TDK-dolgozat

Készítette:

Bódi Martin

Szegedi Tudományegyetem Természettudományi és Informatikai Kar
Informatikai Intézet
Szoftverfejlesztés Tanszék

Kódminőség javítása GPT Modellekkel: Clean Code elvek automatikus alkalmazása Python nyelvű projektekre

Készítette:

Bódi Martin

programtervező informatikus
III. évf. BSc hallgató

Témavezető:

Dr. Antal Gábor

adjunktus

Szeged
2024

ABSZTRAKT

A rosszul karbantartott kód használata jelentősen csökkentheti a produktivitást és növelheti a hibák és sebezhetőségek létezésének valószínűségét. A clean code (tiszta kód) elveket nem, vagy mérsékelten követő projekteken sokkal nehezebb és kockázatosabb lehet dolgozni, hiszen ezen elvek elhanyagolása rossz karbantarthatósághoz vezethet. Olyan projektek esetén is, ahol külön figyelnek ezen elvekre, bizonyos idő után elkerülhetetlen, hogy a kód minősége romoljon. A kód refaktorálása, karbantartása nagy költséget jelenthet, így szükségesek lennének az erre a célra készített, automatikus módszerek. Azonban a kódok tisztaságának ellenőrzése és az esetleges problémák mitigálása kifejezetten komplex feladatok.

A nagy nyelvi modellek (Large Language Models, LLMs), mint például az OpenAI GPT modelljei, képesek kifejezetten jól teljesíteni olyan komplex problémák esetén is, amelyeket tradicionális algoritmusok csak nehézkesen tudnak kezelni. A dolgozatban az imént említett modell két verzióját (3.5-Turbo, 4-Turbo) használtuk arra, hogy Python nyelvű projekteken automatikus karbantartást végezzünk. A két modell által elvégzett karbantartásokat két felhő alapú statikus kódelemző szolgáltatás (Qodana, SonarQube) segítségével vizsgáltuk meg, összpontosítva a kódminőséget, karbantarthatóságot jellemző metrikákra. Az eredmények alapján mindkét modell képes jelentősen javítani a kódban lévő bad smellek arányát.

KULCSSZAVAK

- Automatizált kódtisztítás
- Tiszta kód (Clean Code)
- GPT
- Gépi Tanulás

Tartalomjegyzék

1. fejezet

Bevezetés

Egyre növekvő népszerűségnek örvendeznek az úgy nevezett tiszta kód (Clean Code) [1] elvek, melyeket Robert C. Martin foglalt össze és népszerűsítette el könyvében. Az elvek lényege, hogy irányt mutassanak a fejlesztőknek, hogy hogyan készítsenek úgy kódot, hogy könnyen karbantartható, értelmezhető legyen. Ezáltal megkönnyítve az együttműködést és a kód továbbfejlesztését, továbbá csökkentheti a hibák számát és annak esélyét, hogy új hibákat hozunk létre. Mindezen túl növeli a kód megbízhatóságát és skálázhatóságát. Számos kutatást [2,3,4,5] készítettek, melyek alátámasztják ezen állításokat.

A projektek mérete ahogyan nő és olyan új funkcionalitásokat implementálnak, amelyekkel a fejlesztés kezdetekor nem számoltak, úgy egyre nő a bad smellek száma és egyre nehezebbé válik a kód karbantartása, továbbfejlesztése. Ezt refaktorálással szokták orvosolni, amely azonban kifejezetten költséges feladat. Emiatt szükség lehet olyan automatizált megoldásra, amely megoldja, vagy legalábbis segíti a fejlesztőket a kódtisztításban.

A tiszta kód elvek ellenőrzése, mérése, valamint az elvek, már meglévő kódbázisra alkalmazása viszont meglehetősen összetett és bonyolult feladatok. Hiszen az elvek nem egzakt módon fogalmaznak, nem egy bizonyos programozási nyelvre fókuszálnak és a program szemantikájára is kitérnek. Például a SOLID elvek - melyek részét képezik a tiszta kód elveknek - megkötik, hogy mely logikailag összetartozó működéseket szabad egységbe szervezni, és melyeket kell szétbontani. Emiatt az automatizáláshoz intelligens algoritmusra lehet szükség.

Az elmúlt években hatalmas népszerűségnek örvendeznek a nagy nyelvi modellek (LLMs), mivel képesek kifejezetten jól teljesíteni olyan komplex problémák esetén is, amelyeket tradicionális algoritmusokkal nagyon nehéz kezelni. Munkánk során az OpenAI két népszerű modelljét (GPT 3.5-Turbo és 4-Turbo) használtuk arra, hogy automatikus karbantartást végezzünk Python nyelvű projekteken és arra kerestük a választ, hogy képesek-e a kód minőség javításában segítséget nyújtani. A nagy nyelvi modellekről és a GPT-ről a kapcsolódó munkák fejezetben részletesebben tárgyalunk.

Sajnos nincs elérhető adatbázis, amely tartalmazna kódokat és azok megtisztított állapotát, sem olyan, amely kifejezetten tiszta kód elveket sértő kódokat tartalmazna. Emiatt nekünk kellett adatokat gyűjteni és úgy döntöttünk, hogy Python nyelven írt GitHub projekteket gyűjtünk, ezáltal gyakorlati példákon tesztelhetjük a modellek teljesítményét, egy olyan programozási nyelven, amely a TIOBE népszerűségi indexe alapján a legnépszerűbb nyelv. A kódhalmazunk GPT-3.5-Turbo esetén több, mint 49 ezer kódsorból, GPT-4-Turbo esetén több, mint 10 ezer kódsorból tevődött össze.

Az eredmény kiértékelésére a legalkalmasabb egy olyan mesterséges intelligencia alapú program lehetne, amely értékel egy megadott kódot aszerint, hogy mennyire tiszta. Azonban ilyen alkalmazás jelenleg nincs, ezért mi az eredmény mérésére statikus kódelemzők felhő alapú szolgáltatásait használtunk (Qodana, SonarQube). Ezáltal rendelkezésünkre állt számos olyan adat, amely a kód tisztaságát, karbantarthatóságát és értelmezhetőségét méri. Többek közt megmértük a code smellek számát, a technical debt és a cognitive complexity mértékét.

Tapasztalataink alapján mindkét modell képes jelentősen javítani a bad smellek arányát. Ez 3.5-Turbo esetén azt jelentette, hogy a bad smellek száma 43,8%-ára, míg 4-Turbo esetén 38,7%-ára csökkent a kezdeti állapothoz képest. Jelentős különbség azonban, amely torzíthatja is az összképet, hogy GPT-3.5-Turbo esetén a kimeneti fájlok ~17%-a szintaktikai hibát tartalmazott. Ez több, különböző, a modell által vétett hibából eredeztethető. Viszont GPT-4-Turbo esetén egyetlen egy szintaktikai hiba sem fordult elő a kimeneti fájlokban. Fontos megjegyezni, hogy a 3.5-Turbo hibái nem egy bizonyos bemeneti projekthez köthetők és több olyan projekten is előfordultak, melyeken a 4-Turbo modellt is lefuttattuk.

A dolgozat további része a következőképpen van szerveze. A 2. fejezetben említést teszünk a témakörünkhöz kapcsolódó munkákról és összehasonlítjuk az eredményüket, kutatásuk célját és módszertanukat a miénkkel. A 3. fejezetben bemutatjuk és elmagyarázzuk a módszertant, amelyet alkalmaztunk a kutatásunk során. Ezt követően elmagyarázzuk az eredményünk kiértékelésére használt módszertanokat, metrikákat 4. fejezetben. Ezután megvitatjuk a felmerülő problémákat, és lehetőségeket, amelyekkel tovább lehetne optimalizálni az eredményt (5. fejezet). Ezzel érintve olyan területeket, melyek további kutatást igényelhetnek. Ezt követően a 6. fejezetben említést teszünk olyan tényezőkről, amelyek veszélyt jelenthetnek a munkánk validitására. Végezetül összegezzük a kutatásunk eredményét a 7. fejezetben.

2. fejezet

Kapcsolódó Munkák

Munkánk számos aktív kutatási területet érint, melyek nagy befolyással lehetnek a szoftverfejlesztői iparra. Kezdve a meglévő projektek refaktorálási módjaitól, az ezt végrehajtó rendszerek fejlesztésén át, a refaktorálás hatásainak szerteágazó vizsgálatáig, mind vizsgált kérdések. Refaktorálással próbálják a szoftverben előforduló sérülékenységeket, kód minőség hibákat megszüntetni, ezáltal könnyítve a fejlesztést. Az elmúlt években nagy népszerűséget szerző nagy nyelvi modellek hatását is vizsgálják széleskörű témakörökben, tudásukat összehasonlítják. Emellett a mesterséges intelligencia nem csak a generálás terén, hanem a mérés területén is egyre gyakrabban használatos. Kiemelendő, hogy jelenleg nem található olyan kutatási cikk, amelyben nagy nyelvi modelleket használnának arra, hogy kód minőséget javítsanak. Emiatt számos problémát és kérdést elsőként kellett megoldanunk.

Fontos kérdés, hogy milyen elveket kell egy jó minőségű kódnak követnie. A clean code elvek ebben segítenek, melyet számos kutatás érint. Többek közt Lj. Kazi, S. Mihajlović és M. Bhatt cikke [1] érinti a minőségi tulajdonságait a tiszta kódoknak, a kód tisztaságának mérését és a refaktorálást is. Minőségi tulajdonságnak számít, hogy a kód könnyen tesztelhető, fenntartható, érthető, emellett a hatékony és a biztonságos (nehezen sebezhető). Összegyűjtötték továbbá a vonatkozó elvárásokat, Robert C. Martin könyve alapján. Ezek megkötést tesznek a különböző kód részekre (például: függvény, osztály, komment), az elnevezésekre, a formázásra és a működésre is. Emellett kigyűjtötték azokat a tevékenységeket is, melyek támogatják a tiszta kód fejlesztését. Ide tartozik a teszt alapú fejlesztés (Test Driven Development, TDD), a SOLID elvek követése, az egység tesztelés (unit teszt), a code smell heurisztikák használata, a refaktorálás és még sok egyéb technika, amely segíthet a tiszta kód fejlesztésében. A kód tisztaságának mérésére olyan kvantitatív metrikát ajánlottak, amely a kód olvashatóságát méri, illetve a kód olvashatóság alapú klasszifikációját. Ezen módszereket részletesen az alábbi cikkek tárgyalják. [9,10,11] Refaktorálás tekintetében kitértek az adatbázisok használatára, illetve arra, hogy előnyösebb ez a megközelítés a szintaxis fák használatával szemben. Említést tettek a funkcionális programozásra való refaktorálás előnyeiről.

A tiszta kód előállításának gyakorlatairól és segítő eszközeiről számos más cikk is tárgyal [4,3]. Az imént említett első kutatásban kifejezetten arra fókuszáltak, hogy mely folyamatok és eszközök képesek segíteni a tiszta kód fenntartását, hogy hosszú életű lehessen a kód. Hasonlóan az előző bekezdésben tárgyalt kutatáshoz, említést tesznek a teszt alapú fejlesztés és a statikus kódelemzés használatáról. Emellett kiemelik a kód felülvizsgálását, ellenőrzését más fejlesztő által (code review), amely nagy mértékben képes lehet csökkenteni a hibák számát. Ezt arra alapozzák, hogy két külön fejlesztő eltérő tudással, szakmai háttérrel rendelkezhet. Ezáltal más szempontból is ellenőrzésre kerül a kód. Továbbá említést tesznek a gyakran ismételt feladatok automatizálásáról, az ezt támogató rendszerek használatáról (Continuous Integration/Continuous Delivery, CI/CD), mely könnyítheti, gyorsítja és átláthatóbbá teszi a rendszert. Munkánk során használtunk automatizálást a statikus elemzések automatikus futtatására és az elavult riportok törlésére, mely jelentősen megkönnyítette az eredmény kiértékelését, elemzését.
A második említett cikk [3] esetében kifejezetten olyan aspektusokra fókuszáltak, melyek nem változtatják meg a funkcionalitást és programozási nyelvtől függetlenül, strukturálisan megvalósíthatóak. Kitértek a rossz kód néhány aspektusára, amelyekről már bővebben tettek említést az előző részben említett kutatásban [1], illetve röviden tárgyalták a rossz kód hatását az alkalmazásokon és felhasználókon. Valamint a különböző minőségi attribútumokat biztosító kód írási és refaktorálási technikákat ismertettek.

Probléma azonban, hogy a fejlesztők nem tudnak előre felkészülni olyan funkcionalitások támogatására, amelyeket még nem tudnak. Emiatt a továbbfejlesztett szoftver technológiai adóssága folyamatosan nő. Tehát lehet, hogy számos módszert használnak a minőségi kód előállítása érdekében, de idővel számukra is szükséges lehet a kód refaktorálása. Ezen okokból kifolyólag jelenleg is fejlesztenek több olyan rendszert, amely a kód minőségét mitigálja, vagy segít ebben. Ilyen eszközhalmaz például a FaultBuster [12,13], munkájuk során kitértek arra, melyet mi is tárgyaltunk, hogy a kód refaktorálás nem egyszerű feladat. Ugyanis detektálni kell először, hogy hol van probléma, majd el kell dönteni, hogy mely kódrészt kell refaktorálni. Ezt követően refaktorálunk, majd le is kell ellenőrizni, hogy elrontott-e valamilyen működést a változtatás. Kitérnek továbbá arra, hogy a fejlesztői környezetek számos refaktorálási lehetőséget nyújtanak, azonban az olyan eszközök, amelyek tényleg automatikusan képesek refaktorálni, még kutatás alatt állnak. A FaultBuster statikus kód elemzéssel megtalálja a bad smelleket, majd futtat rajtuk több tisztító algoritmust, végezetül az integrált teszteket lefuttatja. Ezen eszköz használata hasonló lehet, a mi általunk készített alkalmazásnak, a különbség viszont, hogy míg az ők hagyományos algoritmusokat használtak és építettek egybe, addig mi a teljes feladatkört rábíztuk az általunk használt nagy nyelvi modellre. Ugyanis a mi esetünkben a GPT modellek nem kaptak információ arról, hogy hol fordul elő, milyen bad smell.

A refaktorálásra azonban nem csak a kódminőség terén fókuszálnak, kifejezetten fontos kutatási terület ugyanis a sérülékenységek automatikus javítása. Többek közt fejlesztettek egy olyan end-to-end keretrendszert [14], amely a biztonsági hibák detektálására és javítására fókuszál. A kutatási terület nehézsége, hogy biztonsági résekre, sérülékenységekre nem lehet a funkcionalitásnál használt egység teszteket írni, ugyanis ezen hibák váratlan és kezeletlen helyzetekből adódnak.

