ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

бакалавра
(освітньо-кваліфікаційний рівень)

на тему *«Розробка веб-додатку аналізу даних про активність у системах контролю версій»*

Виконала:
здобувачка *4* року навчання
групи *4зПР*
напряму підготовки (спеціальності)
121-Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Єршова Христина Олексіївна
(підпис, прізвище та ініціали)
Керівник *Жарікова Марина Віталіївна*
(підпис, прізвище та ініціали)
Рецензент *к.т.н., доцент Григорова А.А.*
(прізвище та ініціали)
Нормоконтролер *Комісаров О. С.*
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРЗВТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *Програмна інженерія*
(шифр і назва)
Спеціальність *121-Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри *програмних засобів і технологій*
_____ О.Є. Огнєва
«___» _____ 2026 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Єршової Христини Олексіївни
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) *Розробка веб-додатку аналізу даних про активність у системах контролю версій*

керівник проєкту (роботи) *Жарікова Марина Віталіївна, д.т.н., професор*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «16» січня 2026 року №84-с

2. Строк подання здобувачем проєкту(роботи) *10.06.2026*

3. Вихідні дані до проєкту (роботи) *літературні та періодичні джерела, матеріали переддипломної практики*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз предметної галузі та існуючих рішень для оцінювання активності розробників у системах контролю версій; проєктування веб-додатку для аналізу даних про активність в системах контролю версій; розроблення та програмна реалізація веб-додатку для аналізу даних про активність в системах контролю версій; тестування та впровадження веб-додатку для аналізу даних про активність в системах контролю версій

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
8 таблиць, 38 рисунків

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *14.01.2026*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів проєкту (роботи)	Примітка
1	Отримання завдання	14.01.2026	Виконано
2	Підбір літератури	17.01.2026-21.01.2026	Виконано
3	Аналіз предметної області	22.01.2026-26.01.2026	Виконано
4	Розробка та обґрунтування завдання	27.01.2026-12.02.2026	Виконано
5	Розробка концептуальної моделі	13.02.2026-19.02.2026	Виконано
6	Моделювання та проєктування системи	20.02.2026-02.03.2026	Виконано
7	Розробка веб-додатку	03.03.2026-15.04.2026	Виконано
8	Тестування веб-додатку	16.04.2026-20.04.2026	Виконано
9	Оформлення пояснювальної записки	21.04.2026-02.05.2026	Виконано
10	Захист кваліфікаційної роботи	23.06.2026	Виконано

Здобувач _____ *Єршова Х.О.*
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ *Жарікова М.В.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра: 97 сторінок, 38 рисунків, 8 таблиць, 1 додаток, 22 джерела.

Мета роботи:

Розроблення веб-додатку аналізу даних про активність у системах контролю версій, який забезпечує підвищення ефективності моніторингу внеску розробників у командну розробку програмного забезпечення шляхом автоматичного збору та аналізу даних з репозиторіїв.

Об'єкт дослідження:

Процес збору та аналізу даних щодо активності учасників у системах контролю версій для оцінювання їхнього індивідуального внеску в командну розробку програмного забезпечення..

Предмет дослідження:

Методи, моделі та програмні засоби збирання, опрацювання та візуалізації статистики активності розробників на основі даних із систем контролю версій.

Інструменти та засоби розробки:

Для розробки веб-додатку було використано мову програмування Python, веб-фреймворк Flask, бібліотеки PyGitHub для взаємодії з GitHub API, Pandas для аналізу даних, Plotly для побудови інтерактивних графіків, HTML, CSS, JavaScript та Bootstrap 5 для реалізації клієнтської частини.

Результати роботи:

Створення веб-додатку DTracker, який може бути використаний тім-лідами, менеджерами проєктів, аналітиками та командами розробників для автоматизованого аналізу активності учасників програмних проєктів та підтримки процесів технічного управління.

Новизна рішення:

Розроблено веб-орієнтоване рішення, що поєднує можливості підключення до GitHub, інтерактивну візуалізацію активності учасників команди, автоматичне виявлення неактивних розробників і формування PDF-звітів в єдиному інтерфейсі.

Практична цінність:

Можливість застосування розробленого веб-додатку менеджерами проєктів, тім-лідами та технічними керівниками для оперативного оцінювання продуктивності команди, прийняття управлінських рішень на основі фактичних даних, а також для підвищення прозорості та відповідальності у процесі командної розробки.

КЛЮЧОВІ СЛОВА: *ВЕБ-ДОДАТОК, СИСТЕМА КОНТРОЛЮ ВЕРСІЙ, РЕПОЗИТОРІЙ, GIT, КОМАНДА РОЗРОБНИКІВ, АКТИВНІСТЬ, PYTHON, FLASK, DTRACKER, API, АНАЛІЗ ДАНИХ.*

АНОТАЦІЯ

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

Робота присвячена розробленню веб-додатку аналізу даних про активність у системах контролю версій.

У першому розділі проведено аналіз предметної галузі оцінювання ефективності командної розробки програмного забезпечення та досліджено сучасні інструменти моніторингу активності розробників у Git-середовищах. Виконано порівняльний аналіз існуючих рішень, зокрема GitHub Insights, GitLab Analytics, Gource та інших систем аналітики, визначено їхні функціональні можливості, переваги та недоліки.

Другий розділ присвячений проєктуванню веб-додатку та його архітектури. На основі проведеного аналізу сформовано функціональні та нефункціональні вимоги до програмного продукту DTracker. У роботі виконано проєктування архітектури веб-додатку, розроблено UML-діаграми, ER-модель бази даних та структуру взаємодії програмних компонентів системи.

У третьому розділі створено серверну частину веб-додатку з використанням мови програмування Python та фреймворку Flask. Для роботи з GitHub API використано бібліотеку PyGitHub, для аналізу статистичних даних – бібліотеку Pandas, а для побудови інтерактивних графіків – бібліотеку Plotly. Клієнтська частина системи реалізована із застосуванням HTML5, CSS3, JavaScript та Bootstrap 5.

Розроблений веб-додаток забезпечує автоматичний збір статистики активності розробників, аналіз commit history, виявлення періодів неактивності та формування PDF-звітів. Він підключається до репозиторіїв, автоматично збирає метрики активності кожного учасника, будує інтерактивні графіки та виявляє тривалі перерви в роботі.

Четвертий розділ присвячений питанням тестування та впровадження веб-додатку DTracker.

Практичне значення роботи полягає у створенні програмного продукту, який може використовуватися тім-лідами, менеджерами проєктів та аналітиками для моніторингу ефективності командної розробки програмного забезпечення. Сфера застосування веб-додатку – технічне управління програмними проєктами, оцінювання продуктивності команд тім-лідами, проджект-менеджерами та аналітиками.

ABSTRACT

The bachelor's thesis consists of an introduction, four sections, conclusions, a list of sources used and appendices.

The thesis is devoted to the development of a web application for analyzing data on activity in version control systems.

The first chapter analyzes the subject area of assessing the effectiveness of team software development and examines modern tools for monitoring developer activity in Git environments. A comparative analysis of existing solutions, in particular GitHub Insights, GitLab Analytics, Gource and other analytics systems, is performed, and their functional capabilities, advantages and disadvantages are determined.

The second chapter is devoted to the design of the web application and its architecture. Based on the analysis, functional and non-functional requirements for the DTracker software product are formed. The work designs the architecture of the web application, develops UML diagrams, an ER database model, and a structure for the interaction of the system's software components.

In the third chapter, the server part of the web application was created using the Python programming language and the Flask framework. The PyGitHub library was used to work with the GitHub API, the Pandas library was used to analyze statistical data, and the Plotly library was used to build interactive graphs. The client part of the system was implemented using HTML5, CSS3, JavaScript, and Bootstrap 5. The developed web application provides automatic collection of developer activity statistics, commit history analysis, detection of periods of inactivity, and generation of PDF reports. It connects to repositories, automatically collects activity metrics for each participant, builds interactive graphs, and detects long breaks in work.

The fourth chapter is devoted to the issues of testing and implementation of the DTracker web application.

The practical significance of the thesis lies in the creation of a software product that can be used by team leaders, project managers and analysts to monitor the effectiveness of team software development. The scope of the web application is technical management of software projects, assessment of team performance by team leaders, project managers and analysts.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

PDF – Portable Document Format

IDE – Integrated Development Environment (інтегроване середовище розробки)

API – Application Programming Interface (програмний інтерфейс застосунку)

Git – розподілена система керування версіями

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

JS – JavaScript

HTTP – HyperText Transfer Protocol

REST – Representational State Transfer

UI – User Interface (інтерфейс користувача)

ORM – Object-Relational Mapping

ВСТУП

Сучасна індустрія розробки програмного забезпечення характеризується високими темпами розвитку, широким використанням гнучких методологій управління проєктами та активним застосуванням систем контролю версій. У більшості IT-компаній процес створення програмних продуктів здійснюється командами розробників, які одночасно працюють над різними компонентами програмного забезпечення. Зі зростанням чисельності учасників та складності проєктів дедалі гостріше постає потреба в об'єктивному інструменті оцінювання індивідуального внеску кожного члена команди. Своєчасне виявлення неактивних учасників, відстеження динаміки змін у кодовій базі та прийняття обґрунтованих управлінських рішень – ці завдання можна ефективно вирішити лише за наявності якісного засобу аналітики. За таких умов особливого значення набувають питання контролю активності учасників команди, оцінювання продуктивності процесу розробки та аналізу динаміки виконання проєктів.

Одним із ключових інструментів сучасної розробки є системи контролю версій, серед яких найбільшого поширення набув Git. На базі Git функціонують популярні платформи спільної розробки, зокрема GitHub, GitLab та Bitbucket, які накопичують значний обсяг даних про діяльність розробників: історію комітів, pull request, merge request, статистику змін коду та інші показники. Аналіз цих даних дозволяє отримувати важливу інформацію щодо ефективності командної роботи, стабільності процесу розробки та потенційних ризиків затримки виконання проєктів.

Поширення інструментів контролю версій відкрило можливості для автоматизованого моніторингу діяльності учасників проєкту. Однак, більшість доступних аналітичних рішень або мають обмежену функціональність, або орієнтовані виключно на великі корпоративні структури й не враховують потреб невеликих команд та стартапів. Зокрема, вони часто не забезпечують гнучкої візуалізації статистики, автоматичного

виявлення тривалих перерв у роботі, комплексного аналізу активності окремих учасників команди та формування детальних звітів для менеджерів проєктів. Крім того, переважна кількість таких засобів реалізована у вигляді десктопних застосунків або вбудованих панелей конкретних платформ, що суттєво знижує їхню доступність та гнучкість. Це обумовлює актуальність розроблення спеціалізованих систем аналітики активності розробників.

Веб-орієнтований підхід до розробки аналітичних інструментів надає низку переваг: незалежність від операційної системи клієнта, зручний доступ через браузер без встановлення додаткового програмного забезпечення, а також можливість підключення до репозиторіїв різних сервісів з єдиного інтерфейсу. Саме тому актуальним є створення веб-додатку, що забезпечує наочну візуалізацію активності розробників, порівняння внеску за довільними часовими інтервалами, збір статистики із різних Git-платформ та виявлення потенційних проблем у роботі команди.

Отже, актуальність теми кваліфікаційної роботи полягає у необхідності створення сучасного веб-додатку для автоматизованого збору, обробки та візуалізації даних про активність учасників програмних проєктів у системах контролю версій. Такий програмний продукт дозволить підвищити ефективність управління командами розробників, спростити процес оцінювання продуктивності та покращити контроль за виконанням програмних проєктів.

Метою кваліфікаційної роботи є розроблення веб-додатку аналізу даних про активність у системах контролю версій, який забезпечує підвищення ефективності моніторингу внеску розробників у командну розробку програмного забезпечення шляхом автоматичного збору та аналізу даних з репозиторіїв.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– проаналізувати предметну галузь оцінювання ефективності командної розробки програмного забезпечення;

– дослідити існуючі інструменти аналізу Git-репозиторіїв;

– визначити функціональні та нефункціональні вимоги до програмного продукту;

– виконати проєктування архітектури веб-додатку;

– розробити структуру бази даних;

– реалізувати серверну та клієнтську частини системи;

– реалізувати модулі аналізу активності та візуалізації даних;

– забезпечити формування PDF-звітів;

– провести тестування та впровадження програмного продукту.

Об'єктом дослідження є процес збору та аналізу даних щодо активності учасників у системах контролю версій для оцінювання їхнього індивідуального внеску в командну розробку програмного забезпечення.

Предметом дослідження є методи, моделі та програмні засоби збирання, опрацювання та візуалізації статистики активності розробників на основі даних із систем контролю версій.

У процесі виконання роботи використовувалися:

– мова програмування Python;

– веб-фреймворк Flask;

– бібліотека PyGitHub для взаємодії з GitHub API;

– бібліотека Pandas для аналізу даних;

– бібліотека Plotly для побудови інтерактивних графіків;

– HTML, CSS, JavaScript та Bootstrap 5 для реалізації клієнтської частини.

Методи дослідження: на першому етапі проведено аналіз наявних програмних рішень для відстеження активності в командах розробників, зокрема GitHub Insights, GitLab Analytics, Gource та інших, з метою виявлення їхніх переваг, недоліків і функціональних обмежень. На підставі цього аналізу сформульовано функціональні та нефункціональні вимоги до розроблюваної системи. На наступних етапах спроєктовано архітектуру

рішення, реалізовано програмний код та виконано тестування отриманого продукту.

Наукова новизна роботи полягає в розробці веб-орієнтованого рішення, що поєднує можливості підключення до GitHub, інтерактивну візуалізацію активності учасників команди, автоматичне виявлення неактивних розробників і формування PDF-звітів в єдиному інтерфейсі.

Результати виконання кваліфікаційної роботи полягають у створенні веб-додатку DTracker, який може бути використаний тім-лідами, менеджерами проєктів, аналітиками та командами розробників для автоматизованого аналізу активності учасників програмних проєктів та підтримки процесів технічного управління.

Практична цінність результатів полягає в можливості застосування розробленого веб-додатку менеджерами проєктів, тім-лідами та технічними керівниками для оперативного оцінювання продуктивності команди, прийняття управлінських рішень на основі фактичних даних, а також для підвищення прозорості та відповідальності у процесі командної розробки.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ ДЛЯ ОЦІНЮВАННЯ АКТИВНОСТІ РОЗРОБНИКІВ У СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

1.1. Особливості командної розробки програмного забезпечення

Сучасна індустрія програмного забезпечення характеризується високою складністю проєктів, необхідністю швидкої адаптації до змін та постійною взаємодією між учасниками команд розробки. У більшості випадків програмні продукти створюються не окремими розробниками, а командами, до складу яких входять backend- та frontend-розробники, тестувальники, DevOps-інженери, системні аналітики, дизайнери та менеджери проєктів.

Ефективність командної роботи безпосередньо впливає на якість програмного продукту, строки виконання задач та фінансові результати компанії. У зв’язку з цим особливого значення набувають засоби моніторингу активності учасників команди та аналізу процесу розробки.

Одним із ключових елементів сучасної розробки програмного забезпечення є системи контролю версій. Вони забезпечують збереження історії змін, підтримують паралельну роботу над кодом та дозволяють координувати діяльність великої кількості учасників проєкту. Найбільш поширеною системою контролю версій сьогодні є Git.

Git дозволяє:

– відстежувати зміни у файлах проєкту;

– зберігати історію комітів;

– організовувати паралельну роботу над різними функціональними гілками;

– здійснювати злиття змін;

– аналізувати внесок окремих учасників.

На базі Git функціонують популярні платформи спільної розробки, серед яких:

– GitHub;
– GitLab;
– Bitbucket.

Використання таких платформ створює значний обсяг даних про процес розробки, які можуть бути використані для оцінювання продуктивності команди.

До таких даних належать:

– кількість комітів;
– активність у pull request;
– частота змін;
– кількість змінених рядків коду;
– участь у code review;
– часові проміжки між комітами;
– активність за днями тижня та годинами;
– тривалі перерви у роботі.

Аналіз зазначених метрик дозволяє:

– оцінювати завантаженість команди;
– виявляти ризики затримок;
– визначати нерівномірність розподілу задач;
– контролювати динаміку розробки;
– виявляти зниження активності окремих учасників;
– покращувати процес управління програмними проєктами.

Таким чином, задача автоматизованого аналізу активності розробників у Git-середовищах є актуальною та має практичне значення для сучасних IT-компаній.

1.2. Програмне забезпечення для моніторингу репозиторіїв

Засоби моніторингу репозиторіїв – це спеціалізовані програмні комплекси, призначені для збору, опрацювання та відображення інформації про діяльність кожного учасника командної розробки. Такі системи, як правило, взаємодіють із платформами контролю версій через відповідні програмні інтерфейси та дозволяють здійснювати детальний аналіз активності на базі фіксованих подій: комітів, відкриття та злиття гілок, перегляду запитів на включення змін.

Головна функція подібних засобів – зробити процес розробки прозорим для керівників різного рівня та надати об'єктивну аналітичну основу для прийняття рішень.

Сучасні рішення у цій галузі охоплюють такі ключові складові:

– збір даних – отримання відомостей про коміти, гілки, запити на злиття та інші дії учасників через програмні інтерфейси сервісів контролю версій;

– опрацювання даних – агрегування, фільтрування та структурування зібраної інформації для подальшого аналізу;

– візуалізація – побудова графіків, діаграм, теплових карт, що відображають внесок кожного учасника в розвиток проєкту;

– експорт звітів – збереження аналітичних матеріалів у форматі PDF-документів для презентацій або внутрішнього використання;

– фільтрація за часовим діапазоном – вибір довільних часових інтервалів для порівняння активності або відстеження прогресу проєкту.

Серед основних цілей програмного забезпечення (далі – ПЗ) цього класу виділяють:

– оцінювання результативності кожного розробника;

– визначення найактивніших і найменш активних учасників;

– виявлення загальних тенденцій у темпах розробки;

– побудову прозорої системи обліку внеску;

– спрощення управлінських рішень;

– формування об'єктивної звітності.

Перевагами подібних рішень є:

– прозорість розробки: аналітичний інструментарій дає чітке уявлення про участь кожного члена команди в проєкті. Керівники можуть легко відслідковувати обсяг змін у коді, частоту комітів, участь у злитті гілок тощо. Це допомагає виявити як найбільш результативних, так і потенційно неактивних учасників;

– рішення, обґрунтовані даними: завдяки графічному представленню активності менеджери можуть оперативно реагувати на уповільнення роботи, оптимізувати розподіл навантаження між учасниками та планувати дедлайни й обсяг задач більш ефективно;

– автоматизація звітності: система здатна генерувати звіти автоматично на основі даних з репозиторіїв, що підвищує точність та знижує ризик суб'єктивних оцінок;

– мотиваційний ефект: усвідомлення того, що активність фіксується й відображається в статистиці, спонукає учасників до відповідальнішого ставлення до завдань і підвищує рівень самодисципліни;

– масштабованість: система здатна обробляти дані як невеликих команд, так і великих проєктів із десятками учасників.

