Shape

Description automatically generated with low confidence**Технически университет – София**

Дипломна работа

„Дизайн и реализация на система за генериране на функционални тестове чрез машинно обучение по подадени описание и съществуваща семантика“

**Изготвил:**

Студент: *Иван Ивайлов Милев*

Факултет: *ФКСT*

Специалност: *КСИ*

|  |  |
| --- | --- |
| *Дата:* | **Научен ръководител:** |
| *Град София* | */д-р инж. Д. Андреев/* |

Факултетен номер: *121218022*

*ЗАМЕНИ СЪС ЗАДАНИЕЗАМЕНИ С ДЕКЛАРАЦИЯ ЗА АВТОРСТВО*

Съдържание

[Въведение 4](#_Toc107870473)

[I. Анализ на съществуващи решения 5](#_Toc107870474)

[AthenaTest [1] 5](#_Toc107870475)

[GitHub Copilot [2] 6](#_Toc107870476)

[EvoSuite [3] 7](#_Toc107870477)

[Randoop[4] 9](#_Toc107870478)

[Проучване 9](#_Toc107870479)

[Language Models are Few-Shot Learners [5] 9](#_Toc107870480)

[Evaluating Large Language Models Trained on Code[6] 12](#_Toc107870481)

[Improving automatically generated code from Codex via Automated Program Repair[7] 15](#_Toc107870482)

[Abstract Syntax Trees[8] 15](#_Toc107870483)

[A Pre-Trained Model for Programming and Natural Language [10] 17](#_Toc107870484)

[Модели 19](#_Toc107870485)

[Обобщение 20](#_Toc107870486)

[II. Функционално описание на разработката 21](#_Toc107870487)

[Цел 21](#_Toc107870488)

[Описание 21](#_Toc107870489)

[Програмен език 22](#_Toc107870490)

[Библиотеки и инструменти 22](#_Toc107870491)

[NLTK [11] 22](#_Toc107870492)

[Pandas [12][13] 22](#_Toc107870493)

[PyDriller [14] 23](#_Toc107870494)

[AST [15] 23](#_Toc107870495)

[Open AI [16] 26](#_Toc107870496)

[Jupyter[17] 27](#_Toc107870497)

[Anaconda [18] 27](#_Toc107870498)

[GitHub Actions 28](#_Toc107870499)

[Интегрирана среда за разработка [19] [20] 28](#_Toc107870500)

[Обобщение 28](#_Toc107870501)

[III. Описание на реализацията на разработката 29](#_Toc107870502)

[Структура на софтуера 30](#_Toc107870503)

[Подсистема за генерация на код 30](#_Toc107870504)

[Подсистема за валидация на резултатите 40](#_Toc107870505)

[Обобщение и бъдещо развитие 47](#_Toc107870506)

[IV. Използвани данни и проведени експерименти 47](#_Toc107870507)

[Експеримент 1 47](#_Toc107870508)

[Изводи: 48](#_Toc107870509)

[Експеримент 2: 49](#_Toc107870510)

[Тестови извадки: 49](#_Toc107870511)

[Изводи: 51](#_Toc107870512)

[Експеримент 3 51](#_Toc107870513)

[Тестови извадки: 52](#_Toc107870514)

[Изводи: 54](#_Toc107870515)

[Експеримент 4 55](#_Toc107870516)

[Тестови извадки: 55](#_Toc107870517)

[Изводи: 57](#_Toc107870518)

[Експеримент 5 57](#_Toc107870519)

[Тестови извадки: 60](#_Toc107870520)

[Изводи 61](#_Toc107870521)

[Експеримент 6: 61](#_Toc107870522)

[Тестови извадки: 62](#_Toc107870523)

[Изводи: 64](#_Toc107870524)

[Експеримент 7: 64](#_Toc107870525)

[Тестови извадки: 64](#_Toc107870526)

[Актуална версия: 65](#_Toc107870527)

[Тестови извадки: 68](#_Toc107870528)

[Изводи: 69](#_Toc107870529)

[Допълнително обучение на GPT-3/Codex 70](#_Toc107870530)

[V. Примерна употреба и валидация на системата 71](#_Toc107870531)

[Пример: 72](#_Toc107870532)

[VI. Заключение 75](#_Toc107870533)

[VII. Използвана литература 76](#_Toc107870534)

[VIII. Приложение 78](#_Toc107870535)

# Въведение

В днешно време софтуерът е навсякъде и конкуренцията в софтуерният сват става все по-голяма. Всяка компания се цели да има най-добрия продукт за съответната категория. Това налага и стремежа на технологичните компании към това да могат да доставят възможно най-бързо новите си продукти и техните нови функционалности. Новите функционалност обаче не трябва да са за сметка на качеството на софтуера. За да се подсигури то следва продуктите да се тестват. Тестването е една от най-важните фази при разработката на софтуер, значимостта на тестването е отбелязана от всички компании, които инвестират десетки хиляди в тестери и инженери, които да разработват автоматични тестове и системи, които да ги изпълняват като продължителна интеграция и продължително доставяне. Един критичен дефект на системата би могъл да значи фалит за една компания.

За съжаление времето за доставяне на продукт често е най-големият фактор, което налага правенето на компромиси – пропускане на създаването на автоматизирани тестове. Писането на такива тестове е задача, която отнема не малко време и е не тривиална. Такива компромиси водят до големи разходи на компаниите. Това което се случва е, че с нарастването на софтуера започват да се появяват дефекти, които съществуват дълго време без да бъдат забелязани. Това от своя страна прави проследимостта им трудна, което забавя и оправянето на такъв дефект. Такива проблеми бавят целия процес и оскъпяват разработката на даден продукт многократно.

В този контекст автоматизираното тестване е основополагащ елемент в изграждането на софтуер. То е ключов елемент в процеса на работа, който дава увереност на разработчиците, че качеството на продукта не е засегнато от новата функционалност, която е добавена. За това и са разработени множество софтуерни рамки (frameworks), които да улесняват писането на различни видове тестове. Тестовете са много и различни видове, от които основните са модулни, интеграционни, функционални и на цялата система. Функционалните и интеграционни тестове се грижат за това да валидират дали дадена подсистема работи както бихме очаквали, те са ключови за верификацията на правилната работа на системата. За съжаление тестовете нямат директен отпечатък върху крайния продукт и поради факта, че отнемат много време често се пропускат.

Идеята зад тази дипломна работа е да намали цената на писането на тестове, дори да я направи клоняща към нула. За целта е предложен подход, с който се генерират автоматизирани тестове, които да бъдат валидирани от програмист. Дипломната работа се фокусира върху генерацията на интеграционни и функционални тестове, които са ключов елемент от плана за тестване на даден софтуер. Избрани са тези тестове, защото за разлика от модулните проверяват цели подсистеми и така гарантират работата на цялата система, докато при модулните тестове е възможно да се пропуснат дефекти при комуникацията на различните модули. Функционалните и интеграционните тестове се имплементират с код, който комуникира с програмния интерфейс на програмата, докато често пъти тестовете на цялостната система се правят ръчно или с различни софтуерни рамки според вида на програмата, което ги прави трудни за автоматизация.

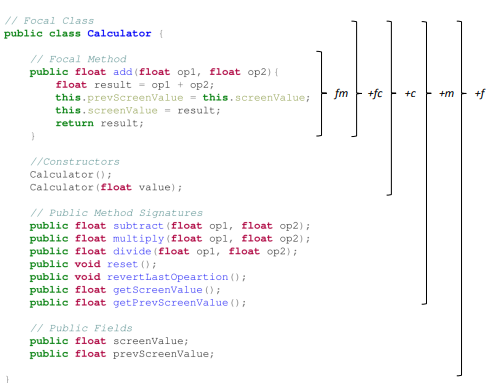
Дипломната работа цели да подобри качеството на разработвания софтуер като намали цената и времето, което необходимо за разработването на автоматизации по валидацията на качеството. По този начин програмистите ще имат повече време за разработка на нови функционалности без това да е за сметка на качеството им.

# Анализ на съществуващи решения

Тематиката за автоматичното генериране на код и в частност тестове не е нова, съществуват множество системи и статии по тематиката. Част от тези системи имат за цел генерация на модулни тестове, а други цялостна генерация на текст и код, които да бъдат използвани от програмистите с цел да пестят тяхното време. Софтуерни системи и статии по въпроса ще бъдат разгледани.

## AthenaTest [1]

AthenaTest e е разработка за генерация на модулно тестване използвайки трансформатори и фокусен контекст. Тя се различава от други с това, че не е базирана на метрика за покритие на тестовете, а на контекста, който трябва да бъде валидиран от тест. При тази разработка подходът е от тип “sequence to sequence”, който се състои от процес с две стъпки при тренирането на модела. Първата стъпка е тренирането на модел върху голям обем от Java код, който после да бъде допълнително трениран (fine-tuning) за генериране на код. За целта е създаден най-големият набор от именувани данни за модулни тестове с техните контексти – тестван метод, дефиниция на класа на тествания метод, конструктор на тествания клас, методи на тествания клас и полета. Данните са кръстени Methods2Test и могат да бъдат намерени и използвани свободно в GitHub.



Фигура Нагледна репрезентация на различните фокусни контексти

Целият процес за генериране на тестове е както следва:

1. Събират се данни – съществуващи тестове и контексти
2. Тестовете се асоциират с техните контексти
3. Използва се общият модел
4. Прави се допълнителното трениране, след което вече се генерират тестовете

Процеса може да се види на фигура 2.

Graphical user interface, diagram

Description automatically generated

Фигура процес по трениране на модел спрямо фокусен контекст[1]

## GitHub Copilot [2]

GitHub Copilot е приложение, чиято цел е да бъде пълноценен помощник, който да се държи като втори, помощник програмист. Самото приложение е имплементирано като плъгин (plugin) на средата за разработка Visual Studio Code. GitHub Copilot работи по механизъм подобен на този на обикновеното подсказване при въвеждане на код, разликата се състои в това, че вместо да допълва една дума може да допълни цяла функция или метод на база на дадено описателно име и/или коментар, който описва действието на съответния фрагмент от код.

Реално GitHub Copilot представлява генератор на код, генериращ от дадено описание. Това е постигнато с помощта на изкуствен интелект и по-конкретно с модел разработен от OpenAI в колаборация с Microsoft. Моделът е много сходен на публично достъпния модел на OpenAI Codex и е базиран на огромния модел с 12 милиарда параметъра трансформатор (transformer) GPT-3. Моделът е трениран използвайки хиляди публични хранилища за код включително и всички такива от GitHub. Моделът, съответно и GitHub Copilot, се справя най-добре с код на Python, тъй като най-много такива данни са присъствали при тренирането на модела.



Фигура диаграма от високо ниво за механизма на работа на GitHub CoPilot [2]

За да може да генерира предложения GitHub Copilot използва отворения файл в Visual Studio Code, както и другите файлове, които са част от отвореното работно пространство. С този контекст моделът може да направи предложение, което е конкретно за контекста на проекта.

На практика продуктът работи най-добре при писане на малки функции с описателни имена и точно описание на какво трябва да извърши функцията.



Фигура Автоматично допълнение от GitHub CoPilot[2]

## EvoSuite [3]

EvoSuite е система, която може автоматично да генерира модулни тестове за класове написани на Java. Това се постига, чрез използване на хибриден подход, който генерира и оптимизира цели тестови пакети, които се стремят към постигане на високо покритие върху тествания клас. Самите тестове са малки по размер и ефективни, използвайки достатъчно клаузи от тип “assert”, които да са достатъчни за описанието и поддръжката на съществуващата логика. Системата има за цел да предоставя тестове, на които разработчиците да разчитат при по нататъшни пренаписвания на различни фрагменти. Тези тестове също се очаква да гарантират сегашната работа на софтуера.

Системата има следните функционалности:

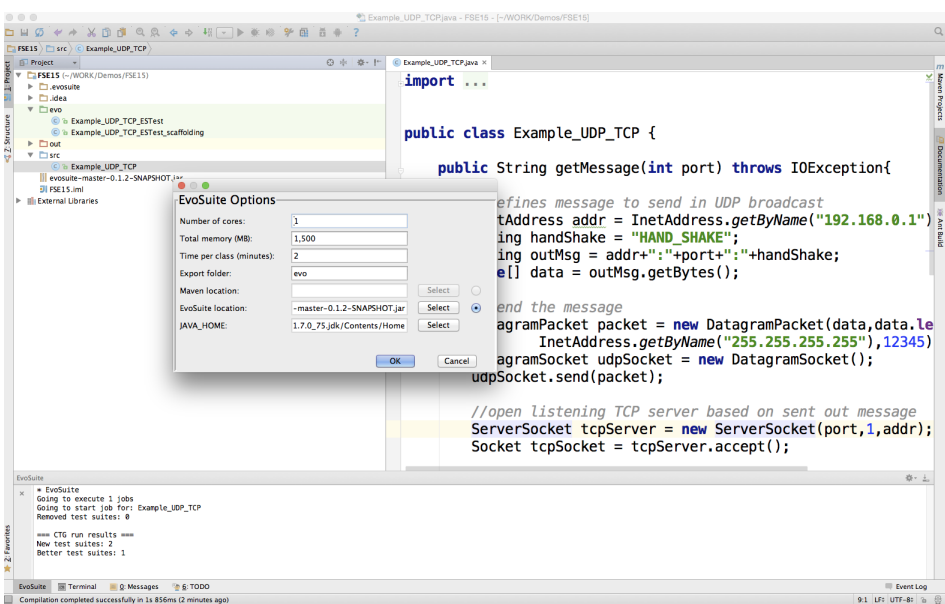
* Генериране на Junit4 тестове за избрани класове
* Оптимизация на тестове за конкретна критерия на покритие – линии, разклонения, изходи и тестове за мутации
* Генерираните тестове се минимизират – размерът им се свежда до минимум запазвайки същото ниво на покритие
* Генериране на Junit проверяващи клаузи, които да проверяват моментното състояние на класа
* Пускане на тестовете във виртуална среда за по-голяма сигурност
* Виртуална файлова система
* Виртуални компютърна мрежа

Системата може е достъпна по няколко начина:

Първият, от които е като инструмент с конзолен интерфейс, за да се използва по този начин се сваля .jar файл, чрез който се използва системата.

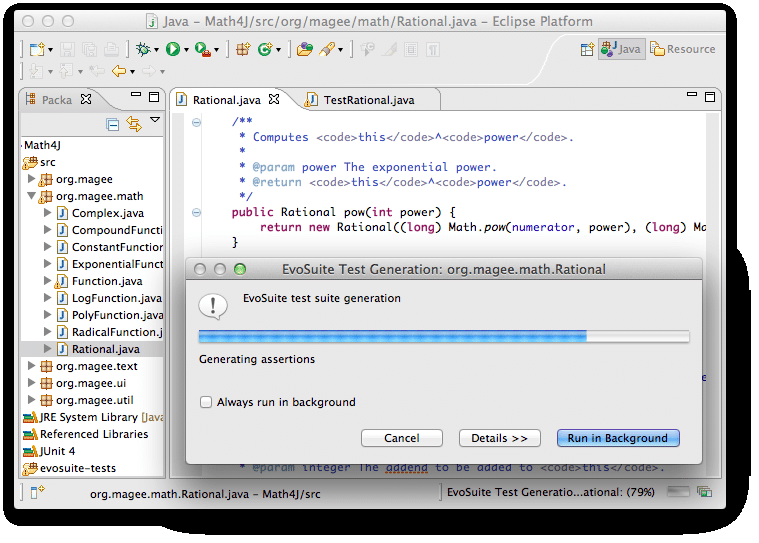
Вторият вариант е като притурка (plugin) към средата за разработка. Към момента такива притурки съществуват за InteliJ IDEA и Eclipse.

След като се инсталира нужната притурка за InteliJ IDEA, тестове се генерират като се избере клас, група от класове или пакет и чрез десен клик на мишката се избере генериране на EvoSuite Test. Потребителският интерфейс може да бъде видян на Фигура 5.



Фигура Потребителски интерфейс на EvoSuite в InteliJ IDEA

При Eclipse приложението работи по аналогичен начин:



Фигура Потребителски интерфейс на EvoSuite в Eclipse IDE

Освен тези два плъгина съществува такъв и за системата за непрекъсната интеграция – Jenkins. С нея може да се добави стъпка, която да генерира и изпълнява тестове за всички новодобавени класове.

## Randoop[4]

Randoop е система за генериране на модулни тестове за Java. Системата генерира тестове с Junit. Част от функционалността на Randoop съществува и за .Net.

Алгоритъмът, който използва Randoop за генериране на тестове е чрез итеративно генерация на произволни тестове, като се използва обратна връзка от генерирания тест, за да се подобри и стигне до финалната му форма. На практика се генерират псевдо-произволни извиквания към методите и конструкторите на тествания клас, след което генерираният тест се пуска и спрямо резултатите от извиканите методи се създават проверки.

Тази система се използват главно за намиране на бъгове в съществуващите системи. Често пъти генерираните тестове не са лесно четими за програмистите и за това се са само част от системата за непрекъсната интеграция.

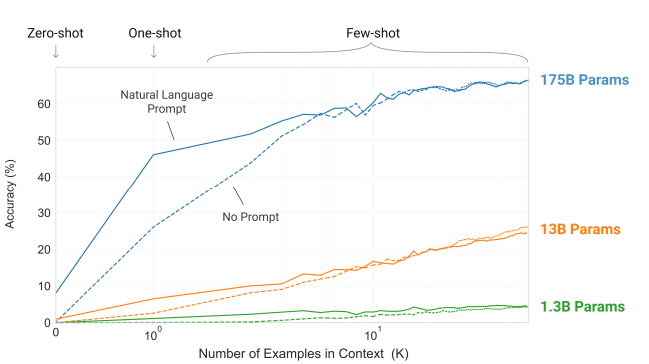
## Проучване

### Language Models are Few-Shot Learners [5]

В този научен труд се изследва как мащабирането на модел от тип трансформатор взаимодейства с неговата продуктивност. Като се изследват няколко модела с основна разлика в броя параметри. Крайният продукт е наречен GPT-3 и използва 175 милиарда параметъра, като е обучаван върху 400 милиарда „думи“ (token-а). Разглежда се и концепцията за даване на нула, един или няколко примера на модела – изводите са, че моделът може се справя изключително добре за много задачи, без да има нужда от допълнително трениране (fine-tuning).

Практически статията не показва нова архитектура на модела, тя е базирана на стария модел разработен от OpenAI – GPT-2, основната разлика е в големината – GPT-3 е много мащабирана версия на своя предшественик.

На графиката на Фигура 7 могат да бъдат видени как с увеличаване на броя параметри се подобрява и увеличава производителността на модела за контекстни задачи. От това, че с увеличаването на дадените примери се увеличава и производителността следва и че създаденият модел разбира от дадената му допълнително контекстната информация.



Фигура GPT-3 точност спрямо брой дадени примери и брой параметри на модела [5]

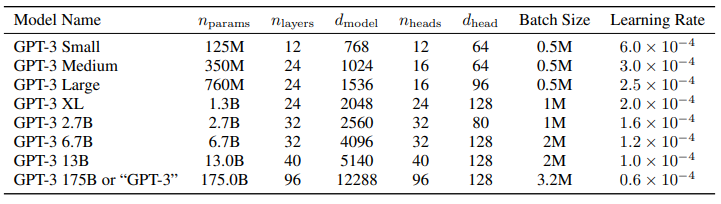
В статията е отделено специално внимание на ученето чрез нула или няколко примера.

