Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Iványi Balázs

PROGRAMOZÁS OKTATÁS TÁMOGATÁSA MESTERSÉGES INTELLIGENCIÁVAL ÉS JÁTÉKOS FELHASZNÁLÓI FELÜLETTEL

DR. EKLER PÉTER

BUDAPEST, 2023

Tartalom

Összefoglaló

Mérnökinformatikus hallgatóként számos alkalommal ütköztem bele a programozás tanulásának nehézségeibe. Az alapok elsajátítása különösen nehéz, hiszen egy teljesen más gondolkodásmódot, analitikus gondolkodást kell kialakítani, hogy le tudjuk fordítani ötleteinket gép számára is értelmezhető formára. Ezenfelül a különböző nyelvek egyedi megoldásai, a különböző szintaxisok és memóriakezelési módszerek megértése is nehézséget jelenthetnek egy gyakorlatlan programozó számára. A programozástudás számonkérése sem egyszerű feladat, hiszen olyan feladatokat kell kitalálni, amikben megjelennek a számonkért nyelv sajátosságai, azonban nem szabad túl szigorúan értékelni, hogyha lemarad néhány pontosvessző, vagy zárójel, amik papíron programozva sokszor nem tűnnek fel a dolgozatírók számára.

A laborok, gyakorlatok és házi feladatok jó lehetőséget biztosítanak arra, hogy a diákok visszajelzést kapjanak munkájukról, azonban ez sokszor nem elegendő ahhoz, hogy jó eredményeket érjenek el a számonkérések során is, ezért rendkívül fontos a rendszeres, önálló gyakorlás. Dolgozatombanbemutatom ötletemet, egy interaktív programozás tanulási és gyakorlási platformot, amivel a felhasználók fejleszthetik kódértési képességeiket. Lehetőségük van különböző nyelveken és nehézségi szinteken gyakorolni a „Mit ír ki a kód?” és „Mit csinál a kód?” jellegű feladatokat, amelyek különösen népszerűek a programozás zárthelyiken. A játszva tanulás hatásos módszer a figyelem és motiváció fenntartására, és megkönnyíti a diákok helyzetét a számonkérésekre való felkészülés során.

Összehasonlítom a különböző ellenőrzési módszereket, bemutatom előnyeiket és hátrányaikat, valamint megmutatom, hogy mennyire erős oktatásfejlesztési eszközt jelenthet a mesterséges intelligencia.

Abstract

As a software engineer student, I encountered numerous difficulties of learning programming. Learning the basics is particularly difficult, as you need to develop a completely different mindset, an analytical way of thinking, to translate your ideas into a machine-interpretable form. In addition, understanding the specific solutions of different languages, different syntaxes and memory management methods can be difficult for an inexperienced programmer. It is not an easy task to test one's knowledge of programming, since one must come up with exercises that reflect the specificities of the language being taught, but one should not be too strict about missing a few semicolons or brackets that are often not visible to the writer when programmed on paper.

Labs, exercises and homework provide a good opportunity for students to get feedback on their work, but this is often not enough to achieve good results in the exams, so regular independent practice is crucial. In my thesis, I present my idea of an interactive programming learning and practice platform for users to improve their code understanding skills. They can practice "What does the code print?" and "What does code do?" exercises, which are particularly popular in programming final exams. Gamified learning is an effective way to maintain attention and motivation and makes it easier for students to prepare for exams.

I will compare different methods of testing, present their advantages and disadvantages, and show how *AI* can be a powerful educational development tool.

1 Bevezetés

Utolsó előtti félévem során, az Önálló laboratórium tárgy elvégezhető feladatai között találtam rá „Az *OpenAI* és *ChatGPT* alkalmazási területeinek vizsgálata” című témára, ami azonnal felkeltette érdeklődésemet, hiszen tanulmányaim elejétől kezdve különösképpen foglalkoztat engem a mesterséges intelligencia. Végül egy cég külsős témáját végeztem el a tárgy során, azonban a lelkesedésemből nem veszítve, fél évvel később a TDK dolgozatom témájaként készítettem el ötletemet: az interaktív, programozásgyakorló játékomat, a *Secret Scriptet*.

Az ötletem egyszerű, azonban újszerű, nem találtam hasonló jellegű programozós játékot, ami oktatási célzattal készült és felhasználja az új, rendkívül erős nagy nyelvi modelleket. Egy olyan platform kialakítása volt a célom, amin a felhasználók egyrészt programozási feladatokat generáltathatnak az *OpenAI API* használatával, akár csak a programozási nyelv és a kívánt nehézségi szint kiválasztásával, vagy akár egy nem túlzottan komplikált algoritmus természetes nyelvű leírásával, majd elmenthetik ezen generált kódpéldákat a feladványokat tároló adatbázisba. Másrészt, játszhatnak a felületen, amely játéknak a lényege a következő: a játékos kiválasztja a gyakorlandó programozási nyelvet és a kívánt nehézségi szintet, valamint játékmódot, és a paramétereknek megfelelő feladványhalmazból véletlenszerűen kap egy feladatot, amelyet meg kell fejtenie. A feladvány tulajdonképpen egy függvény, amit a játékosnak elemeznie kell, figyelve az adott nyelv sajátosságaira, majd játékmódtól függően írásban el kell magyaráznia annak működési mechanizmusát a felületen biztosított helyen, vagy ki kell javítania a kódban található hibát, vagy adnia kell egy *promptot,* ami hasonló eredményt produkál. A megfejtést ellenőrzi a felület, majd amennyiben a megoldás helyes, a játékost ponttal jutalmazza és ad egy újabb feladatot.

A játék részét képezi több gyakorolható nyelv, a különböző nehézségi szintek és játékmódok, illetve a pontozótábla, amin a játékosok összemérhetik pontszámaikat.

Célom az volt, hogy innovatív módszereket alkalmazva megkönnyítsem a programozás tanulását, és hogy példát mutassak, hogy már mindenki számára elérhető eszközkészlettel, és esetlegesen az MI bevetésével sikeresen megreformálható az oktatásnak számos területe. Véleményem szerint a különböző területek népszerűsítésének legfontosabb része, hogy interaktív eszközöket biztosítsunk a témák alapszintű

megértéséhez: legyen az akár egy ismertető videókat tartalmazó platform, vagy esetleg egy játék.

Motivációm részét képezte az is, hogy feladatom által lehetőségem akadt felfedezni az *OpenAI API-jának* képességeit, össze tudtam vetni a különböző nyelvi modellek tudását, és sokat tanulhattam arról, hogyan lehet úgy utasításokat (továbbiakban: *promptokat*) megfogalmazni, hogy azok az elvárt eredményt produkálják.

Dolgozatom során bemutatok különböző megoldásokat, amikkel a felület funkcióit valósítottam meg, megismertetem a nehézségeket jelentő problémákat, illetve visszatekintő jelleggel megmutatom, hogy hasonló megoldások fejlesztése esetén a tapasztalataim alapján mire érdemes ügyelni még jobban.

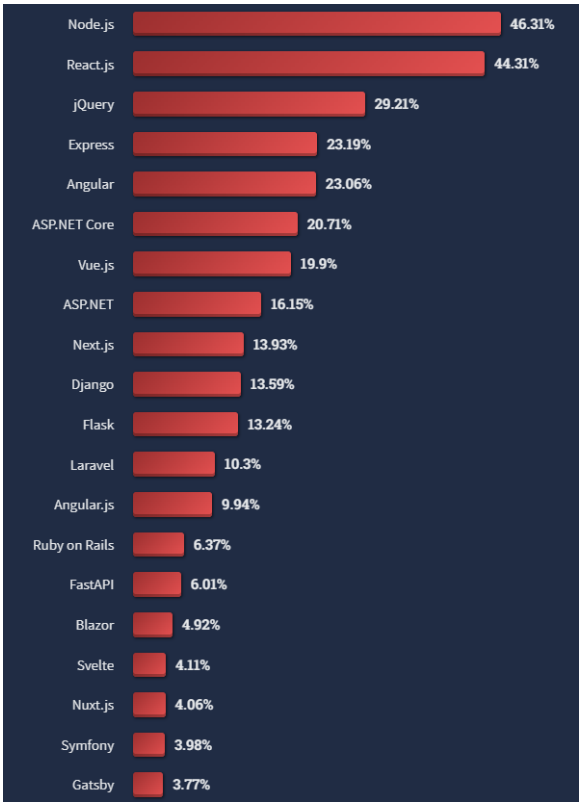
Az általam megvalósított megoldás és eljárások jó alapjául szolgálhatnak hasonló rendszerek fejlesztésének, melyek úgy gondolom a jövő programozás oktatásában fontos szerepet fognak játszani.

1.1 *Technológiaválasztás*

Tanulmányaim során lehetőségem volt sok különböző technológiát megismerni, ami rendkívül hasznosnak bizonyult, hiszen ez a tudás megfelelő alapot biztosított ennek a nagyobb volumenű projektnek a megvalósításához. A házi feladatok esetében kellett már hasonló, *full-stack* jellegű munkát végeznem, azonban ezeknél adottak voltak a használandó technológiák. Ennél a feladatnál azonban szabad kezet kaptam a technológiák megválogatásában, így nekem kellett kitalálnom, hogy mivel tudnám leginkább a képzeletemnek megfelelően kivitelezni az ötletemet.

A kezdetektől fogva egy könnyűsúlyú, könnyen skálázható, egyszerűen megérthető és használható platformot szerettem volna kialakítani, amire egy böngészőbeli vékonykliens tűnt a legalkalmasabbnak. Az alapismereteken kívül nem rendelkeztem átfogó tudással a webfejlesztés tekintetében, és úgy gondoltam, hogy valami újszerűbb és kezdők számára barátságosabb eszközt használnék fel, mint a régóta használt, keretrendszer nélküli *HTML*, *CSS* és *JavaScript* hármas. Két technológiát találtam, amiket megfelelőnek tartottam az előzőekben leírtakhoz: a *Google Angular* nevű keretrendszerét, illetve a *Meta* (korábban *Facebook*) *React* nevű keretrendszerét. Utánaolvastam a két technológiának és úgy találtam végül, hogy a *React* (vagy *ReactJS*) hatékonyabb ilyen jellegű feladatokhoz, illetve népszerűbb is (az alábbi ábrán [1]

találhatók a legnépszerűbb keretrendszerek), így célszerűbbnek tűnt ennek a részletesebb megismerése.



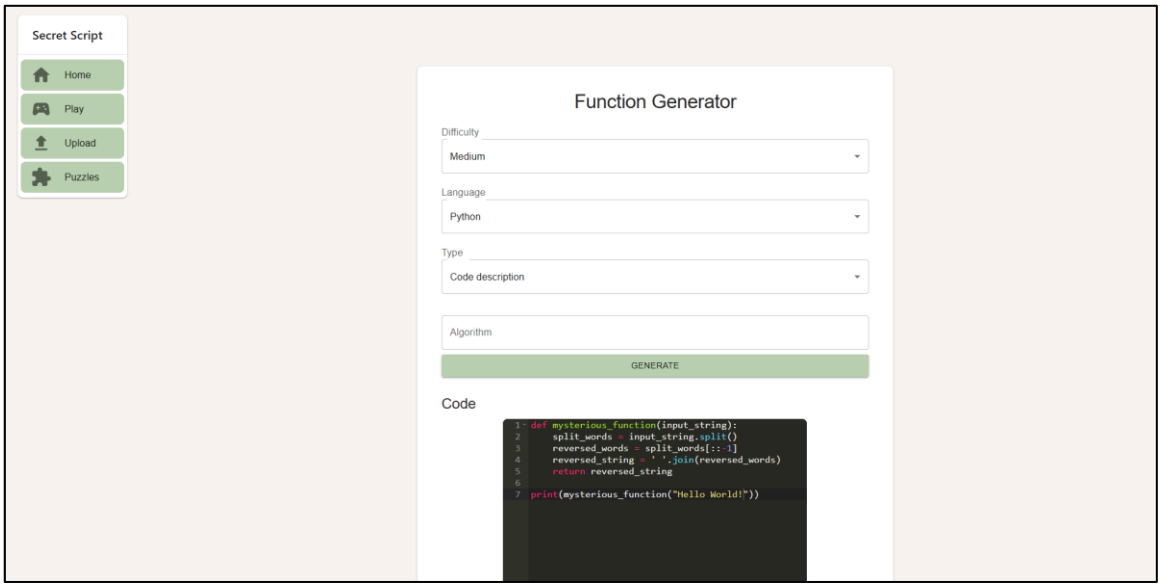
1. ábra A webes keretrendszerek elterjedtsége

A feladványok eltárolásához szintén egy sokak által használt rendszert választottam, hisz ez előnyös lehet a későbbi munkalehetőségek szempontjából, így egy *MySQL* adatbázist hoztam létre. A felület és az adatbázis közötti kommunikációt *Node.JS-szel* hajtom végre.