Серед недоліків таких засобів варто відзначити:

– обмеженість у вимірюванні якості: попри значний обсяг статистичних даних, система не здатна повноцінно оцінити якість коду, складність реалізованих алгоритмів чи архітектурні покращення;

– ризик хибної інтерпретації: без достатнього розуміння аналітичних показників керівники можуть робити некоректні висновки;

– тиск на команду: якщо аналітика використовується як єдиний критерій оцінки праці, це може спричинити стрес або штучне завищення активності;

– залежність від зовнішніх API: системи, що працюють через Git API, стикаються з обмеженнями на кількість запитів і можливими змінами в структурі даних.

1.3. Цільова аудиторія системи

Веб-додаток для аналізу активності в системах контролю версій може бути орієнтований на широке коло фахівців, які займаються організацією, оцінюванням та вдосконаленням командної роботи у сфері розробки програмного забезпечення.

До основних категорій користувачів належать:

– *тімліди* – фахівці, відповідальні за продуктивність команди, розподіл завдань і виявлення «вузьких місць». Вони зацікавлені в оперативному отриманні аналітики щодо активності кожного учасника;

– *прожект-менеджери* – спеціалісти, що стежать за виконанням проєктів у визначені терміни. Для них важлива загальна динаміка роботи команди, прогрес у виконанні завдань та виявлення активних і пасивних періодів;

– *технічні директори та керівники IT-підрозділів* – управлінці, що оцінюють ефективність команд на рівні організації та приймають рішення щодо кадрового складу;

– *аналітики продуктивності* – фахівці, яким необхідна достовірна й структурована інформація з Git-систем для формування звітів і діаграм.

– *HR-фахівці в IT-сфері* – спеціалісти, що здійснюють моніторинг професійного розвитку співробітників і використовують аналітику активності для оцінювання персоналу.

– *самостійні розробники та фрілансери* – індивідуальні програмісти, які прагнуть отримати об'єктивну картину власної активності та структурувати робочий час.

1.4. Метрики оцінювання ефективності командної розробки

Для оцінювання ефективності роботи команди програмної розробки використовуються різноманітні кількісні та якісні показники. Найбільш поширеними серед них є метрики активності у системах контролю версій.

Основними метриками є:

Кількість комітів.

Дана метрика характеризує інтенсивність внесення змін до репозиторію. Висока кількість комітів може свідчити про активну участь розробника у проєкті, однак не завжди прямо визначає продуктивність.

Кількість змінених рядків коду.

Метрика враховує:

– додані рядки;
– видалені рядки;
– модифіковані рядки.

Вона дозволяє оцінити масштаб виконаних змін, однак також не може бути єдиним показником ефективності, оскільки якісний код не завжди є великим за обсягом.

Частота комітів.

Аналіз часових інтервалів між комітами дозволяє оцінювати регулярність роботи учасників команди та стабільність процесу розробки.

Активність у pull request.

Pull request є важливим елементом collaborative development. Аналіз їх кількості та стану дозволяє визначити:

– рівень взаємодії між розробниками;
– інтенсивність code review;
– швидкість інтеграції змін.

Періоди неактивності.

Тривалі перерви між комітами можуть свідчити про:

– перевантаження розробника;

– проблеми в організації роботи;
– ризики затримки виконання задач;
– зниження продуктивності.

Активність за часовими інтервалами.

Дослідження активності за днями тижня та годинами дозволяє:

– аналізувати робочі навантаження;
– виявляти перевантаження команди;
– оцінювати дотримання робочого графіка.

Слід зазначити, що жодна окрема метрика не може повністю характеризувати ефективність роботи розробника. Саме тому сучасні системи аналітики використовують комплексний підхід до оцінювання діяльності команд.

1.6. Особливості оцінювання командної розробки

Оцінювання продуктивності учасників командного програмного проєкту, має низку суттєвих особливостей, що відрізняють його від оцінювання індивідуальної діяльності. Головна складність полягає в колективному характері роботи, великій кількості змінних та значному впливі людського чинника.

Можливість аналізу повної історії комітів, авторства змін та супровідних повідомлень у системах контролю версій (Git-платформи) дозволяє формувати об'єктивні метрики продуктивності. Це складає так званий цінний аналітичний ресурс.

Водночас команди мають чіткий розподіл ролей: розробники бекенду та фронтенду, тестувальники, DevOps-інженери, архітектори, тімліди. Внесок різних учасників може проявлятися по-різному, наприклад, тестувальник може не виконувати комітів у кодову базу, але відігравати ключову роль у забезпеченні якості продукту. Отже, оцінювання має враховувати рольову специфіку та розподіл відповідальності.

За Agile-методологією навантаження у проєктах розподіляється по спринтах нерівномірно. Одна частина розробників більше залучена на старті (планування, архітектура), інша частина – наприкінці циклу (тестування, інтеграція). Це необхідно брати до уваги при інтерпретації показників активності.

Тим паче, ефективність розробника не завжди вимірюється кількістю комітів. Якість коду, комунікація, участь у code review та здатність до командної взаємодії – це критично важливі, але менш формалізовані аспекти. Тому доцільно поєднувати кількісні метрики з якісним оцінюванням.

Крім того, на роботу команди можуть впливати фактори, що не відображаються в системі контролю версій: технічні збої, форс-мажорні обставини, внутрішні конфлікти.

Водночас, оцінювання має бути динамічним – важливо не лише фіксувати підсумковий внесок, а й своєчасно виявляти зниження або зростання активності в процесі роботи.

1.7. Аналіз існуючих інструментів моніторингу активності розробників

У сучасних умовах інтенсивної командної розробки ПЗ особливого значення набуває завдання об'єктивного оцінювання внеску окремих учасників. Для вирішення цього завдання існує низка спеціалізованих інструментів, орієнтованих на збір, відображення та аналіз даних із систем контролю версій. У цьому розділі розглянемо найпоширеніші з них.

Серед проаналізованих систем:

– GitHub Insights;
– GitLab Analytics;
– Gource;
– Open Source Contributions Dashboard.

1.7.1. GitHub Insights

GitHub надає вбудований набір аналітичних інструментів GitHub Insights, що дозволяє отримувати статистичні дані про активність репозиторію.

GitHub Insights – це вбудований інструмент статистики активності, призначений для команд, що розробляють ПЗ у середовищі GitHub (зокрема в рамках GitHub Enterprise). Основна мета цього інструменту – надати керівникам проєктів наочні показники залученості учасників команди на основі графіків активності, кількості комітів, запитів на злиття тощо.

Функціональні можливості GitHub Insights включають побудову хронологічних графіків комітів, порівняння рівнів активності в різні часові відрізки, зведену статистику участі кожного члена команди, аналіз змін у темпі роботи. Завдяки вбудованій інтеграції з репозиторіями GitHub усі дані оновлюються автоматично, без потреби встановлювати сторонні компоненти. Доступ до інструменту здійснюється через браузер.

Серед переваг GitHub Insights – зручна візуалізація даних та можливість порівняння часових інтервалів, що дозволяє виявити зміни в темпі розробки. Водночас, інструмент має суттєві обмеження, такі як відсутність аналізу на рівні рядків коду, відсутність автоматичних сповіщень про зниження активності, підтримка лише однієї платформи GitHub, без інтеграції з GitLab чи Bitbucket. Відсутність десктопного застосунку також обмежує офлайн-доступ.

Приклад графіку, який будує GitHub Insights, наведено на рис. 1.1.

Якщо узагальнити, основні можливості GitHub Insights:

– аналіз кількості комітів;
– статистика pull request;
– відображення внеску учасників;
– аналіз активності за періодами;
– графіки розвитку репозиторію.

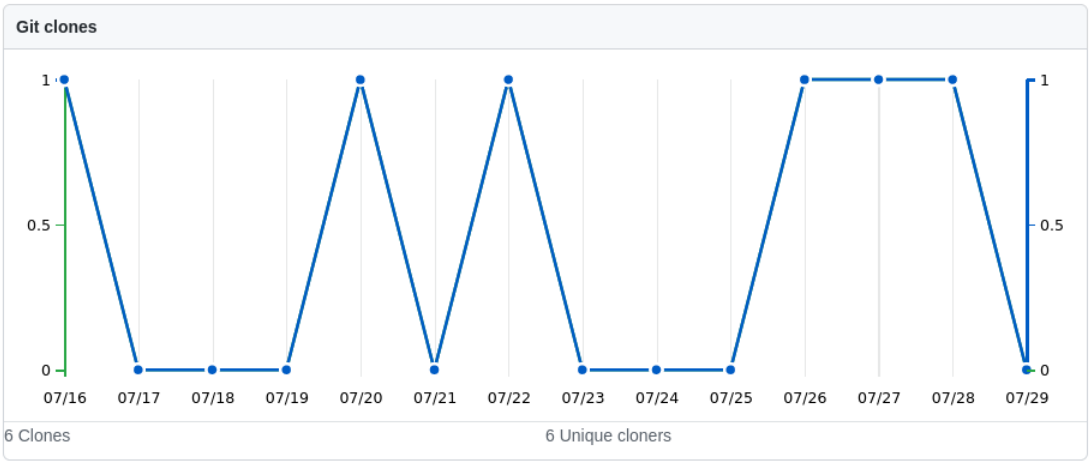

Рисунок 1.1. Приклад графіку у GitHub Insights

Перевагами GitHub Insights є:

– інтеграція безпосередньо у платформу GitHub;
– простота використання;
– інтерактивна візуалізація даних.

Недоліки GitHub Insights такі:

– обмежені можливості кастомізації;
– відсутність глибокого аналізу продуктивності;
– обмежений набір метрик;
– неможливість формування складних звітів;
– відсутність автоматичного виявлення тривалих перерв у роботі.

1.7.2. GitLab Analytics

GitLab Analytics – аналітична панель, інтегрована безпосередньо в платформу GitLab, яка містить вбудовані інструменти аналітики та моніторингу DevOps-процесів.

Призначена для відстеження активності розробників і команд у процесі розробки ПЗ та надання менеджерам і тімлідам даних щодо ефективності роботи команди через аналіз метрик з репозиторіїв GitLab.

Інструмент дозволяє переглядати кількість комітів, злиттів та відкритих/закритих задач; відображати внесок кожного учасника у графічному вигляді; відстежувати зміни в конкретних модулях; аналізувати активність по днях, тижнях або спринтах; оцінювати участь у code review та частоту комітів. Передбачена й групова аналітика – агреговані дані за командами.

Перевагами GitLab Analytics є комплексна візуалізація та можливість аналізу не лише комітів, а й участі в рев'ю. Разом із тим інструмент має певний набір метрик без можливості кастомізації, відсутні автоматичні сповіщення про зниження активності, а доступ можливий виключно онлайн.

Приклад графіку, який будує GitLab Analytics, наведено на рис. 1.2.

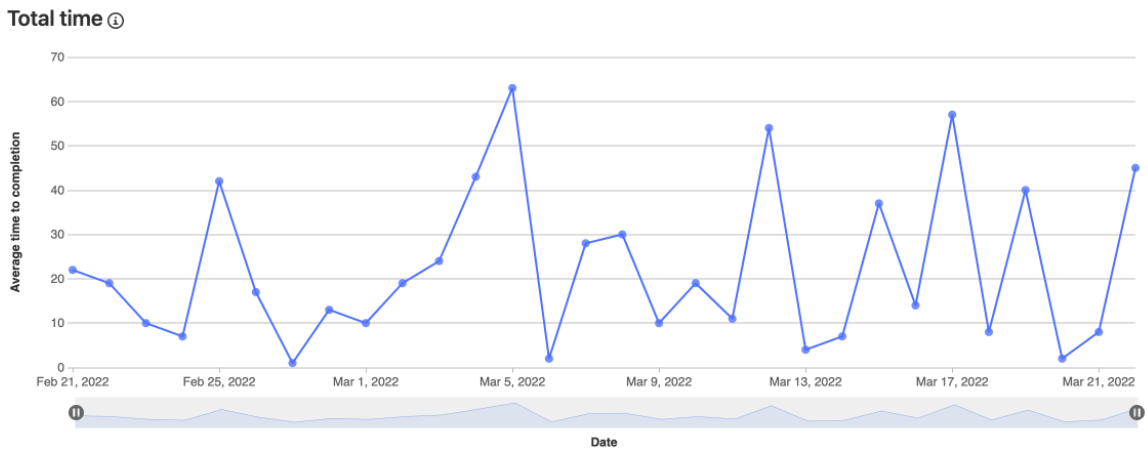

Рисунок 1.2. Приклад графіку у GitLab Analytics

Основні функції GitLab Analytics:

– аналіз commit activity;
– статистика merge request;
– оцінювання cycle time;
– моніторинг CI/CD процесів;
– аналіз продуктивності команди.

Перевагами GitLab Analytics є:

– інтеграція з DevOps-процесами;
– підтримка self-hosted середовищ;
– гнучкі засоби адміністрування.

Недоліками GitLab Analytics є:

– складність налаштування;
– частина функціоналу доступна лише у платних версіях;
– значне навантаження на серверну інфраструктуру;
– недостатня гнучкість побудови користувацьких звітів.

1.7.3. Gource

Gource є програмою візуалізації історії репозиторіїв у вигляді анімованого графічного представлення, тобто програмний засіб для побудови анімованої візуалізації хронології змін у Git-репозиторіях. Інструмент перетворює дані про коміти та активність учасників на інтерактивну графіку у вигляді дерева залежностей, де кожен вузол відповідає окремому файлу або учаснику проєкту. Gource активно використовується для демонстрацій, презентацій та наочного відображення командної роботи у відкритих проєктах.

Серед його функцій анімована візуалізація структури репозиторію в режимі реального часу, хронологічне відтворення змін за весь період існування проєкту, підтримка різних систем контролю версій (Git, Mercurial, Bazaar), можливість збереження результату у відеоформат, кросплатформеність (Windows, macOS, Linux).

Головними перевагами є наочність і простота запуску. Водночас, Gource не надає текстових звітів і кількісних метрик, не дозволяє проводити детальний аналіз активності учасників, не має прямої інтеграції з GitHub чи GitLab API (вимагає попереднього клонування репозиторію) та позбавлений можливості фільтрації даних.

Приклад візуалізації в Gource наведено на рис. 1.3.

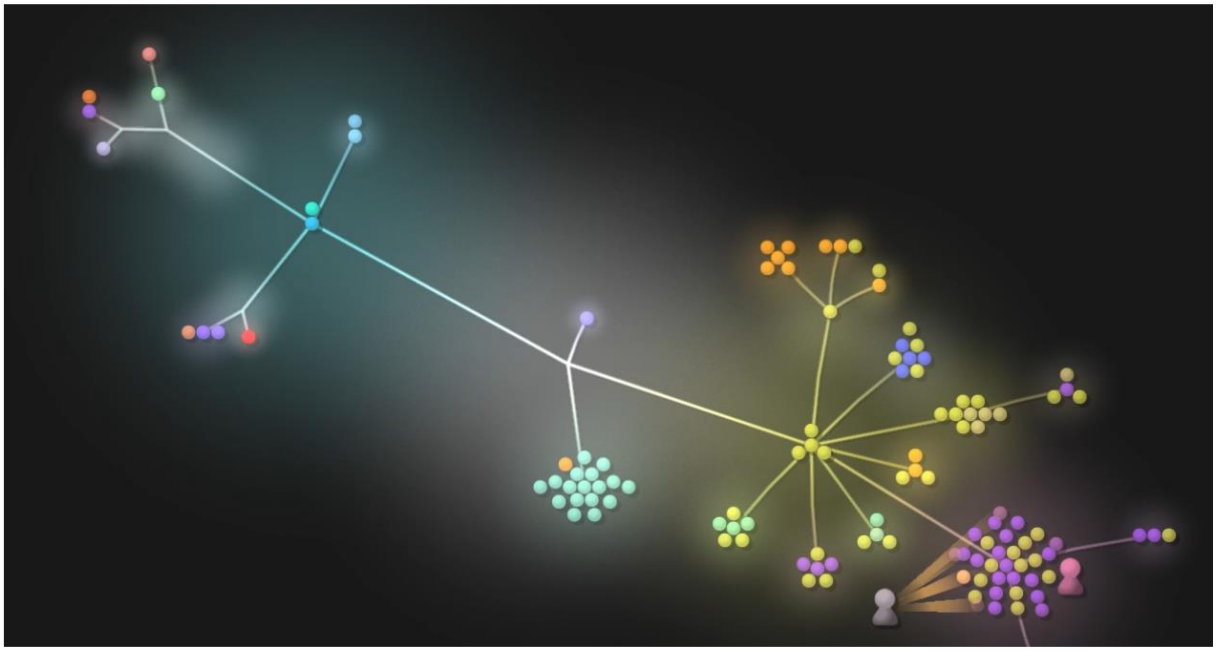

Рисунок 1.3 Приклад візуалізації у Gource

Можливості Gource:

– анімована візуалізація розвитку проєкту;
– відображення активності розробників;
– аналіз історії змін.

Перевагами Gource є:

– наочне представлення процесу розробки;
– підтримка різних систем контролю версій;
– можливість створення презентацій.

Недоліками Gource є:

– відсутність детальної аналітики;
– відсутність статистичних звітів;
– складність використання для управлінського аналізу;
– орієнтованість переважно на візуалізацію.

1.7.4. Open Source Contributions Dashboard

Open Source Contributions Dashboard – веб-орієнтований інструмент для збору та відображення активності розробників, що працюють над відкритими програмними проєктами, який вирішує проблему відсутності зручної аналітики внеску учасників у Git-репозиторії, надаючи координаторам спільнот та керівникам команд зручні засоби для оцінювання активності учасників.

Основними його можливостями є перегляд статистики за репозиторіями, аналіз індивідуального внеску, підтримка кількох платформ (GitHub, GitLab, Bitbucket), інтерактивні графіки та системи фільтрації і сортування даних, можливість експорту звітів.

Серед переваг – підтримка кількох Git-платформ одночасно та зручна оцінка індивідуального внеску. Недоліками є відсутність деталізації на рівні рядків коду, обмежена аналітика динаміки внеску, виключно веб-орієнтований доступ та відсутність системи сповіщень.

Приклад аналітики Open Source Contributions Dashboard наведено на рис. 1.4.

Рисунок 1.4. Приклад аналітики Open Source Contributions Dashboard

1.7.5. Інші інструменти аналізу Git-репозиторіїв

Крім вище розглянутих, існують інші системи аналізу активності розробників:

– SonarQube;
– Code Climate;
– Waydev;
– Pluralsight Flow;
– GitStats.

Вони забезпечують:

– аналіз якості коду;
– побудову аналітичних звітів;
– оцінювання продуктивності команд;
– виявлення проблем у процесі розробки.

Однак більшість таких систем мають один або декілька недоліків:

– висока вартість ліцензій;
– складність інтеграції;
– залежність від корпоративної інфраструктури;
– обмеження безкоштовних версій;
– недостатня гнучкість налаштування.

1.7.6. Порівняльний аналіз існуючих рішень

Для визначення особливостей існуючих систем було проведено порівняльний аналіз основних інструментів моніторингу активності розробників.

У таблиці 1.1 наведено зведені результати порівняльного аналізу.

