri** într-o fereastră de **două ore**, deoarece a oferit o separare mai clară între activitatea de rafală și cea fără rafală și rezultate statistice mai consecvente. Diferite praguri ar putea conduce la constatări diferite.

Am implementat noi înșine detecția rafalelor. Aceasta îmbină perioadele de activitate care se suprapun în evenimente unice, astfel încât rafalele care se suprapun sunt combinate pentru a evita numărarea dublă și a reduce zgomotul. De exemplu, o rafală de la 15:00 la 17:00 este marcată ca un singur eveniment, chiar dacă se suprapune ușor cu un alt interval. Acest lucru funcționează bine pentru metrici și alerte, deoarece previne notificările multiple pentru același vârf de activitate. Odată ce o rafală este identificată, următoarea începe după fereastra fixă (de ex., 2 ore), chiar dacă activitatea continuă, ceea ce poate împărți o perioadă lungă în rafale multiple.

II.3.3 Analiza Datelor și Rezultate

RQ8: *Se corelează frecvența în rafală a commit-urilor cu o șansă mai mare de a induce probleme raportate de Sonar-Qube*?

Am utilizat testul Mann-Whitney U pentru a compara commit-urile de tip rafală cu cele non-rafală, deoarece este non-parametric și gestionează dimensiuni inegale ale grupurilor ($n_{\text{burst}} = 616$; $n_{\text{non-burst}} = 4167$). Urmând abordarea din [TEHG23], am rulat mai întâi testul Shapiro-Wilk [SW65] și am confirmat că datele nu aveau o distribuție normală, ceea ce a justificat alegerea metodei non-parametrice.

Am rulat un test bilateral (two-sided). Ipoteza nulă (H_0) a fost că nu există nicio diferență în ceea ce privește problemele induse între perioadele cu rafală și cele fără rafală. Am obținut $U = 1\,184\,240.50$ și $p = 0.0014$, deci respingem H_0. Commit-urile non-rafală au avut un număr mediu mai mare de probleme (medie $= 4.54$, mediană $= 1$) comparativ cu cele de tip rafală (medie $= 2.88$, mediană $= 1$).

Constatarea 9: Commit-urile non-rafală pot fi asociate cu mai multe probleme SonarQube decât commit-urile de tip rafală, sugerând mai multe probleme de calitate detectabile atunci când munca este distribuită în timp, mai degrabă decât grupată. Sunt necesare analize suplimentare pentru a confirma acest lucru.

RQ9: *Sunt anumite tipuri din Specificația Convențională a Commit-urilor (CCS) mai prevalente în timpul perioadelor de rafală?*

Am rulat un test de independență Hi-pătrat (Chi-square) pentru a verifica dacă categoriile CCS sunt distribuite diferit în perioadele de rafală față de cele fără rafală. Am verificat ca valoarea așteptată în fiecare celulă să fie de cel puțin 5 [McH13] și am aplicat corecția Benjamini-Hochberg pe valorile p [dTF+19].

`refactor` iese în evidență cu $\chi^2 = 67.0005$, un p ajustat mult sub 10^{-14} și V-ul lui Cramér $= 0.1184$ (o legătură slabă, dar solidă). Am observat 227 de commit-uri `refactor` în rafale și 909 în afara acestora. Doar 20% din commit-urile `refactor` apar în rafale, dar `refactor` reprezintă 36,9% din toate commit-urile de tip rafală, față de 21,8% din cele non-rafală. Așadar, refactorizarea ocupă o felie mult mai mare din activitate în timpul rafalelor. `fix`, `build`, `docs`, `style` și `ci` înclină, de asemenea, către perioadele fără rafală, dar mărimile efectului lor ($V < 0.08$) sunt neglijabile. `perf`, `feat`, `chore` și `test` sunt distribuite uniform.

Constatarea 10: Commit-urile de refactorizare tind să fie mai frecvente în timpul sesiunilor în rafală, în timp ce majoritatea celorlalte tipuri de commit-uri prezintă o preferință redusă sau inexistentă.

RQ10: *Există o corelație între anumite tipuri de commit-uri CCS și introducerea a mai multe probleme SonarQube în commit-urile de tip Rafală (Burst) versus Non-Rafală (Non-Burst)?*

Am rulat un test bilateral Mann-Whitney U pentru fiecare tip de commit CCS pentru a vedea dacă numărul de probleme SonarQube diferă între commit-urile de tip rafală și cele non-rafală. Am aplicat corecția Holm [dTF+19] pentru toate cele zece teste și am notat numărul median de probleme din fiecare grup pentru a verifica direcționalitatea.

Niciuna dintre valorile p ajustate nu a scăzut sub 0,05. `perf` a avut un p brut de 0,036 cu o reducere mediană în rafală, dar valoarea sa p ajustată prin metoda Holm a crescut la 0,364. `docs` a înclinat în direcția opusă (mai multe probleme în rafală), cu un p ajustat $= 0.869$. Toate celelalte categorii (`build`, `chore`, `fix`, `refactor`, `feat`, `style`, `test`, `ci`) au avut valori p ajustate la sau peste 0,868.

Chiar și după comparații multiple, nu există dovezi reale că vreun tip CCS introduce în mod sistematic mai multe probleme SonarQube în commit-urile de tip rafală versus cele non-rafală.

Constatarea 11: Nu există dovezi clare că un anumit tip de commit produce în mod constant mai multe probleme de calitate a software-ului în sesiunile cu rafală față de cele fără rafală.

II.3.4 Concluzii

Acest capitol a analizat modul în care frecvența în rafală a commit-urilor interacționează cu semantica commit-urilor și efectul lor asupra calității software-ului.

Pentru **RQ8**, commit-urile în rafală nu au introdus mai multe probleme decât perioadele non-rafală, contrar așteptărilor noastre inițiale. Secvențele rapide de commit-uri pot reflecta o muncă concentrată și coordonată, mai degrabă decât modificări pripite. Echipele ar putea sprijini aceste perioade cu procese de revizuire mai bune sau cu automatizare.

Pentru **RQ9**, commit-urile de refactorizare au fost deosebit de frecvente în timpul rafalelor, indicând un tipar în care refactorizarea se grupează. Echipele ar putea folosi acest lucru pentru a gestiona mai strategic eforturile de refactorizare.

Pentru **RQ10**, nu am găsit diferențe semnificative între tipurile de commit-uri și problemele SonarQube în intervalele cu rafală versus cele fără rafală. Cercetările viitoare ar trebui să analizeze severitatea problemelor și *code smells* specifice pentru a informa mai bine practicile de asigurare a calității (QA).

Constatările noastre se aliniază parțial cu [NMB10] și [NGM+19], care au semnalat rafalele ca fiind predictori de defecte, dar diferă prin sugerarea faptului că rafalele reflectă refactorizare mai degrabă decât o activitate problematică. Rezultatele se potrivesc cu [XF14] în ceea ce privește beneficiile coordonării echipei, dar contrastează cu [WNLB22], deoarece noi legăm rafalele în principal de refactorizarea sistematică, mai degrabă decât de remedieri rapide și perturbatoare.

Per ansamblu, acest studiu sugerează o regândire a presupunerilor despre modificările rapide ale codului și indică potențiale beneficii în productivitate și calitate. Munca viitoare poate include analize la nivel de proiect, interviuri cu dezvoltatorii și o privire mai atentă asupra tipurilor specifice

de probleme.

II.4 Similaritatea AST a problemelor de calitate a codului în Python și JavaScript

Trecând de la investigația longitudinală a modului în care evoluează problemele de cod, a modului în care acestea se corelează cu intenția dezvoltatorului prin categoriile CCS și a impactului rafalelor de commit-uri asupra calității generale a software-ului, acest capitol se concentrează pe structura de bază a acestor probleme utilizând Arborii Sintactici Abstracți (AST). Mai precis, acest capitol prezintă o comparație a similarității problemelor de cod la nivel de AST în două limbaje de programare: Python și JavaScript. AST-urile extrase sunt comparate folosind tehnici bazate pe procesarea limbajului natural (NLP) și distanța de editare a arborilor bazată pe grafuri. Mai mult, a fost efectuată o inspecție manuală a problemelor de tip *code smells* pentru a analiza mai bine diferențele calitative în tiparele problemelor.

Studiul pe care se bazează acest capitol a fost realizat împreună cu studenții doctoranzi Rareș Patcaș și Oskar Picuș, și cu coordonatoarea mea, prof. Simona Motogna. Contribuțiile mele au constat în extragerea, procesarea și analiza datelor, precum și în discutarea rezultatelor referitoare la comparația AST. Rezultatele au fost trimise spre publicare în [RDPM].

II.4.1 Motivație

Limbajele de programare continuă să evolueze, apărând unele noi și cele existente lansând actualizări frecvent. Python și JavaScript urmează un program anual de lansare[1][2], în timp ce Java livrează actualizări la fiecare șase luni[3]. Acest ritm pune presiune pe dezvoltatori și pe fluxurile de lucru de asigurare a calității (QA).

Calitatea software-ului rămâne o preocupare constantă și ar putea necesita și mai multă atenție odată cu codul generat de AI [CFI+24, LLCW+24]. Instrumentele de analiză statică (SAT) sunt utilizate pe scară largă pentru a surprinde problemele de calitate în stadii incipiente [MM22, LPS+23], instrumente precum SonarQube, Codacy și PMD fiind populare printre practicieni [LAL15, MVS23]. Acestea parsează codul sursă în Arbori Sintactici Abstracți și aplică reguli specifice limbajului bazate pe standarde de programare, bune practici și anti-șabloane, semnalând probleme precum *code smells*, complexitatea excesivă și încălcările convențiilor.

Acest capitol se concentrează pe similaritatea AST-urilor legate de regulile SonarQube atunci când acele reguli vizează aceeași problemă conceptuală în Python și JavaScript. O similaritate ridicată ar însemna că există valoare în transferul regulilor de la un limbaj la altul, ceea ce ar putea reduce decalajul în suportul limbajelor (750 de reguli pentru Java versus 86 pentru Ruby).