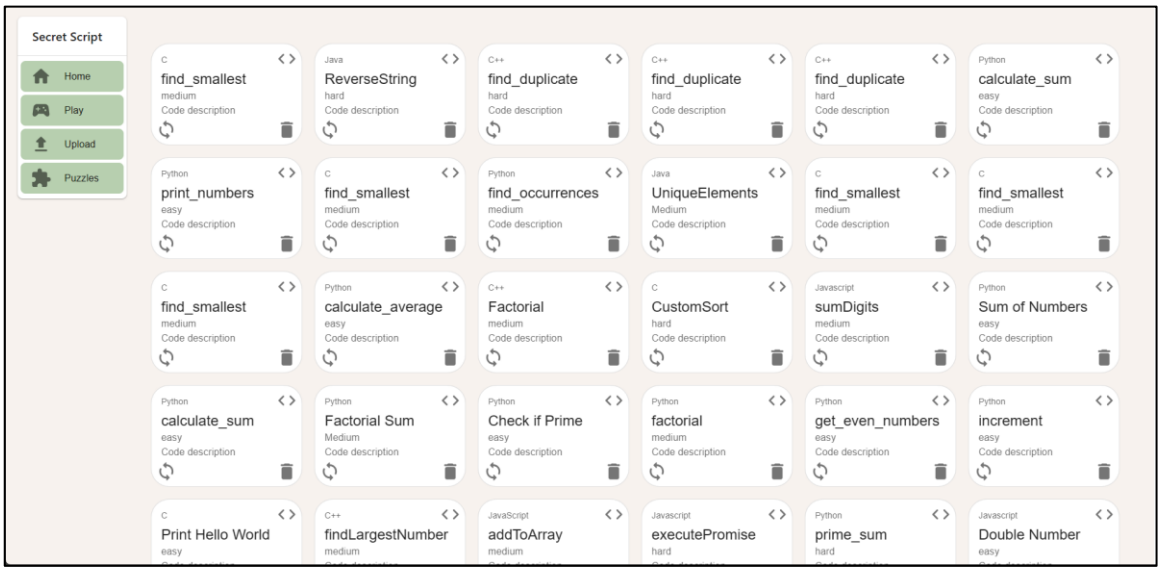
A következő fejezetben részletesen ismertetem a használt technológiákat, bemutatom, miért bizonyultak megfelelő választásnak a játék megvalósításához, illetve megmutatom, hogy milyen erős eszközöket biztosít az *OpenAI* [2], ami a közeljövőben átdefiniálhatja, hogy hogyan készítünk programokat.

A második fejezet tehát a felhasznált eszközök alapismereteiről, a harmadik fejezet a tervezési megfontolásokról és a technológiák felhasználási módjáról szól. A negyedik fejezetben ismertetem a generatív MI számára írt utasítások tulajdonságait és használatát. Az ötödik fejezet a különböző ellenőrzési módszerekről szól, a hatodik fejezetben megmutatom, hogy milyen funkciókat tartalmaz a játékom, példákkal

bemutatom azok használatát, bemutatom az elért eredményeket. A hetedik fejezet az eredmények összefoglalását és a továbbfejlesztési lehetőségeket ismerteti.



2. ábra *Upload Page*; Feladatgeneráló felület



3. ábra *Puzzle Page*; A feladatokat egységesen megjelenítő oldal

2 A kutatás során felhasznált technológiák bemutatása

Ebben a fejezetben bemutatom a választott technológiákat, amelyekkel sikerült elérnem a prezentált felületeket és amiket felhasználtam a munka során megvalósított rendszerem elkészítéséhez.

2.1 *React/ReactJS*

A *React* egy rendkívül erős, nyílt forráskódú, ingyenes *JavaScript* könyvtár, amellyel hatékonyan lehet felhasználói felületeket létrehozni különböző komponensek segítségével. Fenntartásáért a *Meta*, illetve egy egyéni fejlesztőkből álló közösség felelős.

2.1.1 Virtuális DOM

A *Document Object Model* (röviden *DOM*) [3] egy olyan webes dokumentumleíró, mely a *HTML* vagy *XML-dokumentumok* szerkezetét objektumokból álló faként ábrázolja, aminek minden egyes eleme (például a *HTML-címkék*, vagy a szöveges elemek) a fa egy-egy csomópontja. A *DOM* biztosítja, hogy a programok interakcióba léphessenek a weblappal, lehetővé téve annak dinamikus manipulációját és módosítását.

A virtuális *DOM* egy olyan koncepció, amivel optimalizálták a tényleges (azaz nem virtuális) *DOM* frissítésének a folyamatát. A virtuális *DOM* nem egy weboldal tényleges reprezentációja, hanem annak egy „könnyebb szerkezetű”, memóriában eltárolható formája. Amikor *Reactban* módosítunk egy felületet leíró kódot, azok a módosítások először a virtuális *DOM-on* mennek végbe. A *React* összeveti a korábbi *DOM-ot* az újabbal, hogy azonosítsa a végrehajtandó változtatásokat (ezt angolul *diffingnek* nevezik), végül csak ezeket a szükséges a változtatásokat hajtja végre a valós *DOM-on* (ezt pedig *reconciliationnek* hívják). [4]

A virtuális *DOM* a *React* keretrendszer egyik legfontosabb jellemzője, hiszen jelentősen javítja a webes alkalmazások teljesítményét és hatékonyságát, egyszerűvé és „olcsóvá” teszi a *DOM* manipulációt, illetve használatakor nem keletkezik memóriaszemét.

2.1.2 Deklaratív megközelítés

A deklaratív programozási paradigma lényege, hogy a programozó azt írja le, hogy *mit* szeretne elérni, nem pedig azt, hogy *hogyan*. Ez a jelen kontextus értelmében azt jelenti, hogy a fejlesztőnek elegendő azt leírni, hogy hogyan nézzen ki a felület, nem kell imperatív módon, lépésről-lépésre meghatároznia, hogy a böngésző milyen módon frissítse a felületet. Ezeket a háttérfolyamatokat a *React* automatikusan elvégzi az előző alfejezetben leírtak szerint.

A deklaratív megközelítés másik példája az úgynevezett *JavaScript XML* (vagy röviden *JSX*) fájlok használata. Ez tulajdonképpen egy kiegészített *JavaScript*, amelynek segítségével *HTML-szerű* kódot lehet írni *JavaScript* kódon belül. Elég leírni, hogy mit szeretnénk látni a felületen (például egy gombot valamilyen felirattal) anélkül, hogy manuálisan kéne ezeket a *HTML* elemeket *JavaScriptben* létrehozni.

A bemutatott deklaratív megoldás jelentősen növeli a megírt kód érthetőségét, kiszámíthatóbbá teszi a felhasználói felületet és növeli a teljesítményt.

2.1.3 Komponensek

A komponens alapú megközelítés lehetővé teszi a kényelmes kódolást és a jól átlátható kódot. A komponens lényegében egy újra felhasználható felületelem. A komponensek modularitása lehetővé teszi, hogy kisebb, egymástól függetlenül működő felületelemekből komplex, nagyméretű alkalmazásokat készítsünk. Minden komponens saját állapottal, működési logikával és megjelenéssel rendelkezik, ezáltal jelentősen könnyebb a hibakeresés, a tesztelés, illetve a karbantartás is.

A komponensek által megvalósul a *Separation of Concerns* (*SoC*) tervezési elv, hiszen minden komponensnek saját feladata van, így minimalizálva az átfedést közöttük. Ez hozzájárulhat a dinamikusabb csapatok közötti kooperációhoz, hiszen amíg valaki egy adott komponensen dolgozik, azzal párhuzamosan lehet dolgozni annak a nagyobb felületbe való integrálásán is.

A szoftvereknek robusztusaknak, de rugalmasaknak kell lenniük, amire a *React* komponenses megközelítése hatékony és jól skálázható megoldást kínál.

2.1.4 Harmadik féltől származó tartalmak

Ebben a fejezetben ismertetem a tudnivalókat a *third party*, azaz harmadik féltől származó tartalmakról, és bemutatom, miért és hogyan célszerű használni azokat.

2.1.4.1 Mik azok a harmadik féltől származó tartalmak?

A harmadik féltől származó tartalmak olyan külső fejlesztőktől származó komponensek és könyvtárak, amiket be lehet integrálni a *React* alkalmazásokba. Ezek előre elkészített megoldásokat kínálnak, amikkel rengeteg időt nyerhetnek a fejlesztők, hiszen nem kell nulláról elkészíteniük az adott komponenseket. Sok komponens nagyon széles körben használt, ami a megbízhatóságukról tanúskodik, azonban mindig nagyon oda kell figyelnünk, hogy biztosak lehessünk abban, hogy az általunk használt, mástól származó komponensek **csak olyan** funkcionalitást végeznek el, amit mi szeretnénk.

A harmadik féltől származó tartalmak egyéb kockázatokkal is járhatnak: előfordulhat, hogy két külön féltől származó komponensnek másféle függőségei vannak egy harmadik komponens felé, ezáltal kompatibilitási problémát okozva; illetve, ha nem frissítik, akár elavulttá válhatnak bizonyos komponensek, ami által biztonsági kockázatot is jelenthetnek.

2.1.4.2 *Material UI* (*MUI*)

A *Material UI* [5], vagy röviden *MUI* az egyik leghasználtabb harmadik féltől származó felhasználói felület keretrendszer. Olyan előre elkészített komponenseket kínál, amik a *Google Material Design* alapelveihez igazodnak, aminek a lényege, hogy egységes megjelenítést teremtsen a különböző nézeteken és platformokon. A *MUI* tehát ezen elvek szerint stilizált komponenseket kínál, többek között gombokat, navigációs menüket, szövegdobozokat stb., amikkel koherens dizájnt és ízléses megjelenést lehet elérni.

A *MUI* további nagy előnye, hogy egyszerűen használható és rendkívül testreszabható: elérhetőek előre megtervezett elemek is, azonban lehetőség van sajátos arculatú weboldalt készíteni a dizájn nélküli alapelemekkel.

A projektem kezdetén csupán az alap webfejlesztési tapasztalatom (*HTML, CSS, JavaScript*) és a tervezési elvekre vonatkozó alapismeretem volt meg, nem ismertem semmilyen kiegészítő keretrendszert, sőt, még a *React* „ökoszisztémát” sem. Ahogy egyre több problémába ütköztem, felfedeztem, hogy sokan kínálnak megoldásokat

azokra, akár teljes komponensek formájában is. Miután felfedeztem a *Material UI-t*, egyre több és több komponenst használtam fel a kínálatukból, míg a végére szinte minden alapelem abból a könyvtárból származott, illetve a saját komponenseimet is azokból az elemekből rakosgattam össze.

2.2 *MySQL*

A *MySQL* egy nyílt-forráskódú relációs adatbázis-kezelő rendszer, amely 1995-ben került piacra, és azóta rendszeresen frissítik. Jelenleg a második legnépszerűbb adatbázis-kezelő rendszer, nem sokkal lemaradva az *Oracle-től*. Hatékonyan és strukturáltan tárolja az adatokat, valamint hatékony adatmanipulációt tesz lehetővé *SQL* nyelv használatával.

Népszerű választás mindenféle méretű felhasználásra, hiszen rendkívül felhasználóbarát (részletes dokumentációval rendelkezik), biztonságos, jól skálázható, ezenfelül ingyenes is. Ezen tulajdonságai miatt robosztus alkalmazásoknak szolgálhat alapjául. [6]

2.3 *Node.JS*

A *Node.JS* egy olyan platformfüggetlen, nyílt-forráskódú szerverkörnyezet, amely lehetővé teszi szerveroldali *JavaScript* kódok futtatását. Ezáltal lehetővé vált *full-stack* alkalmazások készítése csupán *JavaScript* használatával. *A Chrome V8 JavaScript-motorjára* épül, és jól skálázható, hatékony webalkalmazások készítését teszi lehetővé.

2.3.1 A *Node.JS* alapjai

A *Node.JS* aszinkron módon, eseményvezérelt architektúrával működik, ami által biztosítja, hogy az IO-műveletek ne blokkolják a többi folyamatot. Ez azt jelenti, hogy az adatbázishoz irányuló műveletek (beillesztés, törlés, módosítás) vagy fájlok írása, olvasása nem állítják le a többi folyamatot, így gyorsabb működést és reszponzív felhasználói felületet biztosítva.

Ezenfelül lehetővé teszi a sokak által használt *Node Package Manager (NPM)* környezet használatát, ami megkönnyíti a *Node.JS* csomagok megosztását, telepítését és frissítését is. Az *NPM-ben* különböző csomagok hatalmas könyvtára áll rendelkezésre.

2.3.2 *Node.JS* és *React*

A két említett technológia kombinációja nagyon gyakori a webalkalmazások körében. A bevezetőben említettem, hogy a *Node.JS* által lehetséges *egy* programozási nyelvet használva *full-stack* alkalmazást készíteni, én is főként emiatt a tulajdonsága miatt döntöttem ennek használata mellett, ezáltal jobban elmélyíthettem *JavaScript* ismereteimet.

A *React* felelős a felhasználói felületen történő interakciókért, míg a *Node.JS* kezeli a háttérhívásokat. A kettő kombinációja lehetővé teszi, hogy felületi elemek adatbázis felé irányuló kommunikációt indítsanak, amelynek eredménye megjelenhet a felhasználói felületen. Ezenfelül lehetővé teszi a weboldal szerveroldali *renderelését* is (*Server-Side Rendering, SSR*), ami által az oldal kezdeti nézete a szerveren kerül feldolgozásra, ezáltal csökkentve az oldal betöltési idejét, hogy a kliensek hamarabb látható eredményt kaphassanak.