A clean code és a rafaktorálás hatását számtalan kutatásban és esettanulmányban vizsgálják. Katona József szem-követő rendszer segítségével hasonlította össze a tiszta és piszkos programkódok kognitív nehézségét [5]. Statisztikai elemzése azt mutatta ki, hogy jelentős különbség van az átlagos szem fixálódás hossza között. Amely azt jelentette, hogy jelentősen több ideig tartott értelmezni az egyes részeken a nem tiszta kódok esetén. Ezáltal megerősítette azt az állítást, hogy az értelmezhetőséget jelentősen segítik a clean code elvek.

Hemming Grimeland Koller kutatása [2] során refaktorált egy kódbázist, majd kialakított két fejlesztői csoportot. A kontroll csoport az eredeti, elavult kódon dolgozott, míg a teszt csoport az új, tiszta kódon. A kisérlet folyamán kiosztottak feladatokat mindkét csoportnak és meglepően azt tapasztalták, hogy 2/3 feladatot hamarabb megoldott a kontroll csoport. Ez alapján gondolhatnánk, hogy lehet mégsem segíti az olvashatóságot a tiszta kód, azonban ezen negatív eredmény mellett is az mutatkozott meg, hogy a tiszta kódon dolgozó csoport sokkal jobb minőségű kódot állított elő, mint a kontroll csoport

Nem csak a clean code hatását vizsgálják, fontos kérdés ugyanis a refaktorálás hatása is, hiszen elképzelhető, hogy a refaktorálás eredményeképpen nehezebb lesz dolgozni a kódbázison esetlegesen a nagy mennyiségű változás, vagy valamilyen más okból kifolyólag. Különösen fontos kérdés zárt, nagy méretű, ipari rendszerek esetén a refaktorálás hatása, melyet több empirikus kutatás is vizsgált [15,16]. Eredményüket tekintve azt tapasztalták, hogy megfelelő refaktorálási stratégiák esetén jelentős fenntarthatóság növekedést lehet elérni, illetve jelentősen javulhat a kód minősége. Negatív hatást akkor tapasztaltak, ha túl kicsi méretben alkalmaztak refaktorálást, vagy nem megfelelő refaktorálási stratégiákat használtak. Kutatásunkban mi is tapasztaltunk hasonlót, ugyanis a kis méretű projektek, fájlok esetén sokkal kisebb mértékű fejlődést tudtunk elérni, mint nagyobb fájlok esetén.

A nagy nyelvi modelleket (Large Language Models, LLMs) széleskörű kutatási területen használják jelentős eredmények elérésére, többek közt felhasználták őket kézi exoskeletonok irányításának az inkrementális tanulásának a támogatására [17], de vizsgálják az LLMs hatását a nukleáris medicínákra is [18]. A legtöbb nagy nyelvi modell a transformer architektúrára épül, amely tartalmaz egy encoder és egy decoder komponenseket.

Encoder alapú modellek például a BERT [6] és a CodeBERT [7], melyek specifikusan programkódok kezelésre lettek tervezve. Ezen modellek maszkolt tokenekkel működnek és

megpróbálják kikövetkeztetni, hogy mi lehetett eredetileg a kimaszkolt részeken. Ehhez a maszkolt tokeneket körülvevő környezetet használják fel. Ebből kifolyólag szöveg generálására nem célszerű használni az encoder modelleket.

A decoderek ezzel szemben kifejezetten szöveg generálására hivatottak. Decodert használ a generatív előre tanított transzformer (Generative Pre-trained Transformer, GPT) architektúra is. Ezáltal az OpenAI népszerű GPT modelljei is, melyek közül jelenleg két modell és ezek finomhangolt változatai frissek és relevánsak (3.5-Turbo, 4-Turbo).

Az imént említett két GPT modellt az internetről készített friss állapotképből kinyert adatbázison tanították. Ezáltal hatalmas méretű, mennyiségű adat állt rendelkezésükre, emiatt széleskörű a tudásuk. Az interneten nagy mennyiségű programkód is található, ennek köszönhetően kód generálására is kifejezetten alkalmasak.

Nem meglepő, hogy kutatják az LLM-ek felhasználási módjait a szoftverfejlesztésben is, kutatják, hogy mely instrukció (prompt) képes adott problémát megoldani. Egy ilyen kutatásban [19] kerestek promptokat, olyan problémákra, mint: kódminőség, refaktorálás, követelmény kielégítés, rendszer tervezés, gyors prototípus készítés. Kutatásuk azt mutatta ki, hogy a megfelelő prompt minta használata jelentősen képes csökkenteni az LLM-ek által vétett hiba mennyiségét.

Kapcsolódó projekt Emilia Hansson és Oliwer Ellréus szakdolgozata [20], melyben összehasonlították a ChatGPT és a GitHub Copilot által kiadott kódok helyességét és minőségét. A Copilot egy másik népszerű LLM, amelyet a GitHubon fellelhető hatalmas kódbázison tanítottak. A kutatás pillanatában a ChatGPT a GPT-3.5 modellt használta, amely mai viszonylatban már elavult, viszont eredményük számunkra pozitív. Habár a GPT-3.5 tapasztalataik alapján 87,33%-os helyesség arányt ért el, amely kicsit alacsonyabb a Copilot 89%-os arányánál, de a GPT által generált kódsorok 98,52%-a nem sértett meg kód minőségi elvet, míg a Copilot csak 94,07%-ot ért el. Algoritmusokra vetítve jelentősebb a különbség, mivel a GPT 80,7%-ban nem vétett kód minőségi hibát, addig a Copilot esetén ezt az arány csak 60,7%-nak találták.

Az eredmény kiértékelésére a legalkalmasabb egy olyan mesterséges intelligencia alapú program lehetne, amely értékel egy megadott kódot aszerint, hogy mennyire tiszta. Hasonlót már megvalósítottak, ugyanis fejlesztettek egy tiszta kód írását tanító platformot Clean CaDET [8], amely tartalmaz egy olyan mesterséges intelligencia alapú eszközt, amely ellenőrzi, hogy van-e bad smell a kódban és ha igen, akkor felajánl egy tiszta megoldást. Ez azonban számunkra nem alkalmas, mivel mi a kód tisztaságát nem binárisan kezeltük (tiszta vagy sem), hanem mint tisztasági mérték. Hiszen ahhoz, hogy megtudjuk mondani mennyire képes javítani a kód minőségét a két modell, ahhoz hasznosabb információt adnak a számszerűsített értékek.

3. fejezet

Módszertan

Az általunk alkalmazott folyamat 7 lépésre bontható. Elsőként kigyűjtöttük a teszt adathalmazt. Ezután elő-feldolgoztuk a bemeneti adatot. Ezt követően kiválasztottuk a megfelelő promptot az adott futásra, a promptokat már előre megterveztük. A 4. lépésben dekoráltuk a promptokat további információkkal, amennyiben szükséges volt. Ezt követte a tisztítása procedúra. Ezután következett az utó-feldolgozás, ahol kinyertük az eredmény kódokat a tisztítási procedúra eredményéből. Végezetül kiértékeltük a procedúra teljesítményét.

3.1 Adatgyűjtés

Azért, hogy az eredményünk és kiértékelésünk reprezentatív legyen valós projektek esetén is, az adathalmazunkat különböző méretű GitHub projektekből alkottuk meg. Emellett készítettünk egy saját projektet is (principle_violation), amely lényegében egy fájlból állt. Ebben függvényenként eltérő clean code elvet sértettünk meg, ezzel azt szerettük volna biztosítani, hogy a főbb clean code elvek mindegyikére biztosan legyen azt megsértő példa. Továbbá ezen fájl jó reprezentációja lehet a GPT és az aktuális prompt határainak olyan tekintetben, hogy mely elvek megsértését képes orvosolni és melyeket nem.

Technikai szempontból a felhasznált projektek másolatát a saját GitHub projektünkben (repository) tároltuk. Annak érdekében, hogy a Git ne alprojektként (sub-repository) tekintsen a klónozott projektekre, nem közvetlen projektünkbe klónoztuk a repository-kat, hanem bemásoltuk oda. Ezután töröltük a Git konfigurációs fájlokat tartalmazó ’.git’ mappát, ugyanis ezen fájlok jelenléte esetén nem tárolódik el valójában a mi projektünkben a felhasznált repository-k tartalma, hanem az eredeti projekt változtatásaként kezeli a Git.

A tisztítást és kiértékelést GPT 3.5-Turbo és GPT-4-Turbo esetén is végrehajtottuk, azonban jelentős különbség van futtatási költség tekintetében a két modell közt. Tokenre nézett árazás szempontjából 20-szoros árral rendelkezik a GPT-4-Turbo, emiatt ezen modell futtatásakor a tisztítást és kiértékelést a teljes projekthalmaz egy részhalmazán futtattuk csak.

A felhasznált projektek nevét, illetve méretbeli adatait az I. és II. táblázatok tartalmazzák.

Projektnév	Tokenszám	Kódsor	Fájlszám
AutoRCCar	7623	1032	13
deep-text-corrector	21515	2354	8
gpt-engineer	77827	11271	82
graph_nets	120355	12337	24
principle_violation	1067	132	1
social_mapper	27656	2872	10
Video-Pre-Training	36768	4165	22
you-get	148635	14978	138
Összesen	**441446**	**49141**	**298**

I. Táblázat: *GPT-3.5-Turbo input projektek*

Projektnév	Tokenszám	Kódsor	Fájlszám
AutoRCCar	7623	1032	13
deep-text-corrector	21515	2354	8
principle_violation	1067	132	1
social_mapper	27656	2872	10
Video-Pre-Training	36768	4165	22
Összesen	**94629**	**10555**	**54**

II. Táblázat: *GPT-4-Turbo input projektek*

3.2 Előfeldolgozás

Mivel a nagy nyelvi modellek fő építőelemei a neuron-hálok, ezért nem lehet pontosan meghatározni, megkötni a modell viselkedését. Ugyanis a neuron-hálók működését teljes mértékben a tanító adatbázis határozza meg és a kész modell működése úgynevezett black box technológia, vagyis nem feltétlen értjük, hogy mit miért csinál és nem tudunk egyértelmű logikát leszűrni. Jelenleg fontos kutatási kérdés emiatt az, hogy hogyan lehet megkötni, kontrollálni egy már létező modell működését. Jól szemlélteti a probléma nagyságát, hogy mikor elérhetővé vált az OpenAI ChatGPT-je, elég könnyen ki lehetett játszani a prompt-tal megkötött viselkedését a modellnek. Ugyanis hiába volt meghatározva, hogy illegális, etikátlan kérdésekre ne válaszoljon, könnyen rávehető volt, hogy megosszon ilyen jellegű információt is. Mostanra már normalizálódott kissé a helyzet, részben a moderation API-nak köszönhetően, azonban továbbra sem lehetetlen kijátszani a modelleket.

Fontos kérdés lehet ugyanakkor, hogy mi történik, ha a bemeneti kód tartalmaz olyan kommenteket, melyek nem szándékosan ugyan, de prompt injectionként működnek. Vagyis, ha elkezdi követni a modell a komment utasításait. [21, 22]

Mi esetünkben a legjelentősebb kérdés a modell által visszaadott eredmény formátuma, hiszen számunkra csakis a kód fontos, semmilyen magyarázó szövegre nincs szükségünk. Emellett a magyarázó szövegek csökkentik a kód lehetséges méretét is. Utasításainkban (prompt) - a következő fejezetben látható módon – ugyan igyekeztünk megkötni, hogy ne adjon magyarázó szöveget és ne tegye kódblokkba (``` kód… ```) az eredmény kódot, ezt a modell a valószínűség alapú jellege miatt nem mindig tudta végrehajtani. Ha kódblokk kerül az eredménybe, abból ki kell nyerni a kódot. Azonban elképzelhető, hogy tartalmaz ilyen kódblokkot az eredeti kód, melyet Python kódok esetén dokumentálásnál alkalmaznak példa használat megadására. Hogy meg tudjuk különböztetni a kód részeként szereplő és a GPT által hibásan használt kódblokkokat, a Markdown elkódolására van szükség olyan szintaxissal, amely az adott nyelven nem fordul elő. Mivel mi csak Python nyelvű projektekre fókuszáltunk, ezért ez a helyettesítő szöveg jó lehet ’$$MD’, amelyet az alapján választottunk meg, hogy Pythonban nem használatos a $ karakter, ezáltal nehezen összetéveszthető nyelvi elemmel, az MD pedig a Markdown rövidítése.

Továbbá fontos megemlíteni, hogy az OpenAI API számos limitációval rendelkezik, melyet az éppen használt modell és a kérést (API request) indító szervezet csomagja határoznak meg. Ezen korlátozások határt szabnak naponkénti, percenkénti és kérésenkénti maximális tokenszámra. A bemeneti és kimeneti szöveget egy, vagy több karakteres egységekre bontják ezt nevezzük tokennek. Technikai részlet, hogy sajnos jelenleg nem rendelkezik az OpenAI API olyan hívási lehetőséggel, mellyel lekérdezhető lenne modellekre nézve bármely korlátozás. Emiatt manuális kigyűjtéssel tároltuk el konfigurációs fájlban a modellek adatait. Továbbá rendszeres aktualizálással tartottuk frissen.

Az olyan korlátozások esetén, melyek nem egy kérésre vonatkoznak, az exponenciálisan növekvő időközönkénti újra próbálás az ajánlott módszer, melyet mi is használtunk. A kérésenkénti korlátozás ennél azonban sokkal komplexebb probléma. Ezen limitáció valójában két korlátozásból áll: a teljes kontextus és a válasz méretére vonatkozó korlátozások. A kontextus tartalmazza az összes üzenetváltást a kliens és az API közt. Technikai szempontból ez egy tömb, amely hasító táblákat tartalmaz. Minden ilyen hasító tábla (dictionary) tartalmaz egy szerepkör (role) és egy tartalom (content) kulcs érték párt. Ennek kezdeti eleme egy úgynevezett rendszer szintű (system) prompt, amelyet arra használhatunk, hogy a modell viselkedését befolyásoljuk az általunk kívánt irányba. Munkánk során a következő fejezetben olvasható promptokat ebben a részben adtuk meg. Ezt követően olyan objektum párosok jönnek a tömbben, melyek egy-egy üzenetváltást valósítanak meg. A kontextus tokenszám korlátozása ezen tömb minden elemében vett összesített tokenszámra

vonatkozik. Míg a válasz limitációja minden egyes üzenetváltás esetén a válasz méretére vonatkozik.