Am ales Python și JavaScript deoarece sunt populare, împărtășesc constructe similare (funcții, clase, instrucțiuni de flux de control) care permit comparații structurale în ciuda diferențelor de sintaxă și au o acoperire comparabilă a regulilor (414 reguli legate de probleme în Python, 424 în JavaScript).

Contribuțiile noastre sunt:

[1] https://peps.python.org/pep-0602/
[2] https://github.com/tc39/proposals
[3] https://openjdk.org/jeps/3

- analiza unui subset de reguli comune SonarQube între Python și JavaScript și compararea lor prin intermediul reprezentării AST
- aplicarea măsurilor de similaritate structurală și semantică pentru a caracteriza tiparele interlingvistice și provocările similarității regulilor
- evaluarea măsurii în care similaritatea bazată pe AST poate susține transferul interlingvistic al regulilor.

II.4.2 Abordare

Adoptăm metodologia GQM [BCR94] și ne definim obiectivul după cum urmează: **Analizarea** AST-urilor legate de regulile SonarQube specifice limbajului pentru Python și JavaScript **cu scopul de** a evalua similaritatea structurală și semantică între limbaje **în ceea ce privește** problemele de codare care abordează aceleași defecte conceptuale **din perspectiva** utilizatorilor instrumentelor de analiză statică **în contextul** proiectelor open-source scrise în Python și JavaScript.

Întrebarea de cercetare este următoarea: **Cât de similare sunt AST-urile asociate regulilor SonarQube din Python și JavaScript care abordează aceeași problemă de codare?** Mai multe metrici de similaritate sunt apoi definite pentru a oferi răspunsul la această întrebare de cercetare.

Metodologie

Primul pas al metodologiei este extragerea regulilor SonarQube stabilite de comun acord din proiecte open-source pentru ambele limbaje. Apoi convertim codul problemei utilizând Tree-sitter. Următorul și ultimul pas este compararea similarității, atât prin revizuire manuală, cât și prin automatizare, utilizând tehnici NLP și tehnici bazate pe grafuri.

Set de date

Am construit un set de date format din proiecte open-source în Python și JavaScript cu metadate SonarQube și fragmente de cod pentru analiza AST. Pentru JavaScript, am rulat SonarQube 10.3.0 pe repozitoare mature din studii anterioare [CBZK19, SDC24], obținând 64.591 de probleme. Pentru Python, am utilizat setul de date Python Software Quality Dataset [MBP24], bazat pe aceeași versiune SonarQube, și l-am sub-eșantionat la 457.935 de probleme *OPEN* valide pentru a reduce dezechilibrul de dimensiune.

Pentru fiecare problemă, am extras liniile afectate și le-am parsat cu Tree-sitter [LAAZ23]. Apoi am ierarhizat regulile comune SonarQube în funcție de apariția lor minimă în ambele limbaje și le-am păstrat pe primele 10, evitând regulile declanșate rar [LML22, BLRS20, MM20a]. Acestea acoperă o gamă de categorii de calitate a codului.

Analiza similarității bazată pe NLP

Am normalizat AST-urile în reprezentări Tree-sitter liniarizate și am redus diferențele de sintaxă interlingvistice prin înlocuirea deterministă a token-urilor pe etichetele AST (de ex., de la **module** la **program**, de la **function_definition** la **function_declaration**). Maparea completă se află în pachetul de replicare.

Am folosit măsuri lexicale bazate pe seturi (Jaccard, Dice, Overlap), TF-IDF cu similaritate cosinus și Sentence-BERT [RG19] pentru latura semantică.

Analiza similarității bazată pe AST

Parsăm fragmentele de cod cu Tree-sitter și normalizăm AST-urile în același mod ca în analiza NLP. Arborii sunt înveliți în obiecte `LabelTree` și trecuți către AP-TED, configurat cu costuri unitare pentru inserții și ștergeri, în timp ce operațiunile de redenumire costă proporțional cu similaritatea etichetelor nodurilor prin intermediul `difflib.SequenceMatcher` din Python.

Am rulat calcule de distanță pe perechi pe douăzeci de procese paralele (*workers*) cu salvare periodică a stării (*checkpointing*), apoi am convertit distanțele în scoruri de similaritate pe intervalul $[0, 1]$, aceeași scară ca și măsurile NLP.

Un nivel de referință (*baseline*) simplu este:

$$\text{sim}_{ij} = \frac{1}{1 + d_{ij}},$$

dar acesta ignoră dimensiunea arborelui. Astfel, utilizăm **TSED**:

$$\text{TSED}_{ij} = \max\left(1 - \frac{d_{ij}}{\max(|\text{AST}_i|, |\text{AST}_j|)}, 0\right)$$

TSED se normalizează în raport cu arborele mai mare, oferind comparații mai corecte între diferitele dimensiuni ale AST-urilor.

Deoarece AP-TED este costisitor din punct de vedere computațional, pentru fiecare regulă am luat numărul mai mic dintre cele două limbaje și am sub-eșantionat aleatoriu setul mai mare pentru a se potrivi. De exemplu, regula S1481 are 748 de instanțe în JavaScript și 34.792 în Python, așa că am ales 748 de AST-uri din Python. Acest lucru a redus timpul de execuție de la peste două zile la un nivel gestionabil, menținând în același timp reprezentativitatea.

II.4.3 Rezultate și Discuții

Rezultate ale similarității bazate pe NLP

Scorurile au fost normalizate în intervalul $[0, 1]$, unde 1 înseamnă o similaritate perfectă. Tabelele complete se regăsesc în pachetul de replicare.

Indicele Jaccard s-a situat între 0,39 și 0,45, sugerând o anumită similaritate structurală pentru majoritatea regulilor. Regulile S1135 și S125 au obținut cele mai mari scoruri, ambele referindu-se la comentarii, deoarece Tree-sitter a păstrat doar markerii structurali, nu și conținutul.

Similaritatea de tip Overlap a fost ridicată pentru toate regulile, arătând că nodurile AST sunt în mare măsură incluse unele în altele. Problemele de tip *code smells* sunt legate de utilizarea repetată a unor structuri sintactice similare în ambele limbaje.

Indicele Dice, cuprins între 0,52 și 0,8, indică o similaritate moderată: elemente comune, dar și diferențe reale derivate din stiluri de programare variate. Acest lucru consolidează faptul că problemele *code smells* nu sunt aleatoare și prezintă o uniformitate structurală la nivel interlingvistic.

Metrica TF-IDF confirmă acest lucru, în special pentru *S125* și *S1135*. Celelalte reguli au obținut scoruri scăzute, ceea ce înseamnă că fragmentele de cod care introduc *smells* diferă în ceea ce privește frecvența și distribuția token-urilor, dezvoltatorii folosind constructe structurale similare în moduri diferite pentru a introduce același *smell*.

Sentence-BERT a arătat o aliniere semantică interlingvistică limitată, cu medii cuprinse între 0,56 și 0,66.

Per ansamblu, analiza NLP a identificat tipare consistente pentru regulile referitoare la codul

comentat (S125, S1135) și pentru S1186 (antete de funcții goale). Pentru celelalte reguli, diferențele structurale dintre Python și JavaScript indică diferențe în practicile de programare sau inconsecvențe în modul în care sunt definite aceste *smells*, ceea ce lasă loc pentru investigații suplimentare.

Analiza similarității bazată pe grafuri

Rezultatele TSED raportează statistici descriptive pentru fiecare regulă, în același format ca și analiza NLP. Cele mai mari medii sunt pentru S125 (0,70) și S1135 (0,77). Pentru ambele, mediana și cuartila superioară ating valoarea 1, ceea ce înseamnă că multe perechi de AST-uri sunt identice din punct de vedere structural. Însă abaterile lor standard mari (0,34 pentru S125, 0,36 pentru S1135) arată că unele instanțe diverg mult, producând o dispersie mai largă. Așadar, aceste reguli prezintă adesea o aliniere interlingvistică puternică, dar consecvența nu este uniformă în toate exemplele.

Analiza bazată pe grafuri arată că doar unele categorii de reguli, în principal cele care se bazează pe tipare sintactice simple, au o similaritate interlingvistică mai mare și sunt candidați puternici pentru transpunerea automată. Regulile cu valori TSED mai scăzute reflectă constructe specifice limbajului și necesită o adaptare regulă cu regulă. Unele reguli pot fi generalizate, în timp ce altele necesită strategii hibride care combină normalizarea structurală și semantică.

Pe lângă S125 și S1135, majoritatea regulilor au valori medii TSED scăzute, precum S3358 (0,14), S1481 (0,22) și S1763 (0,26), care dezvăluie complexități mai profunde specifice limbajului. S3358 implică operatori ternari imbricați care arată foarte diferit la nivel de AST, S1481 semnalează variabilele locale neutilizate în sintaxe de declarare variate, iar S1763 detectează codul inaccesibil în constructe specifice limbajului. Regulile intermediare precum S1186 (0,43), S1854 (0,27) și S905 (0,29) se situează între cele două extreme, prezentând o suprapunere structurală parțială cu variații dependente de limbaj.

Per ansamblu, analiza bazată pe grafuri, asemenea celei NLP, evidențiază două grupuri distincte de reguli. Regulile cu structuri simple, independente de limbaj (cum ar fi cele referitoare la comentarii) prezintă o similaritate ridicată și pot fi transferate cu ajustări minime. Regulile care captează constructe specifice limbajului (operatori ternari imbricați, variabile neutilizate, cod inaccesibil) prezintă o similaritate mai scăzută și necesită o adaptare atentă. Această distincție arată unde este fezabilă transpunerea automată a regulilor și unde sunt necesare abordări hibride, oferind o bază concretă pentru lucrările viitoare privind adaptarea regulilor de analiză statică între limbaje.

