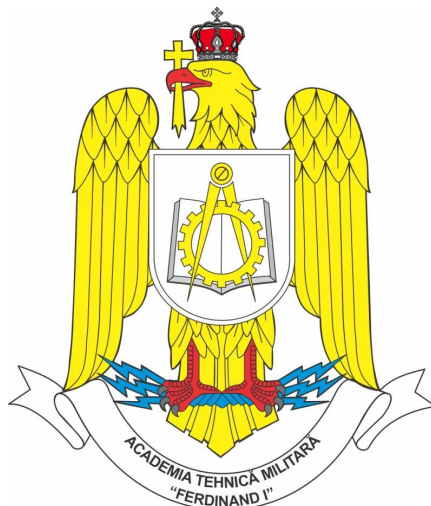


NECLASIFICAT

ROMÂNIA
MINISTERUL APĂRĂRII NAȚIONALE
ACADEMIA TEHNICĂ MILITARĂ
„FERDINAND I”

FACULTATEA DE SISTEME INFORMATICE ȘI SECURITATE CIBERNETICĂ
Specializarea: Calculatoare și sisteme informatice pentru apărare
și securitate națională



PLATFORMA DE VIZUALIZARE EVOLUȚIE COD

CONDUCĂTOR ȘTIINȚIFIC:

Ș.L. dr. ing. Cristian CHILIPIREA

ABSOLVENT:

Stud. sg. maj. Alexandru ALEXANDRU

Conține _____ file

Inventariat sub nr. _____

Poziția din indicator: _____

Termen de păstrare: _____

BUCUREȘTI

2022

NECLASIFICAT

1 din 69

ABSTRACT

This paper presents a web platform solution for code evolution visualization. The developed platform brings in addition a suite of functionalities, in addition to the one mentioned earlier, which will offer different aspects related to the evolution of the code. First of all, it helps the user to improve the implemented code, as follows: the user has the possibility to automatically identify unusual events in the evolution of a code, can detect if a group of students collaborated to perform a task, check the security of the code, receive certain explanations based on the code written by him (its complexity, some solutions in case of bugs) and to extract certain features that provide information about the contribution of students. It also benefits from a system that evaluates a repository based on contributions.

REZUMAT

Această lucrare prezintă o soluție pentru vizualizarea evoluției codului a unui produs software. Platforma realizată aduce în plus o suită de funcționalități, pe lângă cea menționată mai devreme care să ofere diferite aspecte legate de evoluția codului. În primul rând ajută utilizatorul în vederea îmbunătățirii codului implementat, astfel: utilizatorul are posibilitatea să identifice automat evenimente neobișnuite în evoluția unui cod, poate să detecteze dacă un grup de studenți a colaborat în vederea realizării unui task, să verifice securitatea codului, să primească anumite explicații pe baza codului scris de el (complexitatea acestuia, ce realizează acea funcție scrisă de el, să primească niste soluții în cazul unor bug-uri) și să extragă anumite caracteristici care oferă informații despre contribuția studenților. Totodată acesta beneficiază și de un sistem care evaluează un repository pe baza contribuțiilor aduse.

Capitolul 1 prezintă scopul lucrării și importanța acesteia, urmând ca în capitolul următor să se prezinte soluții actuale care abordează acest subiect. Urmează o scurtă descriere a modalităților de control a versiunii, a ceea ce înseamnă Git și ce poate face acesta, a modelului GPT-3 și OpenAI și a conceptului de Sandbox. În capitolele 4, 5 și 6 prezintă arhitectura sistemului, modul de implementare a funcționalităților și ce biblioteci și pachete s-au folosit pentru a realiza acest lucru. Capitolul 7 prezintă testele care s-au efectuat pentru a fi sigur că rezultatele pe care le oferă funcționalitățile implementate sunt corecte. Capitolul 8 prezintă interacțiunea dintre utilizator și aplicație și diagramele UML, urmând ca în capitolul 9 să se traseze posibilități de îmbunătățire ale proiectului și concluziile trase în urma implementării acestuia.

CUPRINS

| | |
|--|----|
| ABSTRACT | 2 |
| REZUMAT | 3 |
| LISTĂ DE ABREVIERI..... | 7 |
| LISTĂ DE FIGURI..... | 8 |
| LISTĂ DE TABELE..... | 9 |
| 1. INTRODUCERE | 10 |
| 1.1. Prezentare generală | 10 |
| 1.2. Importanța temei | 10 |
| 1.3. Utilitatea sistemului..... | 10 |
| 2. STADIU ACTUAL | 11 |
| 2.1. Soluții similare..... | 11 |
| 2.1.1. CVSscan: Visualization of Code Evolution [4]..... | 12 |
| 2.2. Articole de cercetare..... | 15 |
| 2.2.1. Understanding Source Code Evolution Using Abstract Syntax Tree Matching [5] | 15 |
| 3. CONCEPTE TEORETICE..... | 16 |
| 3.1. Controlul versiunii [6]..... | 16 |
| 3.1.1. Clasificare | 16 |
| 3.1.2. Caracteristici | 19 |
| 3.2. Git [6] | 20 |
| 3.2.1. Caracteristici | 20 |
| 3.2.2. Structuri de date | 22 |
| 3.3. OpenAI | 23 |
| 3.3.1. Tokens | 23 |
| 3.3.2. Modele | 24 |
| 3.4. GPT-3 [7] | 25 |
| 3.5. Sandbox [8] | 25 |
| 3.6. Regresie liniară [9] | 26 |
| 4. STRUCTURA PROIECTULUI | 27 |

NECLASIFICAT

| | | |
|--------|---|----|
| 4.1. | API-uri și biblioteci folosite..... | 27 |
| 4.1.1. | Nodegit..... | 27 |
| 4.1.2. | Valgrind [10]..... | 28 |
| 4.1.3. | OpenAI API | 29 |
| 4.1.4. | Scikit-learn | 30 |
| 4.1.5. | React..... | 30 |
| 4.1.6. | NodeJs | 32 |
| 4.2. | Cerințe proiect | 34 |
| 4.3. | Definirea arhitecturii proiectului..... | 34 |
| 5. | IMPLEMENTARE | 36 |
| 5.1. | Afișarea evoluției codului unui repository | 38 |
| 5.2. | Identificarea introducerii unor bucăți mari de cod..... | 39 |
| 5.3. | Verificarea leak-urile de memorie | 40 |
| 5.4. | Crearea de sugestii în scop educativ pe baza codului scris..... | 40 |
| 5.5. | Detecția similarităților între fișiere | 42 |
| 5.6. | Compararea a două repository-uri..... | 42 |
| 5.7. | Executarea codului | 43 |
| 5.8. | Gestionarea unui repository privat..... | 44 |
| 5.9. | Realizarea opțiunilor pentru modificarea codului..... | 44 |
| 5.9.1. | Opțiunea de commit..... | 44 |
| 5.9.2. | Opțiunea de push..... | 46 |
| 5.10. | Extragerea caracteristicilor unui set de repository-uri | 46 |
| 5.11. | Creare script pentru auto-commit | 46 |
| 5.12. | Creare script pentru ștergerea fișierelor temporare..... | 47 |
| 5.13. | Realizarea unui sistem de evaluare | 48 |
| 6. | REZULTATE TESTARE | 50 |
| 7. | INTERACȚIUNEA UTILIZATOR-APLICAȚIE..... | 51 |
| 7.1. | Diagrame UML | 52 |
| 8. | CONCLUZII ȘI DIRECȚII VIITOARE DE CERCETARE | 56 |
| 8.1. | Probleme întâmpinate în implementare | 56 |

NECLASIFICAT

NECLASIFICAT

| | | |
|--------|------------------------------------|----|
| 8.2. | Rezultate obținute..... | 56 |
| 8.3. | Îmbunătățiri ale proiectului | 57 |
| 8.4. | Funcționalități suplimentare..... | 57 |
| 9. | BIBLIOGRAFIE | 58 |
| 10. | ANEXE..... | 59 |
| 10.1. | Anexa A..... | 59 |
| 10.2. | Anexa B..... | 60 |
| 10.3. | Anexa C..... | 61 |
| 10.4. | Anexa D..... | 62 |
| 10.5. | Anexa E | 63 |
| 10.6. | Anexa F | 64 |
| 10.7. | Anexa G..... | 65 |
| 10.8. | Anexa H..... | 66 |
| 10.9. | Anexa I | 67 |
| 10.10. | Anexa J..... | 68 |
| 10.11. | Anexa K | 69 |

NECLASIFICAT

LISTĂ DE ABREVIERI

| | | |
|-----|--------|---|
| 1. | CVS | Concurrent Versions System |
| 2. | VCM | Version Control Management |
| 3. | AST | Abstract Syntax Tree |
| 4. | CVCS | Centralized Version Control Systems |
| 5. | DVCS | Distributed Version Control Systems |
| 6. | HTTP | Hypertext Transfer Protocol |
| 7. | FTP | Fyle Transfer Protocol |
| 8. | SSH | Secure Shell |
| 9. | IDE | Integrated Development Environment |
| 10. | API | Application Programming Interface |
| 11. | GPT-3 | Generative Pre-trained Transformer 3 |
| 12. | JIT | Just Intime Compiler |
| 13. | DPI | Dynamic Binary Instrumentation |
| 14. | DPA | Dynamic Binary Analysis |
| 15. | GCC | GNU Compiler Collections |
| 16. | LTS | Long Term Support |
| 17. | UML | Unified Modeling Language |
| 18. | JSX | JavaScript Syntax Extension |
| 19. | CPU | Central Processing Unit |
| 20. | DOM | Document Object Model |
| 21. | DBSCAN | Density-based spatial clustering of applications with noise |
| 22. | SRC | Source |

LISTĂ DE FIGURI

| | |
|--|---------------|
| <i>Fig. 2.1-1 Aspectul liniilor în CVSscan: a) Bazat pe fișiere b) Bazat pe linii.....</i> | <i>13</i> |
| <i>Fig. 2.1-2 Prezentare generală a instrumentului CVSscan.....</i> | <i>14</i> |
| <i>Fig. 2.1-3 Reprezentarea a 2 versiuni succesive pentru un program.....</i> | <i>15</i> |
| <i>Fig. 3-1 Reprezentare sistem de control al versiunilor locale</i> | <i>17</i> |
| <i>Fig. 3-2 Reprezentare sistem de control al versiunilor centralizate</i> | <i>18</i> |
| <i>Fig. 3-3 Reprezentare sistem de control al versiunilor distribuite.....</i> | <i>19</i> |
| <i>Fig. 4-1 Reprezentarea pașilor pe care instrumentul valgrind le urmează</i> | Error! |
| Bookmark not defined. | |
| <i>Fig. 5-1 Arhitectura sistemului</i> | <i>35</i> |
| <i>Fig. 7-1 Rezultatul platformei.....</i> | <i>50</i> |
| <i>Fig. 7-2 Rezultatul de pe Github.....</i> | <i>50</i> |
| <i>Fig. 8-1 Pagina de start a platformei</i> | <i>51</i> |
| <i>Fig. 8-2 Pagina pentru vizualizarea evoluției codului</i> | <i>52</i> |
| <i>Fig. 8-3 Diagrama de activități pentru vizualizare evoluție cod</i> | <i>53</i> |
| <i>Fig. 8-4 Diagrama de activități verificare leak-uri de memorie</i> | <i>54</i> |
| <i>Fig. 8-5 Diagrama de activități pentru rularea codului.....</i> | <i>54</i> |
| <i>Fig. 8-6 Diagrama cazurilor de utilizare</i> | <i>55</i> |

LISTĂ DE TABELE

| | |
|---|-----------|
| <i>Tabel 1 Structura unei cereri HTTP pentru OpenAI API.....</i> | <i>41</i> |
| <i>Tabel 2 Parametrii necesari pentru descărcarea unui repository privat.....</i> | <i>44</i> |

1. INTRODUCERE

Proiectul își propune implementarea unei platforme web în scop educativ care să atingă următoarele puncte:

1. Să permită utilizatorului să descarce unui repository de pe GitHub și să vizualizeze evoluția codului.
2. Identificarea unor comportamente ciudate în evoluția acestuia (introducerea unor bucăți mari de cod).
3. Realizarea unui sistem care să permită utilizatorului să aducă modificări codului.
4. Detectia unor grupuri de studenți care au colaborat în vederea realizării unui task (teme individuale, exerciții etc.).
5. Verificarea securității codului.
6. Posibilitatea utilizatorului de a rula codul, de a obține anumite hint-uri în rezolvarea unor bug-uri, de a primi explicații asupra unor funcții, programe scrise de acesta (de exemplu complexitatea unui algoritm de sortare).

1.1. Prezentare generală

Proiectul se compune din două direcții. Prima oferă o privire în ansamblu a evoluției codului unui repository, cum se transformă produsul software de la commit la commit, oferind posibilitatea utilizatorului să realizeze anumite operații asupra repository-ului la care lucrează, să primească sugestii în scop educativ pe baza codului scris de el și să verifice leak-urile de memorie.

A doua direcție se concentrează pe detectarea de anomalii în dezvoltarea produsului software, pe extragerea anumitor caracteristici care să ofere cât mai multe informații care pot să sugereze colaborarea unor studenți în rezolvarea unei teme sau dacă un student a lucrat în mod constant în realizarea ei.

1.2. Importanța temei

Vizualizarea evoluției codului unui produs software este foarte importantă în zilele noastre deoarece oferă o mai bună înțelegere a modului în care codul trece dintr-o stare în alta, dacă modificările aduse sunt substanțiale sau nu și oferă informații care să determine modul de a scrie codul.

1.3. Utilitatea sistemului

Soluția propusă este utilă deoarece analizează produsul software la nivel de cod sursă oferind o mai bună înțelegere a modului în care evoluează acesta și

oferind opțiuni utilizatorului de interacționa și lucra cu acesta în fiecare moment al evoluției.

Totodată sugestiile pe care acesta le poate primi pe baza codului scris îi formează utilizatorului o idee despre cum să-ți îmbunătățească stilul de a scrie cod și cum să înțeleagă funcțiile mai complicate.

De asemenea soluția pentru extragerea anumitor caracteristici ale repository-urilor poate determina o analiză completă asupra modului în care persoanele participante la proiect au depus o muncă constantă sau nu.

2. STADIU ACTUAL

2.1. Soluții similare

Un număr de sisteme pentru identificarea diferențelor între programe au fost dezvoltate de-a lungul timpului. Câteva din aceste sisteme sunt:

Yang [1] este un sistem de identificare a modificărilor sintactice “relevante” între două versiuni ale unui program, filtrând cele mai irelevante diferențe produse de diff. Această soluție se potrivește cu arbori de analiză (AST) și poate chiar potrivi arbori structurali diferiți folosind euristică. Instrumentul Yang nu se poate ocupa de redenumirea variabilelor sau modificări de tip și, în general, se concentrează mai mult pe găsirea unei similarități sintactice maxime între doi arbori de analiză.

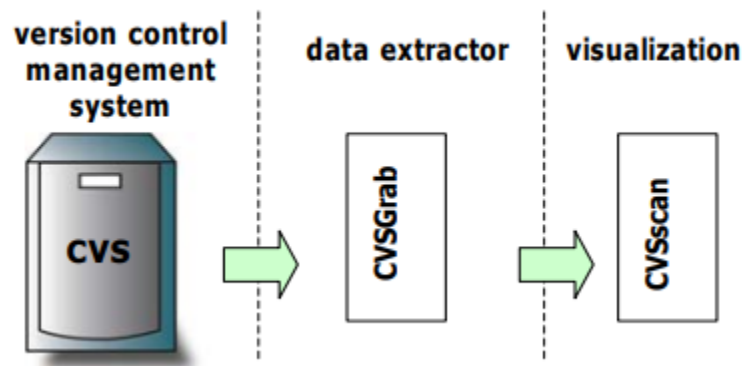
Horwitz [2] este o soluție care se bazează pe un sistem care se concentrează pe găsirea semantică decât pe cea sintactică, pentru a găsi schimbările în programe. Două programe sunt identice din punct de vedere semantic dacă succesiunea valorilor observabile produse este aceeași, chiar dacă sunt diferite din punct de vedere textual. De exemplu, cu această abordare semantică-conservatoare transformările, cum ar fi dezvoltarea codului sau reordonarea instrucțiunilor, nu ar fi semnalate ca o modificare. Algoritmul lui Horwitz rulează pe un subset limitat din C care nu include funcții, pointeri sau matrici.

Jackson și Ladd [3] propun un instrument de diferențiere care analizează două versiuni ale unei proceduri pentru a identifica schimbările în dependențe între variabilele formale, locale și globale. Abordarea lor este insensibilă la numele variabilelor locale, dar sistemul lor nu efectuează nicio analiză globală nu ia în considerare schimbările de tip și sacrifică soliditatea de dragul suprimării diferențelor false.

2.1.1. CVSScan: Visualization of Code Evolution [4]

CVSScan este o soluție propusă și implementată de 3 ingineri de la universitatea din Eindhoven. În timpul ciclului de viață al unui sistem software, codul sursă este schimbat de multe ori. Pe scurt este un afișaj orientat pe linie a codului în schimbare, unde se află fiecare versiune reprezentată printr-o coloană, și unde direcția orizontală este utilizată pentru timp, afișajele separate conectate afișează diferite valori, cum ar fi și codul sursă în sine. O mare varietate de opțiuni este furnizată pentru a vizualiza o serie de aspecte diferite.

Datele provin din gestionarea controlului versiunilor CVS sistem (VCM). Pentru a decupla CVS de vizualizarea în sine, extragerea datelor se face printr-un instrument separat: CVSgrab. Astfel se poate folosi instrumentul de vizualizare cu orice VCM, o dată a este implementat un extractor de date adecvat. Elementul central al a Sistemul VCM este un depozit care stochează toate versiunile unui anumit fișier.



Abordarea acestora se bazează pe calcularea lungimii liniilor de cod pentru a oferi o umplere eficientă din punct de vedere al spațiului pentru a afișa fișierele și structura acestora. Acest lucru permite să se vizualizeze mai mult cod sursă pe același ecran. În plus, această abordare se concentrează pe un singur fișier la un moment dat, pentru a oferi o imagine cuprinzătoare asupra evoluției sale, permițând utilizatorilor să facă corelații între modificările apărute în timp.

Pentru aranjarea verticală a liniilor într-o bandă de versiune, aceștia au propus două abordări. Primul, fiind bazat pe fișiere, folosește pe coordonata y poziția liniei locale. Acest aspect oferă un o vedere intuitivă despre organizarea fișierelor și evoluția dimensiunii. A doua abordare, este bazat pe linii, folosește pe coordonate y poziția liniei globale. Această abordare păstrează ordinea rândurilor aceleiași versiuni, introduce spații goale în care liniile au fost șterse anterior sau vor fi introdus într-o versiune viitoare. În acest aspect, fiecare linie globală l are a

poziție Y fixă pe toată durata vizualizării. Asta permite identificarea ușoară a blocurilor de cod care rămân constante în timp sau care sunt adăugate sau șterse.

