[STUDENCKIE GENY]

Dokumentacja projektowa PZSP2

WERSJA 2.0.0

[16.11.2023]

Semestr 23Z

Zespół nr 14 w składzie:
Krzysztof Fijałkowski
Adam Jeliński
Rafał Szczepaniak
Wojciech Zarzecki

Mentor zespołu: dr hab. inż. Krzysztof Cabaj prof. PW

Spis treści

1 Wprowadzenie

1.1 Cel projektu

Oczekuje się, że system będzie w stanie przetwarzać warianty typu SNV dla całego genomu, korzystając z kilku wybranych algorytmów, oraz umożliwiać przeliczenie online wariantów typu INDEL. Dzięki temu użytkownicy będą mogli szybko i skutecznie oceniać potencjalną patogenność wariantów splicingowych, co może mieć kluczowe znaczenie w diagnostyce genetycznej i medycynie personalizowanej. W założeniu, platforma będzie stanowiła uzupełnienie funkcjonalności popularnej bazy adnotacji wariantów dbNSFP.

1.2 Wstępna wizja projektu

Mikroserwisowa aplikacja postawiona na bazie docker compose mająca na celu przetwarzanie i cachowanie wariantów genetycznych. Aplikacja będzie udostępniała REST API oraz prosty interface użytkownika. Aplikacja będzie umożliwiać optymalne dodanie nowego algorytmu.

2 Metodologia wytwarzania

Link do repozytorium publicznego: https://github.com/wz7475/genetics-app

Wszystkie decyzje architektoniczne decydujące o technologii etc. Podejmowane są podczas spotkań całym zespołem.

Samo wytwarzanie aplikacji będzie na bazie następującego pipeline:

- Github issues dla opisu większych funkcjonalności lub zadanie na platformie Asana
- Odpowiedni branch na każde zadanie
- Przed merge code review osoby z zespołu
- Merge

Nazwy commit-ów są zgodne z konwencją conventional commits: przedrostek określający rodzaj commit-a np. feat - nowa funkcja, refactor - zmiana nie wpływająca na funkcjoność systemy oraz po dwukropku krótki opis dokonanych z main.

oprócz tego wykorzystaliśmy Github actions w celu uruchamiania testów jednostkowych oraz systemowych w niezależnym środowisku. Zaimplementowane zostały dwa githubowe ‘workflowy’ to znaczy pipeliny:

- jeden z nich (zaimplementowany w pliku python-app.yml) jest odpowiedzialny za ciągłą integracje (continuous integration) i jest on uruchamiany wtedy, kiedy jest robiony push na brancha innego niż main, lub kiedy jest tworzony pull request do maina. Pipeline ten uruchamia lintera (flake8) w celu zrobienia statycznej analizy kodu, następnie uruchamia testy jednostkowe. Dzięki niemu kod w repozytorium ma pewnego rodzaju gwarancję jakości, oraz mamy pewność że na naszego głównego brancha nie zostanie dołączony kod który nie działa (brak wstecznej kompatybilności, zły kod).
- drugi jest odpowiedzialny za deployment (kod w deploy-app.yml) i jest uruchamiany tylko wtedy, gdy pull request jest mergowany do maina. Co do funkcjonalności robi to samo co pierwszy pipeline, natomiast dodatkowo ma step w którym wywołuję zdalnie komendę na serwerze bigubu.ii.pw.edu.pl za pomocą ssh. Tą komendą uruchamiamy skrypt na serwerze który clonuje,

lub pulluje kod z repozytorium (w zależności czy on już był kiedyś clonowany) następnie buduje kontenery i uruchamia aplikację.

Testy są uruchamiane na ten moment lokalnie, natomiast docelowo będą uruchamiane na specjalnie przeznaczonym do ich uruchamiania kontenerze.

3 Analiza wymagań

3.1 Wymagania użytkownika i biznesowe

Problemy do rozwiązania:

- Brak dostępnych rozwiązań do wygodnego wykorzystywania algorytmów do splicingu genotypów
- Czas wykonania algorytmów jest bardzo długi
- Do przetwarzania i przechowywania mamy bardzo duże ilości danych (ponad 1 miliard wariantów SNV i wiele wariantów INDEL)

Cele i potrzeby biznesowe:

- Stworzenie narzędzia które pozwala na wygodne zarządzanie algorytmami do splicingu które szybko się wykonują

Wymagania Biznesowe:

- Projekt dotyczy stworzenia platformy do wspomagania adnotacji wariantów splicingowych dla danych z sekwencjonowania następnej generacji (NGS). Platforma umożliwia adnotowanie wariantów typu SNV oraz INDEL w celu identyfikacji wariantów o dużym znaczeniu klinicznym.

Wymagania Użytkowe:
Użytkownicy, prawdopodobnie genetycy lub naukowcy, potrzebują narzędzia do szybkiego i skutecznego oceniania potencjalnej patogenności wariantów splicingowych.

- Możliwość wysłania plików TSV z wariantami do analizy
- Możliwość komunikacji przez REST API, bez użycia interfejsu webowego
- Możliwość pobrania wynikowego pliku analizy

Wymagania Systemowe:
System powinien akceptować pliki TSV z listami wariantów, przetwarzać i analizować warianty, być rozszerzalny, umożliwiać łatwe aktualizacje i dodawanie nowych algorytmów. Powinien także dostarczać dokładne i wiarygodne wyniki adnotacji, co jest kluczowe dla klinicznych interpretacji wariantów splicingowych.

- cache wcześniej przetworzonych wariantów SNV
- analiza online wariantów INDEL

3.2 Wymagania funkcjonalne i niefunkcjonalne

Wymagania funkcjonalne:

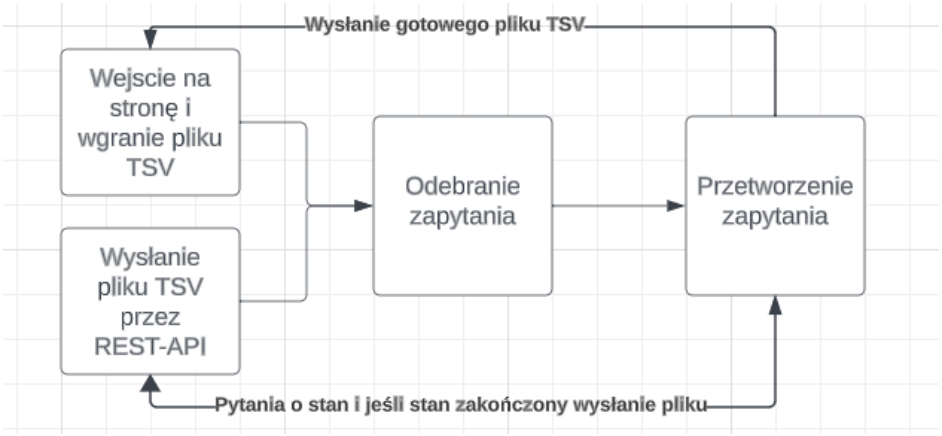
- umożliwienie przetwarzanie wariantów INDEL przy pomocy kilku gotowych algorytmów
- umożliwienie analizy wariantów SNV na podstawie wcześniej przetworzonych danych
- możliwość wgrania pliku wejściowego w formacie tsv i pobrania pliku tsv z adnotacjami