II.4.4 Concluzii

Acest capitol a explorat diferențele structurale dintre problemele de codare extrase din SonarQube în două limbaje de programare diferite, Python și JavaScript. Folosind setul de date al problemelor, am construit AST-urile codului care a generat problemele respective și le-am comparat similaritatea folosind tehnici multiple. Scopul nostru a fost să investigăm posibilitatea traducerii problemelor de calitate între limbaje.

Am identificat trei categorii de reguli SonarQube în ceea ce privește similaritatea lor între limbaje, tiparele legate de AST și tiparele determinate de o complexitate ridicată indicând faptul că AST-urile sunt insuficiente pentru această sarcină. Descoperirile noastre arată că AST-urile singure nu pot susține transferul interlingvistic al regulilor. Din perspectivă de cercetare, această contribuție oferă lecții valoroase și sugerează că traducerea problemelor legate de calitate între limbaje este o problemă non-trivială care necesită alte abordări de cercetare. Pentru practicieni, subliniază faptul că instrumentele de analiză statică (SAT) ar putea avea o aplicabilitate limitată în detectarea problemelor de calitate între limbaje din cauza diferențelor structurale și semantice.

Partea III

Code Smells

III.1 Metode de Inteligență Artificială în Refactorizarea Software: O Recenzie Sistematică a Literaturii

Acest capitol reprezintă primul pas în explorarea refactorizării, deoarece investighează integrarea Inteligenței Artificiale și a Învățării Automate în cadrul acestui proces software. Pe parcursul său, analizăm 156 de articole publicate între 2010 și 2024 și cartografiem stadiul actual (*state-of-the-art*) al aplicațiilor metodelor AI de-a lungul diferitelor etape ale ciclului de viață al refactorizării, oferind în același timp perspective asupra tendinței publicațiilor și a principalelor direcții de concentrare în acest subdomeniu al software-ului.

Această cercetare [MBM24] a fost realizată în colaborare cu coordonatoarea mea, prof. Simona Motogna, și studenta doctorandă Vasilica Moldovan. Contribuțiile mele au constat în analiza datelor, automatizarea fluxurilor pentru extragerea datelor, metodologie și strategia de căutare, precum și amenințările la adresa validității.

III.1.1 Motivație

Refactorizarea este un proces important în dezvoltarea software, având ca scop îmbunătățirea structurii, mentenabilității și reutilizabilității codului fără a-i schimba funcționalitatea [Fow99]. A rămas un interes continuu de cercetare datorită impactului său asupra calității codului, a performanței și a consumului de energie [CBMP14, TDPT+21, VSPL18].

Ultimii ani au cunoscut un interes crescând pentru aplicarea Inteligenței Artificiale (AI) și în special a Învățării Automate (ML) în problemele de inginerie software. Acest lucru este determinat de progresul rapid al algoritmilor AI/ML și de disponibilitatea unor seturi mari de date software. Printre exemple se numără extragerea de date din repozitoarele software, detectarea de *code smells*, generarea de cazuri de testare și efortul de dezvoltare [BCRP20]. Instrumente precum WEKA [Uni] și Scikit-Learn [PVG+11] au oferit domeniului un impuls suplimentar.

Același lucru este valabil și pentru refactorizare. Prin această lucrare, dorim să oferim o imagine de ansamblu amplă asupra modului în care refactorizarea și tehnicile AI/ML interacționează. Studii anterioare au analizat refactorizarea, dar fie în ansamblu [AAK+20], fie din perspectiva atributelor de calitate [ADA18].

Scopul principal al acestui capitol este o recenzie sistematică a literaturii (SLR) care cartografiază stadiul actual al aplicării metodelor AI/ML în etapele refactorizării. SLR-ul urmează un protocol definit, cu studii primare selectate din surse ale literaturii științifice. Partajăm setul de date rezultat pentru comunitatea de cercetare. Analiza cantitativă identifică tendințele și subiectele deschise din acest domeniu.

III.1.2 Metodologia de Cercetare

Abordarea noastră privind recenzia literaturii se bazează pe [RB22], definind-o ca o recenzie exploratorie a literaturii (*scoping literature review*) cu scopul de a descrie domeniul AI/ML în relație cu refactorizarea sistemelor software și de a cartografia studiile din acest domeniu. Am investigat standardele existente în domeniu [Pau21] și studii similare în ingineria software [AAK+20, KC07, CD10,

ADA18]. De asemenea, am urmat liniile directoare descrise în [KC07] și am documentat fiecare pas, așa cum este rezumat în Tabelul III.1.1. Pașii 1 și 2 aparțin planificării, pașii 3 și 4 desfășurării recenziei, iar pasul 5 se referă la raportare. Această secțiune descrie detalii despre fiecare pas.

Tabela III.1.1: Protocolul urmat pentru SLR

Nr.	Pas	Descriere
1	Obiectiv	Impactul AI/ML asupra refactorizării
2	Întrebări de Cercetare	Care faze ale refactorizării? Care metode AI/ML?
3	Strategia de Căutare	Baze de date pentru studiile primare Ce cuvinte cheie vor fi utilizate? Ce informații sunt extrase?
4	Criterii de selecție	Criterii de includere Criterii de excludere
5	Sinteza datelor	Informații care urmează să fie sintetizate Analiza datelor

Întrebări de Cercetare

Obiectivul de cercetare constând în discutarea stadiului actual al domeniului aplicării metodelor AI-/ML în etapele ciclului de viață al refactorizării este structurat în următoarele întrebări de cercetare:

- **RQ11:** *Ce faze ale ciclului de viață al refactorizării sunt abordate?* Refactorizarea este un proces complex care încorporează mai multe faze, începând de la detectare, prioritizare și recomandare, procese care ar trebui însoțite de documentare și testare și care ar putea fi prezise. Există mai multe clasificări/denumiri ale acestor faze (cum ar fi în [BCH+05] sau [AAK+20]). Ne propunem să investigăm modul în care studiile selectate vizează aceste faze.
- **RQ12:** *Care sunt metodele AI/ML utilizate?* Explorarea noastră va investiga ce tip de metode AI sunt utilizate și care sunt cele mai frecvente metode AI/ML aplicate pentru refactorizare.

III.1.3 Rezultate și Discuții

Rezultatele pentru **RQ11** arată că majoritatea efortului de cercetare este îndreptat către etapele de detectare și recomandare din ciclul de viață al refactorizării. Detectarea rămâne cea mai populară zonă, reprezentând peste 37% dintre lucrări, în timp ce recomandarea urmează îndeaproape cu aproape 33%. În schimb, etape precum documentarea, prioritizarea și testarea sunt vizibil sub-explorate. Pentru a răspunde la **RQ12**, am descoperit că, din punct de vedere metodologic, învățarea supervizată este cea mai frecventă abordare, fiind utilizată în peste 56% dintre studii. Cele mai prevalente metode AI identificate includ Pădurile Aleatoare (*Random Forests*), Algoritmii Genetici, Mașinile cu Vectori de Suport (*Support Vector Machines*), Rețelele Neuronale Convoluționale (*CNNs*) și Arborii de Decizie. În mod specific, CNN-urile sunt preferate pentru sarcinile de detectare, în timp ce Algoritmii Genetici sunt utilizați predominant pentru recomandările de refactorizare.

Implicațiile acestor constatări sunt următoarele:

- Peste 72% din cercetările analizate se concentrează pe propunerea de noi soluții, în timp ce evaluarea, validarea și cercetarea bazată pe experiență sunt subreprezentate. Este necesară o validare empirică mai mare pentru ca aceste soluții AI să fie adoptate în practicile software din lumea reală.

- O concentrare mai mare pe Prioritizare și Predicție ar putea reduce semnificativ costurile de dezvoltare și ar putea îmbunătăți performanța.
- Interesul crescut pentru metodele AI hibride sugerează că cercetătorii combină mai multe tipuri de învățare pentru a îmbunătăți fiabilitatea și adaptabilitatea modelelor.

III.1.4 Concluzii

Acest capitol prezintă o recenzie sistematică a literaturii (SLR) care examinează utilizarea AI în refactorizarea software, urmând liniile directoare stabilite [KC07]. Dintr-un grup inițial de 1.986 de lucrări, au fost reținute 156 de studii, conturând stadiul cercetării și adoptării AI/ML în refactorizare. Tendințele din 2010 până în 2024 arată o creștere constantă, cu o accelerare notabilă după 2016 și vârfuri în 2022-2023, reflectând un interes susținut al comunității pentru aplicarea AI/ML în refactorizare.

Am abordat întrebările de cercetare de bază referitoare la etapele ciclului de viață care sunt tratate de metodele AI și la tehnicile care domină. Cele mai multe contribuții se concentrează pe *detectare*, urmată de *recomandare* și *predicție*. *Prioritizarea* și *predicția* par deosebit de promițătoare pentru îmbunătățirile conduse de ML, având potențialul de a reduce costurile de dezvoltare [Fow99]. Abordările hibride sunt, de asemenea, explorate din ce în ce mai mult, având ca scop creșterea acurateței și robusteții prin combinarea metodelor.

Un decalaj cheie constă în deficitul de studii de evaluare, validare și experiență, care sunt esențiale pentru adoptarea în lumea reală. Cele mai comune metode AI sunt Pădurile Aleatoare, Algoritmii Genetici, Mașinile cu Vectori de Suport (SVM), Rețelele Neuronale Convoluționale (CNN) și Arborii de Decizie, aplicate diferit de-a lungul etapelor ciclului de viață, în funcție de adecvarea la sarcină. Majoritatea modelelor (56,41%) utilizează învățarea supervizată, urmată de abordări nesupervizate și semi-supervizate, indicând beneficiile datelor etichetate pentru obținerea unor predicții precise.