2.3.3 *Node.JS* és *MySQL*

A *Node.JS* és *MySQL* szerver együttes használata lehetővé teszi, hogy a felületről kezdeményezett adatbázishívások végbemenjenek a tényleges adatbázison. A *Node.JS* szerver csatlakozni tud a *MySQL* adatbázishoz (a *MySQL2 npm* csomag segítségével), és végrehajthatja a beillesztés, törlés, módosítás adatbázisműveleteket Ez biztosítja a valós idejű adatmegjelenítést és adatmanipulációt is, ezáltal interaktív felhasználói felületet nyújtva. A kettő együttes használata jól skálázható szoftver készítését teszi lehetővé, hiszen a *Node.JS* több párhuzamos kérés kiszolgálására képes, míg a *MySQL* szerver kiválóan képes kezelni a nagy adatmennyiséget.

2.4 *Mesterséges intelligencia, OpenAI*

Ebben a fejezetben bemutatom általánosságban a mesterséges intelligenciát és az *OpenAI* szervezetet.

2.4.1 Bevezetés

A mesterséges intelligencia (röviden MI) korunk átalakító erejévé vált. Rendkívül rövid idő alatt robbant be a köztudatba, és manapság a mindennapjaink során is lehetetlen elkerülni. Felettébb vegyes megítéléssel bír, ezt a különböző hangvételű cikkeken kívül a sok eltérő véleményű emberrel folytatott beszélgetéseimen és vitáimon keresztül is

megtapasztaltam. Rendkívül érdekesnek tartom a témát és azt a tényt is, hogy milyen sokféleképpen lehet ezt megközelíteni, a szigorúan földhözragadt technikai tényektől kezdve egészen az utópisztikus összeesküvéselmélet-szerű teóriákig.

A következő fejezetekben az MI alapjait és fontosságát, az *OpenAI* eddig elért eredményeit és kereskedelmi felhasználhatóságát fogom bemutatni, valamint a végén a felmerülő etikai kérdésekről és a közvélemény megítéléséről fogok írni.

Összességében pozitívan ítélem meg az MI témakört és azok közé szeretnék tartozni, akik megpróbálják úgy felhasználni a mesterséges intelligenciát, hogy az a lehető legsokrétűbben segítse az emberiséget. Az oktatás és annak megreformálása szintén fontos és érdekes feladat, és úgy gondolom, hogy manapság rendkívül jó eszközök vannak a kezünkben, csak megfelelően kell tudni alkalmazni azokat.

2.4.2 Mesterséges intelligencia

A mesterséges intelligencia kifejezés manapság már eléggé elcsépeltnek tekinthető. Lényegében néhány algoritmustól kezdve egészen komplex nyelvi modellekig egyaránt jellemeznek bármit mesterséges intelligenciaként. A tapasztalatom az, hogy minél kevésbé laikus valaki a témában, annál kevesebb dologra alkalmazza az MI szót, mondván, hogy *„Az csak egy szimpla XY algoritmus, semmi több.”*. Általában olyan dolgokra használjuk a kifejezést, amelyek olyan feladatokat hajtanak végre, amik intelligens lényekkel hozhatók kapcsolatba.

A 20. század elején bebizonyosodott, hogy a számítógépek rendkívül bonyolult feladatok elvégzésére is képesek lehetnek, mint például a sakkozás, vagy akár matematikai tételek bebizonyítása, ez azonban nem több, mint feladatorientált problémamegoldás. Nem megyek bele a továbbiakban az *intelligencia* meghatározásának filozófiai mélységeibe, hanem bemutatom, hogy miért bír nagy fontossággal mai életünkben az MI.

A mindennapjaink során már mindenfelé MI támogatott rendszerekbe ütközünk: a hangvezérelt asszisztens a telefonunkon, az internetes üzletek akciós hirdetésinek sokasága az email fiókunkban, a különböző platformok ajánlórendszerei, vagy akár a 4-es metró „sofőrje”. Ezenfelül kiemelendő a mesterséges intelligencia használata az orvostudományban, ahol néhány műtétet már teljes egészében MI végez, illetve a gazdasági szektorban, ahol a folyamatok jelentős részét szintén algoritmusok tömkelege

támogatja. A sokrétű felhasználásából látható, hogy az MI több, mint puszta eszköz, mára kulcsfontosságú tényezővé vált, ami napról napra megváltoztatja életünket.

2.4.3 Általános célú MI

Az általános célú mesterséges intelligencia (angolul *Artificial General Intelligence*, *AGI*) egy koncepció egy olyan MI-ről, amely képes akár olyan dolgokat megérteni, megtanulni, illetve elvégezni, melyekkel korábban még nem találkozott. A bizonyos feladatokra specializált MI-től abban különbözik, hogy képes egy adott területen megszerzett tudását más területen alkalmazni.

2.4.4 *OpenAI*

Az *OpenAI* egy 2015-ben alapított kutatási szervezet, amelynek célja, hogy biztonságos általános célú mesterséges intelligenciát hozzon létre. Kutatásaikat többségében nyíltan végzik, és eredményeik nagy részét publikálják. Víziójuk közé tartozik, hogy egy mindenki számára elérhető általános célú mesterséges intelligenciát fejlesszenek ki.

Az *OpenAI* oldalán számos különböző mérföldkő elérése során kiadott tudományos cikk szabadon elérhető. Eredményeik közül kiemelendő az *OpenAI Gym*, amely egy megerősítéses tanulási algoritmusok fejlesztésére szolgáló eszköz, illetve a következő fejezetben bemutatott modellek.

2.4.4.1 Modellek

A modellek bemutatása előtt muszáj megemlítenünk a generatív mesterséges intelligencia alapvető tulajdonságait. Azért *generatív*, mert képes új tartalmak generálására, legyen az szöveg, kép, zene, vagy bármilyen más adat. Az input adatok feldolgozása mellett, annak kontextusához releváns eredményt állítanak elő, a nagy mennyiségű tanító adathalmazból felismert minták alapján.

Az *OpenAI* 2018-ban mutatta be a *Generative Pre-trained Transformer* (*GPT*) nyelvi modellt, amely egy olyan neurális hálózat, amit emberi agyhoz hasonló működésre terveztek. Ezt követte 2021-ben a *Dall-E*, ami egy képek készítésére való generatív MI, illetve a legismertebb *OpenAI* termék, a *ChatGPT* 2022-ben került bemutatásra, ami egy komplex feladatok elvégzésére képes, *GPT* alapú csetrobot.

Az alábbi táblázatban bemutatom a különböző modelleket [7]:

Modell	**Kiadás éve**	**Elért áttörés**
***GPT*-1**	2018 február	Nyelvi modell; bizonyította a további kutatások szükségességét
GPT-2	2019 február	Nyelvi modell; *GPT*-1-hez képest tízszeres paraméter szám és tanító adathalmaz; koherens és releváns válaszok
GPT-3	2020 június	Nyelvi modell; *GPT-2*-höz képest több, mint százszoros növekedés; kontextus mélyebb megértése; kevesebb szükséges instrukció
GPT-4	2023 március	Nyelvi modell; komplex *promptok* jobb megértése; kép alapú *promptok*
ChatGPT	2022 november	Nagy nyelvi modell alapú csetrobot
DALL-E	2021 január	A *GPT-3* verziója, szöveg alapú képgenerálásra
***DALL-E* 2**	2022 november	*API*; magasabb felbontás
***DALL-E* 3**	2023 október	*ChatGPT*-be integrált szövegalapú képgeneráló modell
***Whisper* [8]**	2022 szeptember	Beszédfelismerő modell

Táblázat 1 - *GPT* modellek bemutatása

2.4.5 Kereskedelmi felhasználás

Az előbb felsorolt modellekhez elérhetők különböző szabadon felhasználható *API-k*. A modellekhez egyedi árak tartoznak, melyek a nyelvi modellek esetében ezer *tokenenként*, az audió modellek esetén másodpercenként, a képgeneráló modelleknél képenként van meghatározva.

Számos nagyobb cég beépítette szolgáltatásaiba a modelleket: például a *Bing AI* a *Microsoft GPT-4* alapú böngészést támogató asszisztense (*copilot*), vagy a *Github Copilot*, amely az *OpenAI Codex API-ját* használja, amely a *GPT-3* kódolásra specializált leszármazottja.

2.4.6 Etikai kérdések és közvélemény

Habár az *OpenAI* áttörései jelentősnek bizonyultak, nem minden csoport fogadta azokat felhőtlen lelkesedéssel. Sok kérdést vetett fel a nonprofit státusz profitorientáltra váltása, illetve sokan tartanak attól, hogy lesznek olyanok, akik esetleg nem jó célokra használják fel az egyre csak fejlődő modelleket.

A *ChatGPT* ismert problémás jelensége az úgynevezett *hallucináció*, mely során a modell tényként állít pontatlan, vagy esetleg valótlan információkat. Számomra humoros volt, amikor először belebotlottam, ugyanis megkértem a modellt, hogy írja le nekem a magyar himnuszt. A jólismert első versszak hibátlan volt, azonban a továbbiakban úgy *gondolta* a modell, hogy a tényleges sorok helyett inkább sajátokat ír, Kölcsey Ferenc stílusában. Meglepő volt a hasonlóság, egyébként nem is tűnt fel azonnal, hogy hallucinált sorokat olvastam.

A képgeneráló modellekkel szemben leggyakrabban felvetett aggály, hogy művészek képeit beleegyezés nélkül használták fel tanító adatként, illetve megesett az is, hogy egy generált kép nyert művészeti pályázatot, ami a művészi közösségben nagy felháborodást váltott ki. A szerzői jogok megsértése és a plágium a *ChatGPT* használata kapcsán is felmerült. A mesterséges intelligencia által generált tartalmak szerzői jogi vonatkozásai a közeljövő fontos kérdései. [9]

Jelenleg ez a terület jogilag és etikailag szürke zóna. Nehéz igazságot teremteni a szembenálló érvek között, hiszen mindegyikben van ráció. A generatív MI megítélése vegyes: nagyon sokan nyitottak, nagyon sokan szkeptikusak velük kapcsolatban, egyesek pedig (jogosan) tartanak a társadalmi (főként munkaerő piaci) vonzataitól.

3 A tervezett architektúra bemutatása

Ebben a fejezetben bemutatom a szoftverem architektúráját, és a kialakítására vonatkozó főbb tervezői döntéseket.

3.1.1 Bevezetés

A játéknak a legtöbb funkciója *API* hívásokra épül. A kódfeladatok generálásánál és a megoldások validálásánál *a GPT API-hoz* [10], a feladatok lekérdezésénél, feltöltésénél és módosításánál pedig a szerver *API-hoz* kezdeményez hívást a program.

A tanulmányaim során megtanultam, hogy egy szoftver készítése során a felelősségeket szét kell választani, különben nagyon törékennyé válhat a program. *React* alkalmazásoknál nem igazán szokás háromrétegű architektúráról beszélni. Az úgynevezett *React Hookok* bevezetése előtt inkább a *"Container-Presentational"* tervezési minta volt a bevett szokás, amiben a *Presentational* komponensek állapotmentes osztályok vagy funkcionális komponensek voltak, és kizárólag a dolgok kinézetéért feleltek. A *Container* komponensek pedig az üzleti logikáért feleltek: *API* hívásokat és különböző állapotokat kezeltek. Ez a szétválasztás számos előnnyel járt, például a felhasználói felülettől függetlenül lehetett tesztelni az üzleti logikát és jobban strukturált volt a kód. A *Hookok* bevezetésével azonban ez megváltozott.

A *Hookoknak* [11] köszönhetően használhatóvá váltak bizonyos *React* funkciók a funkcionális komponensekben (amik régebben csak a megjelenésért feleltek), ilyen például az állapotok kezelése. Ez azt jelenti, hogy nem kell életciklussal rendelkező osztályokat írni amiknek a kezelése bonyolultabb mint a funkcionális komponenseké (amellett, hogy több ismétlődő kódot, „*boilerplate*”-et is tartalmaznak), hiszen a *Hookokkal* pontosan ugyanolyan viselkedést érhetünk el az egyszerűbb kezelhetőség mellett.

Az említett újítás által azonban elmosódtak a határvonalak a két réteg között, hiszen egy funkcionális komponens tartalmazhat megjelenésért felelős részeket és ahhoz tartozó üzleti logikát is. A komponensek viszont kisebbek és könnyebben újra felhasználhatók lehetnek. Habár komponensek szintjén nehezebb lett elkülöníteni a felelősségeket, az *SoC* elv még mindig nem sérült meg, hiszen a komponens megjelenítésért felelős része nem tartalmaz ületi logikát.

3.1.2 A szerver-kliens architektúra

Ez a fejezet az architektúra megvalósításának technikai részleteiről szól.

3.1.2.1 Adatbázis

A feladványok eltárolása miatt elengedhetetlen volt egy adatbázis létrehozása. A feladat korai szakaszában volt egy olyan gondolatom, hogy akár lehetne menet közben generálni a feladatokat, és akkor lényegében kifogyhatatlan feladattáram lenne, de ez egyrészt magában rejtett olyan hibákat, hogy ismétlődhetnének a feladatok, vagy esetleg hogy valamiért mégsem rejti el a függvény nevét a *GPT*, ami játszhatatlanná tenné a játékot. Másrészt pedig nem lennének egyenlő esélyek a játékosok között, hogyha nem ugyanazokat a feladatokat kéne megoldaniuk.

Előfordulhat, hogy készítek majd a továbbiakban egy „homokozó" jellegű játékmódot, ahol végtelenül generálódnak az újabb és újabb kódpéldák, de az említett megfontolások miatt inkább létrehoztam egy *MySQL* szervert, aminek a sémája nem túl komplikált, hiszen csak magukat a feladványokat tárolom.

