Metodyka tworzenia kodu

W projekcie przyjmujemy prosty, ale uporządkowany workflow oparty na **feature branchach**, zbliżony do GitHub Flow.
Kluczową zasadą jest **brak bezpośrednich pushy na gałąź `main`** -- wszelkie zmiany trafiają na `main` wyłącznie poprzez **Pull Requesty** po **code review** i przejściu **CI**.

Struktura gałęzi

`main`

- zawsze zawiera wersję nadającą się do „wydania" -- kod buduje się, testy przechodzą, pipeline CI jest zielony;
- gałąź jest chroniona:
  - brak możliwości bezpośredniego pusha,
  - wymagany Merge Request i co najmniej jedna akceptacja recenzenta.

Gałęzie funkcjonalne (`feature/*`)

- tworzone z `main` dla każdej zmiany lub niewielkiego zestawu powiązanych zadań, np.:
  - `feature/cicd`
  - `feature/extract-bug-fix-pairs`
  - `feature/db-statistics-endpoint`
- po zakończeniu pracy i zmergowaniu do `main`, gałąź jest usuwana, aby utrzymać porządek w repozytorium.

Poprawny i uargumentowany wybór języków programowania

Wybór języka

Do implementacji narzędzi tworzących bazę danych oraz narzędzi analizy danych w projekcie zostanie wykorzystany język **Python**.

Zgodność z charakterem problemu

Problem, który rozwiązujemy, obejmuje przede wszystkim:

1. **Ekstrakcję danych z repozytoriów** (Firefox, CPython, PostgreSQL itp.) -- praca z Gitem, historią commitów, diffami.\
2. **Przetwarzanie tekstu/kodu źródłowego** -- wyszukiwanie fragmentów C++, parsowanie, analiza różnic między wersjami plików.\
3. **Budowę i utrzymanie bazy danych** -- tworzenie rekordów, zapisywanie statystyk, wykonywanie podstawowych zapytań analitycznych.\
4. **Analizę i eksplorację danych** -- zliczanie typów błędów, tworzenie zestawień, przygotowanie danych do uczenia maszynowego.\

5. **Eksport danych** -- dostarczenie danych w formacie nadającym się do trenowania modeli ML.

Zadania te mają charakter skryptowy, wymagają pracy z dużą liczbą plików, analizą danych i automatyzacją --- co doskonale pokrywa się z profilem Pythona.

Zalety Pythona w tym projekcie

1. Bogaty ekosystem do pracy z repozytoriami i kodem

- Biblioteki takie jak **GitPython** czy **PyDriller** ułatwiają:
  - iterację po commitach i gałęziach,
  - pobieranie diffów i metadanych,
  - filtrowanie zmian dotyczących plików C++.
- Istnieją wiązania do parserów C++ (np. **libclang**) oraz biblioteki oparte na **tree-sitter**, umożliwiające analizę składniową kodu.

2. Wygodne przetwarzanie tekstu i danych

- Python oferuje bogate narzędzia pracy na dowolnych strukturach tekstu i plików.\
- Biblioteki takie jak **pandas** umożliwiają szybką eksplorację danych, raportowanie oraz agregację statystyk błędów.

3. Integracja z bazami danych

- Python posiada wsparcie dla baz relacyjnych i nierelacyjnych (PostgreSQL, SQLite, MySQL, MongoDB).\
- Ułatwia to:
  - definiowanie struktury rekordów,\
  - wstawianie i aktualizowanie danych,\
  - wykonywanie złożonych zapytań analitycznych.

4. Wsparcie dla uczenia maszynowego

- Python jest dominującym językiem ML --- biblioteki takie jak **scikit-learn**, **PyTorch**, **TensorFlow**, **transformers** ułatwiają pracę na danych.\
- Wykorzystanie Pythona już na etapie ekstrakcji pozwoli zachować spójność całego pipeline'u danych.

5. Szybkość prototypowania i czytelność

- Python pozwala szybko tworzyć prototypy i płynnie przechodzić do pełnych modułów.\
- Jego składnia ułatwia code review i współpracę zespołową.

6. Możliwość optymalizacji krytycznych fragmentów

- Python umożliwia korzystanie z modułów napisanych w C/C++ lub optymalizowanych bibliotek --- bez utraty elastyczności.

Alternatywy i ich ocena

C++

- Wolniejsze prototypowanie, trudniejsze zarządzanie zależnościami.

- Nadaje się głównie do fragmentów wymagających wysokiej wydajności.