III.2 Mentenabilitatea Software-ului și Predicția Refactorizărilor pe Baza Problemelor de Datorie Tehnică

Mentenabilitatea software-ului rămâne un factor decisiv în ciclul de viață al dezvoltării software, având un impact direct asupra costurilor, timpului și alocării resurselor. Prin urmare, pe măsură ce software-ul evoluează, refactorizarea acestuia devine esențială pentru a îmbunătăți calitatea, lizibilitatea și extensibilitatea. În acest subcapitol, ne propunem să explorăm două direcții principale: în primul rând, predicția mentenabilității software-ului și a numărului de refactorizări de cod necesare, valorificând problemele de datorie tehnică și Învățarea Automată. În al doilea rând, o extensie a obiectivului anterior, dar prin compararea directă a problemelor SonarQube din trei proiecte între diferite versiuni: TuxGuitar, JEdit și FreeMind, unde ne propunem să observăm cum evoluează un proiect și care sunt categoriile de probleme care se schimbă între versiuni. Ca atare, ne vom concentra pe al doilea obiectiv și îl vom menționa pe scurt pe primul.

Acest studiu [BM23] a fost realizat în colaborare cu colega mea, Vasilica Moldovan, iar contribuțiile mele au fost legate de analiza aprofundată a problemelor SonarQube din cele trei proiecte analizate.

III.2.1 Motivație

A ieșit la iveală faptul că există o legătură puternică între mentenabilitatea sistemelor software și problemele de datorie tehnică. Aceste probleme pot fi identificate utilizând instrumente de analiză statică (SAT), cum ar fi SonarQube. La o primă vedere, un număr mic de probleme este responsabil pentru un procent mai mare din mentenabilitate și viceversa. Prin urmare, una dintre problemele codului descoperite în procesul de analiză statică este reprezentată de datoria tehnică.

Cercetările confirmă o corelație puternică între problemele software detectate, mentenabilitate și refactorizare [ABJ10, Mar04, WDS16]. Refactorizarea este importantă pentru reducerea datoriei tehnice și îmbunătățirea scalabilității; atunci când problemele excesive degradează mentenabilitatea, aceasta servește la restabilirea calității codului. Pentru a susține acest lucru, industria utilizează instrumente de analiză statică, cum ar fi SonarQube, PyLint și FindBugs[1], pentru a identifica datoria, alături de utilitare precum [2], care detectează refactorizările între versiuni. Unele dintre aceste instrumente sunt, de asemenea, descrise în I.1.

Obiectivul studiului realizat este dublu: să prezică numărul de refactorizări care trebuie efectuate și, pe baza acestui număr, să clasifice mentenabilitatea fiecărei componente software care a fost refactorizată folosind algoritmi ML. Al doilea obiectiv este de a compara în profunzime problemele SonarQube pentru trei proiecte Java prin executarea SonarQube pe versiunile lor, extragerea problemelor și rularea unei analize statistice cuprinzătoare asupra acestora.

Totuși, accentul acestui subcapitol este pus pe cel de-al doilea obiectiv, deoarece acesta reprezintă contribuția mea semnificativă, și prezintă rezultatele primului obiectiv.

III.2.2 Predicția Refactorizării: Experiment și Rezultate

Experimentul s-a concentrat pe colectarea și pregătirea datelor pentru a prezice mentenabilitatea software-ului și numărul de refactorizări necesare. Am rulat o analiză cuprinzătoare pe trei proiecte Java open-source: jEdit[3], FreeMind[4] și TuxGuitar[5]. Am analizat datoria tehnică din jEdit 5.5, FreeMind 1.0.1 și TuxGuitar 1.5.2, precum și refactorizările efectuate în versiunile următoare (jEdit 5.6, FreeMind 1.1.0, TuxGuitar 1.5.3). Setul de date privind datoria tehnică fusese deja stabilit în lucrări anterioare [Mol20, MM20a].

Pentru a construi setul de date, am rulat SonarQube pe proiecte pentru a extrage atributele datoriei tehnice (severitate, datorie, tipul problemei) și RefactoringMiner pentru a număra refactorizările efectuate pe fiecare componentă între versiuni, lucru care a servit drept *groundtruth* (adevăr de bază). Setul de date final a îmbinat aceste surse, legând metricile datoriei tehnice de numărul de refactorizări per componentă. Am împărțit datele în 70% pentru antrenare și 30% pentru testare.

Am formulat problema predicției atât ca pe o sarcină de clasificare, cât și de regresie. Pentru clasificare, componentele au fost împărțite în trei clase de mentenabilitate pe baza numărului de refactorizări: „Foarte bună" ("Great") (< 5 refactorizări), „Bună" ("Good") (5–20) și „Slabă" ("Poor") (> 20). Am încercat două abordări pentru reprezentarea datelor.

Prima abordare, cu date comprimate la nivel de componentă, a oferit performanțe predictive mai bune. ExtraTrees Classifier a fost cel mai bun model, cu acuratețea, sensibilitatea (*recall*), precizia și Scorul F1 toate la 0,92. Pentru regresie, ExtraTrees Regressor s-a clasat, de asemenea, pe primul loc,

[1], https://spotbugs.github.io/
[2]RefactoringMiner, https://github.com/tsantalis/RefactoringMiner
[3]jEdit, http://www.jedit.org/
[4]FreeMind, https://freemind.sourceforge.net/
[5]TuxGuitar, https://sourceforge.net/projects/tuxguitar/

cu un R-pătrat de 0,92 și un RMSE de 2,75. Rețeaua Neuronală Recurentă (RNN) din a doua abordare a fost mai puțin eficientă, atingând doar o acuratețe a clasificării de 0,57 și un R-pătrat de 0,50.

Analizând clasificarea pe categorii, clasa „Bună" a avut cele mai mari metrici, în timp ce clasa „Slabă" a avut cele mai mici valori, reflectând dezechilibrul din setul de date inițial. Am comparat predicțiile cu Indicele de Mentenabilitate (MI) [OH92]. Au existat unele neconcordanțe: de exemplu, MI a clasificat FreeMind ca fiind „Slab" ("Bad"), în timp ce rezultatele noastre au sugerat contrariul, ceea ce susține ideea că Indicele de Mentenabilitate tradițional ar putea să nu capteze cu precizie complexitățile sistemelor orientate pe obiect [MM21].

III.2.3 O privire mai profundă asupra problemelor SonarQube

Pentru a prezenta specificul problemelor SonarQube și a le compara între versiuni, am observat că toate cele trei seturi de date au arătat îmbunătățiri în ceea ce privește numărul total de probleme, sugerând o refactorizare continuă. Cu toate acestea, calitatea schimbărilor a diferit semnificativ. **JEdit** a avut cel mai mare număr de probleme și, deși numărul total de probleme a scăzut de la 139.905 în versiunea 5.5 la 63.899 în versiunea 5.6, încălcările regulilor de tip BUG au crescut de la 2,86% la 9,59%, ceea ce înseamnă că, per ansamblu, calitatea codului a scăzut pe măsură ce severitatea a crescut. **Freemind** a înregistrat un progres constant, cu o îmbunătățire de 20% a calității codului pe măsură ce numerele brute au scăzut, menținând o proporție similară de 97% *code smells* și 2% erori, deși indicele său de mentenabilitate a rămas „Slab", arătând că s-ar putea ca metrica să nu se aplice bine în studiile empirice. **TuxGuitar** a obținut cel mai bun rezultat, cu fiecare categorie de probleme scăzând de la versiunea 1.5.2 la 1.5.3 și o scădere a distribuției erorilor, reflectând un indice de mentenabilitate „Satisfăcător". Revenind, calitatea generală poate scădea în ciuda unui număr total mai mic de probleme dacă numărul erorilor severe crește, așa cum s-a văzut în JEdit, în timp ce proiecte precum TuxGuitar arată că munca de refactorizare dă roade, rezultând în mai puține erori, mai puține vulnerabilități și o mentenabilitate mai bună.

III.2.4 Concluzii

Acest subcapitol a analizat legătura predictivă dintre datoria tehnică și mentenabilitatea software-ului, utilizând un set de date din trei proiecte Java open-source: jEdit, FreeMind și TuxGuitar. Am combinat analiza statică SonarQube cu datele RefactoringMiner pentru a testa două abordări de predicție a numărului de refactorizări și de clasificare a componentelor.

Primul obiectiv a arătat că o comprimare a problemelor de datorie tehnică în valori medii la nivel de componentă depășește performanța analizării secvențelor granulare de probleme. Extra-Trees Classifier a atins o acuratețe de 0,92, mult peste RNN (0,57). ExtraTrees Regressor a condus, de asemenea, în sarcina de regresie, cu un R-pătrat de 0,92. Metricile agregate referitoare la severitate și la tipul problemei par solide pentru anticiparea nevoilor de mentenanță, în timp ce complexitatea secvențială a problemelor individuale poate adăuga zgomot.

Al doilea obiectiv a arătat că toate cele trei proiecte au avut mai puține probleme totale între versiuni, dar *calitatea* modificărilor a variat. jEdit a prezentat o tendință îngrijorătoare: mai puține *code smells*, dar erorile au crescut de la 2% la 9%, indicând o scădere a stabilității. TuxGuitar a avut

cea mai bună traiectorie, cu reduceri în toate categoriile și un indice de mentenabilitate mai bun.

III.3 O Evaluare Comparativă a Metodelor Inteligente pentru Detectarea Problemelor de Tip "Code Smell" în Python

În timp ce obiectivul capitolului precedent a fost de a investiga predicția numărului de refactorizări necesare, în vederea îmbunătățirii mentenabilității componentelor software utilizând date despre datoria tehnică, mergem mai departe pentru a atinge obiectivul acestui capitol: și anume, de a evalua șapte dintre cele mai utilizate modele de învățare automată și un algoritm genetic pentru detectarea problemelor de tip *code smell*. Pe parcursul studiului prezentat în continuare, vom examina și compara performanța fiecărui model pentru zece *code smells* distincte, vom identifica modelul cel mai potrivit pentru fiecare *code smell* analizat, vom evidenția punctele forte și limitările și, în final, vom augmenta setul de date din [MBP24] prin adăugarea de noi valori ale metricilor și de clasificări ale *code smells*.

Acest studiu [VAMM] a fost realizat în colaborare cu coordonatoarea mea, prof. Simona Motogna, și studenta doctorandă Vasilica Moldovan. Contribuțiile mele au constat în crearea setului de date, metodologie și pregătirea experimentului.