Таблиця 1.1. Порівняння існуючих систем аналізу Git-репозиторіїв

Функція	**GitHub Insights**	**GitLab Analytics**	**Gource**	**OSCD**	**DTracker**
Платформа	Web	Web	Web	Desktop	Web
Відстеження рядків коду	+	+	+	-	+
Аналіз комітів	+	+	+	+	+
Аналіз pull request	+	+	-	+	+
Інтерактивні графіки	+	+	+	+	+
Виявлення перерв у роботі	-	-	-	Частково	+
Сповіщення про неактивних	+	+	+	-	-
Візуалізація активності	+	+	+	+	+
Порівняння за періодами	+	+	+	-	+
PDF-звіти	-	Частково	-	+	+
Гнучка кастомізація	Низька	Середня	Низька	Середня	Висока
Безкоштовне використання	Частково	Частково	+	-	+
Кілька Git-сервісів	+	-	-	+	- (в майбутньому)
Простота інтеграції	+	Середня	+	Середня	+

В результаті аналізу встановлено, що існуючі рішення не повністю задовольняють потреби технічного управління програмними проєктами. Основними недоліками є:

– відсутність комплексного аналізу активності;
– недостатня гнучкість формування звітності;
– складність інтеграції;
– обмежені можливості кастомізації;

– відсутність спеціалізованих механізмів виявлення тривалих перерв у роботі.

1.8. Визначення вимог до програмного продукту

Аналіз особливостей процесу оцінювання внеску у командну розробку та порівняльне дослідження аналогів дають змогу сформулювати вимоги до розроблюваного веб-додатку DTracker.

1.8.1. Функціональні вимоги

Функціональними вимогами до розроблюваного веб-додатку:

– багатокористувацький режим;

– підключення до GitHub-репозиторію через назву або персональний токен доступу, що вводить користувач;

– отримання та опрацювання даних з репозиторію через GitHub API після ініціювання оновлення з формуванням аналітики про діяльність кожного розробника;

– автоматичне формування переліку учасників репозиторію з можливістю переходу до індивідуальних звітів;

– автоматичний аналіз commit history, активності окремих учасників, pull request, виявлення тривалих перерв у роботі;

– побудова і відображення інтерактивних графіків: змін по тижнях, найчастіше змінюваних файлів, динаміки за останні тижні;

– збереження сформованих звітів у форматі PDF на пристрій користувача;

– надання статистики за часовими інтервалами;

– виведення попереджень про учасників, що не здійснювали комітів протягом визначеного часового інтервалу.

1.8.2. Нефункціональні вимоги

Нефункціональними вимогами до веб-додатку є:

– масштабованість;

– зручний веб-інтерфейс – система реалізована у вигляді веб-додатку, доступного через браузер на будь-якій операційній системі;

– високу швидкодію – час генерації аналітики не перевищує 20 секунд для команди середнього розміру, інтерфейс реагує на дії користувача менш ніж за 1 секунду;

– безпечне зберігання токенів доступу у захищеному вигляді, не передаючи їх стороннім сервісам;

– адаптивність інтерфейсу – у разі виникнення помилок при запиті до репозиторію система інформує користувача з поясненням причини;

– інтерфейс україномовний.

– можливість подальшого розширення функціоналу.

1.8.3. Технологічні вимоги

Технологічні вимоги до розроблюваного веб-додатку визначають доцільність використання:

– мови програмування Python;

– веб-фреймворку Flask;

– бібліотеки PyGitHub для роботи з GitHub API;

– бібліотеки Pandas для аналізу даних;

– бібліотеки Plotly для інтерактивної візуалізації;

– HTML, CSS та JavaScript для клієнтської частини;

– Bootstrap 5 для адаптивного інтерфейсу.

Висновки до розділу

У першому розділі було проаналізовано предметну галузь оцінювання ефективності командної розробки програмного забезпечення та досліджено сучасні системи моніторингу активності розробників у Git-середовищах.

Встановлено, що системи контролю версій є важливим джерелом даних для аналізу процесу розробки програмного забезпечення. Досліджено основні метрики оцінювання активності розробників, серед яких кількість комітів, активність у pull request, часові інтервали між змінами та статистика внеску учасників.

Проведено порівняльний аналіз існуючих рішень, зокрема GitHub Insights, GitLab Analytics та Gource. Виявлено їхні переваги та недоліки, а також встановлено необхідність створення більш гнучкої системи аналітики.

На основі виконаного аналізу сформовано функціональні та нефункціональні вимоги до веб-додатку DTracker, який має забезпечити автоматизований аналіз активності розробників, інтерактивну візуалізацію даних та формування звітів для підтримки процесів управління програмними проєктами.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-ДОДАТКУ ДЛЯ АНАЛІЗУ ДАНИХ ПРО АКТИВНІСТЬ В СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

2.1. Середовище розробки

Для розробки веб-додатку DTracker використовувалось інтегроване середовище розробки Visual Studio Code (VS Code) – один із найпопулярніших інструментів для веб-розробки та роботи з Python. VS Code надає розширену підтримку налагодження, автодоповнення коду, управління розширеннями та зручний інтерфейс для роботи з репозиторіями через вбудовану інтеграцію з Git. Завдяки багатому набору розширень (Pylance, Python, GitLens тощо) середовище забезпечує ефективну розробку як серверної (Python/Flask), так і клієнтської (HTML/CSS/JS) частин застосунку.

Для управління залежностями та ізоляції середовища виконання застосовувався інструмент virtualenv, що дозволяє створювати окремі Python-оточення для кожного проєкту та уникати конфліктів між версіями бібліотек.

2.2. Мова програмування та фреймворк серверної частини

Основу серверної частини веб-додатку становить мова Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Python вирізняється широкою екосистемою бібліотек для наукових, аналітичних та інженерних задач, що робить її зручним інструментом для швидкого прототипування і створення стабільних програмних систем.

Python було обрано як основну мову завдяки її універсальності, читабельності синтаксису та великому набору бібліотек для опрацювання даних, взаємодії з API та генерації звітів. Використання версії Python 3.11+

забезпечує доступ до сучасних можливостей та підвищену продуктивність виконання коду.

Для організації серверної частини використовується фреймворк Flask – легкий і гнучкий фреймворк для Python, що надає мінімальний необхідний набір інструментів для побудови веб-застосунків. Flask дозволяє швидко розгорнути REST API, зручно організовувати маршрутизацію запитів та взаємодіяти з шаблонами сторінок через вбудований механізм Jinja2. На відміну від важких фреймворків, Flask залишає архітектурні рішення на розсуд розробника, що спрощує підтримку та масштабування проєкту.

2.3. Доступ до системи контролю версій

GitHub – веб-платформа для хостингу коду, що базується на розподіленій системі контролю версій Git. GitHub забезпечує інструменти для спільної роботи, управління версіями та розподіленого доступу до коду.

Платформа є основним джерелом аналітичних даних для системи DTracker: через REST API GitHub надається доступ до метрик комітів, учасників, файлів та іншої статистики репозиторіїв. Система контролю версій Git використовувалася як під час розробки застосунку, так і в якості джерела даних для аналізу. Git зберігає повну хронологію змін, дозволяє відстежувати активність учасників, управляти гілками та виконувати злиття.

Система DTracker взаємодіє з віддаленими Git-сервісами через відповідні API для збору статистики про внесок учасників проєкту за допомогою функцій бібліотеки PyGitHub (рис. 2.1).

```python
def connect(token, repository):
    connect_result = {'message': "З'єднання успішне!", 'success': False, 'repo': None}

    try:
        g = Github(f"{token}")
        repo = g.get_repo(f"{repository}")
        connect_result['success'] = True
        connect_result['repo'] = repo
    except Exception as e:
        connect_result['message'] = f"[ERROR] Невідома помилка: {str(e)}"

    return connect_result
```

Рисунок 2.1. Приклад отримання даних за допомогою PyGitHub

2.4. Загальна концепція веб-додатку

У результаті аналізу предметної галузі було визначено необхідність створення веб-додатку DTracker, призначеного для автоматизованого збору та аналізу даних про активність учасників програмних проєктів у системах контролю версій Git.

Основною метою системи є надання тім-лідам, проєктним менеджерам та аналітикам інструментів для:

– оцінювання продуктивності команд;
– аналізу активності розробників;
– виявлення перерв у роботі;
– побудови інтерактивних візуалізацій;
– формування звітної документації.

Веб-додаток DTrack реалізується як клієнт-серверна система з використанням мови програмування Python та фреймворку Flask.

До складу системи входять:

– клієнтська частина;
– серверна частина;
– модуль роботи з GitHub API;

– модуль аналітики;
– модуль побудови графіків;
– модуль генерації PDF-звітів;
– база даних.

2.5. Архітектура веб-додатку

Для реалізації DTracker було обрано багаторівневу архітектуру, яка забезпечує:

– модульність;
– масштабованість;
– простоту супроводу;
– незалежність компонентів.

Архітектура системи складається з наступних рівнів:

1. Presentation Layer – клієнтський інтерфейс.
2. Application Layer – серверна логіка.
3. Data Processing Layer – аналітичний модуль.
4. Data Access Layer – робота з базою даних.
5. External API Layer – інтеграція з GitHub API.

Структурна схема архітектури веб-додатку представлена на рис. 2.2.

Основною перевагою такої архітектури є можливість незалежного розвитку окремих модулів системи.

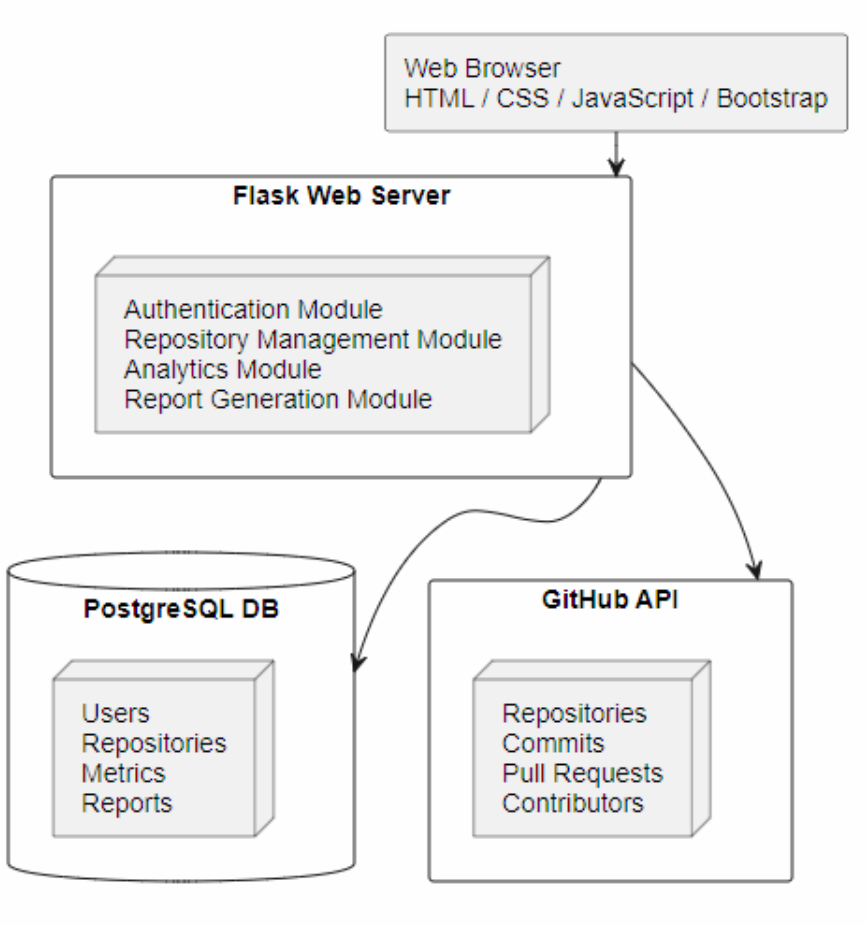

Рисунок 2.2. Структурна схема архітектури веб-додатку

2.6. Проєктування функціональної структури системи

Для опису функціональної структури системи було використано UML-моделювання.

Основними користувачами системи є:

– адміністратор;

– тім-лід;

– аналітик;

– менеджер проєкту.

Загалом сценарії використання веб-додатку цими акторами є однаковими, тому узагальнимо всіх акторів категорією «користувач»

Діаграму варіантів використання веб-додатку представлено на рис. 2.3.

Рисунок 2.3. Діаграма варіантів використання DTracker

Основними варіантами використання є:

– авторизація користувача;
– підключення репозиторію;
– збір та оновлення даних розробника;
– аналіз його активності;
– побудова графіків і формування PDF-звітів.

Відповідно, структура сценаріїв роботи користувача з веб-додатком представлена на рис. 2.4.

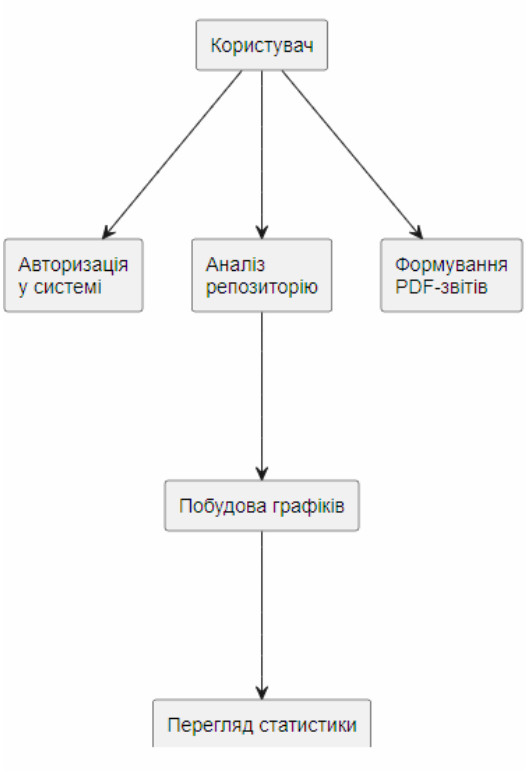

Рисунок 2.4. Структура сценаріїв роботи користувача з веб-додатком

Відповідно до визначеної структури сценаріїв використання та загального архітектурного рішення веб-додатку, побудуємо діаграму компонентів, яка відображає взаємодію основних програмних модулів системи (рис. 2.5).

Найважливіший процес веб-додатку – процес аналізу репозиторію – може бути представлений діаграмою послідовності (рис. 2.6).

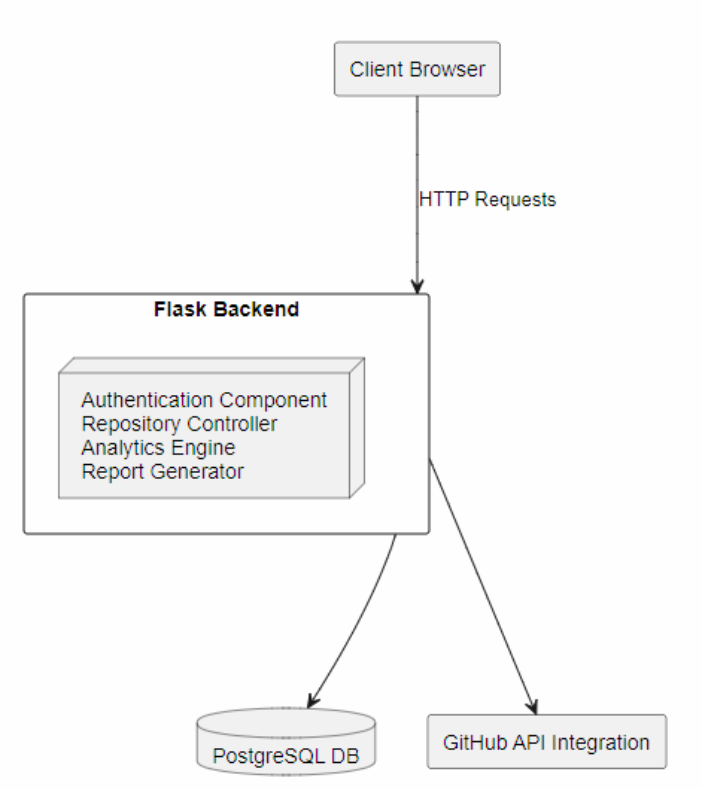

Рисунок 2.5. Діаграма компонентів веб-додатку

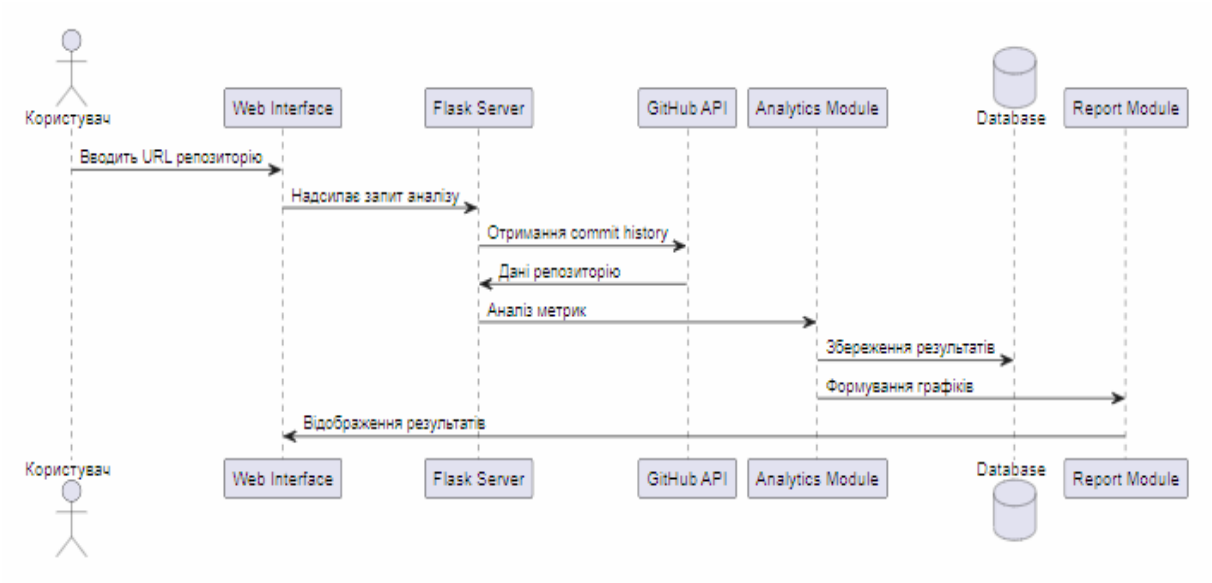

Рисунок 2.6. Діаграма послідовності доступу до репозиторію

2.7. Проєктування бази даних

Для зберігання інформації в системі використовується реляційна база даних PostgreSQL.

Основними сутностями є:

– Users;
– Repositories;
– Contributors;
– Commits;
– Metrics;
– Reports.

База даних повинна:

– забезпечувати швидкий доступ до інформації;
– підтримувати масштабування;
– гарантувати цілісність даних;
– підтримувати збереження історії аналізу.

Опис структури таблиць бази даних представлено в таблицях 2.1-2.3.

Таблиця 2.1. Users – інформація про користувачів системи

Поле	**Тип**	**Опис**
user_id	INTEGER	Ідентифікатор користувача
username	VARCHAR	Логін
email	VARCHAR	Email
password_hash	VARCHAR	Хеш пароля

Таблиця 2.2. Repositories – дані про підключені репозиторії

Поле	**Тип**	**Опис**
repo_id	INTEGER	Ідентифікатор репозиторію
user_id	INTEGER	Власник
repository_name	VARCHAR	Назва
repository_url	TEXT	URL репозиторію

Таблиця 2.3. Commits – інформація про commit activity

Поле	**Тип**	**Опис**
commit_id	INTEGER	Ідентифікатор
repo_id	INTEGER	Репозиторій
contributor_id	INTEGER	Автор
additions	INTEGER	Додані рядки
deletions	INTEGER	Видалені рядки

ER-діаграма бази даних (рис. 2.7) подає взаємозв’язки між сутностями системи.