Java / Kotlin / C

- Mocne w backendzie, słabsze w ML i szybkiej analizie danych.

Bash, Perl

- Mało czytelne, słabo skalują się w dużych projektach.

Podsumowanie

Python: - najlepiej wspiera charakter projektu, - integruje się z ML, - umożliwia tworzenie czytelnego, modularnego, testowalnego kodu.

Wykorzystywane technologie

Komponenty systemu

1. MongoDB (Baza danych)

Technologia: MongoDB 6 w kontenerze Docker

Zadania:

- przechowywanie rekordów zawierających pary `code_buggy` i `code_fixed`
- przechowywanie metadanych repozytoriów
- przechowywanie etykiet błędów

Wybór technologii:

- MongoDB jako NoSQL dobrze poradzi sobie z przechowywaniem półstrukturalnych danych (fragmenty kodu mogą mieć różną długość i strukturę)
- Schemat dokumentowy pozwala na elastyczne rozszerzanie etykiet

2. FastAPI (Backend API)

Technologia: FastAPI >=0.121, hostowane przez uvicorn.

Zadania:

- warstwa dostępu do bazy danych (prosty CRUD + wyszukiwanie z użyciem filtrów i sortowanie)

Wybór technologii:

- proste w użyciu
- niewymagana wysoka złożoność i wydajność w naszym projekcie

Kluczowe endpointy:

- `POST /entries/` - dodawanie nowego rekordu
- `GET /entries/{id}` - pobranie rekordu po ID

- `GET /entries/` - lista wszystkich rekordów
- `PUT /entries/{id}` - aktualizacja rekordu
- `DELETE /entries/{id}` - usunięcie rekordu
- `POST /entries/query/` - zaawansowane filtrowanie z parametrami sort/limit

3. Scraper (Ekstraktor danych z repozytoriów)

Technologia: Python 3.12

Zadania:

- analiza repozytoriów Git projektów C++
- filtrowanie commitów zawierających poprawki błędów (regex na komunikaty commitów: "fix", "bug", "patch" itp.)
- ekstrakcja par `code_buggy` i `code_fixed`
- wstępne automatyczne przypisanie etykiet

Będziemy korzystać z Github API.

Narzędzia analizy statycznej kodu

Aplikacja będzie wykorzystywał narzędzia do statycznej analizy kodu w celu usprawnienia procesu etykietowania błędów.

- `cppcheck`
- `clang-tidy`

Etykiety z tych narzędzi będą pogrupowane w etykiety o szerszym zakresie np.

- Klasa ogólna: MemError
  - błędy z `cppcheck`: memLeak, doubleFree, mismatchAllocDealloc, nullPointerDereference ...

4. CLI (Narzędzie konsolowe)

Technologia: Python 3.12

Zadania:

- interaktywny interfejs dla użytkownika końcowego
- przeglądanie, filtrowanie i wyszukiwanie rekordów
- ręczne dodawanie i edycja rekordów
- eksport danych do plików CSV/JSON
- wywoływanie scrapera

Aplikacja bedzię korzystać z **plików konfiguracyjnych**, które będą określać jakie repozytoria ma analizować scraper i w jaki sposób ma to robić.

Konteneryzacja

Technologia: Docker

Wszystkie komponenty są konteneryzowane.

Pipeline CI

Projekt wykorzystuje **GitHub Actions** do automatyzacji testowania i kontroli jakości kodu.

Konfiguracja CI

Plik `.github/workflows/ci.yml` definiuje pipeline uruchamiany przy:

- każdym pushu na gałąź `main`
- każdym Pull Requeście

Kroki pipeline'u

1. Checkout kodu

2. Konfiguracja środowiska Python

3. Instalacja zależności

4. Analiza jakości kodu - flake8

```
flake8 cli fastapi_app scraper
```

5. Analiza bezpieczeństwa - bandit

```
bandit -r cli fastapi_app scraper
```

Przyszłe rozszerzenia CI

Pipeline CD

Nie planujemy rozszerzenia o CD, gdyż rozwiązanie ma w zamyśle działać lokalnie.

Testy jednostkowe i integracyjne

- testy jednostkowe dla wszystkich modułów
- testy integracyjne

Minimum Viable Product (MVP)

Wypełniona baza danych wraz z systemem odpowiedzialnym za jej stworzenie oraz CLI umożliwiający dostęp do bazy.