Az adatbázis séma:

```
Tábla neve: Puzzles
Attribútumok:
    ○ name: VARCHAR(45) – a feladvány neve
    ○ language: VARCHAR(45) – programozási nyelv
    ○ difficulty: VARCHAR(45) – a feladvány nehézsége (könnyű/közepes/nehéz)
    ○ prompt: TEXT – milyen prompttal lehet a kódot generáltatni
    ○ code: TEXT – a kódfeladvány tartalma
    ○ explanation: TEXT – a feladvány magyarázata
    ○ solution: TEXT – a feladvány megoldása
    ○ type: VARCHAR(45) – a feladvány típusa
```

A név, nyelv, nehézség szint és kód attribútumok egyértelműek szükségességüket tekintve, azonban a többi fontosságát kifejtem.

A *prompt* attribútumot az egyik játékmódhoz hoztam létre. Az az információ van benne tárolva, hogy az adott kódfeladatot milyen *prompttal* lehet legenerálni. Erről az *Eredmények értékelése* című fejezetben számolok be részletesen.

Az *explanation* attribútumot az egyik ellenőrzéstípushoz használom. Az ellenőrzési módszereket egy későbbi fejezet során fejtem ki, itt csak röviden összefoglalom. Az *explanation* a feladvány *ChatGPT* által írt magyarázata. Ugyanilyen jellegű magyarázatot kell adnia a felhasználóknak a játék során is, ezáltal lehetőségem

van összehasonlítani a két szöveget a *TensorFlow Universal Sentence Encoder* eszközével és koszinusz hasonlóság felhasználásával.

A solution paramétert a megoldás kimenetének tárolására hoztam létre. Ez szintén az ellenőrzésnél használatos, hiszen a *Python* kódokat le lehet futtatni az alkalmazáson belül. Ez azonban nem a felhasználó megoldását ellenőrzi, hanem az én feltételezésemnek az alátámasztására szolgál, miszerint nem kell lefuttatni lokálisan a kódokat, mert a megoldást tökéletesen képes ellenőrizni a *GPT*. Erről az *Ellenőrzési módszerek* fejezetben írok részletesebben.

A *type* paraméter az adott kódfeladat típusát tárolja a játékmódra vonatkozóan. Ennek értékei a következők lehetnek: *Code description, Error finder, Prompt construction, Output.*

3.1.3 Szerver

A felhasználói felület és az adatbázis közötti kommunikációért, valamint néhány üzleti logikai funkcionalitásért a *Node.JS* szerver felelős. A szerver csatlakoztatva van a *MySQL* adatbázishoz, ezáltal képes kiszolgálni a különböző kéréseket.

A különböző *API* végpontok adatbázis műveletekért, *Python* kód futtatásért és szöveg hasonlóság kiszámításáért felelnek, ezeket mutatom be ebben a fejezetben részletesen.

API végpontok:

- GET /API/get-Puzzles: lekéri az adatbázisból az összes feladványt
- GET /API/get-Puzzle: lekéri az adatbázisból a nyelv és nehézségi szint paramétreknek megfelelő feladványokat
- POST /API/save-Puzzle: elmenti az új (paraméterben kapott) feladványt az adatbázisba
- PUT /API/update/:id: frissíti az adott azonosítóval rendelkező feladatot
- DELETE /API/delete/:id: törli az adott azonosítóval rendelkező feladatot az adatbázisból
- POST /API/run-Python: lefuttatja a kapott Python kódot és visszaadja annak eredményét

- POST /API/similarity: kiszámolja és visszaadja a két kapott szöveg hasonlóságát

Egy példa az *SQL* műveletek közül:

```
const query = "INSERT INTO Puzzles (name, language, difficulty, prompt,
code, explanation, solution, type) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
const values = [
  name, language, difficulty, prompt, code, explanation, solution, type
];
db.query(query, values, (error) => {
  console.log(req.body);
  if (error) {
    return res.status(500).json({ error: error.toString() });
  }
  res.status(201).json({ message: "Puzzle saved successfully" });
});
```

Adatbázisok esetében különös veszélyt jelenthetnek az *SQL* injektálást kihasználó támadások. Ennek a lényege, hogy a támadó a lekérdezés kódjába „injektálhat” parancsokat vagy lekérdezéseket, amiket az adatbázis így lefuttathat. A példakódomból látszik, hogy ezt figyelembe vettem, és paraméterezett lekérdezést írtam, ahol a ’?’ karakterek miatt az adatbázis tudni fogja, hogy adatok következnek, nem pedig lefuttatandó kódok (azok tartalmától függetlenül).

A *Python* kódok futtatását úgy végzem, hogy az *API* hívás során érkező kódot beleírom egy ideiglenes fájlba, majd az *exec* paranccsal futtatom azt:

```
const tempFile = path.join(os.tmpdir(), `temp-Python-${Date.now()}.py`);
fs.writeFile(tempFile, PythonCode, (err) => {
  if (err) {
    res.status(500).json({ error: err });
    return;
  }
  exec(`Docker run --rm -v ${tempFile}:/app/temp.py Python-runner Python
/app/temp.py`,
    (error, stdout, stderr) => {
      fs.unlink(tempFile, (err) => {
        if (err) console.error("Error deleting temp file:", err);
      });
      if (error) {
        res.status(500).json({ error: stderr });
      } else {
        console.log(stdout);
        res.json({ output: stdout });
      }
    });
```

Látható, hogy külön-külön *Docker* konténereken belül futtatom a kódokat. Ezt is biztonsági megfontolásból teszem, hiszen rendkívül súlyos következményei lehetnének annak, ha tetszőleges kódokat direktben futtathatnának a felhasználók a szerveren. A *Docker* lehetővé teszi, hogy virtuális környezeteket hozzunk létre, és azokat mindentől elkülönítve használjuk. Ezen felül a *Docker* konténereknek limitálhatók az erőforrásai, így biztosítható, hogy ne használjanak azok túl sokat a szerver erőforrásaiból.

3.1.4 Kliens

Mint azt a *Kutatás során felhasznált technológiák bemutatása* fejezetben említettem, alkalmazásom felhasználó felülete főként a *Material UI* könyvtár elemeit használja, mivel azok eleve alkalmazzák a *Google* tervezési elveit.

Az alkalmazásom fő komponense tartalmazza a weboldal baloldalán megjelenő navigációs menüt, amely mindegyik lapnál látható. Ezenfelül egy *Router* elemet tartalmaz, ami a lapok közötti navigációért felelős. Fontos azonban kiemelni, hogy itt nem tényleges navigációról beszélünk. Sokszor említettem a *React* komponensek modularitását és komponálhatóságát, és a *Routerrel* való navigáció esetében pontosan ezek vannak kihasználva. Lényegében csak **egy darab** oldalunk van, és a „navigáció” során ennek az oldalnak a tartalmát változtatjuk meg különböző komponensekre. A *Router* elemnek két paramétere van, amik azt definiálják, hogy melyik útvonalhoz melyik megjelenítendő komponens tartozzon.

A navigáció:

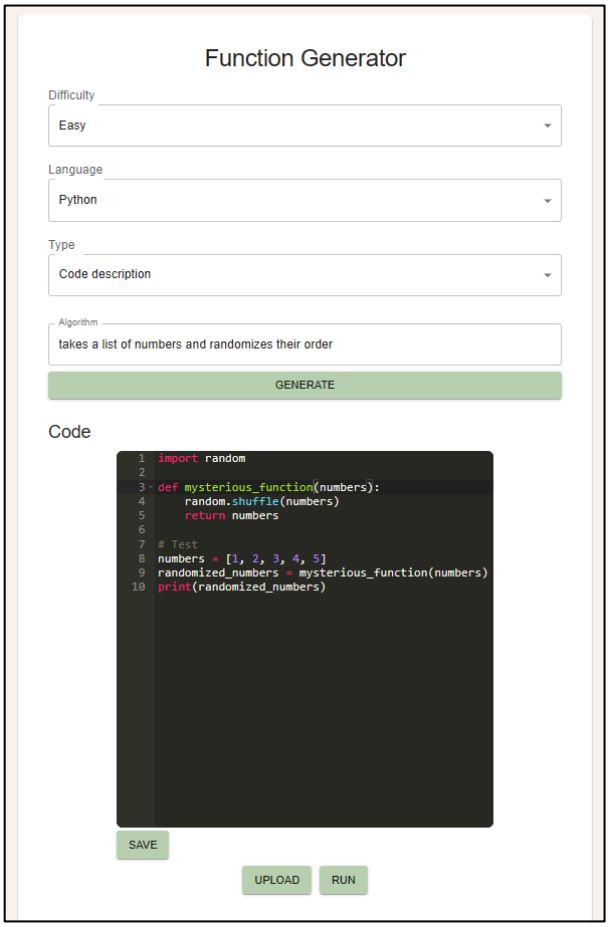
```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/Upload" element={<Upload />} />
  <Route path="/Lobby" element={<Lobby />} />
  <Route path="/Puzzles" element={<Puzzles />} />
  <Route path="/Play" element={<Play />} />
  <Route path="*" element={<ErrorPage />} />
</Routes>
```

3.1.4.1 *Upload Page*

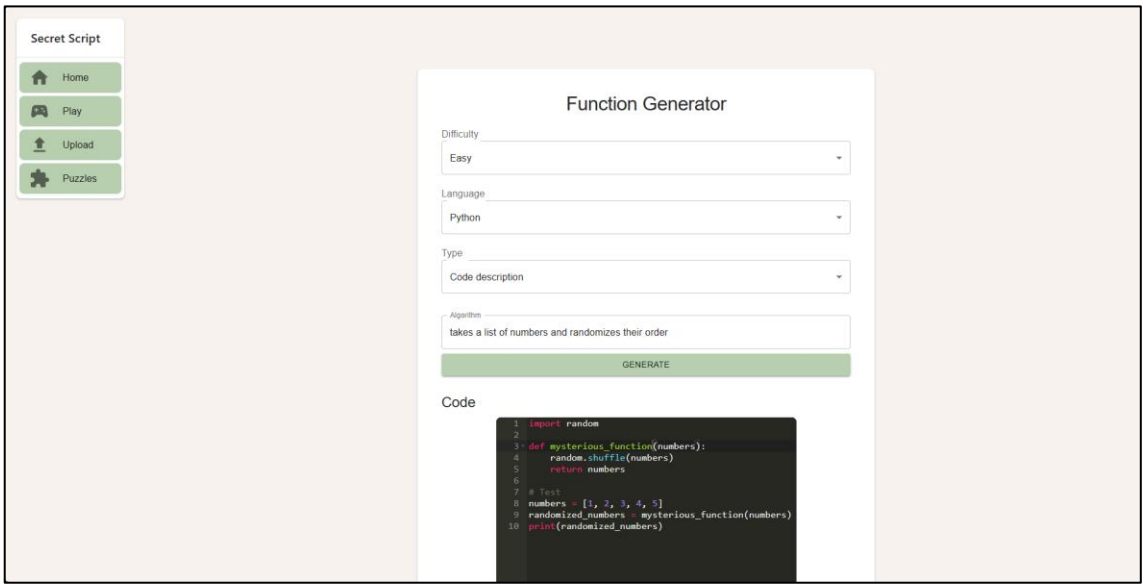
Ezen az oldalon lehet feladatokat generáltatni és feltölteni. Legördülő menüket tartalmaz, amiken kiválasztható a programozási nyelv és a nehézségi szint, valamint egy szövegdoboz, ahova opcionálisan a legenerálni kívánt algoritmust lehet leírni.

A generate gomb hatására *API* hívás indul az *OpenAI GPT API-jához*, és a válasz érkezéséig egy töltőikon jelenik meg, hogy az oldal tanúskodjon arról, hogy megfelelően fut, nem akadt el. A *GPT API* hívásokól a következő fejezetben fogok részletesebben írni.

A megérkezett válasz megjelenik egy kódszerkesztő komponensen, ami egy harmadik féltől származó *AceEditor* [12] nevű komponens. Ez támogatja a kódkiegészítést és a hasonló hasznos fejlesztői környezeti funkciókat. Ha esetleg valaki kiegészítené a kódot, vagy átírna benne valamit, akkor azt szabadon megteheti, és változtatásait elmentheti a *Save* gombbal. A feladat feltöltését az *Upload* gombbal lehet kezdeményezni.



4. ábra A feladatgeneráló komponens



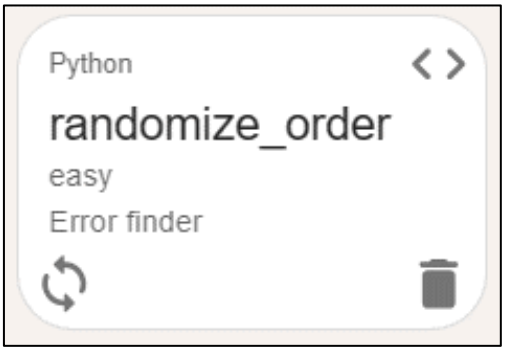
5. ábra Az Upload Page

3.1.4.2 *Puzzle Page*

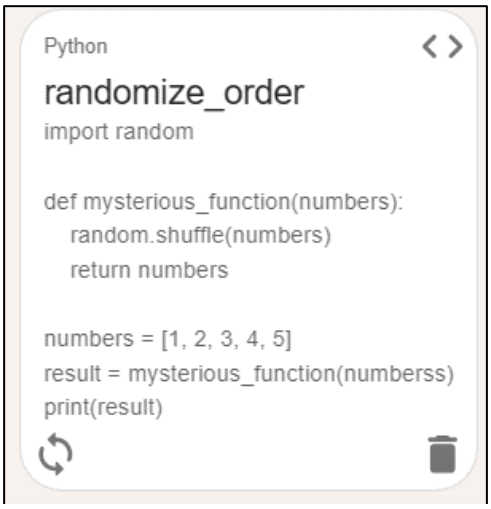
Ez az oldal az adatbázisban tárolt feladatokat listázza, ergonomikus módon, forgatható kártya-szerű megjelenítéssel. Az ötletet a nyelvtanulásom során használt szótanuló kártyácskákról mintáztam, amin a kártya két oldalán egy szó volt leírva két különböző nyelven. Habár ezzel nem áll szándékomban hasonló jellegű kódmemorizálásra buzdítani a felhasználókat, úgy gondoltam, ezáltal egy gusztusos felületet lehetne kialakítani, ahol megtekinthetők, szerkeszthetők és törölhetők a különböző programozási feladatok.