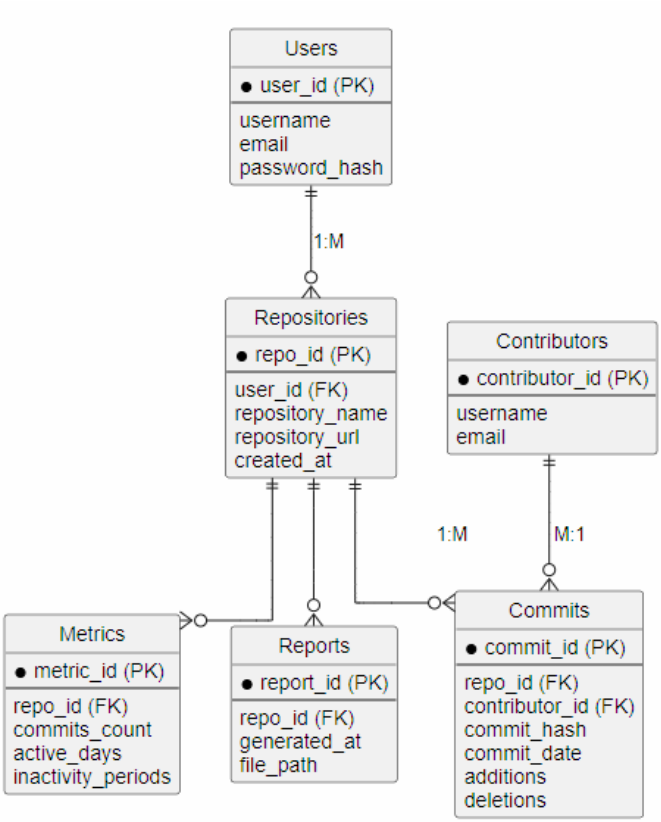

Рисунок 2.7. ER-діаграма бази даних

2.8. Робота з даними

У процесі розробки системи DTracker для збору, опрацювання та відображення аналітики використовується набір спеціалізованих бібліотек Python.

PyGitHub – бібліотека-обгортка для офіційного GitHub REST API. Вона реалізує ключову функцію системи – збір відомостей про активність учасників проєкту. PyGitHub підтримує автентифікацію через персональний токен доступу, що дозволяє працювати як з публічними, так і з приватними репозиторіями.

Pandas – бібліотека для опрацювання структурованих даних, що надає інструменти для фільтрування, сортування, агрегації та групування даних у форматі таблиць DataFrame. Pandas застосовується для збору, фільтрування, групування та агрегації інформації, отриманої з GitHub API. Завдяки структурам DataFrame розробник може зручно маніпулювати великими обсягами даних і готувати їх до відображення.

Plotly – бібліотека для створення інтерактивних графіків та діаграм. На відміну від Matplotlib, Plotly генерує графіки, придатні для безпосереднього відображення у браузері: вони підтримують масштабування, переміщення та виведення підказок при наведенні курсора. Це критично важлива властивість для веб-орієнтованого рішення.

NumPy – розширення мови Python, що додає підтримку великих багатовимірних масивів і матриць разом із бібліотекою математичних функцій. NumPy забезпечує високу продуктивність при числових обчисленнях, що особливо важливо при побудові аналітичних графіків на великих масивах даних.

python-dotenv – бібліотека, що зчитує пари ключ-значення з .env-файлу та завантажує їх як змінні середовища. Це дозволяє зберігати токени доступу та конфігурацію поза кодом, що підвищує безпеку системи.

2.9. Клієнтська частина

Клієнтська частина веб-додатку DTracker реалізована засобами стандартного стеку веб-технологій: HTML5, CSS3 та JavaScript (ES6+). Для прискорення розробки інтерфейсу та забезпечення адаптивності застосовується CSS-фреймворк Bootstrap 5, що надає готові компоненти: навігаційні панелі, картки, модальні вікна, кнопки тощо.

Для реалізації динамічних елементів інтерфейсу без перезавантаження сторінки використовується Fetch API JavaScript, що дозволяє асинхронно звертатися до серверних кінцевих точок та оновлювати вміст сторінки за результатами відповіді. Це забезпечує більш плавний і відзивчивий досвід взаємодії користувача з системою.

Для відображення інтерактивних графіків у браузері застосовується бібліотека Plotly.js (рис. 2.9), яка дозволяє вбудовувати у веб-сторінку різноманітні типи діаграм – лінійні графіки, стовпчасті діаграми, теплові карти тощо – з підтримкою масштабування та інтерактивних підказок.

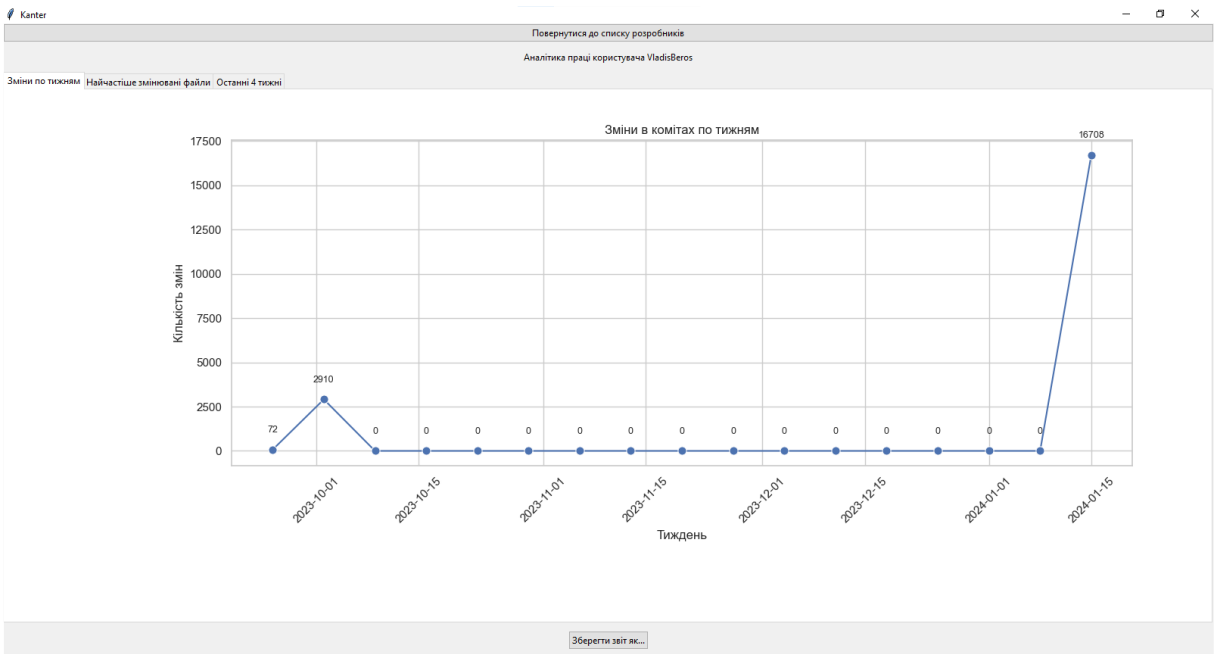

Рисунок 2.9. Графік, побудований за допомогою Plotly

2.9.1. Проєктування інтерфейсу користувача

Інтерфейс системи реалізується за допомогою:

– HTML5;
– CSS3;
– JavaScript;
– Bootstrap 5.

Основними сторінками системи є:

– сторінка авторизації;
– головна панель;
– сторінка репозиторіїв;
– аналітична панель;
– сторінка звітів.

Інтерфейс містить наступні основні елементи:

– навігаційну панель;
– таблиці статистики;
– інтерактивні графіки;
– форми введення;
– модулі фільтрації.

2.9.2. Проєктування модуля аналітики

Модуль аналітики реалізується з використанням бібліотеки Pandas.

Основними функціями модуля є:

– обробка commit history;
– агрегація статистики;
– визначення активності розробників;
– пошук тривалих перерв;
– побудова часових рядів.

Для побудови інтерактивних графіків використовується бібліотека Plotly.

Передбачено побудову:

– гістограм;
– лінійних графіків;
– heatmap активності;
– діаграм внеску учасників.

2.9.3. Проєктування модуля генерації звітів

Модуль генерації звітів забезпечує:

– автоматичне створення PDF-файлів;
– експорт графіків;
– формування статистичних таблиць;
– створення звітів для менеджерів.

До складу звіту входять:

– загальна інформація про репозиторій;
– статистика комітів;
– активність розробників;
– графіки;
– виявлені періоди неактивності.

Висновки до розділу

У другому розділі було виконано проєктування веб-додатку DTracker. Визначено архітектуру системи, описано основні програмні модулі та механізми їх взаємодії.

Було розроблено UML-діаграми варіантів використання, компонентів та послідовності, що дозволило формалізувати структуру та логіку функціонування системи.

Спроєктовано структуру бази даних та ER-діаграму, визначено основні сутності та їх взаємозв’язки. База даних PostgreSQL зберігає дані, отримані через GitHub API, після їх аналізу веб-додатком.

Також було виконано проєктування інтерфейсу користувача, модуля аналітики та підсистеми формування звітів, що створює основу для подальшої програмної реалізації веб-додатку DTracker.

РОЗДІЛ 3. РОЗРОБЛЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ ДЛЯ АНАЛІЗУ ДАНИХ ПРО АКТИВНІСТЬ В СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

3.1. Загальна характеристика процесу розроблення

Після завершення етапу проєктування було виконано програмну реалізацію веб-додатку DTracker, призначеного для аналізу активності розробників у системах контролю версій Git.

Основною метою реалізації було створення:

– масштабованого веб-додатку;
– зручного інтерфейсу користувача;
– механізму автоматичного збору статистики;
– системи побудови інтерактивних візуалізацій;
– підсистеми формування PDF-звітів.

Під час розроблення використовувалися такі технології:

– Python;
– Flask;
– PyGitHub;
– Pandas;
– Plotly;
– PostgreSQL;
– HTML5;
– CSS3;
– JavaScript;
– Bootstrap 5.

3.2. Реалізація архітектури веб-додатку

Модульна архітектура веб-додатку DTracker (рис. 3.1) реалізована за класичною клієнт-серверною моделлю з розділенням на три логічні рівні:

рівень представлення (клієнтська частина), рівень бізнес-логіки (серверна частина) та рівень доступу до даних (інтеграція з зовнішніми API).

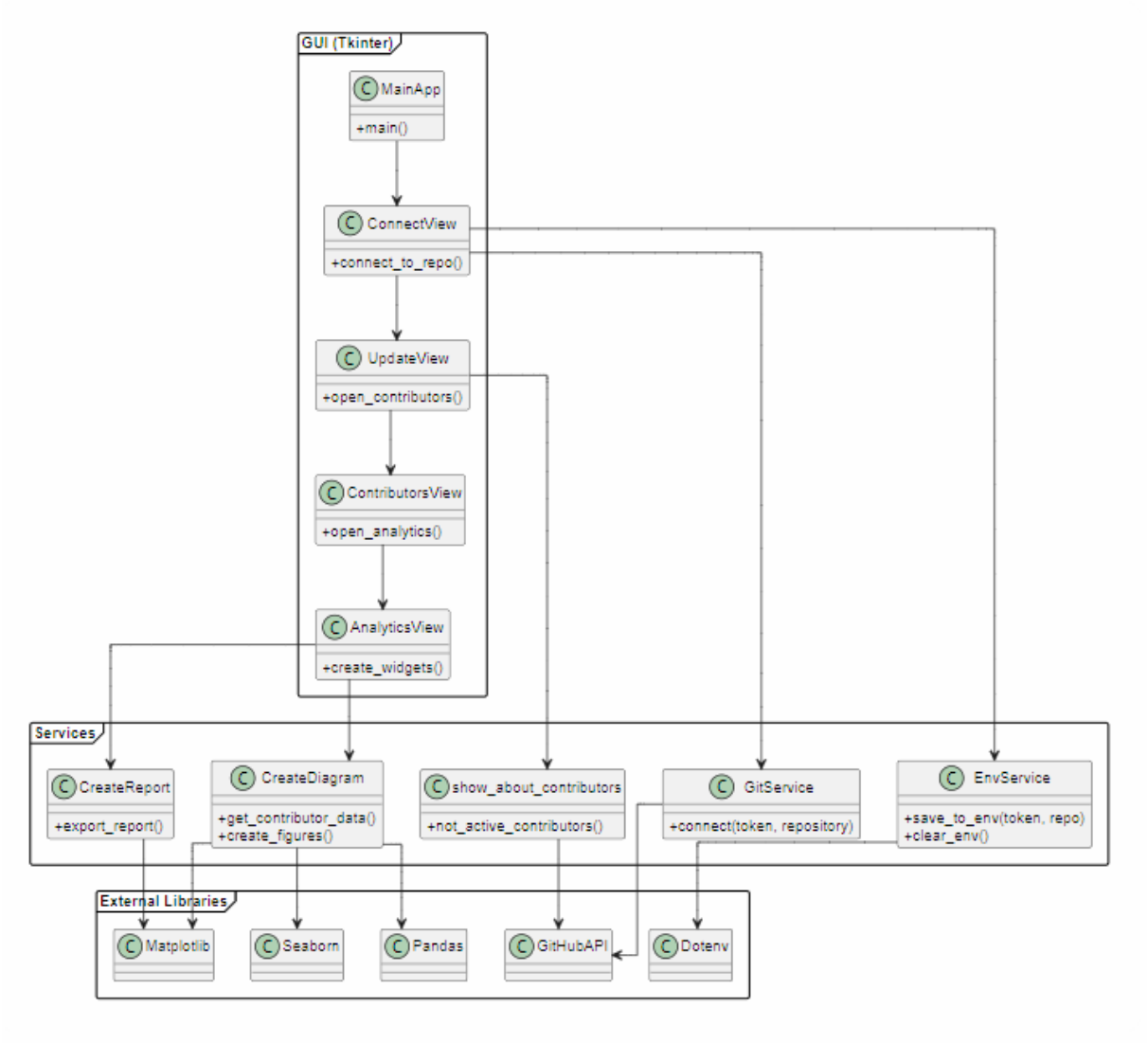

Рисунок 3.1. Модульна архітектура системи

Серверна частина побудована на базі фреймворку Flask і містить модулі для: автентифікації та зберігання токена доступу, взаємодії з GitHub API через PyGitHub, опрацювання та агрегації даних засобами Pandas, генерації інтерактивних графіків через Plotly, формування PDF-звітів та обслуговування HTTP-запитів від клієнтської частини.

Клієнтська частина реалізована у вигляді динамічних HTML-сторінок з CSS (Bootstrap 5) та JavaScript. Для взаємодії з сервером без

перезавантаження сторінки застосовується Fetch API. Інтерактивні графіки відображаються безпосередньо в браузері засобами Plotly.js.

Взаємодія з GitHub здійснюється через офіційний REST API. Користувач вводить персональний токен, який зберігається в захищеній сесії та використовується для автентифікації при кожному зверненні до API. Застосунок формує запити до GitHub API, отримує дані у форматі JSON, після чого виконується їхня обробка та фільтрація.

Таким чином, реалізована архітектура DTracker забезпечує чітке розмежування відповідальності між компонентами, що спрощує підтримку та розширення системи. Вся логіка опрацювання даних і взаємодії з API зосереджена на сервері, а клієнтська частина відповідає лише за відображення результатів.

3.3. Реалізація серверної частини системи

Серверна частина реалізована мовою Python із використанням фреймворку Flask.

Основними перевагами Flask є:

– простота інтеграції;
– модульна структура;
– підтримка REST-архітектури;
– гнучкість налаштування;
– висока швидкість розроблення.

3.3.1. Структура проєкту

Структура програмного проєкту представлена на рис. 3.2.

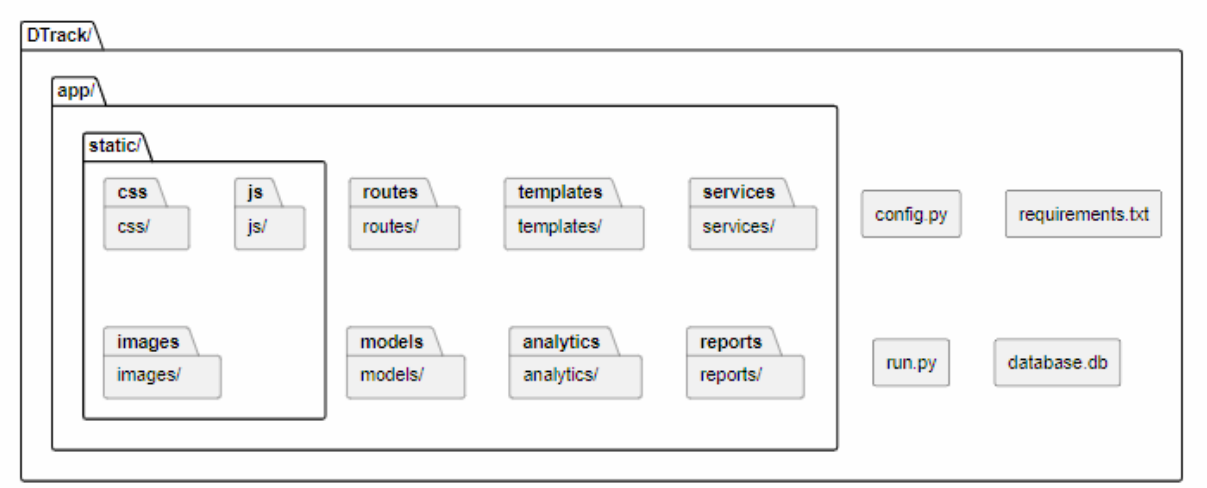

Рисунок 3.2. Структура програмного проєкту

Основні каталоги:

– routes – маршрути Flask;
– models – моделі бази даних;
– services – модулі роботи з GitHub API;
– analytics – модулі обробки статистики;
– reports – генерація PDF-звітів;
– templates – HTML-шаблони;
– static – статичні ресурси.

3.3.2. Структура класів та модулів

Серверна частина системи DTracker організована у вигляді набору спеціалізованих модулів, кожен з яких відповідає за окрему функціональність:

– GitService – центральний сервіс для взаємодії з GitHub API. Здійснює підключення до репозиторію за токеном, кешування результатів підключення та надання об'єкта репозиторію для подальшого опрацювання.

– AnalyticsService – сервіс для збору та агрегації аналітичних даних про активність конкретного учасника репозиторію.

– DiagramService – сервіс для побудови графіків і діаграм на основі зібраних даних із використанням Plotly.

– ReportService – сервіс для генерації та експорту PDF-звітів.

– ContributorService – сервіс для перевірки активності учасників та виявлення тих, хто тривалий час не здійснював комітів.

Клієнтська частина організована у вигляді шаблонів Jinja2 та статичних файлів:

– connect.html – сторінка введення токена та ідентифікатора репозиторію.

– dashboard.html – головна панель із кнопками оновлення даних та списком учасників.

– analytics.html – сторінка детальної аналітики конкретного учасника з вбудованими графіками.