Wymagania niefunkcjonalne:

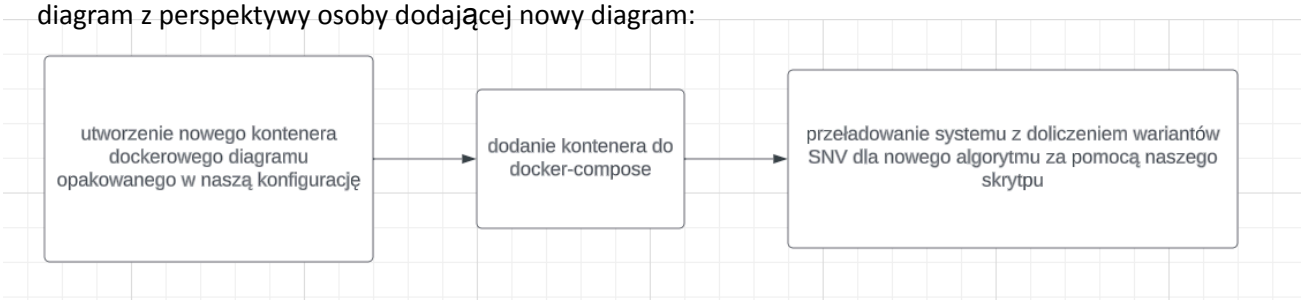
- umożliwienie przetwarzania 4 tys. wariantów podczas jednego zapytania
- łatwe dodanie nowego algorytmu - dodanie kolejnego algorytmu do istniejących w formie np. odpowiednio skonfigurowanego kontenera dockerowego
- Zapisywanie wyników analizy wariantów INDEL do bazy, w celu uniknięcia wielokrotnego przetwarzania
- Pre obliczenie wszystkich możliwości wyników algorytmów typu SNV
- rest API umożliwiające umożliwiająca wgranie pliku wejściowego i pobranie wyjściowego
- minimalistyczna interfejs graficzny - pojedyńcza strona umożliwiająca wgranie pliku wejściowego i pobranie wyjściowego
- Stworzenie całej architektury na bazie mikroserwisów i obrazów dockerowych (oprócz front-endu) używając “docker compose” w celu łatwego przeniesienia do np kubernetesa
- Stworzenie dokumentu z “step-by-step” instrukcją uruchomienia, instalacji i aktualizacji systemu

3.3 Przypadki użycia

- diagram z perspektywy użytkownika przeliczającego warianty:



- diagram z perspektywy osoby dodającej nowy diagram:



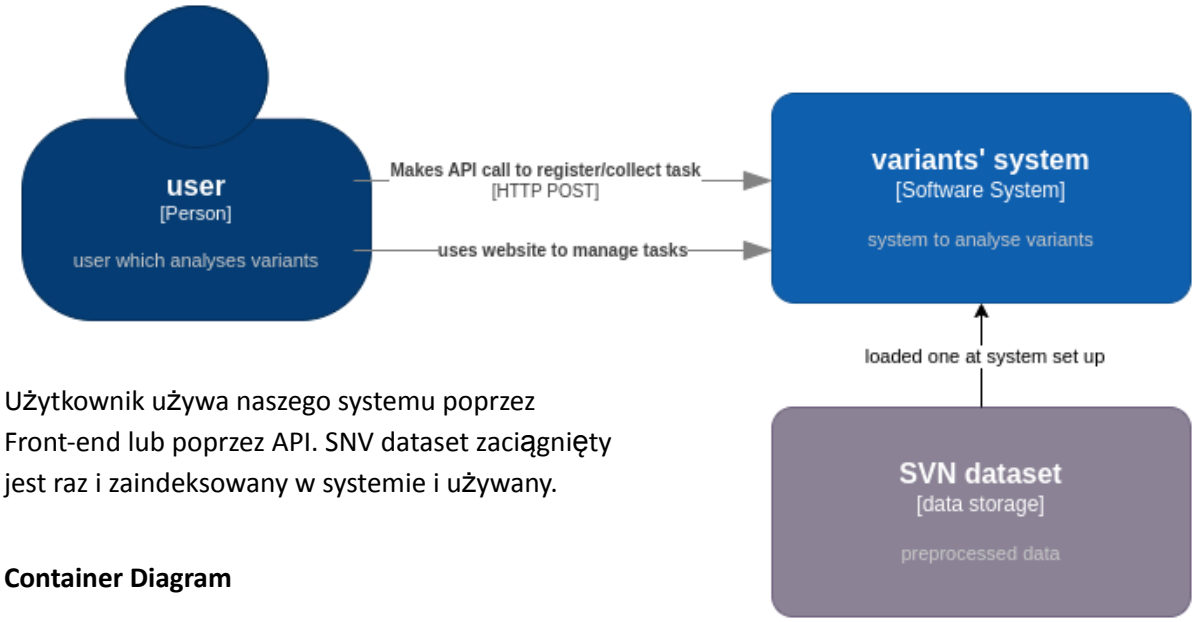
3.4 Potwierdzenie zgodności wymagań

Zatwierdzam specyfikację wymagań, jako spełniających potrzeby Klienta.	…… Data i podpis Właściciela tematu