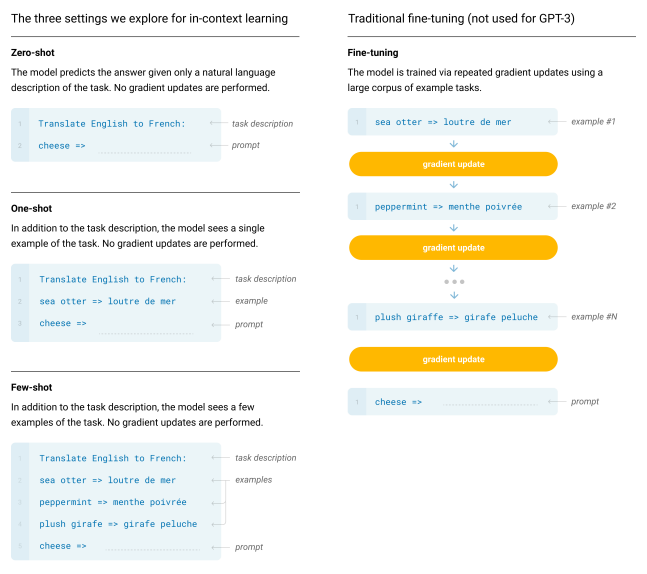
1. Допълнително-обучение (Fine-tuning):   
   Това е един от най-популярните похвати през последните години. Идеята зад него е да се използва голям съществуващ модел, чиито тежести да бъдат актуализирани чрез „допълнително трениране“ на модел със стотици хиляди надписани примери. Основният плюс на този подход е, че дава много добри резултати за конкретни тестове, основният минус е, че за всяка задача са нужни големи количества качествени данни, които са надписани. Което значи и че моделът може да изгуби своите „общи“ знания, което е сходно с това моделът да е твърде специфичен към данните, с които е обучаван (over fit).
2. Няколко-изстрела (Few-shot) – няколко примера:   
   Този термин се използва при настройка на модела при която се дават няколко демонстрационни примера заедно със задачата, която моделът трябва да изпълни. Важно е да се отбележи, че при това извикване не се променят тежестите на мрежата.
3. Един-изстрел (One-shot) – един пример:   
   Този термин, носи същото значение като няколкото-изстрела, единствената разлика е, че този път се дава само един пример за задачата.
4. Нула-изстрела (Zero-Shot) – нула примера:   
   При този вид извикване на модела, не се дават никакви примери и се изследва как моделът се справя без да му се дават никакви примери.

Всички тези извиквания на модела могат да бъдат видени на долната фигура 8.

Резултати и сравнение на това как моделът се подобрява с мащабирането си могат да бъдат видени в Таблица 1.

Таблица Сравнение на модели с различна мащабираност на GPT-3





Фигура Примери за нула примери, един пример, няколко примера и допълнително обучение.

### Evaluating Large Language Models Trained on Code[6]

Evaluating Large Language Models Trained on Code е статия описваща процеса по създаване и оценка на най-големия модел за генериране на код от OpenAI – Codex. Codex е базиран на най-големия езиков модел на компанията GPT-3. Тъй като GPT-3 показва не лоши резултати в генерацията на код на Python, докато моделът не е трениран за това компанията смята, че при допълнително трениране на модела е напълно възможно да се постигнат много добри резултати.

За целта се прави допълнително трениране върху GPT-12B. Използваните данни за тази операция са основно от хранилища за код в GitHub, данните съдържат в себе си 179 GB уникални python файла, като всеки файл е не повече от 1 MB. След което е направена допълнителна филтрация, която цели да премахне всички генерирани файлове, резултата е 159 GB данни.

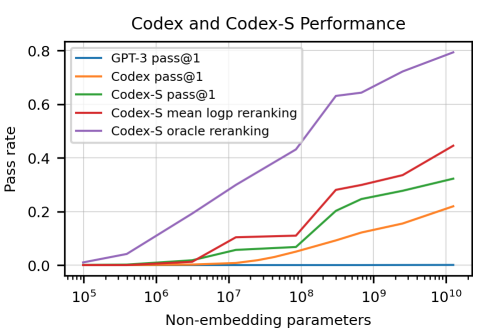
След тренирането на Codex разбира се идва и най-важната част – оценката на модела. Това е трудна и сложна задача, защото генерирането на решение не може да бъде оценено както генерацията на текст. Един от най-популярните подходи за оценка на генериран текст и код е оценката BLEU.

BLEU е абревиатура за bilingual evaluation understudy – двуезична оценка. Тази оценка се появява като решение за оценяването на качеството на генериран текст при машинни преводи. Тоест оценката дава количествена репрезентация на качеството на превод направен от модел в контекста на моделите базирани на машинно обучение. Идеята на тази оценка е да даде колко близък е преводът направен от машината до този направен от човек, който е професионален преводач. От тук следва и това че оценката винаги е между 0 и 1. [25]

Начинът на работа на BLEU се базира на два входа: низ кандидат и референтен низове. Резултатът би бил 1 ако кандидат низът е сходен до референтните низове и 0 ако сходство отсъства. Тъй като в естествения език думите имат различно значение според контекста си BLEU поддържа и така наречените n-gram-и, които представляват дума заедно със съседните ѝ няколко (общо n на брой). [25]

BLEU работи изключително добре в контекста на преводите, но за жалост при генерирането на код, чиято идея е да решава даден проблем не е най-добрият начин за оценка. Програмистите оценяват даден метод, клас, модул спрямо това дали той работи правилно или не, а не по това колко сходно е написаното решение до такова, което вече съществува. Тази оценка за това дали даден код работи правилно или не типично се автоматизира чрез писането на автоматични тестове. Този вид оценяване гарантира качество на кода и затова дори е създаден начин на работа базиран на него – Разработка чрез тестове (Test Driven Development).

От тук следва, че е много по-логично оценката на модела да се случва по същия начин – дали за генерирания код минават написани предварително тестове. За целта и OpenAI създават 164 проблема заедно с модулни тестове за всеки. Тези проблеми са така направени, че да могат да тестват различни специфики на езика, алгоритми и проста математика, които са нужни за решенията. Тези данни могат да са свободно достъпни и могат да бъдат използвани от всекиго. Върху тези проблеми Codex се справя изключително добре както се вижда от графиката на Фигура 9.



Фигура Минаващи примери спрямо брой параметри за Codex

За жалост резултатите не са толкова добри при тестове на реални проблеми. За това се налага и нов подход при оценяването на модела.

Както вече бе описано метрики базирани на точни съвпадения са неподходящи за оценка на генериран код, защото не обхващат комплексната природа на решенията на проблем.

За това и изключително по-добър вариант в случая е оценка, чрез тестове. Тъй като моделът може да генерира различни решения то се вкарва в употреба и така нареченото оценяване eval@k.[26] За този подход се генерират k примера за всеки проблем, като проблем се смята за решен ако за един от генерираните примери минат всички тестове.

Тъй като изискването за това един пример да реши всички тестове не е съвсем точна оценка за това дали Codex генерира работещи решения е избрана вариация на този подход, при която се генерират n > k примера за всяка задача и след това се взима бройката на тези решения, за които минават всички тестове.

На практика формулата за оценка е следната:

Този подход е много по-подходящ и показателен за това как се справя моделът с генерирането на работещи решения по зададено описание.

Таблица Сравнение на GPT-NEO, GPT-J (модели базирани на GPT-3 https://github.com/eleutherai/gpt-neo), TabNine и Codex модели върху HumanEval данни.

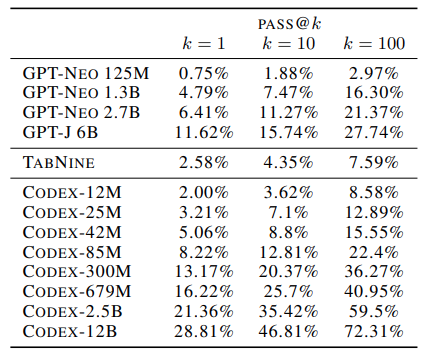
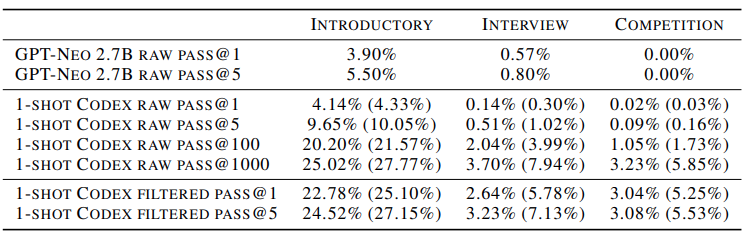
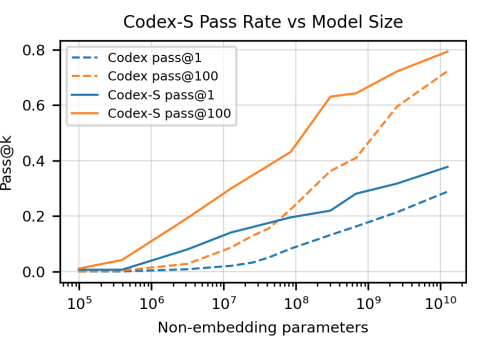


Таблица Сравнение на GPT-NEO и Codex върху данните от статията APPS[27][28]





Фигура Оценка спрямо големината на модела Codex

Моделът, който предоставят OpenAI е с изключително добри възможности в генерирането на код. Освен това статията дава и добри насоки за това как трябва да се оценява генериран код, този вид оценки се използват

От графиката на Фигура 10 се вижда, че моделът на Codex е с изключително добри възможности в генерирането на код.

### Improving automatically generated code from Codex via Automated Program Repair[7]

В днешно време съществуват огромни модели от ранга на Codex и AlphaCode, които се справят изключително добре в решаването на множество и различни задачи от програмирането. Въпреки това се оказва, че генерираният код не се справя толкова добре с много от задачите – практически Codex има около 25% успеваемост при задачи за начинаещи и едва 3% за задачи от състезателно естество. Процентите на AlphaCode са съответно 20.36% и 7.75%. това се дължи на факта, че и двата модела са езикови трансформационни модели т.е. са базирани на последователността от думи, които получават и предполагат коя би била следващата дума. Това значи, че тези модели нямат добро разбиране за компютърните езици и съответно семантиката на програмите. Тази незадълбоченост води до чести синтактични и алгоритмични грешки, които сриват успеха на моделите в задачите.

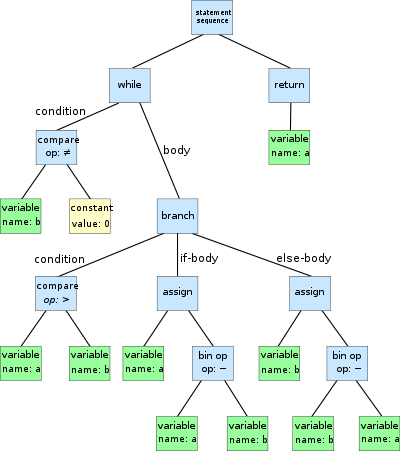
Затова в тази статия се разглеждат различните видове автоматична поправка на програмните грешки. Използвайки съществуващи такива системи се вижда веднага подобрение особени при трудните задачи за Codex това подобрение е от 5 до 9 решени задачи от общо 53.

Оказва се, че най-честите грешки, които допускат моделите при генерирането на код са грешки при използване на аритметичен оператор, грешно използване на типове, липса на затварящи скоби, кавички, защото моделът не е довършил генерацията. Съществуващите системи за Java помагат с част от тези проблеми. Освен тези инструменти се разглежда и самият Codex, който предоставя функционалност, с която може да редактира подаден код спрямо дадено описание. Изводът, до който се стига е, че ефективността на Codex е силно зависима от даденото му описание, като за по-трудни проблеми е важна точно локализация и конкретна информация.

Статията стига до изводите, че въпреки обещаващия си потенциал големите модели и конкретно Codex няма добри резултати. Традиционните методи за поправка на програми не помагат особено, но самият Codex в режим на редактиране има голям потенциал при правилно зададени инструкции. Голяма част от тези недостатъци произлизат от малкото познание на езиковите модели за програмната структура – абстрактни синтактични дървета, работа с типове и т.н.

### Abstract Syntax Trees[8]

В компютърните науки абстрактните синтактични дървета, още познати и като синтактични дървета са структура от данни, която репрезентира абстрактната синтактична структура на даден текст, който използва формален език - най-често сорс код. Всеки възел в дървото е конструкция, която участва в анализирания файл – if, while, дефиниция на променлива, return и т.н.



Фигура Пример за абстрактно синтактично дърво [9]

Абстрактното синтактично дърво се различава от конкретното синтактично дърво по това, че абстрактното дърво не съдържа в себе си подробности от типа на какви кавички се използват за дефиницията на низове и други. Конкретните синтактични дървета често се използват за контекстен анализ на кода. Често пъти контекстните синтактични дървета се построяват като се обогати модела на абстрактното синтактично дърво.

Абстрактното синтактично дърво от своя страна се използва от много компилатори, които създават тази структура от данни като резултат от една от фазите на компилатора. Често пъти се използва за междинна стъпка на репрезентацията на програмата, която се използва от другите фази на компилатора.

Абстрактното синтактично дърво може да бъде променяно и използвано за оптимизации на кода без да се променя самият сорс код. По този начин компилаторът може да оптимизира програмата използвайки дървото за анализ и в последствие променяйки го, но в същото време запазвайки непроменен сорс кода на програмата. Друго много важно предимство на дървото е, че то не съдържа в себе си ненужни пунктуационни детайли – точки и запетайки за край на реда, скоби, кавички и други. Важно е да се отбележи и че във възлите на дървото се съдържа информация свързана с това къде се намира дадения възел, с помощта на тази информация компилаторите могат да дават конкретна информация за това къде и какъв проблем е възникнал по време на компилация на кода.

За всяко абстрактно синтактично дърво има следните изисквания:

1. Типовете на променливите трябва да се запазват заедно с тяхното местоположение.
2. Редът на изпълнимият код трябва да бъде указан експлицитно, ясно и еднозначно
3. Левите и десните компоненти на булеви операции трябва да се запазят в правилен ред.
4. Идентификаторите и стойностите, които са им присвоени трябва да бъдат запазени

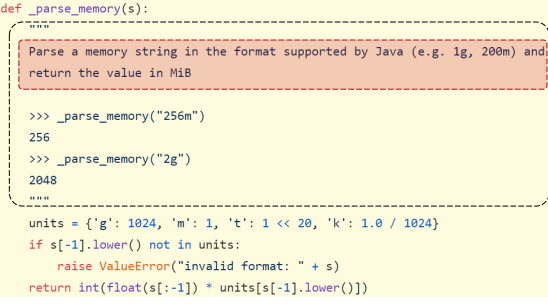
В контекста на дипломната работа структурата от данни се използва за анализ и оценка на различни сорс код файлове.

### A Pre-Trained Model for Programming and Natural Language [10]

В статията на тема предварително трениран модел за програмни и естествени езици се представя моделът CodeBERT, който е предварително трениран модел върху бимодални данни за програмни езици и естествени такива.

Идеята на модела е даде генерализирана репрезентация за връзката между програмните езици и естествените в контекста на техническа документация, търсене на функционалност в кода и т.н.

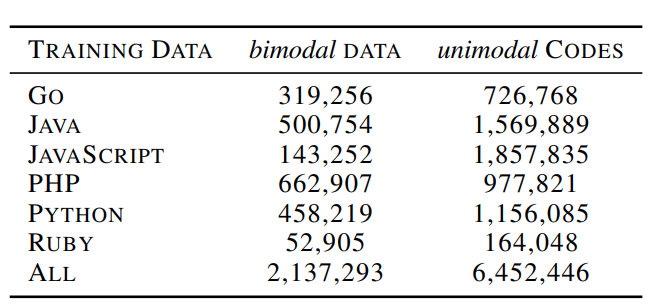
CodeBERT е с архитектура базирана на невроните мрежи - трансформатори. Моделът е трениран върху бимодални и унимодални данни. Бимодални са тези, при които съществуват програмен код и свързаната с него техническа документация (Фигура 12). Докато унимодлни данни са тези, които имат или само код или съответно само текст написан на естествен език (английски).



Фигура

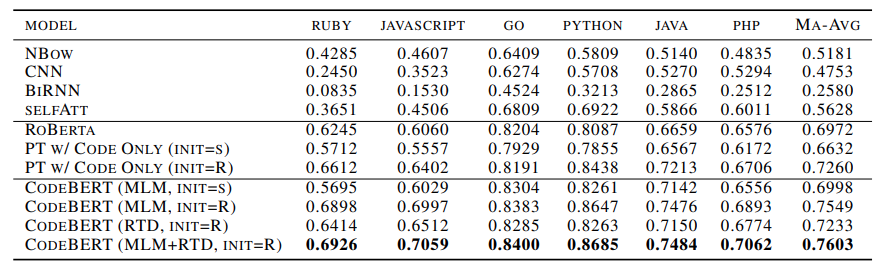
Моделът е обучен върху милиони данни, техният обем може да бъде видян в Таблица 4.

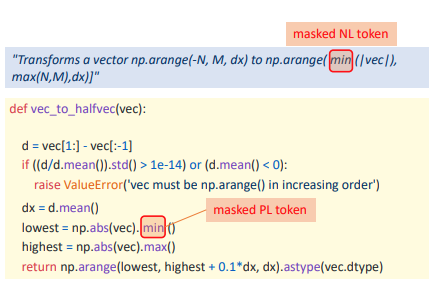
Таблица Данни използвани за обучението на CodeBERT



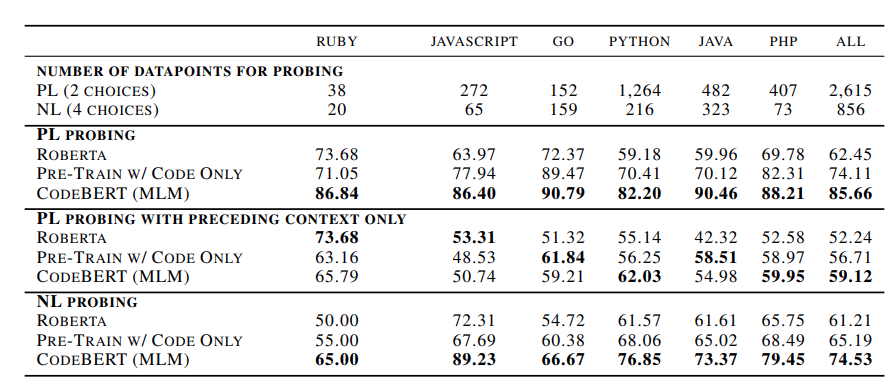
Моделът в предварително-обучената си форма се справя най-добре със задачи от тип локализация на код по дадено описание и генерация на документация от код. Освен това моделът се изследва и за това колко добре може да предположи какъв низ трябва да се сложи при маскиран такъв в код или документация. За изследването са маскирани думите min и max в различни места от кода и документацията му (Фигура 13). Резултати и сравнение от различни модели могат да бъдат намерени в Таблица 5.

Таблица сравнение на различни модели и тяхното представяне при локализация на код.