– static/js/charts.js – клієнтський скрипт для ініціалізації та оновлення графіків Plotly.

Структура класів та модулів представлена на діаграмі класів (рис. 3.3).

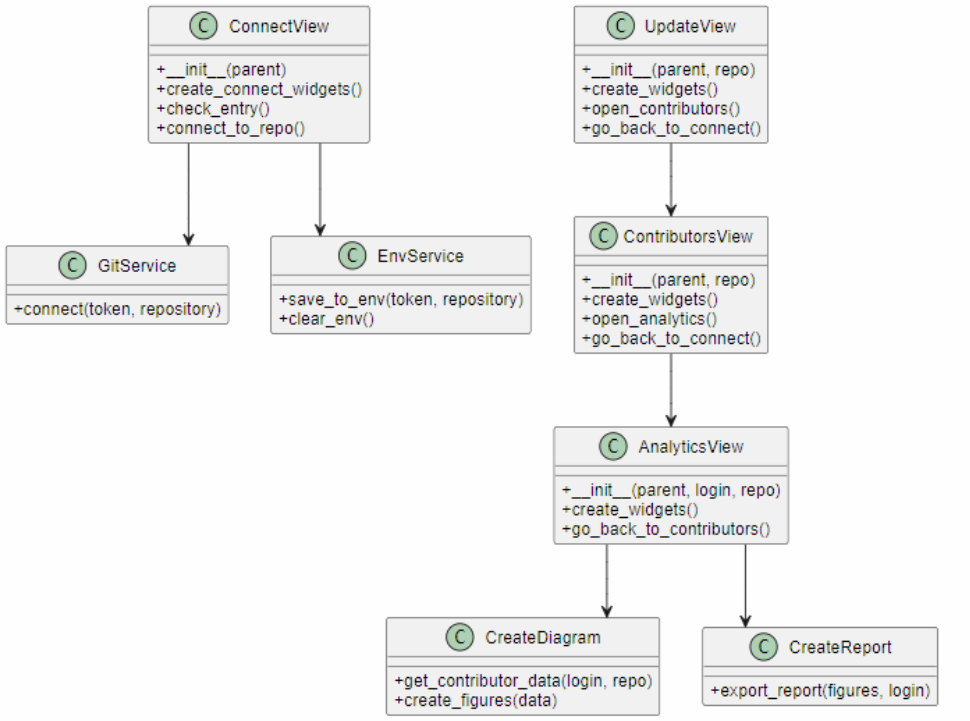

Рисунок 3.3. Діаграма класів веб-додатку

3.3.3. Реалізація взаємодії з GitHub API

Модуль GitService є основним сервісом системи, що відповідає за взаємодію з GitHub API через бібліотеку PyGitHub. Він забезпечує автентифікацію за допомогою персонального токена доступу, отримання об'єкта репозиторію та кешування підключень для оптимізації продуктивності.

Для мінімізації кількості звернень до GitHub API при повторних запитах застосовується кешування результатів підключення з використанням декоратора lru_cache з модуля functools.

Метод connect() приймає токен та ідентифікатор репозиторію, повертаючи словник із результатом операції: статусом успіху, повідомленням та об'єктом репозиторію. Усі виняткові ситуації перехоплюються та повертаються у зрозумілому для клієнта форматі.

Після отримання об'єкта репозиторію він передається до інших сервісів: AnalyticsService для отримання статистики учасників, ContributorService для перевірки активності та DiagramService для побудови візуалізацій.

Основними задачами модуля є:

– авторизація через GitHub Token;
– отримання інформації про репозиторій;
– збір commit history;
– аналіз pull request;
– отримання списку contributors.

Приклад підключення до GitHub API представлено в лістингу 3.1.

Лістинг 3.1. Приклад підключення до GitHub API

```
from github import Github
token = "GITHUB_TOKEN"
g = Github(token)
repo = g.get_repo("user/repository")
commits = repo.get_commits()
for commit in commits:
```

```
print(commit.commit.author.name)
```

Відповідний клас ConnectView підключення до GitHub API представлено на рис. 3.4. На рис. 3.5 та 3.6 передставлено відповідно код методу create_connect_widgets() та методу connect_to_repo() для підключення до репозиторію GitHub.

```
class ConnectView(ttk.Frame):  6 usages
    def __init__(self, parent):  Vlad
        super().__init__(parent)
        self.parent = parent
        self.create_connect_widgets()
```

Рисунок 3.4. Клас ConnectView

```
def create_connect_widgets(self):  1 usage  Vlad +1 *
    load_dotenv()

    style = ttk.Style()
    style.configure( style: "Large.TButton", font=("Arial", 16))

    label_token = ttk.Label(self, text="GIT токен", font=("Arial", 16))
    label_token.pack(pady=(20, 0))
    self.token_entry = ttk.Entry(self, width=50, font=("Arial", 16), show="•")
    self.token_entry.pack()
    self.token_entry.bind("<KeyRelease>", self.check_entry)

    label_repo = ttk.Label(self, text="Репозиторій", font=("Arial", 16))
    label_repo.pack(pady=(10, 0))
    self.repo_entry = ttk.Entry(self, width=50, font=("Arial", 16))
    self.repo_entry.pack()
    self.repo_entry.bind("<KeyRelease>", self.check_entry)

    repo_name = os.getenv("REPO_NAME", "")
    git_token = os.getenv("GIT_TOKEN", "")
    self.repo_entry.insert( index: 0, repo_name)
    self.token_entry.insert( index: 0, git_token)

    self.connect_button = ttk.Button(self,
                                     text="Підключитися до Git-репозиторію",
                                     style="Large.TButton",
                                     command=self.connect_to_repo)
    self.connect_button.pack(pady=(10, 0))
```

Рисунок 3.5. Код методу create_connect_widgets()

```
def connect_to_repo(self):  1 usage  Vlad +1
    token = self.token_entry.get()
    repository = self.repo_entry.get()

    connect_result = GitService.connect(token, repository)

    if connect_result['success']:
        messagebox.showinfo( title: "Успіх",  message: f"{connect_result['message']}")
        EnvService.save_to_env(token, repository)
        repo = connect_result['repo']

        self.destroy()
        update_view = UpdateView(self.parent, repo)
        update_view.pack(fill='both', expand=True)
    else:
        messagebox.showerror( title: "Помилка",  message: f"{connect_result['message']}")
```

Рисунок 3.6. Код методу connect_to_repo() для підключення до репозиторію

Приведений код забезпечує:

– авторизацію у GitHub API;

– підключення до репозиторію;

– отримання історії комітів.

3.3.4. Діаграма станів

Система DTracker функціонує як послідовність переходів між станами, що відповідають етапам роботи користувача з веб-додатком.

Діаграму станів наведено на рис. 3.4. Початковим є стан підключення, в якому система очікує введення облікових даних. Після успішної автентифікації система переходить у стан оновлення, де доступне оновлення даних та перехід до списку учасників.

При виборі конкретного учасника система переходить у стан аналітики, де відображаються графіки та стає доступним експорт звіту. При будь-якому виявленні помилок під час запиту до API система повертається до відповідного попереднього стану та виводить повідомлення про помилку.

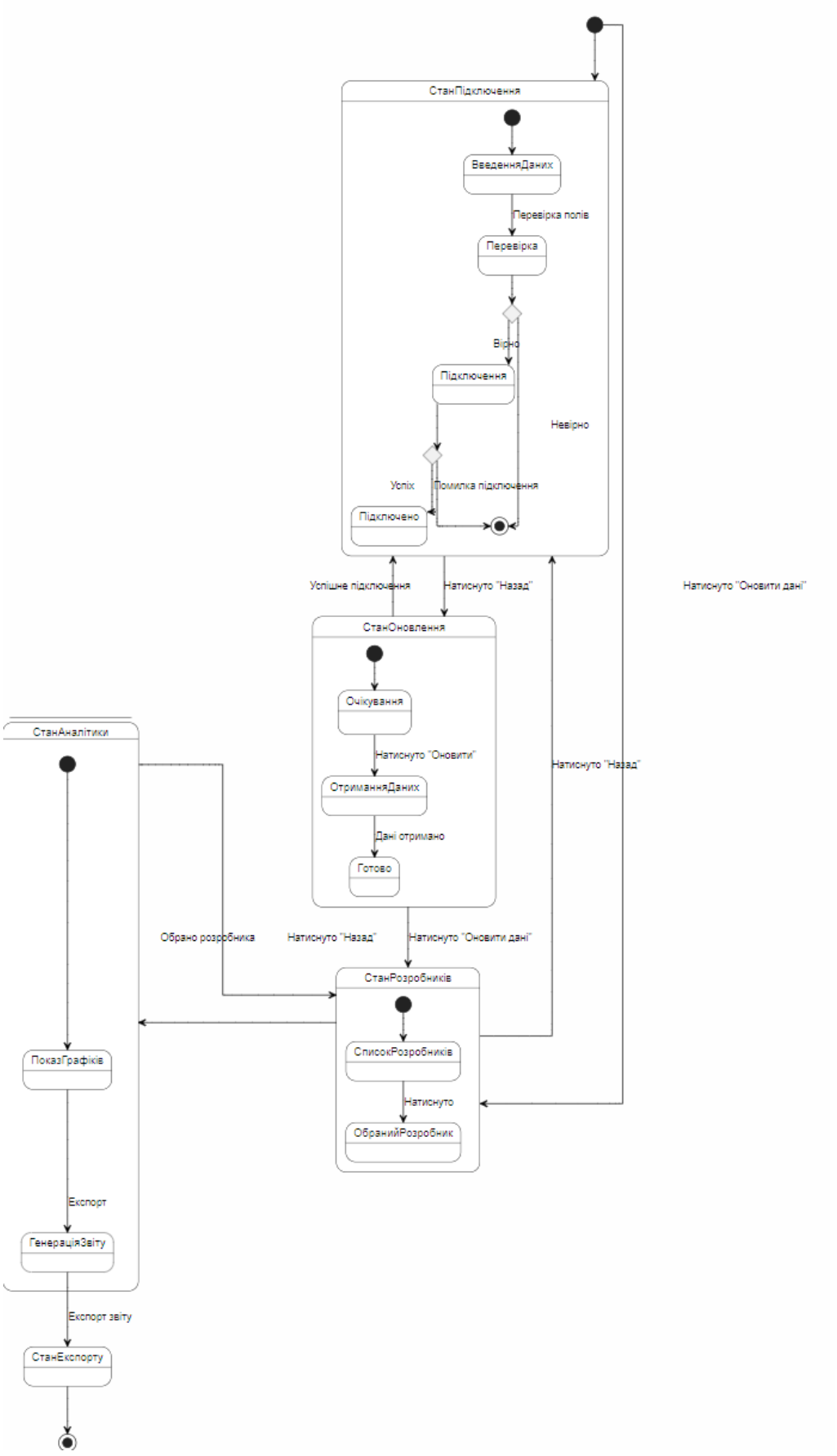

Рисунок 3.4. Діаграма станів веб-додатку

3.4. Реалізація моделі бази даних

Для роботи з базою даних використовувалася ORM-бібліотека SQLAlchemy.

Основними моделями системи є:

– User;
– Repository;
– Contributor;
– Commit;
– Metric;
– Report.

Реалізація моделі Repository представлена в лістингу 3.2.

Лістинг 3.2. Реалізація моделі Repository

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
class Repository(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    url = db.Column(db.String(500))
    created_at = db.Column(db.DateTime)
```

Перевагами використання ORM є:

– автоматичне створення SQL-запитів;
– спрощення взаємодії з БД;
– підвищення переносимості системи.

3.5. Реалізація модуля аналітики

AnalyticsService відповідає за збір і структурування статистики про активність конкретного учасника репозиторію.

Метод get_contributor_data() приймає логін розробника та об'єкт репозиторію, послідовно аналізує всі гілки репозиторію для запобігання втраті комітів і повертає зведений словник метрик.

Для уникнення повторного аналізу одних і тих самих комітів (присутніх у кількох гілках) використовується множина seen_shas, до якої додається SHA кожного коміту при першій обробці. У процесі збору даних фіксуються: загальна кількість комітів, кількість доданих і видалених рядків коду, найчастіше змінювані файли (топ-5), часова динаміка активності по датах та середня кількість змін на коміт.

Зібрані дані структуруються у форматі, зручному для подальшої передачі до сервісу побудови графіків. Усі звернення до зовнішнього API обгортаються в блок обробки виключень, що дозволяє системі продовжувати роботу навіть у разі збоїв при зверненні до окремих гілок.

Аналітичний модуль реалізовано за допомогою бібліотеки Pandas.

Основними задачами модуля є:

– обробка commit history;
– агрегація статистики;
– аналіз активності користувачів;
– визначення періодів неактивності;
– формування таблиць статистики.

Аналіз commit activity здійснюється, як представлено в лістингу 3.3.

Лістинг 3.3. Аналіз commit activity

```
import pandas as pd
data = pd.DataFrame(commits_data)
commits_per_user = data.groupby("author").size()
print(commits_per_user)
```

Відповідний клас AnalyticsView представлено на рис. 3.5.

Код методу get_contributor_data() представлено на рис. 3.6 (частина 1) та рис. 3.7 (частина 2).

```
class AnalyticsView(ttk.Frame):  2 usages  Vla
    def __init__(self, parent, login, repo):
        super().__init__(parent)
        self.parent = parent
        self.login = login
        self.repo = repo
        self.create_widgets()
```

Рисунок 3.5. Клас AnalyticsView

```
@staticmethod  1 usage  Vlad
def get_contributor_data(login, repo: Repository.Repository):
    data = {
        'total_commits': 0,
        'total_lines_added': 0,
        'total_lines_deleted': 0,
        'total_files_changed': 0,
        'avg_lines_per_commit': 0,
        'first_commit_date': None,
        'last_commit_date': None,
        'commit_frequency': None,
        'most_changed_files': {},
        'commits_by_date': []
    }

    seen_shas = set()
    all_commits = []

    branches = list(repo.get_branches())

    for branch in branches:
        try:
            commits = repo.get_commits(author=login, sha=branch.name)
            for commit in commits:
                if commit.sha not in seen_shas:
                    seen_shas.add(commit.sha)
                    all_commits.append(commit)
        except Exception as e:
            print(f"[ERROR] Помилка при обробці гілки '{branch.name}': {e}")

    if not all_commits:
        return data

    all_commits.sort(key=lambda c: c.commit.author.date, reverse=True)
```

Рисунок 3.6. Код методу get_contributor_data() (частина 1)

```
data['total_commits'] = len(all_commits)
data['first_commit_date'] = all_commits[-1].commit.author.date
data['last_commit_date'] = all_commits[0].commit.author.date

for commit in all_commits:
    stats = commit.stats
    commit_date = commit.commit.author.date.date()

    data['commits_by_date'].append({
        'date': commit_date,
        'additions': stats.additions,
        'deletions': stats.deletions,
        'changes': stats.additions + stats.deletions
    })

    data['total_lines_added'] += stats.additions
    data['total_lines_deleted'] += stats.deletions

    for file in commit.files:
        filename = file.filename
        if filename not in data['most_changed_files']:
            data['most_changed_files'][filename] = 0
        data['most_changed_files'][filename] += 1
        data['total_files_changed'] += 1

data['avg_lines_per_commit'] = round(
    (data['total_lines_added'] + data['total_lines_deleted']) / data['total_commits'], 2
)
days_active = (data['last_commit_date'] - data['first_commit_date']).days
data['commit_frequency'] = round(data['total_commits'] / max(1, days_active), 2)
data['most_changed_files'] = dict(
    sorted(data['most_changed_files'].items(), key=lambda item: item[1], reverse=True)[:5]
)

return data
```

Рисунок 3.7. Код методу get_contributor_data() (частина 2)

Приведений код дозволяє:

– групувати коміти;

– обчислювати активність учасників;

– будувати статистичні показники.

3.6. Реалізація механізму виявлення перерв у роботі

Однією з ключових функцій системи є виявлення тривалих перерв у роботі розробників.

ContributorService забезпечує функцію автоматичного виявлення неактивних учасників репозиторію.

Функція get_inactive_contributors() отримує список усіх учасників через repo.get_contributors(), визначає часову межу (7 днів від поточного моменту) та для кожного учасника виконує запит до GitHub API на наявність комітів за цей період.

Учасники, від яких протягом тижня не надходило жодного коміту, додаються до списку неактивних і повертаються функцією для подальшого відображення у вигляді попередження в інтерфейсі.

Кожне звернення до API обгортається в блок обробки виключень з виведенням повідомлення про помилку до журналу – це дозволяє системі продовжувати роботу навіть у разі збоїв при зверненні до даних окремих учасників.

Алгоритм роботи механізму виявлення перерв у роботі:

1. Отримання часових міток комітів.

2. Сортування за датою.

3. Обчислення часових інтервалів.

4. Виявлення перевищення допустимого порогу.

Реалізація алгоритму представлена в лістингу 3.4.

Лістинг 3.4. Реалізація алгоритму виявлення перерв у роботі

```
from datetime import datetime
dates = sorted(commit_dates)
for i in range(1, len(dates)):
    delta = dates[i] - dates[i - 1]
    if delta.days > 7:
        print("Detected inactivity period")
```

Код функції not_active_contributors(), яка реалізує відповідний алгоритм, приведено на рис. 3.8.