Uwagi

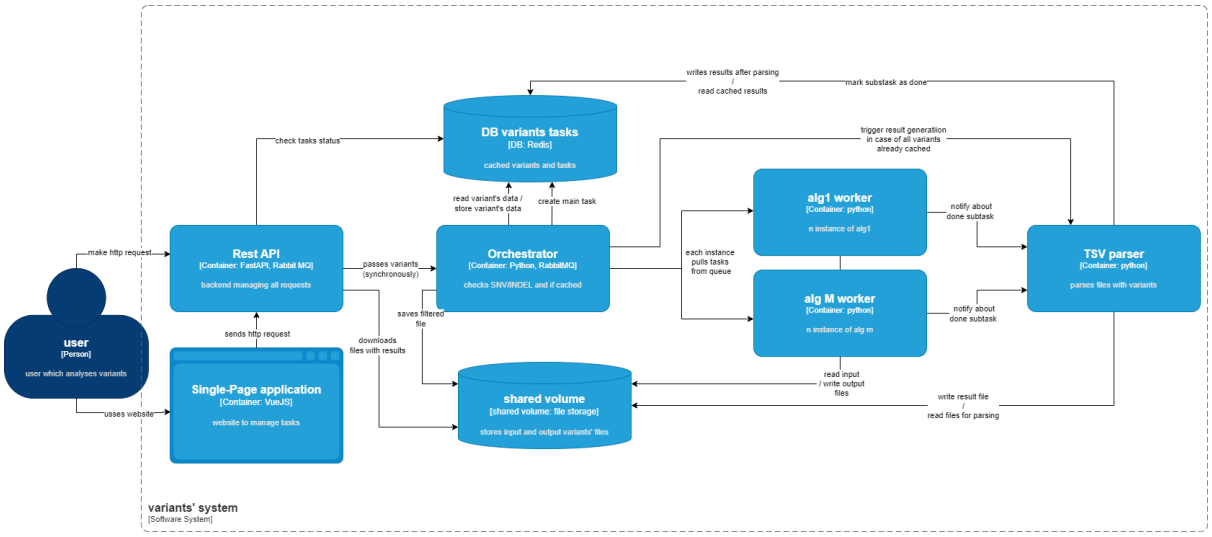
4 Definicja architektury

Context diagram



Użytkownik używa naszego systemu poprzez Front-end lub poprzez API. SNV dataset zaciągnięty jest raz i zaindeksowany w systemie i używany.

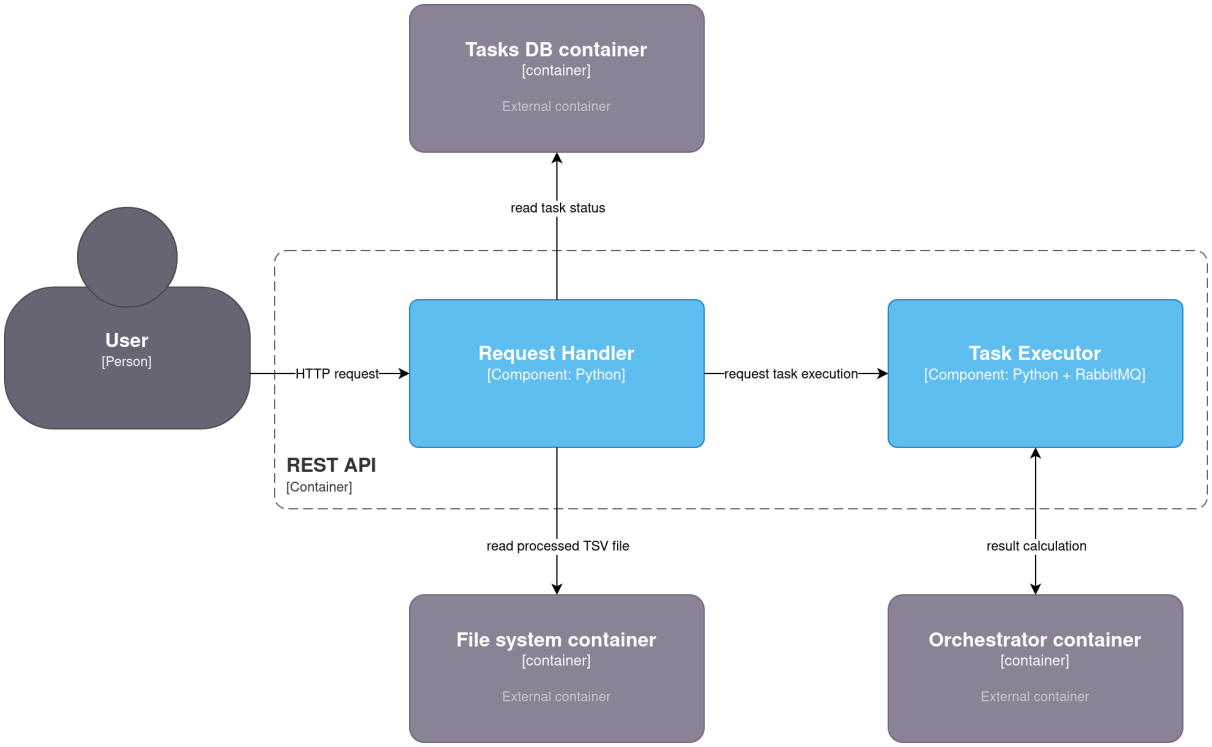
Container Diagram



Każdy kolejny serwis jest osobnym kontenerem dockerowym. Architektura kontenerów została wypisana tylko dla bardziej rozbudowanych elementów.

Rest API

Server wystawiający interfejs typu REST umożliwiający przyjmowanie zapytań od użytkownika - oraz sprawdzanie statusu/rezultatów. Przekazuje żądanie przetwarzania wariantów do Orchestrator-a, a w przypadku żądania sprawdzenia statusu sprawdza czy zadanie oraz jego podzadania są oznaczone jako wykonane.