Mi esetünkben a válasz méretére vonatkozó korlátozás a mérvadó, ugyanis a bemeneti és kimeneti információ mennyiség körülbelül azonos. Emiatt szükségszerű lehet a túl nagy méretű fájlok feldarabolására. Mielőtt kitérnénk ennek a megvalósítására, fontos megjegyezni, hogy mekkora méretre szeretnénk darabolni. Ugyanis, ha pontosan akkora részekre daraboljuk fel fájlunkat, amely maximális tokenszámot ér el, akkor nincs lehetősége a GPT modellnek a kód elrendezésén, tagolásán javítani, mivel ez a tokenszám növekedésével járna. Ebből kifolyólag egy küszöbértéket használtunk, amellyel megszoroztuk a kimeneti limitációt és ez lett a kód fregmensek maximális mérete. Végrehajtottunk egy kisebb méretű tesztet néhány példa fájlon, mely során ezt a küszöbértéket körülbelül 0,90-nek találtuk a legjobbnak. Ekkor tűnt minimálisnak az elvesztett kód mennyisége ugyanis, ha nagyobb értéket adtunk meg akkor rövidített a modell olyan kommentek kíséretében, mint például „a többi változatlan”, vagy „ez a metódus megtartotta eredeti állapotát”. Túl kicsi érték esetén viszont kérdéses mennyire hátráltatja a teljesítményt, hogy csak a teljes kód egy kis kontextusát látja a modell. Ezért munkánk során az imént említett 0,90-es szorzó értéket használtuk.

A bemeneti kód fregmentálása azonban bonyolult feladat, ugyanis nem lehet akárhol félbevágni a kódot, mert ha így adnánk be a modellnek, valószínűleg kijavítaná a szintaktikai hibát és ezáltal nem lehetne helyesen konkatenálni a részeket. Emellett kérdéses a teljesítményre mért hatása is a szintaktikailag hibás bemenetnek. Ezért célszerű az egyes kódblokkok, kódrészek (pl.: ciklus, értékadás, függvény, osztály) egyben hagyása és ezen komponensek közt vágni. Nem lehet azonban akárhol vágni anélkül, hogy jelentősen tovább bonyolítanánk a feladatot. Csak a globális hatókörben elérhető egységek között lehet bontani biztonságosan anélkül, hogy extra komplikációkat hoznánk. Az egységek behatárolására célszerű lehet absztrakt szintaxis fát (Abstract Syntax Tree, AST) használni, amely faként tárolja el a kód felépítését. Munkánkban mi is ezt a megközelítést alkalmaztuk, mely során a bemeneti kódot értelmeztük az AST modul segítségével és ezen fa alapján határoztuk meg a kódfregmenseket. Problémát jelent azonban ezen megközelítés esetén hogyha az eredeti kód szintaktikailag helytelen, ugyanis ekkor nem tudjuk AST-ként értelmezni. Emiatt a Python 2 kódok esetén nem tudunk feldarabolni. Mi esetünkben a kódhalmazban egyetlen, kis méretű fájl tartalmazott ilyen kódot, ezért ez nem jelentett problémát.

Nem egyértelmű azonban ezen a ponton, hogy hogyan határozzuk meg a fregmenseket. A legegyszerűbb megközelítés az lehet, hogy az elejétől kezdve mohó módon ellenőrizzük az eddigi globális részek méretét + a jelenlegi komponens méretét és ha túllépi a megadott határt, akkor vágunk. A vágás pontjáig vesszük az AST elemeit és visszaalakítjuk Python kóddá. Ez a megoldás működőképes, azonban teljesen elveszíti az eredeti kód tagolását és az összes komment is

megszűnik, ugyanis kommenteket nem tartalmazhat AST. Ezáltal torzulhatna a kiértékelés pontossága, ugyanis a kód tisztaságának egy nem elhanyagolható része a tagolás és a kommentek megfelelő használata. Ezen okokból kifolyólag jobb megoldásra volt szükségünk, munkánkban egy olyan hibrid megoldást választottunk, amely az AST könyvtár egy számunkra kedvező tulajdonságát használta ki. Ugyanis az egyes AST részfákból kinyerhető az adott fa kezdeti és végső sorszáma melyet az eredeti bementi kód alapján számoz. Ezáltal megvalósítható lett egy olyan hibrid rendszer melyben az AST-t csak az egyes részfák végének meghatározására használtuk, a tokenek számolását és a kódrészek vágását viszont az eredeti kódon végeztük. Ennek köszönhetően sikerült feldarabolnunk úgy a kódot fregmensekre, hogy megtartottuk a tagolást és a kommenteket is.

3.3 Prompt tervezés

A GPT API hívása esetén az előző szegmensben említett kontextusban meg kell adjunk egy rendszer szintű instrukciót (prompt), illetve egy felhasználó szintű bemenetet. Többféleképpen is lehetőség van utasítást kiadnunk: rendszer utasítás, felhasználói utasítás. Annak érdekében, hogy szigorúbban vegye az utasításunkat mi rendszer szintű üzenetként adtuk meg promptjainkat. Ezzel esetlegesen javítva az instrukció és a kód elkülönülését, mivel felhasználói üzenetként magát a kódot adtuk át nyers szövegként. Amennyiben több felhasználói üzenetet is küldtünk volna (több tisztítás egy kontextuson belül) és rendelkeztünk volna külön, csak az adott bementre szóló utasítással, akkor célszerű lehetett volna azt a felhasználói üzenet részeként küldeni, hogy ne legyen hatása a többi üzenetre. Viszont az előzőleg említett API limitációs okok miatt egy kontextussal csak egy fájlt tisztítottunk.

A megfelelő prompt választása igen fontos feladat, ugyanis a modell újra tanítása mellett ez az egyetlen effektív módja az eredmény javításának. Több promptot is megnéztünk, kezdetben terjengősebb megfogalmazást alkalmaztunk minden részpontot részletesen kifejtve. Effektivitását tekintve ugyanakkor úgy tapasztaltuk néhány manuális teszt során, hogy a GPT kezét ezáltal inkább csak megkötjük, mint sem segítjük a modellt a feladat megértésében. Például a kezdeti promptok esetén részletesebben kifejtettük mit jelent a clean code, milyen elveket tartalmaz, de a végső 4 promptban már ezt nem tettük.

Az általunk készített 4 végleges promptokat az alábbi stratégia alapján hoztuk létre. kezdetben nagyon általánosan, röviden arra utasítottuk a modellt, hogy tisztítsa meg a beadott kódokat és megadtuk az eredmény elvárt formátumát (csak kód). Ezt követően minden generációban bővítettük a promptot extra utasításokkal, ezáltal további lehetőségeket meghatározva a modell számára, amellyel javíthatja a kód minőségét. Ez alapján készítettük el az alábbi utasításokat (az előző

generációból megmaradt, változatlan szöveget **...** jelöli):

- **gen-0: „**

  Clean the codes given to you with clean code principles!

  RESPONSE must be PLAIN TEXT, with EVEN THE UNCHANGED PARTS OF THE CODE INCLUDED!

  ”

- **gen-1-spacing: „ …**

  Use spacing, empty lines if necessary for readability

  ”

- **gen-2-simplify: „ …**

  You can simplify the way functions work as long as the functionality and results stay the same as before.

  ”

- **gen-3-rename: „ …**

  You can rename elements that are not accessible to other files, thus preserving compatiblity with them.**”**

Összességében kezdetben *gen-0* promptunk egyszerűen kód minőség tisztítást hajt végre, a következő generációban explicit megmondjuk, hogy foglalkozzon a kód elrendezésével is, ezután megengedjük a végeredményt nem befolyásoló működésbeli egyszerűsítést és változtatást, végezetül megengedjük az egyes lokális elemek átnevezését, amennyiben az nem járhat a kompatibilitás elvesztésével.

Prompton kívüli befolyásoló tényező a temperature (hőmérséklet), amely lehetőséget ad a modell kreativitási szabadságának a beállítására. Ezáltal a modell kevésbé szigorúan veszi az utasításokat és elképzelhető, hogy ennek köszönhetően olyan helyzetet is meg tud oldani, amelyre nem adtunk utasítást. Egy nagyobb méretű fájlon, többszöri futtatással röviden megnéztük milyen hatással van a temperature szigorú és laza prompt esetén. Ehhez a prompt tervezés során létrehozott legszigorúbb utasításunkat vetettük össze a *gen-0* prompttal. A tesztet 0**,** 0,25**,** 0,5 és 1 értékeken néztük meg, 1-nél nagyobb érték esetén túlzottan kreatív lett a modell és hibás unicode karaktereket generált, ezáltal API hibát okozott és nem kaptunk eredményt. A Qodana és SonarQube statisztikái alapján azonban nem tapasztaltunk egyértelmű korrelációt a temperature értéke és a kód minősége közt. Minimálisan úgy tűnt kötött prompt esetén rontja a temperature növelése a minőséget, ezért végül a 0-t választottuk értékként.

3.4 Prompt dekoráció

Az imént említett promptok azonban nem kezelik le az olyan speciális eseteket, melyek az elő-feldolgozás során jelentkezhetnek. Az előfeldolgozásban ugyanis előfordulhat, hogy fregmentáljuk a kiindulási kódot, így nem árthat tudatni a modellel, hogy ez megtörtént, nehogy véletlen töröljön olyan kódrészeket, amelyeket nem használ a jelenlegi kódrész (például import). Emellett a Markdownokat (```) is átkódoljuk (alap esetben: $$MD), fontos lehet tudatni a modellel, hogy ezek miért vannak jelen a kódban, illetve amennyiben szeretné használni a kódban a kódblokk Markdownt, akkor ezt a kódolt verziót használja.

Ellenben nem racionális minden esetben a prompt részeként közölni ezen lehetséges elő-feldolgozási lépéseket. Mivel ezzel egyrészt csökkentjük a tokenekek számát sokszor feleslegesen, másrészt összezavarhatjuk a GPT-t ezáltal. Például, ha nem is tartalmaz kódolt Markdownt a bement. Emiatt ezen utasításokat csakis abban az esetben fűztük hozzá (dekoráltuk ezáltal a promptot), amennyiben előfordult ennek szükségességét kiváltó elő-feldolgozási lépés. Az elkészített dekorátor promptokat dinamikusan hoztuk létre, ezáltal a konfigurációnak megfelelően beállított Markdown és Markdown helyettesítő elemet tartalmazzák. Az alábbiakban a Markdown promptban $$MD helyére alap esetben ``` és $$MD_REPLACEMENT helyére $$MD helyettesül be. A split (feldarabolás) prompt a már említett módon működik.

A két dekorátor prompt:

- **markdown: „**

  The file contained code block markdowns: '$$MD' which were replaced by: '$$MD_REPLACEMENT' to not mess with the output. If you want to use: '$$MD' make sure to use: '$$MD_REPLACEMENT' instead!

  ”

- **split: „**

  This is only a fragment of the original file, make sure you do not remove/rename anything in the global scope to maintain compatibility!

  ”

3.5 Tisztítási procedúra

Az előfeldolgozás, prompt dekoráció, kód tisztítás, utó-feldolgozás és a kiértékelés automatizálható lépéseit automatizáltuk konzolos alkalmazások segítségével. Készítettünk egy fő alkalmazást, amely a megadott elérési útvonalon található fájlokat megtisztítja és kiírja az eredményt egy opcionálisan választható könyvtárba. Parancssori argumentummal

paraméterezhetővé tettük, ezáltal megadható többek közt, hogy: átmásoljuk-e a nem kódként érzékelt fájlokat, hol található a promptokat tartalmazó konfigurációs fájl. Ennek köszönhetően rugalmasan használható az alkalmazás. Készítettünk ezentúl egy olyan segédalkalmazást, amely a főalkalmazást futtatja egy instrukció halmazra, ezáltal automatizálva a promptok futtatását.

Az alkalmazás futás idejét tekintve kritikusan lassító lépések: az API hívása és válaszra várakozás, fájlok írása olvasása. Annak érdekében, hogy jelentősen optimalizáljuk a futásidőt, párhuzamosítottunk minden olyan folyamatot, amely esetén racionális az aszinkronitás használata. Köszönhetően az OpenAI aszinkron kliensének, az API hívása is párhuzamosítható. A megtartott fájlok másolása különösen lassú folyamat lehet, ugyanis ezek lehetnek akár nagy méretű adatbázisok, adatfájlok is. Ezen fájlok megtartására kifejezetten gyakorlati használat esetén lehet szükség, illetve amennyiben tesztelni szeretnénk az alkalmazás szemantikai helyességét a tisztítás után.

3.6 Utófeldolgozás

Az utófeldolgozási lépésben kezeljük az említett hibás GPT kimeneti formátumot és az elkódolt kód blokk Markdownok dekódolását. Amint a korábbi fejezetekben említettük, a nagy nyelvi modellek kimenetének pontos megkötése nem lehetséges, emiatt előfordulhat, hogy az utasításaink ellenére a GPT kimenete tartalmaz nem kód elemeket is. Ez tipikusan magyarázó szöveget jelent és ez esetben a kódot magát ``` kód blokkokba helyezi. Rosszabb esetben elfogyhat a kimeneti tokenek száma, ilyenkor félbevágódik az output és amennyiben használt kezdő kódblokkot, akkor nem tudja lezárni a token limitáció miatt. Ez az eset kezelhető azáltal, hogy ha előfordul kódblokk Markdown a kimenetben, akkor mohó módon ameddig kettő vagy több Markdown fordul elő, addig a két Markdown közti sorokat hozzáfűzzük a kimenethez. Amikor már csak 1 kódblokk van, akkor a blokk utáni részt fűzzük a kimenethez. Ezen módszer hibás lehet olyan esetben, ha a GPT kimenete tartalmaz felesleges, nem kód részeket jelölő Markdownokat, illetve, ha a Markdownok elhelyezkedése nem a fenti logika alapján történik: például csak egy Markdown van, de az a kód végén. Katasztrofális esethez vezet ugyanakkor, ha nem tartalmaz kódot, csak utasításokat a kód javításához, vagy szöveges üzenetként, hogy nem változtatott a kódon. Ekkor a kimeneti kódhoz hozzáfűződik ugyanis egy nyelvileg nem értelmezhető szöveg.