```python
import datetime
from tkinter import messagebox

def not_active_contributors(repo):  2 usages  VladisBeros
    contributors = list(repo.get_contributors())
    inactive_logins = []
    one_week_ago = datetime.datetime.utcnow() - datetime.timedelta(days=7)

    for contributor in contributors:
        try:
            commits = list(repo.get_commits(author=contributor.login, since=one_week_ago))
            if not commits:
                inactive_logins.append(contributor.login)
        except Exception as e:
            print(f"[ERROR] Помилка при обробці користувача {contributor.login}: {e}")

    if inactive_logins:
        inactive_list = "\n".join(inactive_logins)
        messagebox.showwarning(
            title: "Неактивні розробники",
            message: f"Протягом останнього тижня не було комітів від наступних розробників:\n\n{inactive_list}"
        )
```

Рисунок 3.8. Код функції not_active_contributors()

Наведений код дозволяє автоматично:

– визначати періоди неактивності;

– попереджати менеджерів;

– аналізувати ризики затримки проєкту.

3.7. Реалізація інтерактивної візуалізації

DiagramService перетворює зібрані аналітичні дані на набір інтерактивних графіків, готових для відображення у браузері.

Метод create_figures() приймає словник даних і повертає словник об'єктів графіків Plotly, де ключем є назва відповідного графіка.

Перший графік – динаміка змін по тижнях – будується як лінійний графік (scatter plot) на основі агрегованих по тижнях даних про кількість змін у комітах. Агрегація виконується засобами Pandas з використанням

групувальника Grouper з тижневою частотою. Над кожною точкою графіка відображається числова позначка з фактичним значенням.

Другий графік – найчастіше змінювані файли – відображає горизонтальну стовпчасту діаграму (horizontal bar chart) топ-5 файлів за кількістю правок. Якщо дані про файли відсутні, відображається інформаційне повідомлення.

Третій графік – зміни за останні чотири тижні – являє собою групову стовпчасту діаграму (grouped bar chart), що порівнює кількість доданих (зелений колір) та видалених (червоний колір) рядків коду по тижнях. Дані фільтруються відносно поточної дати за допомогою Pandas Timedelta.

Для побудови графіків використано бібліотеку Plotly.

Усі графіки генеруються у форматі JSON-опису Plotly, що дозволяє передати їх до клієнтської частини та відобразити в браузері за допомогою Plotly.js без необхідності збереження зображень на сервері.

Система підтримує:

– лінійні графіки;
– гістограми;
– кругові діаграми;
– heatmap активності;
– часові графіки.

Побудова графіка активності представлена в лістингу 3.5.

Лістинг 3.5. Побудова графіка активності

```
import plotly.express as px
fig = px.line(
    data_frame=data,
    x="date",
    y="commits",
    title="Commit Activity"
)
fig.show()
```

Схема побудови візуалізацій представлена на рис. 3.9.

На рис. 3.10, 3.11, 3.12 зображено відповідно алгоритми побудови лінійного графіку, створення діаграми найчастіше змінюваних файлів, створення діаграми змін за останні 4 тижні.

Інтерактивні графіки дозволяють:

– аналізувати динаміку активності;
– оцінювати навантаження команди;
– виявляти аномалії.

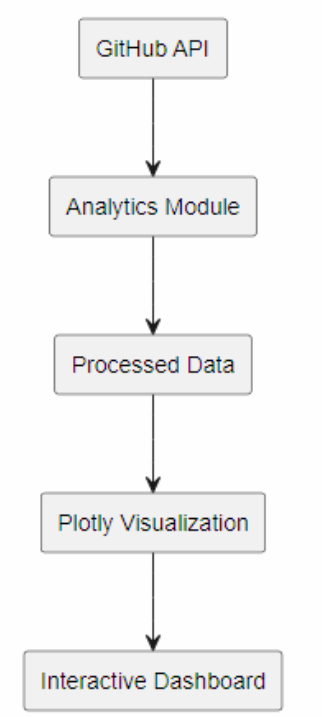

Рисунок 3.9. Схема побудови візуалізацій

```
first_diagram = Figure(figsize=(5, 4))
first_axis = first_diagram.subplots()

sns.lineplot(data=df_weekly, x='date', y='changes', marker='o', markersize=8, ax=first_axis)

for i, row in df_weekly.iterrows():
    first_axis.text(row['date'], row['changes'] + max(df_weekly['changes']) * 0.05,
        s: f"{int(row['changes'])}", ha='center', va='bottom', fontsize=9
    )

first_axis.set_title('Зміни в комітах по тижням')
first_axis.set_xlabel('Тиждень')
first_axis.set_ylabel('Кількість змін')
first_axis.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
first_axis.tick_params(axis='x', rotation=45)
first_diagram.tight_layout()
figures['Зміни по тижням'] = first_diagram
```

Рисунок 3.10. Алгоритм побудови лінійного графіку

```
if data['most_changed_files']:
    second_diagram = Figure(figsize=(5, 4))
    second_axis = second_diagram.subplots()

    files = list(data['most_changed_files'].keys())
    changes = list(data['most_changed_files'].values())

    bars = second_axis.barh(files, changes, color=sns.color_palette("Blues_d"))
    second_axis.set_title('Найчастіше змінювані файли')
    second_axis.set_xlabel('Кількість змін')

    for bar in bars:
        width = bar.get_width()
        second_axis.text(width + max(changes) * 0.01, bar.get_y() + bar.get_height() / 2,
                s: f'{int(width)}', va='center')

    second_diagram.tight_layout()
    figures['Найчастіше змінювані файли'] = second_diagram
else:
    second_diagram = Figure(figsize=(5, 4))
    second_axis = second_diagram.subplots()
    second_axis.text( x: 0.5,  y: 0.5,  s: 'Немає даних про зміни файлів', ha='center', va='center')
    figures['Найчастіше змінювані файли'] = second_diagram
```

Рисунок 3.11. Алгоритм створення діаграми найчастіше змінюваних файлів

```
third_diagram = Figure(figsize=(5, 4))
third_axis = third_diagram.subplots()

last_4_weeks = df[df['date'] >= (pd.to_datetime('today') - pd.Timedelta(weeks=4))]

if not last_4_weeks.empty:
    weekly_data = last_4_weeks.groupby(
        pd.Grouper(key='date', freq='W-MON')
    ).agg({
        'additions': 'sum',
        'deletions': 'sum'
    }).reset_index()

    weekly_data['week_str'] = weekly_data['date'].dt.strftime('%Y-%m-%d')

    bar_width = 0.35
    x = np.arange(len(weekly_data))

    third_axis.bar(x - bar_width / 2, weekly_data['additions'], width=bar_width, color='green', label='Додані рядки')
    third_axis.bar(x + bar_width / 2, weekly_data['deletions'], width=bar_width, color='red', label='Видалені рядки')

    third_axis.set_title('Зміни за останні 4 тижні')
    third_axis.set_xticks(x)
    third_axis.set_xticklabels(weekly_data['week_str'], rotation=45)
    third_axis.legend()
else:
    third_axis.text( x: 0.5,  y: 0.5,  s: 'Немає даних за останні 4 тижні', ha='center', va='center')

third_diagram.tight_layout()
figures['Останні 4 тижні'] = third_diagram

return figures
```

Рисунок 3.12. Алгоритм створення діаграми змін за останні 4 тижні

3.8. Реалізація клієнтської частини

Клієнтська частина побудована з використанням:

– HTML5;

– CSS3;

– JavaScript;

– Bootstrap 5.

Bootstrap забезпечує:

– адаптивність інтерфейсу;

– підтримку мобільних пристроїв;

– сучасний зовнішній вигляд.

Основні сторінки системи

До складу веб-інтерфейсу входять:

– сторінка авторизації;
– dashboard;
– сторінка репозиторіїв;
– сторінка аналітики;
– сторінка звітів.

Приклад HTML-шаблону представлено в лістингу 3.6.

Лістинг 3.6. Приклад HTML-шаблону

```
<div class="container">
    <h1>DTracker Dashboard</h1>
    <div class="row">
        <div class="col-md-6">
            <div id="chart"></div>
        </div>
    </div>
</div>
```

3.8.1. Реалізація системи авторизації

Для забезпечення безпеки системи реалізовано механізм авторизації користувачів.

Функціональні можливості:

– реєстрація;
– авторизація;
– вихід із системи;
– хешування паролів;
– керування сесіями.

Реалізація хешування пароля представлена в лістингу 3.7.

Лістинг 3.7. Реалізація хешування пароля

```
from werkzeug.security import generate_password_hash
password_hash = generate_password_hash(password)
```

Даний підхід забезпечує:

– захист облікових записів;

– безпечне зберігання паролів;
– підвищення рівня кібербезпеки системи.

3.8.2. Реалізація модуля формування PDF-звітів

ReportService відповідає за побудову та збереження PDF-звітів про активність розробника.

Метод generate_report() приймає словник графіків Plotly та логін розробника, після чого генерує PDF-документ із титульною сторінкою та набором сторінок із графіками.

Для перетворення графіків Plotly у растрові зображення перед їхнім включенням до PDF застосовується бібліотека kaleido.

Кожен графік зберігається у тимчасовий буфер у форматі PNG, після чого вставляється до PDF-документа засобами бібліотеки reportlab. Титульна сторінка містить ім'я учасника та дату формування звіту. Після завершення формування документ передається на завантаження клієнту через HTTP-відповідь Flask без збереження на диску.

Для створення звітів використано бібліотеку ReportLab.

Звіт містить:

– статистику репозиторію;
– таблиці активності;
– графіки;
– аналітичні висновки.

Генерація PDF-документа здійснюється, як представлено в лістингу 3.8.

Лістинг 3.8. Генерація PDF-документа

```
from reportlab.pdfgen import canvas
c = canvas.Canvas("report.pdf")
c.drawString(100, 750, "DTracker Report")
c.save()
```

Структура PDF-звіту наведена на рис. 3.13.

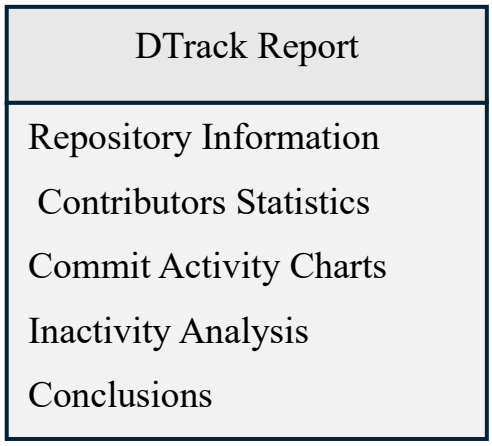

Рисунок 3.13. Алгоритм створення діаграми змін за останні 4 тижні

Реалізація методів CreateReport та export_report() наведена на рис. 3.14.

```
class CreateReport(ttk.Frame):  2 usages  VladisBeros
    @staticmethod  1 usage  VladisBeros
    def export_report(figures, login):
        file_path = filedialog.asksaveasfilename(defaultextension=".pdf", filetypes=[("PDF files", "*.pdf")], title="Зберегти звіт як PDF")
        if not file_path:
            return

        with PdfPages(file_path) as pdf:
            from matplotlib import pyplot as plt

            fig, ax = plt.subplots(figsize=(8.27, 11.69))
            ax.axis('off')
            ax.text( x: 0.5,  y: 0.9,  s: f"Аналіз роботи розробника {login}", fontsize=16, ha='center', va='top', weight='bold')
            pdf.savefig(fig)
            plt.close(fig)

            for fig in figures.values():
                pdf.savefig(fig)

            messagebox.showinfo( title: "Успіх",  message: f"Звіт про {login} успішно збережено.")
```

Рисунок 3.14. CreateReport та статичний метод export_report()

3.8.3. Реалізація REST-маршрутів Flask

Для організації взаємодії між клієнтською та серверною частинами реалізовано REST-маршрути.

Серверна частина веб-додатку надає набір HTTP-кінцевих точок (ендпоінтів), до яких звертається клієнтська частина через Fetch API.

Основні маршрути:

– GET / – повернення стартової сторінки з формою введення токена та репозиторію.

– POST /connect – обробка форми підключення, виклик GitService.connect() та збереження об'єкта репозиторію в сесії.

– POST /update – ініціювання оновлення даних, виклик ContributorService та повернення списку неактивних учасників.

– GET /contributors – повернення JSON-списку учасників репозиторію.

– GET /analytics/<login> – повернення сторінки аналітики для обраного учасника з вбудованими графіками Plotly.

– GET /report/<login> – генерація та повернення PDF-звіту для завантаження.

Для захисту маршрутів, що потребують автентифікації, застосовується декоратор перевірки сесії. Якщо токен або об'єкт репозиторію відсутній у сесії, користувач перенаправляється на стартову сторінку.

Приклад Flask route представлено в лістингу 3.9.

Лістинг 3.9. Приклад Flask route

```
@app.route('/dashboard')
@login_required
def dashboard():
    return render_template('dashboard.html')
```

3.8.4. Реалізація механізму асинхронної обробки

Оскільки аналіз великих репозиторіїв може займати значний час, у системі реалізовано можливість подальшого використання асинхронної обробки задач.

Основні переваги:

– зменшення навантаження на сервер;
– покращення швидкодії;
– підвищення стабільності системи.

Для реалізації асинхронної обробки в процесі подальшого розвитку веб-додатку можуть бути використані наступні засоби:

– Celery;
– Redis;
– Flask Background Tasks.

Висновки до розділу

У третьому розділі було розглянуто процес програмної реалізації веб-додатку DTracker.

Було реалізовано серверну частину системи на базі Flask, модулі інтеграції з GitHub API, аналітичний модуль із використанням бібліотеки Pandas, систему побудови інтерактивних графіків на основі Plotly та підсистему генерації PDF-звітів.

Також було реалізовано клієнтський веб-інтерфейс, механізми авторизації, REST-маршрути та засоби забезпечення інформаційної безпеки.

Отриманий програмний продукт забезпечує автоматизований аналіз активності розробників у Git-середовищах та може використовуватися для підтримки процесів технічного управління програмними проєктами.

У DTracker повністю реалізовано наступні сценарії взаємодії користувача з функціональністю:

1. Підключення до репозиторію: користувач вводить токен доступу та ідентифікатор репозиторію, після чого система встановлює з'єднання з GitHub API та ініціює збір даних.

2. Оновлення даних: користувач ініціює перезбір актуальної статистики з репозиторію. Система перевіряє активність учасників за останній тиждень і відображає список неактивних розробників.

3. Перегляду аналітики учасника: користувач обирає конкретного розробника зі списку учасників репозиторію і переглядає деталізовані графіки його активності.

4. Експорту PDF-звіту: користувач зберігає аналітичні матеріали у форматі PDF для подальшого використання або презентації.

5. Управління обліковими даними: користувач може зберегти токен для автоматичного завантаження при наступному запуску або очистити збережені дані при завершенні роботи.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ ВЕБ-ДОДАТКУ ДЛЯ АНАЛІЗУ ДАНИХ ПРО АКТИВНІСТЬ В СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

4.1. Мета та задачі тестування системи

Після завершення етапу програмної реалізації веб-додатку DTracker було проведено комплексне тестування системи з метою перевірки:

– коректності функціонування;
– стабільності роботи;
– безпеки;
– продуктивності;
– зручності використання.

Основними задачами тестування були:

– перевірка роботи серверної частини;
– тестування взаємодії з GitHub API;
– перевірка аналітичного модуля;
– тестування побудови графіків;
– перевірка формування PDF-звітів;
– оцінювання навантаження на систему;
– перевірка механізмів безпеки.

4.2. Організація процесу тестування

Для тестування системи використовувалися:

– модульне тестування;
– інтеграційне тестування;
– функціональне тестування;
– навантажувальне тестування;
– тестування безпеки;
– користувацьке тестування.

Загальна схема процесу тестування представлена на рис. 4.1.

Рисунок 4.1. Загальна схема процесу тестування

4.3. Модульне тестування

Модульне тестування використовувалося для перевірки окремих компонентів системи.

Тестувалися:

– модулі роботи з GitHub API;
– аналітичний модуль;
– система авторизації;
– генерація звітів;
– робота з базою даних.

Для тестування використовувався фреймворк pytest.

Приклад модульного тесту:

```
def test_repository_connection():
repo = github_service.connect_repository( “user/test_repo” )
assert repo is not None
```

Результати модульного тестування наведені в табл. 4.1.

Таблиця 4.1. Результати модульного тестування

Модуль	Кількість тестів	Успішно	Помилки
GitHub API Module	12	12	0
Analytics Module	15	15	0
Authentication Module	10	10	0
Reports Module	8	8	0
Database Module	11	11	0

В цілому результати тестування підтвердили коректність роботи окремих компонентів системи.

4.4. Інтеграційне тестування

Інтеграційне тестування виконувалося для перевірки взаємодії між модулями системи.

Було перевірено:

– взаємодію Flask із PostgreSQL;

– роботу аналітичного модуля з GitHub API;

– формування звітів на основі отриманих даних;

– передачу даних між серверною та клієнтською частинами.

Схема інтеграційного тестування представлена на рис. 4.2.

У процесі тестування було встановлено, що всі модулі системи коректно взаємодіють між собою.

4.5. Функціональне тестування

Функціональне тестування дозволило перевірити реалізацію всіх функціональних вимог системи.

Основні перевірені функції:

– авторизація користувачів;

– підключення репозиторіїв;
– аналіз commit history;

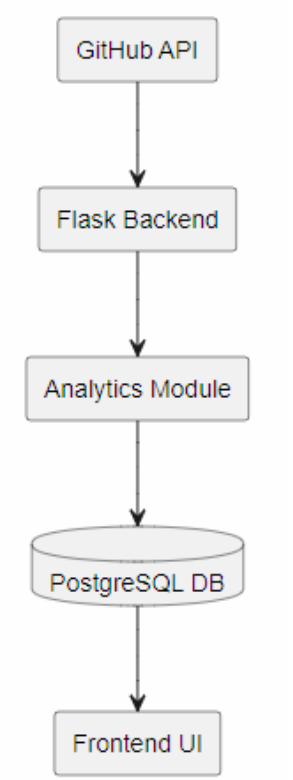

Рисунок 4.2. Схема інтеграційного тестування

– побудова графіків;
– генерація PDF-звітів;
– виявлення періодів неактивності;
– фільтрація статистики.

Результати функціонального тестування наведені в табл. 4.2.

4.6. Навантажувальне тестування

Для оцінювання продуктивності системи було проведено навантажувальне тестування.

Таблиця 4.2. Результати функціонального тестування

Функція	**Очікуваний результат**	**Результат**
Авторизація	Вхід у систему	Успішно
Аналіз репозиторію	Отримання статистики	Успішно
Побудова графіків	Відображення графіків	Успішно
PDF-звіти	Генерація документа	Успішно
Аналіз активності	Формування метрик	Успішно
Виявлення перерв	Відображення попереджень	Успішно

Основними параметрами тестування були:

– кількість одночасних користувачів;

– час відповіді сервера;

– використання оперативної пам’яті;

– навантаження на процесор.

Конфігурація тестового середовища

Тестування проводилося на сервері з такими характеристиками:

– CPU: Intel Core i7;

– RAM: 16 GB;

– SSD: 512 GB;

– ОС: Ubuntu Server.

Результати навантажувального тестування представлені в табл. 4.3.

Таблиця 4.3. Результати навантажувального тестування

Кількість користувачів	**Середній час відповіді**
10	0.4 с
25	0.8 с
50	1.3 с
100	2.1 с

Графік залежності часу відповіді від навантаження приведено на рис. 4.3.

Отримані результати свідчать про достатню продуктивність системи для використання у невеликих та середніх командах розробки.

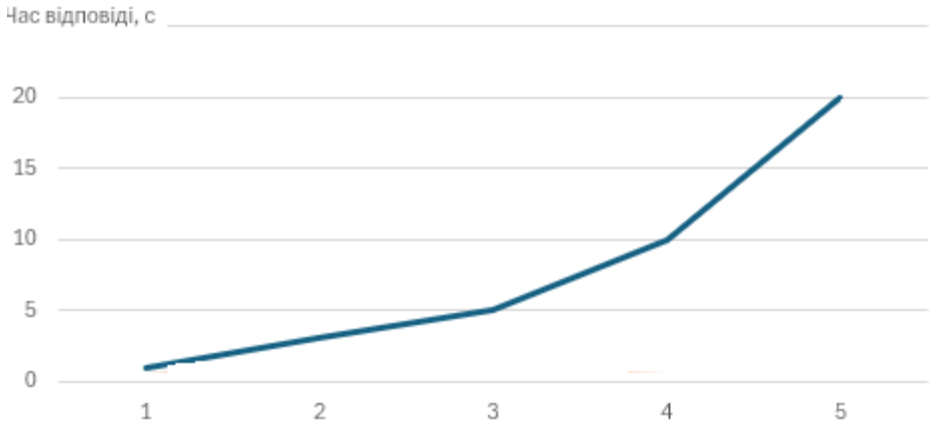

Рисунок 4.3. Графік залежності часу відповіді від навантаження

4.7. Тестування безпеки системи

Особливу увагу було приділено тестуванню інформаційної безпеки веб-додатку.

Було перевірено:

– захист від SQL Injection;
– захист від XSS-атак;
– CSRF-захист;
– захист паролів;
– безпечне зберігання GitHub-токенів.

Приклад перевірки CSRF-захисту: `form.hidden_tag()`.

Результати тестування підтвердили достатній рівень захищеності системи.

4.8. Користувацьке тестування

Для оцінювання зручності використання системи було проведено користувацьке тестування.

До тестування були залучені:

– студенти;

– викладачі;

– розробники програмного забезпечення.

Основними критеріями оцінювання були:

– зручність інтерфейсу;

– швидкість роботи;

– зрозумілість навігації;

– якість візуалізації даних.

Результати користувацького тестування наведено в табл. 4.4.

Таблиця 4.4. Оцінювання системи користувачами

Критерій	**Середня оцінка**
Зручність інтерфейсу	4.8
Швидкість роботи	4.7
Якість графіків	4.9
Зручність звітів	4.6

Користувачі позитивно оцінили функціональні можливості системи та зручність роботи з нею.

4.9. Впровадження веб-додатку

Після завершення тестування було виконано впровадження веб-додатку DTracker.

Основними етапами впровадження були:

1. Налаштування серверного середовища.

2. Розгортання PostgreSQL.

3. Налаштування Flask-сервера.

4. Підключення GitHub API.

5. Налаштування веб-сервера Nginx.

6. Тестування працездатності.

Схему розгортання системи наведено на діаграмі рис. 4.4.

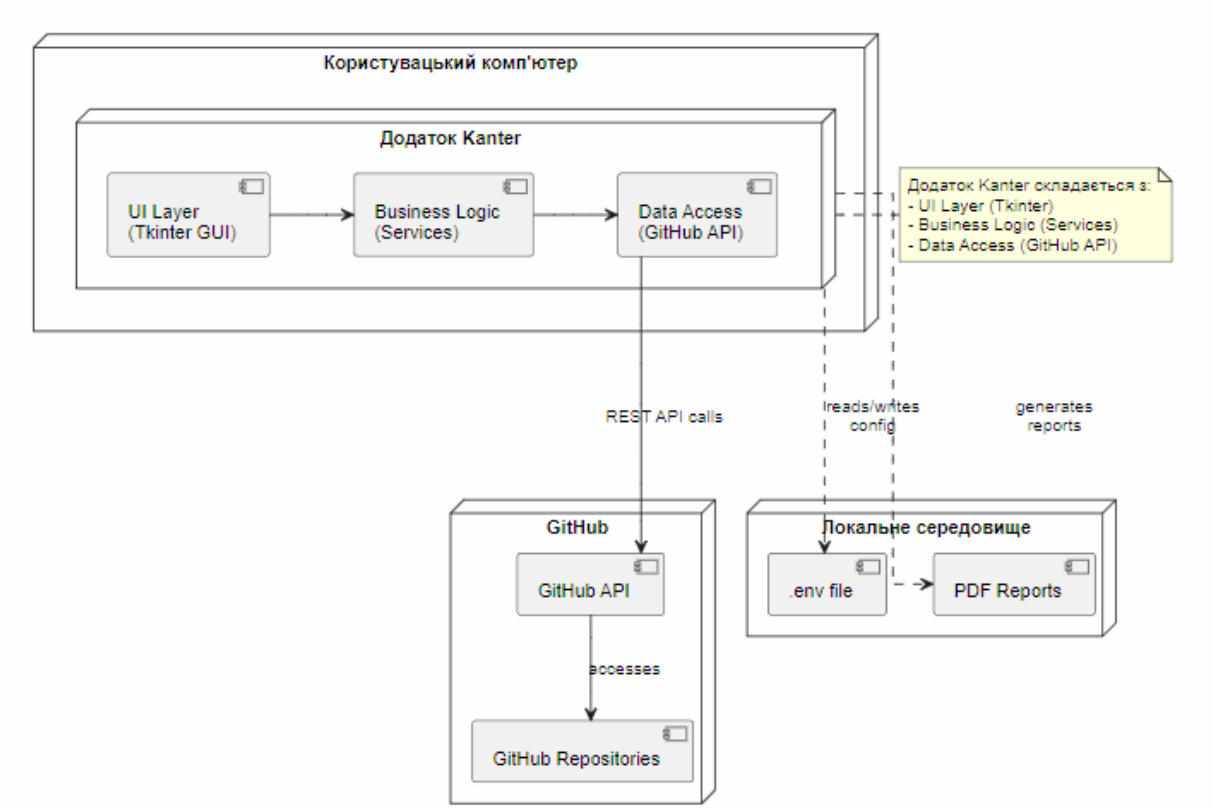

Рисунок 4.4. Діаграма розгортання системи

Діаграму пакетів веб-додатку приведено на рис. 4.5.

4.10. Налаштування серверного середовища

Для розгортання системи використовувалася операційна система Ubuntu Server.

Було встановлено:

– Python 3.11;

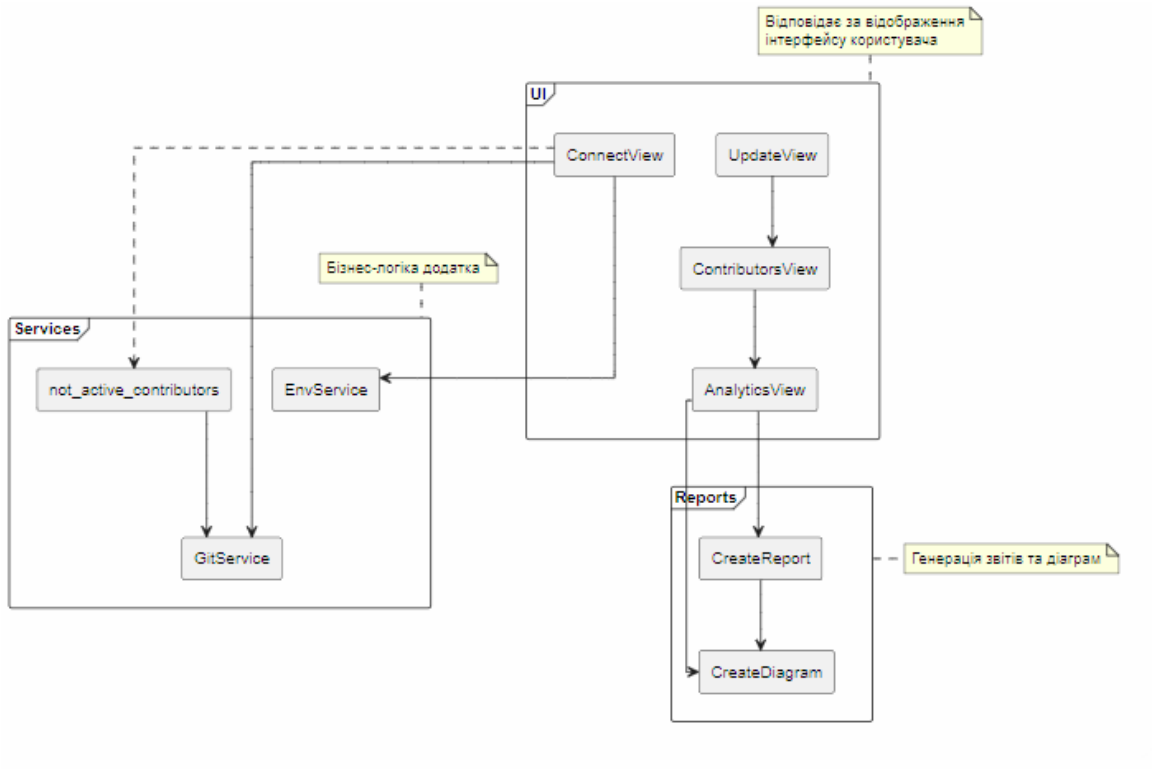

Рисунок 4.5. Діаграма пакетів веб-додатку

– Flask;
– PostgreSQL;
– Nginx;
– Gunicorn.

Приклад запуску Gunicorn представлено в лістингу 4.1.

Лістинг 4.1. Приклад запуску Gunicorn

```
bash id="5qtj4r" gunicorn -w 4 run:app
```

Приклад конфігурації Nginx представлено в лістингу 4.2.

Лістинг 4.2. Приклад конфігурації Nginx