Фигура Маскиране на фрагменти от код за поправка от CodeBERT

Таблица 6 Резултати от проведени експерименти след допълнително трениране на CodeBERT според спецификата на задачата.

Всички показни данни от експерименти в Таблица 6 са направени след допълнително-обучение на модела, от това следва и нуждата от добри данни, с които моделът да бъде допълнително-обучен за конкретният проблем.

### Модели

От горе изнесената информация се вижда, че големите проучвания на тематиката с генерация на текст имат своите разлики – основно в обем и данни, с които са тренирани, също така и за задачите към които се стремят. Техните прилики, разлики и специфики са описани в Таблица 7.

Таблица 7 Сравнение между моделите CodeBERT, GPT-3, Codex

|  |  |  |  |
| --- | --- | --- | --- |
| Модел (име): | Архитектура | Мащаб | Цел |
| CodeBERT | Базирана на трансформатори | Параметри – 125 милиона  Данни – 8 милиона примера | Генерация на документация и локализация на код по описание.  Предвиден за допълнително-обучение за конкретни задачи свързани с генерация на код от текст и съответно генерация на текст от код. |
| GPT-3 | Базирана на трансформатори | Параметри – 175 милиарда  Данни – 300 милиарда думи (tokens) | Модел целящ се да има общо разбиране за естествения език. Най-големият модел за обработка на естествен език към момента, който се справя изключително добре на много и различни задачи свързани с естествения език особено ако бъде използван подхода за задаване на вход с няколко-изстрела.  Не е предвиден за разбиране и допълване на код без нуждата от допълнително-трениране, за което ще са нужни няколко стотин хиляди примера |
| Codex | Базирана на трансформатори | Параметри -12 Милиарда  Данни – 159 GB сорс код | Допълнително-трениран модел базиран на GPT-3, чиято цел е генерирането на код от описание на естествен език. Основната цел е да генерира функции, които извършват описаното в документиращ низ (doc string). |

## Обобщение

От изложената информация се вижда, че основния фокус в сферата през последните години е бил върху генерацията на модулни тестове. Като са разглеждани различни подходи, за да се постигне това – машинно обучение на база фокусен контекст и псевдо-произволна генерация.

От друга страна генерацията на текст и код набира скорост. Създаване на модели, които „намират“ връзка между естествен език и програмен код стават все по-популярни и в момента са едни от най-големите такива модели с възможности да генерират код по описание. От своя страна генерирането на код поражда нуждата за системи, които да могат да оправят допуснати от моделите тривиални грешки.

Направените изследвания свързани с генерирането на модулни тестове в комбинация със съществуващите модели за генериране на код създават предпоставка за това да се пристъпи към следващото ниво в генерирането на автоматизирани тестове - функционалното тестване.

# Функционално описание на разработката

## Цел

Целта на настоящата дипломна работа е разработката на система, която да генерира тестове базирани на критериите описани в потребителската история, която е разработвана от програмиста. Генерираните тестове са функционални и интеграционни. Идеята да съществува такъв инструмент е да се подобри продуктивността на програмистите в ежедневната им работа като се намали времето, което те инвестират в писане на автоматични тестове, а освен това и да се подобри качеството на разработвания продукт.

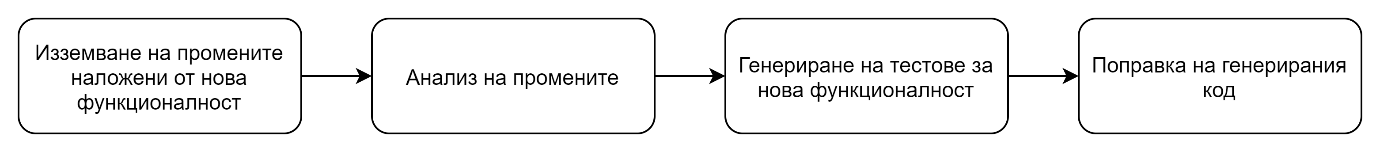
Идеята за автоматична генерация на код не е нова, както е описано в първа глава на дипломната работа съществуват разработки с различни подходи на тази тематика. От използване на изкуствен интелект до системи, които генерират псевдо произволни извиквания на съществуващи методи.

Важно е да се отбележи, че разработките, които съществуват на тематиката за генерация на тестове се фокусират основно и само върху генерация на модулни тестове. При тези проекти най-важно е разбирането за модула (класа) и няма нужда от знание за цялата системата докато в системата, която се разработва за тази дипломна работа е важно да се тества цяла функционалност, а не конкретен модул.

Решението, което е разработено в тази дипломна работа е добра демонстрация на концепцията – използване на дълбоко обучение за генериране на тестове. Към момента то може да се използва като начална версия на тестове, които да бъдат доразработени от програмиста или съответно инженерите по качеството.

## Описание

Дипломната работа има една основна задача – генерирането на функционални и интеграционни тестове. Тази генерация е разделена основно на три части: анализ на новата функционалност, генерация на тестове за новата функционалност и поправка на генерираните тестове. (Фигура 14)

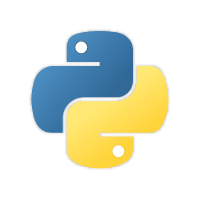


Фигура Основен алгоритъм на дейност на дипломната работа

Системата трябва да обслужва следният сценарии: при поява на нова версия на софтуера, който се изследва, да бъде засечена и подложена на анализ, от който да се продуцират функционални и/или интеграционни тестове за новата функционалност (част от новата версия).

За генерацията и анализа на новата функционалност ще бъдат използвани инструменти използвани при машинното обучение тъй като, точно този подход се използва за генерирането на код и поправката му. Имайки това предвид за програмната реализация на проблема с генериране на код за функционални тестове са използвани редица технологии. Основните технологии използвани за продукта са Python и модели разработени от OpenAI, както и популярният стек за работа с големи обеми от данни за Python, а именно Numpy, Pandas и популярната библиотека за работа с естествени езици NLTK. Освен тези библиотеки са използвани и синтактични дървета както и библиотеката PyDriller за по-лесна обработка на хранилища за код от тип git.

## Програмен език

Езикът за програмиране, който е избран за реализацията на дипломната работа е python. За разработката на този софтуер това е един от най-добрите избори, защото е най-популярният сред обществото, което се занимава с изкуствен интелект и анализ на данни. Нормалното следствие от популярността на езика в тези среди е и изключително добрата му поддръжка на тематиката изразяваща се в множество различни библиотеки, които се разработват за анализ и обработка на данни както и такива свързани с машинно обучение.

## Библиотеки и инструменти

### NLTK [11]

Библиотеката, която е избрана за обработка на естествен език е NLTK. NLTK е една от водещите платформи за разработка на програми на Python, които работят с данни от естествен език. Библиотеката предоставя лесни за използване интерфейси включвайки в себе си множество ресурси от данни. Библиотеката съдържа в себе си множество под-библиотеки за класификация, токенизация, коренуване, парсиране, определяне на частите на речта, определяне на „емоцията “ и други. Основните причини за избора на тази библиотека е това, че тя е създадена за Python и в себе си съдържа всичко, което би потрябвало в разработката на дипломната работа и далеч повече. Oсвен това е една от най-използваните, с което идват и множеството ресурси свързани с нея.

### Pandas [12][13]



Pandas е библиотека с отворен код, чието начало е поставено през 2008 година като през 2009 кодът на библиотеката става публично достъпен. Името на библиотеката идва от думите на английски за панел и данни - “**Pan**el” и “**Da**ta”. Библиотеката предоставя функционалности за обработка на големи масиви от данни. Основни нейни функционалности са:

* Бързи и ресурс ефективни обекти от тип рамки за данни (DataFrame), които са предвидени за манипулация и индексация
* Инструменти за сериализация и десериализация – запазване на данните в често използвани формати като CSV, Microsoft Excel, SQL и HDF5
* Автоматична подредба на данните като се използват етикети, а не само цифрови индекси. Библиотеката следи за липсващи данни.
* Размера на матрицата с данни може да бъде сменян лесно и бързо. Т.е. операции с модификация на колони и редове са позволени и работят бързо.
* Мощен механизъм за групиране (филтриране) на данните по критерии
* Висока производителност при сливане на големи обеми от данни
* Висока производителност на библиотеката – имплементирана е с технологии от ниско ниво, които да подсигурят тази производителност, а именно C и Cython.
* Една от най-използваните библиотеки в академичните среди. Широко използвана в много индустрии – Финанси, Икономика, Невронаука, Статистика, Реклама, Анализ на големи данни.

Тази библиотека бе избрана за разработката на дипломната работа тъй като реализацията на генериране на автоматични тестове изисква работа с голям обем от данни – съществуващ и генериран код.

### PyDriller [14]

PyDriller е библиотека, която дава абстракция и интерфейс върху командите и работата с Git хранилище. Идеята на библиотеката е да предостави възможност на различните разработчици да анализират и обработват данни от различни хранилища на код по лесен начин.

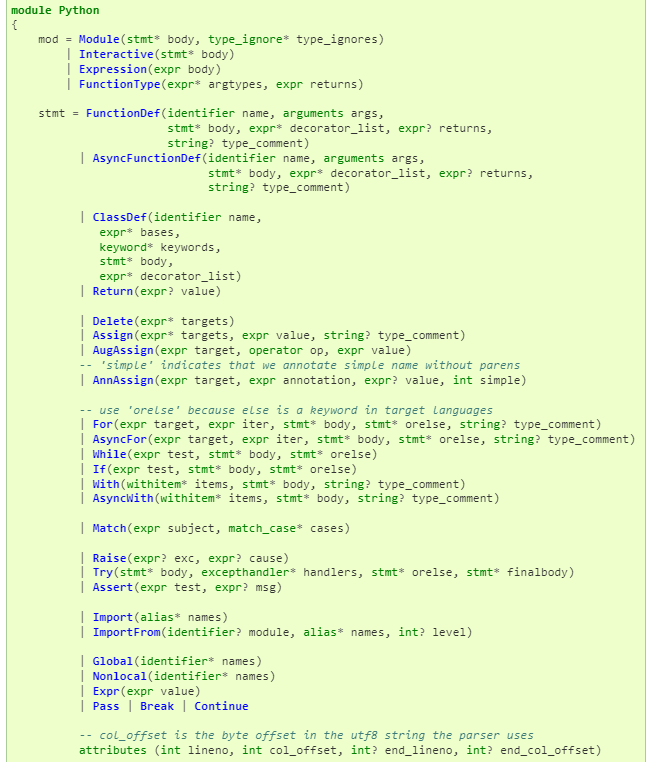
Библиотеката се състои от 3 основни обекта – хранилище (Repository), ревизия/версия (Commit) и модифициран файл (Modified File).

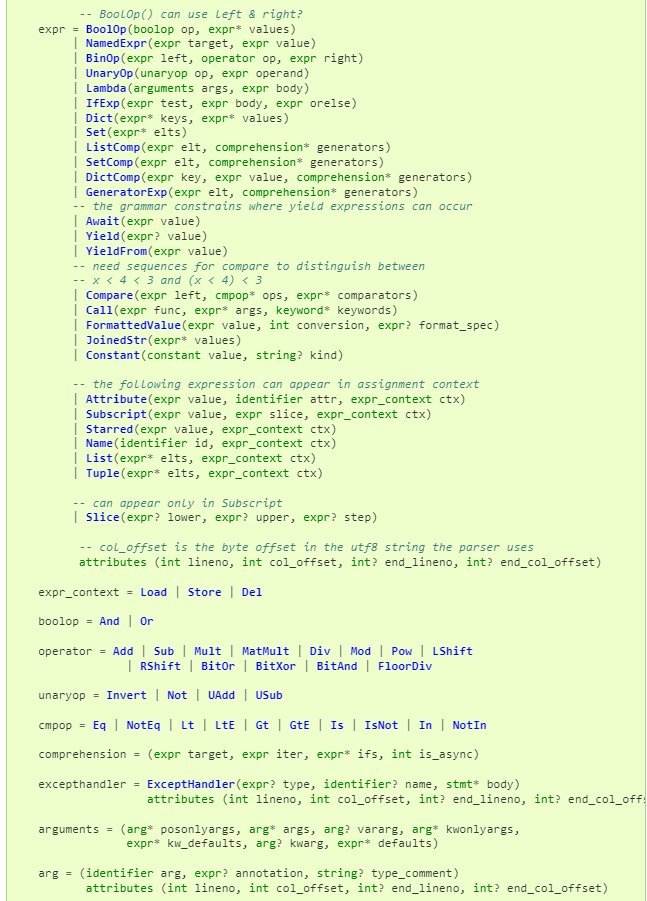
За целта на дипломната работа се използват различните ревизии и съответно от всяка се разграничават по-съществените файлове, тези които са медиатори и дефинират конкретна функционалност от тези, които са листа на дървото на зависимостите на софтуера.

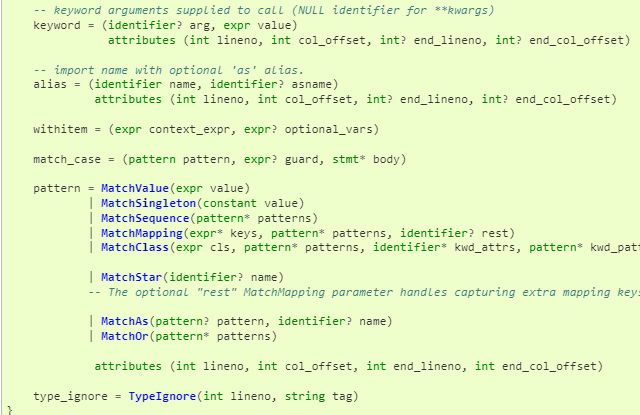
### AST [15]

AST – на български абстрактно синтактично дърво е модул вграден в езика Python, който се използва във всички програми написани на езика. Неговата цел е да разбира и работи със синтактическата граматика на езика.

Модулът дава следната абстрактна граматика върху езика:







### Open AI [16]

Open AI е компания, която се занимава с проучване и създаване на системи базирани на изкуствен интелект. Целта на компанията е да постигне изкуствен интелект с общо предназначение, който да помага на човечеството.

Open AI предоставя множество и различни модели, като интернет услуги. Достъпът и използването им става посредством REST и HTTP. За по-лесна употреба е създадена библиотека за Python, която обгръща тези заявки в удобен за използване Python интерфейс.

За целите на дипломната работа се използват основно два модела – GPT-3 и Codex.

Избрана е тази библиотека, защото тя има най-добро представяне в генерирането на код. Работата с тези модели е от наречения от някои Software 3.0, където вече важно е какъв вход ще се подаде на модела, за да може той да генерира резултати.

Конкурентите на тези модели нямат достатъчно добро представяне без нуждата от допълнителна настройка (fine-tuning), което е трудна и скъпа задача.

### Jupyter[17]

Jupyter е проект с отворен код. Създаден е през 2014 година като прераства от проекта IPython, който е проект за работа с езика python в команден ред (read-eval-print loop). Това, с което Jupyter надгражда и в частност тетрадките, които предлага е възможност за писане на код в така наречените блокове код, които могат да бъдат изпълнени директно в тетрадката, като контекста на изпълнения фрагмент остава в паметта. Практически тетрадката наподобява спряна с дебъгер програма, която може да бъде дописвана и изпълнявана блок по блок, като може предишни блокове да се преизпълняват. Това позволява изключително бърза итерация на различни експерименти, като се запазва състояние, което се смята за стабилно от разработчика. Точно това прави и платформата изключително популярна в света на машинното обучение, защото то изисква работа с големи обеми от данни върху, които да бъдат правени експерименти. С помощта на тази система, няма нужда големите обеми от данни да се зареждат в паметта многократно, също така ако е нужда някаква обработка тя може да бъде извършена един път и след това да се експериментира с обработените данни.

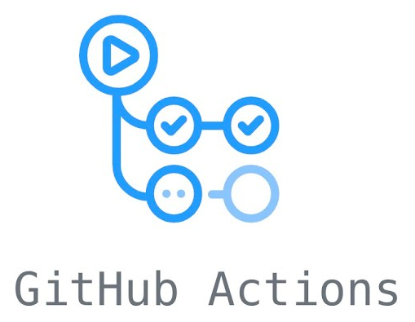
За целта на тази дипломна работа има нужда от сканиране, преглеждане и анализ на големи обеми код, което отнема съществено време. За това и е избрана тази система, с нейна помощ процеса по изграждането на инфраструктурата за анализ на хранилища за код бе изградена много по-бързо тъй като намали многократно времето, което би отнело да се изследват различни решения, които минават и обработват хранилищата за код.

### Anaconda [18]

Анаконда е водещата платформа за пакетен мениджър в изчислителните науки за езиците Python и R. За разработката на тази дипломна работа голяма част от използваните библиотеки са точно от този домейн на програмирането, което поставя платформата пред най-големият пакетен мениджър в Python света – pip. Друг бонус на платформата е, че се поддържа от PyCharm, както и прави опит за платформена независимост. Тази независимост се изразява в записването на библиотеките, на които зависи нашата среда по начин, чрез който при смяна на системата например от Windows на Linux ще потърси дали библиотеката съществува за Linux и ако има валидна версия, ще инсталира нея.

В разработката на системата за генериране на тестове анаконда се използва както за пакетен мениджър така и за виртуална среда. Виртуалната среда дава възможност за избор на конкретна версия на език за разработка, като средата е напълно базисна – има само базовите библиотеки по начало. Това позволява инсталация само на нужните за проекта библиотеки, което в последствие улеснява преноса на системата – на нова машина се създава същата виртуална среда и проекта е готов за експлоатация.

### GitHub Actions

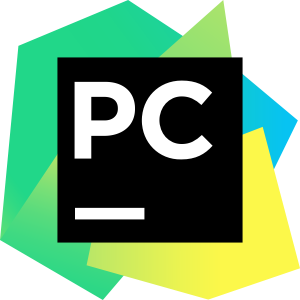
Тъй като дипломната работа по своята същност генерира артефакти, които трябва да бъдат част от процеса на продължително интегриране или построяване логичното нещо е разработения продукт да стане част от веригата за продължителни интегриране. Тъй като дипломната работа се намира в GitHub, най-добре свързаният софтуер за продължително интегриране е именно направеният от GitHub – GithubActions.

GitHub Actions предлага множество екшъни в своя пазар (Github Actions Marketplace), измежду такива, които предлагат функционалност за използване на anaconda виртуална среда, изпълняване на код на Python и създаване автоматично на заявка за сливане (Pull Request). Тези функционалности се групират в последователност наречена работен процес (workflow).

Предоставената гъвкавост и лесна интеграция правят Github Actions лесен избора за настоящата дипломна работа.

### Интегрирана среда за разработка [19] [20]

Изборът на интегрирана среда за разработка бе сведен до два софтуера – PyCharm и Visual Studio Code. От двете среди е избрана PyCharm. Причината за този избор е по-добрата поддръжка за езика без нуждата от допълнителни модули и настройка на всеки от тях.

PyCharm поддържа Python както и виртуални среди, което прави разработката изключително лесна. Средата за разработка има и дебъгер в себе си, който е изключително функционален и лесен за работа. Освен това в PyCharm има и всички очаквани функционалности като допълване, преименуване на тип, метод и клас заедно с всички негови референции. Автоматичната създаване на модули в проекта също е изключително полезно, в себе си средата съдържа и графичен интерфейс за Git както и много други функционалности.