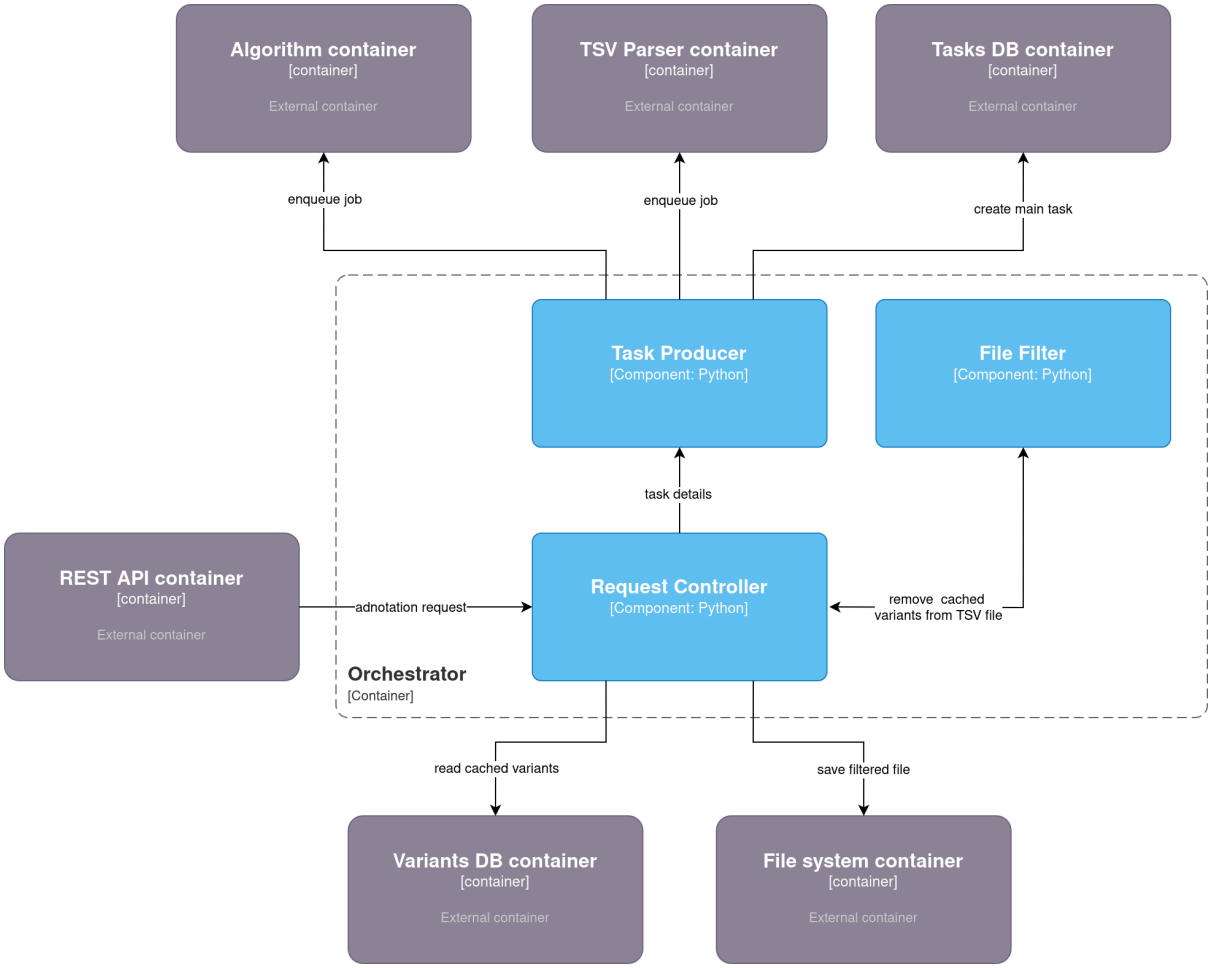


Single Page application

Jest to aplikacja, która jest jedną stroną. Aplikacja, napisana w Vue.js umożliwiająca wygodną interakcję z systemem. Daje użytkownikowi możliwość tworzenia nowych zadań i podgląd ich statusu.

Orchestrator

Komponent, który sprawdza, czy dane SNV/INDEL są w głównej bazie danych. Jeśli nie są: tworzy plik TSV tylko z wariantami które muszą być policzone, wstawia ten plik na shared volume oraz tworzy nowe zadanie do przetwarzania danych przez workery z algorytmami. Zapisuje informację o utworzeniu głównego zadania, podzadań oraz powiązań między nimi bazie Task DB. Jeżeli warianty są już policzone, wysyła task do TSV parsera aby on utworzył plik wynikowy i wstawił go do shared volume.



Task & Variants DB

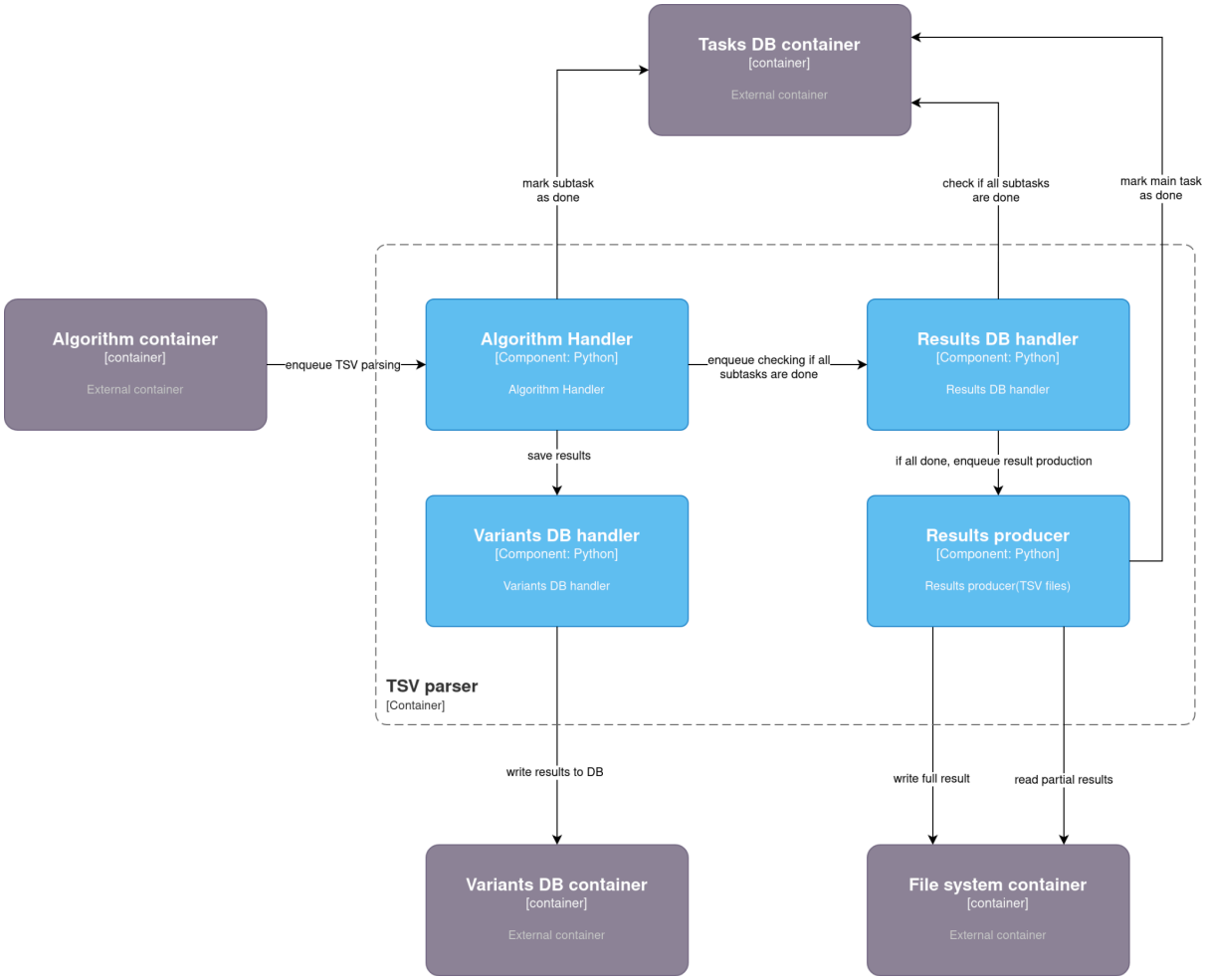
Baza danych typu klucz-wartość, która przechowuje warianty SNV/INDEL oraz ich adnotacje. Jest wykorzystywana do przetrzymywania przeliczonych wariantów SNV, wpisywania z-cache-owanych wariantów INDEL przez TSV parser-a, oraz odczytu przez TSV parsera danych w celu utworzenia plików wynikowych. Używana jest również do zakolejkowane zadań oraz ich statusu. Status danych tasków wpisuje TSV parser.