Az oldal betöltéskor egy hívást kezdeményez, amellyel lekéri az összes feladatot, majd elrendezi azokat a felületen, kis kártyácskák formájában. A kártyák a jobb felső sarkukban található kód ikonnal forgathatók meg.

Példa egy kártyára:



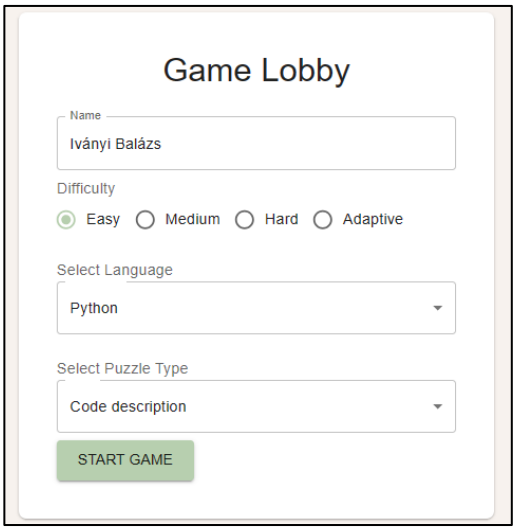
6. ábra A kártya előlapja



7. ábra A kártya hátlapja

3.1.4.3 *Lobby Page*

Ez az oldal a játék előtere. Itt lehet beállítani a játékosnevet, valamint a gyakorolni kívánt programozási nyelvet, a játék nehézségi szintjét és játékmódját. A *Start Game* gombbal lehet elindítani a játékot.



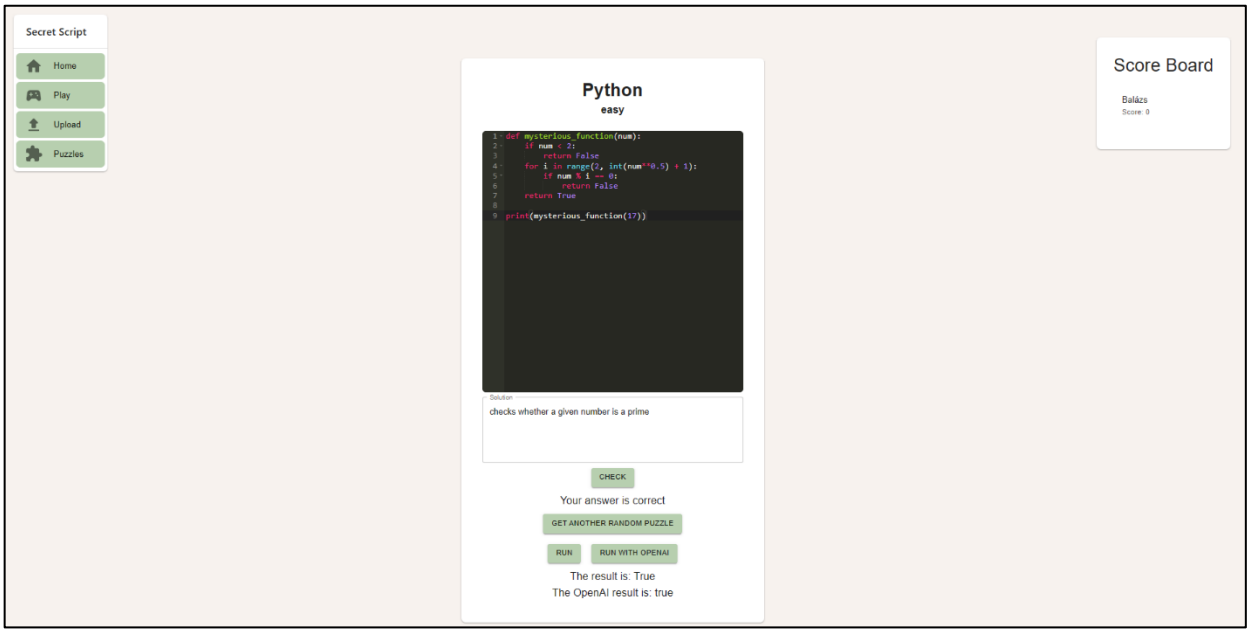
8. ábra A játékbeállító felület

3.1.4.4 *Play Page*

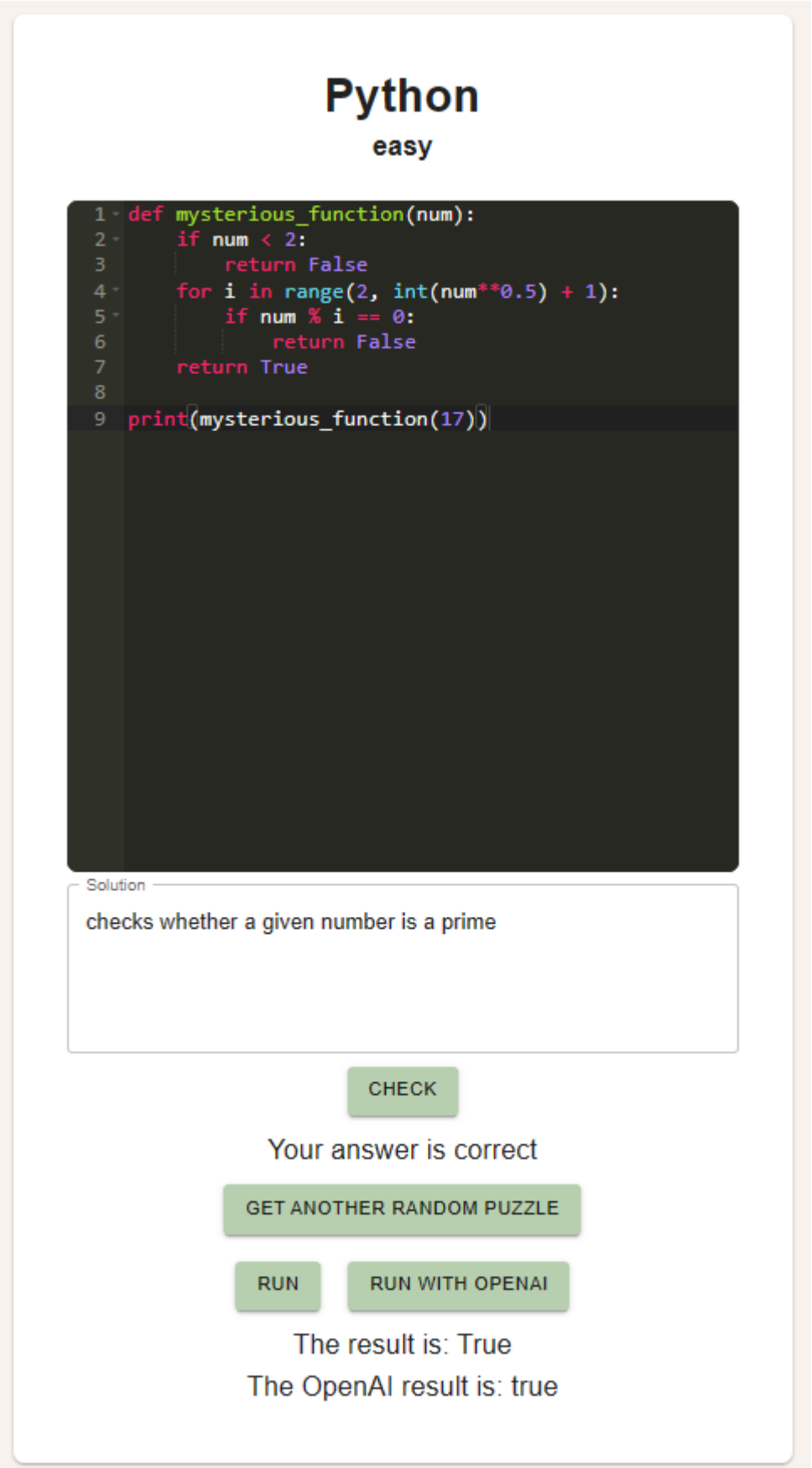
Ez az oldal az alkalmazásomnak a lényege, itt lehet játszani a játékkal. A kódfeladványok az *Upload Page-ben* is használt kódszerkesztő komponensben jelennek meg, azzal a különbséggel, hogy itt nem szerkeszthető annak tartalma. A jobb felső sarokban található a pontozótábla, ezenfelül egy szövegdobozt, a *Check* gombot és *Python* nyelv esetében egy *Run* és *Run with OpenAI* gombot tartalmaz az oldal. Ennek részleteiről az *Ellenőrzési módszerek* fejezetben írok.

A játékosnak a kód értelmezése után a kiválasztott játékmód függvényében meg kell magyaráznia angolul a kód működési mechanizmusát, vagy meg kell találnia a kódban található hibát, vagy ki kell találnia a *promptot*, amivel az adott feladatot lehet legenerálni, vagy ki kell találnia, mit ír ki a kód. Ha a játékos leírta megoldását, akkor a *Check* gombbal ellenőrizheti azt. Gombnyomásra lefut az ellenőrzés és kiértékelődik a megoldás, ami helyes, részben helyes vagy helytelen lehet.

Ha a játékos jó megoldást ér el, és továbbmegy a következő feladatra, akkor a pontozótáblán növekszik a pontszáma.



9. ábra Play Page



10. ábra A játék főkomponense

4 *Promptok*

Az egyik legkritikusabb generatív mesterséges intelligenciával kapcsolatos fogalom az úgynevezett *prompt* fogalma [13], ez magyarul talán utasításként fordítható le, de a továbbiakban csak az angol megfelelőjét fogom használni. Ebben a fejezetben azt mutatom be, hogy mi az a *prompt*, miért is fontos ez a generált tartalom alakításában, és milyen bevett gyakorlatok vannak ezek konstruálása során. Ezenfelül ismertetem, hogy milyen nehézségekbe ütköztem a játékban használt *promptok* kialakítása során és hogyan lehet ezeket elkerülni.

4.1.1 Mi az a *prompt*?

A *prompt* egy generatív MI bemenete, lényegében valamiféle instrukció. Ez az instrukció ismerteti a kontextust a modellel, ami által a kívánt eredményt kaphatjuk. Egy feltett kérdésként/kérésként vagy egy ismertetett témaként érdemes rá tekinteni.

Például egy szöveg alapú MI *promptja* lehet az, hogy „Magyarázd el, hogy mi a Neumann-architektúra”, vagy egy kép alapúé, hogy „Egy számítógépező kiskutya, rajzfilm stílusban”. Az, hogy egy *prompt* mennyire értelmes, vagy mennyire összetett, nagyban befolyásolhatja a kapott eredményt. Habár az utóbbi példa elég abszurd, a *DALL-E 3* modell könnyűszerrel remek eredményt nyújt (leszámítva, hogy az egér a billentyűzet közepébe csatlakozik):

11. ábra *DALL-E* 3-mal készített kép

4.1.2 Miért fontos a *prompt*?

A *promptoknak* kulcsfontosságú szerepük van a generatív mesterséges intelligenciáknál:

- iránymutatás nyújtanak a generáló folyamatokhoz (hogy milyen kontextus irányában induljanak el): egy többértelmű, vagy túl széleskörűen értelmezhető *prompt* olyan végeredményhez vezethet, amely nem igazán kapcsolódik a kívánt témához
- lehetővé teszik, hogy kontrolláljuk a generálandó eredményt: ha részletes, egymáshoz passzoló kulcsszavakból álló leírást alkotunk, akkor hatással lehetünk az eredmény stílusára, tartalmára, vagy akár hangvételére is
- egy megfelelően strukturált *prompt* akár a hatékonyságon is javíthat: csökkentheti a finomhangolás során zajló oda-vissza ugrásokat.

4.1.3 A jó *prompt* ismérvei

Jó *prompt* írásához nem feltétlen szükséges kódolási tapasztalat; kreativitás és némi kitartás is elegendő hozzá. Van azonban néhány jó gyakorlat, amiket érdemes szem előtt tartani a *promptok* megírásakor [14]:

- érdemes kiemelni, hogy melyik információ vagy tartalom a legfontosabb
- strukturálni kell a *prompt*ot: meg kell határozni a szerepeket, megfelelő kontextust kell közölni, majd meg kell adni az instrukciókat
- konkrét, változatos példákat kell bemutatni, a pontosabb eredmények érdekében
- le kell szűkíteni a kimenet terjedelmét, ezáltal elkerülhetők az esetleges pontatlanságok
- a komplex feladatokat, kisebb, egyszerűbb részfeladatokra kell bontani
- érdemes megkérni a modellt saját magának validálására

4.1.4 A *promptok* típusai

- *Direct prompt*: a *Direct prompt*, vagy másnéven *Zero-shot* a legegyszerűbb típusú *prompt*. Az utasításon kívül nem tartalmaz semmilyen magyarázatot. Az utasítást kérdés formájában is meg lehet

fogalmazni, illetve lehetőség van a modellt egy *szerepbe* bújtatni. (A programom esetében ez a szerep lényegében egy programozásoktató.)