Друга важна функционалност е поддръжката на Jupyter тетрадки директно в интерфейса на средата. По този начин тетрадката използва същата виртуална среда като целия проект, освен това използва и същия контекст – т.е. вижда и може да работи с модулите от проекта.

Всичко, което е изброено се поддържа и от Visual Studio Code, чрез инсталиране на допълнителни плъгини, като всеки трябва да се настрои отделно. За това и PyCharm бе избран за разработката на тази дипломна работа.

## Обобщение

Описаните в тази глава технологии и инструменти ще бъдат използвани за реализацията на решението. Те са избрани, защото предоставят необходимата за изпълнението на задачата функционалност. Голяма част от използваните ресурси са безплатни или имат задоволително ограничение на безплатното си ползване. Библиотеките и избраните инструменти са избрани съобразно избраният език за разработка на дипломния проект.

Системата, която ще бъде разработена трябва да бъде използвана в Github Actions. Github actions ще бъде използван за детекция на нови функционалности, които ще бъдат анализирани и използвани за генериране на тестове. След генерацията на тези тестове те ще бъдат обработени за подобряване на тяхното състояние и добавени като част от хранилището за код.

# Описание на реализацията на разработката

Реализацията на системата представлява система, която генерира тестове базирани на променени файлове и съобщението, което е предоставено към новата версия (commit) в Git хранилище. Системата има за цел да генерира тестове с приоритети както следва:

1. Генерираният код да се компилира (интерпретира) успешно.
2. Генерираният код да използва библиотека за тестване.
3. Генерираният код да може да бъде пуснат като тест и да минава успешно.
4. Генерираният код да осъществява възможно най-голямо покритие на добавения код.

За да може да се оцени работата на системата следва да се изгради нужната инфраструктура и за това и системата се разделя условно на две части: основна част, която се грижи за генерацията на код и подсистема, която има за цел да оценява генерираният код, резултатите от тази подсистема ще бъдат показани в глава IV.

Целта на това решение е да предостави нов подход в писането и по-конкретно генерирането на интеграционни и функционални тестове. Към този момент основен фокус на проучване са били модулните тестове. За статиите, които са повлияли имплементацията на системата за дипломната е описано в глава II от книжното тяло.

Важно е да се отбележи, че по своята същност интеграционните и функционалните тестове не се различават твърде много от модулните. Практически този вид тестване е върху обекти, които в много случаи са медиатори или фасади на подсистеми. Трудното при този вид тестване е да се локализират ключовите класове. Имайки предвид сходството на модулните и функционалните тестове то вече постигнатите резултати в сферата на модулното тестване могат да бъдат пренесени и за генерация на тестове от по-високо ниво.

Имплементацията на дипломната работа практически генерира тестове от всякакво естество и тъй като класификацията на това един тест от коя категория е не тривиална задача в дипломната работа не се валидира автоматично дали генерираните тестове са строго функционални и интеграционни.

Тъй като голяма част от разработката на дипломната работа се състои в провеждането на експерименти с различни параметри и входове по време на проучването са използвани IPython и Jupyter тетрадка за бързи експерименти, които да докажат, че даден подход би бил възможен. С помощта на тези инструменти е постигнато значително намаление във времето за разработка като множество различни експерименти свързани с генерирането на код са изпълнени за значително по-малко време и по този начин избора на подходящ формат за входа на модела става много по-ефективно.

## Структура на софтуера

Както стана ясно от предходната глава структурата на софтуера е разделена условно на две части: генерация и валидация на тестове. Това разделение е и логичното разделение за софтуера тъй като за клиентите на приложението единственото важно е да получат генерираните тестове, докато по време на разработка е от изключителна важност да се знае как се справя моделът и съответно дали посоката, която е избрана е правилна.

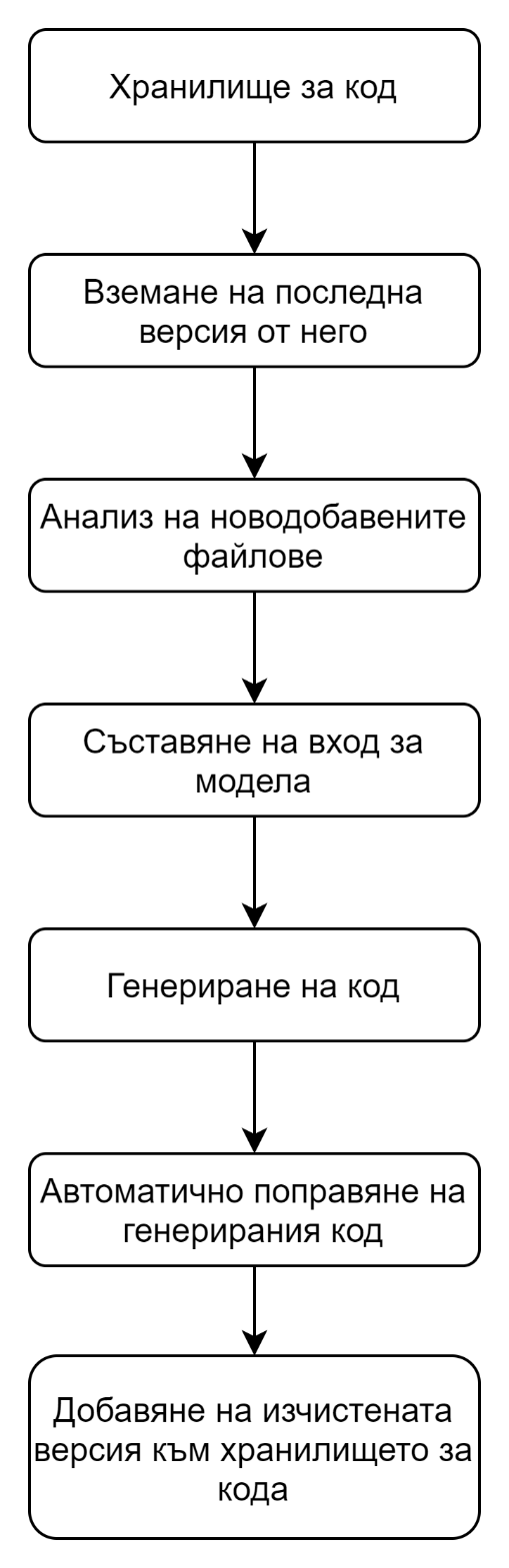
Практически важността на подсистемата за валидация на резултатите по никакъв начин не отстъпва на тази, която генерира тестовия код. Двете системи са развивани паралелно по време на разработката на софтуера.

### Подсистема за генерация на код

Ролята на тази система е да получи хранилище за код и да генерира тестове за последната качена версия в него, това се случва както е показано на Фигура 15 (тръбопроводът на генерация на код като диаграма) и Фигура 16 (имплементацията на тръбопроводът).

Тази подсистема има няколко основни задачи описани на фигурата отдолу:

1. Изземане и анализ на новодобавения код
2. Анализ на придружаващото съобщение на новодобавения код
3. Конструиране на съобщението, за вход на модела
4. Генериране на тестове
5. Автоматично поправяне на генерирания код
6. Добавяне на финалната версия на кода в хранилището за код



Фигура Тръбопровод (Pipeline) на генериране на тестове



Фигура Генерация на тестови код за даден клон от git хранилището

#### Изземане и анализ на новодобавения код и придружаващото го съобщение

За вземането на новодобавения код се използва по-горе описаната библиотека PyDriller. С нейна помощ лесно могат да се обикалят различните версии на хранилището. Тази библиотека е избрана специфично заради функционалностите, които предоставя свързани с файловете, които участват при разглеждане на конкретна версия на хранилището.

При тази стъпка се разграничават Python файлове от други променени, добавени и изтрити файлове. След това от тази извадка се оставят само файлове, в които има добавени методи и последната филтрация е разделянето на файлове, които са със сорс код и такива, които са с тестове (използват се при оценка на генерираните тестове – по-детайлно описание за това в секцията за подсистемата за оценка на генерирания код).

След като файловете са разделени се извършва задълбочен анализ върху сорс кода. Този анализ се състои в няколко стъпки:

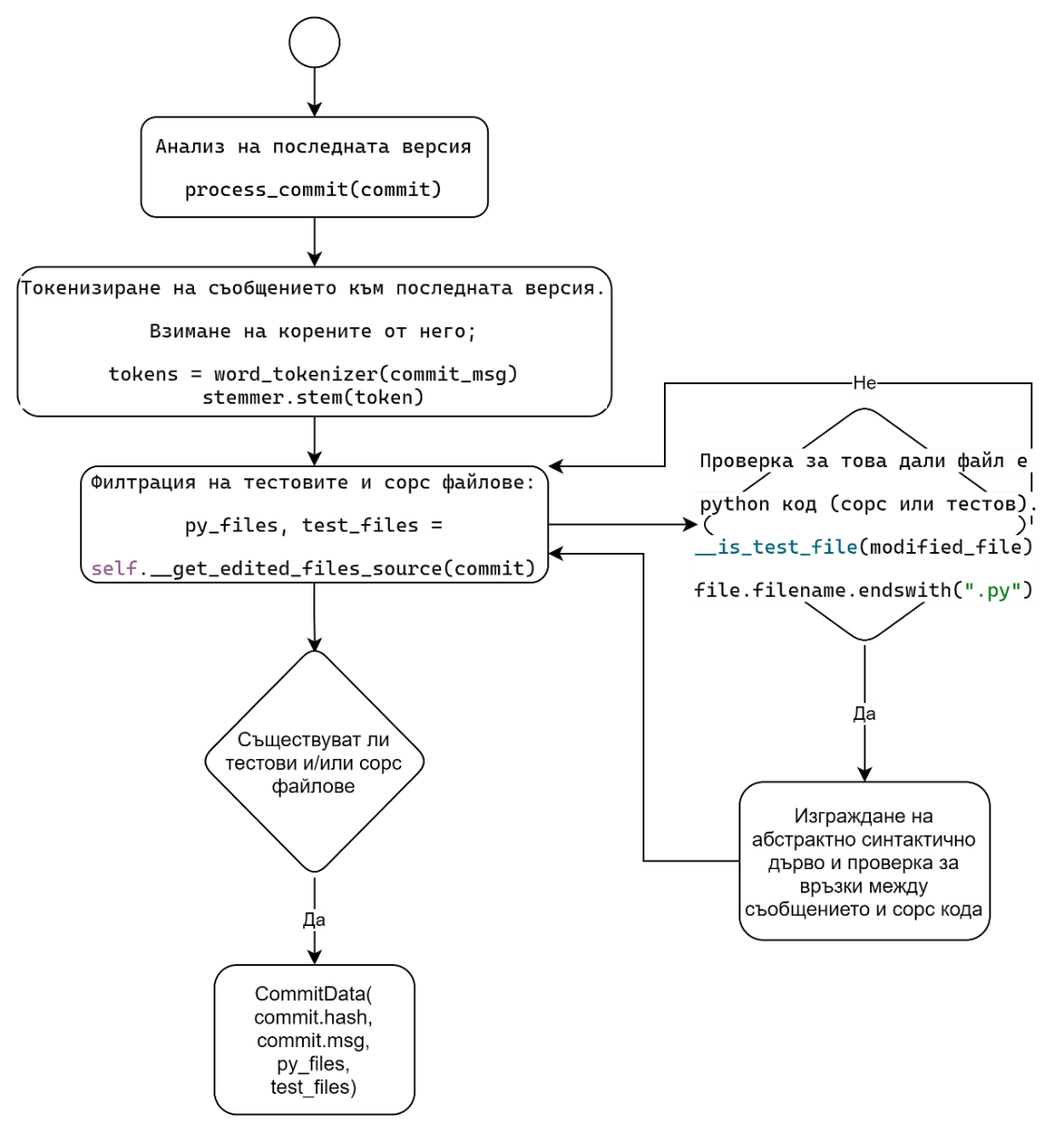
1. Съобщението, което съпътства версията (commit message) се разделя на отделни думи. Всяка от тези думи се заменя с корена си. За целта се използва библиотеката nltk и нейният агресивен коренуващ алгоритъм:

self.stemmer = SnowballStemmer(language='english')

self.stemmer.stem(w) for w in commit\_msg\_tokens

1. За всеки сорс код файл се създава абстрактно синтактично дърво. В дървото се преглеждат всички върхове, които са от тип метод и клас, ако корените от съобщението съществуват в името на метод или клас то тази връзка се записва, за да бъде използвана на следващ етап.

За тази операция основно се грижи обект от типа GitHistoryDataSetParser. Диаграма на дейностите, която разглежда обработката на една версия може да бъде видяна на Фигура 17.



Фигура Анализ на файловете от версия на git хранилище

#### Конструиране на съобщението, за вход на модела[1][5][21]

Преди да се обясни как е имплементирано конструирането на входа за модела Codex e важно да се отбележи, че Codex е огромен модел. Този модел използва над 12 милиарда параметъра и е трениран върху повече от 175 GB сорс код. След това масово трениране невронната мрежа вече е придобила така наречените мета познания. Мета познания означава, че моделът знае как да се учи да учи след всичките данни, които са минали през мрежата по време на тренирането.

До този момент програмирането в компютърните науки е имало няколко етапа. Всичко започва от преди появата на компютри за общи цели т.е. всяка задача е изисквала специализиран хардуер. След това се появяват компютри за общи цели, а с тях и оригиналното програмиране наречено „софтуер 1.0“. При него програмистът конструира и описва алгоритъма на работа посредством код. Следващата стъпка е навлизането на машинното обучение, при което алгоритъмът практически се замества от черна кутия (алгоритъм използван за машинно обучение), а работата на програмиста се състои в подбора на правилните данни, с които да го обучи. Този вид програмиране е така наречения „софтуер 2.0“. С появата на тези нови модели от ранга на GPT-3 и Codex се задава следващият етап от програмирането – конструкция на вход за модела. Някой наричат този етап „софтуер 3.0“.

При модели от ранга на Codex вече няма нужда от допълнително трениране на мрежата – за конкретна задача. Работата с тези модели много наподобява дискусия с умно подрастващо дете, за което е от изключителна важност как ще му бъде зададен въпрос и с колко и какъв контекст, който да му помогне да отговори. Затова и множествената итерация от различни входни данни е важна, за да се прецени кои входове до какви изходи водят. [21]

От казаното дотук следва, че е изключително важно как се конструира входът. Както е отбелязано задачата трябва да бъде зададена „правилно“ и с нужния контекст.

От съществуващи статии по темата за генериране на модулни тестове е стигнат до извода, че контекста, който се дава при трениране на трансформаторите е изключително важен. Типично колкото по-голям толкова по-добре.[1] Затова и подобен подход е взет при конструирането на входни данни за Codex.

Подходът при конструкцията на входни данни при Codex e процес от две стъпки. Първата стъпка има за цел да избере кой фрагмент от код да бъде включен за контекст на генерирането на код за тестове, а втората се концентрира върху задаването на задачата за модела.

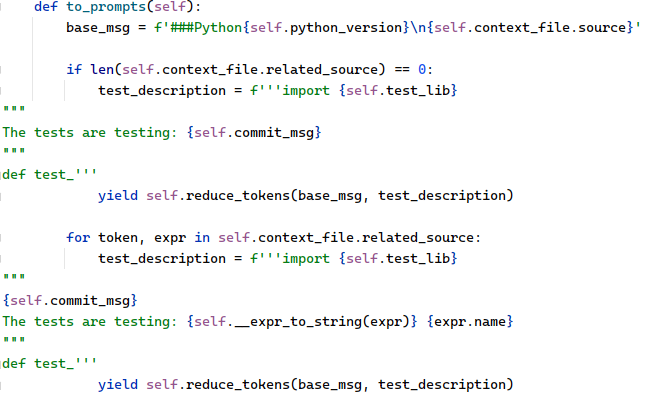
Изборът на фрагмент от код се състои от сортиране на филтрираните сорс файлове за дадената версия. Сортирането става на база два фактора – брой класове и функции от същия проект, които се използват във файла и броя референции от имената на класовете и методите във файла към съобщението за версията (сборът на двете определя рангът на сорс файла). Тези критерии са избрани тъй като типично (не задължително) файловете, които използват в себе си много други модули от системата са медиатори и/или фасади, които дефинират връзките между под-елементите на системата т.е. дефинират някаква функционалност. Отново целта на системата е да генерира функционални и интеграционни тестове затова и локализирането на такива файлове е важно. Вторият критерии е свързан със съобщението, което е написано за новодобавения код, тук презумпцията е, че ново въведената функционалност е описана и следователно компонентите, които са свързани с нея използват близка или сходна реч.

След като сорс файловете се сортират, се подбират трите с най-висок резултат. Именно те ще бъдат използвани за създаването на входни данни, с които да бъде извикан моделът.

Втората стъпка е конструкцията на заданието към модела. Важно е модела да знае, коя библиотека за тестване ще се използва. За целите на имплементацията е избрана pytest поради, голямата ѝ популярност сред най-големите публични проекти в python екосистемата. На модела бива казано, че трябва да бъдат създадени тестове като контекст за техните сценарии се дава съобщението на версията (commit message). Много пъти за жалост това не е достатъчно конкретно и много случаи се пропускат, за това се създават и допълнителни входни данни, които дават конкретни класове и методи, които трябва да бъдат тествани. Класовете и методите, чиито имена се използват се взимат от референциите на сорс файла (ако съществуват такива) към думи от съобщението.

След като се избере и задание за задачата се добавя def test\_, което да подскаже на модела, че трябва да се довърши функция, която е тест.

След като се комбинират заданието на задачата и сорс кода се създава цялостното съобщение, което може да бъде видяно на Фигура 18.



Фигура конструиране на входни данни за модела на допълване (Codex)

Важно е да се отбележи, че моделът има ограничение на думите (tokens), които могат да бъдат изпратени. Затова и ако е много голям сорс файлът то той се съкращава, докато не се стигне до размер, който удовлетворява изискваният (8000 думи).

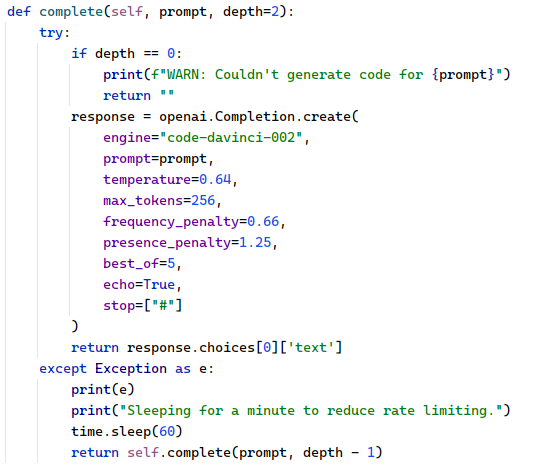
#### Генериране на тестове[22]

След като вече има входни данни остава моделът да даде своето предположение. Интерфейсът, с който се използва моделът е на Python, а практически той от долу изпраща заявка до сървърите на OpenAI. Тъй като моделът в момента е в бета версия, използването му е безплатно, но с ограничение от 60 заявки на минута и се налагат изчаквания при пренасищане със заявки.