III.3.1 Motivație

Un "*Code smell*" este orice problemă la nivelul bazei de cod care poate duce la probleme mai critice [Fow99]. Surprinderea timpurie a acestor *smells* are un impact pozitiv asupra sistemului final [Fow99, TPB+17]. Deoarece *smells* sunt mai degrabă descrise decât definite formal, implementările diferitelor instrumente oferă rezultate diferite [LPS+23].

Instrumentele de analiză statică (SAT) detectează eficient aceste *code smells* [Ao21]. Folosim termenul de "avertismente *code smell*" (*code smell warnings*) pentru ceea ce raportează SAT-urile.

De asemenea, *code smells* depind de limbaj [BHM+19, TPB+17, SMKA17]. Creșterea popularității limbajului Python ridică problema gradului în care soluțiile existente se potrivesc cu specificul său. Problemele de tip *code smell* din Python pot lua forme care sunt greu de descris și detectat [CCMX16, CCM+18].

Inteligența Artificială (AI) a influențat profund Ingineria Software, iar detectarea *code smells* a beneficiat de modelele ML [MBM24]. O recenzie sistematică a literaturii [MBM24] a semnalat o nevoie clară de evaluare. Performanța variază foarte mult în funcție de modele, seturi de date, limbaje și tipuri de *smell* [DNPT+18]. Rezultatele pentru Python pot diferi de cele pentru Java din cauza tipizării dinamice și a tranziției de la Python 2 la 3 [TFAL21].

Scopul principal al acestui studiu este de a evalua cât de potrivite sunt modelele de detectare ML existente pentru problemele de tip *code smell* în proiecte Python mari.

Contribuțiile noastre sunt:

- O evaluare critică a 7 modele ML și a unui algoritm GA pentru detectarea *code smells*;
- O comparație a performanței fiecărui model pentru zece *code smells*;

- Cel mai potrivit model pentru fiecare *smell*;
- O versiune îmbunătățită, disponibilă public, a setului de date din [MBP24].

III.3.2 Designul studiului

Acest capitol prezintă o cercetare de evaluare, urmând definiția din [WMMR06], care evaluează diferite metode inteligente de detectare a *code smells* în proiecte Python de mari dimensiuni. În cazul cercetării de evaluare, accentul nu se pune pe noutatea tehnicii propuse, ci pe relevanța cunoștințelor revendicate de evaluare. Cercetarea de tip evaluare este recomandată [WMMR06] atunci când se investighează tehnicile existente în practică (în cazul nostru, modele ML/AI) care abordează aceeași problemă (în cazul nostru, detectarea *code smells*). Rezultatele acestui tip de cercetare oferă o comparație critică și dovezi legate de aceste tehnici, utilizând date și indicatori independenți și oferind o opinie independentă pentru mai multe soluții existente.

Formulăm întrebările noastre de cercetare și conducem studiul pentru a exprima observații cu privire la modul în care diferitele modele AI luate în considerare se comportă pentru problema specifică a detectării *code smells* în proiectele Python.

Ne propunem să studiem potrivirea celor mai utilizate metode AI în detectarea problemelor de tip *code smell*. Ca metodologie, experimentul nostru explicativ, așa cum este descris în [JCP08], urmează abordarea Goal-Question-Metric (GQM) [BCR94]:

Scop: *A evalua*
Problema: *potrivirea*
Obiectiv: *metodelor ML/AI pentru detectarea "code smells"*
Context: *în proiecte Python*
Punct de vedere: *din perspectiva metricilor de evaluare a performanței*

Scopul nostru este detaliat în continuare în următoarele întrebări de cercetare:

- **RQ13:** *Care este performanța metodelor ML pentru rezolvarea problemei de detectare a "code smells"?*
- **RQ14:** *Care este potrivirea metodelor ML pentru fiecare "code smell"?*

Experimentul este proiectat după cum urmează:

- definirea setului de *code smells* ce urmează a fi detectate utilizând regulile din pachetul de replicare
- crearea unui set de date cu ajutorul a trei instrumente: Radon, PySmell, Understand
- utilizarea metricilor extrase pentru a atinge pragurile pentru anumite *code smells* selectate
- antrenarea și evaluarea modelelor AI/ML alese (DT, SVM, NB, LSTM, RF, CNN, kNN, GA) și producerea rezultatelor
- compararea rezultatelor în ceea ce privește performanța și potrivirea lor.

III.3.3 Analiză, Evaluare și Rezultate

Pentru toate modelele utilizate, am folosit două configurații ale setului de date: 80% pentru antrenare și 20% pentru testare. Singura excepție este algoritmul genetic, care a utilizat o distribuție a setului de date de 60% pentru antrenare, 20% pentru validare și 20% pentru testare.

Așa cum s-a precizat anterior, pentru fiecare model am calculat valorile pentru acuratețe (*accuracy*), sensibilitate (*recall*), precizie (*precision*) și Scorul F1 (*F1-Score*) și le-am comparat.

RQ13: *Care este performanța metodelor AI/ML pentru rezolvarea problemei de detectare a "code smells"?*

Răspunsul la această întrebare a venit din evaluarea individuală a fiecărui model. O scurtă descriere a rezultatelor va fi prezentată pentru fiecare experiment.

Pentru *Arborii de Decizie* (Decision Trees), omiterea tehnicilor de echilibrare a dus la o acuratețe și o precizie mai mari, în timp ce aplicarea echilibrării a îmbunătățit sensibilitatea (*recall*) pentru clasele minoritare. Pentru Pădurile Aleatoare (Random Forests), arborii mai puțin adânci i-au depășit în mod constant pe cei mai adânci în ceea ce privește precizia și scorul F1, sugerând că limitarea adâncimii împiedică modelul să capteze zgomotul.

Modelul *kNN* cu $k = 5$ s-a dovedit superior celui cu $k = 10$ în privința preciziei și a scorului F1. În experimentele SVM, nucleul (*kernel-ul*) liniar a depășit constant nucleul RBF la toate metricile.

Gaussian Naive Bayes a apărut ca un model performant pentru datele dezechilibrate, oferind o precizie și un *recall* mai mari decât alte variante. Natura sa probabilistică i-a permis să gestioneze clasele minoritare acolo unde alte modele au întâmpinat dificultăți.

Modelele *LSTM* și *CNN* au performat, în general, mai bine atunci când au utilizat o funcție de pierdere (*loss function*) de tip Sigmoid. Cu toate acestea, aceste modele complexe au avut frecvent performanțe inferioare comparativ cu modelele mai simple, bazate pe instanțe sau probabilistice, din cauza dimensiunii limitate și a dezechilibrului ridicat al setului de date.

Pentru *Algoritmul Genetic*, creșterea iterațiilor de la 50 la 100 a condus la o optimizare îmbunătățită pentru aproape toate metricile.

Constatarea 12: Analiza indică o variabilitate semnificativă în performanța modelelor, influențată de dezechilibrul și complexitatea setului de date. Modelele mai simple depășesc în general modelele neuronale complexe, subliniind eficacitatea abordărilor bazate pe instanțe și a celor probabilistice pe seturi de date limitate și dezechilibrate.

RQ14: *Care este potrivirea metodelor ML pentru fiecare "code smell"?*

În ceea ce privește această întrebare, rezultatele indică faptul că niciun model nu este optim la nivel universal. *kNN* a obținut cele mai mari scoruri F1 pentru șase din zece categorii, inclusiv **LongMethod** și **LongMessageChain**. Pe de altă parte, *Naive Bayes* a fost cel mai eficient pentru *smells* subreprezentate precum *LongScopeChaining* și *LongLambdaFunction*. Validarea statistică prin intermediul testului Nemenyi a confirmat că, deși *NB* și *kNN* sunt extrem de competitive, diferența de performanță dintre ele este adesea nesemnificativă statistic. Cu toate acestea, ambele depășesc semnificativ abordările de învățare profundă (*deep learning*) în acest context specific.

Constatarea 13 Deși anumite modele AI demonstrează puncte forte în detectarea anumitor *code smells*, performanța generală rămâne inadecvată pentru o implementare practică de încredere din cauza scorurilor F1 moderate. Probleme precum dezechilibrul setului de date și disponibilitatea limitată a datelor au un impact substanțial asupra preciziei detectării, subliniind necesitatea unor seturi de date echilibrate și a unei selecții și configurări atent adaptate a modelelor.

III.3.4 Concluzii

Acest capitol evaluează cât de potrivite sunt opt modele AI/ML pentru detectarea problemelor de tip *code smell* în proiecte Python. Scopul este dublu: de a afla performanța metodelor AI/ML și de a determina potrivirea lor pentru fiecare *code smell* în parte.

Am creat un set de date format din 23 de proiecte Python din [MBP24]. Am extras metrici de cod utilizând Radon [Rub24] și Understand [ST24] și am detectat *code smells* utilizând PySmell [Che24] pentru a construi setul de antrenare. Am scanat 1359 de lansări (*releases*), obținând un set de date final de 3.191.908 intrări.

Am efectuat o evaluare comparând performanța modelelor și evaluând cât de potrivit este fiecare model pentru detectarea unor *code smells* specifice. Constatările indică faptul că nu există o soluție universală pentru detectarea *code smells*. Variabilitatea performanței este afectată de configurația modelului, de proprietățile setului de date și de metricile de evaluare. Algoritmii genetici au produs rezultate încurajatoare atunci când au fost ajustați cu un număr mai mare de iterații.

În ceea ce privește eficacitatea pe diferite tipuri de *code smell*, modelele mai simple, precum kNN și NB, depășesc frecvent abordările mai complexe precum LSTM și CNN. Cu toate acestea, rezultatele sunt puternic afectate de distribuția neuniformă a setului de date și de configurațiile specifice ale modelelor.