Ezután a bemenetben elkódolt és a GPT által helyesen használt Markdown kódjeleket dekódoljuk, hogy az eredeti formájukban legyenek jelen és kiírjuk az eredményt a célfájlba.

3.7 Kiértékelés

Dedikált clean code kiértékelő eszköz hiányában olyan eszközöket kerestünk, amelyek képesek olyan metrikák mérésére, melyek tükrözheti a kód tisztaságának mértékét. Két alkalmasnak tűnő statikus elemző alkalmazást találtunk: JetBrains Qodana és SonarSource SonarQube. Kezdetben lokálisan futtatható változatukat próbáltuk ki azzal a céllal, hogy automatikusan kigyűjtsük adatfájlba a mérések adatait, azonban a kimeneti fájlok struktúrája és az almappákra vonatkozó statisztikai adatok hiánya körülményesebbé tette ezt a megoldást, mintha az elemzők felhő alapú szolgáltatásait használnánk és manuálisan kigyűjtenénk az adatokat. Ezért végül a felhő alapú szolgáltatásokat használtuk. Az ellenőrzést automatizáltuk GitHub események (GitHub Actions) segítségével, ezáltal amint commitoltunk változtatást, az elemzők a változtatással rendelkező fájlokat újraértékelték. Később ezen mérésekből nyertük ki a számunkra érdekes adatokat.

A két elemző jelentősen eltérő metrikákat mér. A Qodana jobban hasonlít a tradicionális értelemben vett statikus kódelemzőkre, mivel főként csak PEP8 konvenciók sértését és olyan hibák jelenlétét képes detektálni, melyeket a fejlettebb fejlesztői környezetek is képesek. Figyelmeztetéseit közepes (moderate) és magas (high) súlyossági kategóriába csoportosítja. Ezzel szemben a SonarQube képes fenntarthatósági és komplexitás mérésekkel. A fenntarthatóság és könnyen értelmezhetőség clean code alapelvek, ezért a SonarQube alkalmasabbnak tűnt kódok tisztaságának mérésére. Metrikákat tekintve a fenntarthatóság kategóriában rendelkezik bad smell (code smell) méréssel, amely a kódban előforduló olyan részek számát jelenti, amelyek nehezen érthetőek, esetleg strukturális vagy vezérlési hiányossággal rendelkeznek. Ezáltal nehezítve a kód karbantartását. Emellett képes mérni technical debt-et, amely a bad smellek javításának az időigénye. Ez alapján megtudható a spórolt idő mennyisége azáltal, hogy automatikusan tisztítottuk fájljainkat. A komplexitást tekintve cyclomatic és cognitive complexity-t képes mérni, ebből mi a cognitive complexity-t használtuk, ugyanis ezen metrika a kódvezérlés áramlásának értelmezési nehézségét méri.

Készítettünk ezentúl saját konzolos alkalmazást, amellyel egyszerű statisztikai adatokat mértünk. Ennek segítségével mértük meg az egyes mappák esetén a tokenek, kódsorok, fájlok és szintaktikai hibát tartalmazó fájlok számát. A szintaktikai hibákat AST segítségével detektáltuk.

4. fejezet

Eredmények

A 3.1-es fejezetben említett kód adatbázison futtattuk a GPT 3.5-Turbo és 4-Turbo modelleket. Eredményt tekintve a 3.5-Turbo kevésbé tudta követni a kimenetre vonatkozó megkötéseket, ezért számos fájl esetén magyarázó szöveg, szintaktikai hiba került az eredmény fájlokba a modell hibájából. Ezen hibák pontatlanságot okozhatnak a mérések tekintetében, de úgy döntöttünk nem szűrjük ki a hibás fájlokat a statisztikában, mivel ez is a 2 modell képességei közti különbséget mutatja.

4.1 Hibás eredmények

Bármely nagy nyelvi modell esetén fennáll annak az esélye, hogy nem a kért működést mutatja, hiszen belefuthatunk olyan esetbe, amelynél nem tud a modell jó eredményt adni. Ekkor előfordulhatnak akár nem várt hibák is. Munkánk során azt tapasztaltuk, hogy a GPT-3.5-Turbo gyengébben teljesített az utasítások követésében és a megfelelő végrehajtásban. Ez nem meglepő, mivel a GPT-4-Turbo egy sokkal komplexebb modell, amely esetén jelentős fejlődést értek el az előbbi verzióhoz képest. A kódblokkok előfordulási gyakoriságáról nem tudunk ugyan statisztikát mutatni, mivel azt a rendszerünk automatikusan kezelte, hasonlóan az ezek körül előforduló magyarázó szövegek esetén sem. Nagyobb probléma volt, amikor a modell az utasításunk kérése ellenére nem adta vissza a teljes kódot változatlanul (például III. kódrészlet), amennyiben nem tudta ennél tisztábbá tenni, hanem szöveges formában közölte, hogy változatlan maradt a kód. Ezen probléma egy enyhébb előfordulása, amikor ezt kódon belül teszi meg például kommentekben (IV. kódrészlet). Habár ekkor nem feltétlen eredményez szintaktikai hibát a fájl, de teljesen automatikus felhasználás esetén elromlik a program. Enyhébb probléma, ugyanis ekkor a feladatot képes volt megoldani, csak a kimenet volt rossz. Ezen hibák torzítják a statisztikai adatokat, mivel az eredeti bad smellek eltűnnek a fájlból, de ugyanakkor a kódelemzők hibát jeleznek a természetes nyelv előfordulására, Qodana esetén például minden egyes szóra. Ezáltal negatív és pozitív irányban is torzulhatnak az adatok. Emellett olykor hibát vétett a dokumentációs információt ellátó """ konstans

szöveg objektumok esetén. Ezen szöveges objektumok ugyanis komplikált feladatot mutathatnak egy modell számára, mivel több sorral később záródnak csak le sok esetben. Ebbe több alkalommal is belezavarodott a GPT-3.5-Turbo (például: I. kódrészlet) és a záró tag párt véletlen a kimenet legvégére tette, vagy teljesen elhagyta a tag párt, ezáltal hatalmas mennyiségű szintaktikai hibát eredményezve.
Ugyanakkor olyan egyszerű hibákba is beleesett, mint a hibás zárójelezés (II. kódrészlet).

A hibás kimenet detektálása nehéz feladat a teszt projektek méretének köszönhetően, ugyanakkor jó közelítése lehet ennek azon fájlok száma, amelyek szintaktikai hibát tartalmaztak. Hibás működés és szöveges kimenet esetén ez egyértelműen hibát okoz és sok esetben akkor is, ha kommentek segítségével rövidíti a kimenetet, mivel a komment nem utasítás ezért függvényen és egyéb blokkon belül, ha csak egy komment található az szintaktikailag helytelen. Tehát a szintaktikai hibás fájlok számával becsülhetjük ezen esetek előfordulását, amely azt mutatja, hogy GPT-3.5-Turbo esetén ~17% esély van rá, hogy egy fájl hibás legyen, míg GPT-4-Turbo esetén egyetlen ilyen eset sem volt. Mivel ez csak becslés ezért ez nem jelent 100% hibamentességet, ugyanis előfordulhat olyan bement, amely esetén a GPT-4-Turbo is hibás eredményt ad.

Ezen hibákon túl a funkcionalitás megtartása is kérdéses, amely a most említett problémáknál jelentős mértékben bonyolultabb, főként a grafikus felhasználó felülettel (GUI) rendelkező alkalmazások esetén. A funkcionalitás automatikus tesztelésével nem foglalkoztunk.

4.2 Szintakai hibás kódrészlet példák

I. kódrészlet: *Az alábbi kódrészlet jól szemlélteti a modell összezavarodását többsoros kommentek esetén, ebben az esetben mind a kezdő, mind a záró tag elmaradt.*

```
Reference:
OpenCV-Python Tutorials - Camera Calibration and 3D Reconstruction
http://opencv-python-
tutroals.readthedocs.org/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

import cv2
import numpy as np
import glob
```

II. kódrészlet: *Az alábbi esetben a zárójelek megfelelő lezárása nem sikerült.*

```
client_socket.connect(('192.168.1.100', 8000)
```

III. kódrészlet: *A következő esetben csak instrukciókat adott a modell, nem kódot.*

```
The code provided is well-structured and follows many clean code principles. However,
there are a few areas where improvements can be made:

1. **Function and Variable Names**: Ensure that function and variable names are
descriptive and follow a consistent naming convention. For example, `create_model` can
be renamed to `initialize_or_load_model`.
. . .

By applying these principles, the code can be further improved in terms of readability,
maintainability, and extensibility.
```

IV. kódrészlet: *Ebben az esetben kommentekkel jelezte, hogy változatlan az adott rész.*

```
def project_and_apply_input_bias(logits, output_projection, input_bias):
    # Project and apply input bias logic

def apply_input_bias_and_extract_argmax_fn_factory(input_bias):
    """Factory function for loop function."""
    # Factory function logic
```

4.3 Meglátások

Felmerült általunk, hogy vajon mennyi kód veszett el a modell hibájából adódóan. Erre részben választ ad az I. és II. diagram, melyek a tokenek és a kódsorok számának változását mutatják GPT-4-Turbo és GPT-3.5-Turbo esetén. Ebből az ábrából adódóan azt mondhatjuk, hogy a prompt választása jelentősen befolyásolja a kimeneti méretet, ugyanis a nagyságrendbeli eltérésektől eltekintve az arányok hasonlóak mindkét modell esetén. Ugyanakkor jelentős mértékben nagyobb méret csökkenés figyelhető meg GPT-3.5-Turbo esetén. Emellett GPT-4-Turbo esetén a méretbeli különbség a tisztítási eredmények esetén sokkal kevésbé ingadozó, sokkal inkább konstans csökkenés történt az eredeti állapothoz képest. Ezt a csökkenést többek közt az is okozhatja, hogy tartalmaz az egyik bemeneti projekt kikommentelt kódokat, ezeket ugyanis minden esetben törölte a modell. Fontos megjegyezni, hogy a méret csökkenés nem feltétlen rossz vagy jó, de elég jó előjele lehet annak, ha valamilyen probléma adódott. Megfigyelhető a X. diagram alapján, hogy amikor a legkisebb volt a hibás fájlok aránya GPT-3.5 esetén (*gen-1-spacing*), akkor volt a legnagyobb méretű a kimenet (leszámítva a kezdeti állapotot).

A Qodana által jelzett hibák esetén külön mértük végül a statisztikát, ugyanis amikor természetes szöveg kerül a kódba, akkor ezt egy magas szintű (high severity), statement has no effect (a kifejezésnek nincs hatása) nevű figyelmeztetéssel jelzi a Qodana. Méghozzá minden egyes

szóra, emiatt ahogyan a IX. diagramon is látszik, jelentősen megugrott a magas szintű figyelmeztetések száma. Ugyanakkor ahogy a VIII. diagram is mutatja, a közepes szintű figyelmeztetések száma mind GPT-3.5-Turbo, mind GPT-4-Turbo esetén csökkent. GPT-4-Turbo esetén már *gen-0* esetén is körülbelül felére csökkent a jelzések száma és ez a későbbi promptok esetén csak javult. A magas szintű figyelmeztetések száma GPT-4-Turbo esetén közel felére is képes volt csökkenni, legrosszabb esetben is 69 helyett csak 60 probléma maradt. A GPT-4-Turbo ezen adatok szerint a Qodana által jelzett hibákat közel felére képes csökkenteni, míg a GPT-3.5-Turbo a közepes szintű hibák számát képes csökkenteni, de nagy mennyiségű magas szintű hibát hoz létre ezáltal.

A GPT-3.5-Turbo ahogy a X. diagramon is látható, viszonylag konstans eséllyel hibázik (~17%), GPT-4-Turbo esetén viszont nem keletkezett egyszer sem szintaktikai hibás fájl, ezért arra következtethetünk, hogy a prompt választás nem nagy mértékben befolyásoló tényező a hibás működés által keletkező hibák létrejöttében.

A SonarQube statisztikái tekintve azt mondhatjuk, hogy a IV. diagramon látható code smellek és a XI. diagramon látható technical debt tendenciája közel azonos, mely nem meglepő hiszen a technical debt, a code smellek számából számított érték. GPT-3.5-Turbo esetén elmondható, hogy már *gen-0* esetén közel megfeleződik az mindkét statisztika értéke és ez később tovább javul, technical debt esetén ez *gen-2-simplify* esetén közel harmadolódik.
GPT-4-Turbo esetén két érték azonban jobban eltér egymáshoz viszonyítva, mint a másik modell esetén, ugyanis a technical debt már *gen-0* esetén harmadára csökken és ez később kevesebb, mint hatodára is csökkent. Ezzel szemben a code smellek száma *gen-0* esetén megfeleződött és a többi prompt esetén ekörül stagnált. Összességében azt mondhatjuk, hogy a GPT-4-Turbo akár hatodára, míg a GPT-3.5-Turbo körülbelül harmadára képes csökkenteni a technical debt-et. A code smellek száma pedig mindkét modell esetén legalább a felére csökkenthető.

Az elkövetkezendőkben említünk olyan statisztikákat is, amelyek 100 tokenre vetítenek le adott értéket. Ez alatt azt értjük, hogy 100 token esetén átlagosan hányszor fordul elő az adott mért mennyiség. Azért tokenekre számoltuk és nem kódsorra, mivel egy sor kód általában nagyon hosszú (van kód), vagy csak white space karakterekből áll. Emiatt a tokenek pontosabb értéket adhatnak, ha egy adott egységre vonatkozóan szeretnénk mérni statisztikát.

Megnéztük emellett a code smellek számát 100 tokenre levetítve, melyet a V. diagram szemléltet. Ebből leolvasható, hogy a GPT-4-Turbo képes volt kevesebb, mint felére csökkenteni ezt az értéket ezáltal legyőzve a GPT-3.5-Turbo-t, amely, habár nagyobb kezdeti értékkel rendelkezett és képes volt harmadára is csökkenteni ezt az értéket, de nem sikerült jobbat elérnie a GPT-4-Turbonál még úgy sem, hogy jelentős részleteket kihagyott az eredeti kódból.