Shared volume

Wspólna przestrzeń dyskowa dla wszystkich modułów, służąca do przetrzymywania plików TSV. Orchestrator umieszcza tam pliki TSV które będą później przetwarzane przez workerów. Algorithm workery umieszczają w niej przeliczone fragmenty plików TSV a TSV parser pobiera z niej te fragmenty i składa je w całość jeśli będzie taka możliwość.

TSV parser

Zaciąga wyniki przeliczonych algorytmów z shered volume po otrzymaniu informacji od algorytmów o ukończeniu przeliczania. Wpisuje wyniki do bazy danych DB variants, oraz wpisuje wykonanie tasków DB tasks. Tworzy również wynikowe pliki TSV na podstawie danych z DB variants które wpisuje do shered volume.



alg1 worker

Jest to algorytm dostarczony przez klienta zapakowany w moduł definiujący użycie tasków i dostępu do bazy danych. Zasada działania:

- przyjmuje zadania (paczki wielu wariantów) z przypisanej do niego kolejki
- oblicza adnotacje dla otrzymanych wariantów dla konkretnego algorytmu
- wstawia przeliczony plik TSV do shered volume oraz powiadamia TSV workera o przeliczeniu wariantów dla danego taska

W celu zwiększenia wydajności planujemy wystawić wiele instancji tego serwisu, load balancing jest zrobiony na bazie kolejki rabbit mq.

algN worker

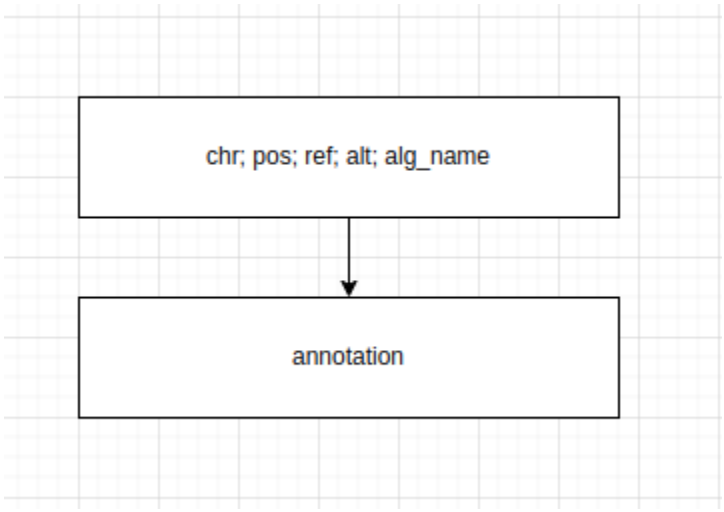
To samo co alg1 worker, dla innego algorytmu, jest na diagramie w celu zobrazowania istnienia wielu algorytmów.

5 Dane trwałe

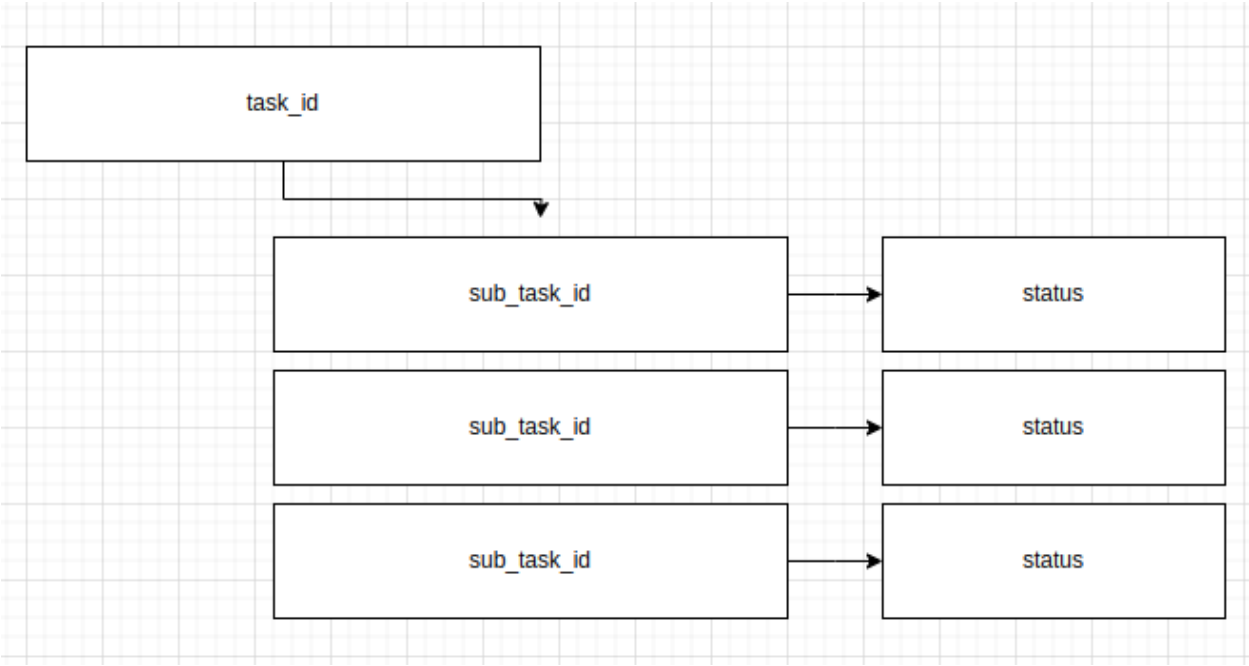
5.1 Model danych

baza klucz wartość

model adnotacji



model systemu zadań



5.2 Przetwarzanie i przechowywanie danych

Stosujemy 2 rodzaje nośników danych trwałych o różnych właściwościach. Jest to baza klucz wartość - Redis oraz tzw file storage - dysk do zapisu fizycznych plików.