Rezultatele sunt valoroase atât pentru cercetători, cât și pentru practicieni, deoarece concentrarea asupra calității codului în proiectele Python este un subiect relevant. Pentru cercetători, evaluarea sprijină identificarea abordărilor eficiente de detectare și deschide noi direcții pentru investigații viitoare. Pentru practicieni, oferă îndrumări practice privind alegerea și configurarea modelelor.

Partea IV

Concluzii si muncă viitoare

IV.1 Concluzii Finale

Toate capitolele de până în acest punct au avut menirea de a sublinia importanța calității software-ului în domeniul ingineriei software. Pe parcursul acestei teze, am trecut de la construirea de instrumente pentru extragerea datelor, la crearea de seturi de date și rularea de experimente empirice privind clasificarea commit-urilor, comportamentul dezvoltatorilor, detectarea problemelor, AST-uri și predicția defectelor, cu scopul de a genera noi cunoștințe în literatura de specialitate. Toate aceste cunoștințe sunt menite să îi ajute pe dezvoltatori să scrie un cod mai bun, să acorde atenție defectelor software și să urmeze recomandările generale pentru bazele lor de cod. De asemenea, se intenționează să ajute cercetătorii să analizeze mai bine acest domeniu, să extrapoleze din constatările noastre, să construiască pe baza seturilor noastre de date și să atingă noi culmi în ceea ce privește calitatea software-ului.

IV.1.1 Principalele Contribuții și Impactul

În paragrafele următoare, îmi voi sublinia contribuțiile și voi explica modul în care fiecare piesă se potrivește cu celelalte pentru a construi cercetarea completă pe care ne bazăm.

- PARTEA 1: Fundamente - În această parte, ne-am concentrat pe fundamentul teoretic pe care ne-am construit studiile, pe aplicația care ne-a permis să construim setul de date și pe setul de date în sine. Acest lucru pregătește terenul pentru studiile din capitolele ulterioare.
  - În capitolul **PyDs Builder**[Ber25], contribuția mea a constat în **construirea mecanismului și a instrumentului în sine, cu tot ceea ce implică acest lucru.** Artefactul aplicației este disponibil public în [Ber24].
  - În capitolul **Setul de Date privind Calitatea Software-ului în Python** (**Python Software Quality Dataset**)[MBP24], contribuția mea a fost crearea artefactului setului de date [BMP24].

  Toate seturile de date (chapter) sunt disponibile public, sub licența CC BY 4.0, respectând bunele practici recomandate de comunitatea de cercetare și încurajând investigarea ulterioară a datelor noastre. Instrumentul pe care l-am dezvoltat ne-a permis să construim seturi de date într-un mod mai fluid și, de asemenea, să facilităm crearea de noi seturi de date de către alți cercetători, specifice nevoilor lor.
- PARTEA 2: Probleme la Nivel de Cod (*Code Issues*) - În această parte, am utilizat setul de date creat anterior pentru a rula experimente empirice, cum ar fi analiza longitudinală a evoluției problemelor SonarQube, clasificarea commit-urilor CCS și intenția dezvoltatorului, analiza frecvenței în rafală a commit-urilor (*commit burstiness*) și modul în care aceasta se raportează la clasificarea commit-urilor, precum și încercarea de a traduce încălcările regulilor SonarQube între Python și JavaScript. Primele trei studii sunt înlănțuite, în special prin seturile de date generate de fiecare studiu care sunt utilizate succesiv, așa cum se poate observa în diagramă, în timp ce ultimul este independent.
  - În capitolul **Studiu Longitudinal**[BMM26b], contribuția mea a fost **crearea artefactului setului de date [BMM26a] și analiza statistică inferențială pentru a treia întrebare de**

cercetare.

- În capitolul **Clasificarea Commit-urilor**[BMPM25], contribuția mea a constat în **crearea artefactului setului de date [BMMP25], precum și în analiza statistică și rezultatele pentru primele două întrebări de cercetare**. De asemenea, am evaluat și rulat modelul LLM pentru detectarea CCS.
- În capitolul **Frecvența în Rafală a Commit-urilor** (**Commit Burstiness**), contribuția mea a fost **crearea artefactului setului de date [BMM25] și analiza și rezultatele pentru cele trei întrebări de cercetare**. Unul dintre pașii importanți în obținerea rezultatelor a fost proiectarea și implementarea algoritmului care ne-a permis să legăm problemele SonarQube de commit-ul care le-a indus, între două lansări consecutive.
- Ultima piesă a acestei părți, și anume articolul AST, a utilizat o abordare diferită pentru a valida dacă problemele de tip *code smells* pot fi transferate interlingvistic. Contribuția mea aici a constat într-o parte a setului de date creat, interpretarea analizei AST și investigarea manuală a trei reguli (S3776, S3558, S1763).

Munca noastră aduce dovezi empirice referitoare la compoziția și distribuția problemelor SonarQube, la evoluția pe termen lung a problemelor și investigarea intenției dezvoltatorilor și a rafalelor de commit-uri, împreună cu analiza bazată pe AST între diferite limbaje de programare. Pentru comunitatea practicienilor, am oferit analiza aprofundată a proiectelor într-o manieră multidimensională, legând problemele de calitate de commit-uri. Lucrarea noastră poate contribui la creșterea gradului de conștientizare cu privire la prevalența *code smells*, erorilor sau vulnerabilităților pe care dezvoltatorii ar trebui să le prioritizeze pentru remediere.

- PARTEA 3: Code Smells - Această parte s-a concentrat mai mult pe predicția *code smells* și a refactorizării prin intermediul modelelor ML. Rolul meu aici a fost mai mult despre extragerea și pregătirea datelor pentru a fi utilizate în antrenament, în timp ce coautorii mei au rulat experimentele ML.
  - În articolul SLR [MBM24], contribuția mea a fost crearea setului de date și proiectarea metodologiei.
  - În articolul privind Predicția Mentenabilității [BM23], următoarea piesă din diagramă, **contribuția mea a constat în procesarea, analiza și interpretarea SonarQube a celor trei aplicații**.
  - În al treilea și ultimul studiu din partea a 3-a, accentul se mută pe evaluarea metodelor predictive ML pentru detectarea *code smells*. Contribuția mea aici a fost **proiectarea metodologiei și crearea setului de date**, alimentând datele în modelele ML.

  Contribuția acestei părți reprezintă o cercetare de tip evaluare pentru utilizarea metodelor ML în problemele de inginerie software, cum ar fi detectarea *code smells* și predicția refactorizării. Din perspectiva unui practician, deschide oportunitatea de a construi instrumente specializate pe baza rezultatelor noastre.

Toate părțile acestei teze pot fi agregate într-o singură constatare: **calitatea software-ului în proiectele open-source este predominant o problemă de mentenabilitate, determinată de *code smells*, modelată de comportamentul dezvoltatorilor și abordabilă printr-o combinație de analiză statică și modele ML/LLM.** S-a prezentat faptul că problemele de tip *code smells* domină peisajul problemelor pe toate dimensiunile temporale: de la commit-uri individuale la evoluții pe parcursul anilor. Intenția dezvoltatorului, așa cum este captată prin clasificarea commit-urilor, dezvăluie că adăugările

de funcționalități și refactorizarea sunt principalii purtători de noi probleme; cu toate acestea, perioadele de dezvoltare în rafală nu agravează acest efect. Aceste constatări, bazate pe peste 90 de proiecte și aproximativ 90.000 de commit-uri clasificate, formează un argument puternic pentru integrarea unui feedback de calitate axat pe commit-uri și condus de analiza statică în fluxurile de lucru de dezvoltare de zi cu zi. Așa cum am arătat cu clasificarea commit-urilor bazată pe LLM (scor F1 de 76%), Modelele de Limbaj de Mari Dimensiuni pot deja să îmbunătățească fluxul calității software-ului prin automatizarea sarcinilor care anterior erau manuale, cum ar fi categorizarea intenției dezvoltatorului la scară largă. Pe măsură ce aceste modele continuă să se îmbunătățească, rolul lor în detectarea problemelor, prioritizarea și chiar recomandările automate de refactorizare se va extinde probabil și mai mult.

Din perspectivă metodologică, teza cuprinde diferite metode empirice în Ingineria Software [Pau21]: recenzia sistematică a literaturii [MBM24], studii de caz [BM23], experimente [BMPM25, LMBM, VAMM] și evaluări longitudinale [BMM26b].

Din perspectiva tipului de cercetare [PFMM08], am realizat cercetări de: soluție [Ber25, MBP24], validare [RDPM], evaluare [BMM26b, VAMM, BM23] și exploratorii [BMPM25, LMBM].

IV.1.2 Muncă Viitoare

Pe baza rezultatelor pe care le-am obținut până acum și a expertizei pe care am acumulat-o, prevedem două direcții importante care pot ghida munca viitoare:

Utilizarea instrumentelor AI Generative în rezolvarea problemelor specifice de inginerie software, pe baza instrumentelor și a seturilor de date pe care le deținem.

- Instrumentul PyDs poate fi extins pentru a suporta extragerea de date din diferite proiecte, pentru a executa diferite SAT-uri și a extrage date din ele. O altă direcție aici este implementarea unor sisteme de evaluare comparativă (*benchmarking*) pentru LLM-uri, cu scopul de a evalua modul în care unele modele funcționează la diferite detectări, predicții sau recomandări atunci când vine vorba de calitatea software-ului.

- Setul de date poate fi augmentat cu probleme în diferite limbaje, cum ar fi Ruby, Rust și Go, unele care nu sunt atât de acoperite în literatura de specialitate, cum ar fi Java. De asemenea, poate viza proiecte din diferite domenii software. Putem utiliza LLM-urile pentru a genera date sintetizate pentru experimente viitoare.

- Analiza clasificării commit-urilor CCS poate valorifica LLM-uri mai noi pentru a îmbunătăți scorul F1 de 76% obținut cu CodeLlama, permițând o analiză cu o granularitate mai fină și mai de încredere a intenției dezvoltatorului.