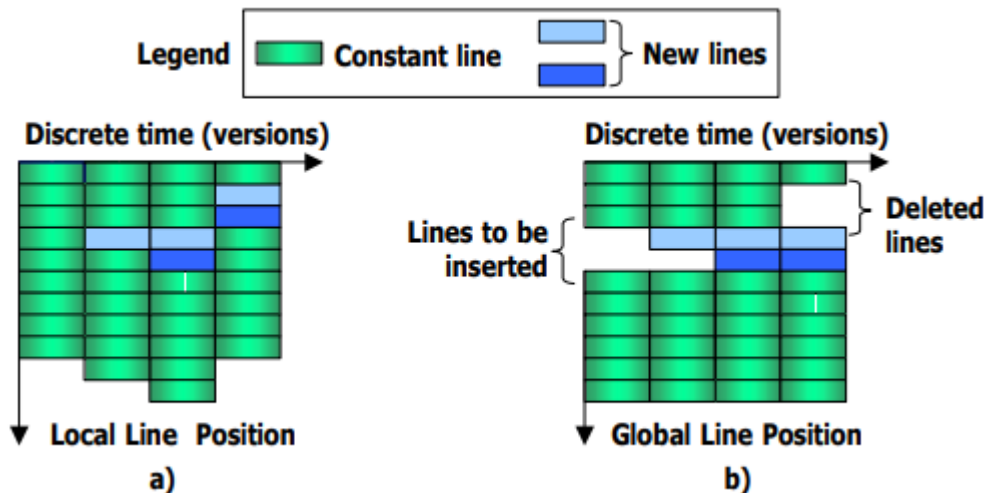


Fig. 2.1-1 Aspectul liniilor în CVSscan: a) Bazat pe fișiere b) Bazat pe linii

Un factor cheie în înțelegerea tiparelor apărute în vizualizarea evoluției codului este de a le corela cu alte informații despre program. Pe lângă vizualizarea pe linie a evoluției codului, CVSscan oferă două vizualizări suplimentare de metrice și o nouă vizualizare a textului asupra fragmentelor de cod selectate.

CVSscan permite utilizatorului să coreleze informații despre evoluția software cu detalii specifice codului sursă și informații statistice generale. Prin intermediul vizualizărilor metrice, utilizatorii pot obține vizual informații statistice despre linii (de exemplu, durata de viață a unui linie la o poziție globală dată) sau versiuni (de exemplu, autorul unei versiuni sau dimensiune). Vizualizarea codului pe două niveluri oferă detalii la cerere despre un fragment de cod: corpul textului, autorii liniilor și evoluția textului. Utilizatorul poate selecta fragmentul de interes prin simpla periere a zonei de evoluție a fișierului.

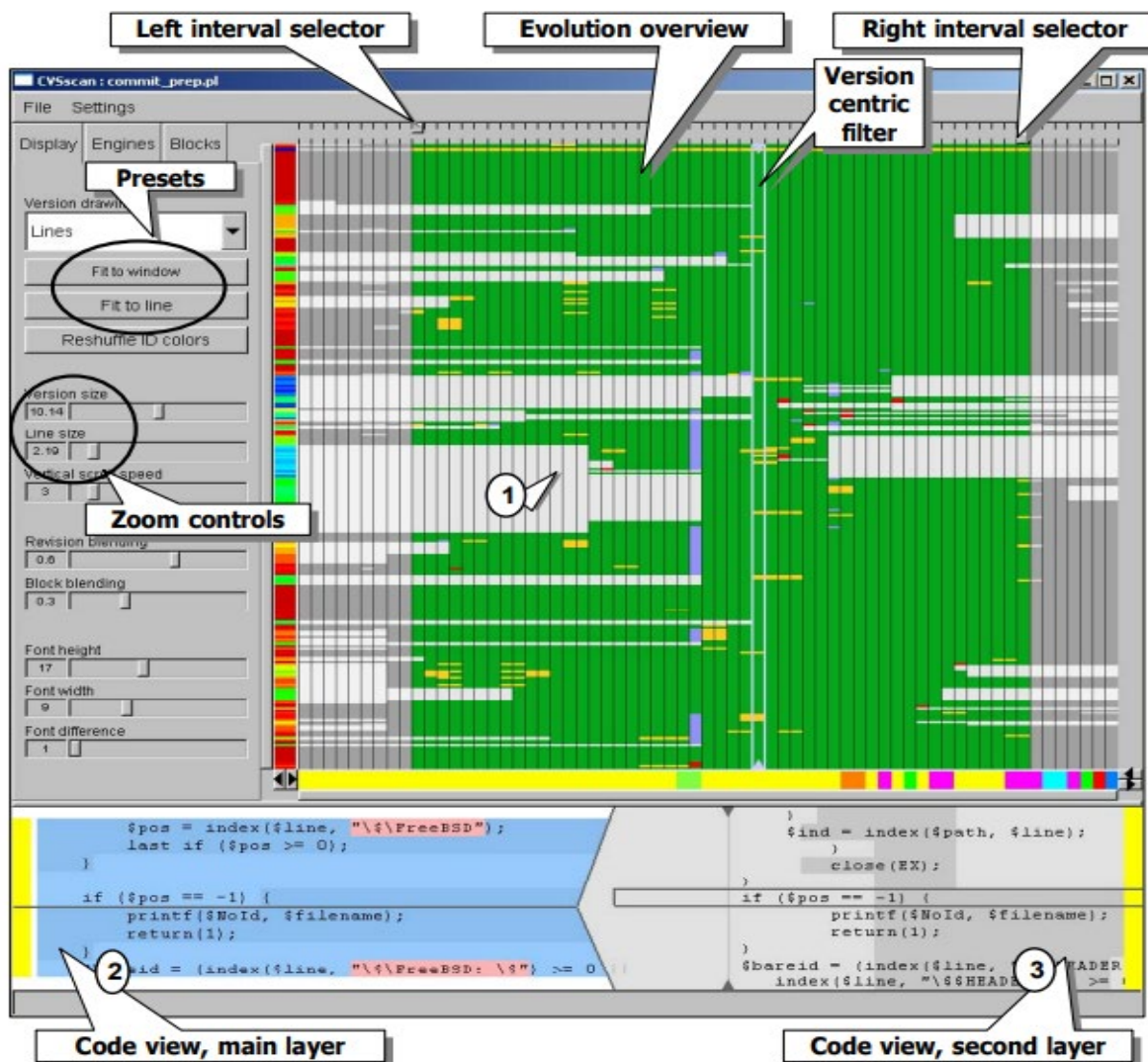


Fig. 2.1-2 Prezentare generală a instrumentului CVSscan

Pentru a înțelege mai bine ce reprezintă figura (Fig. 2.1-2) versiunea fișierului și numărul rândului de sub mouse (1) sunt afișate în detaliu în vizualizările text (2,3).

2.2. Articole de cercetare

În secțiunea următoare se va prezenta o scurtă sinteză a unui articol aflat în legătură cu tema proiectului care motivează necesitatea soluției proiectului.

2.2.1. Understanding Source Code Evolution Using Abstract Syntax Tree Matching [5]

Lucrarea explică importanța exploatării repository-urilor software la nivel de cod sursă deoarece pot oferi o mai bună înțelegere a modului în care evoluează software-ul. Aceștia prezintă un instrument pentru compararea rapidă a codului sursă al diferitelor versiuni ale unui program C. Abordarea se bazează pe potrivirea parțială a arborelui de sintaxă abstractă și poate urmări modificări simple ale variabilelor, tipurilor și funcțiilor globale.

Aceste modificări pot caracteriza aspecte ale evoluției software-ului utile pentru a răspunde la întrebări de nivel superior. În special, lucrarea ia în considerare modul în care ar putea fi utilizate pentru a informa proiectarea sistemului într-un mod cât mai dinamic. Rezultatele acestei lucrări se bazează pe măsurători ale diferitelor versiuni open source BIND, OpenSSH, Apache, Vsftpd și nucleul Linux.

În capitolul 2 autorii explică cum au început analizarea celor două versiuni de program pentru a produce arbori de sintaxă abstractă (AST), pe care le parcurg în paralel pentru a realiza un map de tip și nume.

| | |
|---|---|
| <pre>typedef int sz_t; int count; struct foo { int i; float f; char c; }; int baz(int a, int b) { struct foo sf; sz_t c = 2; sf.i = a + b + c; count++; }</pre> | <pre>int counter; typedef int size_t; struct bar { int i; float f; char c; }; int baz(int d, int e) { struct bar sb; size_t g = 2; sb.i = d + e + g; counter++; } void biff(void) { }</pre> |
|---|---|

Fig. 2.1-3 Reprezentarea a 2 versiuni succesive pentru un program

Aceștia oferă ca exemplu figura de mai sus unde exemplul din stânga este versiunea inițială, iar instrumentul lor descoperă că funcția **baz** este neschimbată, pentru că deși fiecare linie a fost modificată sintactic, funcția în fapt este structural aceeași și produce aceeași ieșire. Aceștia determină, de asemenea, că tipul **sz_t** a fost redenumit **size_t**, **count** a fost redenumit **counter**, structura **foo** a fost

redenumită **bar**, iar funcția **biff()** a fost adăugată. Se observă că dacă ar fi fost aplicată o diferență orientată pe linii, aproape toate liniile din program ar fi fost marcate ca fiind schimbate și ar fi oferit foarte puține informații utile.

Pentru a putea oferi rezultate, instrumentul trebuie să găsească o bijecție între numele vechi și numele noul din program, chiar dacă funcțiile și declarațiile de tip au fost reordonate și modificate. Pentru a face acest lucru, instrumentul începe prin a găsi numele de funcții care sunt comune între versiunile de program (aceștia pleacă de la presupunerea că numele funcțiilor nu se schimbă foarte des).

În capitolul 3 explică modul în care aceștia au folosit instrumentul pentru a caracteriza modificarea software-ului și pentru a ghida proiectarea acestuia într-un mod dinamic.

Autorii lucrării au constatat că instrumentul lor este eficient și oferă câteva perspective asupra evoluției software-ului. Au început să extindă instrumentul dincolo de potrivirea AST, pentru a măsura metrice de evoluție precum cuplarea comună sau coeziunea.

3. CONCEPTE TEORETICE

3.1. Controlul versiunii [6]

Sistemele pentru controlul versiunii sunt aplicații ce înregistrează schimbări ce au loc asupra unui fișier sau grup de fișiere de-a lungul timpului, astfel încât să ne putem aminti anumite versiuni mai târziu. Deși în general se folosește pentru fișiere ce conțin cod sursă, ele pot fi folosite pentru orice tip de fișiere.

Aceste sisteme permit readucerea unor fișiere (sursă, text, imagini, etc.) la o versiune anterioară, compararea schimbărilor ce au avut loc de-a lungul timpului, și multe altele. Astfel, dacă se face vreo greșală, sau se pierde fișiere/modificări, acestea se pot recupera rapid.

3.1.1. Clasificare

3.1.1.1. VCS

Metoda de control al versiunii aleasă de mulți oameni este să copieze fișierele într-un alt director (poate un director marcat cu ora, dacă sunt deștepți). Această abordare este foarte comună, deoarece este atât de simplă, dar este, de asemenea, incredibil de predispusă la erori. Este ușor să uitați în ce director vă aflați și să scrieți accidental în fișierul greșit sau să copiați fișierele pe care nu intenționați să le faceți. Pentru a face față acestei probleme, programatorii au

dezvoltat cu mult timp în urmă VCS-uri locale care aveau o bază de date simplă care păstra toate modificările fișierelor sub controlul revizuirii.

Unul dintre cele mai populare instrumente VCS a fost un sistem numit RCS, care este distribuit și astăzi cu multe computere. RCS funcționează păstrând seturi de patch-uri (adică diferențele dintre fișiere) într-un format special pe disc; apoi poate re-crea cum arăta orice fișier în orice moment prin adunarea tuturor patch-urilor.

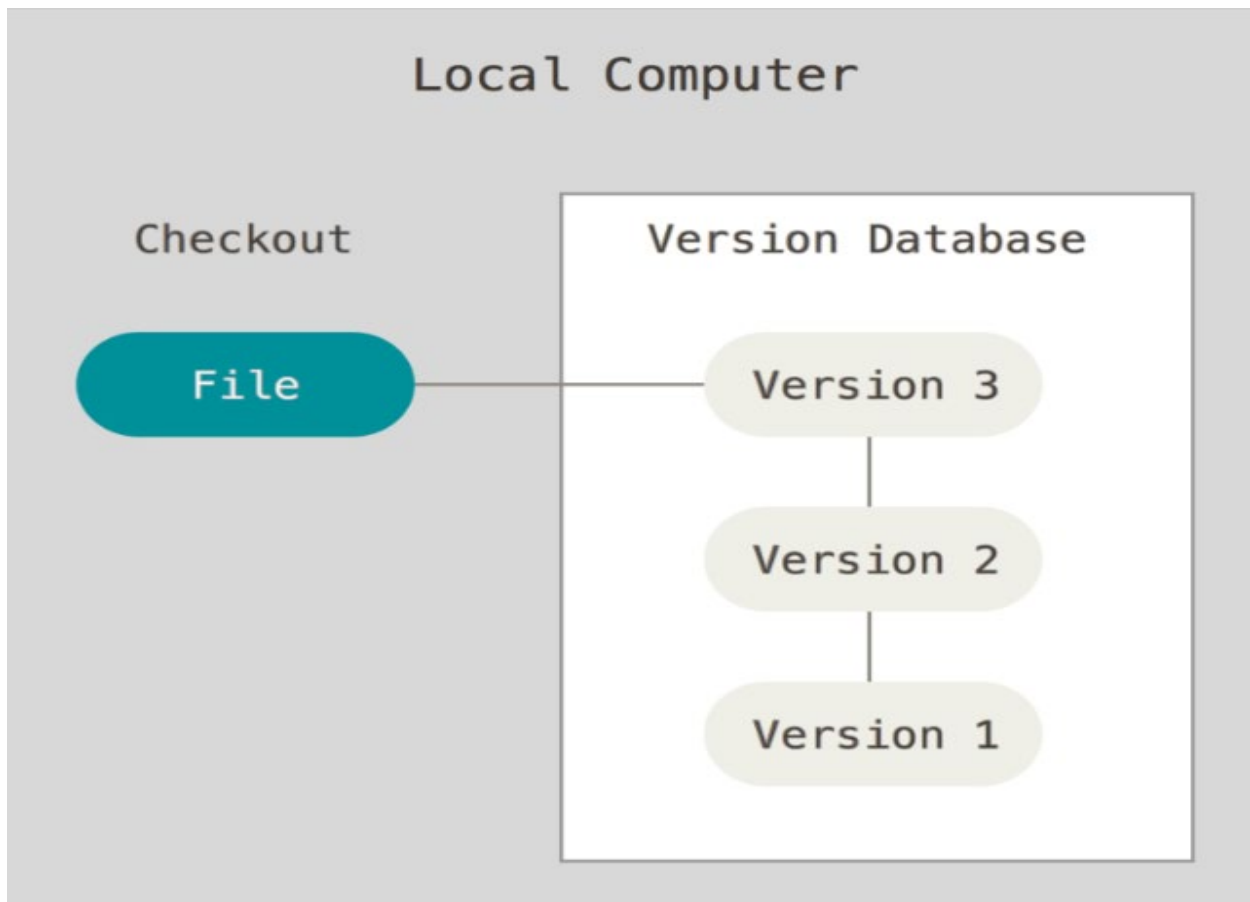


Fig. 3-1 Reprezentare sistem de control al versiunilor locale

3.1.1.2. CVCS

Următoarea problemă majoră pe care o întâmpină oamenii este că trebuie să colaboreze cu dezvoltatorii pe alte sisteme. Pentru a face față acestei probleme, au fost dezvoltate sisteme centralizate de control al versiunilor (CVCS). Aceste sisteme (cum ar fi CVS, Subversion și Perforce) au un singur server care conține toate fișierele versionate și un număr de clienți care verifică fișierele din acel loc central. De mulți ani, acesta a fost standardul pentru controlul versiunilor.

Această configurare oferă multe avantaje, în special față de VCS-urile locale. De exemplu, toată lumea știe într-o anumită măsură ce fac toți ceilalți din proiect. Administratorii au un control fin asupra cine poate face ce și este mult mai ușor să administrezi un CVCS decât să te ocupi de bazele de date locale pentru fiecare client.

Cu toate acestea, această configurație are și unele dezavantaje serioase. Cel mai evident este singurul punct de eșec pe care îl reprezintă serverul centralizat. Dacă acel server se defectează timp de o oră, atunci în timpul acelei ore nimeni nu poate colabora deloc sau salva modificările versiunilor la orice lucru la care lucrează. Dacă hard disk-ul pe care se află baza de date centrală devine corupt și nu s-au păstrat copiile de rezervă adecvate, pierzi absolut totul - întreaga istorie a proiectului, cu excepția oricăror instanțanee unice pe care oamenii le au pe mașinile locale. VCS-urile locale suferă de aceeași problemă - de fiecare dată când aveți întreaga istorie a proiectului într-un singur loc, riscați să pierdeți totul.

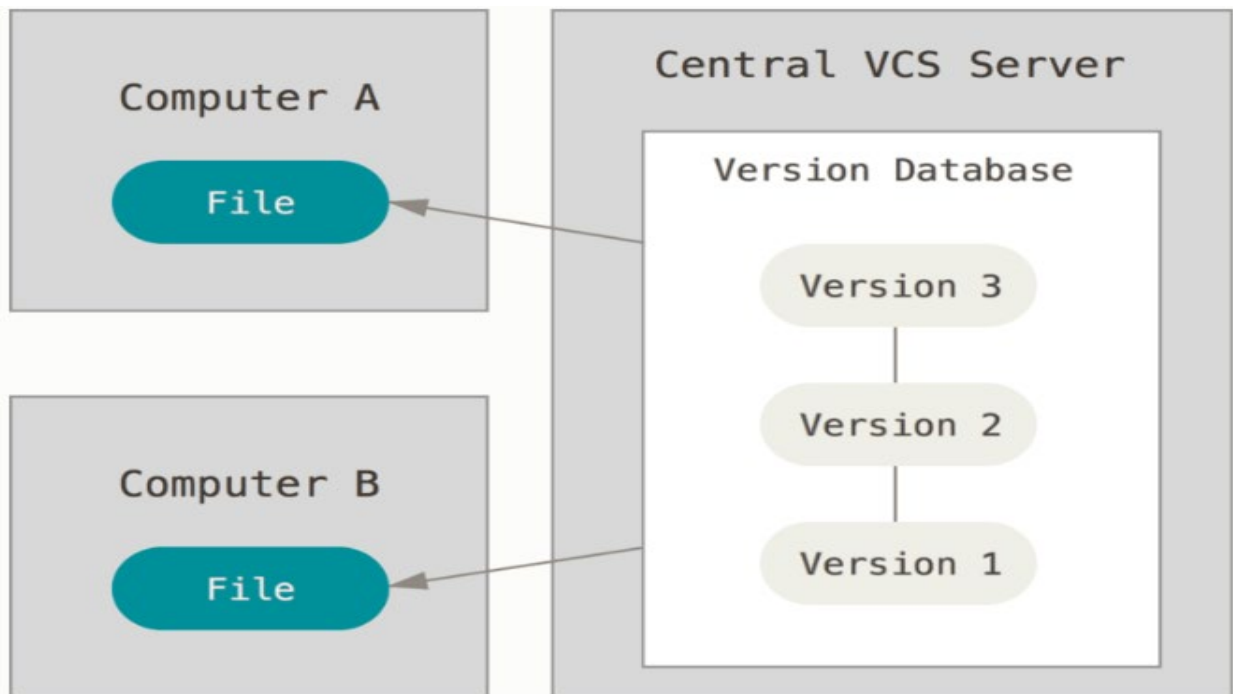


Fig. 3-2 Reprezentare sistem de control al versiunilor centralizate

3.1.1.3. DVCS

Aici intervin sistemele de control al versiunilor distribuite (DVCS). Într-un DVCS (cum ar fi Git, Mercurial, Bazaar sau Darcs), clienții nu verifică doar cel mai recent instantaneu al fișierelor; mai degrabă, ele oglindesc complet depozitul, inclusiv istoricul complet al acestuia. Astfel, dacă vreun server moare și aceste

sisteme colaborau prin acel server, oricare dintre depozitele clientului poate fi copiat înapoi pe server pentru a-l restaura. Fiecare clonă este într-adevăr o copie de rezervă completă a tuturor datelor.