Интерфейсът за работа с модела има множество параметри:

* Engine - идентификатор на модела, който ще бъде използван
* Prompt – входни данни, които моделът ще допълни
* Temperature – високите стойности означават, че моделът ще бъде „по-креативен“. Тъй като моделът практически е обучен да предположи коя е следващата дума в последователността с някакъв процент на вероятност. Проблемът с това винаги да се избира думата с най-голяма вероятност е, че възможността от безкрайно повторение е не малка. Тук идва и параметърът температура, който води началото си от статистическата термодинамика – колкото по-висока е температурата толкова по-голям е шанса да се появят състояния с ниски количества на енергия. Което в контекста на езиков модел означава, че колкото по-висока е тази стойност толкова по-голям е шанса да се появи дума, която не е сред най-логичните следващи думи („по-креативна“ дума). [24]
* max\_tokens – максималният брой думи, които ще бъдат допълнени от модела
* frequency\_penalty – стойност между -2.0 и 2.0, като положителните стойности „наказват“ думи, които вече съществуват в текста дотук.
* presence\_penalty – подобно на frequency\_penalty с разликата, че вдига вероятността моделът да започне нова „тема“
* best\_of – колко предположения да създаде, от които да избере най-доброто или n-те най-добри, ако моделът е настроен да връща повече от един резултат.
* echo – дали да връща входните данни заедно с предположението или не
* stop – символи или низове при чиято поява да спре генерацията на предположения.

Стойностите на горе изписаните аргументи могат да се видят на фигура 19. Важно е да се отбележи, че стойностите на frequency\_penalty и presense\_penalty са ключови, защото при тези по подразбиране от 0 се генерират множество тестове, които се повтарят.

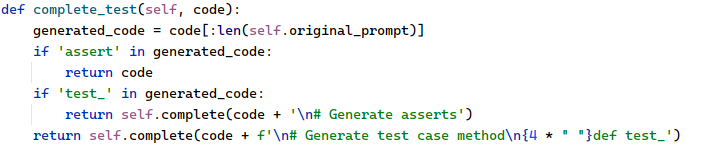


Фигура Дефиниция и извикване на Codex от python

Тъй като има ограничения при използването на модела както е отбелязано по-горе при настъпване на изключение се изчаква една минута, след която минута отново се извиква моделът. Това може да се повтори не повече от 2 пъти.

Тъй като моделът е настроен да връща само един резултат, то само първият резултат от отговора се използва за по-нататъшна обработка.

Ако в резултата отсъстват клаузи за проверка, то моделът се вика повторно с вече получените данни и добавен коментар, който казва на модела да добави клаузи от тип assert. (Фигура 20)



Фигура проверка дали има клаузи за проверка в резултата от мдоела и ако няма повторно извикване с искане за добавянето им.

#### Автоматично поправяне на генерирания код

Генерираните редове код от Codex типично не биха били изпълними заради множество и различни грешки.

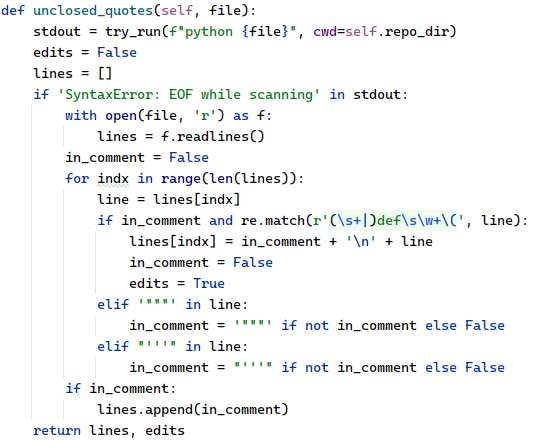
Codex се справя много добре да преведе задачата дадена му с думи към код, но типично този код не се компилира. Това се дължи на факта, че Codex както и други такива модели са базирани на езикови трансформатори, които се справят добре с решаването на различни задачи, но нямат добро разбиране за задачата както и за семантиката на програмата, която пишат. Този проблем поражда и нуждата от автоматична поправка на генерирания код. Тази поправка типично е по-лесна за строго типизираните езици като Java, това се вижда и от вече съществуващите инструменти за такива езици. Използването на такъв инструмент значително подобрява работата на генераторите на код. След използването на два такива инструмента за Java върху генерирани от Codex решения на алгоритмични задачи подобренията са видими – от 37 до 42 решени задачи с лесна сложност и от 5 на 9 решени задачи от средна сложност.[7]

За Python няма утвърдени системи/библиотеки, които да могат да оправят проблеми в кода, за това като част от дипломната работа е създадена фаза, която се състои от три стъпки:

1. Опит за създаване на абстрактно синтактично дърво и поправка на синтактични грешки
2. Проверка за не-затворени кавички и коментари за много редове и затваряне на такива ако съществуват
3. Викане на Codex за поправка на грешки при интерпретация
4. Добавяне на изпуснати import клаузи за код рефериран от същият проект

Първата част е най-проста. Създаването на абстрактно синтактично дърво е много свободно по отношение на кода, не проверява дали може да се изпълни практически го възприема като низ. Въпреки това ако се появят синтактични грешки, често пъти са много трудни за оправяне автоматично и за това такива редове директно се премахват.

Втората част от проверката е за това дали няма грешно затворени коментари и кавички. Тази част проверява дали няма дефиниции на функции в многоредов коментар и дали всички коментари са затворени. Често се случва многоредов коментар да бъде започнат, но поради достигане на лимита на генерирани думи да не се затвори, което в последствие води до синтактична грешка. Затова е важно всички такива коментари да са затворени, както и едноредовите стрингове да са затворени. Функцията, която оправя такива проблеми може да бъде видяна на Фигура 21.



Фигура Функция, която проверява и оправя коментари, които не са затворени.

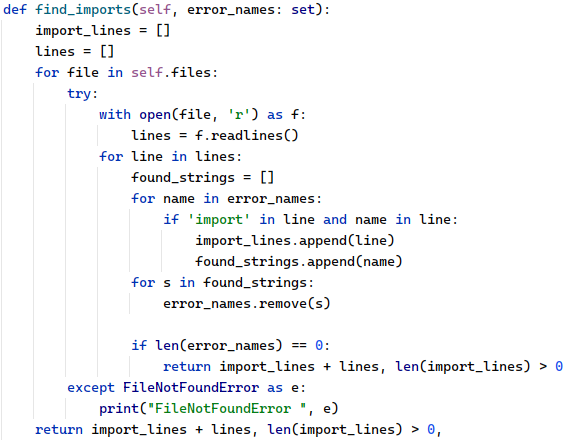
След като всички кавички са затворени, отново се извиква Codex моделът, но този път в режим на редакция (Edit), като за този режим му се оказва, че трябва да направи кода интерпретируем/компилируем.

През март 2022 OpenAI пускат нова версия на Codex, която може да променя код, а не само да го допълва. Често срещани проблеми при генерирания код са недопълнен последен ред, неправилен брой скоби, извикване на методи и използване на класове, които не са добавени във файла както и различни алгоритмични проблеми. За първите два проблема Codex Edit се справя доста добре.[7]

Точно за тях е използван и в дипломната работа. Кавичките са изнесени като отделна стъпка, защото след множество тестове бе установено, че моделът е неспособен да прецени сам къде и кои кавички трябва да затвори.

Както вече бе описано реферирането на код, който не е от този файл е често срещан проблем за това и последната стъпка е оправянето на такива проблеми. За целта е разработен алгоритъм, който изпълнява оправения до тук код и при наличие на NameError търси във всички файлове на проекта за import клауза, която е за проблемния метод или клас. Типично този подход работи, тъй като всичко, което се тества е част от проекта или се използва в проекта. Алгоритъмът за намиране може да бъде видян на Фигура 22.

За разпознаването на проблематичните методи или класове от съобщението за грешка се използва следният регулярен израз: re.compile(r"NameError:\sname '(\w+)' is not defined")

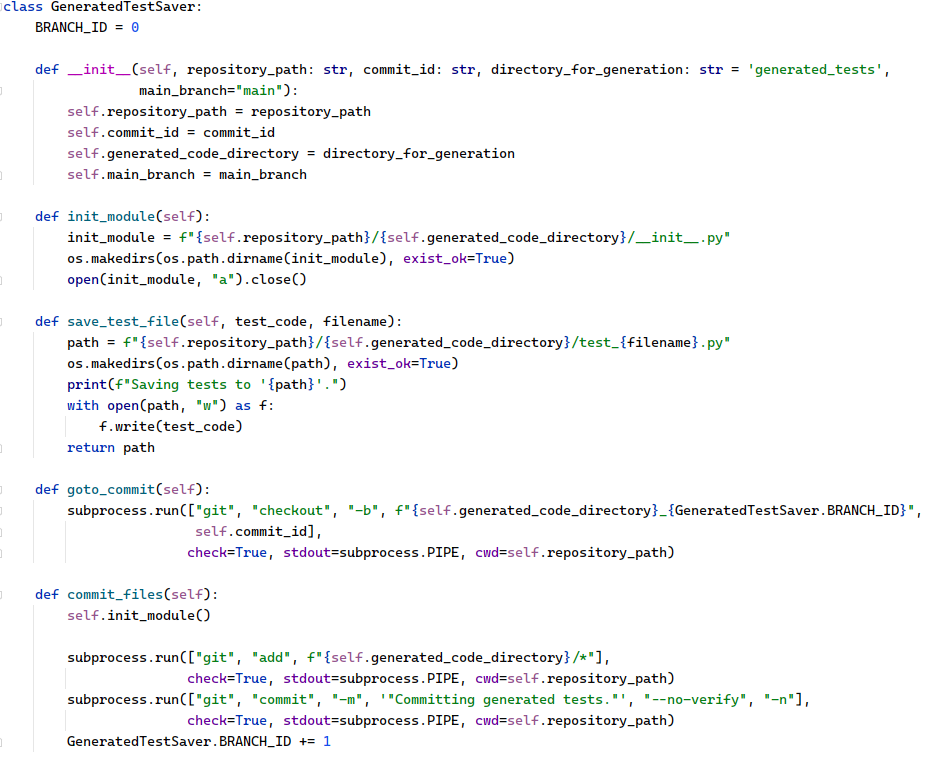


Фигура Функция, която намира несъществуващи референции и връща import клаузи, с които да бъдат добавени.

#### Добавяне на коригираната версия в хранилището за код

След като генерираният код е минал през нужната обработка, за да се приведе в компилируем вид, следва да се запази. За целта се създава нов клон от основния за хранилището. В папка с името на новия клон се създават файлове с произволни имена, които започва с test\_ последвани от произволно генериран низ и завършват на .py. След като всички файлове са готови се създава нова версия в новия клон.

Този процес се случва посредством извикване на функции в командния ред както следва:



Фигура GeneratedTestSaver – клас, който се грижи за запазването на вече генерираните тестове

След запазването на версията винаги хранилището се връща към основния си клон.

### Подсистема за валидация на резултатите

След като подсистемата за генериране на тестове започна да продуцира първите тестове се появява нуждата от това да се измерва как се справя. Така се породи и нуждата от инфраструктура, с която да може да се оценява колко добре се справя системата за генериране на код.

Оценяването на генериран текст и конкретно на генериран код е сложна задача. Комплексността на оценката идва от това, че при генерацията на свободен текст и код няма еднозначно решение, което да се нарече правилно.

Най-наивният начин разбира се е директно сравнение с референтно решение. Неефективността на тази метрика се проявява първо при превеждането на текст. Типично има повече от един начин даден текст да се преведе от един на друг език. От там води началото си и BLEU оценката. За тази оценка практически се сравнява генерираният текст с референтен такъв, който се смята за верен. За жалост тази оценка не е добър показател при генерирания код, тъй като две напълно верни решения могат да са много различни – например рекурсивно и итеративно решение на един и същ проблем. [23]

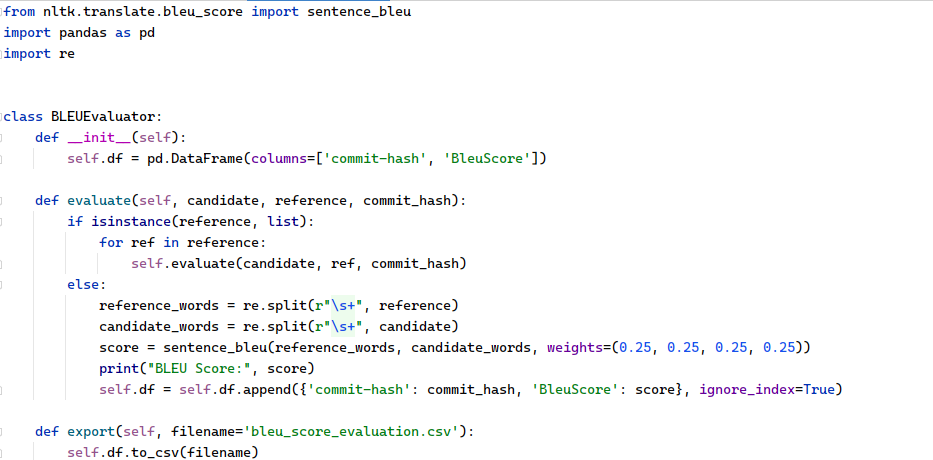
За това се появява и модификация на BLEU наречена CodeBLEU, но тя също не прави добра обективна оценка на генерирания код, тъй като отново разчита твърде много на текста, който е изписан.

Като програмисти основната метрика при оценката на една програма или подпрограма е това дали тя върши работата, която се очаква от нея коректно. За това и за добра програма се смята тази, за която минават тестове (автоматични или ръчни), които валидират, че тя се държи коректно за различни входове.

За това се появява и метриката pass@k. Тази метрика е обективно по-добра от други, като BLEU, CodeBLEU, сравнение на синтактични дървета и други. За това и тази метрика се използва широко в сферата на генериране на код. Повече за нея е описано в първа глава от този труд.[6]

Подходът pass@k работи много добре при генериране на код, който решава някакъв проблем. Тук идва и разликата с разработката в тази дипломна работа – целта тук е да се генерират тестове. Следвайки логиката от по-рано най-добрия начин за оценка на генериран код е обективната оценка, която използват програмистите. При код това е дали кодът работи правилно (дали минава успешно различни тестове сценарии), докато при тестовете това е тяхното покритие.

Както вече бе спомената BLEU не е добра метрика, но за сметка на това е много лесна за пресмятане, затова е и първата метрика, която е добавена в проекта. Използването става като се сравнява генерираният тест с всички тестове от оригиналната версия, затова и по-рано при анализа на версия се записват и тестовете ако има такива. Тази логика се споделя между модулите.



Фигура Оценка на генериран код от тип BLEU

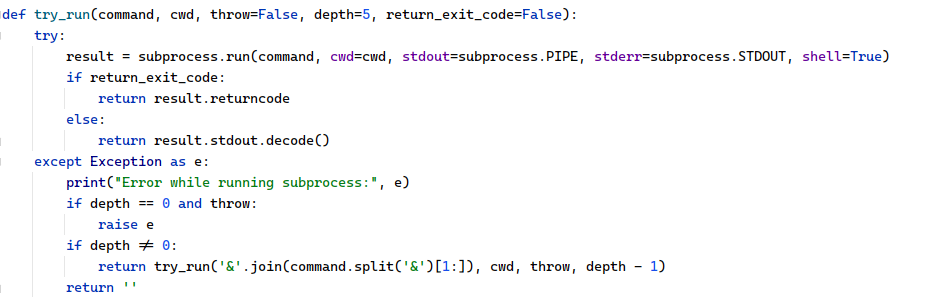
Покритието е най-добрата метрика в случая за това е и крайната цел на тази подсистема. За да се достигне обаче трябва да бъдат изпълнени няколко изисквания преди това.

Както бе разгледано в другата подсистема, преди теста да стигне до състояние, в което може да бъде пуснат трябва първо да се компилира, а след това и успешно да се интерпретира. Тези две стъпки са изключително важни, а и не са тривиални. За това са отделни метрики.

Замерването на метрики за компилация/интерпретация и пускане на тест дори при такъв, който е изряден не е тривиална задача. Различните проекти имат различни зависимости и нужда от различни среди. От тук вече следва и че първата стъпка е да се създаде правилната среда, в която да се тестват генерираните тестове. В проекта това е постигнато като се създава виртуална среда чрез pip, според специфичния проект се задават правилните команди, които да се изпълнят в директорията на тествания проект.

Тъй като, когато се тества цяло хранилище се започва от самото начало на историята му създаването на среда за всяка версия не съвпада точно се правят множество повторения на командите така, че да се изключат невалидните части от командата докато остане само валидна чат (Фигура 25).

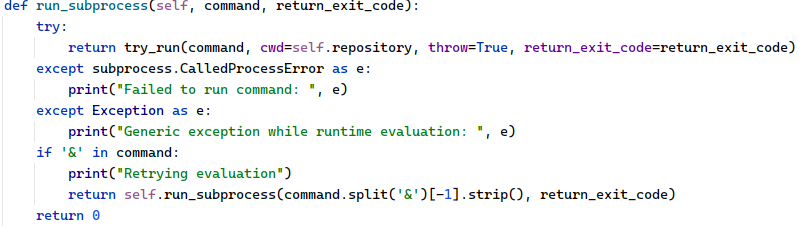
След като средата е активирана следва тестова проба за това дали кодът се компилира. Ако кодът се компилира успешно то процесът ще върне като резултат 0, в противен случай според грешката ще е стойност различна от 0 (Фигура 26).



Фигура изпълнение на команда в командния ред с нужните обработки при настъпване на грешка.

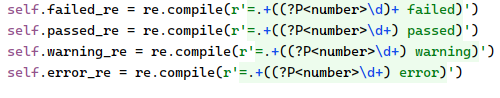


Фигура Оценка на компилацията на файл

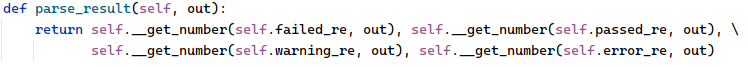


Фигура

След компилацията следва една от най-важните метрики - дали генерираните тестове се изпълняват – ако да, дали е успешно. Това оценяване се изпълнява като се използва вече създадената среда и пускането на тестовете с командата pytest. Резултата се вади от низ, който е събран от стандартния изход. Има четири състояния, които се събират за всеки тест: минава, не минава, предупреждения, грешка. Събират се чрез регулярни изрази (фигура 28).