A cognitive complexity-t és ennek a 100 tokenre levetített értékét ábrázoló diagramok (VI. és

VII. diagram) megmutatják, hogy a komplexitás csökken mindkét modell esetén, ugyanakkor GPT-4-Turbo esetén viszonylag kis mértékben (~25%), amely azt is mutathatja, hogy a komplexitás csak kis mértékben csökkenthető, a nagy méretű kód továbbra is nehezebben értelmezhető. GPT-3.5-Turbo esetén ugyanakkor hatalmas mértékű változást láthatunk, amely valószínűleg a sok kihagyott kódrész miatt van. Jól mutatja ezt a VII. diagram, amelyen az látható, hogy a GPT-4-Turbo esetén hiába volt összességében javulás, az egyes részek komplexitása gyakorlatilag konstans. Míg GPT-3.5-Turbo esetén az egyes részekre vonatkozó komplexitás drasztikusan csökken.

5. fejezet

Diszkusszió

Ebben a fejezetben a jövőben választható tovább haladási irányokat tárgyaljuk a jelenlegi helyzetből. Megemlítünk hiányosságokat és lehetőségeket, amelyek munkánkat körül veszik.

5.1 Szintaktikai hibás bemenet

Módszertanunk szignifikánsan alapszik a bemeneti fájlok helyességén, ugyanis amennyiben helytelen a bemeneti fájl, nem tudjuk fregmensekre (kód részekre) darabolni. Amely nagy méretű fájlok esetén azt jelenti, hogy nem tudunk értelmes eredményt elérni. További kérdés még, a legacy (elavult) kódok kezelése. Python 2 kódok esetén például az AST könyvtár által használt Python 3 értelmező nem képes érvényes szintaktikaként kezelni a régi verziójú kódot. Ezen esetekben érdemes lehetne detektálni a hiba jelenlétét és tájékoztatni a felhasználót. A felhasználó ez esetben megoldhatná a kérdéses hibákat, elavult kód esetén például egy új verzióra konvertáló program (például: 2to3) segítségével. Ezáltal nem futnánk problémás esetekbe ezen szempontokból.

5.2 Megbízhatóság tesztelés

Mivel a nagy nyelvi modellek valószínűségi alapokon adnak kimenetet, ezért fontos kérdés, hogy mennyire megbízhatóan képesek az adott eredmény az adott bemenettel hozni. Erre célszerű teszt lehet a tisztítási procedúra többszöri alkalmazása a bemeneti adatokon. Mi esetünkben ilyen nem történt, viszont a bemeneti kódhalmaz méretével próbáltuk növelni a mintaelemszámot.

Tesztelni kellene ugyanakkor a hibás kimenetek esélyét mind szintaktikai mind szemantikai szempontból. Míg mi esetünkben a GPT-3.5-Turbo gyakran és a GPT-4-Turbo nem vétett szintaktikai hibát, azonban ez nem zárja ki annak lehetőségét, hogy megtörténjen. A funkcionalitás megtartását ellenben semennyire sem teszteltük, mivel munkánk arra fókuszált, hogy képesek-e ezek a modellek a kód minőség mitigálására, azonban igen fontos kérdés a helyes működés gyakorlati felhasználás esetén. Ezért a jövőben unit tesztekkel szeretnénk ezt ellenőrizni. Emellett

10 véletlenszerűen választott eredmény kód esetén végeztünk manuális összehasonlítást, mely során úgy tűnt, hogy a funkcionalitást képes megtartani.

5.3 Optimalizációs lehetőségek

Habár kezdetben több promptot is használtunk, tesztelni a nagy kódhalmazon mérvadó mértékben csak a 4 fő promptunkat teszteltük. Elképzelhető, hogy jelentős teljesítmény emelkedés elérhető eltérő utasítás megválasztásával.

Érdemes megemlíteni az input tömörítést, amely segítségével csökkenthető lenne a tisztítási költség, ezáltal lehetőség lenne nagyobb méretű teszthalmazon való futtatásra. Ezzel azonban munkánk során nem foglalkoztunk.

A GPT modellek esetén lehetőség van finomhangolásra (fine tune), amely segítségével a már meglévő általános célú modelleket saját tanító adatainkkal hangolhatjuk az általunk megoldani kívánt feladatra. A finomhangolás azért is jelenthet fejlődést, mivel ekkor képesek lehetünk a modellt megtanítani az általunk kívánt feladatra. Ezáltal pontosan azt az utasítást kapná, amelyen tanítva lett és így kevésbé állna fenn a félreértelmezés lehetősége. Tehát, ha létrehoznánk egy nagyobb méretű adatbázist, amely tartalmazna refaktorálás előtti kódokat és ezek clean code szakértők által megtisztított változatait, akkor ezt felhasználva finomhangolásra több szempontból is fejlődést érhetnénk el. Egyrészt növelhetnénk a modell effektivitását a tisztításban. Másrészt a hibás kimenetek valószínűségét csökkenthetnénk, ezáltal akár GPT-3.5-Turbo esetén is elérhető lenne a megfelelő kimeneti formátum. Kikerülhető lenne például a rövidítés azáltal, hogy tartalmazna adatbázisunk tiszta kód párosokat, amelyek esetén nincs szükség változtatásra, de mégis elvárnánk, hogy ugyanazt, amit adtunk neki, visszaadja. Illetve a kimenet formátumát (kód nyers szövegként) is jobban megköthetnénk.

Eltérő nagy nyelvi modell használata is jelentős javulást eredményezhet, kifejezetten akár olyan modell használata, amely specifikusan kódolásra lett tanítva. Egyre több nagy nyelvi modell elérhető, így a lehetőségek egyre csak nőnek.

Végezetül javíthatnánk méréseink pontosságán amennyiben létrehoznánk egy olyan mesterséges intelligencia alapú kódértékelő programot, amely kifejezetten clean code elvek alapján pontozna. Ezáltal a komplexebb, egyszerű elemzés által nem eldönthető elvek alapján is értékelhetnénk az eredményt, ezzel pontosabb képet kapva az általunk feltett kérdésekre.

6. fejezet

Validitást fenyegető tényezők

Eredményeink validitását (helyessége) több tényező is fenyegetheti. Kezdve a kis tesztbázis mérettel és mintaelemszámmal. Ugyanis elképzelhető, hogy ennél jelentősen nagyobb méretű teszt esetén nagy mértékben eltérő eredményt kapnánk. Elképzelhető, hogy a kódbázis, amelyet használtunk pont olyan elemeket tartalmaz, amelyek esetén átlagon felül teljesítenek a GPT modellek, ugyanígy az is elképzelhető, hogy a valós érték alatti teljesítményt mértünk. Ezen okokból szükségszerű lehet nagyobb méretű projekteken, többszöri ismétléssel elvégezni a tisztítási procedúrát és az alapján vonni le konklúziót. Munkánk során annak érdekében, hogy csökkentsük a kis mintahalmazból adódó hibákat, több projektet használtunk, melyek mérete is különböző nagyságrendű. Ezáltal többszínű kódhalmazunk lett.

A modellek által vétett szintaktikai és szemantikai hibák elképzelhető, hogy eltorzítják a valós eredményeket. Megeshet, hogy valójában ritkán vét szintaktikai hibát a modell és hogy rosszul kezeltük le azon eseteket, amikor szövegesen adják meg, hogy nem változott semmi. Ezáltal jelentősen rontva a valós teljesítmény adatokon.

Problémát jelenthet az is, ha a dekorátor promptokat rosszul választottuk meg, emiatt elrontva az egyébként jó teljesítményt.

Felmerülhet a lehetősége adatfelviteli hibáknak is a manuálisan felvitt adatok miatt. Nemmellesleg az általunk használt kódelemzők is lehet, hogy hibásan működnek bizonyos esetekben. Főleg, hogy az eszközök fejlesztésének még relatív kezdeti szakaszában járnak a cégek. Az is kérdéses, hogy mennyire jól tükrözik a tiszta kód elveket az elemzők adatai. Ettől függetlenül ezek a kódelemzők akadémiai és ipari környezetben is széles körben használtak és folyamatosan fejlesztik őket, emiatt megbíztunk abban, hogy nem tévednek. Számtalan olyan clean code tulajdonság van ugyanis, amelyet nehéz mérni és esetekben szubjektivitás kérdése, hogy valaki szerint áttekinthető vagy sem.

Végezetül az általunk írt alkalmazások, mint minden összetett alkalmazás valószínűleg tartalmaznak kisebb-nagyobb kijavítatlan hibákat, amelyek rosszabb esetben jelenthetnek olyan hibát is, amely befolyásolhatja eredményeinket. Ezt alapos kód reviewval igyekeztünk elkerülni.

7. fejezet

Konklúzió

Összességében munkánk megmutatta a GPT-3.5-Turbo és GPT-4-Turbo értékét az automatikus kód minőség javításában clean code elvek segítségével, a gyakorlatban előforduló Python projekteken. Mivel nem készült még kutatás melyben nagy nyelvi modelleket használtak kód minőség javítására, ezért számos problémát és kérdést elsőként kellett megoldanunk. Eredményeink fontos statisztikai adatokat tartalmaznak, melyek segíthetnek eldönteni, hogy alkalmasak-e ezen modellek a fejlesztőket segíteni, a kód minőségének karbantartásában.

Legfőbb megállapításaink:

1. A GPT-3.5-Turbo habár képes a technical debt mértékét 33%-ra és a code smellek számát 44%-ra csökkenteni, de mivel nem képes kellő precizitással követni az utasításokat, ezért ~17% eséllyel szintaktikai hibás kimenetet ad. Emiatt nem tűnik alkalmasnak teljesen automatizált rendszer alapjaként, de felülvizsgált körülmények még így is jelentős segítséggel lehet a fejlesztők számára.
2. A GPT-4-Turbo ezzel szemben képes a technical debt mértékét 1/6-ra csökkenteni és a code smellek számát 38,7%-ra csökkenteni, mindemellett egyszer sem tapasztaltunk szintaktikailag hibás kimenetet. Emiatt ez a modell alkalmasnak bizonyult akár teljesen automatizálva működni is, ám mielőtt ezt megtehetnénk, még szükségszerű a kimeneti funkcionalitás megtartásának a részletes tesztelés.
3. A nagy nyelvi modellek ezáltal kiemelten alkalmasnak bizonyulnak a kód minőség mitigálására.

Tehát a GPT modellek jelentős előrehaladást jelenthetnek az automatikus kódminőség javítás előrehaladásában, munkánk átlageredményt számítva GPT-3.5-Turbo és GPT-4-Turbo esetén összesen több mint 8, legjobb eredményt nézve 12 napot igénylő hiba javítást megoldottunk. Azonban további kutatás és fejlesztés szükséges a teljesítmény optimalizálására, a korlátozások felfedésére. Munkánk alapjául szolgálhat későbbi automatizált kódminőség javító kutatásoknak.

Függelék

Táblázatok

Az alábbi (V.) táblázat a GPT-3.5-Turbo, az azt követő (VI.) táblázat a GPT-4-Turbo futtatása esetén kapott adatokat tartalmazza, melyeket úgy formáztunk, hogy ha az eredeti állapothoz képest javulást mutatható ki, akkor zöld háttérszínt használtunk. Ha romlást, akkor narancsszínt használtunk. Ha nem történt változás akkor szürke színű háttért használtunk.

Összesített adatok - GPT-3.5-turbo-0125 (V. Táblázat)														
Prompt	Projektnév	Alap				Qodana			SonarQube				Kalkulált	
		Tokenek száma	Kódsor	Fájlok száma	Syntax Hibás fájlok	Moderate	High	Összesen	Code Smell	Debt	Cyclomatic Complexity	Cognitive Complexity	Cognitive C. / 100 LoC	CS / 100 LoC
input	AutoRCCar	7623	1032	13	1	16	4	20	22	2,32 h	107	160	2,10	0,29
	deep-text-corrector	21515	2354	8	0	40	1	41	23	3,72 h	234	285	1,32	0,11
	gpt-engineer	77827	11271	82	0	130	12	142	81	28,00 h	694	578	0,74	0,10
	graph_nets	120355	12337	24	0	84	10	94	42	4,58 h	1072	821	0,68	0,03
	principle_violation	1067	132	1	0	11	0	11	9	1,17 h	24	44	4,12	0,84
	social_mapper	27656	2872	10	0	226	48	274	214	75,00 h	457	1161	4,20	0,77
	Video-Pre-Training	36768	4165	22	0	132	16	148	83	4,82 h	538	462	1,26	0,23
	you-get	148635	14978	138	0	632	61	693	745	240,00 h	2342	3309	2,23	0,50
gen-0	AutoRCCar	6974	916	13	4	20	8	28	17	2,00 h	87	134	1,92	0,24
	deep-text-corrector	9878	1083	8	3	38	109	147	2	0,33 h	49	43	0,44	0,02
	gpt-engineer	57339	8177	82	23	117	711	828	62	26,00 h	407	285	0,50	0,11
	graph_nets	112760	11459	24	10	87	139	226	14	1,43 h	530	331	0,29	0,01
	principle_violation	911	116	1	0	6	1	7	2	0,48 h	24	44	4,83	0,22
	social_mapper	20688	2143	10	0	132	40	172	99	31,00 h	395	899	4,35	0,48
	Video-Pre-Training	34411	3883	22	2	101	105	206	29	2,35 h	400	330	0,96	0,08
	you-get	116554	12141	138	9	628	155	783	436	144,00 h	1767	2348	2,01	0,37
gen-1-spacing	AutoRCCar	7225	967	13	3	16	4	20	19	2,07 h	99	151	2,09	0,26
	deep-text-corrector	21026	2311	8	0	40	1	41	23	3,72 h	234	285	1,36	0,11
	gpt-engineer	66211	9648	82	24	141	459	600	59	26,00 h	529	454	0,69	0,09
	graph_nets	118031	12102	24	7	88	22	110	15	1,70 h	584	413	0,35	0,01
	principle_violation	941	126	1	0	9	0	9	5	0,98 h	24	44	4,68	0,53
	social_mapper	22152	2667	10	2	172	53	225	33	3,03 h	83	161	0,73	0,15
	Video-Pre-Training	34499	4015	22	3	104	105	209	29	2,35 h	382	314	0,91	0,08
	you-get	133515	14051	138	9	607	204	811	559	151,00 h	1669	2057	1,54	0,42
gen-2-simplify	AutoRCCar	6988	940	13	0	33	2	35	21	2,18 h	114	159	2,28	0,30
	deep-text-corrector	16206	1834	8	1	41	15	56	10	1,45 h	134	159	0,98	0,06
	gpt-engineer	63182	8983	82	28	139	1479	1618	46	7,78 h	402	279	0,44	0,07
	graph_nets	107353	10883	24	6	92	13	105	34	3,88 h	709	501	0,47	0,03
	principle_violation	846	98	1	0	9	0	9	2	0,38 h	20	33	3,90	0,24
	social_mapper	19163	2188	10	1	109	30	139	25	2,52 h	100	178	0,93	0,13
	Video-Pre-Training	31070	3683	22	2	100	40	140	31	3,02 h	372	331	1,07	0,10
	you-get	125137	13092	138	13	583	87	670	367	102,00 h	1430	1811	1,45	0,29
gen-3-rename	AutoRCCar	7075	956	13	3	27	2	29	16	2,02 h	94	137	1,94	0,23
	deep-text-corrector	17844	2005	8	1	37	6	43	10	1,45 h	134	159	0,89	0,06
	gpt-engineer	61881	8874	82	29	133	148	281	42	7,42 h	471	381	0,62	0,07
	graph_nets	107388	11024	24	4	91	10	101	21	2,45 h	756	524	0,49	0,02
	principle_violation	919	118	1	0	6	0	6	2	0,48 h	23	39	4,24	0,22
	social_mapper	18627	2000	10	1	107	75	182	23	2,35 h	101	178	0,96	0,12
	Video-Pre-Training	30713	3591	22	3	100	109	209	27	2,27 h	380	309	1,01	0,09
	you-get	124019	12814	138	11	557	62	619	393	125,00 h	1670	2254	1,82	0,32