În plus, multe dintre aceste sisteme se descurcă destul de bine cu mai multe depozite de la distanță cu care pot lucra, astfel încât să puteți colabora cu diferite grupuri de oameni în moduri diferite simultan în cadrul aceluiași proiect. Acest lucru vă permite să configurați mai multe tipuri de fluxuri de lucru care nu sunt posibile în sistemele centralizate, cum ar fi modelele ierarhice.

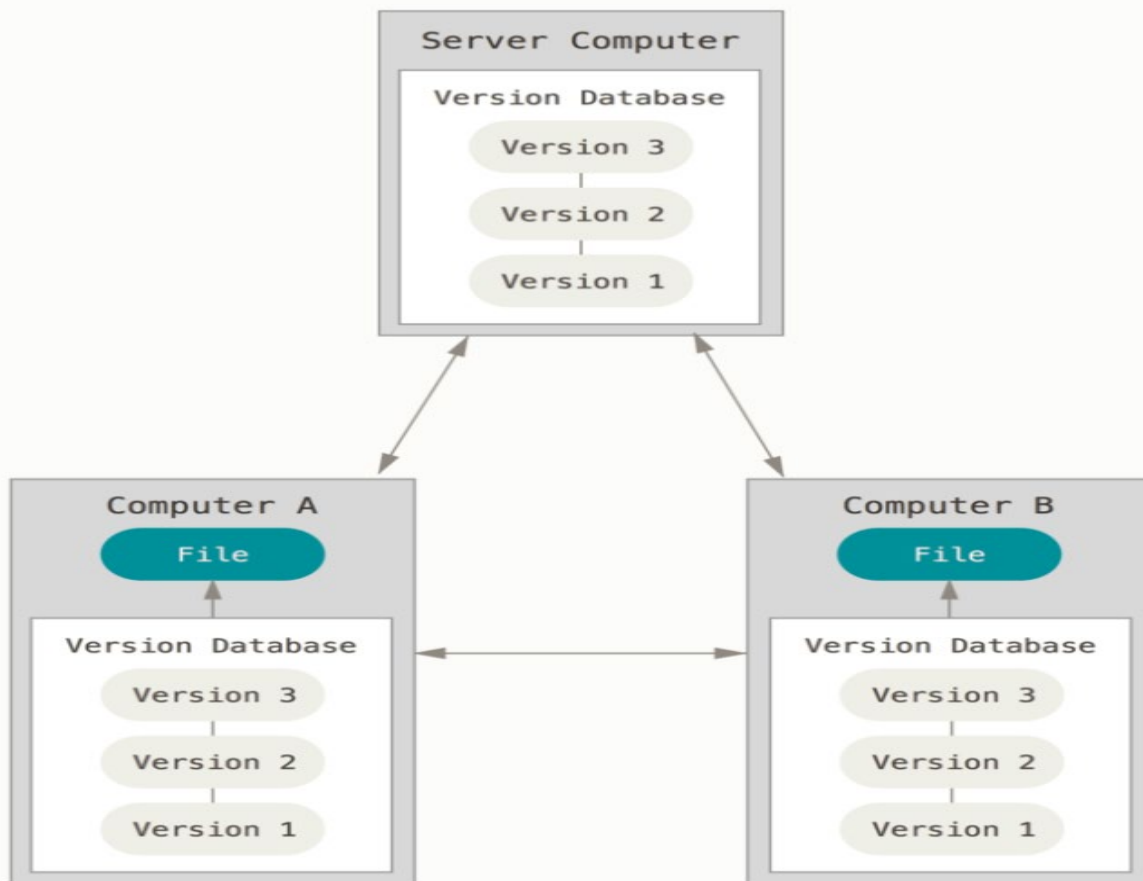


Fig. 3-3 Reprezentare sistem de control al versiunilor distribuite

3.1.2. Caracteristici

Un istoric complet al modificărilor pe termen lung a fiecărui fișier. Aceasta înseamnă fiecare schimbare făcută de mulți indivizi de-a lungul anilor. Modificările includ crearea și ștergerea fișierelor, precum și modificările conținutului acestora. Diferite instrumente VCS diferă în ceea ce privește cât de bine gestionează redenumirea și mutarea fișierelor. Acest istoric ar trebui să

includă, de asemenea, autorul, data și note scrise cu privire la scopul fiecărei modificări. Deținerea istoricului complet permite revenirea la versiunile anterioare pentru a ajuta la analiza cauzei rădăcină a erorilor și este esențial atunci când trebuie să remediați problemele în versiunile mai vechi de software.

Ramificare și contopire. VCS menține mai multe fluxuri de lucru independente unele de altele, oferind, de asemenea, posibilitatea de a îmbina aceste lucrări împreună, permițând dezvoltatorilor să verifice dacă modificările de pe fiecare ramură nu intră în conflict. Multe echipe de software adoptă o practică de ramificare pentru fiecare caracteristică sau poate ramificare pentru fiecare versiune, sau ambele. Există multe fluxuri de lucru diferite din care echipele pot alege atunci când decid cum să folosească facilitățile de ramificare și fuziune în VCS.

Trasabilitate. Posibilitatea de a urmări fiecare modificare efectuată în software și de a o conecta la software-ul de management al proiectelor și de urmărire a erorilor, cum ar fi Jira, și posibilitatea de a adnota fiecare modificare cu un mesaj care descrie scopul și intenția modificării poate ajuta nu numai la analiza cauzei principale, și alte criminalistice.

3.2. Git [6]

Git este un software pentru urmărirea modificărilor în orice set de fișiere, folosit de obicei pentru coordonarea muncii între programatori care dezvoltă codul sursă în colaborare în timpul dezvoltării software. Obiectivele sale includ viteza, integritatea datelor și suport pentru fluxuri de lucru distribuite, neliniare (mii de ramuri paralele care rulează pe sisteme diferite).

Principiu de funcționare: fișierele/sursele sunt păstrate într-un repository pe un server accesibil tuturor dezvoltatorilor care are ca scop sincronizarea surselor. Fiecare dezvoltator are o copie locală a repository-ului asupra căreia va efectua modificări. Aceste modificări se comunică server-ului la momentul ales de utilizator.

Git stochează informația ca pe o listă de snapshot-uri, astfel, cu fiecare commit git salvează starea curentă a proiectului. Dacă un fișier nu a fost modificat, acesta salvează doar o referință către fișier dintr-un commit anterior. Este un sistem de gestionare distribuit, fiecare client având păstrat local întregul proiect. Astfel, deși se lucrează cu copia locală, avem acces la toate fișierele și la întregul istoric de dezvoltare.

3.2.1. Caracteristici

Suport puternic pentru dezvoltarea neliniară: Git acceptă ramificarea (branching) și îmbinarea rapidă (merging) și include instrumente specifice pentru

vizualizarea și navigarea unui istoric de dezvoltare neliniar. În Git, o presupunere de bază este că o modificare va fi îmbinată mai des decât este scrisă, deoarece este transmisă diverșilor recenzenți. În Git, ramurile sunt foarte ușoare: o ramură este doar o referință la un commit. Cu comiterile sale parentale, structura completă a ramurilor poate fi construită.

Dezvoltare distribuită: la fel ca Darcs, BitKeeper, Mercurial, Bazaar și Monotone, Git oferă fiecărui dezvoltator o copie locală a istoricului complet al dezvoltării, iar modificările sunt copiate dintr-un astfel de depozit în altul. Aceste modificări sunt importate ca ramuri de dezvoltare adăugate și pot fi fuzionate în același mod ca o ramură dezvoltată local.

Compatibilitate cu sistemele și protocoalele existente: arhivele pot fi publicate prin protocolul de transfer hipertext (HTTP), protocolul de transfer de fișiere (FTP) sau printr-un protocol Git, fie printr-un socket simplu, fie prin Secure Shell (SSH). Git are, de asemenea, o emulare de server CVS, care permite utilizarea clienților CVS existenți și a pluginurilor IDE pentru a accesa depozitele Git.

Gestionarea eficientă a proiectelor mari: Git este foarte rapid și scalabil, iar testele de performanță efectuate de Mozilla au arătat că era cu un ordin de mărime mai rapid decât unele sisteme de control al versiunilor; preluarea istoricului versiunilor dintr-un depozit stocat local poate fi de o sută de ori mai rapidă decât preluarea lui de la serverul de la distanță.

Autentificare criptografică a istoriei: istoricul Git este stocat în așa fel încât ID-ul unei anumite versiuni (un commit în termeni Git) depinde de istoricul complet de dezvoltare care a condus la acea comitere. Odată publicat, nu este posibil să se schimbe versiunile vechi fără a fi observat. Structura este similară cu un arbore Merkle, dar cu date adăugate la noduri și frunze.

Design bazat pe toolkit: Git a fost conceput ca un set de programe scrise în C și mai multe script-uri shell care oferă pachete în jurul acelor programe. Deși majoritatea acestor scripturi au fost rescrise de atunci în C pentru viteză și portabilitate, designul rămâne și este ușor să legați componentele împreună.

Strategii de îmbinare conectabile: Ca parte a designului setului său de instrumente, Git are un model bine definit de îmbinare incompletă și are mai mulți algoritmi pentru finalizarea acestora, culminând cu a spune utilizatorului că nu poate finaliza automat îmbinarea și că este necesară editarea manuală.

Ambalare periodică explicită a obiectelor: Git stochează fiecare obiect nou creat ca fișier separat. Deși este comprimat individual, acesta necesită mult spațiu și este ineficient. Acest lucru este rezolvat prin utilizarea pachetelor care stochează un număr mare de obiecte comprimate delta între ele într-un singur fișier (sau flux de octeți de rețea) numit packfile. Pachetele sunt comprimate folosind euristica că

fișierele cu același nume sunt probabil similare, fără a depinde de aceasta pentru corectitudine. Un fișier index corespunzător este creat pentru fiecare fișier pachet, indicând offset-ul fiecărui obiect din fișierul pachet. Obiectele nou create (cu istoric nou adăugat) sunt încă stocate ca obiecte individuale, iar reambalarea periodică este necesară pentru a menține eficiența spațiului. Procesul de împachetare a depozitului poate fi foarte costisitor din punct de vedere computațional. Permițând obiectelor să existe în depozit într-un format liber, dar generat rapid, Git permite ca operațiunea costisitoare a pachetului să fie amânată până mai târziu, când timpul contează mai puțin, de exemplu, sfârșitul unei zile de lucru. Git face automat reambalarea periodică, dar reambalarea manuală este posibilă și cu comanda `git gc`.

3.2.2. Structuri de date

Git are două structuri de date: un index mutabil (numit și stadiu sau cache) care memorează în cache informații despre directorul de lucru și următoarea revizuire care urmează să fie comisă; și o bază de date de obiecte imuabilă, numai pentru atașare. Indexul servește ca punct de legătură între baza de date de obiecte și arborele de lucru. Baza de date de obiecte conține cinci tipuri de obiecte:

- Un blob (obiect binar mare) este conținutul unui fișier. Blob-urile nu au un nume de fișier adecvat, marcaje temporale sau alte metadate (Numele unui blob în interior este un hash al conținutului său). În git, fiecare blob este o versiune a unui fișier, deține datele fișierului.
- Un obiect arbore este echivalentul unui director. Conține o listă de nume de fișiere, fiecare cu niște biți de tip și o referință la un obiect blob sau arbore care este acel fișier, link simbolic sau conținutul directorului. Aceste obiecte sunt un instantaneu al arborelui sursă. (În totalitate, acesta cuprinde un arbore Merkle, ceea ce înseamnă că doar un singur hash pentru arborele rădăcină este suficient și folosit efectiv în commit-uri pentru a identifica cu precizie starea exactă a structurilor întregi de arbore a oricărui număr de subdirectoare și fișiere.)
- Un obiect commit leagă obiectele arborescente împreună în istorie. Conține numele unui obiect arborescent (din directorul sursă de nivel superior), o marcat de timp, un mesaj de jurnal și numele zero sau mai multe obiecte de comitere părinte.
- Un obiect etichetă este un container care conține o referință la un alt obiect și poate conține metadate adăugate legate de un alt obiect. Cel mai frecvent, este folosit pentru a stoca o semnătură digitală a unui obiect de comitere corespunzătoare unei anumite ediții a datelor urmărite de Git.

- Un obiect packfile este o versiune zlib comprimată din diverse alte obiecte pentru compactitate și ușurință de transport prin protocoalele de rețea.

Fiecare obiect este identificat printr-un hash SHA-1 al conținutului său. Git calculează hash-ul și folosește această valoare pentru numele obiectului. Obiectul este introdus într-un director care se potrivește cu primele două caractere ale hashului său. Restul hash-ului este folosit ca nume de fișier pentru acel obiect.

Git stochează fiecare revizuire a unui fișier ca un blob unic. Relațiile dintre blob-uri pot fi găsite prin examinarea arborelui și a obiectelor commit. Obiectele nou adăugate sunt stocate în întregime folosind compresia zlib. Acest lucru poate consuma rapid o cantitate mare de spațiu pe disc, astfel încât obiectele pot fi combinate în pachete, care utilizează compresia delta pentru a economisi spațiu, stocând blob-urile ca modificări în raport cu alte blob-uri.

În plus, git stochează etichete numite refs (prescurtare pentru referințe) pentru a indica locațiile diferitelor comiteri. Acestea sunt stocate în baza de date de referință și, respectiv, sunt:

- Capete (heads - branches): referințe denumite care sunt avansate automat către noul commit atunci când se face un commit peste ele.
- HEAD: Un cap rezervat care va fi comparat cu arborele de lucru pentru a crea un commit.
- Etichete (tags): ca referințele la ramuri, dar fixate pentru un anumit commit. Folosit pentru a eticheta puncte importante din istorie.

3.3. OpenAI

API-ul OpenAI poate fi aplicat practic la orice sarcină care implică înțelegerea sau generarea unui limbaj natural sau cod. Oferă un spectru de modele cu diferite niveluri de putere potrivite pentru diferite sarcini, precum și capacitatea de a-ți regla propriile modele personalizate. Aceste modele pot fi folosite pentru orice, de la generarea de conținut până la căutare semantică și clasificare.

3.3.1. Tokens

Modelele folosite de OpenAI înțeleg și procesează textul prin descompunerea lui în simboluri. Token-urile pot fi cuvinte sau doar bucăți de caractere. De exemplu, cuvântul “hamburger” este împărțit în simbolurile ”ham”, ”bur” și ”ger”, în timp ce un cuvânt scurt și obișnuit precum ”pere” este un singur simbol. Multe jetoane încep cu un spațiu alb, de exemplu “da”.

Numărul de jetoane procesate într-o anumită solicitare API depinde de lungimea atât a intrărilor, cât și a ieșirilor. Ca regulă generală, 1 simbol are

aproximativ 4 caractere sau 0,75 cuvinte pentru textul în limba engleză. O limitare de reținut este că solicitarea textului și completarea generată combinate nu trebuie să depășească lungimea maximă a contextului modelului (pentru majoritatea modelelor, aceasta este de 2048 de jetoane sau aproximativ 1500 de cuvinte).

3.3.2. Modele

API-ul este alimentat de un set de modele cu capacități și puncte de preț diferite. Modelele Davinci, Curie, Babbage și Ada se bazează pe GPT-3. Seria Codex este un descendent al GPT-3 care a fost instruit atât pe limbaj natural, cât și pe cod. OpenAI Codex are cunoștințe ample despre modul în care oamenii folosesc codul și este mult mai capabil decât GPT-3 în generarea de cod, în parte, deoarece a fost instruit pe un set de date care include o concentrație mult mai mare de cod sursă public.

Este un nou instrument de învățare automată care traduce textul în limba engleză în cod. Codex este conceput pentru a accelera munca programatorilor profesioniști, precum și pentru a ajuta amatorii să înceapă codificarea. Potrivit OpenAI, versiunea actuală a Codex are o precizie de 37 la sută în sarcinile de codare, spre deosebire de zero la sută a lui GPT-3.

- Davinci este cel mai capabil motor și poate îndeplini orice sarcină pe care celelalte modele le pot îndeplini și adesea cu mai puține instrucțiuni. Pentru aplicațiile care necesită multă înțelegere a conținutului, cum ar fi rezumarea pentru un anumit public și generarea de conținut creativ, Davinci va produce cele mai bune rezultate.
- Curie este extrem de puternic, dar foarte rapid. În timp ce Davinci este mai puternic când vine vorba de analiza textului complicat, Curie este destul de capabil pentru multe sarcini nuanțate, cum ar fi clasificarea sentimentelor și rezumarea. Curie este, de asemenea, destul de bun în a răspunde la întrebări și a efectua întrebări și răspunsuri și ca chatbot de serviciu general.
- Babbage poate îndeplini sarcini simple, cum ar fi clasificarea simplă. Este, de asemenea, destul de capabil când vine vorba de clasarea Căutării semantice cât de bine se potrivesc documentele cu interogările de căutare.
- Ada este de obicei cel mai rapid model și poate efectua sarcini precum analiza textului, corectarea adresei și anumite tipuri de sarcini de clasificare care nu necesită prea multe nuanțe. Performanța Adei poate fi adesea îmbunătățită oferind mai mult context.

3.4. GPT-3 [7]

GPT-3 este un model de învățare automată a rețelei neuronale antrenat folosind date de pe internet pentru a genera orice tip de text. Dezvoltat de OpenAI, necesită o cantitate mică de text de intrare pentru a genera volume mari de text relevant și sofisticat generat de mașină.

Rețeaua neuronală de învățare profundă a GPT-3 este un model cu peste 175 de miliarde de parametri de învățare automată. Pentru a pune lucrurile la scară, cel mai mare model de limbaj instruit înainte de GPT-3 a fost modelul Turing NLG de la Microsoft, care avea 10 miliarde de parametri. De la începutul anului 2021, GPT-3 este cea mai mare rețea neuronală produsă vreodată. Ca rezultat, GPT-3 este mai bun decât orice model anterior pentru a produce text suficient de convingător încât să pară că un om ar fi putut să-l scrie.