Opis danych w aplikacji

Algorytm N jest reprezentacją kolejnych pojawiających się algorytmów. Taki algorytm na wejściu przyjmuje wariant - czwórkę wartości chr, pos, ref, alt i zwraca adnotację - jego rezultat. Wyróżniamy 2 rodzaje wariantów - SNV i INDEL.
Piątka wartości: chr, pos, ref, alt (identyfikujące wariant) oraz nazwa algorytmu to klucz, a wartość z nim skojarzona to adnotacja danego wariantu przez wskazany algorytm.

Używamy tasków i ich numerów w celu kontroli aktualnych stanów zapytań przychodzących. W momencie tworzenia zadania do policzenia odpowiednich wariantów Orchestrator wpisuje numer task-a do bazy Redis i podaje numer taska Algorytmowi który podaje go TSV parserowi który w odpowiednim momencie oznacza zadanie za wykonane, nie ma potrzeby pokazywania wartości wariantów z taskami

Musiemy przetrzymywać pliki ponieważ algorytmy dostarczone potrzebują plik na wejściu i zwracają one również plik. Klientowi zwracamy również wynik w postaci pliku.

Sposoby składowania tych danych:

Baza klucz wartość Redis - na celu trzymanie tasków, pod-tasków i ich stanów oraz adnotacji wariantów. wybrana ze względu na wysoką wydajność i schemat podobny do naszego problemu.

shared Volume -file storage, współdzielony wolumen na tymczasowe pliki, potrzebne algorytmom.

Analiza zajętego miejsca

Jeden wariant z adnotacją zajmuje 264B zapisu para klucz wartość w Redis. Na 10 GB dysku zmieści się ok. 40 milionów adnotacji.

Czas generacji danych

Adnotacji wariantu algorytmem spip zajmuje ok. 0.05s, a algorytmem pangolin ok. 3.20s.

Czas odczytu

Czas odczytu jednego wariantu zajmuje ok. 0.004s

6 Specyfikacja analityczna i projektowa

Link do repozytorium: https://github.com/wz7475/genetics-app

Język programowania: Python, ECMAScript

Framework-i: FastApi, RabbitMQ, Redis, Docker, Vue.js

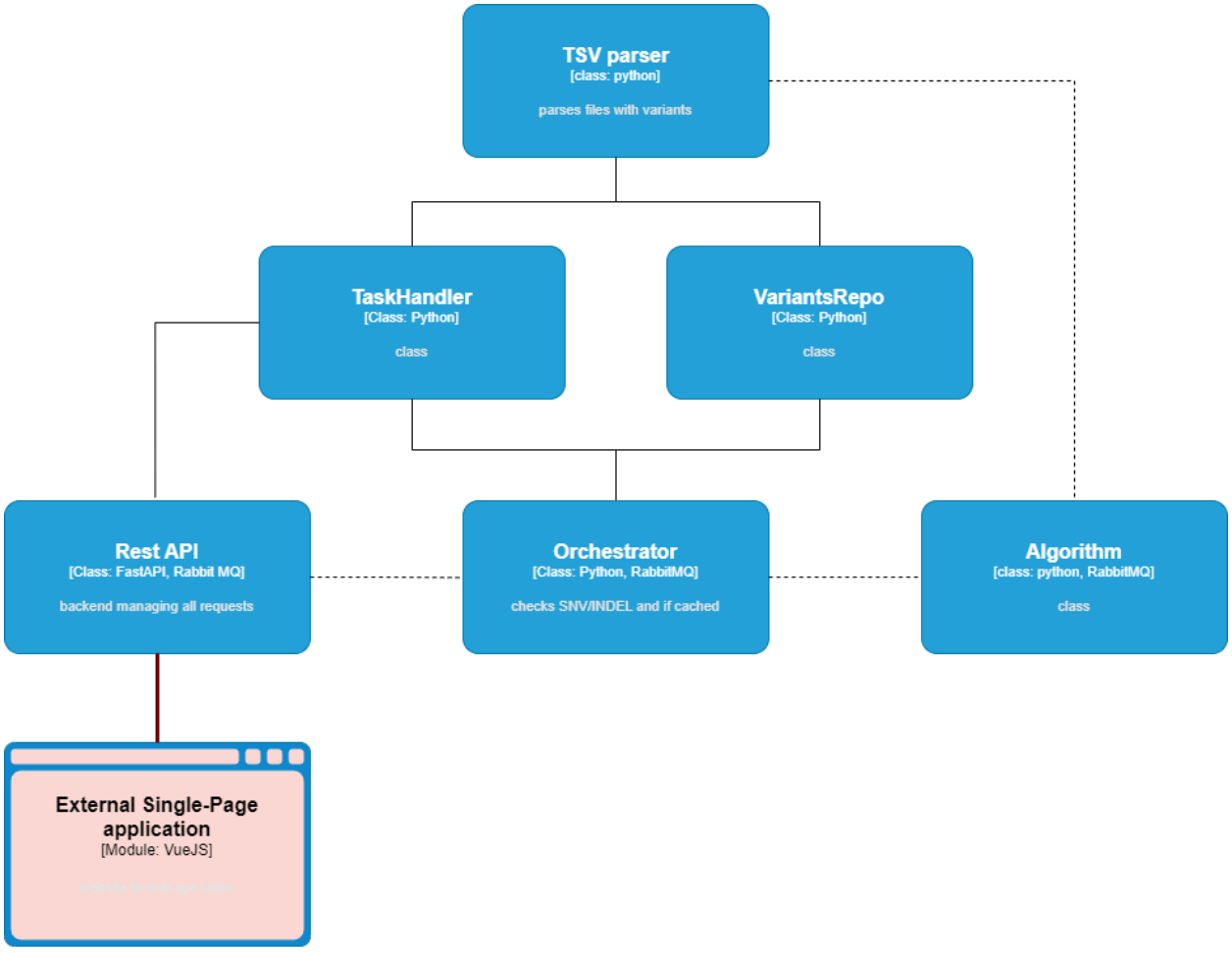
Środowisko programowania: Pycharm, VS Code

Uruchamianie: Program uruchamiany jest za pomocą Dockera przy użyciu pliku konfiguracyjnego docker-compose.yml

Wdrażanie: wdrażanie jest wykonywane przez workera na githubie który przez ssh uruchamia skrypt bashowy który z kolei używa Docker. Ze względu na to, że serwer politechniki ma konkretne wymagania co do nazewnictwa, oraz łatwo o wystąpienie konfliktów portów oraz ograniczeń maszyny na cele prezentacji użyliśmy serwisu gcloud.