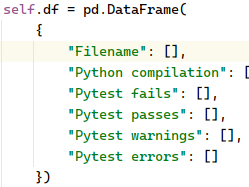


Фигура регулярни изрази за събиране на данни от стандартния изход на изпълнението на тестовете



Фигура обработка на стандартния изход ит изпълненето на тестове

Всички данни, които се събират се събират в рамка за данни (DataFrame), която има вида от Фигура 29. Използва се тази библиотека, за да може в бъдеще тези данни да се манипулират още в програмата, към момента се използва и за експорт към .csv формат за по-нататъшна обработка. (Фигура 30)

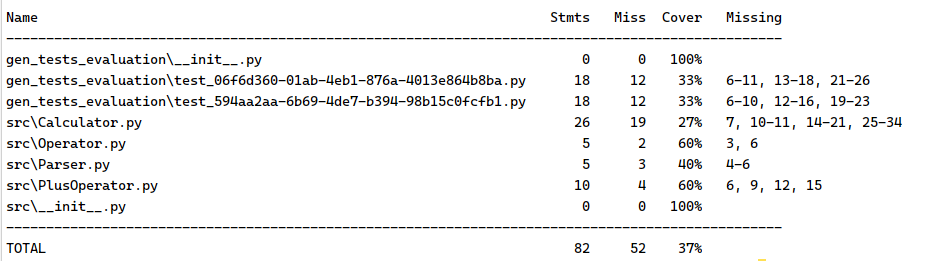


Фигура Формат на рамката за данни

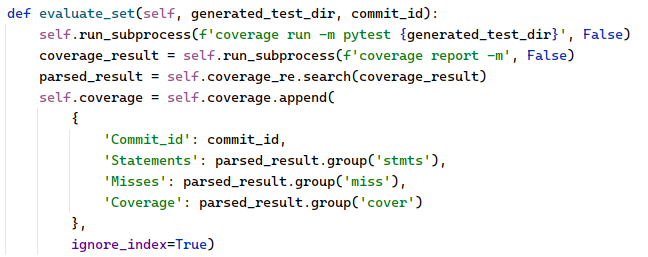
След постигане на задоволителни резултати на метриката за оценка на пускането на тестовете се въвежда и по-рано споменатата метрика – покритие на кода. Тази метрика е най-големият показател за дипломната работа. Тя е имплементирана с помощта на библиотеката coverage, която мери покритието. Изходът ѝ представлява таблица като тази на Фигура 31. За да се обработи този изход се използва следния регулярен израз:

self.coverage\_re = re.compile(r"TOTAL\s+(?P<stmts>\d+)\s+(?P<miss>\d+)\s+(?P<cover>\d+)")

Тази оценка се извиква след генерирането на всички тестове за конкретната версия. (Фигура 32)

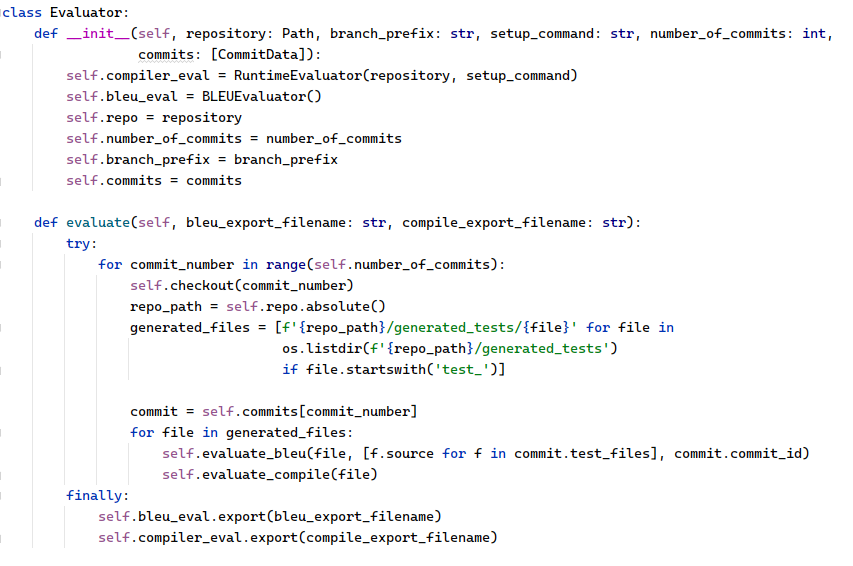


Фигура Изход от пускане на тест с покритие



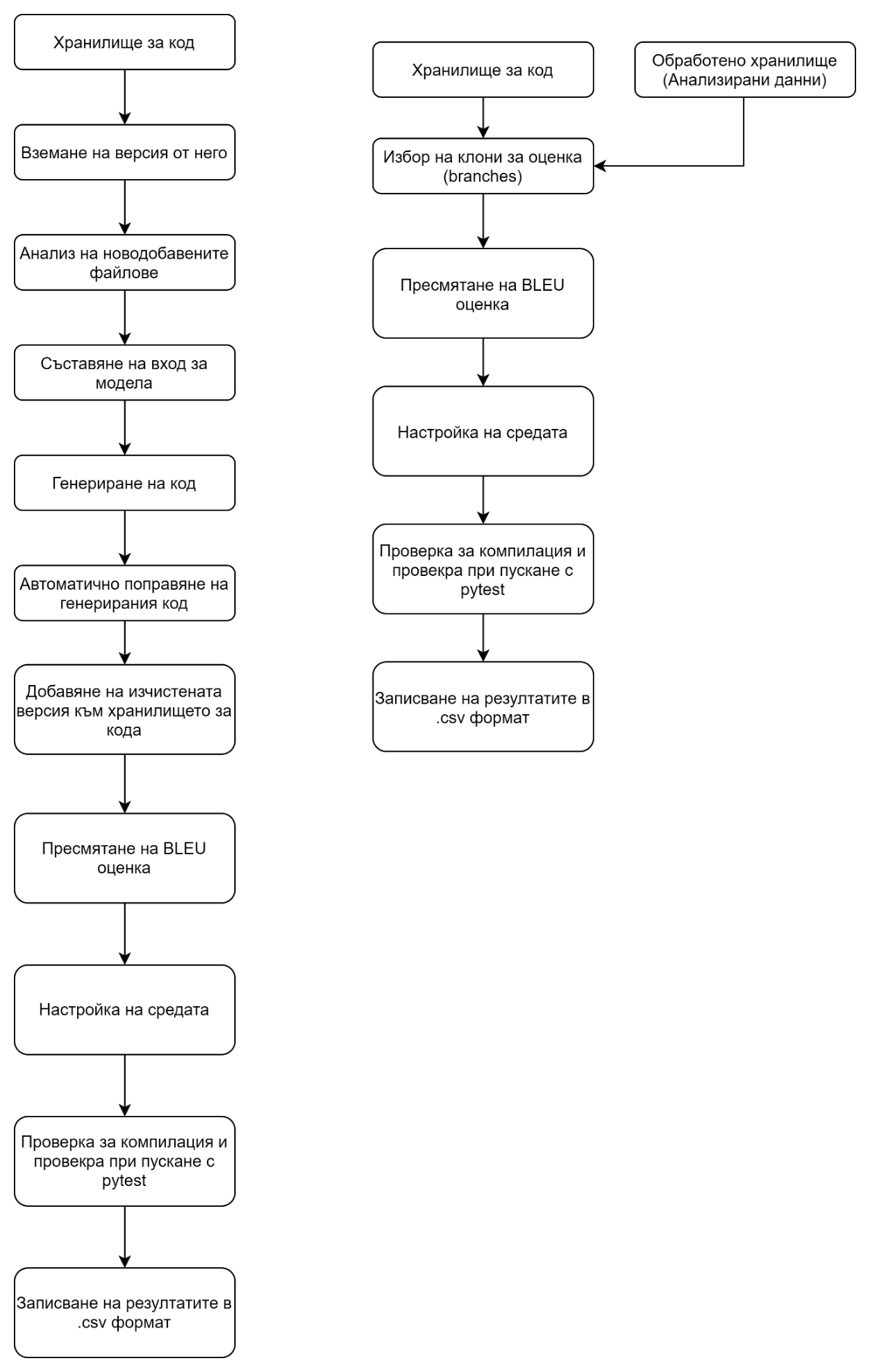
Фигура Метод за събиране на данни свързани с покритието на генерираните тестове

Тази подсистема има две тръбопровода (pipeline) за използване. И двата могат да бъдат видени на фигура 34. При десния тръбопровод се използва класът Evaluator, на който се подават данни за анализ и какъв е префиксът на клоните, в които е генерираният код. Методът evaluate е този, който дирижира цялата оценка (Фигура 33).



Фигура

Вторият тръбопровод е включване на системата след генерирането на код от тръбопровода на другата подсистема.



Фигура тръпопроводи за оценка на системата. Ляво – оценка комбинирана с генерирането на код, Дясно – самостоятелна оценка.

## Обобщение и бъдещо развитие

Изградената система е добро демонстрационно решение на предложения подход. Практически системата работи в целия си вид – използвайки хранилище за код при появата на нова версия тя бива анализирана и се генерират тестове използвайки този анализ. Част от генерираните тестове са успешно интерпретируеми и валидират новата функционалност.

Бъдещото развитие на продукта е основно в две направления:

Първото, от които е добавянето на съществуващия код за анализ, в момента се анализира само новодобавения. Част от този анализ трябва да станат и съществуващите тестове към момента, с пример от тях моделът може да започне да дава по-добри генерации

Второто направление е поправката на генерираният код, към момента поправката на генерираният код не е много прецизна. Съобщенията, които се изпращат към модела, могат да бъдат обогатени спрямо съобщенията, които излизат при интерпретация на генерирания код. Друг аспект е изолирането на проблемния фрагмент от код – това би забързало значително моделът в неговия отговор, както и прецизността му.

Системата е така изградена, че да позволява лесното разширение и по двете направление. Всяко едно от тях е под-система в реализацията на продукта и използва независими компоненти за изпълнението на стъпките в съответно анализа и поправката на кода.

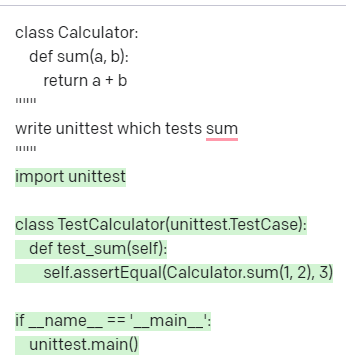
# Използвани данни и проведени експерименти

В хода на развитието на софтуерния продукт са правени множество тестове от практиката, които целят да оценят работата на системата към конкретния момент. Получените резултати и направеният анализ върху тях са използвани за задаване на посока на развитието на проекта.

Всички тестове са направени на компютър с операционна система windows 10, и виртуална среда Anaconda.

## Експеримент 1

Като всяка идея така и тази започна с малки експерименти, които да потвърдят дали въобще е възможно да се реализира. Тези първоначални тестове са направени в онлайн средата за работа с OpenAI модели: <https://beta.openai.com/playground?mode=complete>

Първият проведен експеримент е с елементарен клас Calculator (Фигура 35)

Фигура 35 Генериране на тестове за клас Calculator

Предивид разгледаните недостатъци на генерирането на код от модели за работа с език в първа галва следващият експеримент бе да се изгенерира същия тест репрезентирайки кода с неговото абстрактно синтактично дърво (Фигура 36). Това, което се забелязва е, че генерираният тест е много по-изчерпателен, но за съжаления низовата репрезентация на синтактичното дърво заема много повече думи (tokens).



Фигура Генериране на тестове използвайки Абстрактно синтаткично дърво

### Изводи:

Въпреки, че чрез синтактичното дърво се получават по-изчерпателни проверки в теста използват значително повече думи. Както вече бе споменато моделът има ограничения на колко думи, могат да се подават и генерират. Лимитът се постига при първия файл, който е с над 200 реда код, ако се изпозлва синтактично дърво за репрезентацията на кода. Затова и този вариант отпада, тъй както вече е споменато повече контекст би следвало до води до по-добри резултати

## Експеримент 2:

Този експеримент е проведен след базовата имплементация на събирането на данни – без съобщението към версия да има отношение към данните също така към този момент няма и избор на кои файлове са най-ценни.

Преди този експеримент има много опити, които завършват с 0% компилируемост и 0% пускане на тестове, тъй като проекта изисква собствена среда за работа, която не е осигурена.

За целите на експеримента вече се подбират и по-сериозни данни. За тестови проект, за който да се генерират тестове е избрана една от най-големите библиотеки за интернет приложения – flask.

След базов анализ от 4,719 версии (commits) остават 282, които имат промени в сорс файлове и тестове. От тези 282 успяват да се генерират тестове само за 71, за тези 71 промени са генерирани 96 файла.

Резултата на BLEU е приблизително 0.

|  |  |
| --- | --- |
| Всички версии: | 4 719 |
| Версии, които се смятат за валидни: | 282 |
| Версии, за които се генерират тестове: | 71 |
| Брой генерирани файлове: | 96 |
| Компилируеми генерирани файлове [%]: | 32,64 % |
| Файлове, които могат да бъдат пуснати като тестове [%]: | 2.1% |

### Тестови извадки:

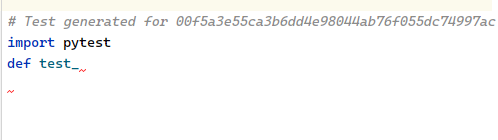
Произволни извадки, които показват генерирания код:



Фигура генериран тест за проекта flask



Фигура генериран тест за проекта flask



Фигура генериран тест за flask, който е провален опит.

### Изводи:

1. Процента на генерирани тестове спрямо възможности е нисък – около 25%. След инспекция се оказва, че много от съобщенията, които се подават нямат общо с кода, който участва в конкретната версия: Fixed code, Fixed bugs и т.н.
2. Средата за работа е изключително важна. Без нея тестовете не могат да се пуснат, командата python test\_file.py, не проверява нищо тъй като python се интерпретира и никой от тестовете не се извиква следователно много грешки се пропускат.

## Експеримент 3

Надграждайки от предходните експерименти се добавя допълнителна филтрация на данните, които ще се оценяват. За по-правилна преценка дали файл е релевантен то трябва да съществуват добавени методи за конкретната версия.

За този експеримент се добавят два нови големи проекта. Най-големият проект с отворен код, който е писан на Python за работа с http протокола httpie, който има над 1700 версии в GitHub - <https://github.com/httpie/httpie>

Освен този проект е избран и още един – PyDriller - <https://github.com/ishepard/pydriller> Този проект се използва в разработката на дипломната работа и има над 700 версии в GitHub.

В тази версия на дипломната работа се използват и средите на съответните проекти.

Резултати за httpie:

|  |  |
| --- | --- |
| Всички версии: | 1725 |
| Версии, които се смятат за валидни: | 152 |
| Версии, за които се генерират тестове: | 103 |
| Брой генерирани файлове: | 161 |
| Компилируеми генерирани файлове [%]: | 5.2% |
| Файлове, които могат да бъдат пуснати като тестове [%]: | 2.5% |

Резултати за PyDriller:

|  |  |
| --- | --- |
| Всички версии: | 4 719 |
| Версии, които се смятат за валидни: | 81 |
| Версии, за които се генерират тестове: | 62 |
| Брой генерирани файлове: | 92 |
| Компилируеми генерирани файлове [%]: | 87.09 % |
| Файлове, които могат да бъдат пуснати като тестове [%]: | 86.95 % |

Генерацията на тестовете отнема не малко време: Pydriller ~ 33 минути; httpie ~ 94 минути.

### Тестови извадки:

Произволни извадки, които показват генерирания код:



Фигура Генерирани тестове за PyDriller



Фигура Генериран тест за PyDriller

### Изводи:

1. Pydriller има много по-висок успех за това, колко теста могат да бъдат пуснати от httpie. Това се дължи на по-простата му структура. Важно е да се отбележи, че това е възможност да се пуснат. Отново минават 0% от тестовете, забелязват се изключително много грешки при изпълнението на кода. Например реферират се типове и функции без да са добавени с import клаузи в кода.
2. Времето за оценка на модела нараства многократно.
3. Не може работата за един проект да се паралеризира, тъй като хранилището е едно и то не може да бъде в повече от един клон, а дори да се направят множество хранилища има ограничения на броя заявки към моделът за минута, което ограничение се достига и без паралелна обработка.

## Експеримент 4

Основният проблем засегнат при експеримент 4 е отсъствието на import-и. При този експеримент се имплементира добавянето на нужните референции към методи и класове, които се използват в тестовите методи. Освен това при този експеримент се добавят вече и сортиране на сорс код файловете по описаната по-горе процедура.

Също от миналите експерименти се забелязва, че има тестове, които нямат нито една проверка. За да се реши този проблем се използва вече генерираният код за нови входни данни, като се добавя условие за това да се добавят оператори за проверка.

Отново обект на експеримента е проекта httpie.

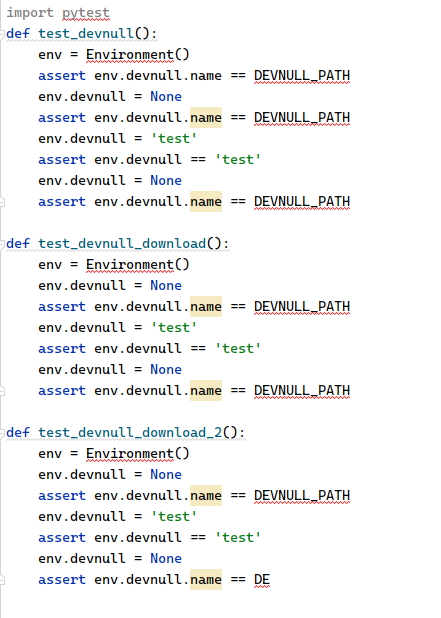
Резултати за httpie:

|  |  |
| --- | --- |
| Всички версии: | 1725 |
| Версии, които се смятат за валидни: | 152 |
| Версии, за които се генерират тестове: | 103 |
| Брой генерирани файлове: | 161 |
| Компилируеми генерирани файлове [%]: | 100 % |
| Файлове, които могат да бъдат пуснати като тестове [%]: | 2.5% |

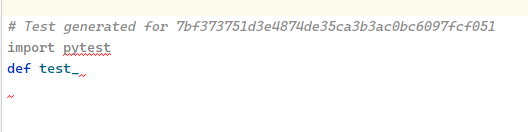
Повторните извиквания на модела забавят още повече генерацията на код и тя вече отнема приблизително 4 часа за този проект. Това е и причината да не са пускани другите проекти.

### Тестови извадки:

Произволни извадки, които показват генерирания код:



Фигура Генериран тест за httpie



Фигура Генериран тест за httpie



Фигура Генериран тест за httpie

### Изводи:

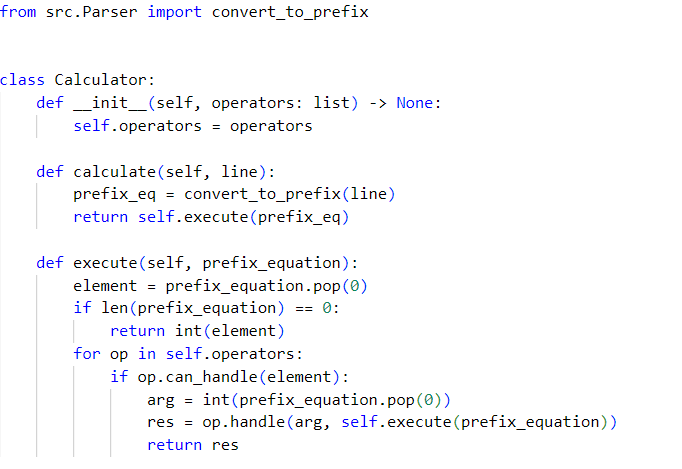
1. Добавянето на import-и и извикването на допълнителни пъти Codex прави тестовете по-завършени и съответно командата python test\_file минава за всички файлове.
2. Допълнителното извикване на модела, забавя генерацията много и времето нараства твърде много, което пречи на итерацията на различни параметри и подходи.
3. Процентът на тестове, които могат да се изпълняват продължава да е нисък поради грешки от вида на не затворени двойки кавички и скоби.

## Експеримент 5

След експеримент 4 се вижда, че експериментирането е станало много бавен процес. Това налага и появата на нов проект, който да бъде тестван. Този проект е написан специално, за да бъде използван като опитна постановка, на която да се тества основният проект.