Procesarea limbajului natural include ca una dintre componentele sale majore generarea limbajului natural, care se concentrează pe generarea de text natural în limbajul uman. Cu toate acestea, generarea de conținut ușor de înțeles uman este o provocare pentru mașinile care nu cunosc cu adevărat complexitățile și nuanțele limbajului. Folosind text de pe internet, GPT-3 este antrenat să genereze text uman realist.

GPT-3 a fost folosit pentru a crea articole, poezii, povestiri, știri și dialoguri folosind doar o cantitate mică de text de intrare care poate fi folosit pentru a produce cantități mari de copii de calitate. De asemenea, folosit pentru sarcini conversaționale automate, răspunzând oricărui text pe care o persoană îl tastează pe computer cu o nouă bucată de text adecvată contextului. GPT-3 poate crea orice cu o structură de text, și nu doar text în limbaj uman. De asemenea, poate genera automat rezumate de text și chiar cod de programare.

3.5. Sandbox [8]

Un mediu sandbox este o mașină virtuală izolată în care se poate executa cod software posibil nesigur fără a afecta resursele de rețea sau aplicațiile locale.

Cercetătorii în domeniul securității cibernetice folosesc sandbox-uri pentru a rula cod suspect din atașamente și adrese URL necunoscute și pentru a observa comportamentul acestuia. Semnele indicatoare includ dacă codul se reproduce, încearcă să contacteze un server de comandă și control, descarcă software suplimentar, criptează date sensibile și așa mai departe. Deoarece sandbox-ul este un mediu emulat fără acces la rețea, date sau alte aplicații, echipele de securitate pot „detona” în siguranță codul pentru a determina cum funcționează și dacă este rău intenționat.

În afara securității cibernetice, dezvoltatorii folosesc, de asemenea, medii de testare sandbox pentru a rula cod înainte de implementarea pe scară largă.

Modul în care funcționează un sandbox depinde de ceea ce este testat. De exemplu, un mediu sandbox folosit pentru a testa programele malware este configurat și funcționează diferit față de un sandbox menit să testeze codul pentru actualizările aplicațiilor. Pentru cercetarea potențialului malware și execuția de cod rău intenționat, un sandbox necesită izolarea de software-ul de producție.

Indiferent de modul în care este utilizat un sandbox, fiecare mediu are câteva caracteristici de bază:

- **Emularea unui dispozitiv real.** Aceasta ar putea fi emularea unui desktop sau a unui dispozitiv mobil. În ambele cazuri, aplicația testată trebuie să aibă acces la aceleași resurse ca și codul analizat, inclusiv CPU, memorie și stocare.
- **Emularea sistemului de operare țintă.** Folosind o mașină virtuală, aplicația trebuie să aibă acces la sistemul de operare. Cu o mașină virtuală, sandbox-ul este izolat de hardware-ul fizic subiacent, dar are acces la sistemul de operare instalat.
- **Mediu virtualizat.** De obicei, un sandbox se află pe o mașină virtuală, astfel încât nu are acces la resurse fizice, dar poate accesa hardware virtualizat.

3.6. Regresie liniară [9]

Analiza de regresie liniară este utilizată pentru a prezice valoarea unei variabile pe baza valorii altei variabile. Variabila care se dorește a fi prezisă se numește variabilă dependentă. Variabila care se utilizează pentru a prezice cealaltă valoare se numește variabilă independentă.

Această formă de analiză estimează coeficienții ecuației liniare, implicând una sau mai multe variabile independente care prezic cel mai bine valoarea variabilei dependente. Regresia liniară se potrivește unei linii drepte sau unei suprafețe care minimizează discrepanțele dintre valorile de ieșire prezise și cele reale. Există calculatoare simple de regresie liniară care folosesc metoda “cel mai mici pătrate” pentru a descoperi linia cea mai potrivită pentru un set de date pereche.

Se poate efectua regresia liniară în Microsoft Excel sau se poate utiliza pachete software statistice, care simplifică foarte mult procesul de utilizare a ecuațiilor de regresie liniară, modelelor de regresie liniară și formulei de regresie liniară. Statisticile SPSS pot fi utilizate în tehnici precum regresia liniară simplă și

regresia liniară multiplă. Se poate efectua metoda de regresie liniară într-o varietate de programe și medii, inclusiv:

- R regresie liniară
- Regresia liniară MATLAB
- Regresia liniară Sklearn
- Regresie liniară Python
- Regresia liniară Excel

Modelele de regresie liniară sunt relativ simple și oferă o formulă matematică ușor de interpretat care poate genera predicții. Deoarece regresia liniară este o procedură statistică stabilită de mult timp, proprietățile modelelor de regresie liniară sunt bine înțelese și pot fi antrenate foarte repede.

4. STRUCTURA PROIECTULUI

4.1. API-uri și biblioteci folosite

În cadrul proiectului s-au utilizat biblioteci și API-uri disponibile pe internet sau puse la dispoziție de către sistemul de operare Windows pentru a rezolva anumite funcționalități necesare în implementarea proiectului. Acest subcapitol prezintă scopul fiecărei biblioteci, structura acesteia pe scurt, integrarea în proiect și funcționalitățile mai importante folosite în proiect.

4.1.1. Nodegit

NodeGit este un modul nativ asincron care permite să apelez libgit2. Asta înseamnă că un proiect node care folosește NodeGit poate face comenzi de git de nivel scăzut fără a avea instalat git pe acea mașină. În node, majoritatea modulelor sunt scrise în javascript. Unele module, cum ar fi modulul fs, sunt scrise în C/C++, deoarece fișierele nu se pot edita din pur javascript. Aceste module sunt numite „native” deoarece codul pentru aceste module este ușor diferit în funcție de SO-ul pe care rulează.

libgit2 este o implementare portabilă, pură în C a metodelor de bază Git, furnizată ca bibliotecă re-intrată care poate fi conectată cu un API solid, permițând scrierea de aplicații Git personalizate cu viteză nativă în orice limbă care acceptă legături C.

În cadrul acestei lucrări s-au folosit apeluri către mai multe funcții puse la dispoziție de această librărie:

- Clone() – clonarea repository-urilor aflate pe GitHub
- Open() – deschide repository-ul clonat

- getCommit() – întoarce commit-ul specificat
- getEntry() – întoarce fișierul specificat
- getBlob() – returnează content-ul fișierului
- getCommits() – returnează un număr de commit-uri specificat
- defaultNew() – folosește credențialele default
- userpassPlaintextNew() – setează un username și un token pentru validarea clonării

Mai multe detalii despre această librărie sunt date în subcapitolul **Implementare software.**

4.1.2. Valgrind [10]

Pentru implementarea rezolvării leak-urilor de memorie s-a folosit valgrind. Valgrind este un instrument de analiză dinamică binară (DPA) care utilizează cadrul de instrumentare dinamică binară (DPI) pentru a verifica alocarea memoriei, pentru a detecta blocajele și pentru a profila aplicațiile. Cadrul DPI are propriul său manager de memorie de nivel scăzut, planificator, handler de fire și handler de semnal. Suita de instrumente Valgrind include instrumente similare:

- Memcheck: urmărește alocarea memoriei în mod dinamic și raportează scurgerile de memorie.
- Helgrind: detectează și raportează blocajele, eventualele curse de date și inversările de blocare.
- Cachegrind: simulează modul în care aplicația interacționează cu memoria cache a sistemului și oferă informații despre erorile din cache.
- Nulgrind: un valgrind simplu care nu face niciodată nicio analiză. Folosit de dezvoltatori pentru benchmark de performanță.
- Massif: un instrument pentru analizarea utilizării memoriei heap a aplicației.

Instrumentul Valgrind folosește un mecanism de dezasamblare și resintetizare în care încarcă aplicația într-un proces, dezasamblează codul aplicației, adaugă codul de instrumentare pentru analiză, o assemblează înapoi și execută aplicația. Utilizează JIT pentru a încorpora aplicația cu codul de instrumentare.

Valgrind Core dezasamblează codul aplicației și transmite fragmentul de cod pluginului de instrumente pentru instrumentare. Pluginul pentru instrument adaugă codul de analiză și îl assemblează înapoi. Astfel, Valgrind oferă flexibilitatea de a scrie propriul nostru instrument peste cadrul Valgrind. Valgrind folosește registre și memorie pentru a instrumenta instrucțiuni de citire/scriere, apel de sistem de citire/scriere, alocări de stivă și heap.

Valgrind oferă pachete în jurul apelului de sistem și se înregistrează pentru apeluri pre și post pentru fiecare apel de sistem pentru a urmări memoria accesată ca parte a apelului de sistem. Astfel, Valgrind este un strat de abstractizare a sistemului de operare între sistemul de operare Linux și aplicația client. Practic valgrind rulează aplicația într-un „sandbox”. În timp ce rulează în acest sandbox, este capabil să insereze propriile instrucțiuni pentru a face depanare avansată și profilare.

Diagrama ilustrează cele 8 faze ale Valgrind:

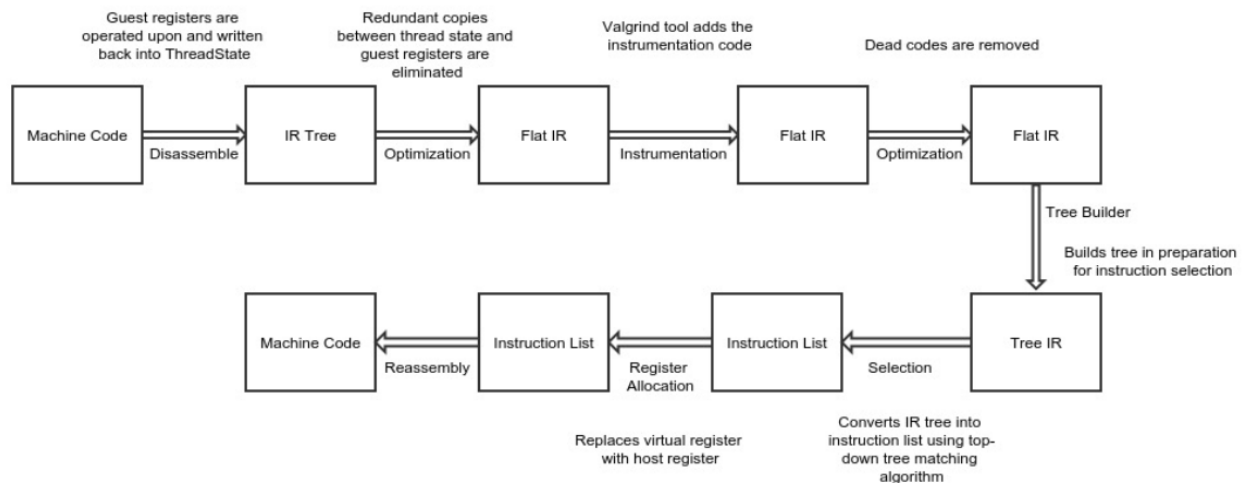


Fig. 4-1 Reprezentarea pașilor pe care instrumentul valgrind le urmează

4.1.3.OpenAI API

Pentru afișarea de sugestii în rezolvare unor bug-uri în cod, explicarea unor funcții sau calcularea complexității acestora am folosit API-ul creat de cei de la OpenAI. Pentru a putea folosi acest API este necesară instalarea librării aferente pentru Node.js. Folosirea acestuia se face pe baza unei cheie generate de aceștia. În cadrul acestei lucrări s-au folosit apeluri către mai multe funcții puse la dispoziție de librăria openai:

- Configuration() – setarea configurației folosind cheia de acces primită
- OpenAIApi() – crearea unei variabile pe baza configurației
- createCompletion() – crearea unui request către API

Mai multe detalii legate de modul de utilizare si de creare de cereri HTTP sunt date în capitolul de **Implementare/Contribuții personale**.

4.1.4. Scikit-learn

Scikit-learn este o bibliotecă software de învățare automată pentru limbajul de programare Python. Dispune de diverși algoritmi de clasificare, regresie și grupare, inclusiv mașini cu vector de suport, păduri aleatorii, creștere a gradientului, k-means și DBSCAN și este proiectat pentru a interopera cu bibliotecile numerice și științifice Python NumPy și SciPy.

Acesta este scris în mare parte în Python și utilizează extensiv NumPy pentru operații de înaltă performanță de algebră liniară și pentru operații cu vectori și matrici. În plus, unii algoritmi de bază sunt scrieți în Cython pentru a îmbunătăți performanța. Mașinile de suport vector sunt implementate de Cython folosind LIBSVM; regresia logistică și mașina de suport liniară vector sunt implementate la fel dar în jurul LIBLINEAR. În astfel de cazuri, extinderea acestor metode cu Python ar putea să nu fie posibilă.

Scikit-learn se integrează bine cu multe alte biblioteci Python, cum ar fi Matplotlib și plotly pentru reprezentare grafică, NumPy pentru vectorizarea matricei, cadre de date Pandas, SciPy și multe altele.

4.1.5. React

React (React.js sau ReactJS) este o bibliotecă JavaScript open-source pentru construirea de interfețe cu utilizatorul bazate pe componente UI. React poate fi folosit ca bază în dezvoltarea de aplicații cu o singură pagină, mobile sau randate pe server cu cadre precum Next.js. Cu toate acestea, React se preocupă doar de gestionarea stării și de redarea acelei stări către DOM, astfel încât crearea aplicațiilor React necesită de obicei utilizarea de biblioteci suplimentare pentru rutare, precum și anumite funcționalități pe partea clientului.

4.1.5.1. Caracteristici

React aderă la paradigma de programare declarativă. Dezvoltatorii proiectează vizualizări pentru fiecare stare a unei aplicații, iar React actualizează și redă componente atunci când datele se modifică. Acest lucru este în contrast cu programarea imperativă.

Codul React este format din entități numite componente. Aceste componente sunt reutilizabile și trebuie formate în folderul SRC urmând Cazul Pascal ca convenție de denumire (capitalizare camelCase). Componentele pot fi redade la un anumit element din DOM folosind biblioteca React DOM.

Componentele de tip funcție sunt declarate cu o funcție care returnează apoi niște JSX.

Componentele bazate pe clasă sunt declarate folosind clase ES6.

Acolo unde componentele de clasă se referă la utilizarea claselor și a metodelor ciclului de viață, componentele funcționale folosesc hook-uri pentru a face față managementului stării și altor probleme care apar la scrierea codului în React.

O altă caracteristică notabilă este utilizarea unui model de obiect de document virtual sau DOM virtual. React creează un cache pentru structura de date în memorie, calculează diferențele rezultate și apoi actualizează eficient DOM afișat în browser. Acest proces se numește reconciliere. Acest lucru permite programatorului să scrie cod ca și cum întreaga pagină este redată la fiecare modificare, în timp ce bibliotecile React redă doar subcomponentele care se schimbă efectiv. Această randare selectivă oferă o creștere majoră a performanței. Economisește efortul de a recalcula stilul CSS, aspectul paginii și randarea pentru întreaga pagină.

Metodele ciclului de viață pentru componentele bazate pe clasă folosesc o formă de agățare care permite executarea codului la punctele stabilite pe durata de viață a unei componente.

- *shouldComponentUpdate()* permite dezvoltatorului să prevină redarea inutilă a unei componente, returnând false dacă nu este necesară o randare.
- *componentDidMount()* este apelat odată ce componenta s-a montat (componenta a fost creată în interfața cu utilizatorul, adesea prin asocierea cu un nod DOM). Acesta este folosit în mod obișnuit pentru a declanșa încărcarea datelor de la o sursă la distanță printr-un API.
- *componentWillUnmount()* este apelat imediat înainte ca componenta să fie demontată. Acesta este folosit în mod obișnuit pentru a șterge dependențele care necesită resurse de componentă care nu vor fi eliminate pur și simplu odată cu demontarea componentei (de exemplu, eliminarea oricăror instanțe *setInterval()* care sunt legate de componentă sau un „eventListener” setat pe „ document” din cauza prezenței componentei)
- *render* este cea mai importantă metodă ciclului de viață și singura necesară în orice componentă. Este de obicei apelat de fiecare dată când starea componentei este actualizată, ceea ce ar trebui să se reflecte în interfața cu utilizatorul.

JSX este o extensie a sintaxei limbajului JavaScript. Similar ca aspect cu HTML, JSX oferă o modalitate de a structura randarea componentelor folosind o sintaxă familiară multor dezvoltatori. Componentele React sunt scrise de obicei folosind JSX, deși nu trebuie să fie (componentele pot fi scrise și în JavaScript

pur). JSX este similar cu o altă sintaxă de extensie creată de Facebook pentru PHP numită XHP.

Hook-urile sunt funcții care le permit dezvoltatorilor să se conecteze la funcțiile React și ale ciclului de viață din componentele funcției. Acestea nu funcționează în cadrul claselor. React oferă câteva hook-uri încorporate, cum ar fi *useState*, *useContext*, *useReducer*, *useMemo* și *useEffect*. Altele sunt documentate în Hooks API Reference. *useState* și *useEffect*, sunt cele mai frecvent utilizate, pentru controlul stării și, respectiv, a efectelor secundare.

Există reguli care descriu modelul de cod caracteristic pe care se bazează hook-urile. Este modalitatea modernă de a gestiona starea cu React.

- Hook-urile ar trebui să fie apelate numai la nivelul superior (nu în interiorul buclor sau declarațiilor if).
- Hook-urile ar trebui să fie apelate numai din componentele funcției React și din hook-urile personalizate, nu din funcții normale sau componente de clasă.

Deși aceste reguli nu pot fi aplicate în timpul execuției, instrumentele de analiză a codului, cum ar fi linters, pot fi configurate pentru a detecta multe greșeli în timpul dezvoltării.

4.1.6. NodeJs

Node.js este un mediu de rulare JavaScript open-source, multiplatformă, care rulează pe motorul V8 și execută cod JavaScript în afara unui browser web. Node.js le permite dezvoltatorilor să folosească JavaScript pentru a scrie instrumente de linie de comandă și pentru script-uri pe partea serverului - rulând scripturi pe partea serverului pentru a produce conținut dinamic al paginii web înainte ca pagina să fie trimisă la browserul web al utilizatorului. În consecință, Node.js reprezintă o paradigmă JavaScript unificând dezvoltarea aplicațiilor web în jurul unui singur limbaj de programare, mai degrabă decât în limbaje diferite pentru scripturile de pe partea de server și de pe partea clientului.

Node.js are o arhitectură bazată pe evenimente capabilă de I/O asincron. Aceste opțiuni de proiectare urmăresc să optimizeze debitul și scalabilitatea în aplicațiile web cu multe operațiuni de intrare/ieșire, precum și pentru aplicații web în timp real.

4.1.6.1. Detalii tehnice

Node.js este un mediu de rulare JavaScript care procesează cererile primite într-o buclă, numită bucla de evenimente.