Środowisko ciągłej integracji: GitHub Actions - wyzwalacze ustawione tak, że rozróżniają pomiędzy mainem, a pozostałymi branchami. Dzięki takiemu rozwiązaniu, programiści działając na swoich branchach, cyklicznie odpalają testy jednostkowe aby zapewnić że kod dołączony do maina będzie działał oraz że będzie wstecz kompatybilny. Natomiast w momencie kiedy nowa funkcjonalność jest dołączana do głównej gałęzi uruchamiany jest również krok odpowiedzialny za automatyczne wdrażanie.

Diagram modułów:



Przerywane linie oznaczają komunikację poprzez komunikaty przy użycie brokera RabbitMQ, ciągłe oznaczają powiązanie modułów.

Statystyki (na podstawie git-fame):

Total commits: 139

Total files: 154

Total lines of code: 24839

Author	lines of code	commits	files	distribution
Wojtek Zarzecki	17858	71	56	71.9/48.9/36.4
Adam Jeliński	5092	25	46	20.5/18.0/29.9
Krzysztof Fijałkowski	1645	35	32	6.5/13.7/19.5
Rafał Szczepaniak	244	8	20	1.0/ 5.0/13.0

(Statystyki są zakłamane z powodu na dodawanie plików z danymi itp)

7 Projekt standardu interfejsu użytkownika

Zgodnie z ustaleniami z klientem projekt nie posiada interface-u użytkownika poza jedną stroną z przyciskiem prześlij, tak więc UX nie istnieje.

8 Specyfikacja testów

Standardy obsługi błędów i sytuacji wyjątkowych:

- dzięki użyciu biblioteki pythonowej fastapi, kontener na którym działa API jest odporny na błędy czasu wykonania, np jeśli wyślę się dane w złym formacie (plik inny niż tsv) to wewnątrz kontenera będzie informacja, że ktoś wysłał złe dane, natomiast użytkownik otrzyma informację że wystąpił błąd po stronie serwera

Rodzaje testów:

- testy jednostkowe API
- testy integracyjne dodawania algorytmów
- testy end-to-end
- testy manualne

Opis sposobu realizacji testów:

- testy jednostkowe sprawdzaliśmy ‘mockując’ bazę danych aby nie używać API do wyciągania danych z faktycznego Redis. Nie jest ich wiele ponieważ nasz aplikacja nie zawiera w sobie dużo logiki biznesowej a jedynie łączy ze sobą wiele modułów. Dlatego uznaliśmy, że nie warto poświęcać bardzo dużego nakładu pracy na ‘mockowanie’ pozostałych komponentów systemu czyli RabbitMQ oraz Redis i położyliśmy nacisk na testy integracyjne
- testy integracyjne sprawdzające czy moduły w naszej aplikacji w poprawny sposób współpracują ze sobą przeprowadziliśmy dodając nowy zaimplementowany przez nas algorytm, a następnie przetestowaliśmy czy dobrze zadziałał z resztą systemu do którego został dodany
- testy end-to-end przeprowadziliśmy wykorzystując wyżej wymieniony algorytm. Sprawdziliśmy w nich czy plik został dostarczony do algorytmu, zadnotowany oraz czy plik wynikowy pokrywa się z założeniami. Oprócz tego sprawdzaliśmy między innymi czy orchestrator poprawnie przydzielił zadanie algorytmowi.
- testy manualne były robione w największym stopniu do frontendu, ale również do API w trakcie rozwijania aplikacji za pomocą programu Postman

Scenariusze testowe:

- w API testowaliśmy endpointy takie jak wysyłanie plików, pobieranie plików oraz sprawdzanie stanu wykonania procesu adnotacji (czy plik już jest gotowy czy nie)
- podstawowym scenariuszem jest wgranie do aplikacji naszego pliku testowego, który następnie zostanie przetworzony tak aby pokrywał się konwencją z naszym przykładowym algorytmem, następnie orchestrator powinien w odpowiedni sposób przydzielić taska algorytmowi a na koniec zapisać plik w bazie

Miary jakości testów:

- zastanawialiśmy się nad użyciem Sonarqube natomiast zrezygnowaliśmy z niego gdyż serwer bigubu.ii.pw.edu.pl przestał odpowiadać
- jako miary postanowiliśmy wybrać pokrycie linii oraz gałęzi kodu, natomiast zabrakło nam czasu na wdrożenie narzędzia które by te miary sprawdzało

9 Wirtualizacja/konteneryzacja

Cały system jest oparty na docker-compose więc jest w pełni skonteneryzowany. Obraz dla każdego serwisu jest opisany w Dockerfile-u oraz opisany jako serwis w docker-compose.yml, kluczowe seriwsy mogą zostać utworzone z dowolną ilością replik. Istnieje możliwość przerobienia sytemu pod k8s w momencie zaistnienia wymagań klienckich.

10 Bezpieczeństwo

Bezpieczeństwo zapewnia klient, zakładamy, że system pracuje w sieci wewnętrznej i tylko autoryzowane osoby mają dostęp do systemu, system przechowuje tylko i wyłącznie dane naukowe.

11 Podręcznik użytkownika

Opis użycia:

System istnieje z perspektywy użytkownika aby wstawić plik do zadnotowania oraz uzyskać plik wynikowy, po wstawieniu obliczanie wartości może zająć długo tak więc na zapytanie /getResult lub sprawdzenie statusu na Frontend danego zadania uzyskamy odpowiedź że zadanie jest przetwarzane lub plik wynikowy

Poprzez frontend:

Aby użyć systemu należy wejść na stronę http://<adres_serwera> oraz wstawić plik gotowy do zadnotowania, informacje o stanie zadnotowania pliku można zobaczyć na tej też stronie

Dokumentacja api:

W repozytorium można znaleźć zestaw zapytania napisanych przy użyciu narzędzia curl.