V. Táblázat: *Összesített statisztikák GPT-3.5-Turbo*

Összesített adatok - GPT-4-turbo-2024-04-09 (VI. Táblázat)														
Prompt	Projektnév	Alap				Qodana			SonarQube				Kalkulált	
		Tokenek száma	Kódsor	Fájlok száma	Syntax Hibás fájlok	Moderate	High	Összesen	Code Smell	Debt	Cyclomatic Complexity	Cognitive Complexity	Cognitive C. / 100 LoC	CS / 100 LoC
input	AutoRCCar	7623	1032	13	1	16	4	20	22	2,32 h	107	160	2,10	0,29
	deep-text-corrector	21515	2354	8	0	40	1	41	23	3,72 h	234	285	1,32	0,11
	principle_violation	1067	132	1	0	11	0	11	9	1,17 h	24	44	4,12	0,84
	social_mapper	27656	2872	10	0	226	48	274	214	75,00 h	457	1161	4,20	0,77
	Video-Pre-Training	36768	4165	22	0	132	16	148	83	4,82 h	538	462	1,26	0,23
gen-0	AutoRCCar	7101	905	13	0	25	3	28	16	1,77 h	114	157	2,21	0,23
	deep-text-corrector	14442	1488	8	0	21	1	22	12	1,80 h	188	222	1,54	0,08
	principle_violation	725	85	1	0	6	0	6	2	0,37 h	18	32	4,41	0,28
	social_mapper	15723	1717	10	0	80	21	101	47	25,00 h	308	615	3,91	0,30
	Video-Pre-Training	30088	3461	22	0	126	16	142	79	4,23 h	508	403	1,34	0,26
gen-1-spacing	AutoRCCar	7004	936	13	0	22	4	26	17	2,00 h	107	160	2,28	0,24
	deep-text-corrector	16100	1674	8	0	26	1	27	15	2,18 h	212	252	1,57	0,09
	principle_violation	738	86	1	0	6	0	6	2	0,37 h	18	32	4,34	0,27
	social_mapper	16083	1776	10	0	67	39	106	58	24,00 h	306	585	3,64	0,36
	Video-Pre-Training	32521	3757	22	0	129	16	145	81	4,55 h	530	447	1,37	0,25
gen-2-simplify	AutoRCCar	6975	940	13	0	24	4	28	16	1,92 h	107	160	2,29	0,23
	deep-text-corrector	13769	1405	8	0	20	1	21	14	2,82 h	193	233	1,69	0,10
	principle_violation	726	85	1	0	6	0	6	2	0,37 h	18	32	4,41	0,28
	social_mapper	14452	1598	10	0	60	30	90	38	5,55 h	247	451	3,12	0,26
	Video-Pre-Training	30502	3483	22	0	123	16	139	77	4,37 h	528	430	1,41	0,25
gen-3-rename	AutoRCCar	7018	925	13	0	23	4	27	16	1,73 h	110	156	2,22	0,23
	deep-text-corrector	13849	1434	8	0	23	0	23	15	2,90 h	193	228	1,65	0,11
	principle_violation	861	96	1	0	6	0	6	2	0,47 h	21	38	4,41	0,23
	social_mapper	14297	1657	10	0	52	22	74	27	4,23 h	290	493	3,45	0,19
	Video-Pre-Training	29259	3347	22	0	121	17	138	76	4,22 h	518	412	1,41	0,26

VI. Táblázat: *Összesített statisztikák GPT-4-Turbo*

Az imént látható adatokat összesítettük olyan táblázatban, ahol generációkra (promptokra) lebontva számoltuk ki az összesített statisztikát. Később a grafikonokat ebből készítettük. GPT-3.5-Trubo esetén ez a VII. táblázat, GPT-4-Turbo esetén ez a VIII. táblázat. Ezen táblázatok esetén az előző táblázatoktól eltérően, nem az eredeti állapothoz hasonlítottuk (formázást tekintve), hanem lineárisan színeztük a legjobb (zöld) és legrosszabb (narancs) elért eredmények közti értékeket.

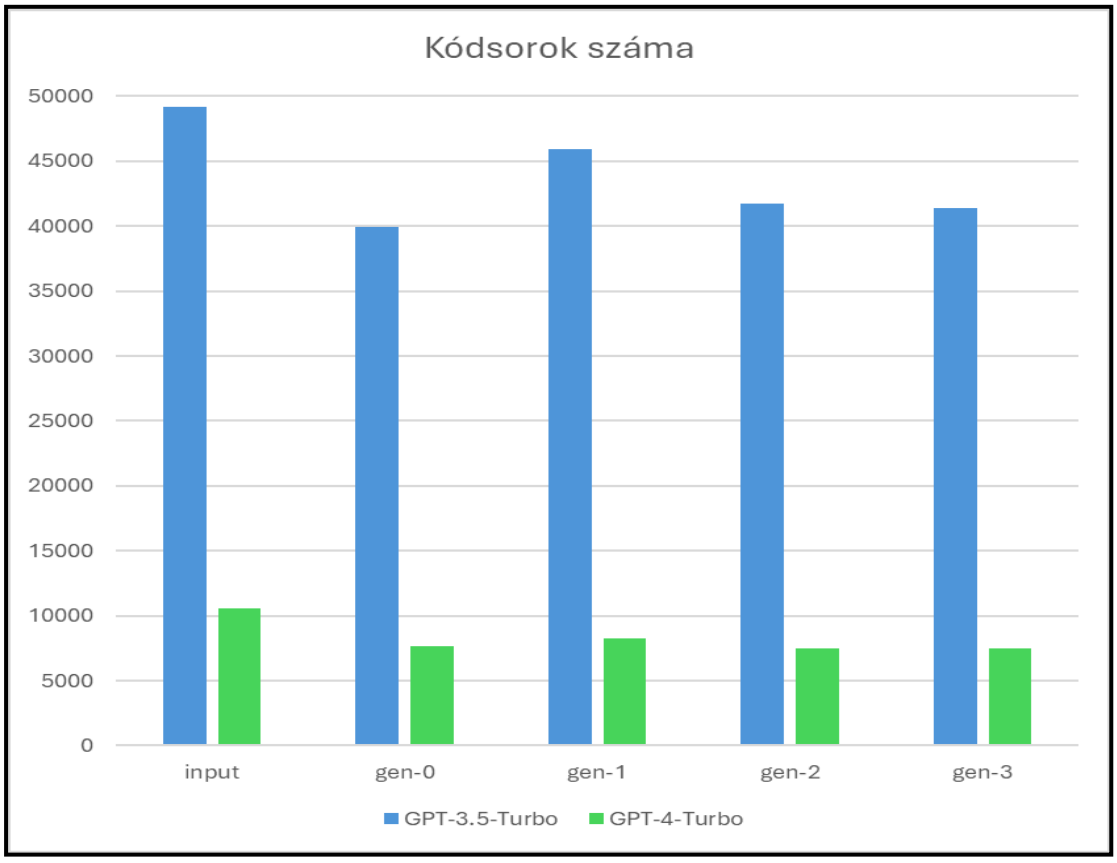
GPT-3.5-Turbo - Generációs Statisztikák (VII. Táblázat)					
Összesített Statisztika Neve	input	gen-0	gen-1	gen-2	gen-3
Kódsor	49141	39918	45887	41701	41382
Tokenszám	441446	359515	403600	369945	368466
Qodana Moderate	1271	1129	1177	1106	1058
Qodana High	152	1268	848	1666	412
Code Smell	1219	661	742	536	534
Debt	359,60 h	207,60 h	190,85 h	123,21 h	143,44 h
Cognitive C.	6820	4414	3879	3451	3981
Syntax Error (arány)	0,34%	17,11%	16,11%	17,11%	17,45%
CoC/100Token	16,65	15,30	12,33	11,51	11,95
CS/100Token	2,88	1,54	1,66	1,23	1,12

VII. Táblázat: *Generációs statisztikák GPT-3.5-Turbo*

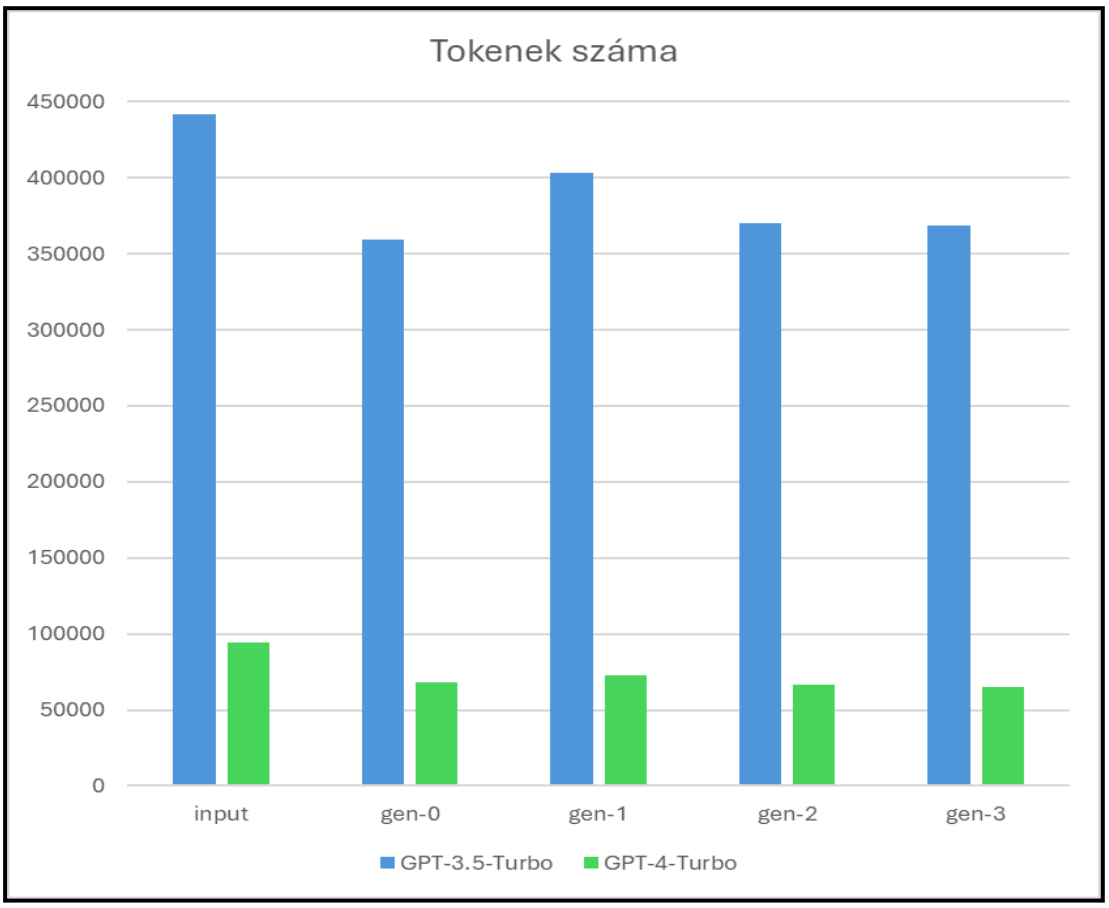
GPT-4-Turbo - Generációs Statisztikák					
Összesített Statisztika Neve	input	gen-0	gen-1	gen-2	gen-3
Kódsor	10555	7656	8229	7511	7459
Tokenszám	94629	68079	72446	66424	65284
Qodana Moderate	425	258	250	233	225
Qodana High	69	41	60	51	43
Code Smell	351	156	173	147	136
Debt	87,02 h	33,17 h	33,10 h	15,03 h	13,55 h
Cognitive C.	2112	1429	1476	1306	1327
Syntax Error (arány)	1,85%	0,00%	0,00%	0,00%	0,00%
CoC/100Token	13,00	13,41	13,20	12,92	13,14
CS/100Token	2,24	1,15	1,22	1,12	1,02