Node.js folosește libuv pentru a gestiona evenimente asincrone. Libuv este un strat de abstractizare pentru funcționalitatea rețelei și a sistemului de fișiere atât pe sisteme bazate pe Windows, cât și pe POSIX, cum ar fi Linux, macOS, OSS pe NonStop și Unix.

Node.js operează pe o buclă de evenimente cu un singur fir, folosind apeluri I/O neblocante, permițându-i să suporte zeci de mii de conexiuni simultane fără a suporta costul comutării contextului firului. Proiectarea partajării unui singur fir între toate solicitările care utilizează modelul de observator este destinată construirii de aplicații extrem de concurente, în care orice funcție care realizează I/O trebuie să utilizeze un callback. Pentru a găzdui bucla de evenimente cu un singur thread, Node.js utilizează biblioteca libuv, care, la rândul său, folosește un pool de fire de dimensiune fixă care gestionează unele dintre operațiunile I/O asincrone neblocante.

Un pool de fire de execuție se ocupă de executarea sarcinilor paralele în Node.js. Apelul principal al funcției firului de execuție postează sarcini în coada de activități partajate, în care firele de execuție din pool-ul de fire se extrag și se execută. Funcțiile de sistem inerent neblocante, cum ar fi rețeaua, se traduc în socket-uri neblocante din partea kernelului, în timp ce funcțiile de sistem care blochează inerent, cum ar fi I/O fișierelor, rulează într-un mod blocant pe propriile fire de execuție. Când un fir de execuție din grupul de fire de execuție finalizează o sarcină, acesta informează firul principal despre aceasta, care, la rândul său, se trezește și execută callback-ul înregistrat.

Un dezavantaj al acestei abordări cu un singur thread este că Node.js nu permite scalarea verticală prin creșterea numărului de nuclee CPU ale mașinii pe care rulează fără a utiliza un modul suplimentar

V8 este motorul de execuție JavaScript care compilează codul sursă JavaScript în codul mașină nativ în timpul execuției.

npm este managerul de pachete preinstalat pentru platforma serverului Node.js. Instalează programe Node.js din registrul npm, organizând instalarea și gestionarea programelor Node.js terțe. Pachetele din registrul npm pot varia de la simple biblioteci de ajutor, cum ar fi Lodash, până la rulanți de sarcini precum Grunt.

Node.js se înregistrează cu sistemul de operare, astfel încât sistemul de operare îl anunță despre conexiuni și emite un apel invers. În timpul de execuție Node.js, fiecare conexiune este o alocare de heap mică. În mod tradițional, procesele sau firele de operare relativ grele gestionau fiecare conexiune. Node.js folosește o buclă de evenimente pentru scalabilitate, în loc de procese sau fire. Spre deosebire de alte servere bazate pe evenimente, bucla de evenimente a lui Node.js nu trebuie apelată în mod explicit. În schimb, apelurile inverse sunt definite, iar

serverul intră automat în bucla de evenimente la sfârșitul definiției de apel invers. Node.js iese din bucla de evenimente atunci când nu mai trebuie efectuate apeluri inverse.

4.2. Cerințe proiect

Proiectul își propune mai multe linii de dezvoltare, pe lângă vizualizarea evoluției codului, întrucât în final proiectul este o platformă educativă care servește atât în folosul profesorului cât și a elevului.

- Platforma web va permite utilizatorului să descarce un repository de pe GitHub având posibilitatea să vizualizeze evoluția codului aflat pe acel repository.
- Identificarea unor comportamente ciudate în evoluția codului (de exemplu: introducerea unor bucăți foarte mari de cod de la un commit la altul). Acestea vor fi marcate și listate pentru o analiză mai rapidă.
- Realizarea unui sistem care să permită utilizatorului să aducă modificări codului. Pe lângă opțiunea de vizualizare, acesta să poată să editeze codul și să facă schimbări (push-uri) în timp real pe repository-ul pe care acesta lucrează.
- Detecția unor grupuri de studenți care au colaborat în vederea realizării unui task (teme individuale, exerciții etc.). Analiza nu va fi una sintactică. Se va încerca pe baza datelor oferite de anumite comenzi de git să se observe de exemplu dacă doi studenți au făcut commit în același timp sau la un interval apropiat.
- Verificarea securității codului. Se va încerca detecția unor fișiere sau bucăți de cod care ar putea fi malițioase pentru alți utilizatori. Pe lângă acest lucru totodată se va verifica și dacă codul scris de noi poate fi expus unor vulnerabilități externe.
- Integrare cu API-ul folosit de GitHub-Copilot. Se dorește folosirea acestuia pentru a da hint-uri studenților cu un caracter cât mai educațional.

4.3. Definirea arhitecturii proiectului

Platforma web realizată este structurată în patru componente:

1. Aplicație react ce oferă o interfață grafică utilizatorului prin care acesta poate folosi toate funcționalitățile descrise în subcapitolul **Cerințe proiect**.

2. Un server principal realizat folosind NodeJs, care se ocupă cu repository-urile pe care dorim se le manipulăm și cu integrarea cu API-ul OpenAI.
3. Un server secundar care se ocupă cu verificarea leak-urilor de memorie folosind valgrind.
4. Și un al treilea server care se ocupă cu rularea și afișarea rezultatului unui fragment de cod.

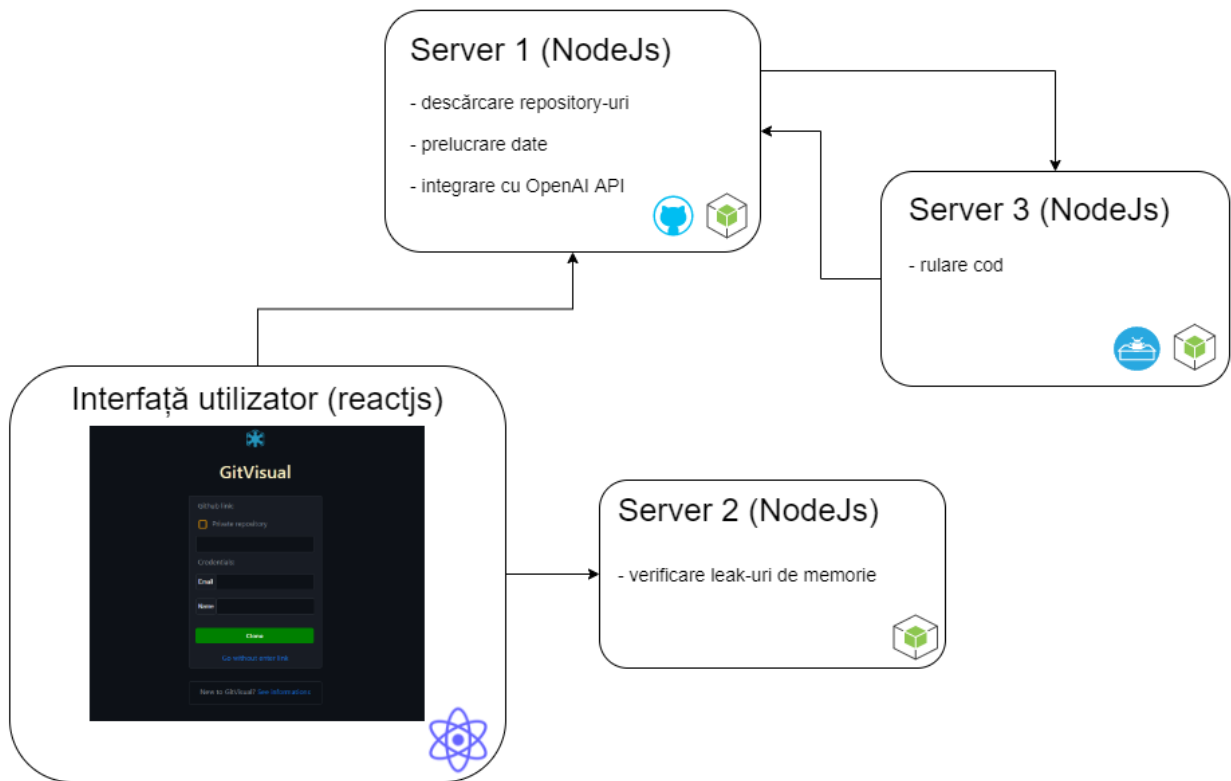


Fig. 4-2 Arhitectura sistemului

Interfața utilizatorului este realizată cu ajutorul framework-ului ReactJs care oferă o suită de pachete și librării care să faciliteze realizarea acesteia într-un mod cât mai rapid și estetic. Interfața este împărțită pe componente, astfel încât funcționalitățile aplicației să fie împărțite pe pagini și să nu depindă una de alta.

Server-ul 1 este server-ul principal al platformei care se ocupă cu majoritatea cererilor HTTP venite de la utilizator. Acesta implementează majoritatea funcționalităților, folosind diferite API-uri și biblioteci (vezi capitolul **Api-uri și Biblioteci folosite**) care să efectuează operațiile dorite. Acesta comunică cu Server 3, care se ocupă rularea codului, astfel: în momentul în care opțiunea pentru rularea codului este aleasă Server 1 primește codul pe care utilizatorul vrea să-l folosească, îl procesează, iar cu ajutorul unui script de python îl trimite mai departe către

Server 3. Acesta la rândul lui procesează cererea și trimite răspunsul către Server 1 care va afișa rezultatul utilizatorului.

Server-ul 2 se ocupă cu verificarea leak-urile de memorie. Acesta primește codul de C sau C++ și cu ajutorul instrumentului valgrind (vezi capitolul de **API-uri și Biblioteci folosite**) se analizează codul primit, rezultatul fiind transmis către interfață.

5. IMPLEMENTARE

Mediul folosit pentru dezvoltarea aplicației a fost sistemul de operare Windows 10 Home, testările intermediare și testarea finală s-au făcut pe același sistem.

Pentru implementarea platformei web a fost necesar crearea unei interfețe grafice prin care utilizatorul să poată efectua funcționalitățile pe care proiectul și le propune și crearea unui server care să facă posibil acest lucru.

Astfel pentru a crea interfața am folosit framework-ul ReactJs împreună cu multitudinea de biblioteci puse la dispoziție de acesta. Pentru a initializa un proiect în React este nevoie ca sistemul pe care se lucrează să aibă o versiune de Node $\geq 14.0.0$ și o versiune de npm ≥ 5.6 , comenzile necesare fiind:

```
npx create-react-app my-app  
cd my-app  
npm start
```

Pentru crearea server-ului, inițializarea și deschiderea acestuia s-au folosit comenzile:

```
mkdir server  
cd server npm init  
npm install express  
node server.js
```

Proiectul a fost structurat astfel:

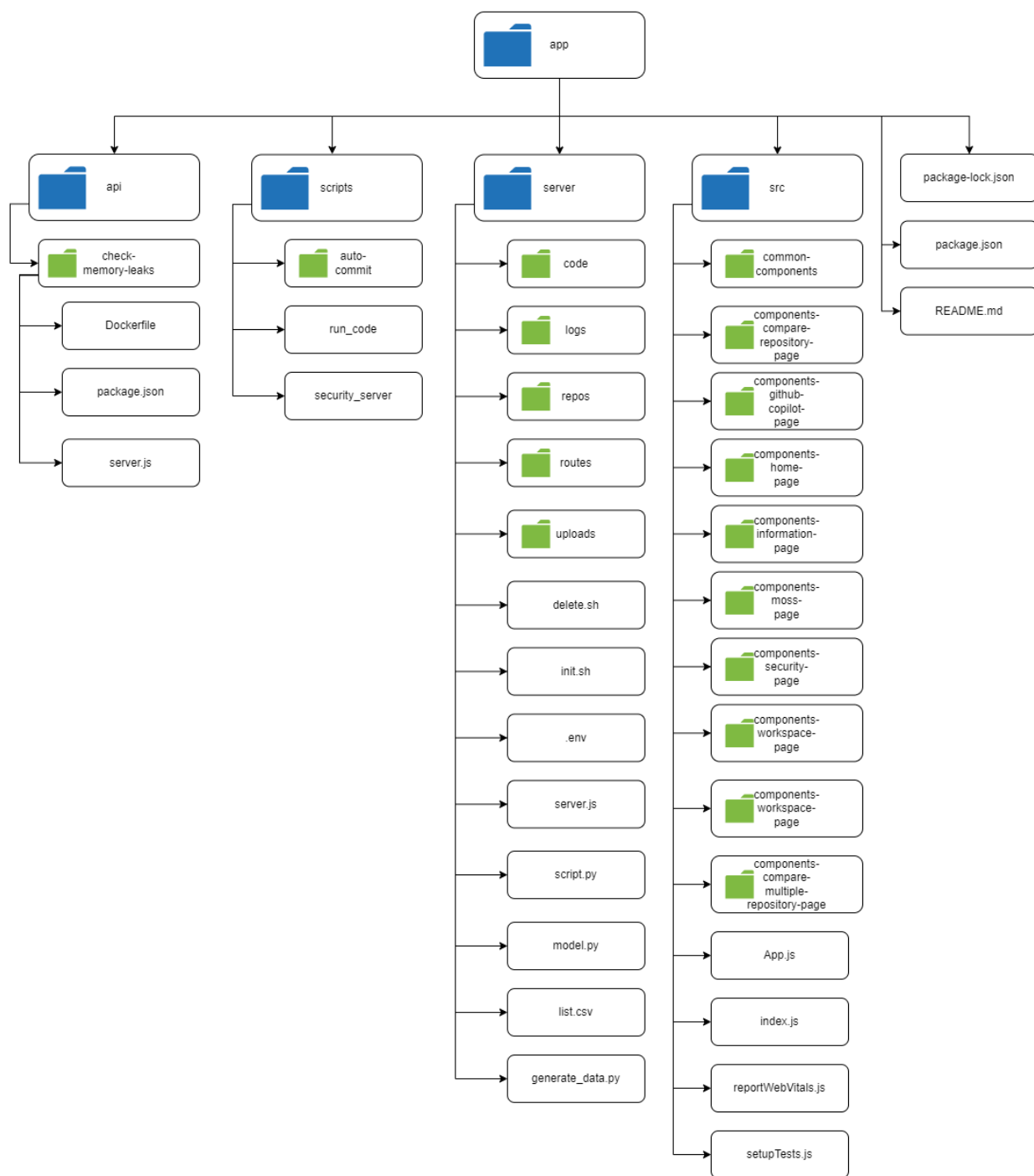


Fig. 5-1 Structura proiectului

Directorul **api/check-memory-leaks** conține fișiere de configurare pentru Server 2, menționat în subcapitolul **Definirea arhitecturii proiectului**. Fișierul *Dockerfile* reprezintă imaginea de docker pe care o aplic pe mașina virtuală de

Ubuntu. Acesta se folosește de *package.json* care conține dependențele necesare pentru ca serverul de NodeJs să poată funcționa și de fișierul *server.js* pentru a putea deschide server-ul care verifică leak-urile de memorie.

Fișierele *run_code.sh* și *security_server.sh* din directorul **scripts** conțin comenzi pentru configurarea celor două servere (Server 3, respectiv Server 2 din subcapitolul **Definirea arhitecturii proiectului**). Directorul **auto_commit** conține un fișier de *.gitignore* și un script de bash cu ajutorul căruia se poate realiza commit-uri cu o frecvență cât mai mare dintr-un repository local care poate fi inițializat sau nu.

Directorul **server** conține sursele necesare pentru inițializare Server 1 din subcapitolul **Definirea arhitecturii proiectului**. Scriptul *delete.sh* șterge datele, care se adună în folderele **code** și **repos** în momentul rulării codului, respectiv descărcării de repository-uri, la un timp dat. *Init.sh* la deschiderea server-ului creează folderele și fișierele necesare (repos, code, logs). Fișierul *.env* conține cheia necesară pentru conectarea la API-ul OpenAi. *Script.py* reprezintă legătura dintre Server 1 (serverul principal) și Server 3 (cel pentru rularea codului). În directorul **routes** sunt toate rutele pe care fișierul *server.js* le folosește pentru rezolvarea tuturor cererilor venite de la utilizator. Fișierul *list.csv* conține datele pe baza cărora am creat modelul. Acestea au fost generate folosind script-ul de python *generate_data.py*. *Model.py* reprezintă script-ul prin care am inițializat modelul și prezice eventualele intrări.

În rădăcina proiectului mai avem fișierul *package.json* care conține informații obișnuite despre proiect (nume, descriere, autor, script-uri și dependențe) și înregistrează metadata despre acesta, *package-lock.json* care conține exact arborele de care este nevoie pentru a instala dependențele necesare folosiri server-ului respectiv și fișierul *README.md* care oferă un rezumat al proiectului.

Directorul **src** conține fișierele sursă necesare pentru realizarea interfeței cu utilizatorul. Fiecare subdirector reprezintă câte o componentă din interfață. Fișierul *index.js* este locul în care se randează componenta principală (în cazul de față *App.js*). *ReportWebVitals.js* este folosit pentru a înregistra rezultatele în consolă sau pentru a le trimite la un punct final, iar *setupTests.js* este un fișier care importă librăria de test pentru React, jest-dom.

5.1. Afișarea evoluției codului unui repository

Având în vedere că platforma WEB trebuie să afișeze codul sursă a fișierelor descărcate de pe GitHub am folosit pachetul pus la dispoziție de npm monaco-editor, menționat și în capitolul **Api-uri și biblioteci folosite**.

Pentru a crea acea tranziție care permite vizualizarea evoluției codului am folosit un stack de div-uri, fiecare div având câte o componentă de tipul Editor, configurat cu parametrii selectați de mine și un index unic care va ajuta ulterior la afișarea lor. Pentru a putea controla ce se afișează la un moment dat este nevoie de un slider care să permită selectarea unei stări și butoane (de exemplu: înainte, înapoi, start, stop). Inițial toate div-urile din acel stack, cu excepția primului care este considerat pagină de start, nu sunt afișate (au proprietatea din css display: none). În momentul în care slider-ul este mutat pe o nouă valoare toate div-urile din acel stack primesc proprietatea din css display: none, cu excepția valorii selectate care rămâne cu proprietatea de display pe block (tot timpul se ține cont de acel index, care este trimis printr-un event handler).

Dacă utilizatorul apasă butonul de start funcția care tratează acest eveniment setează anumite variabile care vor determina ca funcția **useInterval()** să pornească (vezi **Anexa A** și **Anexa B**).

```
setIsRunning(!isRunning);
```

Această funcție are un timp care este predefinit de la început, dar care poate fi modificat de utilizator și realizează acea tranziție menționată în paragraful anterior.

Conținutul fiecărei componente Editor a fost obținut folosindu-mă de biblioteca NodeGit menționat și în capitolul **Api-uri și biblioteci folosite**. După introducerea link-ului dorit, server-ul descarcă repository-ul, creează o listă de commit-uri și o listă cu fișierele prezente la ultimul commit. În momentul în care utilizatorul selectează un fișier, având lista de commit-uri se parcurge fiecare index, se verifică dacă a existat acel fișier în momentul respectiv, se preia codul și se adaugă într-o nouă listă, care va fi trimisă ulterior ca răspuns către frontend. Astfel fiecare componentă Editor va conține codul fișierului de la un anumit commit din timpul dezvoltării aplicației.