```
server { listen 80;
location / {
    proxy_pass http://127.0.0.1:8000;
   }
} ```
```

4.11. Аналіз результатів впровадження

У результаті впровадження веб-додатку DTracker було отримано:

– автоматизований аналіз активності розробників;
– зручний механізм формування звітності;
– інтерактивну візуалізацію статистики;
– підтримку управління програмними проєктами.

Основними перевагами системи є:

– простота використання;
– модульна архітектура;
– підтримка GitHub API;
– масштабованість;
– сучасний веб-інтерфейс.

Система може бути використана:

– у навчальному процесі;
– в IT-компаніях;
– у невеликих командах розробки;
– для аналітики open-source проєктів.

Головний модуль app.py є точкою входу у веб-додаток DTracker. Він відповідає за ініціалізацію Flask-застосунку, налаштування сесій, реєстрацію маршрутів та запуск вбудованого сервера розробки.

При запуску модуль завантажує змінні середовища з .env-файлу за допомогою python-dotenv, ініціалізує Flask-застосунок із секретним ключем для підпису сесій та реєструє всі маршрути. Для зручності розгортання передбачено підтримку конфігурації через змінні середовища: порт запуску, режим налагодження тощо.

Застосунок може бути запущений як локально командою python app.py, так і розгорнутий на продуктивному сервері через WSGI-інтерфейс (наприклад, за допомогою Gunicorn або uWSGI).

Стартова сторінка веб-додатку DTracker містить форму для введення персонального токена доступу до GitHub API та ідентифікатора репозиторію у форматі 'username/repo' (рис. 4.6). Форма реалізована з використанням компонентів Bootstrap 5 та включає поля введення з відповідними мітками, кнопку підключення та область для виведення повідомлень про помилки.

Рисунок 4.6. Меню підключення до репозиторію

Токен відображається у вигляді прихованих символів для запобігання візуальному перехопленню. Кнопка підключення активна лише тоді, коли обидва поля заповнені – це реалізовано через обробник події input, що відслідковує зміни у полях введення. При відправці форми виконується POST-запит до ендпоінту /connect через Fetch API; у разі успіху користувач перенаправляється на головну панель.

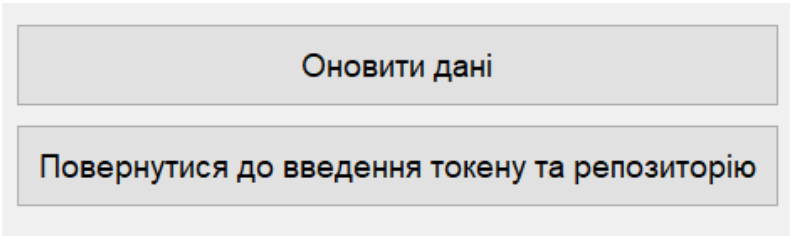

Рисунок 4.7. Віджет UpdateView

Після успішного підключення до репозиторію користувач потрапляє на головну панель системи. Вона містить дві основні кнопки: 'Оновити дані' та 'Відключитися від репозиторію'. При натисканні на 'Оновити дані' виконується асинхронний POST-запит до /update, після чого оновлюється список учасників і, за наявності неактивних розробників, виводиться відповідне попередження.

Список учасників відображається як набір карток Bootstrap з іменами (логінами) розробників. Кожна картка є інтерактивним елементом – при натисканні на неї відбувається перехід до сторінки аналітики відповідного учасника. Для великих команд список підтримує вертикальне прокручування.

Сторінка деталізованої аналітики (рис. 4.8-4.12) відображає три інтерактивні графіки активності обраного учасника, а також надає можливість збереження PDF-звіту. Графіки відображаються у вкладковому інтерфейсі (Bootstrap Tabs): перша вкладка містить лінійний графік змін по тижнях, друга – горизонтальну діаграму найчастіше змінюваних файлів, третя – групову стовпчасту діаграму змін за останні 4 тижні.

Дані для графіків передаються з сервера у вигляді JSON-опису Plotly та ініціалізуються клієнтським скриптом charts.js через виклик Plotly.newPlot(). Це дозволяє використовувати повний набір інтерактивних можливостей Plotly безпосередньо в браузері: масштабування, переміщення, виведення значень при наведенні курсора.

У нижній частині сторінки розміщена кнопка 'Завантажити PDF-звіт', при натисканні на яку виконується GET-запит до /report/<login>, сервер генерує PDF-документ і повертає його як файл для завантаження.

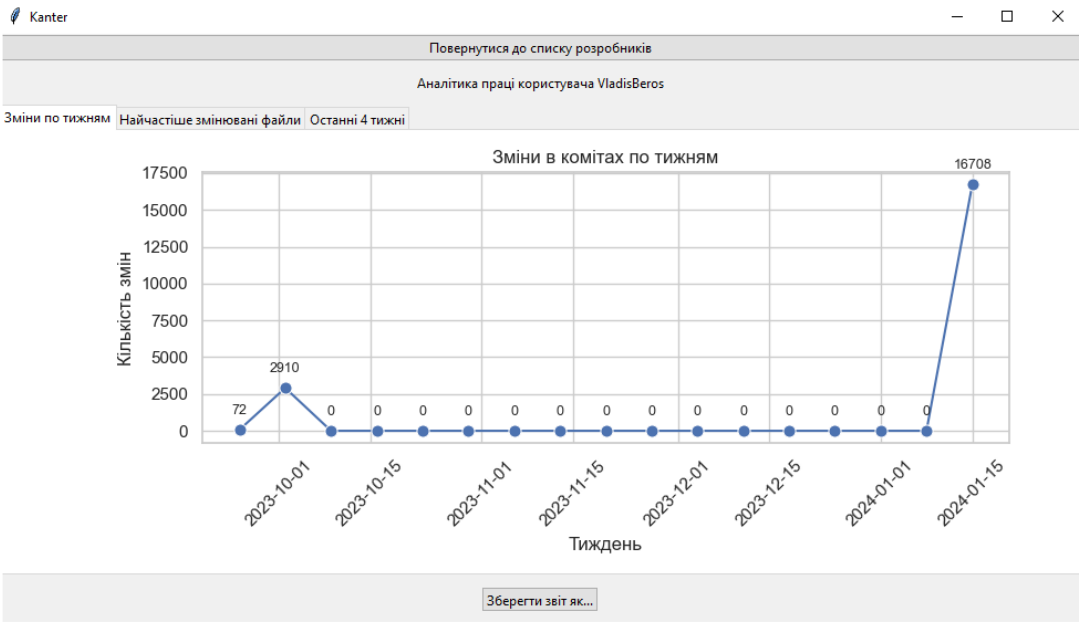

Рисунок 4.8. Віджет з аналітикою AnalyticsView

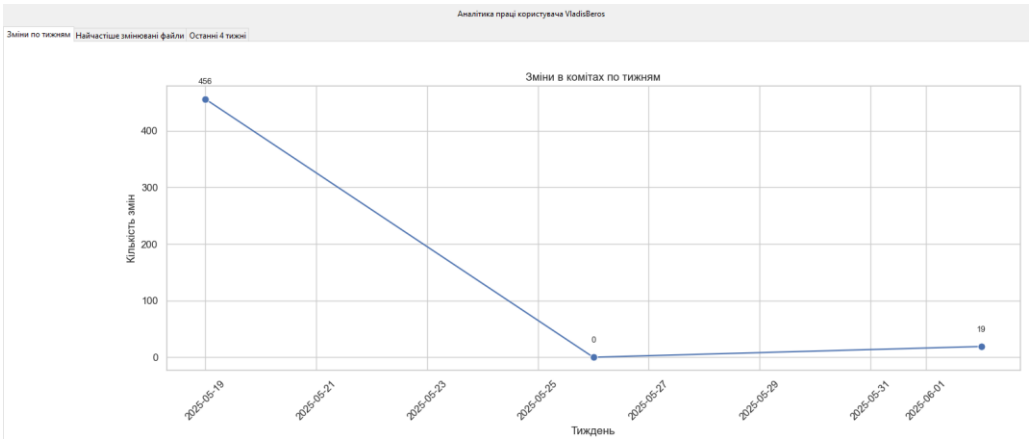

Рисунок 4.9 Лінійний графік

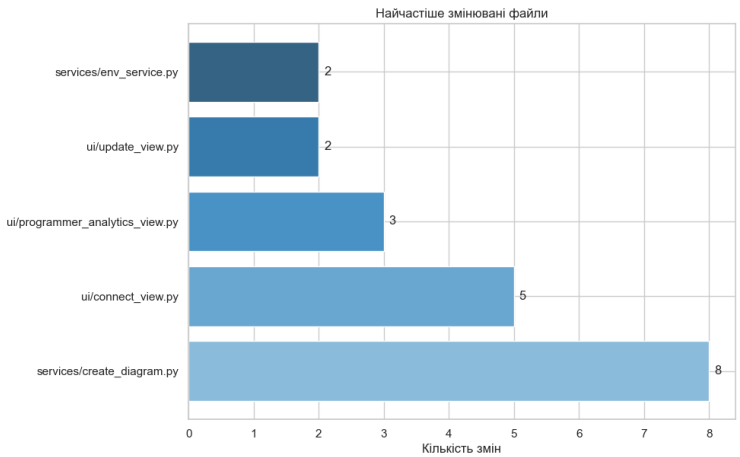

Рисунок 4.10. Приклад найчастіше змінюваних файлів

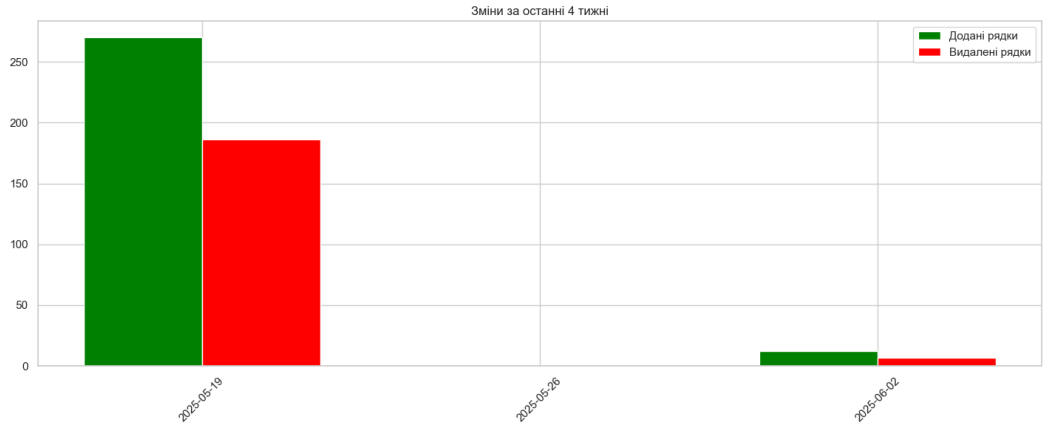

Рисунок 4.11. Приклад аналітики змін за останні 4 тижні

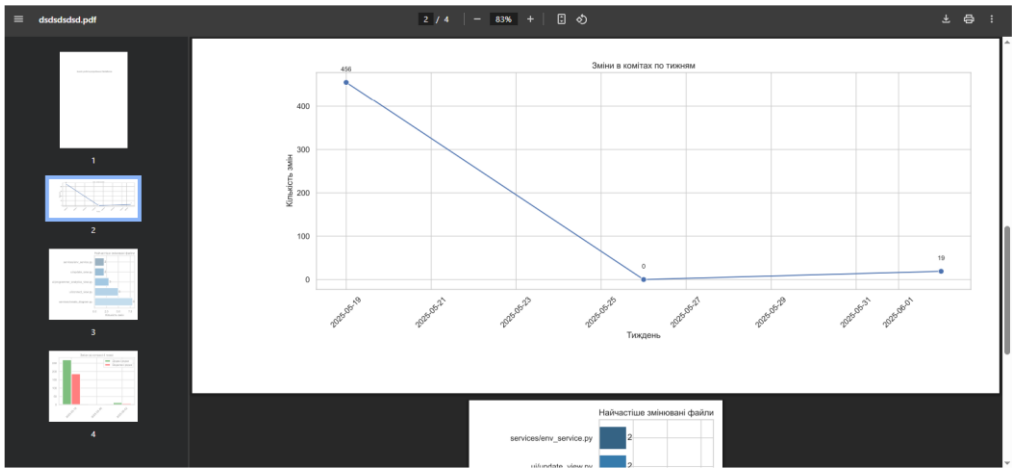

Рисунок 4.12. Приклад збереженого PDF звіту

Висновки до розділу

У четвертому розділі було розглянуто питання тестування та впровадження веб-додатку DTracker.

Було проведено модульне, інтеграційне, функціональне, навантажувальне та користувацьке тестування системи. Результати тестування підтвердили коректність функціонування, достатню продуктивність та стабільність роботи веб-додатку.

Також було виконано розгортання системи у серверному середовищі та перевірено її працездатність у реальних умовах використання.

Отримані результати підтверджують можливість практичного використання веб-додатку DTracker для аналізу активності командної розробки програмного забезпечення.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було розроблено веб-додаток DTracker, призначений для автоматизованого аналізу даних про активність учасників командної розробки у системах контролю версій Git.

У процесі виконання роботи було проведено аналіз предметної області оцінювання ефективності командної розробки програмного забезпечення. Встановлено, що сучасні системи контролю версій є важливим джерелом статистичних даних про процес розробки програмного забезпечення, а аналіз активності розробників дозволяє підвищити ефективність управління програмними проєктами, виявляти потенційні ризики та покращувати координацію роботи команд.

Було досліджено існуючі програмні рішення для моніторингу активності розробників у Git-середовищах, зокрема GitHub Insights, GitLab Analytics, Gource та інші інструменти аналітики. Проведений порівняльний аналіз дозволив визначити їхні основні функціональні можливості, переваги та недоліки. Встановлено, що існуючі системи не повною мірою забезпечують комплексний аналіз активності учасників команд, автоматичне виявлення тривалих перерв у роботі та гнучке формування звітної документації.

На основі виконаного аналізу було сформовано функціональні та нефункціональні вимоги до програмного продукту DTracker. Виконано проєктування архітектури веб-додатку, розроблено UML-діаграми, ER-модель бази даних та структуру взаємодії програмних компонентів системи.

У ході програмної реалізації створено серверну частину веб-додатку з використанням мови програмування Python та фреймворку Flask. Для роботи з GitHub API використано бібліотеку PyGitHub, для обробки статистичних даних – бібліотеку Pandas, а для побудови інтерактивних графіків – бібліотеку Plotly. Клієнтська частина реалізована із застосуванням HTML5, CSS3, JavaScript та Bootstrap 5.

Розроблений веб-додаток забезпечує:

– автоматичне підключення до Git-репозиторіїв;
– збір статистики активності розробників;
– аналіз commit history;
– побудову інтерактивних графіків;
– виявлення періодів неактивності;
– формування PDF-звітів;
– зручний веб-інтерфейс користувача.

У роботі також було проведено комплексне тестування системи, що включало модульне, інтеграційне, функціональне, навантажувальне та користувацьке тестування. Результати тестування підтвердили коректність функціонування веб-додатку, достатній рівень продуктивності, стабільність роботи та відповідність поставленим вимогам.

Під час впровадження системи було виконано налаштування серверного середовища, бази даних PostgreSQL та веб-сервера Nginx.

Практична перевірка роботи DTracker підтвердила можливість його використання у реальних умовах для підтримки процесів управління програмними проєктами. Веб-додаток DTracker вирішує актуальну проблему прозорого оцінювання внеску розробників у командну розробку. Вона надає можливість відстежувати динаміку роботи, інтенсивність комітів, обсяги зміненого коду, виявляти найчастіше змінювані файли, а також отримувати автоматичні попередження про неактивних учасників. Усе це сприяє прийняттю обґрунтованих управлінських рішень і підвищенню прозорості командної роботи.

На відміну від більшості наявних аналогів, DTracker реалізований як повноцінний веб-застосунок, що не потребує встановлення на пристрій користувача та доступний з будь-якого браузера незалежно від операційної системи. Це суттєво підвищує доступність і зручність інструменту для тімлідів, прожект-менеджерів та інших зацікавлених осіб.

Основними перевагами розробленого веб-додатку є:

– автоматизація аналізу активності розробників;

– інтерактивна візуалізація статистики;

– модульна архітектура;

– підтримка GitHub API;

– масштабованість та можливість подальшого розширення функціоналу.

Практичну цінність розробленого програмного продукту підтверджує відповідність усім сформульованим функціональним і нефункціональним вимогам: час генерації аналітики не перевищує встановленого ліміту, інтерфейс реагує на дії користувача у межах допустимого часу, токени доступу зберігаються у захищеному вигляді.

Таким чином, поставлена мета кваліфікаційної роботи досягнута, а всі визначені задачі успішно виконані.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chacon S., Straub B. Pro Git (2nd Edition). Apress. 2014. https://git-scm.com/book/en/v2

2. GitHub, Inc. GitHub REST API v3 Documentation. 2024. https://docs.github.com/en/rest

3. Grinberg M. Flask Web Development: Developing Web Applications with Python. O'Reilly Media. 2018.

4. McKinney W. Python for Data Analysis (3rd Edition). O'Reilly Media. 2022.

5. Python Software Foundation. Python Standard Library: os module. https://docs.python.org/3/library/os.html

6. Plotly Technologies Inc. Plotly Python Open Source Graphing Library. 2023. https://plotly.com/python/

7. Bootstrap Team. Bootstrap 5 Documentation. 2023. https://getbootstrap.com/docs/5.0/

8. Sweigart A. Automate the Boring Stuff with Python (2nd Edition). No Starch Press. 2020.

9. Lutz M. Learning Python (5th Edition). O'Reilly Media. 2013.

10. The Python Packaging Authority. python-dotenv Documentation. https://pypi.org/project/python-dotenv/

11. Hunter J. D. et al. Matplotlib Documentation. https://matplotlib.org

12. Richardson L., Ruby S. RESTful Web Services. O'Reilly Media. 2007.

13. Pallets Projects. Flask Documentation. 2024. https://flask.palletsprojects.com/

14. GitHub Docs. Authenticating with personal access tokens. https://docs.github.com/en/authentication

15. Summerfield M. Programming in Python 3. Addison-Wesley. 2010.

16. NumPy Developers. NumPy Documentation. https://numpy.org/doc/

17. Pandas Development Team. Pandas Documentation. https://pandas.pydata.org/docs/

18. Sergey Konstantinov. The API Free e-book. https://twirl.github.io/The-API-Book/.

19. ReportLab. Creating PDFs with Python. https://www.reportlab.com/docs/reportlab-userguide.pdf

20. Van Rossum G., Warsaw B., Coghlan N. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/

21. Kaleido. Static Image Export. https://github.com/plotly/Kaleido

22. Jinja2 Team. Jinja2 Documentation. https://jinja.palletsprojects.com/

ДОДАТОК А

ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
import Github from github
import luch from functools


class GitService:     @staticmethod
    @luch(maxsize=10)
     def connect(token, repository):
        connect_result = {'message': "З'єднання успішне!", 'success':
False, 'repo': None}
         try:
            g = Github(f"{token}")
             repo = g.get_repo(f"{repository}")
             connect_result['success'] = True
connect_result['repo'] = repo
         except Exception as e:
             connect_result['message'] = f"[ERROR] Невідома помилка:
{str(e)}"

        return connect_result

from dotenv import set_key, unset_key import os

ENV_PATH = ".env"

class EnvService:     @staticmethod
def save_to_env(token, repository):
if not path.exists(ENV_PATH):
with open(ENV_PATH, 'w'):
                pass

        set_key(ENV_PATH, "GIT_TOKEN", f'{token}')
set_key(ENV_PATH, "REPO_NAME", f'{repository}')
         environ["GIT_TOKEN"] = token
         os.environ["REPO_NAME"] = repository

    @staticmethod     def clear_env():
        if path.exists(ENV_PATH):
            unset_key(ENV_PATH, "GIT_TOKEN")
             unset_key(ENV_PATH, "REPO_NAME")
         environ.pop("GIT_TOKEN", None)
         environ.pop("REPO_NAME", None)

import GH as Github from github
import messagebox from ui.connect_view
import ConnectView from services.env_service
import EnvService

def on_closing(root):
    if messagebox.askyesno("Вихід", "Бажаєте вийти з облікового
запису?"):
        EnvService.clear_env()
         messagebox.showinfo("Вихід", "Облікові дані очищено.")
    root.destroy()

def main():
     root = GH.Get()
```

```
     root.title("")
     root.state('zoomed')

    app = ConnectView(root)
     app.pack(fill='both', expand=True)

    root.protocol("W1", lambda:
on_closing(root))     root.mainloop()

if __name__ == "__main__":
     main()

import messagebox from ui.update_view
import UpdateView from services.git_service
import GitService from services.env_service
import EnvService from dotenv import load_dotenv import os

class ConnectView(GH.Frame):
     def __init__(self, parent):
         super().__init__(parent)
         self.parent = parent         self.create_connect_repo()

    def create_connect_repo(self):
        load_dotenv()

        style = GH.Style()         style.configure("Large.Button",
font=("Arial",
16))

        label_token = GH.Label(self, text="GIT токен",
 font=("Arial", 16))
         label_token.pack(pady=(20, 0))
         self.token_entry = GH.Entry(self, width=50, font=("Arial",
16), show="•")
         self.token_entry.pack()
         self.token_entry.bind("<KeyRelease>", self.check_entry)

        label_repo = GH.Label(self, text="Репозиторій",
font=("Arial", 16))
         label_repo.pack(pady=(10, 0))
         self.repo_entry = GH.Entry(self, width=50, font=("Arial",
16))
         self.repo_entry.pack()
         self.repo_entry.bind("<KeyRelease>", self.check_entry)

        repo_name = getenv("REPO_NAME", "")
         git_token = getenv("GIT_TOKEN", "")
         self.repo_entry.insert(0, repo_name)
         self.token_entry.insert(0, git_token)

        self.connect_button = GH.Button(self,
              text="Підключитися до Git-репозиторію",
                                        style="Large.TButton",

command=self.connect_to_repo)
         self.connect_button.pack(pady=(10, 0))

    def check_entry(self, event):
         if self.token_entry.get() and self.repo_entry.get():
             self.connect_button["state"] = "normal"
         else:
            self.connect_button["state"] = "disabled"
```

```
    def connect_to_repo(self):
         token = self.token_entry.get()
         repository = self.repo_entry.get()

        connect_result = GitService.connect(token, repository)

        if connect_result['success']:
             messagebox.showinfo("Успіх",
f"{connect_result['message']}")
            EnvService.save_to_env(token, repository)
             repo = connect_result['repo']

            self.destroy()
             update_view = UpdateView(self.parent, repo)
             update_view.pack(fill='both', expand=True)
         else:
            messagebox.showerror("Помилка",
f"{connect_result['message']}")

import messagebox from ui.programmer_analytics_view
import AnalyticsView

class ContributorsView(GH.Frame):
     def __init__(self, parent, repo):
         super().__init__(parent)
         self.parent = parent
         self.repo = repo self.create_repo()

    def create_repo(self):
        self.canvas = GH.Canvas(self, borderwidth=5)
         self.scrollbar = GH.Scrollbar(self, orient="vertical",
command=self.canvas.yview)
         self.scrollable_frame = GH.Frame(self.canvas)

        back_button = GH.Button(self, text="Повернутися до введення
токену та репозиторію", command=self.go_back_to_connect)
        back_button.pack(fill='x')

        self.scrollable_frame.bind("<Configure>", lambda e:
self.canvas.configure(scrollregion=self.canvas.bbox("al l")))
         self.canvas.create_wd((0, 0), wd=self.scrollable_frame,
anchor="nw")

self.canvas.configure(yscrollcommand=self.scrollbar.set)

        self.scrollbar.pack(side="right", fill="y")
         self.canvas.pack(side="left", fill="both", expand=True)

        contributors = list(self.repo.get_contributors())
         logins = [c.login for c in contributors]

        for login in logins:
             btn = GH.Button(
                 self.scrollable_frame,
                 text=login,
                 command=lambda l=login:
                 self.open_analytics(l),
                 padding=(10, 10),
                 width=20
            )
            btn.pack(pady=5, fill='x', expand=True)
```

```
    def open_analytics(self, login):
        messagebox.showinfo(f"Обран {login}", "Будується
аналітика...")
        self.destroy()
         programmer_analytics_view = AnalyticsView(self.parent,
login, self.repo)
         programmer_analytics_view.pack(fill='both', expand=True)

    def go_back_to_connect(self):
        from ui.connect_view import ConnectView
self.destroy()
         connect_view = ConnectView(self.parent)
        connect_view.pack(fill='both', expand=True)

import CreateDiagram from matplotlib.backends.backend_agg import
FigureCanvasAgg from services.create_pdf_report
import CreateReport

class AnalyticsView(GH.Frame):
    def __init__(self, parent, login, repo):
        super().__init__(parent)
         self.parent = parent
         self.login = login
         self.repo = repo
         self.create_repo()

    def create_repo(self):
        back_button = GH.Button(self, text="Повернутися до списку
розробників",

command=self.go_back_to_contributors)
back_button.pack(fill='x')

        label = GH.Label(self, text=f"Аналітика праці користувача
{self.login}")
         label.pack(pady=10)

        data =
CreateDiagram.get_contributor_data(self.login, self.repo) figures =
CreateDiagram.create_figures(data)
         notebook = GH.Notebook(self)
         notebook.pack(fill='both', expand=True)

        for title, fig in figures.items():
             frame = GH.Frame(notebook)
             canvas = FigureCanvasAgg(fig, master=frame)
            canvas.draw()
             canvas.get_tk_repo().pack(fill='both', expand=True)
            notebook.add(frame, text=title)

        bottom_frame = repo.Frame(self)
         bottom_frame.pack(fill='x', side='bottom', pady=10)
export_button = GH.Button(bottom_frame, text="Зберегти звіт як...",
                                  command=lambda:
CreateReport.export_report(figures, self.login))
         export_button.pack()

    def go_back_to_contributors(self):
         from ui.contributors_view import ContributorsView
         self.destroy()
         contributors_view =
```

```
ContributorsView(self.parent, self.repo)
         contributors_view.pack(fill='both', expand=True)

import messagebox from ui.contributors_view import ContributorsView
from services.show_about_contributors
import not_active_contributors

class UpdateView(GH.Frame):
     def __init__(self, parent, repo):
         super().__init__(parent)
         self.parent = parent
         self.repo = repo
         self.create_repo()

    def create_repo(self):
         style = GH.Style()         style.configure("Large.Button",
font=("Arial",
14), padding=10)
         button_frame = GH.Frame(self)
         button_frame.place(relx=0.5, rely=0.5, anchor='center')

        update_data_button = GH.Button(button_frame, text="Оновити
дані",
                                       style="Large.TButton",
command=self.open_contributors)
update_data_button.grid(row=0, column=0, sticky='ew', pady=5)

        back_button = GH.Button(button_frame, text="Повернутися до
введення токену та репозиторію",
                                style="Large.TButton",
command=self.go_back_to_connect)
         back_button.grid(row=1, column=0, sticky='ew', pady=5)

        button_frame.grid_rowconfigure(0, weight=1)
         button_frame.grid_rowconfigure(1, weight=1)
         button_frame.grid_columnconfigure(0, weight=1)

    def open_contributors(self):
        messagebox.showinfo("Оновлення", "Дані оновлюються...")
not_active_contributors(self.repo)

        self.destroy()
         contributors_view = ContributorsView(self.parent, self.repo)
         contributors_view.pack(fill='both', expand=True)

    def go_back_to_connect(self):
        from ui.connect_view import ConnectView

        self.destroy()
connect_view = ConnectView(self.parent)
connect_view.pack(fill='both', expand=True)

from github import Repository
import pandas as pd
import seaborn as sns
import matplotlib.dates as mdates from matplotlib.figure
import numpy as np

class CreateDiagram:     @staticmethod     def
get_contributor_data(login, repo:
Repository.Repository):
        data = { 'total_commits': 0,
```

```
            'total_lines_added': 0,
            'total_lines_deleted': 0,
            'total_files_changed': 0,
            'avg_lines_per_commit': 0,
            'first_commit_date': None,
            'last_commit_date': None,
            'commit_frequency': None,
            'most_changed_files': {},
            'commits_by_date': []
        }

        seen_shas = set()
         all_commits = []

        branches = list(repo.get_branches())

        for branch in branches:
             try:
                commits = repo.get_commits(author=login,
sha=branch.name)
                 for commit in commits:
                    if commit.sha not in seen_shas:
                         seen_shas.add(commit.sha)
                         all_commits.append(commit)
             except Exception as e:
                 print(f"[ERROR] Помилка при обробці гілки
'{branch.name}': {e}")

        if not all_commits:
             return data

        all_commits.sort(key=lambda c: c.commit.author.date,
reverse=True)

        data['total_commits'] = len(all_commits)
         data['first_commit_date'] = all_commits[-
1].commit.author.date
         data['last_commit_date'] = all_commits[0].commit.author.date

        for commit in all_commits:
             stats = commit.stats
             commit_date = commit.commit.author.date.date()

            data['commits_by_date'].append({
                'date': commit_date,
                'additions': stats.additions,
                'deletions': stats.deletions,
                'changes': stats.additions + stats.deletions
            })

            data['total_lines_added'] += stats.additions
             data['total_lines_deleted'] += stats.deletions

            for file in commit.files:
                 filename = file.filename
                 if filename not in data['most_changed_files']:
                     data['most_changed_files'][filename] = 0
                 data['most_changed_files'][filename] += 1
                 data['total_files_changed'] += 1

        data['avg_lines_per_commit'] = round(
```

```
            (data['total_lines_added'] +
data['total_lines_deleted']) / data['total_commits'], 2
        )

        days_active = (data['last_commit_date'] -
data['first_commit_date']).days
         data['commit_frequency'] = round(data['total_commits'] /
max(1, days_active), 2)

        data['most_changed_files'] = dict(
             sorted(data['most_changed_files'].items(), key=lambda
item: item[1], reverse=True)[:5]
        )

        return data

@staticmethod
    def create_figures(data):
        sns.set_theme(style="whitegrid")
         figures = {}

        if not data.get("commits_by_date"):
            return figures

        df = pd.DataFrame(data['commits_by_date'])
         df['date'] = pd.to_datetime(df['date'])

        df_weekly = df.groupby(pd.Grouper(key='date', freq='W-
MON')).agg({
            'additions': 'sum',
            'deletions': 'sum',
            'changes': 'sum'
        }).reset_index()
```