Paths

POST /uploadFile Create Upload File

Responses

Code	Description	Links
200	Successful Response *Content* **application/json**	No Links
422	Validation Error *Content* **application/json**	No Links

POST /getResult Get Result

Responses

Code	Description	Links
200	Successful Response *Content* **application/json**	No Links
422	Validation Error *Content* **application/json**	No Links

POST /getStatus Get Status

Responses

Code	Description	Links
200	Successful Response *Content* **application/json**	No Links

Code	Description	Links
422	Validation Error *Content* **application/json**	No Links

GET `/availableAlgorithms` Get Available Algorithms

Responses

Code	Description	Links
200	Successful Response *Content* **application/json**	No Links

POST `/getDetailedStatus` Get Detailed Status

Responses

Code	Description	Links
200	Successful Response *Content* **application/json**	No Links
422	Validation Error *Content* **application/json**	No Links

12 Podręcznik administratora

Dodanie nowego algorytmu (wymaganie klienckie):

Załóżmy że nasz nowy algorytm nazywa się XYZ

1. Stworzenie folderu XYZ wewnątrz algorithmworkers z plikami main.py, Dockerfile oraz __init__.py oraz requirements.txt
2. Plik __init__.py powinien zostać pusty, natomiast wewnątrz requirements należy umieścić

```
pika==1.3.2
```

3. Wewnątrz pliku Dockerfile powinny znaleźć się etapy konieczne do instalacji danego algorytmu (pobranie bibliotek oraz zbiorów danych) oraz instrukcje potrzebne do przygotowania naszego kodu:

```
# ========================================
#     Steps shared for all algorithms
# ========================================
WORKDIR /code
COPY ./XYZ /code/algorithmworkers/algorithm
COPY ./common /code/algorithmworkers/common

RUN pip install --no-cache-dir --upgrade -r
/code/algorithmworkers/algorithm/requirements.txt

CMD ["python3", "-m", "algorithmworkers.algorithm.main"]
```

4. Wewnątrz pliku main.py definiujemy klasę, która dziedziczy po klasie Algorithm z modułu algorithmworkers/common. Klasa ta powinna wyglądać następująco:

```
from ..common import Algorithm

class XYZ(Algorithm):
    def __init__(self):
        super().__init__("xyz")

    def run(self):
        # uruchomienie procesu algorytmu
        return returncode

    def prepare_input(self, input_file_path):
        # konwersja pliku input_file do formatu zgodnego z algorytmem

    def prepare_output(self, output_file_path):
        # konwersja pliku z algorytmu do formatu zgodnego z naszym
        # systemem

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

Nasz system dla każdego uruchomienia algorytmu tworzy folder tymczasowy (dostępny pod zmienną self.tmp_dir_name) w którym można tworzyć pliki tymczasowe potrzebne do zapisywania wyjść i wejść dla algorytmu.

Funkcja prepare_input na wejściu otrzymuje ścieżkę do pliku tsv z kolumnami: "Chr", "POS", "Ref", "Alt" oraz "HGVS"

Funkcja prepare_output na wejściu otrzymuje ścieżkę do pliku wynikowego, w którym powinna znaleźć się jedna kolumna o nazwie xyz i zawierać wyniki algorytmu.

Funkcja run **musi** poczekać aż algorytm ukończy pracę (np używając process.wait)

5. Następnie należy wyedytować plik algorithmworkers/config.py i w nim dopisać “xyz” do listy ALL_ALGORITHMS
6. Na koniec należy dodać odpowiedni wpis w docker-compose.yml, który uruchomi nasz algorytm

```
  xyz:
    build:
      context: ./algorithmworkers/
      dockerfile: ./xyz/Dockerfile
    working_dir: /code
    volumes:
      - shared-volume:/code/data
    depends_on:
      rabbitmq:
        condition: service_healthy
    networks:
      - shared-network
```

W celu lepszej implementacji algorytmów zalecamy zapoznanie się z plikami main.py dla algorytmów pangolin oraz spip.

Uruchomienie systemu

Budowa systemu z kodu źródłowego

(wymagane posiadanie docker, oraz uruchomionej w tle aplikacji Docker Desktop) po wykonaniu tych kroków aplikacja będzie gotowa do działania na localhost

- w pierwszej kolejności należy sklonować kod z publicznego repozytorium na github:

```
git clone https://github.com/wz7475/genetics-app
```

- następnie należy przejść do katalogu z repozytorium oraz wykonanie skryptu budującego i uruchamiającego komendą (należy mieć na uwadze że proces budowania może zająć nawet godzinę):

```
./build_and_run.sh
```

Instalacja i konfiguracja systemu

- instalacja systemu sprowadza się do wywołania komend docker-compose
- konfiguracja systemu znajduje się w pliku docker-compose.yml który znajduje się w repozytorium

Aktualizacja oprogramowania

- docelowo (gdyby serwer bigubu działał) to aktualizacja na serwerze będzie automatyczna
- w celu aktualizacji lokalnie należy wykonać komendę:

```
./build_and_run.sh
```

Zarządzanie użytkownikami i uprawnieniami

- aplikacja jest uruchamiana lokalnie dlatego każdy ma dostęp do wszystkiego
- zmiany do repozytorium mogą zrobić użytkownicy dodani jako contributors do repozytorium (członkowie zespołu)

Tworzenie kopii zapasowych i odtwarzanie systemu

- aktualnie po zbudowaniu aplikacji lokalnie ma się swoją kopię bazy danych, która, dopóki nie zostanie usunięta ręcznie to nie zostanie usunięta. W planach są skrypty które pozwalają na automatyczny backup przy każdym automatycznym wdrożeniu (do odpalenia lokalnie jeśli ktoś by chciał)

Zarządzanie zasobami systemu

- docker dostarcza możliwości zarządzania zasobami systemu ponieważ wszystkie komponenty są skonteneryzowane

13 Podsumowanie

Rzeczy które chcieliśmy dodać, ale poprzez fizyczne blokady nie byliśmy w stanie:

- Na serwerze bigubu.ii.pw.edu.pl jest gotowy skrypt odpowiedzialny za wdrażanie aplikacji (CD - deployment), a w github actions przygotowany workflow odpowiedzialny za deployment - brakuje jedynie wygenerowanego klucza ssh dzięki któremu ten pipeline by mógł uruchamiać ten skrypt. Nie jesteśmy w stanie jednak odzyskać klucza gdyż serwer bigubu przestał działać

Rzeczy o których wiemy ale nie było czasu ich zrealizować:

- oddzielny "namespace" dla wariantów oraz dla tasków w bazie Redis, aktualnie są to klucze w tej samej bazie oraz czyszczenie samych tasków/ samej adnotacji nie jest optymalne
- optymalizacja Dockerfile
- brak obsługi rozszerzenia vcf w pliku wejściowym ze względu na starą bibliotekę do Pythona i potrzebny duży nakład pracy na przerobienie tego do aktualnego standardu
- algorytm spip ma ograniczenie do 1000 linii
- kopie zapasowe baz danych
- dzielenie przychodzących plików do zadnotowania w celu rozdzielenia ich między wiele algorithm workerów

Mocne strony:

- System przygotowany pod optymalne dodawanie nowych algorytmów na czym zależało klientowi
- System działa zgodnie z założeniami klienta
- Klient był zadowolony z działa do tego stopnia że planuje go testować w praktyce

Zatwierdzam dokumentację.	…… …….. Data i podpis Mentora