Pachetul monaco-editor pune la dispoziție și un editor care afișează diferențele **MonacoDiffEditor** care compară două surse de cod. Având această caracteristică am creat o tranziție, la fel ca cea descrisă în paragrafele de mai sus, în care utilizatorul selectează o funcție, o bucată de cod și poate să vadă evoluția acesteia (dacă se modifică sau nu pe parcursul dezvoltării proiectului).

5.2. Identificarea introducerii unor bucăți mari de cod

Pentru a identifica introducerea unor bucăți mari de cod pe parcursul vizualizării, am calculat numărul de linii a fișierului la fiecare commit în parte,

astfel în momentul în care se realizează tranziția utilizatorul va primi o avertizare, în cazul în care au fost adăugate prea multe linii.

Această avertizare se bazează pe o medie calculată astfel: se face o sumă a diferențelor între fiecare commit (pe linii), care este împărțită la numărul total de commit-uri.

5.3. Verificarea leak-urile de memorie

Pentru a identifica posibile leak-uri de memorie am creat un API pe o mașină virtuală de Ubuntu versiunea 20.04 LTS. Dockerfile-ul folosit pentru a crea imaginea, pe baza căreia ulterior am creat container-ul de docker, conține pachetele și comenzile necesare funcționării server-ului de NodeJs și a imaginii de docker:

```
FROM alpine:latest
RUN apk add --no-cache nodejs npm
RUN apk add build-base
RUN apk add valgrind
WORKDIR /check-security-app
COPY . /check-security-app
RUN npm install
EXPOSE 3000
ENTRYPOINT ["node"]
CMD ["server.js"]
```

Comenzile folosite pentru a configura întregul sistem sunt următoarele se găsesc în **Anexa C**.

În momentul în care server-ul astfel deschis primește o cerere din partea utilizatorului, compilează codul primit folosind gcc, urmând ca apoi în cazul în care nu există nicio eroare cu ajutorul instrumentului valgrind să verificăm și să trimitem ca răspuns posibilele leak-uri de memorie (vezi **Anexa H**).

5.4. Crearea de sugestii în scop educativ pe baza codului scris

În implementarea acestei cerințe am folosit API-ul oferit de cei de la OpenAI (vezi subcapitolele **OpenAI** și **OpenAI API**). Folosind diferiți algoritmi puși la dispoziție am tratat diferite sugestii pe care utilizatorul le-ar putea solicita (calcularea complexității unei funcții, rezolvarea unui bug, explicarea unei funcții sau clase). În momentul în care utilizatorul alege o opțiune dorită, scrie codul pentru care are nevoie de o sugestie aceste va fi trimis către Server 1, care la rândul

NECLASIFICAT

lui va trimite o cerere către API-ul menționat în paragraful anterior. Această cerere trebuie configurată astfel:

```
const response = await openai.createCompletion("text-  
davinci-002", {  
  prompt: code,  
  temperature: 0,  
  max_tokens: 64,  
  top_p: 1.0,  
  frequency_penalty: 0.0,  
  presence_penalty: 0.0,  
  stop: ["\"\\\"\\\"\""],  
});
```

| Atribut | Explicație |
|-------------------|---|
| Prompt | reprezintă codul care va fi trimis către API-ul respectiv |
| Max-tokens | reprezintă numărul maxim de cuvinte care va fi trimis înapoi către utilizatorul (cu cât numărul este mai mare cu atât răspunsul va fi mai detaliat). Acesta nu poate depăși lungimea maximă admisă de către modelul ales (2048 sau 4096 dacă modelul este mai puternic) |
| Temperature | cu cât valoarea este mai mare modelul își asumă mai multe riscuri. Am ales valoarea 0 (eșantionare maximă) pentru că oferă un răspuns bine definit |
| Top_p | reprezintă eșantionarea de nucleu. Modelul ia în considerare rezultatele obținute cu masa de probabilitate definită. Dacă valoarea este 0.1 se iau token-urile cu probabilitatea 10% |
| Frequency_penalty | crește sau scade probabilitatea modelului de a repeta același rând sau aceleași cuvinte. Poate lua valori în intervalul [-2.0, 2.0]. În cazul valorilor pozitive scade, iar a celor negative crește |
| Presence_penalty | Poate lua valori în intervalul [-2.0, 2.0]. Crește probabilitatea modelului de a genera noi idei despre opțiunea aleasă. Valorile pozitive penalizează noile cuvinte dacă acestea apar deja |
| Stop | API-ul nu va mai genera alte simboluri, în funcție de secvența pe care o setăm (maxim 4) |

Tabel 1 Structura unei cereri HTTP pentru OpenAI API

NECLASIFICAT

Pentru a obține un rezultat cât mai corect trebuie modificată valoarea variabilei *temperature* sau *top_p*, nu ambele.

5.5. Detecția similarităților între fișiere

Pentru a verifica existența unor similarități între fișierele unui repository, inițial extrag toate commit-urile din repository-ul respectiv. Astfel utilizatorul are posibilitatea de a alege momentul la care dorește să verifice eventualele nereguli. Dacă acel moment este selectat acesta are posibilitatea să aleagă un fișier din cadrul proiectului care va fi comparat cu toate fișierele din repository-ul respectiv.

Astfel această comparație se bazează pe calcularea liniilor comune, adăugate și șterse, folosind librăria *git-diff*. Pe baza valorilor obținute se calculează un procent de similaritate.

5.6. Compararea a două repository-uri

Pentru a putea compara două repository-uri și a vedea progresul în paralel este necesară descărcarea acestora și afișarea commit-urilor fiecărui repository.

Cu ajutorul unui editor care afișează diferențele între două surse de cod în paralel, utilizatorul poate să compare fișierele dintre doua commit-uri, acestea fiind descărcate la momentul selecției.

Se calculează o serie de statistici pe baza celor doua repository-uri:

- Numărul de commit-uri realizate: la momentul descărcării repository-ului, se realizează o listă, se parcurge fiecare commit și se calculează frecvența fiecărei date.
- Limbajele folosite: se ia extensia fiecărui fișier, se creează o listă a limbajelor folosite și având mărimea fișierului putem crea o proporție între acestea.
- Autorii care au contribuit la dezvoltarea proiectului: se parcurg toate commit-urile și se extrage autorul fiecăruia, calculându-se o frecvență a acestora.
- Liniile adăugate la fiecare commit pentru un fișier selectat: se parcurge fiecare commit astfel încât să pot extrage codul pentru fișierul selectat din fiecare stare și data la care s-a făcut acea schimbare. Se calculează numărul de linii la fiecare moment și se creează un obiect de tip JSON care conține un vector cu data și numărul de linii la acel moment care va fi transmis și afișat într-un grafic utilizatorului.

5.7. Executarea codului

Pentru a permite utilizatorului să ruleze codul dorit am creat un API pe o mașină virtuală de Ubuntu versiunea 20.04 LTS. Am folosit o imagine de docker care îmi permite să rulez codul primit de la utilizatorul într-un mediu sigur, fără să afecteze mașina virtuală în cazul unor insecurități în cod (vezi subcapitolul **Sandbox**). Fiecare limbaj folosit este o imagine de docker descărcată.

Comenzile folosite pentru a configura întregul sistem se găsesc în **Anexa D**.

După configurarea sistemului server-ul este deschis la adresa ip a mașinii virtuale pe portul ales de mine (în cazul de față 8088).

Utilizatorul trimite codul pe care dorește să-l ruleze către Server 1 (vezi subcapitolul **Definirea arhitecturii**), acesta salvează codul într-un fișier pe care ulterior îl trimite către un script de python. Acest script citește conținutul fișierului primit, escapează caracterele (“\n”, “\t”, “\”) și construiește cererea care va fi trimisă către server-ul configurat mai sus. Astfel acesta rulează codul primit și trimite un răspuns către Server 1, care ulterior va fi afișat utilizatorului.

Cererea care va fi trimisă către server va trebuie configurată astfel:

```
headers = {
    'X-Access-Token': 'my-token',
    'Content-type': 'application/json',
}

image = language

data = '{"image": "glot/' + image + ':latest",
"payload": {"language": "'" + language + "'", "files":
[{"name": "main." + ext + "\", "content": '
data = data + "\"" + contents + "\"}]}}'"
requests.post('http://20.113.87.37:8088/run',
headers=headers, data=data)
```

Unde X-Access-Token va trebui să primească valoarea pe care am ales-o în momentul în care am creat container-ul și am deschis server-ul.

```
--env "API_ACCESS_TOKEN=my-token"
```

Image reprezintă imaginea pentru limbajul pe care trebuie să-l folosim pentru rularea codului.

5.8. Gestionarea unui repository privat

În cazul repository-urilor publice, clonarea acestora de către diferite persoane se poate face cu credențiale predifinite în mod implicit. Pentru a putea descărca repository-urile private trebuie oferite niște credențiale de către utilizator. Totodată acesta trebuie să fie adăugat ca și contribuitor la acel repository.

Funcția `clone()` din librărie de `nodegit` ne oferă această posibilitate, astfel se pot seta niște opțiuni înainte de clonare.

```
cloneOpts = {
  fetchOpts: {
    callbacks: {
      credentials: () =>
Git.Cred.userpassPlaintextNew(userName, token),
      transferProgress: (info) => console.log(info)
    }
  }
}
```

| Atribut | Conținut |
|----------|--|
| Username | este o valoare aleasă de utilizator, poate să fie chiar cel de pe GitHub sau orice altă valoare |
| Token | este o valoare generată de către contul de Github, secretă care trebuie știută doar de către proprietarul contului |

Tabel 2 Parametrii necesari pentru descărcarea unui repository privat

5.9. Realizarea opțiunilor pentru modificarea codului

5.9.1. Opțiunea de commit

Pentru realizarea acestei opțiuni se preia codul curent din editorul afișat utilizatorului. Adăugând la acesta datele: (adresa de email, numele de utilizator), branch-ul pe care se dorește a fi aduse modificările, fișierul care trebuie modificat și mesajul commit-ului.

Pe partea de server se deschide locația repository-ului, se preia index-ul acestuia (`repo.index()`), se deschide fișierul dorit și se scrie noul cod pe care utilizatorul l-a trimis. Cu index-ul obținut mai devreme se adăugă fișierul pentru a fi la zi cu noile modificări (echivalentul comenzii `git add .`). Având fișierul adăugat configurăm commit-ul cu datele oferite de utilizator (email, nume și mesaj), iar cu funcția `commit()` realizăm operația (vezi **Anexa E**).

Pentru opțiunea de auto-commit, care permite utilizatorului să facă mai multe commit-uri consecutive, se setează un interval, la un timp predifinit (4 secunde) care conține o funcție asincronă (pentru ca un commit să nu depindă de altul), unde se preiau datele necesare pentru configurarea commit-ului și se trimit către Server 1 (gestionarea cererii se face la fel ca în paragraful anterior).

Funcția care realizează acest lucru:

```
let func = setInterval(async () => {
  if (event.target.checked == true) {
    let file_code = editorRef.current;
    axios
      .post("http://localhost:8080/workspace/commit", {
        message: message,
        branch: currentBranch,
        directory: localRepo,
        file_name: commitsCode[0].file,
        new_code: file_code
      })
      .then((response) => {
        if (response.data === "Error") {
          alertProperties(true, "error", "Error", "Commit
can't be made!");
        }
      });
  }
  else {
    clearInterval(func);
  }, 4000);
```

5.9.2. Opțiunea de push

Pentru a realiza opțiunea de push datele necesare sunt: email-ul și numele utilizatorului, branch-ul și link-ul repository-ului. Am configurat operația cu datele primite:

```
simpleGit(buffer).add('./*')
  .addConfig('user.email', userEmail)
  .addConfig('user.name', userName)
  .push(link, branch)
  .then(res.json("Done"));
```

5.10. Extragerea caracteristicilor unui set de repository-uri

Pentru a realiza acest lucru este nevoie de un set de repository-uri (în cazul platformei realizate, un fișier .txt care are pe fiecare linie un link).

Am realizat o funcție asincronă care ia fiecare link din lista creată la momentul încărcării fișierului, descarcă repository-ul și parcurge fiecare commit în parte.

În momentul parcurgerii se calculează numărul total de commit-uri, în fiecare lună câte commit-uri au fost realizate, astfel se realizează și o frecvență a commit-urile pe fiecare lună în care s-a contribuit la proiect și câte schimbări au fost realizate în total pe parcursul dezvoltării aplicației.

Pentru a calcula schimbările aduse, pentru fiecare commit am calculat diferența între el și commit-ul părinte. Fiecare diferență este împărțită în pachete (fiecare pachet reprezentând un fișier) și acestea la rândul lor sunt împărțite în bucăți de cod care reprezintă rezultatul diferenței (vezi **Anexa F**).

Astfel cu rezultatul obținut se pot calcula schimbările aduse pe fiecare commit și în fiecare lună (vezi **Anexa I**).

5.11. Creare script pentru auto-commit

Pentru a realiza o serie de teste, în ideea de a vedea cum evoluează codul, este necesar crearea unui script care să facă commit la un interval de timp. Această opțiune este și integrată în platforma web, dar acest script a fost mai ușor de folosit pentru crearea testelor (vezi **Anexa G**).

Script-ul se execută având ca parametrii link-ul de GitHub pe care-l dorim, numele de utilizator și adresa de email.

Se inițializează repository-ul în cazul în care directorul în care se apelează script-ul este gol și se adaugă de la distanță originea pe care o dorim (link-ul de GitHub dat ca parametru).

Se creează un fișier de .gitignore, pe baza unui template pre-setat, unde sunt adăugate fișierele care nu trebuie să fie adăugate.

Se configurează parametrii care vor fi folosiți pentru operațiile de commit și push, adică numele de utilizator și adresa de email dați ca parametrii.

Pentru a face commit la un interval de timp am folosit o structură de tip while care are o valoare pre-setată la care se va repeta, unde se realizează toate comenzile necesare (*git add*, *git commit* și *git push*).

5.12. Creare script pentru ștergerea fișierelor temporare

Pentru a scăpa de fișierele temporare (repository-urile care se descarcă, fișierele care se încarcă și sursele de cod care vor fi trimise) a fost necesar crearea unui script care să permită acest lucru.

Acesta este apelat la un interval de timp folosind librăria *node-cron*, care ne permite să setăm timpul la care să executăm anumite comenzi. Pentru a putea apela scriptul am folosit librăria *child_process*.

```
cron.schedule('59 23 * * *', function () {
  const yoursript = exec('sh ./server/delete.sh',
    (error, stdout, stderr) => {
      console.log(stdout);
      console.log(stderr);
      if (error !== null) {
        console.log(`exec error: ${error}`);
      }
    });
});
```

Script-ul care se apelează șterge fișierele din directoarele (repos, code și uploads – vezi figura (trebuie pus numele)) și notează timpul la care se execută curățarea (pentru a verifica dacă ștergerea se execută conform timpului setat).

```
#!/bin/bash
echo "Start delete..."
timestamp=`date +%Y-%m-%d_%H:%M:%S`
echo "delete $timestamp" > ./logs/delete.log
rm -rf ./server/repos/*
rm -rf ./server/code/*
rm -rf ./server/uploads/*
echo "Delete finished..."
```

5.13. Realizarea unui sistem de evaluare

Pe baza caracteristicilor (vezi subcapitolul **Extragerea caracteristicilor unui set de repository-uri**) am creat un sistem care să ofere o notă, în funcție de contribuția adusă produsului software. Pentru a putea face acest lucru posibil am avut nevoie de un set de date pe baza căruia trebuie creat un model. Am generat 20000 de intrări cu valori pentru fiecare caracteristică diferite ce vor fi salvate într-un fișier de tip .csv. Nota pe care o obține o intrare este calculată astfel:


```

max_commits = max_value_of_an_attribute(data, 'commits')
max_commits_per_month = max_value_of_an_attribute(data,
'commits_per_month')
max_changes = max_value_of_an_attribute(data, 'changes')
max_changes_per_commit = max_value_of_an_attribute(data,
'changes_per_commit')
max_changes_per_month = max_value_of_an_attribute(data,
'changes_per_month')

for item in data:
    grade_1 = (item['commits'] / max_commits) * 10
    grade_2 = (item['commits_per_month'] /
max_commits_per_month) * 10
    grade_3 = (item['changes'] / max_changes) * 10
    grade_4 = (item['changes_per_commit'] /
max_changes_per_commit) * 10
    grade_5 = (item['changes_per_month'] /
max_changes_per_month) * 10
    item['grade'] = round((grade_1 + grade_2 + grade_3 +
grade_4 + grade_5)/5, 2)

```

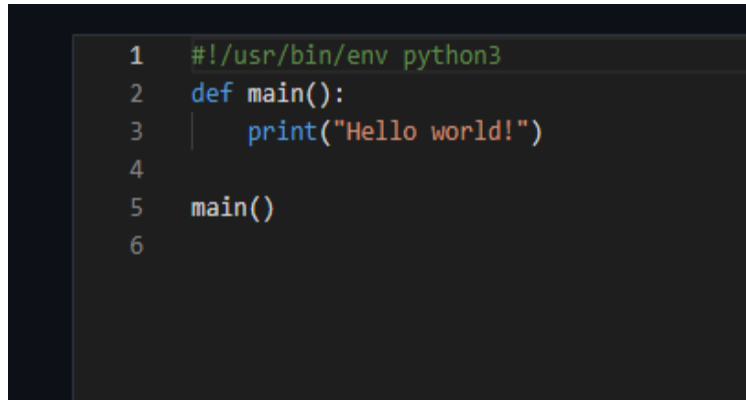
Am folosit funcțiile puse la dispoziție de biblioteca pandas și sklearn (vezi subcapitolul **API-uri și biblioteci folosite**) pentru a crea modelul și prezice o notă pentru eventualele intrări.

Am împărțit setul de date în train și test folosind funcția *train_test_split()* pusă la dispoziție de sklearn. Am separat atributul țintă de attributele predictive. Modelul creat este bazat pe regresie liniară, folosind funcția *LinearRegression()* din aceeași bibliotecă.

Cererea este interpretată de către Server 1 care apelează script-ul model.py (vezi **Anexa J**) pentru fiecare intrare pe care o primește de la utilizator (acesta având opțiunea să evalueze mai multe repository-uri sau doar unul singur, vezi **Anexa K**). Totodată aceste are și opțiunea sa reantreneze modelul, astfel datele pe care le primește după evaluare le poate introduce în fișierul .csv pe baza căruia se generează modelul.

6. REZULTATE TESTARE

Pentru a verifica dacă afișarea evoluției codului extrage datele corecte din repository-ul de GitHub vom introduce un link și vom compara cu datele pe care GitHub-ul ni-i le oferă. Astfel vom selecta un fișier din cadrul proiectului *file.py* și vom compara primul commit pentru acesta.