- Studiul AST poate fi augmentat prin explorarea unor reprezentări de cod mai bogate, cum ar fi grafurile de dependență a programului sau înglobările de cod (*code embeddings*), care captează similaritățile semantice acolo unde AST-urile pure au eșuat. În acest fel, putem introduce LLM-uri pentru a încerca să captăm similaritățile, explorând dacă această direcție ar putea aduce rezultate pozitive.

Analiza comportamentului dezvoltatorilor și a tiparelor de commit-uri în sistemele open-source.

- Studiul longitudinal poate fi reprodus în alte limbaje, pentru a determina dacă dominanța problemelor *code smells* și tiparele lor temporale sunt specifice limbajului sau universale.

- Analiza frecvenței în rafală a commit-urilor (*Commit Burstiness*) ar trebui să examineze tipuri specifice de probleme și să integreze perspective calitative pentru a înțelege mai bine impactul tiparelor de commit asupra calității software-ului.
- Augmentarea rezultatelor existente cu o analiză calitativă a comportamentului dezvoltatorilor (sondaje și interviuri).

Direcția generală ar trebui să fie împachetarea fluxurilor de clasificare și analiză în instrumente integrate CI/CD care să ofere dezvoltatorilor feedback privind calitatea în timp real, la nivel de commit. Acest lucru va ajuta la reducerea decalajului dintre cercetare și practică.

Bibliografie

[AAK+20] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *CoRR*, abs/2007.02194, 2020.

[AAM16] Saleh Almugrin, Waleed Albattah, and Austin Melton. Using indirect coupling metrics to predict package maintainability and testability. *Journal of Systems and Software*, 121:298–310, 2016.

[ABJ10] Erik Arisholm, Lionel C. Briand, and Elvira B. Johannessen. An empirical study on the relationship between software maintainability and bug-proneness. In *2010 IEEE International Symposium on Software Metrics (METRICS)*. IEEE, 2010.

[ADA18] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69, 2018.

[AdM+23] Andrei Ahonen, Marea de Koning, Tyrone Machado, Reza Ghabcheloo, and Outi Sievi-Korte. An exploratory study of software engineering in heavy-duty mobile machine automation. *Robotics and Autonomous Systems*, 165:104424, 2023.

[ALRT19] Francesca Arcelli Fontana, Valentina Lenarduzzi, Riccardo Roveda, and Davide Taibi. Are architectural smells independent from code smells? an empirical study. *Journal of Systems and Software*, 154:139–156, 2019.

[ALT+21] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2021.

[Ao21] Paris Avgeriou and o. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software*, PP, 05 2021.

[BASB17] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 251–262, 2017.

[BCH+05] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, 2005.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. 1994.

[BCRP20] Olimar Borges, Julia Couto, Duncan Ruiz, and Rafael Prikladnicki. How machine learning has been applied in software engineering? In *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 306–313. INSTICC, SciTePress, 2020.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Ber16] Robert A. Berenson. If you can't measure performance, can you improve it? *JAMA Forum*, 2016. Discusses the oft-quoted management adage and its attribution issues.

[Ber24] Liviu Marian Berciu. Pyds builder, May 2024.

[Ber25] L.-M. Berciu. Pydsbuilder - a dataset builder written in python django. *Studia Universitatis Babeș-Bolyai Informatica*, 69(2):5–22, 2025.

[BHM+19] Emery Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Transactions on Programming Languages and Systems*, 41:1–24, 10 2019.

[BLRS20] Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti SaarimÃ€ki. On the diffuseness of technical debt items and accuracy of remediation time when using sonarqube. *Information and Software Technology*, 128:106377, 2020.

[BM23] L. Berciu and V. Moldovan. Software maintainability and refactorings prediction based on technical debt issues. *Studia Universitatis Babeș-Bolyai Informatica*, 68(2):22–40, 2023.

[BMM24] Liviu Marian Berciu, Simona Motogna, and Vasilica Moldovan. Software refactoring slr dataset resource, May 2024.

[BMM25] Liviu Marian Berciu, Vasilica Moldovan, and Simona Motogna. Dataset for "exploring the impact of commit burstiness on software quality", Jul 2025.

[BMM26a] Liviu Marian Berciu, Simona Motogna, and Arthur-Jozsef Molnar. Dataset for "a long-term exploratory study of source code quality issues in open-source python projects" article, Jan 2026.

[BMM26b] Liviu-Marian Berciu, Simona Claudia Motogna, and Arthur-Jozsef Molnar. A long-term exploratory study of source code quality issues in open-source python projects. *Software Quality Journal*, 34:9, 2026.

[BMMP25] Liviu Marian Berciu, Arthur-Jozsef Molnar, Simona Motogna, and Oskar Picus. Replication package for exploring the relation between source code commit information and sonarqube issues, May 2025.

[BMP24] Liviu Marian Berciu, Vasilica Moldovan, and Rares Patcas. The python software quality dataset, May 2024.

[BMPM25] Liviu-Marian Berciu, Simona Motogna, Oskar Picus, and Arthur-Jozsef Molnar. Exploring the relation between source code commit information and sonarqube issues. *Procedia Computer Science*, 270:841–850, 2025. 29th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2025).

[Buj24] M.A. Bujang. An elaboration on sample size determination for correlations based on effect sizes and confidence interval width: a guide for researchers. In *Restorative Dentistry and Endodontics*, volume 49, page e21, 2024. Published 2024 May 2.

[CBMP14] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. On the impact of refactoring operations on code quality metrics. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, page 456–460, USA, 2014. IEEE Computer Society.

[CBZK19] Angelos Chatzimparmpas, Stamatia Bibi, Ioannis Zozas, and Andreas Kerren. Analyzing the evolution of Javascript applications. In *ENASE*, pages 359–366, 2019.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[CCM+18] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. Understanding metric-based detectable smells in python software: A comparative study. *Information and Software Technology*, 94:14–29, 2018.

[CCMX16] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. Detecting code smells in python programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 18–23, 2016.

[CD10] Daniela S. Cruzes and Tore Dybå. Synthesizing evidence in software engineering research. In *Pro-*

ceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10. Association for Computing Machinery, 2010.

[CFI+24] Domenico Cotroneo, Alessio Foggia, Cristina Improta, Pietro Liguori, and Roberto Natella. Automating the correctness assessment of AI-generated code for security contexts. *Journal of Systems and Software*, 216:112–113, 2024.

[Che24] Zhifei Chen. Pysmell: Python code smells detection tool. `https://github.com/chenzhifei731/Pysmell/tree/master`, 2024.

[CHK+01] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Con20] Conventional Commits Specification v1.0.0. `https://www.conventionalcommits.org/en/v1.0.0/`, 2020.

[CSS13] K.K. Chaturvedi, V.B. Sing, and Prashast Singh. Tools in mining software repositories. In *2013 13th International Conference on Computational Science and Its Applications*, pages 89–98, 2013.

[DACA22] George Digkas, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The temporality of technical debt introduction on new code and confounding factors. *Software Quality Journal*, 30(2):283–305, jun 2022.

[Dat25] Datadog. What is static analysis & how does it work?, 2025.

[DLA+18] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. How do developers fix issues and pay back technical debt in the apache ecosystem? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 153–163, 2018.

[DNPT+18] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[dPB+23] Dario Amoroso d'Aragona, Fabiano Pecorelli, Maria Teresa Baldassarre, Davide Taibi, and Valentina Lenarduzzi. Technical debt diffuseness in the apache ecosystem: A differentiated replication. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 825–833, 2023.

[dTF+19] Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A. Furia, and Ziwei Huang. Evolution of statistical analysis in empirical se research: Current state and steps forward. *JSS*, 156:246–267, 2019.

[ESK18] Amir Elmishali, Roni Stern, and Meir Kalech. An artificial intelligence paradigm for troubleshooting software bugs. *Engineering Applications of Artificial Intelligence*, 69:147–156, 2018.

[FMNL21] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing github issues. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 155–159, 2021.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[fSC21] International Organization for Standardization and International Electrotechnical Commission. Information technology, software measurement, software quality measurement, automated source code quality measures. https://www.iso.org/standard/80623.html, 2021.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[git] Github api. `https://docs.github.com/en/rest/`.

[Git24] GitHub, Inc. *About Releases on GitHub*, 2024.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[HBS22] Tjasa Hericko, Saša Brdnik, and Bostjan Sumak. Commit classification into maintenance activities

using aggregated semantic word embeddings of software change messages. In *SQAMIA*, 2022.

[Hs23] Tjasa Hericko and Bostjan sumak. Commit classification into software maintenance activities: A systematic literature review. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1646–1651, 2023.

[JCP08] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer London, London, 2008.

[Jm12] Ronald Jabangwe and Darja mite. An exploratory study of software evolution and quality: Before, during and after a transfer. In *2012 IEEE Seventh International Conference on Global Software Engineering*, pages 41–50, 2012.

[JSM21] Joselito Mota Jr., Railana Santana, and Ivan Machado. Grumpy: an automated approach to simplify issue data analysis for newcomers. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 33–38, New York, NY, USA, 2021. Association for Computing Machinery.

[KC07] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[KCA21] Stratos Kourtzanidis, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. Reposkillminer: identifying software expertise from github repositories using natural language processing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1353–1357, New York, NY, USA, 2021. Association for Computing Machinery.

[KK15] Flavius Kehr and Tobias Kowatsch. Quantitative longitudinal research: A review of is literature, and a set of methodological guidelines. 09 2015.

[KRS13] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. The empirical commit frequency distribution of open source projects. In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13, page 1–8. ACM, August 2013.

[KS07] Georg von Krogh and Sebastian Spaeth. The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems*, 16(3):236–253, 2007.

[LAAZ23] Afshan Latif, Farooque Azam, Muhammad Waseem Anwar, and Amina Zafar. Comparison of leading language parsers–antlr, javacc, sablecc, tree-sitter, yacc, bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*, pages 7–13. IEEE, 2023.

[LAL15] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[LB24] R. Patcas L. Berciu, V. Moldovan. The python software quality dataset. `https://figshare.com/s/ca638816c59abf2195d9`, 2024.