- *Promptok* példákkal:
    - *One-shot prompt*: ennek során egy darab konkrét, világos példát nyújtunk a modell számára
    - *Few-* és *multi-shot prompt*: több példát is mutatunk arról, hogy milyen eredményt kívánunk elérni. Mintákat tartalmazó, komplex feladatoknál jobban működik, mint *zero-shot prompt*
- *Chain of Thought (CoT)*: *CoT promptnál* arra próbáljuk rávenni a modellt, hogy elmagyarázza a gondolatmenetét. Komplex feladatok esetében érdemes *few-shot promptolással* kombinálni
- *Zero-shot CoT*: ez lényegében a *Zero-shot prompt* kiegészítve azzal az utasítással, hogy „gondolkodj lépésről-lépésre”

4.1.5 Az általam használt *promptok*

Miután ismertettem a *prompto*kra vonatkozó általános ismereteket, bemutatom, hogy milyen *promptokat* használok a programozási feladatok generáltatására, illetve a megoldások validálására. Ismertetem gyengeségeiket és erősségeiket, és megmutatok néhány továbbfejlesztési lehetőséget.

4.1.5.1 Feladatgenerálás

A feladatok generálására kétféle mód van: az egyik, hogy a felhasználó kiválasztja a feladat típusát, a programozási nyelvet és nehézségi szintet, amilyen jellegű kódpéldát kapni szeretne, és nem ad utasítást az algoritmussal kapcsolatban, a másik lehetőség, hogy kiválasztja a nyelvet és megfogalmazza (angolul) a kódpéldán megjelenő algoritmust, például *„A function, which takes a list of strings as an argument and prints out every second element from the list.”*. A generate gomb hatására lefut az *OpenAI API* hívás.

A *prompt* részei:

- Egy üzenet rendszer szerepben (*system role*): Ezzel lehet meghatározni a modell viselkedését és stílusát. Ebben azt specifikáltam, hogy a modell viselkedjen programozástanárként, aki különböző tapasztalattal rendelkező diákokat oktat. Az a szerepe, hogy programozási kódpéldákat

generál, amiknek a nevét elrejti, hogy ne lehessen azokból a viselkedésükre következtetni. Válaszát érvényes *JavaScript Object Notation (JSON)* formátumban adja meg, amikben a kulcsok: név, nyelv, nehézségi szint, *prompt*, kód, magyarázat és megoldás. A kéréseket például *"Generate a code in {programming language} with {difficulty} difficulty"* formátumban fogja megkapni.

- Egy üzenet felhasználói szerepben (*user role*): Ez tartalmazza a paramétereket tartalmazó, elvégzendő feladatot. Ennek formátuma megegyezik az előbb említett példáéval (hiszen pont ezért specifikáltuk, hogy ezek az üzenetek ráilleszkedjenek).

A *prompt* során nem mutatok példát a kívánt eredményre, és nem kérem a gondolatmenet részletes leírását sem, azonban programozó tanár szerepbe helyezem a modellt, és világos, konkrét utasítást fogalmazok meg számára. Ezekből megállapítható, hogy az általam használt *prompt* egy *Direct prompt*.

4.1.5.2 Megoldásvalidálás

Egy feladatmegoldás validálásának egyik módja, hogy elküldöm a feladatot és a megoldást is a *ChatGPT API* számára, hogy validálja azt. Ez a hívás a következőkből áll:

- Egy üzenet a rendszer szerepében: ezzel az előzőekhez hasonló módon biztosítjuk a kontextust (programozástanár stb.). Meghatározzuk a feladatát, ami a kódrészlethez tartozó megoldásnak az ellenőrzése. A válaszát három lehetőség közül válassza ki: helyes, részben helyes, helytelen, és amennyiben nem helyes a válasz, nyújtson valami egyszerű iránymutatást a megfejtéshez. A kérések formátuma: *"Please check my solution for this code: {code}. My solution is the following: {solution}"*.

- Ez a feladatgenerálásnál leírtak miatt pontosan ugyanolyan, mint az imént vázolt példa.

4.1.6 Felmerülő problémák

Az előzőekben leírtak alapján látható, hogy az alkalmazásnak a lényege tulajdonképpen néhány meglehetősen komplex *prompt*. A feladat kezdetekor nem volt sok ismeretem a helyes *promptírással* kapcsolatban, illetve azzal sem voltam tisztában,

hogy pontosan mik azok a *tokenek*, és mit jelent az ezer *tokenenkénti* 0,002 $-os ár. Ezek miatt nem szedtem szét a kezdetekben az utasításokat.

A megoldásom tervezése közben felmerült bennem a kérdés, hogy hogyan is tudnám a lehető legegyszerűbb formára alakítani az *API* válaszokat, hogy azok különösebb manipuláció nélkül, gyorsan bekerülhessenek az adatbázisba. Azonnal ráleltem a remek, egyszerű megoldásra, ami által *JSON* formátumban kérem el választ, így annak tartalmát könnyűszerrel ki is menthetem a megfelelő változókba. Ezt a megoldást bátran merem ajánlani, ezzel kapcsolatban semelyik modell esetében sem tapasztaltam semmilyen problémát.

A nagyobb gondot inkább az egyéb felállított szabályok okozták: a név kulcshoz a generált függvény valós neve kerüljön, de a tartalmi részben legyen a neve *„mysterious function”*. Ez azért szükséges, hogy a feladatot egyszerűen azonosítani lehessen az adatbázisban, de a játék során ne derüljön ki a megfejtés egy lényegre törő elnevezés miatt. Sok esetben azonban a *prompt* ellenére a válasz kód része tartalmazta az eredeti függvénynevet, és ez gyakorlatilag használhatatlanná tette a játékot. Erre a megoldást az jelentette, hogy több helyen is kiemeltem a kívánt viselkedést, a *prompt* elején is, meg a *prompt* végén is. Másik megoldásként pedig bevált az újabb modellre váltás is.

A *promptok* összetettségének ellenére meglehetősen jól szuperált a *GPT-3.5-turbo* modell is, de mivel a *GPT-4* a komplex utasításokat már jobban kezeli, átváltottam arra (ennek használata kétszer olyan drága, még ezzel is nehéz volt egész szám fölé tornászni az *API* költségeimet).

4.1.7 Javítási lehetőségek

Miután megismertem a nyelvi modellek és a *promptok* működését, rájöttem, hogy jobb eredményeket lehetne elérni több külön *prompt* használatával, amik egy-egy részfeladatra koncentrálnak. Az oda-vissza, cset jellegű kommunikációval pontosítani lehetne a generált függvényeket, és ki lehetne javítani az esetleges hibákat is.

5 Ellenőrzési módszerek

Ebben a fejezetben bemutatom, hogy a játék során milyen módszereket alkalmazok a felhasználók megoldásainak ellenőrzésére.

Az előzőek során ismertettem, hogy milyen erősségei és gyengeségei vannak a *GPT-nek*, és ezeket szem előtt tartva úgy találtam megfelelőnek, hogy teszteljem a rendszert az ellenőrzési feladatkör tekintetében. A megvalósítás kezdeti szakaszaiban csupán az *OpenAI*-ra hagyatkoztam, de kíváncsi voltam, hogy mennyire helytállóak a válaszai, ezért a szerveroldalon felállítottam egy *Dockeren* belül futó *Python* futtatókörnyezetet. Ez azt a célt szolgálja, hogy lefuttatja a *Python* nyelvű szkripteket, és visszaadja a megoldásukat. Mivel a *GPT* is képes a kód értelmezésére és az eredmény megjóslására kevésbé komplex kódok esetében, ezért ennek eredményét könnyen össze lehet hasonlítani a tényleges futási eredménnyel. Azért használom a jóslás kifejezést, mert tényleges kódinterpretáció nem történik (egyébként már vannak *pluginok*, amik ezt is támogatják, például a *CoderPad*).

5.1 *Kódfuttatás*

Ebben a fejezetben bemutatom, hogy hogyan ellenőrzöm az *OpenAI* kódértelmezési képességeit, és hogy miért merek a *Pythontól* különböző nyelvek esetében erre hagyatkozni.

5.1.1 *Python* futtatókörnyezet

Az architektúra fejezetben írtam arról, hogy milyen biztonsági szempontokat kell szem előtt tartani tetszőleges kódok futtatása esetén, és bemutattam, hogy milyen szerepe van a *Dockernek* ebben.

A futtatási hívás kezdeményezésekor létrejön egy új *Docker* konténer a szükséges függőségekkel, valamint egy ideiglenes fájl, ami a futtatandó kódot tartalmazza. A futtatókörnyezet végrehajtja a fájlnak a tartalmát, és visszaadja annak eredményét. A legvégén megszűnik a konténer és az ideiglenes fájl is.

5.1.2 *OpenAI* „futtatás”

A *Run with OpenAI* gomb megnyomására a felület hívást intéz a *GPT API* felé, aminek tartalma egy *prompt*, ami a feladat kódsorait tartalmazza, valamint az utasítást, hogy a *GPT* értelmezze a kódot és adja vissza annak az eredményét.

5.1.3 A futtatási eredmények összehasonlítása

A két futtatás által kapott eredményt egyszerű sztring komparálással összehasonlítom. Amennyiben megegyeznek, meggyőződhetünk róla, hogy a *GPT* megfelelően értelmezte a feladatot is és a *promptot* is.

5.1.4 Tapasztalatok

Meglepő módon minden tesztelt feladatra helyes (*Python interpreterrel* egyező) megoldást produkált az *OpenAI API*. Ez rendkívül szerencsés, mivel a feladat kezdetekor előre éltem ezzel a feltételezéssel, hiszen a *Pythontól* különböző nyelvek esetében eleve csak a többi ellenőrzési módszerre terveztem hagyatkozni, mert nem állítottam fel az összes gyakorolható programozási nyelvhez futtatókörnyezetet.

Fontos megemlíteni, hogy nem volt a kezdetektől fogva hibamentes a módszer. Az eredménnyel nem volt probléma, amikor azt megfelelő formátumban adta vissza a *GPT*. Előfordult azonban, hogy a *JSON* formátumot elhibázta és lehagyta a kapcsos zárójelek valamelyikét, vagy esetleg nem tartalmazott a válasz semmilyen eredményt, mert a kódfeladatban nem volt használat vagy kiíratás. Ezeket a problémákat a feladatkészítő, illetve feladatellenőrző *promptok* átalakításával könnyen tudtam orvosolni, aminek hatására már rendszeresen jó eredményeket produkált az *API*.

Ennek az eredménynek következtében a többi nyelv esetén bátran mertem hagyatkozni a többi ellenőrzési módszerre.

5.2 *Magyarázat ellenőrzése*

Ezeknél a módszereknél a játékos szöveges, angol nyelvű magyarázatát ellenőrzöm az *OpenAI API* felhasználásával.

5.2.1 Vektor alapú összehasonlítás

A vektor alapú összehasonlításnak a lényege, hogy kettő szöveg hasonlóságát meghatározzuk a belőlük képzett vektorok távolságának kiszámolása által. Minél közelebb van két vektor, annál jobban hasonlít a két szöveg.

Ezt a feladatot szerveroldalon hajtom végre a *TensorFlow Universal Sentence Encoder* modell [15] segítségével. Az említett eszköz vektort képez a bemenő szövegekből, amelyekre koszinusz hasonlóságot számolok. Minél közelebb van a hasonlóság 1.0-hoz, annál jobban hasonlít a két szöveg. A két bemenő szöveg a feladat generálásakor kitöltött *explanation* paraméter, valamint a felhasználó megoldása.

Ez a módszer sajnos nem biztosít túl jó ellenőrzést, hiszen a *GPT-nek* jellemzően nagyon részletes, bőbeszédű válaszai vannak még egyszerű kódpéldák esetében is, míg az emberek hajlamosabbak lényegre törő, rövid válaszokat adni. Voltak ötleteim, hogy milyen módon lehetne alkalmazni ezt valamilyen pontozórendszer felhasználásával, például, hogy 75%-os egyezés felett már pozitívan értékelem a választ, azonban ennek tesztelésekor kiderült, hogy hiába egyezik a **jelentése** a magyarázatoknak, attól függetlenül nagy különbözőséget mutathatnak.

5.2.2 *OpenAI* validáció

A másik, nem futtatást alkalmazó ellenőrző módszer úgy működik, hogy hívást kezdeményezek a *GPT API* felé, amely a kódfeladatot és a felhasználó megoldását küldi el. A *prompt* tartalmazza az utasítást, hogy ellenőrizze a feladatra adott magyarázatot a saját kódértelmezési képességeinek felhasználásával.

A hívás helyes, részben helyes, illetve helytelen választ produkál a megoldás függvényében. Nem helyes válaszok esetén a megoldáshoz iránymutatást is küld, azonban jelenleg ez a funkció nincs kivezetve a felhasználói felületre, ugyanis túlságosan sok információt tartalmaznak ezek a válaszok.