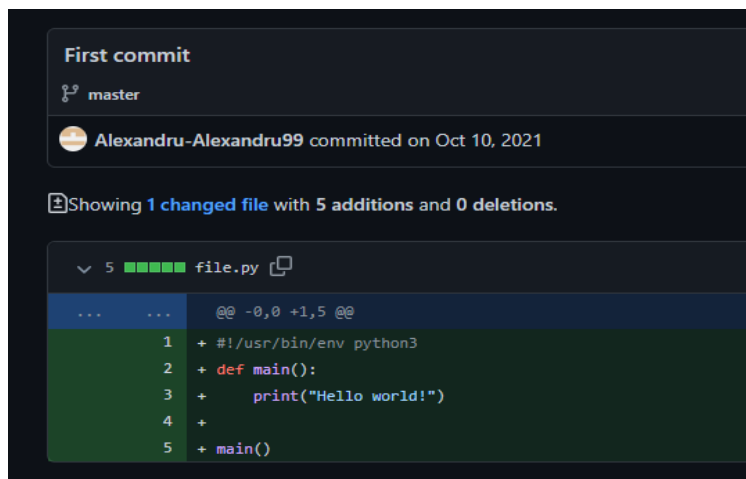


```

1  #!/usr/bin/env python3
2  def main():
3      print("Hello world!")
4
5  main()
6

```

Fig. 6-1 Rezultatul platformei



First commit
master
Alexandru-Alexandru99 committed on Oct 10, 2021
Showing 1 changed file with 5 additions and 0 deletions.

5 file.py

```

...  ...  @@ -0,0 +1,5 @@
1  + #!/usr/bin/env python3
2  + def main():
3  +     print("Hello world!")
4  +
5  + main()

```

Fig. 6-2 Rezultatul de pe Github

După cum se observă identificarea este corectă, comparând rezultatele obținute ele sunt identice.

Pe parcursul dezvoltării proiectului s-au efectuat teste unitare la nivel de funcție pentru asigurarea comportamentului corect al acestora.

Pentru testarea de acceptare s-au executat o serie de teste care să determine dacă rezultatul pe care funcționalitățile ar trebui să le furnizeze sunt corecte. Astfel m-am asigurat că arhitectura sistemului este realizată, fiecare componentă și-a îndeplinit criteriile de calitate descrise, fluxurile de date dintre componente

respectă cerințele generale ale sistemului și sunt implementate conform standardelor în vigoare.

7. INTERACȚIUNEA UTILIZATOR-APLICAȚIE

Pentru utilizarea aplicației utilizatorul trebuie să acceseze link-ul către platforma web.

Acesta va fi întâmpinat de pagina de start a platformei unde trebuie să completeze niște informații înainte de a merge mai departe. Dacă dorește vizualizarea unui repository, utilizatorul trebuie să introducă link-ul către acesta, să-și seteze credențialele (nume utilizator și adresa de email), iar în cazul în care repository-ul este privat, trebuie să furnizeze niște informații în plus. Totodată dacă utilizatorul nu dorește să introducă acum link-ul, are posibilitatea ulterior, alegând opțiune *Go without enter link*.

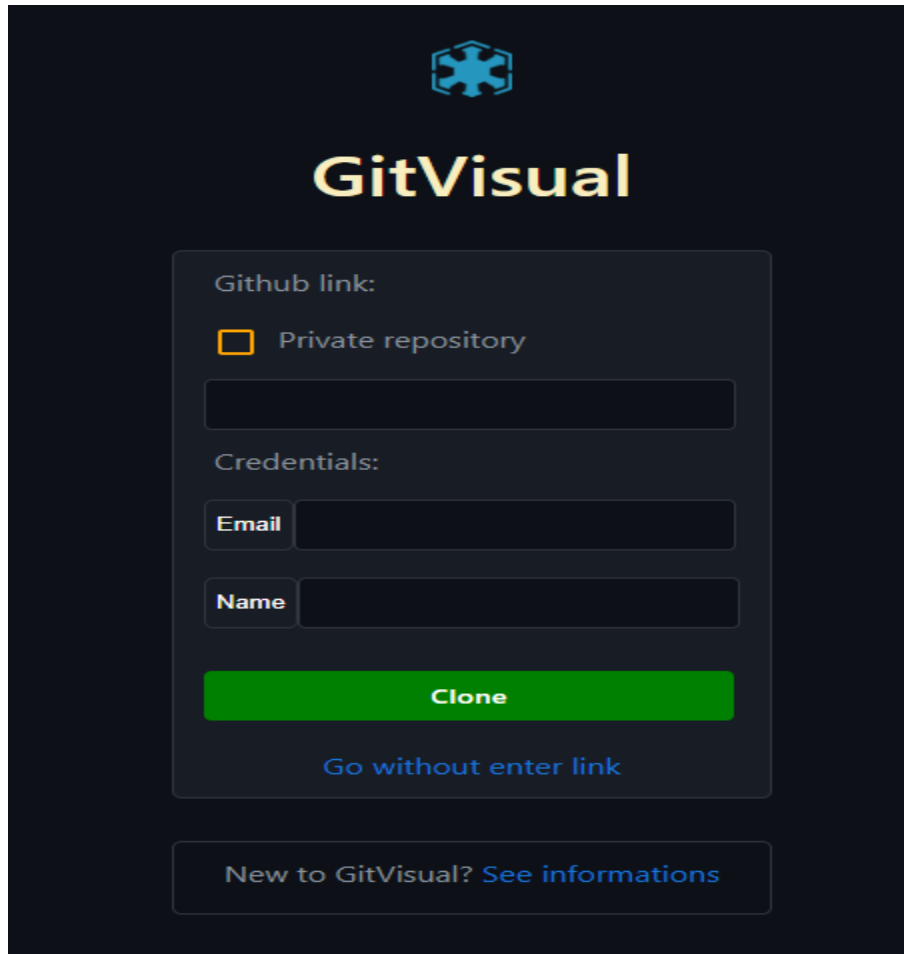


Fig. 7-1 Pagina de start a platformei

După apăsarea butonului *Clone* utilizatorul va fi întâmpinat de următoare pagină, unde va putea selecta ce fișier dorește să vizualizeze din cadrul proiectului.

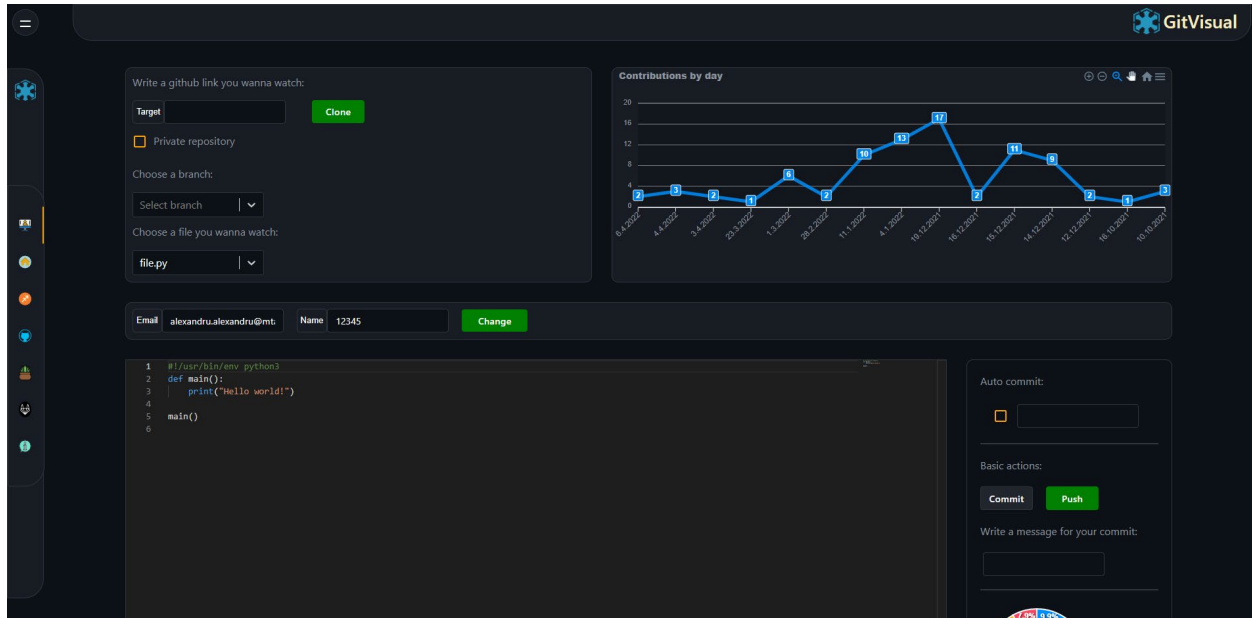


Fig. 7-2 Pagina pentru vizualizarea evoluției codului

Totodată în această pagină utilizatorul poate accesa celelalte funcționalități puse la dispoziție de către platformă. Prin apăsarea pictogramelor din meniul din stânga paginii acesta poate naviga de la o funcționalitate la alta.

7.1. Diagrame UML

Pentru o mai bună înțelegere a fluxului aplicațiilor s-au creat diagrame UML care să prezinte pe scurt și într-un limbaj mai puțin tehnic stările aplicației, modul de funcționare și unele scenarii.

Pentru câteva funcționalități importante s-a creat o diagramă de activități care să prezinte toate stările prin care trece aceasta:

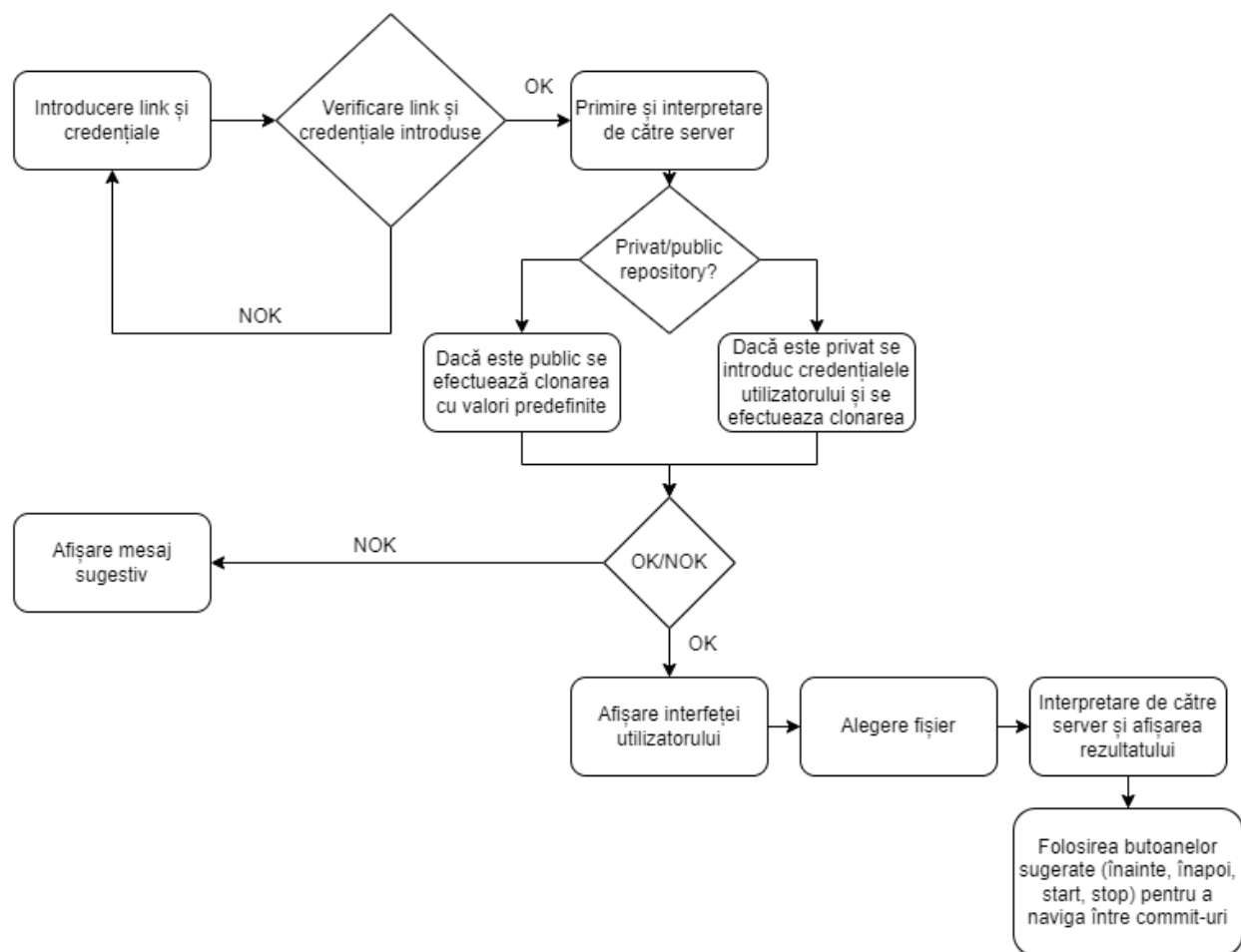


Fig. 7-3 Diagrama de activități pentru vizualizare evoluție cod

Diagrama de mai sus (Fig. 8-1) reprezintă stările prin care trece sistemul pentru a furniza utilizatorului informațiile dorite. Acesta trebuie să introducă link-ul dorit și credențialele sale. Server-ul interpretează cererea și verifică tipul repository-ul. Pentru fiecare efectuează operații diferite. Se verifică dacă descărcarea acestuia a avut loc fără a întâmpina nicio eroare. În cazul **OK** se afișează interfața utilizatorului de unde acesta poate să aleagă ce fișier să urmărească. Server-ul interpretează cererea primită de la utilizator și furnizează rezultatul dorit. În final utilizatorul poate să vadă toate versiunile fișierului folosind butoanele specifice (înainte, înapoi, stop și start).

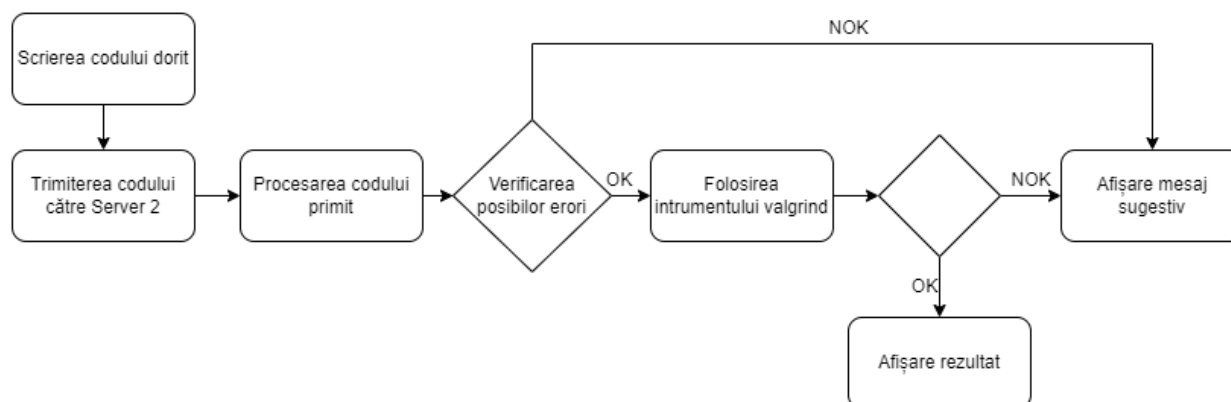


Fig. 7-4 Diagrama de activități verificare leak-uri de memorie

Diagrama de activități (Fig. 8-2) reprezintă stările prin care trece sistemul pentru a verifica leak-urile de memorie a unui fragment de cod. Astfel după ce utilizatorul scrie codul dorit acesta este trimis către Server 2 (vezi subcapitolul **Definirea arhitecturii**) și procesat. Se verifică posibilele erori, în cazul în care nu se întâmpină probleme se folosește comanda valgrind și se afișează rezultatul în interfața utilizatorului. Dacă pe parcursul procesării cererii se întâmpină probleme aceste sunt semnalate utilizatorului prin mesaje sugestive.

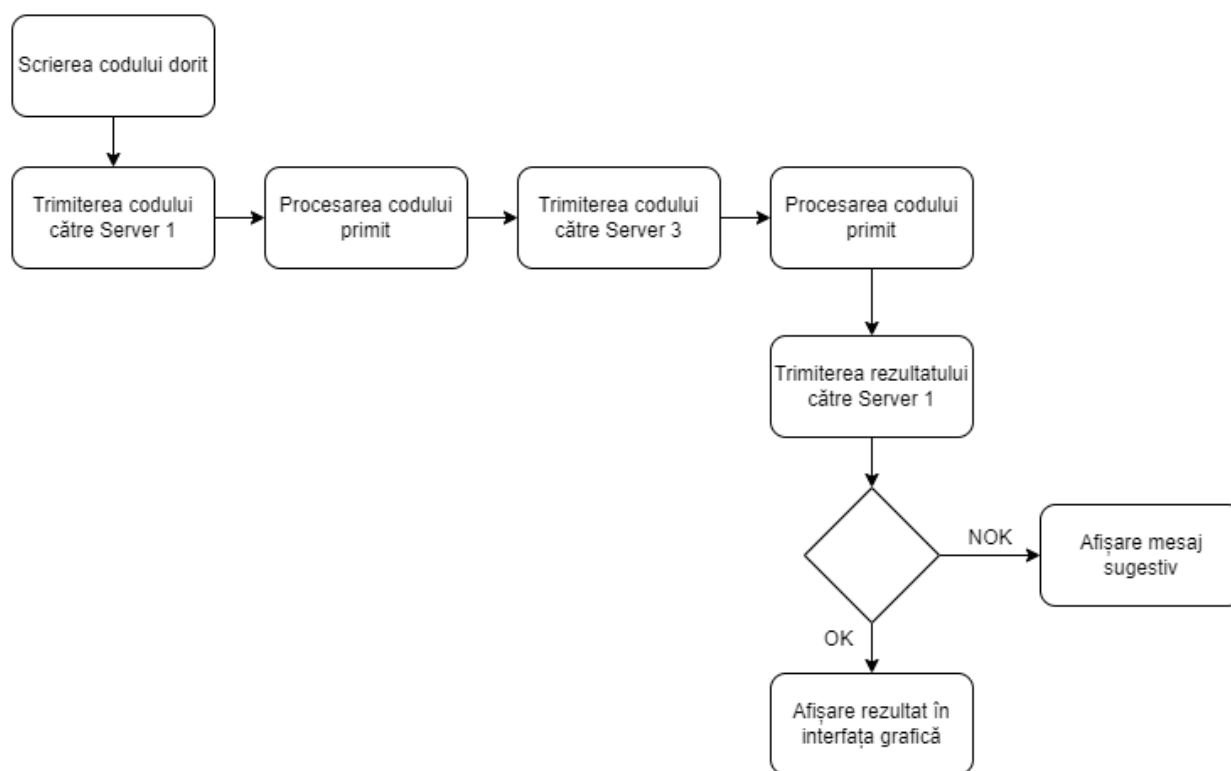


Fig. 7-5 Diagrama de activități pentru rularea codului

Diagrama de activități (Fig. 8-3) reprezintă stările prin care trece sistemul pentru a rula o secvență de cod pe care o utilizatorul o furnizează. Se trimite codul către Server 1 acesta procesează cererea (scrie codul primit într-un fișier pe care-l trimite spre un script de python unde se construiește cererea către Server 3). Acesta procesează codul primit și trimite rezultatul către Server 1 care apoi va fi afișat utilizatorului în interfață grafică. Dacă pe parcursul procesării cererii se întâmpină probleme aceste sunt semnalate utilizatorului prin mesaje sugestive.