[Lee16] Dong Kyu Lee. Alternatives to p value: confidence interval and effect size. *Korean journal of anesthesiology*, 69(6):555–562, 2016.

[Leh80] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LLCW+24] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. 33(5), 2024.

[LLHT20] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. Are sonarqube rules inducing bugs? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 501–511, 2020.

[LLTH19] Valentina Lenarduzzi, Francesco Lomio, Davide Taibi, and Heikki Huttunen. On the fault proneness of sonarqube technical debt violations: A comparison of eight machine learning techniques. *CoRR*, abs/1907.00376, 2019.

[LMBM] Vasilica Andreea Moldovan Liviu Marian Berciu and Simona Motogna. Exploring the impact of commit burstiness on software quality. In *QUATIC 2025*.

[LML22] Francesco Lomio, Sergio Moreschini, and Valentina Lenarduzzi. A machine and deep learning analysis among sonarqube rules, product, and process metrics for fault prediction. *Empirical Software*

Engineering, 27(7):189, 2022.

[LPS+23] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, 198, 2023.

[LS08] G. A Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 39–44, 2008.

[LST19a] Valentina Lenarduzzi, Nyyti Saarimaki, and Davide Taibi. On the diffuseness of code technical debt in java projects of the apache ecosystem. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 98–107, 2019.

[LST19b] Valentina Lenarduzzi, Nyyti Saarimäki, and Davide Taibi. The technical debt dataset. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'19. ACM, September 2019.

[Mar04] Radu Marinescu. An empirical study of the relationship between code smells and refactoring. *Empirical Software Engineering*, 9(4):429–462, 2004.

[MBM24] Simona Motogna, Liviu-Marian Berciu, and Vasilica-Andreea Moldovan. Artificial intelligence methods in software refactoring: A systematic literature review. In *50th Euromicro Conference Series on Software Engineering and Advanced Applications*, 2024.

[MBP24] Vasilica-Andreea Moldovan, Liviu-Marian Berciu, and Rares-Danut Patcas. The python software quality dataset. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 395–398, 2024.

[McH13] Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

[MM20a] Arthur-Jozsef Molnar and Simona Motogna. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[MM20b] Arthur-Jozsef Molnar. and Simona Motogna. Longitudinal evaluation of open-source software maintainability. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 120–131. INSTICC, SciTePress, 2020.

[MM21] Arthur-Jozsef Molnar and Simona Motogna. A study of maintainability in evolving open-source software. In Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 261–282, Cham, 2021. Springer International Publishing.

[MM22] Arthur-Jozsef Molnar. and Simona Motogna. Characterizing technical debt in evolving open-source software. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 174–185. INSTICC, SciTePress, 2022.

[MMD11] Laurie McLeod, Stephen MacDonell, and Bill Doolin. Qualitative research on software development: A longitudinal case study methodology. *Empirical Software Engineering*, 16:430–459, 08 2011.

[Mol20] Arthur-Jozsef Molnar. Collection of technical debt issues in freemind, jedit and tuxguitar open source software. 7 2020.

[MP12] Vishal Midha and Prashant Palvia. Factors affecting the success of open source software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[MV00] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[MVS23] Simona Motogna, Andreea Vescan, and Camelia Serban. Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. *J. Syst. Softw.*, 201, 2023.

[NCK+19] M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, and F. Chew. A longitudinal study of identifying and paying down architecture debt. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 171–180, 2019.

[NGM+19] Malanga Ndenga, Ivaylo Ganchev, Jean Méhat, Franklin Wabwoba, and Herman Akdag. Performance and cost-effectiveness of change burst metrics in predicting software faults. *KIS*, 60, 07 2019.

[NMB10] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. Change bursts as defect predictors.

In *ISSRE*, pages 309–318, 2010.

[ODA+15] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, and R. Tonelli. The jira repository dataset: Understanding social aspects of software development. In *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*, pages 1–4, 2015.

[OH92] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, Nov 1992.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pau21] Paul Ralph (ed.). Acm sigsoft empirical standards for software engineering research, version 0.2.0, 2021.

[PFMM08] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, page 68–77. BCS Learning & Development Ltd., 2008.

[PMB23] Manuela Petrescu, Simona Motogna, and Liviu Berciu. Women in scrum master role: Challenges and opportunities*. pages 49–55, 05 2023.

[Pos23] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. `https://www.postgresql.org/`, 2023.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[PyL23] PyLint Contributors. *PyLint*. PyLint, 2023.

[Ral21] Paul Ralph. Acm sigsoft empirical standards released. *SIGSOFT Softw. Eng. Notes*, 46(1):19, February 2021.

[RB22] Paul Ralph and Sebastian Baltes. Paving the way for mature secondary research: the seven types of literature review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1632–1636. Association for Computing Machinery, 2022.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[RDPM] Liviu Marian Berciu Rares Danut Patcas, Oskar Pikus and Simona Motogna. Ast similarity of code quality issues in python and javascript. submitted to EASE2026.

[RG19] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[RPS+23] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. A comprehensive evaluation of szz variants through a developer-informed oracle. *Journal of Systems and Software*, 202:111729, 2023.

[RTLR18] M. M. Rosli, E. Tempero, and A. Luxton-Reilly. Evaluating the quality of datasets in software engineering. *Advanced Science Letters*, 24(10):7232–7239, 2018.

[Rub24] Micheal Rubik. Radon: Measure cyclomatic complexity, maintainability index, and halstead metrics in python code. `https://radon.readthedocs.io/`, 2024.

[SAB18] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery.

[SDC24] Diego S Sarafim, Karina V Delgado, and Daniel Cordeiro. Detecting code smells in JavaScript: An annotated dataset for software quality analysis. In *Simpósio Brasileiro de Engenharia de Software*

(SBES), pages 313–322. SBC, 2024.

[SGK+09] Diomidis Spinellis, Georgios Gousios, Vassilios Karakoidas, Panagiotis Louridas, Paul J. Adams, Ioannis Samoladas, and Ioannis Stamelos. Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, 233:5–28, 2009.

[SK21] T. Sharma and M. Kessentini. Qscored: A large dataset of code smells and quality metrics. In *2021 IEEE/ACM 18th MSR*, pages 590–594, 2021.

[SMKA17] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, 2017.

[Son24a] SonarQube. Sonarqube documentation, 2024.

[Son24b] SonarQube. What is clean code, 2024.

[Son25a] SonarSource. How new issues are identified, 2025. SonarQube Server 10.3 User Guide.

[Son25b] SonarSource. Managing issues, 2025.

[Sonnd] SonarSource. Cyclomatic complexity: developer's guide, n.d.

[Sri17] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[SSM05] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[ST24] Inc. Scientific Toolworks. Understand: Static analysis and code visualization tool. `https://www.scitools.com/`, 2024.

[SW65] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

[TAA+18] David Trafimow, Valentin Amrhein, Corson N Areshenkoff, Carlos J Barrera-Causil, Eric J Beh, Yusuf K Bilgiç, Roser Bono, Michael T Bradley, William M Briggs, Héctor A Cepeda-Freyre, et al. Manipulating the alpha level cannot cure significance testing. *Front. Psychol.*, 9:699, May 2018.

[TDPT+21] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. How software refactoring impacts execution time. *ACM Trans. Softw. Eng. Methodol.*, 31(2), 2021.

[TEHG23] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, 28(2):30, 2023.

[TFA20] Jie Tan, Daniel Feitosa, and Paris Avgeriou. An empirical study on self-fixed technical debt. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt '20, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[TFA23] Jie Tan, Daniel Feitosa, and Paris Avgeriou. The lifecycle of technical debt that manifests in both source code and issue trackers. *Information and Software Technology*, 159:107216, 2023.

[TFAL21] Jie Tan, Daniel Feitosa, Paris Avgeriou, and Mircea Lungu. Evolution of technical debt remediation in python: A case study on the apache software ecosystem. *Journal of Software: Evolution and Process*, 33(4):e2319, 2021. e2319 smr.2319.

[Tho21] Patrick Thomson. Static analysis: An introduction. *ACM Queue*, 19(4):29–41, 2021.

[TPB+17] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Softw. Eng.*, 43(11):1063–1088, nov 2017.

[Uni] University of Waikato. Weka - data mining software in java. `https://sourceforge.net/projects/weka/`.

[VAMM] Liviu Marian Berciu Vasilica Andreea Moldovan and Simona Motogna. A comparative evaluation of intelligent methods for code smell detection in python. submitted to Scientific Annals of Computer Science.

[VSM13] B. Vasilescu, A. Serebrenik, and T. Mens. A historical dataset of software engineering conferences. In *2013 10th MSR*, pages 373–376. IEEE, 2013.

[VSPL18] Roberto Verdecchia, Ren\'e Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. In Birgit Penzenstadler, Steve Easterbrook, Colin Venters, and Syed Ishtiaque Ahmed, editors, *ICT4S2018. 5th International Conference on Information and Communication Technology for Sustainability*, volume 52 of *EPiC Series in Computing*, pages 365–383. EasyChair, 2018.

[Wan18] Divanshi Priyadarshni Wangoo. Artificial intelligence techniques in software engineering for automated software reuse and design. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*, pages 1–4, 2018.

[WDS16] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 493–501, 2016.

[WM18] Neil Walkinshaw and Leandro Minku. Are 20% of files responsible for 80% of defects? In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, pages 1–10. Association for Computing Machinery, 2018.

[WMMR06] Roel Wieringa, Neil Maiden, Nancy Mead, and Colette Rolland. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. *Requir. Eng.*, 11:102–107, 03 2006.

[WNLB22] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. *ESE*, 27(1), January 2022.

[XF14] Qi Xuan and Vladimir Filkov. Building it together: synchronous development in oss. In *Proceedings 36th ICSE*, page 222–233, 2014.

[ZZQL25] Qunhong Zeng, Yuxia Zhang, Zhiqing Qiu, and Hui Liu. A First Look at Conventional Commits Classification . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 127–139. IEEE Computer Society, 2025.