5.2.3 Tapasztalatok

Mivel a vektor alapú összehasonlítás sok esetben alacsony hasonlóságot mutat, nem alkalmazom élesben az ellenőrzéseknél, hiszen nem szerettem volna elrontani a játékélményt egy túl szigorú pontozási rendszer bevezetésével.

A *hint*, azaz a segítségnyújtás funkciót egyelőre szintén nem tettem elérhetővé az imént említett okok miatt, azonban ez mindenképpen olyan továbbfejlesztési lehetőség, amellyel az elsők között szeretnék foglalkozni.

A *Pythontól* különböző nyelvek esetében a különböző futtatási környezetek felállításának nehézsége, illetve a *GPT API* remek kódértési képességei miatt csak a legutoljára említett *OpenAI-jal* történő ellenőrzést alkalmazom.

6 Eredmények értékelése

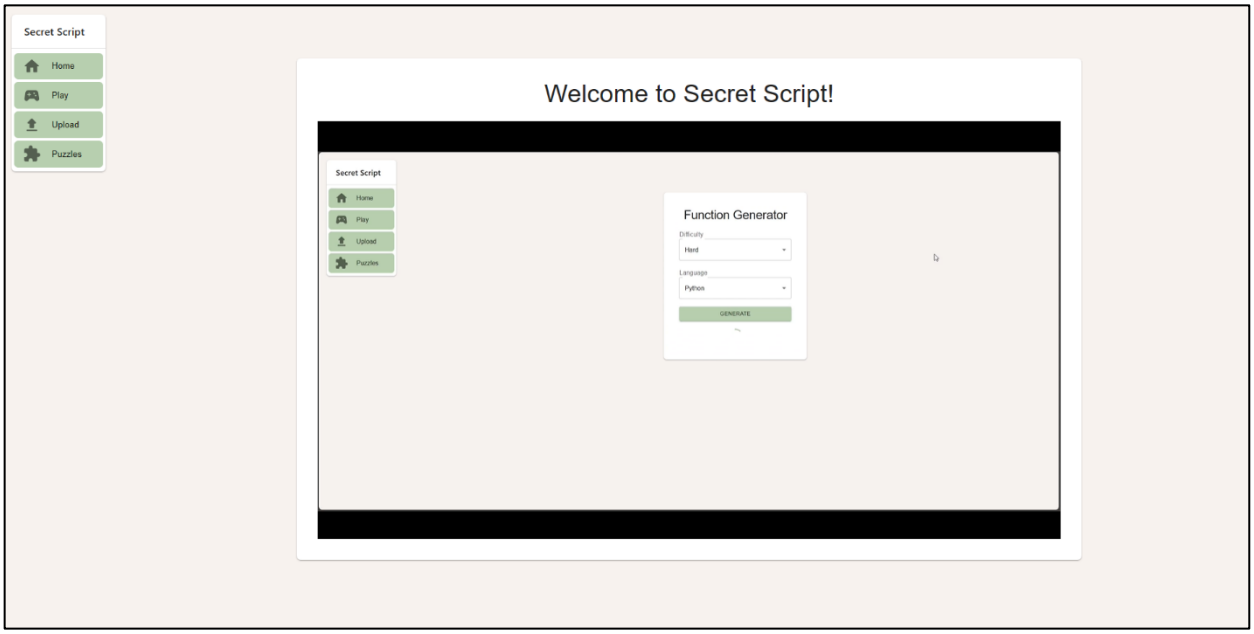
Ebben a fejezetben ismertetem, hogy milyen funkciókra képes a játék az *OpenAI* rendszerén keresztül, és bemutatom, hogyan segíthetik elő az oktatást a hasonló játékos oktatási felületek.

6.1 *Felhasználói felület*

A felhasználói felületet a céljaimnak megfelelően terveztem meg. Igyekeztem olyan letisztult megjelenést kialakítani, aminek középpontjában a feladatok állnak, így a felhasználó figyelme arra összpontosul, egyéb grafikai elemek nem vonják el a figyelmét.

6.1.1 Főoldal (*Home Page*)

A játék főoldalán egy hangtalan, ismétlődő oktatóvideó található, ami alapján megismerhetik a játékosok az összes funkciót, ezáltal átfogó képet nyernek a játékról; ez segíti őket a további használatban is.

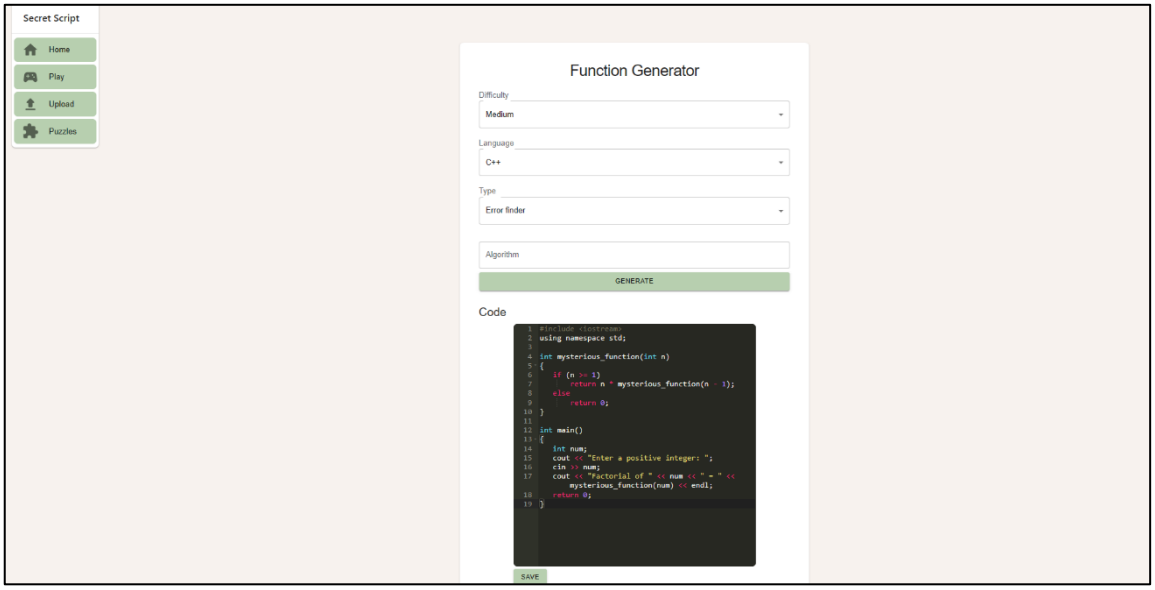


12. ábra Főoldal, középen a *tutoriallal*

6.1.2 Feladatgeneráló oldal (*Upload Page*)

Ezen az oldalon tetszőlegesen lehet bővíteni a kódfeladványok listáját, akár saját kitalált algoritmusokkal is. Amennyiben nem határozzuk meg a kód funkcionalitását, a

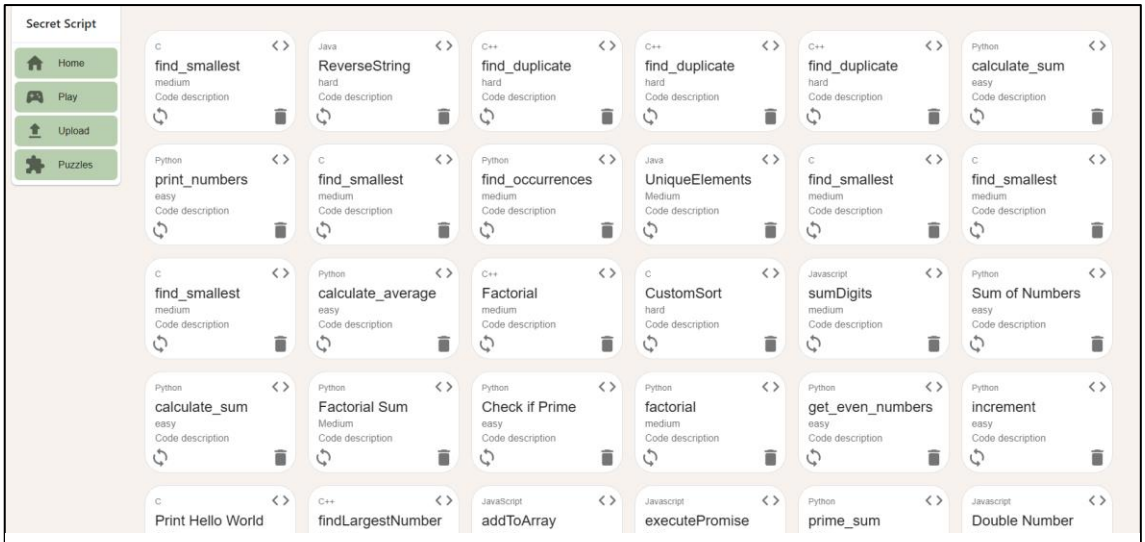
nehézségi szintnek megfelelő kódpélda generálódik, a kiválasztott nyelven, a kiválasztott feladattípushoz A végcél természetesen egy nagyméretű feladatbázis, amely a lehető legtöbb különleges esetet lefedi a nyelvek sajátosságaiból.



13. ábra Feladatgeneráló oldal, amin egy (szándékosan) hibás kód található a hibakeresős feladathoz

6.1.3 Feladatok felsorolása oldal (*Puzzle Page*)

Ezen a felületen interaktív kártyák formájában jelenik meg az összes adatbázisban található feladat. A kártyák elején összegzés található, ami az alapinformációkat tartalmazza: név, nyelv, nehézség, feladattípus. A kártyák hátulján a teljes feladatsor jelenik meg. A kártyákon található ikonokkal a felhasználó módosíthatja, törölheti, illetve megvizsgálhatja a feladatokat.

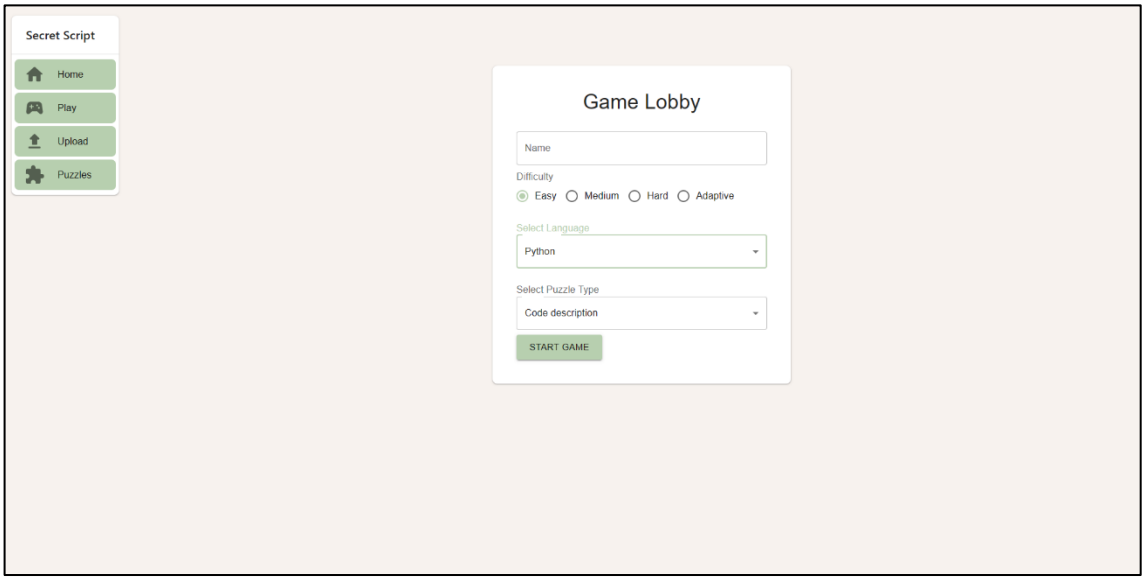


14. ábra A feladatokat listázó oldal

A karbantarthatóság és felhasználóbarát felület kialakítása terén a későbbiekben érdemes lehet egy szűrő és kereső funkciót kialakítani, amik esetleg a különböző attribútumok alapján válogatják szét a feladatokat.

6.1.4 A játék beállítása (*Lobby Page*)

Ezen az oldalon a játékosok konfigurálhatják a játékélményüket. A nyelven és játékmódon kívül a nehézségi szintet is állíthatják. A játékosok választhatják a *C, C++, Java, JavaScript,* valamint *Python* programozási nyelveket, illetve a következő nehézségi szinteket állíthatják be: *easy*, *medium*, *hard*, *adaptive* (könnyű, közepes, nehéz és adaptív). Az adaptív játékmód során a kódpéldák nehézsége 3 helyesen megoldott feladat után növekszik, azonban 1 helytelen esetén csökken. A felhasználó 4 játékmód közül választhat: *Code Description, Error Finder, Prompt Construction, Output* (kódértési feladat, hibakeresős feladat, *prompt engineering* feladat, *„Mit ír ki?”* feladat). A megfelelő konfiguráció kiválasztása után a játékos *Start Game* gombbal indíthatja a játékot.