Baza:

- Dostarczona baza danych zawiera min. 1000 rekordów - par **code_buggy** i **code_fixed** oraz odpowiednio poetykietowanych tak, aby miała dane miały wartość badawczą do wykorzystania do trenowania modeli ML

System:

- System filtruje commity wskazanego repozytorium, przygotowuje pary **code_buggy** i **code_fixed** oraz wstawia je jako rekordy do bazy danych
- System półautomatycznie przypisuje rekordom etykiety błędów za pomocą narzędzi analizy statystycznej
- System umożliwia ręczne przypisanie, edycję etykiet dla rekordów, ręczne dodawanie rekordów

CLI:

- CLI umożliwia przeglądanie i filtrowanie zebranych rekordów, pozyskiwanie danych

Testy akceptacyjne

TA-1: Ekstrakcja danych z repozytorium Git

Scenariusz:
Użytkownik uruchamia scraper poprzez CLI dla repozytorium C++ https://github.com/mozilla-firefox/firefox, wiedłki 01.01.2020 - 01.01.2024, system analizuje historię commitów, wybiera pary, przypisuje do niej etykierę i zapisuje rekordy w bazie.

Kroki:

1. Uruchomienie scrapera poprzez CLI, z plikiem konfiguracyjnym w którym jest podane analizowane repozytorium
2. Filtrowanie commitów po regex (np. `(fix|bug|patch|repair|correct)`) w wiadomości
3. Ekstrakcja diffów dla plików `.cpp`, `.h`, `.cc`
4. Utworzenie rekordów z polami:
    - `code_buggy` - kod przed poprawką
    - `code_fixed` - kod po poprawce
5. Automatyczne przypisanie etykiet do par

Kryteria akceptacji:

- co najmniej 100 rekordów zostanie zapisanych
- dla każdego rekordu `code_buggy != code_fixed`
- etykieta jest poprawnie przypisana do kategorii błędu

TA-2: Interaktywne CLI - przeglądanie i filtrowanie

Scenariusz:
Użytkownik wykorzystuje narzędzie CLI do eksploracji bazy danych oraz wyszukiwania rekordów według kryteriów.

Kroki:

1. Wylistowanie wszystkich rekordów

- użytkownik wpisuje w CLI `list-all`

2. Filtrowanie po etykiecie

- użytkownik wpisuje w CLI `list-label --MemError`

Kryteria akceptacji:

- CLI zwraca tabelę z kolumnami: `code_buggy`,`code_fixed`, `labels`
- filtrowanie po `labels` zawęża wyniki tylko do rekordów z daną etykietą (w tym przypadku MemError)

TA-3: Eksport danych do formatu JSON/CSV

Scenariusz:
Użytkownik eksportuje przefiltrowane rekordy do formatu JSON/CSV nadającego się do użycia jako dataset treningowy dla modeli opartych o architekturę transformer.

Kroki:

1. Eksport wszystkich rekordów

- użytkownik wpisuje w CLI `export-all --JSON` / `export-all --CSV`

2. Eksport przefiltrowany

- użytkownik wpisuje w CLI `export-labels --[labels: MemError] --JSON`

Kryteria akceptacji:

- aplikacja zapisuje na dysku plik z danymi w pożądanym formacie
- filtrowanie po `labels` zawęża wyniki tylko do rekordów z daną etykietą
- format JSON zawiera co najmniej pola: code_buggy, code_fixed, labels
- format CSV zawiera co najmniej kolumny: code_buggy, code_fixed, labels
- plik JSON jest poprawny syntaktycznie
- znaki specjalne w kodzie C++ są prawidłowo zapisane

TA-4: Edycja rekordu

Scenariusz:
Użytkownik ręcznie edytuje rekord aby poprawić etykiety.

- ręczna edycja rekordu w CLI Uzytkownik wpisuje `edit {id} --add-label --[labels: MemError]` lub `edit {id} --remove-label --[labels: MemError]`

Kryteria akceptacji:

- rekord jest odpowiednio edytowany, zmiany są widoczne w bazie

TA-5: Przygotowanie datasetu do treningu transformera

Scenariusz:
Badacz ML eksportuje dane z bazy i używa ich do fine-tuningu modelu CodeBERT na zadaniu klasyfikacji błędów w kodzie C++.

Kroki:

1. Eksport danych treningowych

- użytkownik wpisuje w CLI `export-all --JSON`

2. Załadowanie danych
3. Trening klasyfikatora
4. Ewaluacja na zbiorze testowym z bazy

Kryteria akceptacji:

- po treningu model osiąga większe accuracy niż przed treningiem o co najmniej 10 p.p.