VIII. Táblázat: *generációs statisztikák GPT-4-Turbo*

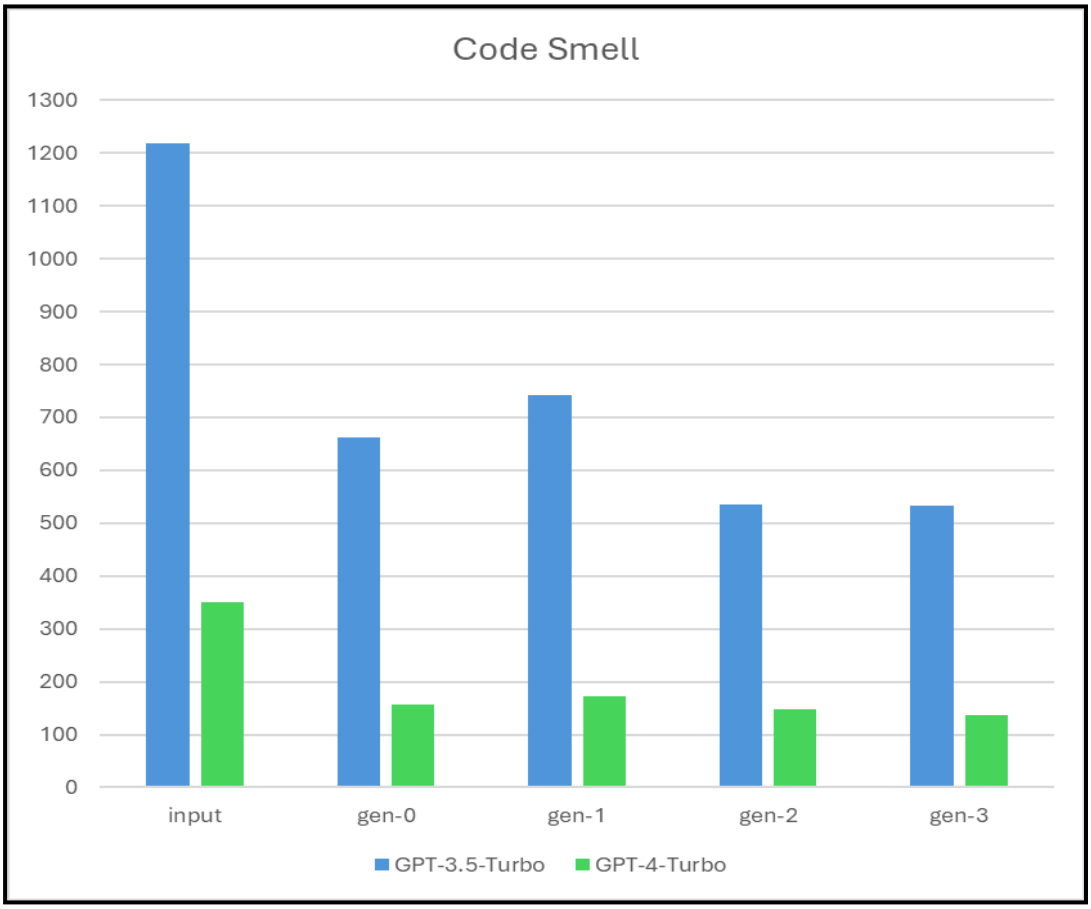
Diagramok



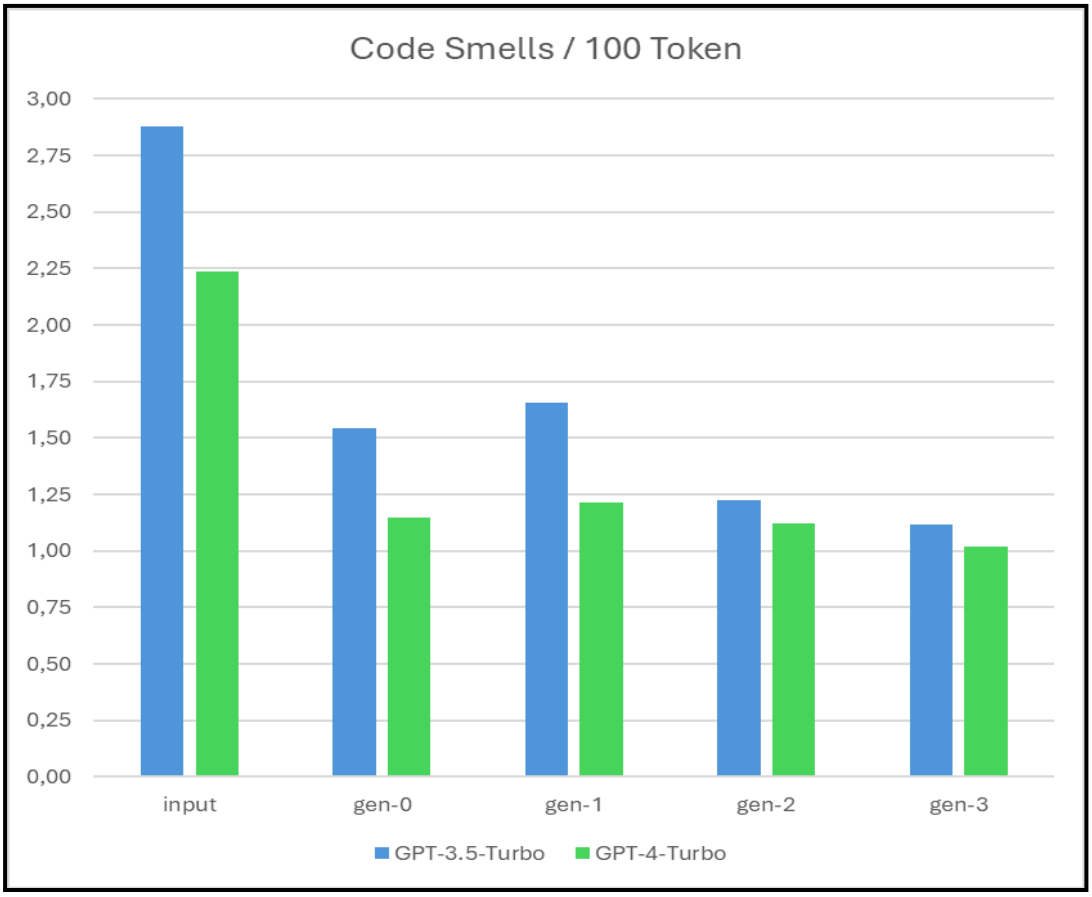
I. Diagram: *kódsorok számának változása*



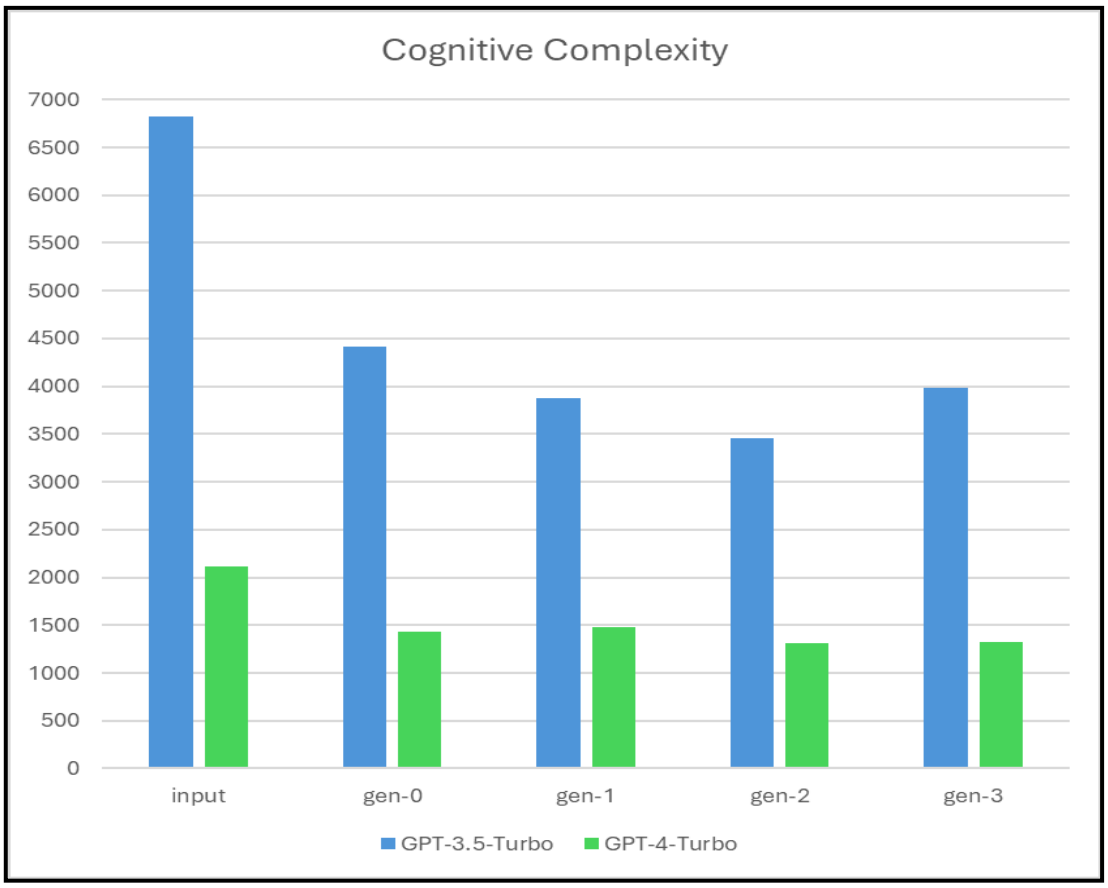
II. Diagram: *tokenek számának változása*



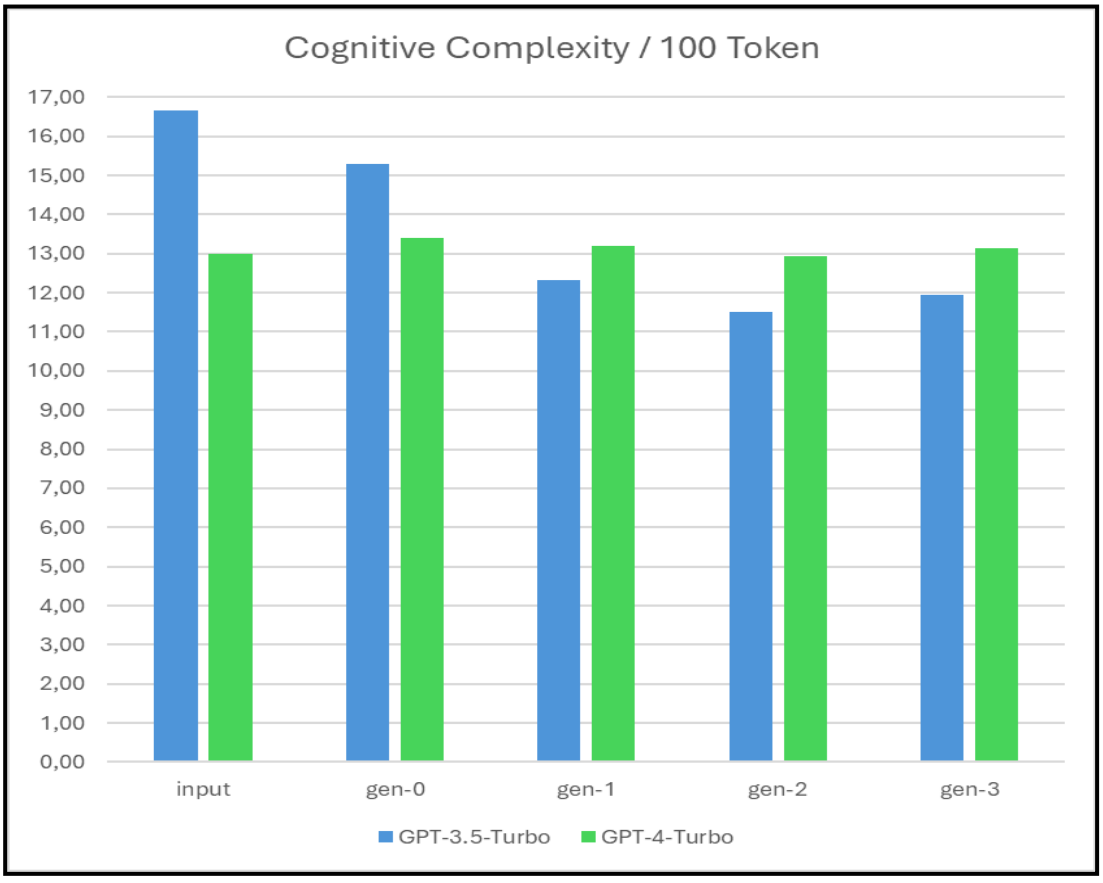
IV. Diagram: *code smellek számának változása*



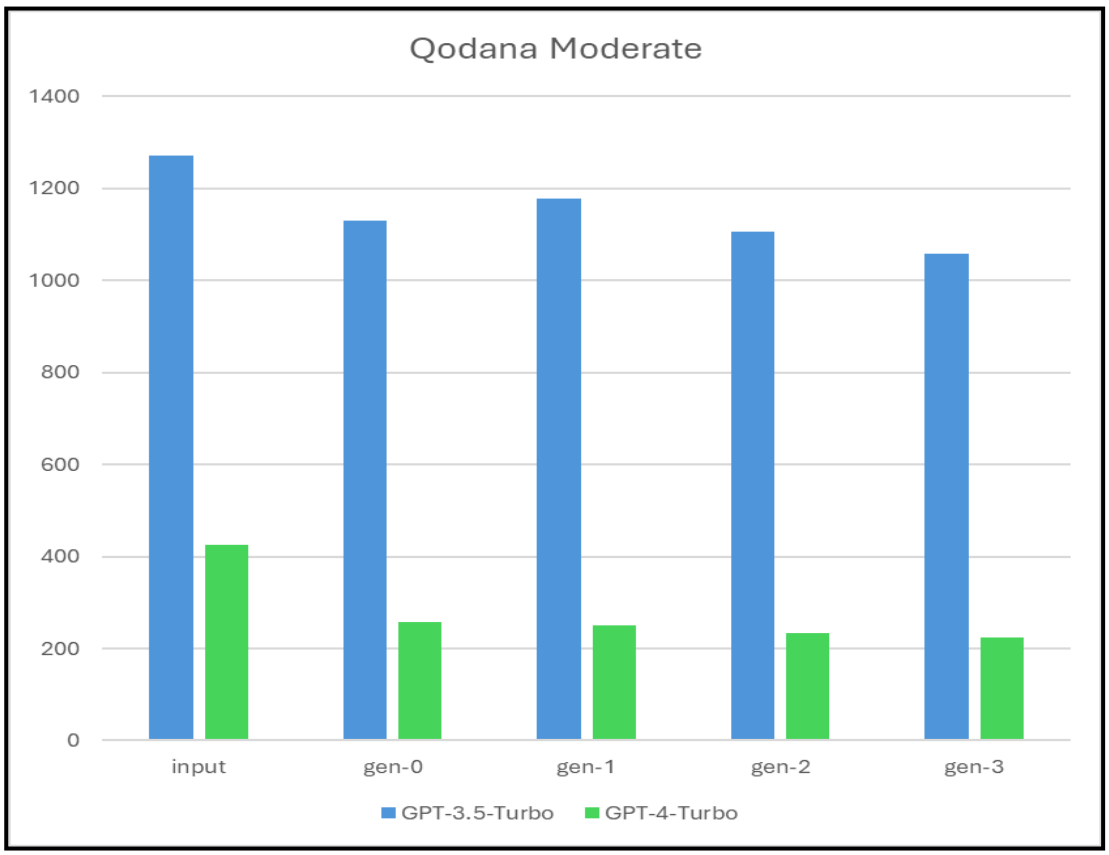
V. Diagram: *code smellek száma 100 tokenre vetítve*