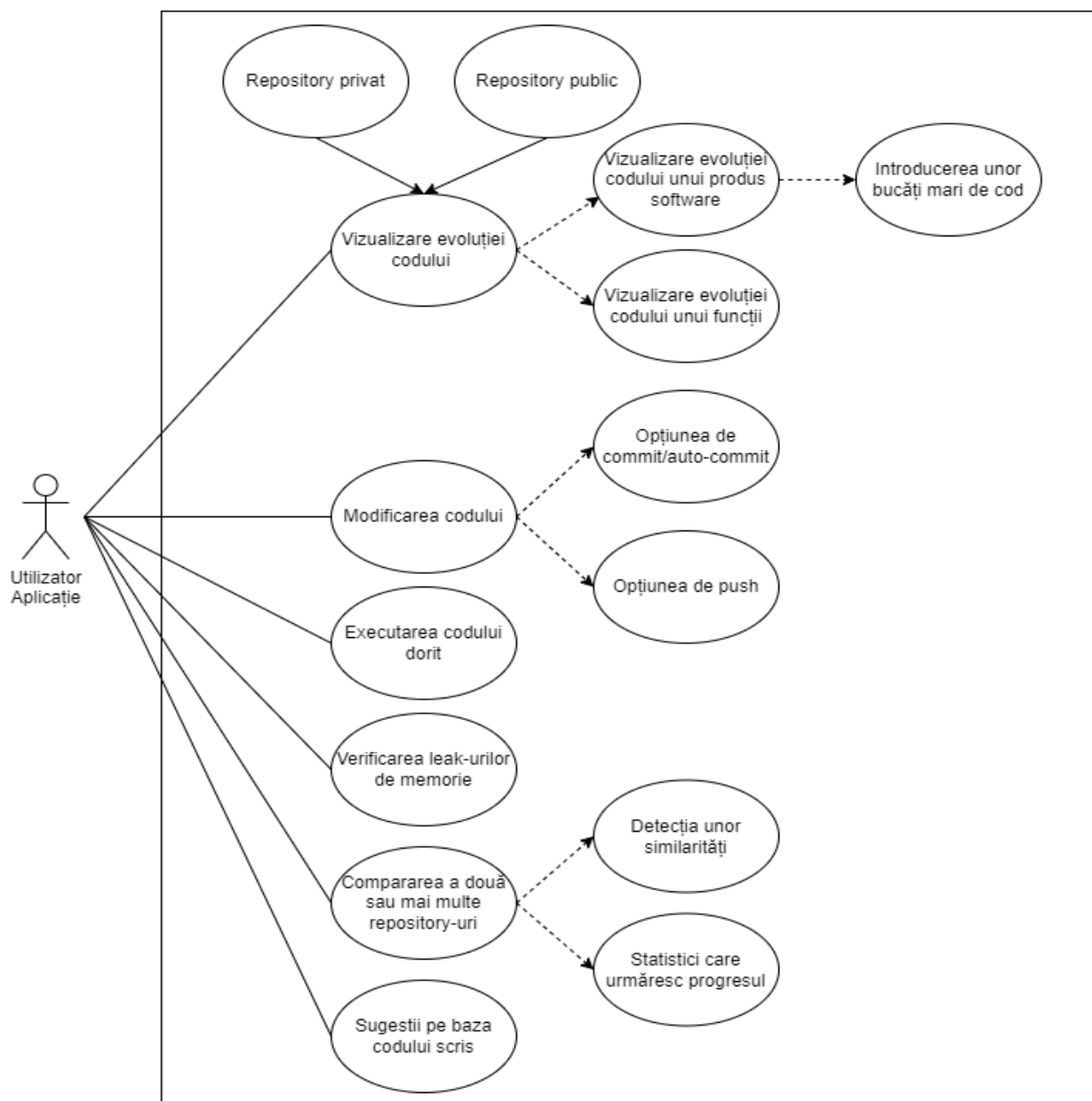


Fig. 7-6 Diagrama cazurilor de utilizare

Diagrama cazurilor de utilizare (Fig. 8-6) prezintă o mai bună ilustrare a modului de interacțiune cu aplicația, arătând fiecare funcționalitate pe care platforma web o pune la dispoziție.

8. CONCLUZII ȘI DIRECȚII VIITOARE DE CERCETARE

8.1. Probleme întâmpinate în implementare

În timpul implementării proiectului au fost întâmpinate unele probleme.

În cadrul funcționalității pentru afișarea evoluției codului a apărut o problemă a timpului de extragere a conținutului fișierelor de la fiecare commit. Soluția pe care am implementat-o inițial descărca repository-ul dorit și extrăgea toate datele pentru fiecare fișier de la fiecare commit în același timp. Pentru proiecte care conțineau puține fișiere, sau conținutul acestora era limitat la un număr mic de linii nu apărea această problemă, în schimb pentru proiecte foarte mari, sau cu fișiere foarte multe timpul de așteptare ajungea la ordinul sutelor de secunde. Soluția care a scăzut acel timp de așteptare a fost mult mai modulară, astfel încât: inițial se descarcă repository-ul dorit și se extrage hash-ul pentru fiecare commit și fișierele de la ultimul commit; având aceste două liste, utilizatorul în momentul în care alege un fișier abia atunci se extrage conținutul fișierului de la fiecare commit. Astfel server-ul ocupându-se de un singur fișier la un moment dat timpul de execuție scade foarte mult.

O altă problemă întâmpinată a fost la extragerea modificărilor aduse între commit-uri. Rezultatul pe care-l furniza funcția după apelarea ei era diferit de fiecare dată, datorită funcțiilor din librăria NodeGit care returnau promisiuni. Astfel după o mai adâncă căutare și investigare a motodelor apelate s-a ajuns la concluzia că încadrarea acestora în funcții asincrone (async) și apelarea lor în structuri await.Promise() va furniza un rezultat corect și același de fiecare dată.

8.2. Rezultate obținute

Platforma web realizată funcționează în parametri normali și implementează majoritatea funcțiilor propuse. Deși performanțele platformei ar putea fi îmbunătățite, aceasta își îndeplinește scopul și reprezintă atât o soluție pentru vizualizarea evoluției codului cât și un punct de plecare pentru dezvoltări ulterioare.

8.3. Îmbunătățiri ale proiectului

Proiectul ar putea beneficia de anumite îmbunătățiri. Ar fi utilă implementarea unui sistem de log-are care să despartă utilizatorul elev de utilizatorul profesor. Astfel funcționalitățile pe care platforma le oferă vor fi și ele împărțite oferind o mai bună organizare și structurare a aplicației.

De asemenea detectarea similarităților între fișiere ar putea fi abordată altfel, oferind mai multe informații clare și sigure. O analiză de sintaxă ar fi o soluție pentru această problemă.

Totodată verificarea securității codului ar putea beneficia de o implementare proprie, fișierele putând fi analizate independent, astfel se oferă un grad de siguranță mai ridicat.

8.4. Funcționalități suplimentare

Deși proiectul oferă funcționalitățile descrise în cerințe acesta ar putea fi îmbunătățit prin adăugarea de funcționalități suplimentare.

Având în vedere că se dorește îmbunătățirea stilului de a scrie cod și realizarea unor programe cât mai rapide, adăugarea unor funcționalități care să sugereze aceste aspecte vor aduce un plus proiectului.

9. BIBLIOGRAFIE

- [1] W. Yang, „Identifying Syntactic differences Between Two Programs,” în *Software - Practice and Experience*, 1991.
- [2] B. S. Scott Chacon, *Pro Git*, APRESS, 2014.
- [3] A. T. J. J. v. W. Lucian Voinea, „CVSscan: Visualization of Code Evolution,” în *Proceedings of the 2005 ACM Symposium on Software Visualization*, Eindhoven, 2005.
- [4] D. J. a. D. A. Ladd., „Semantic diff: A tool for summarizing the effects of modifications,” în *Proceedings of the IEEE International Conference on Software Maintenance*, 1994.
- [5] J. S. F. M. H. Iulian Neamtiu, „Understanding Source Code Evolution Using Abstract Syntax Tree Matching,” în *Proceedings of the 2005 International Workshop on Mining Software Repositories*, Maryland, 2005.
- [6] IBM, „IBM,” [Interactiv]. Available: <https://www.ibm.com/topics/linear-regression#:~:text=Resources-What%20is%20linear%20regression%3F,is%20called%20the%20independent%20variable..> [Accesat 05 06 2022].
- [7] S. Horwitz, „Identifying the semantic and textual differences between two versions of a program,” în *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.
- [8] V. Developers. [Interactiv]. Available: <https://valgrind.org/info/about.html>. [Accesat 05 06 2022].
- [9] [Interactiv]. Available: <https://www.proofpoint.com/us/threat-reference/sandbox#:~:text=In%20the%20world%20of%20cybersecurity,URLs%20and%20observe%20its%20behavior..> [Accesat 29 04 2022].
- [10] [Interactiv]. Available: <https://paperswithcode.com/method/gpt-3>. [Accesat 01 06 2022].

10. ANEXE

10.1. Anexa A

```

<div className="view-content">
  <Stack spacing={0}>
    <div className="start-item-content" key="start"
id={idBuffer + 0}>
      <div className="loading-container">
        <img className='image' style={{ width:
"64px" }} src={loading} alt="loading" />
      </div>
    </div>
    {commitsCode ? commitsCode[0].codes.map(({ order,
code, status }, index) => (
      <div className="item-content"
key={commitsCode[0].file + index} id={idBuffer + (index +
1)}>
        <Editor
          className={commitsCode[0].file + (index
+ 1)}
          height="60vh"
          theme="vs-dark"
          defaultLanguage={language}
          defaultValue={code}
          onMount={handleEditorDidMount}
          onChange={handleEditorChange}
        />
      </div>
    )) : <></> }
  </Stack>
  {isOpen && <Popup
    title="{title}"
    handleClose={togglePopupChange}
  /> }
</div>

```

10.2. Anexa B

```

useInterval(() => {
    let newValue = currentDivEditor + 1;
    if ((newValue <= number)) {
        if (commitsCode[0].stats[newValue] === 0) {
            setCodeStatus(false);
        } else {
            setCodeStatus(true);
        }
        for (var i = 0; i <= number; i++) {
            document.getElementById("index" +
i).style.display = "none";
        }
        document.getElementById("index" +
newValue).style.display = "block";
        setCurrentDivEditor(newValue);
        setSliderValue(newValue);
    }
    else {
        playButtonPressed(false);
        setIsRunning(false);
        setShowPlayButton(!showPlayButton);
    }
}, isRunning ? delay : null);

const handlePlay = () => {
    if (currentFile !== '') {
        console.log(currentDivEditor);
        playButtonPressed(!isRunning);
        onSelectItem(false);
        setIsRunning(!isRunning);
        setShowPlayButton(!showPlayButton);
    } else {
        alertProperties(true, "error", "Error", "Select a
file!");
    }
}

```

10.3. Anexa C

```
#!/bin/bash
#*  install docker
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-
keyring.gpg
echo \
    "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
#*  check docker installation
sudo docker run hello-world
#*  build image
sudo docker build -t memory-leak:latest .
#*  create container
sudo docker run --detach --restart=always --publish
3000:3000 memory-leak:latest
```

10.4. Anexa D

```
#!/bin/bash
#*  install docker
sudo apt-get update
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-
keyring.gpg

echo \
    "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

#*  check docker installation
sudo docker run hello-world

#*  install run image
sudo docker pull glot/docker-run:latest

#*  language images
sudo docker pull glot/python:latest
sudo docker pull glot/php:latest
sudo docker pull glot/clang:latest
sudo docker pull glot/javascript:latest
sudo docker pull glot/csharp:latest

#*  create container
sudo docker run --detach --restart=always --publish
8088:8088 --volume /var/run/docker.sock:/var/run/docker.sock
--env "API_ACCESS_TOKEN=my-token" glot/docker-run:latest
```

10.5. Anexa E

```

Git.Repository.open(buffer)
  .then(function (repoResult) {
    return repoResult.index();
  })
  .then(function (indexResult) {
    index = indexResult;
    buffer = buffer + "/" + file;
    fs.writeFile(buffer, new_code, function (err) {
      if (err) {
        console.log("Error on write file
commit...");
        res.json("Error");
      }
    });
    try {
      index.addByPath(file);

      index.write();
    }
    catch(e) {
      console.log(e);
    }
    return index.writeTree();
  })
  .then(async function (oidResult) {
    try {
      let workingDir = './repos/' + directory;
      statusSummary = await
simpleGit(workingDir).status();
      await simpleGit(workingDir).add('./*')
        .addConfig('user.email', userEmail)
        .addConfig('user.name', userName)
        .commit(message)
        .then(res.json("Done"));
    }
    catch (e) {
      res.json("Error");
    }
  });
});

```

10.6. Anexa F

```

async function my_get_diff(commit) {
  let lines_added_per_commit = [];
  await commit.getDiff(10000)
    .then(async (diffs) => {
      await Promise.all(
        diffs.map(async (diff) => {
          await diff.patches()
            .then(async (patches) => {
              await Promise.all(
                patches.map(async (patch) => {
                  await patch.hunks()
                    .then(async (hunks) =>
                      await Promise.all(
                        hunks.map(async (hunk) => {
                          const lines = await hunk.lines();
                          const l = lines.length;
                          const date = commit.date();
                          lines_added_per_commit.push({
                            l,
                            date
                          });
                        })
                      );
                    );
                })
              );
            });
          );
        })
      );
    })
    .catch(function (e) {
      console.log(e);
      res.json("Error");
    });
  return lines_added_per_commit;
}

```


10.7. Anexa G

```
#!/bin/bash
git_link=$1
username=$2
email=$3
#* Initializare repository
echo "Start repository initialization..."
FILE=.git
if [ -a "$FILE" ]; then
    echo "$FILE exists."
else
    git init
    git remote add origin $git_link
fi
echo "End repository initialization..."
echo "Start .gitignore initialization..."
touch .gitignore
cat "./gitignore_content.txt" > .gitignore
echo "End .gitignore initialization..."
echo "Start user configuration..."
git config --global user.name $username
git config --global user.email $email
echo "End user configuration..."
while true; do
    echo "Start commit..."
    # * Iau fisierele care trebuie adaugate pe github
    my_files=$(ls | grep "C")
    echo "Add files..."
    printf '%s\n' "${my_files[@]}"
    # * Adaug acele fisiere
    git add ${my_files[@]}
    echo "Commit message..."
    current_date=`date`
    git commit -m "${current_date}"
    echo "Push changes..."
    git push origin master
    echo "End commit..."
    sleep 25;
done
```

10.8. Anexa H

```

exec('touch ' + path,
  function (error, stdout, stderr) {
    if (error !== null) {
      console.log("Error on memory leaks check...");
      console.log('exec error: ' + error);
      res.json("Error");
    } else {
      let cmd = 'gcc -o ' + './code/' + name + '-std=c11 -Wall ' + path;
      exec(cmd,
        function (error_gcc, stdout_gcc, stderr_gcc) {
          if (error_gcc !== null) {
            console.log("Error on memory leaks check...");
            console.log('exec error: ' + error_gcc);
            res.json("Error");
          } else {
            exec('valgrind --leak-check=full ' + './code/' + name,
              function (error_valgrind, stdout_valgrind, stderr_valgrind) {
                console.log("End memory leaks check...");
                res.json(stderr_valgrind);
                if (error_valgrind !== null || error !== null || error_gcc !== null) {
                  console.log("Error on memory leaks check...");
                  console.log('exec error: ' + error_valgrind);
                  res.json("Error");
                }
              })
          }
        })
    }
  })
});

```

10.9. Anexa I

```

let table_data = [];
for (let i = 0; i < repos.length; i++) {
  let number_commits = 0;
  let number_commits_per_month = 0;
  let number_changes = 0;
  let number_changes_per_commit = 0;
  let number_changes_per_month = 0;
  let number_months = 0;
  for (let j = 0; j < response1.data.length; j++) {
    if (repos[i] === response1.data[j].repo) {
      number_commits = response1.data[j].counter;
      number_commits_per_month = number_commits /
response1.data[j].months_with_commits[0].length;
      number_months =
response1.data[j].months_with_commits[0].length;
    }
  }
  for (let j = 0; j < response2.data.length; j++) {
    if (repos[i] === response2.data[j].repo) {
      for (let q = 0; q <
response2.data[j].lines_all_commits.length; q++) {
        for (let k = 0; k <
response2.data[j].lines_all_commits[q].length; k++) {
          number_changes +=
response2.data[j].lines_all_commits[q][k].1;
        }
      }
      number_changes_per_commit = number_changes /
response2.data[j].lines_all_commits.length;
      number_changes_per_month = number_changes /
number_months;
    }
  }
  table_data.push(createData(repos[i], number_commits,
parseInt(number_commits_per_month), number_changes,
parseInt(number_changes_per_commit),
parseInt(number_changes_per_month)));
}
setRows(table_data);

```

10.10. Anexa J

```

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LinearRegression
import random
import sys

data = pd.read_csv('list.csv')

train_frame, test_frame = train_test_split(data,
test_size=0.2, random_state=42)

grade = train_frame['grade'].copy()

train_frame.drop('grade',axis='columns',inplace=True)

#! my data
# commits = random.randint(1,100)
# commits_per_month = (int)(commits/6)
# changes = random.randint(100,3000)
# changes_per_commit = (int)(changes / commits)
# changes_per_month = (int)(changes / 6)

commits = sys.argv[1]
commits_per_month = sys.argv[2]
changes = sys.argv[3]
changes_per_commit = sys.argv[4]
changes_per_month = sys.argv[5]
lst = [[commits, commits_per_month, changes,
changes_per_commit, changes_per_month]]
my_data = pd.DataFrame(lst)

model = LinearRegression()
model.fit(train_frame, grade)
print(model.predict(my_data))

```

10.11. Anexa K

```

function get_grade(repo) {
  return new Promise((resolve, reject) => {
    try {
      let options = {
        mode: 'text',
        pythonOptions: ['-u'],
        args: [repo['commits'],
repo['commits_per_month'], repo['changes'],
repo['changes_per_commit'], repo['changes_per_month']]
      };
      PythonShell.run('model.py', options, function
(err, result) {
        if (err) {
          res.json("Error");
        }
        resolve({reponame: repo['reponame'], grade:
result[0]}));
      });
    } catch (e) {
      res.json("Error");
    }
  });
}

router.post('/grades', async (req, res) => {
  const repos = req.body.repos;
  let grades = [];
  await Promise.all(repos.map(async function (repo) {
    grades.push(await get_grade(repo));
  }));
  await Promise.all(repos.map(async function (repo) {
    grades.map(grade => {
      if (grade['reponame'] === repo['reponame']) {
        repo['grade'] = grade['grade'].toString().substring(1,
grade['grade'].toString().length - 1);
      }
    });
  }));
  res.json(repos);
});

```