Проектът представлява калкулатор, който има объркани приоритети на операциите, за да не може Codex да използва математическото си разбиране придобито от множеството алгоритмични задачи, които е виждал.

Основният клас на проекта има следния вид:



Фигура Основен клас за под-проекта Calculator

Този клас е предвиден да бъде използван при всички функционални тестове.

Проектът има 6 версии, които са добавяне на функционалност за събиране, умножение, изваждане, промяна във формата на уравненията, деление и степенуване.

Този проект има добри описания на всяка от версиите си. Нека се разгледа за пример добавянето на умножение:

Added support for MultiplyOperations. Calculator should be able to handle equations of: 2\*4=8 and support both multiplication and summation like: 2+10\*2=22

Това е доста ясно описание, което съдържа в себе си и критерия за приемане.

За този тест се добавят вече още добавки към автоматичното поправяне на генерирания код, а именно затваряне на незатворени кавички. С това вече поправките по генерирания код са две – затварят се всички кавички и се добавят всички реферирани методи и класове.

След доста експериментиране с извадки от кода очакванията са обнадеждаващи и се добавят проверки за това дали конкретни тестове минават или не, колко грешки и забележки има от библиотеката pytest.

Този път за оценка е пуснат новият малък проект за калкулатор и отново httpie.

Проект калкулатор резултати:

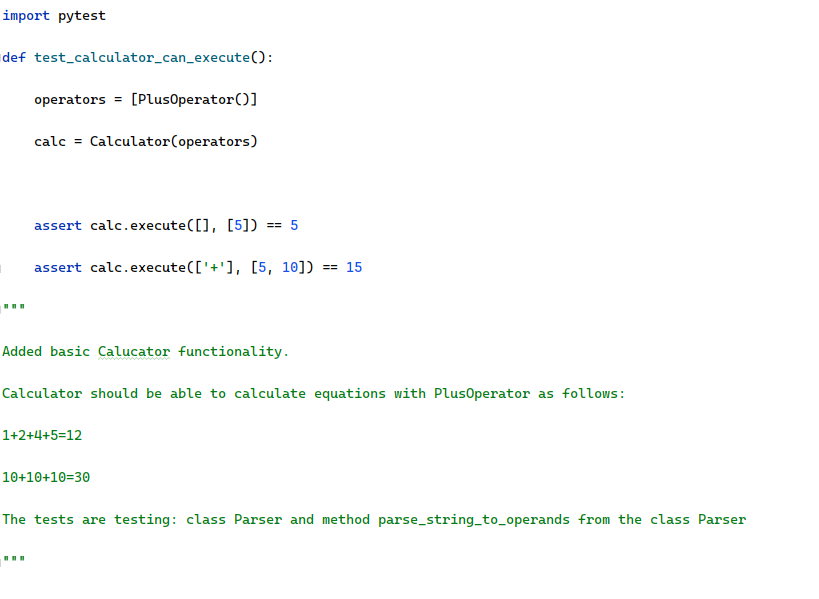
|  |  |
| --- | --- |
| Всички версии: | 6 |
| Версии, които се смятат за валидни: | 6 |
| Версии, за които се генерират тестове: | 6 |
| Брой генерирани файлове: | 16 |
| Изцяло компилируеми файлове [%]: | 0 % |
| Брой тестове, които не минават заради грешка в проверка: | 1 |
| Брой тестове, които минават успешно: | 0 |
| Брой предупреждения от pytest | 16 |
| Брой компилационни грешки в конкретни тестове | 15 |

Проект httpie, чиито резултати отнеха 7 часа:

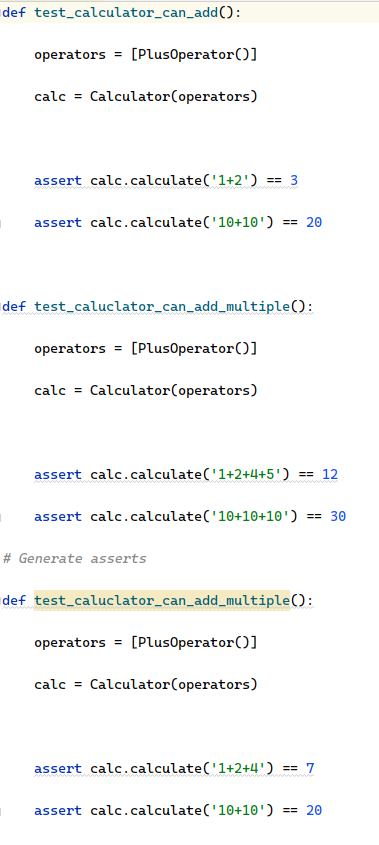
|  |  |
| --- | --- |
| Всички версии: | 1725 |
| Версии, които се смятат за валидни: | 180 |
| Версии, за които се генерират тестове: | 166 |
| Брой генерирани файлове: | 748 |
| Изцяло компилируеми файлове [%]: | N/A [бъг] |
| Брой тестове, които не минават заради грешка в проверка: | 118 |
| Брой тестове, които минават успешно: | 9 |
| Брой предупреждения от pytest | 16 |
| Брой компилационни грешки в конкретни тестове | 663 |

### Тестови извадки:

Произволни извадки, които показват генерирания код:



Фигура Генериран тест за калкулатор



Фигура генерирани тестове за калкулатор

### Изводи

1. Замерването на отделни характеристики на командите е изключително полезно и дава по-детайлна информация за състоянието на системата
2. Малкият проект дава много по-добра яснота за това къде може да има проблем модела (повтарящи се методи)

## Експеримент 6:

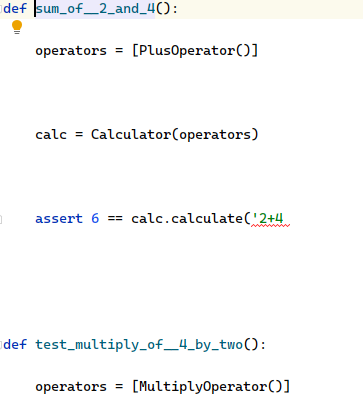
Тъй като при минали тестове е забелязано повторение на едни и същи проверки и почти еднакви тестови случаи (Фигура 42, Фигура 47), в някои случаи дори 100% еднакви са променени параметрите по модела след различни опити в онлайн платформата на OpenAI. Параметрите са такива каквито са описани в III глава на книжното тяло. С помощта на параметрите за температура и наказание при повторения трябва да се намалят до минимум копираните повторения.

Оценка на калкулатор.

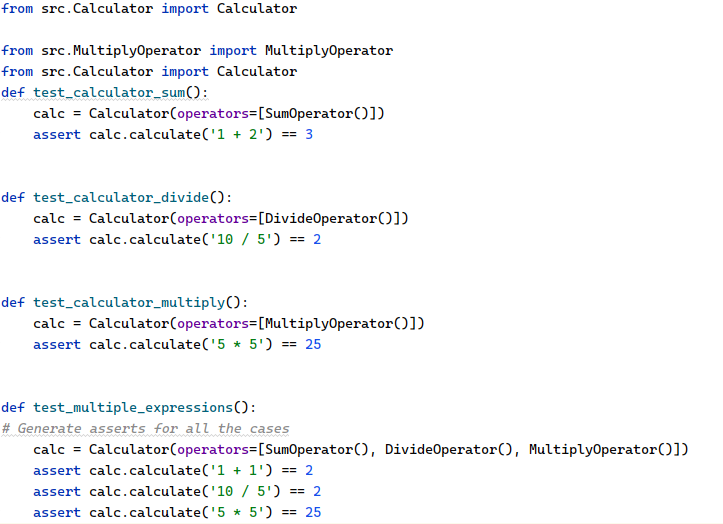
|  |  |
| --- | --- |
| Всички версии: | 6 |
| Версии, които се смятат за валидни: | 6 |
| Версии, за които се генерират тестове: | 6 |
| Брой генерирани файлове: | 16 |
| Изцяло компилируеми файлове [%]: | 18,75 % |
| Брой тестове, които не минават заради грешка в проверка: | 7 |
| Брой тестове, които минават успешно: | 1 |
| Брой предупреждения от pytest | 16 |
| Брой компилационни грешки в тестове | 12 |

### Тестови извадки:

Произволни извадки, които показват генерирания код:



Фигура генериран код със синтактична грешка



Фигура Генериран код за калкулатор

### Изводи:

1. Промяната на параметрите на моделът имат изключително голямо значение, защото повтарящите се тестове генерират грешки за съществуващ метод, които са отстранени
2. Ако моделът е настроен „по-креативно“ се справя по-добре.

## Експеримент 7:

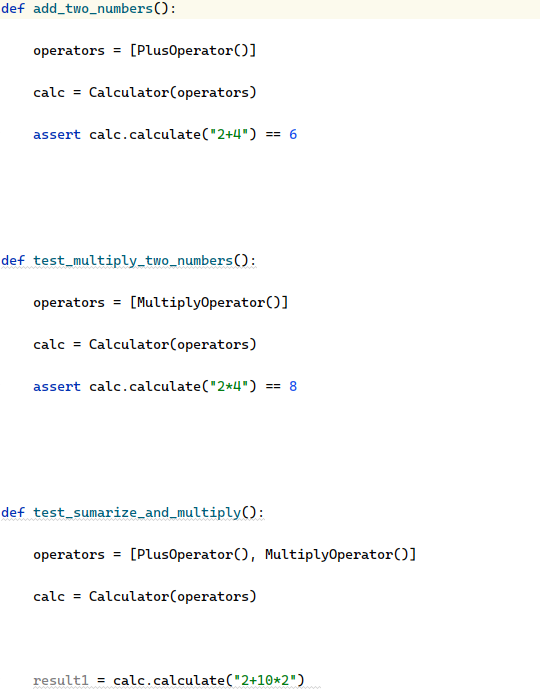
След инспекция на тестовете, които не се компилират се забелязва, че има код, който не е довършен поради лимита на модела. За да се автоматизира оправянето на този проблем е добавена допълнителна стъпка, която извиква Codex модела в режим на редакция със задачата да отстрани компилационни грешки.

Последната версия е пусната само срещу проекта калкулатор.

|  |  |
| --- | --- |
| Всички версии: | 6 |
| Версии, които се смятат за валидни: | 6 |
| Версии, за които се генерират тестове: | 6 |
| Брой генерирани файлове: | 16 |
| Изцяло компилируеми файлове [%]: | 75 % |
| Брой тестове, които не минават заради грешка в проверка: | 30 |
| Брой тестове, които минават успешно: | 0 |
| Брой предупреждения от pytest | 16 |
| Брой компилационни грешки в тестове | 2 |

### Тестови извадки:

Произволни извадки, които показват генерирания код:



Фигура код, който е оправен от Codex виж фигура 48

## Актуална версия:

След инспекция на седмия експеримент са открити няколко проблема, които са отстранени. Отстранените проблеми са свързани с поправката на генерирания програмен код.

|  |  |
| --- | --- |
| Всички версии: | 6 |
| Версии, които се смятат за валидни: | 6 |
| Версии, за които се генерират тестове: | 6 |
| Брой генерирани файлове: | 16 |
| Изцяло компилируеми файлове [%]: | 0 % |
| Брой тестове, които не минават заради грешка в проверка: | 14 |
| Брой тестове, които минават успешно: | 20 |
| Брой предупреждения от pytest | 1 |
| Брой компилационни грешки в тестове | 5 |

Както се вижда от резултатите, вече много по-голяма част от тестовете минават. Това навежда на мисълта вече да се направят замервания и за покритие на тестовете. Тук най-важната метрика е % покритие:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Идентификатор на версията | Извикани редове | Пропуснати редове | Покритие% |
| 0 | 0c023115abd8955c033517a1f9c551ea6ea8a10b | 63 | 41 | 35 |
| 1 | 0c023115abd8955c033517a1f9c551ea6ea8a10b | 63 | 41 | 35 |
| 2 | 0c023115abd8955c033517a1f9c551ea6ea8a10b | 63 | 41 | 35 |
| 3 | de7311be6fdbb0a4ecfe8b84beb10083cace5d8d | 83 | 48 | 42 |
| 4 | de7311be6fdbb0a4ecfe8b84beb10083cace5d8d | 128 | 78 | 39 |
| 5 | 105af9556d1c54a4e1e454e8bf079dfe0b96e061 | 22 | 20 | 9 |
| 6 | 105af9556d1c54a4e1e454e8bf079dfe0b96e061 | 68 | 53 | 22 |
| 7 | 527bd7f43f8dd8d0eba6621f0cc826635060156e | 25 | 4 | 84 |
| 8 | bee966e56f922cda684ae586b2fc3d3eb6acd1e9 | 29 | 7 | 76 |
| 9 | bee966e56f922cda684ae586b2fc3d3eb6acd1e9 | 37 | 28 | 24 |
| 10 | b8f9c9dedb125eddf9f851fc1dcc0dbacb055f03 | 11 | 8 | 27 |
|  | Средна стойност | 53,8181818 | 33,545 | 38,90909 |

Както се вижда средното покритие на генерираните тестове е 39%, като за някои версии достига до 84% от тествания код.

Последната версия е пусната и за httpie, генерирането за целия проект отне 24085 секунди, което са около 6 часа и 41 минути.

Резултати за истинският проект са описани в таблицата отдолу:

|  |  |
| --- | --- |
| Всички версии: | 1725 |
| Версии, които се смятат за валидни: | 103 |
| Версии, за които се генерират тестове: | 103 |
| Брой генерирани файлове: | 492 |
| Изцяло компилируеми файлове [%]: | 20.37 % |
| Брой тестове, които не минават заради грешка в проверка: | 180 |
| Брой тестове, които минават успешно: | 26 |
| Брой предупреждения от pytest | 3 |
| Брой компилационни грешки в тестове | 121 |

От гледна точка на покритие резултатите също са задоволителни:

Всички стойности са средни, пълните стойност могат да бъдат намерени в придружаващия xlsx файл, който може да бъде намерен в глава приложение:

Извикани редове: 96,14

Пропуснати редове: 60

Покритие: 47.5 %

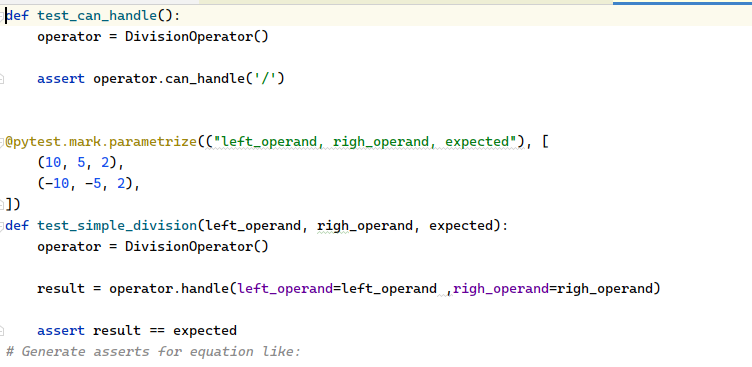
### Тестови извадки:



Фигура



Фигура



Фигура

### Изводи:

1. След поправката на APR системата значително се подобрява ефективността на успешно минаващите тестове.
2. Тъй като вече генерацията се справя много по-добре е добавена и метриката за покритие. От нея се вижда, че за някои версии се генерират изключително добри резултати с над 80% покритие, докато при други е по-ниско стигащо до 10%. Това се дължи основно на разликата в съобщенията, които са част от промяната. По-добри постижения има при по-изчерпателни съобщения като: Added support for minus:Acceptance Criteria:10 - 5 = 55 - 10 = -51 + 3 \* 2 - 5 = -8 и при промени, които са само в една функционалност. Пример е промяната на обработка на входен низ, която налага промени и в различните операции – това са тестовете, с най-слаб резултат.

1. От направените извадки се вижда, че се генерират и по-комплексни тестове като тези от фигури 47 и 48.

## Допълнително обучение на GPT-3/Codex

Има много и различни набори от данни, които се използват при обучението на трансформатори и други модели за генериране на код от естествен език или обратното. За жалост няма готови качествени данни, които да съдържат в себе си функционални тестове обвързани с кода, за който се отнасят и съобщение описващо функционалността.

Тъй като няма готови данни, остава възможността да се обучи допълнително Codex. Тук идеята е всяко хранилище да има собствен допълнително обучен модел, който е обучаван на всички исторически версии на проекта. За целта е създаден класът FineTuningGenerator (Фигура 54). Той събира историческите данни от проекта и създава нужните данни за допълнително обучение на някой от моделите – във файл с формат JSONL.

Данни, които се събират за едно хранилище са от порядъка на няколко стотин примера, което противоречи с препоръката на OpenAI, която гласи, че смисъл за допълнително обучение би имало ако данните, с които ще се учи моделът са с много високо качество и са поне няколко десетки хиляди.

Въпреки това имах желание да пробвам, но за жалост тъй като всеки от примерите е много дълъг тренирането за няколко стотин примера струва от порядъка на няколко стотин щатски долара. Предвид факта, че вероятността да има положителен ефект е минимална, защото данните са малко, с неясно качество, защото са автоматично генерирани (връзката между тестовете и сорс код както тази между сорс и съобщения) този подход не е пробван в разработката на дипломната работа.



Фигура Клас с функционалност за подготвяне на данни за обучение на Codex.

# Примерна употреба и валидация на системата

Изискването към системата е да засича нова версия на разработвания софтуер и да генерира тестове за нея.

За да се удовлетворят тези изисквания е избран GitHub Actions тъй като както е описано във втора предоставя възможност за детекция на промени в състоянието на хранилището на кода съответно и неговата версия.

За да се използва проекта е нужно той да бъде част от хранилището на код, на проекта, за който ще да бъде използван. За да се използва системата е нужен работен процес, който да създава дефинираната виртуална среда от environment.yml, след което да се извика generate\_tests\_code.py.

Примерен такъв процес има в хранилището на дипломната работа. Той може да бъде видян на Фигура 55.

Този процес започва при появата на заявка за сливане (Pull Request). Когато това се случи, хранилището преминава към клона, от който е направена заявката. На този клон се създава средата с всички нужни за нея библиотеки – това се описва от екшъна conda-incubator/setup-miniconda@v2. След като тази стъпка е готова се свалят допълнителните данни нужни за NLTK с командата python –m nltk.downloader punkt stopwords. С тази команда подготовката на системата приключва.

Следва генерирането на тестове – извиква се командата: python generate\_tests\_code.py. Много е важно да се отбележи, че тя разчита на една променлива на средата – OPENAI\_API\_KEY. Тя трябва да бъде дефинирана като тайна в GitHub хранилището.

Последната стъпка е създаване на нова заявка за сливане, в която се съдържат всички генерирани тестове.

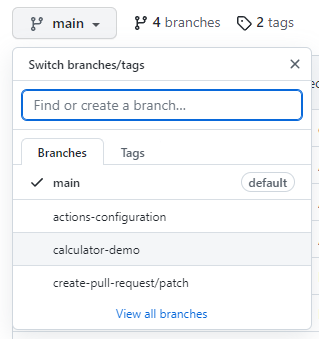


Фигура Работен процес използващ системата за генериране на тестове.