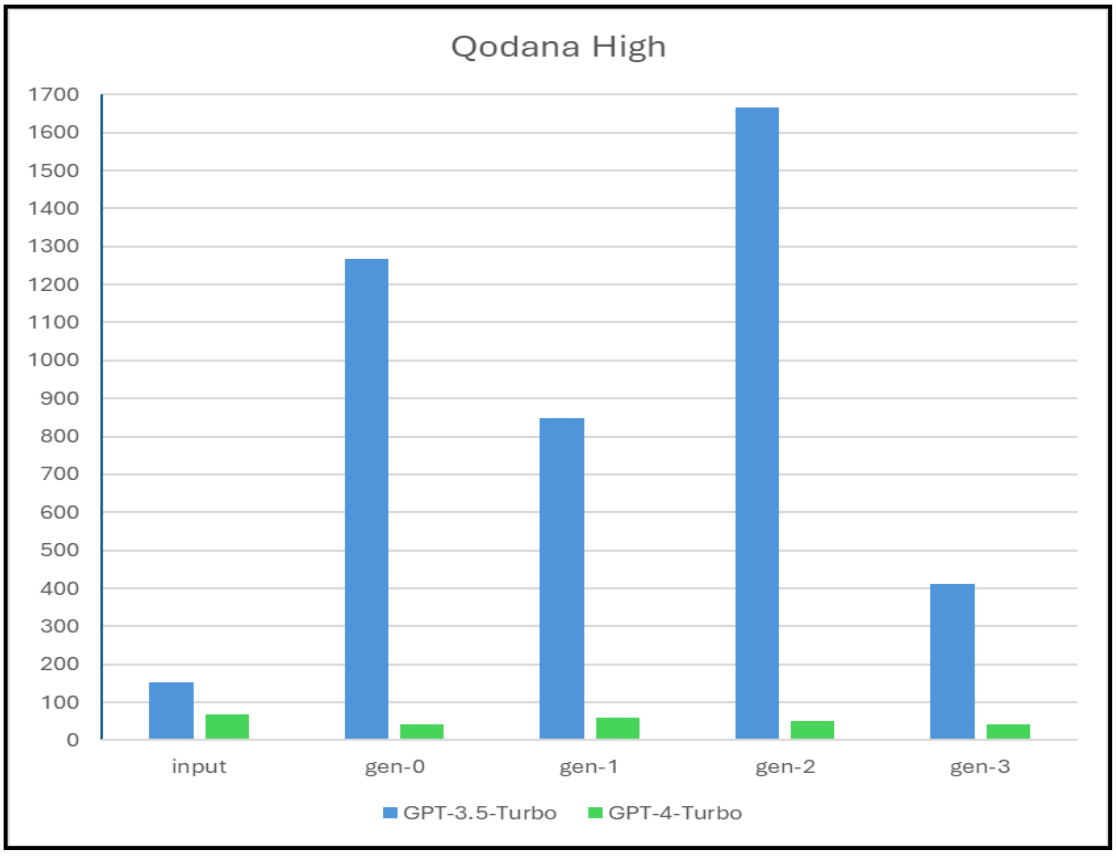
VI. Diagram: *cognitive complexity változása*



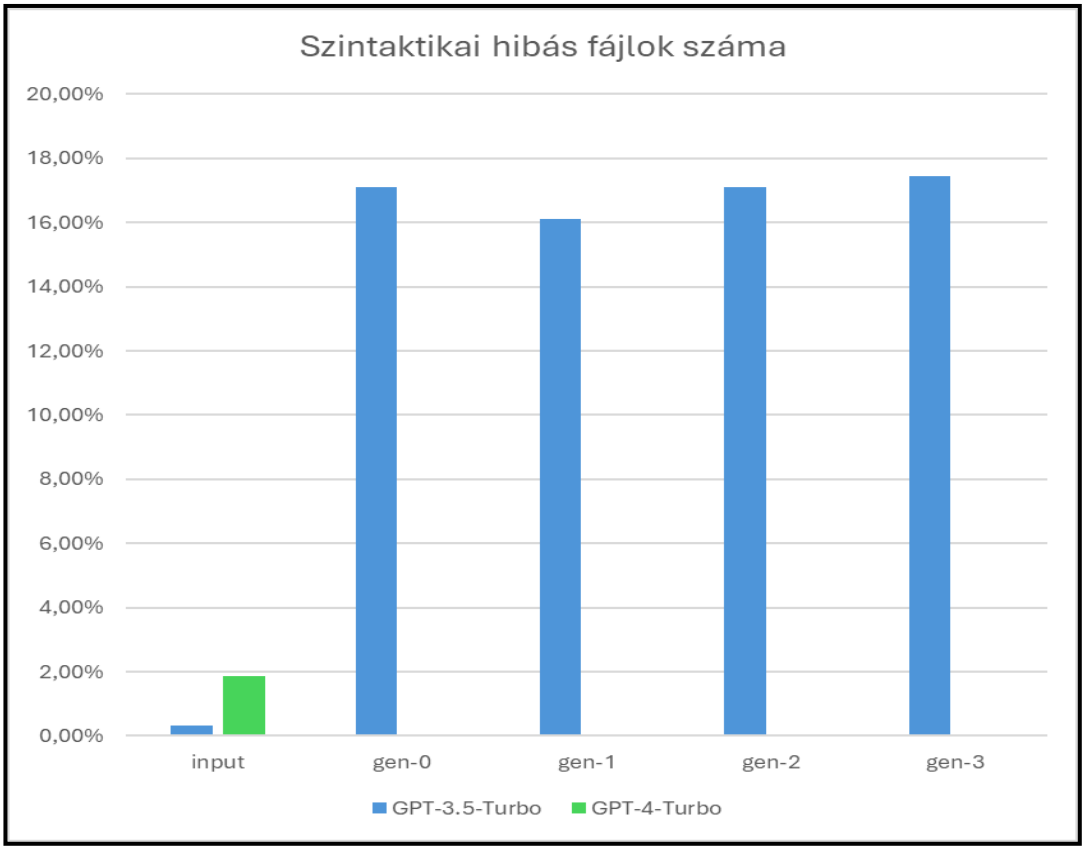
VII. Diagram: *cognitive complexity 100 tokenre vetítve*



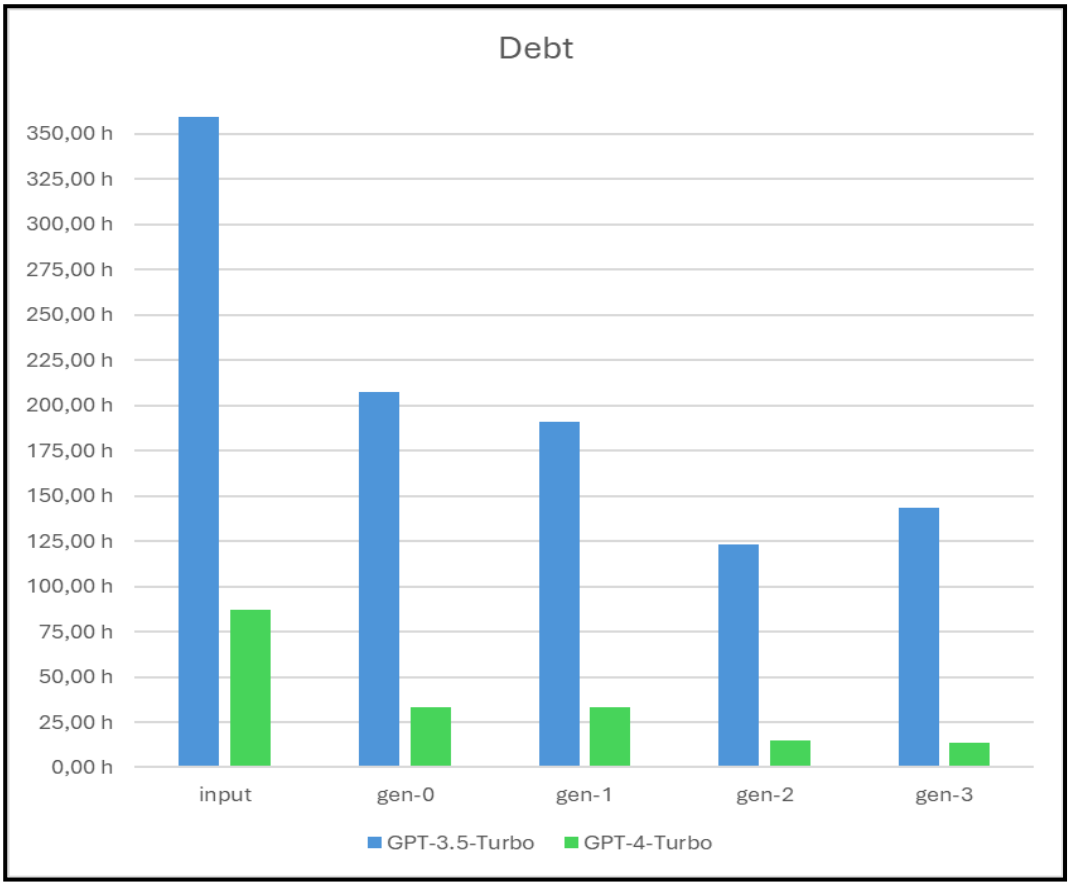
VIII. Diagram: *Qodana közepes figyelmeztetések száma*



IX. Diagram: *Qodana magas figyelmeztetések száma*



X. Diagram: *Szintaktikai hibás fájlok száma*



XI. Diagram: *Technical Debt változása*

Irodalomjegyzék

[1] Lj. Kazi, S. Mihajlović, M. Bhatt: Clean Code Quality Attributes and Measurements: an Initial Review (2022)
https://www.researchgate.net/profile/Sinisa-Mihajlovic-3/publication/366578085_Clean_Code_Quality_Attributes_and_Measurements_an_Initial_Review/links/63a81905097c7832ca6412ea/Clean-Code-Quality-Attributes-and-Measurements-an-Initial-Review.pdf

[2] Henning Grimeland Koller: Effects of Clean Code on Understandability (2016)
https://www.duo.uio.no/bitstream/handle/10852/51127/master.pdf

[3] Brett Hamm, Julia Zochodne, Louay Rafih, Ali El-Dani,Winston Chan
Clean Code
University of Calgary, Topic 9
http://kremer.cpsc.ucalgary.ca/courses/seng403/W2013/papers/09CleanCode.pdf

[4] Björn Latte, Sören Henning, Maik Wojcieszak: Clean Code: On the Use of Practices and Tools to Produce Maintainable Code for Long-Living (2019)
https://oceanrep.geomar.de/id/eprint/45829/1/ELMS2019CleanCode.pdf

[5] József Katona: Clean CodeOn the Use of Practices and Tools to Produce Maintainable Code for Long-Living (2021)
https://acta.uni-obuda.hu/Katona_108.pdf

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019)
https://aclanthology.org/N19-1423/

[7] 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 (2020)
https://aclanthology.org/2020.findings-emnlp.139/

[8] Nikola Luburić, Dragan Vidaković, Jelena Slivka, Simona Prokić, Katarina-Glorija Grujić, Aleksandar Kovačević, Goran Sladić: Clean Code Tutoring: Makings of a Foundation (2022)
https://www.scitepress.org/Papers/2022/108009/108009.pdf

[9] Sangchul Choi, Suntae Kim, Jeong-Hyu Lee, JeongAh Kim: Measuring the Extent of Source Code Readability Using Regression Analysis (2018)
https://www.researchgate.net/publication/326164236_Measuring_the_Extent_of_Source_Code_Readability_Using_Regression_Analysis

[10] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, Fernando Castor, Evaluating Code Readability and Legibility: An Examination of Human-centric Studies (2021)
https://arxiv.org/pdf/2110.00785.pdf

[11] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, Yujin Gao,
Improving code readability classification using convolutional neural networks (2018)
https://doi.org/10.1016/j.infsof.2018.07.006

[12] Gábor Szőke, Csaba Nagy, Lajos Jenő Fülöp, Rudolf Ferenc, Tibor Gyimóthy
FaultBuster: An automatic code smell refactoring toolset (2015)
https://doi.org/10.1109/SCAM.2015.7335422

[13] Gábor Szőke, Csaba Nagy, Rudolf Ferenc, Tibor Gyimóthy
Designing and Developing Automated Refactoring Transformations: An Experience Report (2016)
https://doi.org/10.1109/SANER.2016.17

[14] Judit Jász, Péter Hegedűs, Ákos Milánkovich, Rudolf Ferenc,
An End-to-End Framework for Repairing Potentially Vulnerable Source Code (2022)
https://doi.org/10.1109/SCAM55253.2022.00034

[15] Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, Tibor Gyimóthy
Do automatic refactorings improve maintainability? An industrial case study (2015)
https://doi.org/10.1109/ICSM.2015.7332494

[16] Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, Tibor Gyimóthy
Empirical study on refactoring large-scale industrial systems and its effects on maintainability
https://doi.org/10.1016/j.jss.2016.08.071

[17] Wenyuan Chen, Guangyong Li, Mingwei Li, Wenxue Wang, Peng Li, Xiujuan Xue, Xingang Zhao, Lianqing Liu
LLM-Enabled Incremental Learning Framework for Hand Exoskeleton Control
https://doi.org/10.1109/TASE.2024.3382679

[18] Alberts, I., Mercolli, L., Pyka, T. *et al.* Large language models (LLM) and ChatGPT: what will the impact on nuclear medicine be?. *Eur J Nucl Med Mol Imaging* **50**, 1549–1552 (2023).
https://doi.org/10.1007/s00259-023-06172-w

[19] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, Douglas C. Schmidt
ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design (2023)
https://arxiv.org/pdf/2303.07839v1

[20] Emilia Hansson, Oliwer Ellréus
Code Correctness and Quality in the Era of AI Code Generation - Examining ChatGPT and GitHub Copilot (2023)
https://www.diva-portal.org/smash/get/diva2:1764568/FULLTEXT01.pdf

[21] Fangzhou Wu, Ning Zhang, Somesh Jha, Patrick McDaniel, Chaowei Xiao
A New Era in LLM Security: Exploring Security Concerns in Real-World LLM-based Systems (2024)
https://arxiv.org/abs/2402.18649

[22] Rodrigo Pedro, Daniel Castro, Paulo Carreira, Nuno Santos,
From Prompt Injections to SQL Injection Attacks: How Protected is Your LLM-Integrated Web Application? (2023)
https://arxiv.org/abs/2308.01990