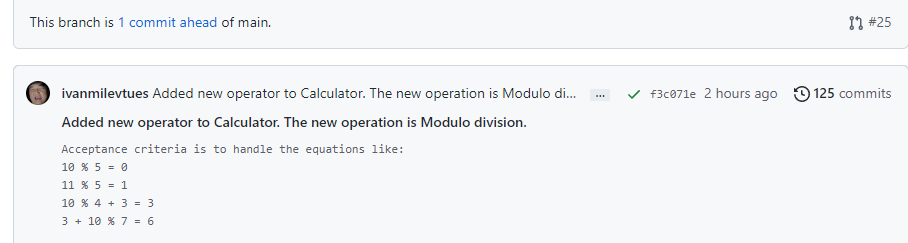
## Пример:

За да се експлоатира системата. Първо се създава нов клон, така наречения клон за нова функционалност (feature branch). В него се добавя новата функционалност на продукта. В случая се създава клон с името calculator-demo (фигура 56).

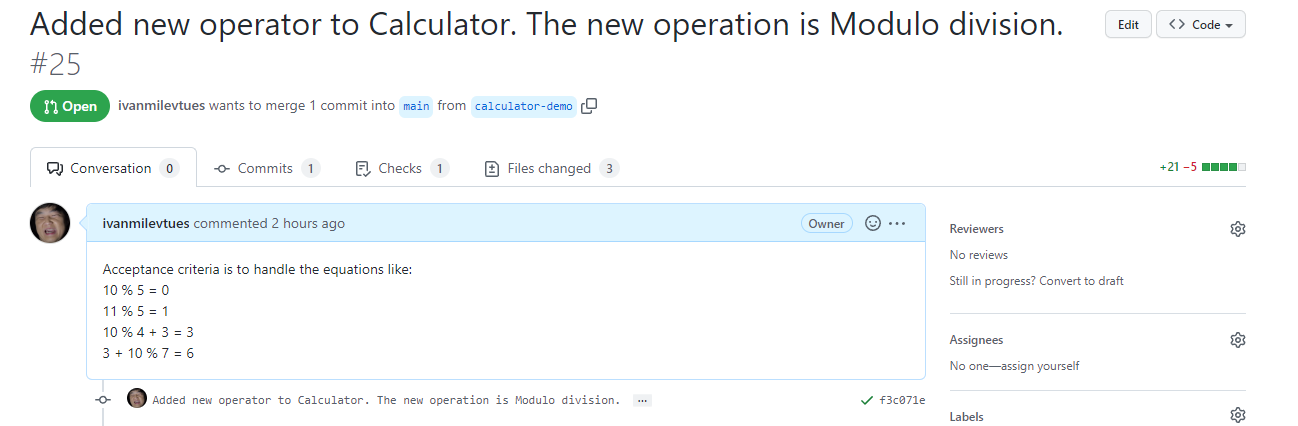
След като се създаде новият клон в него се разработва новата функционалност и се добавя като нова версия (commit) в нея (Фигура 57). След като програмистът реши, че новата версия е готова за употреба се прави заявка за сливане (Фигура 58). Тази заявка ще стартира изпълнения на дефинирания работен процес от (Фигура 55), която ще генерира тестове и ще създаде още една заявка за сливане, която съдържа новите тестове.



Фигура Създаденият нов клон calculator-demo



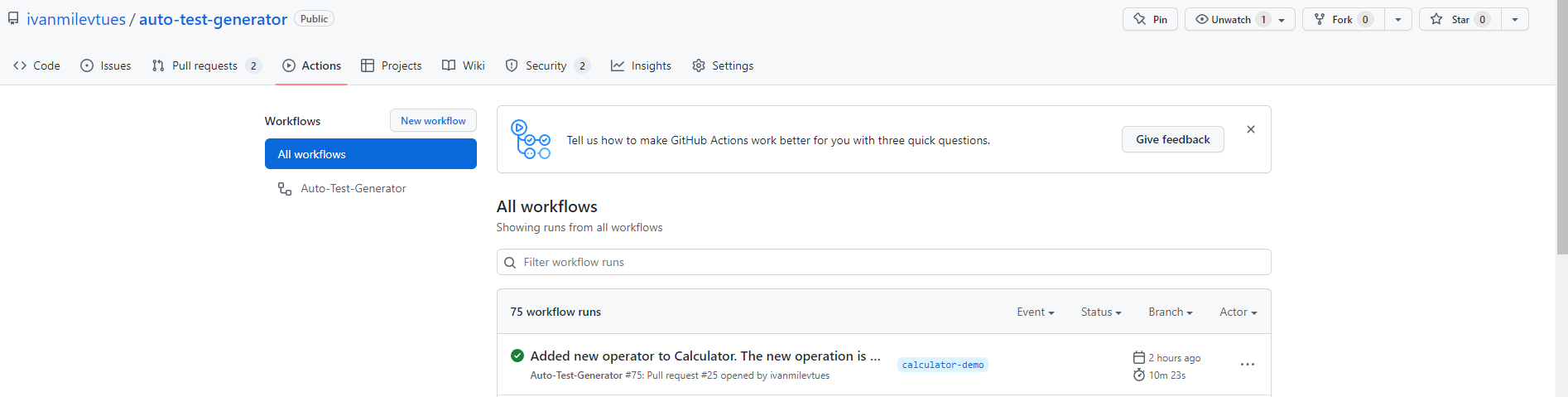
Фигура добавяне на новата функционалност в calculator-demo



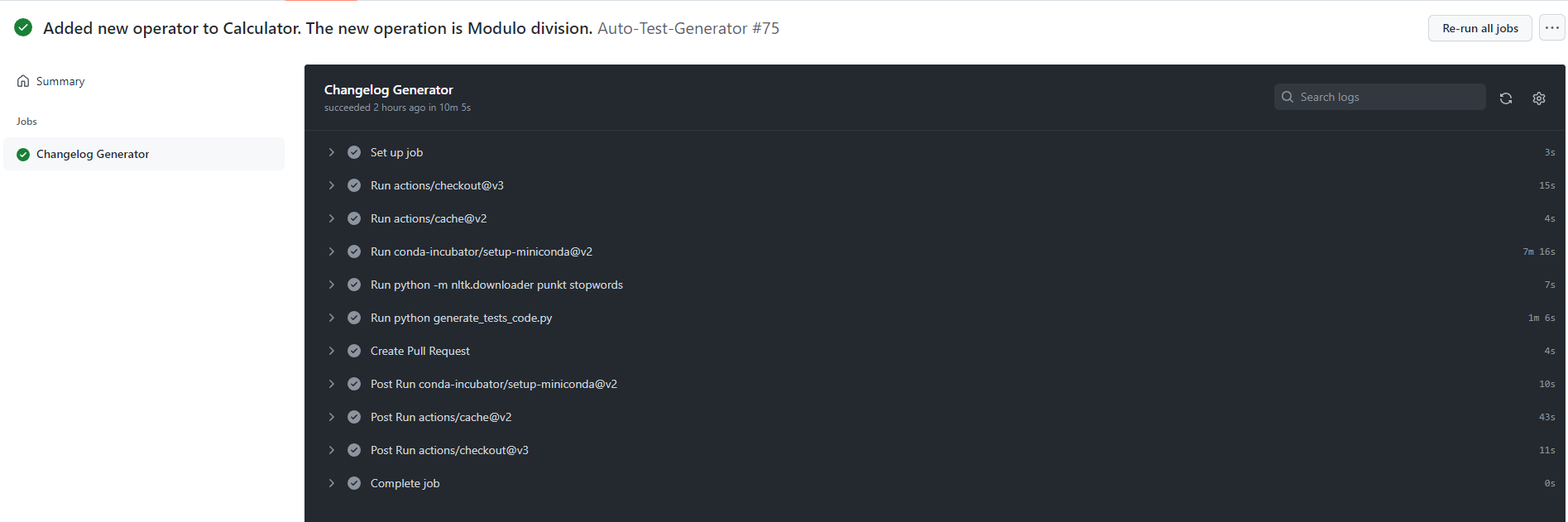
Фигура Създадена заявка за сливане към основния клон на новата функционалност.

Всички изпълнения на GitHub Actions мога да бъдат видени в потребителския интерфейс на Github. Той се намира в страницата на хранилището на кода в подразделение actions (Фигура 59). След като се избере някое конкретно пускане могат да се разгледат изпълнените стъпки и коя, колко време е отнела (Фигура 60).

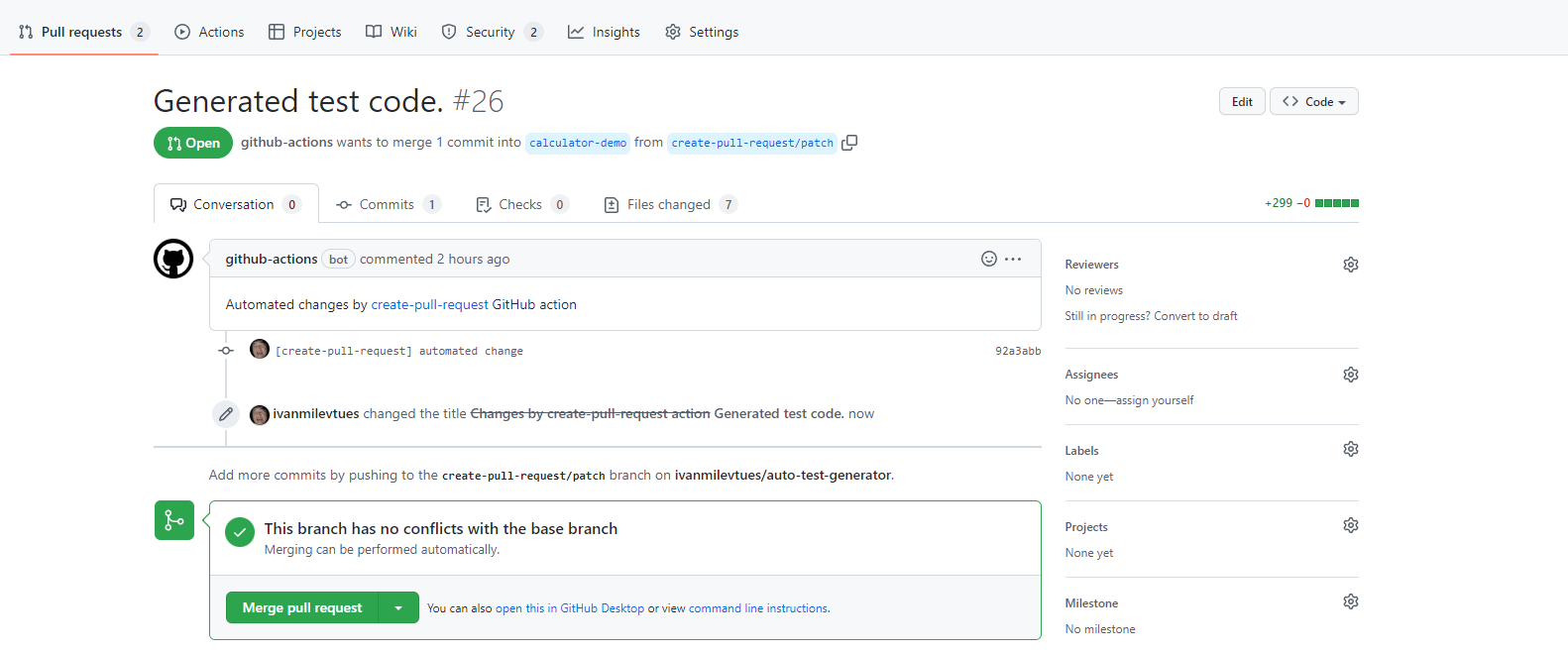
След изпълнението на всички стъпки се създава нова заявка за сливане, както може да бъде видяно на фигура 61 и фигура 62.



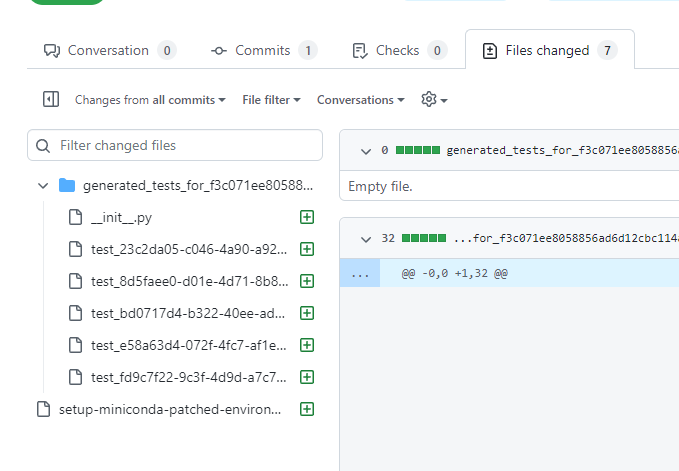
Фигура Списък с изпълнените Github Actions



Фигура Списък с изпълнените стъпки за даден работен процес



Фигура заявка за сливане с генерираните тестове



Фигура Новодобавени файлове към заявката за сливане

За бъдещо развитие би било целият проект обвързан с генерирането на тестове да бъде създаден като отделен GitHub action, но това за момента е невъзможно, тъй като проекта е разработван под windows и е зависим от операционната система, а те не се поддържа за изпълнението на персонални action-и. За да стане това възможно е проектът да бъде преправен така, че да работи под Linux.

# Заключение

В тази дипломна работа е представен нов подход за генериране на автоматични тестове – функционални и интеграционни тестове. Целта на разработената система, която съпътства подхода е да покаже възможностите на най-големите езикови модели в комбинация с познати подходи използвани при модулните тестове и други похвати.

Появата на нови и по-големи езикови модели, които все повече се доближават до модели, които могат да се ползват с общо предназначение променя начина, по който се пише софтуер. Както е споменато в първа глава при обзора на GPT-3 и Codex както и във втора глава, тези модели са огромни и се стремят да бъдат все по-общи, така че да елиминират нуждата от допълнително обучение за конкретна задача.

От разработката става ясно, че това е напълно възможно, Codex е способен да генерира функционални и интеграционни тестове. Оказва се, че при едно по-добро автоматично засичане и поправяне на грешки в кода (генериран или не) тези модели ще отключат още много от възможностите си. В много от случаите в гореизложените примери моделът е близо до това да генерира правилен тест, но малки неща като синтактични грешки и не трудно поправими логически грешки го делят от това теста да е изцяло функциониращ.

За развитието на проекта има две основни посоки – създаване на още по-добра обработка на генерирания код и съответно намиране на оптимални входни данни, които да дават правилния контекст и да задават още по-добре задачата към модела и подобряване на поправянето на генерираните тестове.

Разликите между разработената дипломна работа и съществуващите решения са големи. До тук всички съществуващи решения се опитват да решат проблема на писането на модулни тестове, докато тази дипломна работа се опитва да реши по-абстрактен проблем – генериране на тестове според описанието, което е дадено за съответната функционалност. Много от похватите, които са използвани за постигане на тази цел са заимствани от вече съществуващи решения, но са приложени по нов начин.

За демонстрирането на работата на системата е използван специално разработен за целта проект – счупен калкулатор. Към неговото хранилище е добавен проектът и при добавяне в основния клон се генерират тестове, които се добавят в нов клон, където потребителят може да ги достъпи и разгледа. Работата е достъпна за всеки с профил в GitHub.

Автоматичната генерация на тестове е още млада сфера в софтуерното инженерство. С последните проучвания в областта на генерирането на код и все по-доброто разбиране на връзката между естествени и програмни езици от машинно обучените модели все повече тази идея се доближава до реалността.

# Използвана литература

[1] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, Neel Sundaresan, “Unit Test Case Generation with Transformers and Focal Context”, <https://arxiv.org/pdf/2009.05617.pdf%20%E2%80%93%20unit%20Test%20Case%20Generation%20with%20Transformers%20and%20Focal%20Context>   
- статия

[2] GitHub CoPilot - <https://copilot.github.com>   
- уебсайт

[3] EvoSuite - <https://www.evosuite.org/evosuite/>   
- уебсайт

[4] Radoop - <https://randoop.github.io/randoop/>

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei, „Language Models are Few-Shot Learners“, <https://arxiv.org/pdf/2005.14165.pdf>   
- статия

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, Wojciech Zaremba, “Evaluating Large Language Models Trained on Code”, <https://arxiv.org/pdf/2107.03374.pdf>  
- статия

[7] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, Shin Hwei Tan, “Improving automatically generated code from Codex via Automated Program Repair”, <https://aps.arxiv.org/pdf/2205.10583.pdf>   
- статия

[8]Joel Jones, “Abstract Syntax Tree Implementation Idioms“ <https://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf>  
- статия

[9] Abstract Syntax Tree Image <https://en.wikipedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg>  
- фигура

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, Ming Zhou, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”, <https://arxiv.org/pdf/2002.08155>   
- статия

[11] NLTK, <https://www.nltk.org>   
- уебсайт

[12]”Pandas Overview”, <https://www.nobledesktop.com/learn/python/pandas-overview>  
- уебсайт

[13] “Pandas“, <https://pandas.pydata.org/about/>   
- уебсайт

[14] PyDriller, <https://pydriller.readthedocs.io/en/latest/>   
- уебсайт

[15] Официална документация на Python, <https://docs.python.org/3/library/ast.html>   
- уебсайт

[16] OpenAI, <https://beta.openai.com/docs/introduction>   
- уебсайт

[17] Jupyter, <https://jupyter.org>   
- уебсайт

[18] Anaconda, <https://www.anaconda.com>   
- уебсайт

[19] JetBrains Pycharm, <https://www.jetbrains.com/pycharm/>   
- уебсайт [интегрирана среда за разработка]

[20] Visual studio code, <https://code.visualstudio.com>   
- уебсайт [интегрирана среда за разработка]

[21] Gwern Branwen, “GPT-3 Creative Fiction”, <https://www.gwern.net/GPT-3#prompts-as-programming>   
- уебсайт

[22] OpenAI Completion Documentation, <https://beta.openai.com/docs/api-reference/completions>   
-уебсайт

[23] Amine Elhattami, “A Gentle Introduction to Code Generation Evaluation” <https://towardsdatascience.com/a-gentle-introduction-to-code-generation-evaluation-c8dff8c3d19a>   
- уебсайт

[24] Ben Mann, “How to sample from language models”, <https://towardsdatascience.com/how-to-sample-from-language-models-682bceb97277>   
- уебсайт

[25] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu, “BLEU: a method for automatic evaluation of machine translation” <https://dl.acm.org/doi/10.3115/1073083.1073135>   
- статия

[26] Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, Chris Brown, “FixEval: Execution-based Evaluation of Program Fixes for Competitive Programming Problems”, <https://arxiv.org/pdf/2206.07796>   
- статия

[27] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, Jacob Steinhardt , “Measuring Coding Challenge Competence With APPS”, <https://paperswithcode.com/paper/measuring-coding-challenge-competence-with>   
- статия

[28] <https://paperswithcode.com/dataset/apps>   
- набор от данни

# Приложение

Връзка към програмния код: <https://github.com/ivanmilevtues/auto-test-generator>

Връзка към данните използвани за таблиците в глава IV: <https://github.com/ivanmilevtues/auto-test-generator/tree/main/docs>

Данните за различните проекти (httpie, PyDriller, Flask, Calculator) са в едноименните си xlsx файлове: calculator.xlsx, pydriller.xlsx, BasicPostProcess\_flask.xlsx, httpie.xlsx