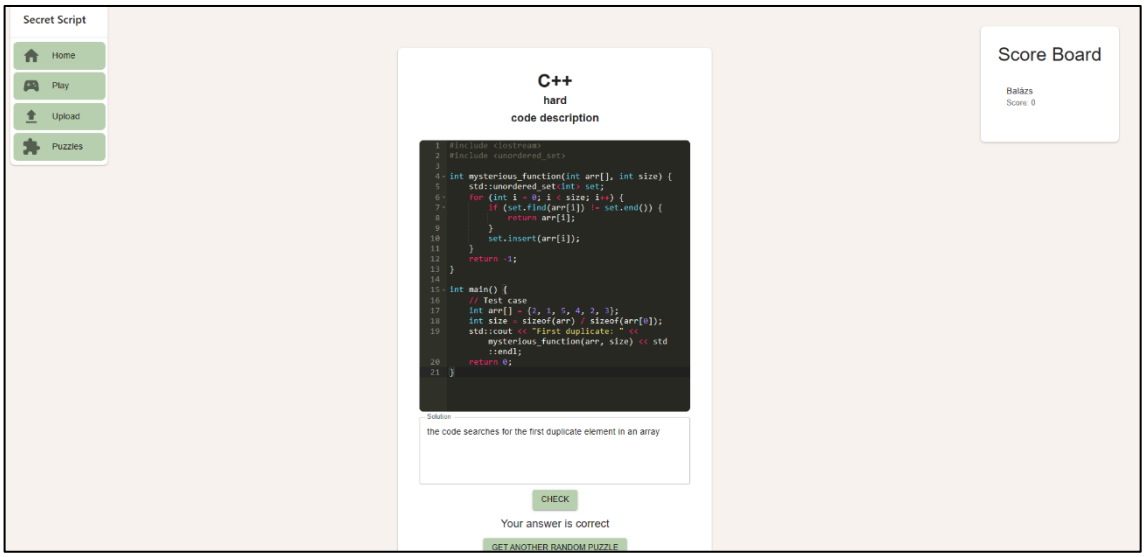
15. ábra A játék kezdeti beállítására szolgáló oldal

6.2 *A játék*

Ebben a fejezetben a különböző játékmódokat ismertetem.

A játékosok négyféle játékmód közül választhatnak, mindegyikkel másféle feladattípusokat gyakorolhatnak be. Mindegyiket bemutatom képekkel, amikre rákattintva megtekinthetők nagyobb méretben a mellékletek között, illetve a nagy ábrákra rákattintva vissza lehet navigálni a korábbi pozícióhoz.

Az első feladattípus a kódértési feladatot gyakoroltatja a felhasználóval. A játékosnak a lehető legpontosabban kell megfogalmaznia, hogy mit csinál a kódrészlet.

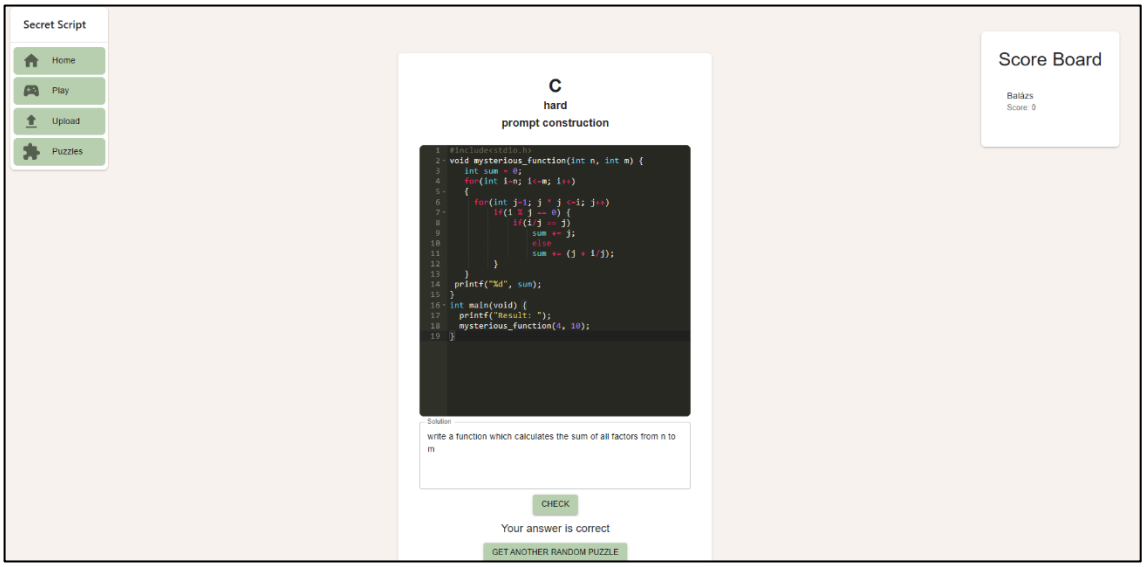


16. ábra A kódelemzős játékmód

A második feladattípus a hibakeresős ZH példák mintájára készült. Valamelyik sorban hiba található, a felhasználó dolga ezt megkeresni és megmagyarázni, hogy miért hibás az adott sor.

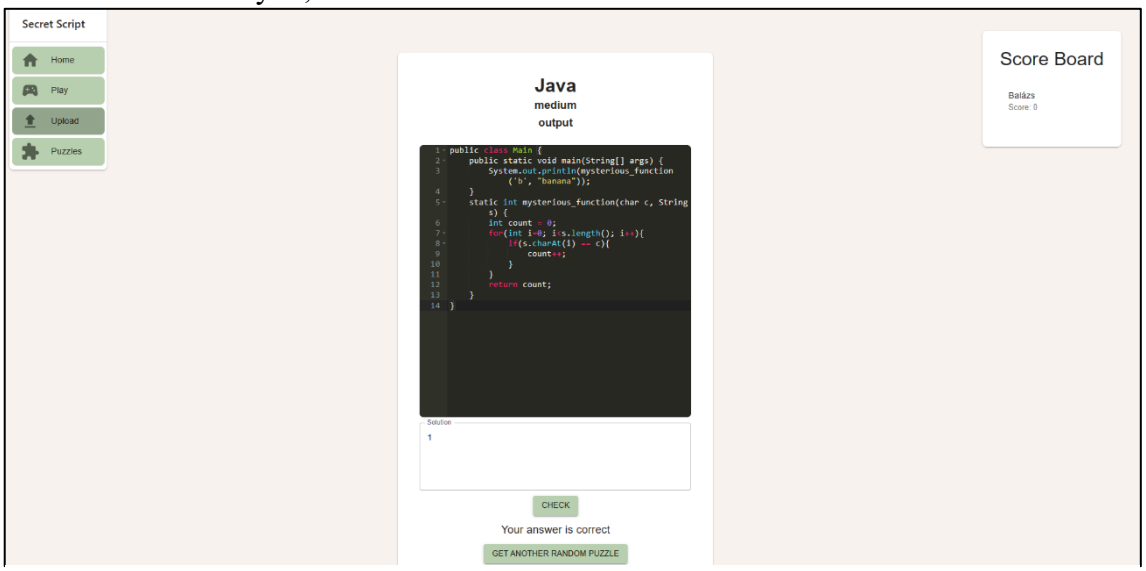
17. ábra A hibakeresős játékmód

A harmadik feladattípus a *prompt engineeringet* gyakoroltatja a játékossal. Habár ez meglehetősen hasonlónak tűnhet az első játékmódhoz, a kettő nem teljesen ugyanaz. Egy bonyolultabb, nem híres (nincsen egy közismert módszer rá, amire könnyedén lehetne hivatkozni, pl. buborékrendezés) algoritmus esetében például kifejezetten nehéz olyan *promptot* írni, ami ugyanolyan kódot produkál, mint a feladat. A dolgozatom témája miatt célszerűnek tűnt egy ilyen jellegű feladattípust is felvenni.



18. ábra A *promptírós* játékmód

A negyedik feladattípust szintén egy tipikus ZH példa inspirálta. Ez a *„Mit ír ki?”* feladat. A játékosnak a feladata megérteni a kódot és megfejtenie, hogy mit fog kiírni. Ezek a feladatok nagyon fontosok, hiszen ezek által lehet igazán megérteni, hogy hogyan működnek az osztályok, a különböző hierarchiák és az interfészek.



19. ábra A *„Mit ír ki?”* játékmód

6.3 *Az oktatás modernizálása*

A motiváció fenntartása rendkívül nehéz feladat, különösképpen repetitív feladatoknál. A játékosítás egy remek módszer ennek áthidalására. Számos alkalommal hiányoltam hasonló jellegű tanulást segítő alkalmazásokat, hiszen sokszor tanulni szerettem volna, de valami könnyebb, nem túl komplikált, akár élvezhető formában.

Számomra a ZH-ra készülés mindig nagyon megterhelő volt, és sosem éreztem úgy, hogy elég lenne a gyakorlásból, különösképpen a programozós tárgyaknál. Úgy gondolom, ez a játék megfelelő gyakorlási lehetőséget biztosít a különböző tudással rendelkező programozók számára, hiszen az egészen könnyű példáktól kezdve az igencsak elgondolkodtatókig mindenféle feladatot lehetséges vele gyakorolni.

A programozás mellett a játékosok *promptírási* képességeiket is fejleszthetik, ami azért nagyon fontos, mert a generatív MI már részét képezi az alap eszköztárnak. Nem gondolom, hogy feltétlenül minden kódsort *ChatGPT-vel* kéne legenerálni, azt pedig főleg nem, hogy a programozási alapok elsajátítása előtt valakinek rá kéne hagyatkoznia, azonban, hogyha megfelelő szaktudással alkalmazza valaki, akkor rendkívül sok időt spórolhat meg.

A felület tanárok számára is hasznos lehet, hiszen így végtelen feladattárat tudnak nyújtani gyakorlásképpen a diákok számára, vagy akár ennek továbbfejlesztésével egy interaktív számonkérő felületet is ki lehetne alakítani.

6.4 *Továbbfejlesztési lehetőségek*

Számos funkciót kitaláltam a projekt elején, amiket idő hiányában nem sikerült megvalósítanom. Ilyen például a *hint*, azaz segítségnyújtás funkció a játékfelületre, valamint egy interaktív oktatómód, ami végigvezeti a felhasználót az alapfunkciókon.

A gyakorolható programozási nyelvek bővítése mellett (amit minden *GPT* által megértett nyelvre ki lehetne terjeszteni), számos játékmóddal ki lehetne egészíteni a programot: például *DALL-E API-val* generált képeknek a „visszafejtésére”, ahol az lenne a felhasználó dolga, hogy kitalálja a képnek a *promptját*, vagy a dolgozatban korábban említett homokozó mód, ahol helyben generált kódpéldákat kell sorra megoldani. Be lehetne vezetni többjátékos módot is, ami szintén rendkívül sok új lehetőséghez vezetne.

7 Összefoglalás

Dolgozatom során bemutattam, hogy a mesterséges intelligencia és néhány közismert keretrendszer felhasználásával létrehozható egy oktatójáték, mellyel tartósan fenntartható a figyelem, és játszva lehet tanulni a különböző programozási nyelveket, valamint a *promptírást*. A tervezési döntéseimnek köszönhetően egy stabil, megbízható rendszert sikerült létrehoznom, melyet könnyedén tovább lehet fejleszteni a robosztus, ugyanakkor rugalmas mivoltának köszönhetően. Robosztus, hiszen fel van készítve a hibák ellen, mint például az *SQL injektálás* vagy az *arbitrary code execution* (tetszőleges, akár rossz szándékú kódfordítás), de emellett rugalmas, hiszen könnyen továbbfejleszthető a *React* modularitása miatt.

Sikeresnek találom a projektemet, hiszen több játékmódot sikerült megvalósítanom, mint a kezdetekben terveztem, és rengeteget tanultam a felhasznált keretrendszerekről, az *OpenAI API-ról*, és a *promptok* megfelelő használatáról is.

Szívesen foglalkoznék továbbra is ezzel a témával és örülnék neki, ha egyaránt hasznosnak bizonyulna a hozzám hasonló helyzetben levő diákok és az oktatók számára.

Rendkívüli változásokat fog hozni az életünkbe az MI, és én szeretnék a részese lenni egy olyan motivált közösségnek, ami nyitott hozzáállással, pozitív célokra használja az MI által nyújtott lehetőségeket.

8 Köszönetnyilvánítás

A kutatás az Európai Unió támogatásával valósult meg, az RRF-2.3.1-21-2022-00004 azonosítójú, Mesterséges Intelligencia Nemzeti Laboratórium projekt keretében.

9 Irodalomjegyzék

[1] J. Desai, „Top 7 Most Used Web Frameworks Among Developers Worldwide 2021,” 2023. [Online]. Available: https://hackernoon.com/top-7-most-used-web-frameworks-among-developers-worldwide-2021-statistics. [Hozzáférés dátuma: 30 10 2023].

[2] „OpenAI,” 2023. [Online]. Available: https://openai.com/. [Hozzáférés dátuma: 30 10 2023].

[3] A. Banks és E. Porcello, „The Virtual DOM,” in *Learning React: functional web development with React and Redux.*, Sebastopol, O'Reilly Media, Inc., 2017, pp. p. 60 - 62.

[4] M. Jhingan, „What is Diffing and Reconciliation in ReactJS?,” 10 6 2022. [Online]. Available: https://hashnode.com/post/what-is-diffing-and-reconciliation-in-reactjs-cl48s2kyo01akr3nv445rc70h. [Hozzáférés dátuma: 30 10 2023].

[5] „MUI,” 2023. [Online]. Available: https://mui.com/. [Hozzáférés dátuma: 30 10 2023].

[6] M. Chand, „Most Popular Databases In The World (2023),” 10 8 2023. [Online]. Available: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/. [Hozzáférés dátuma: 30 10 2023].

[7] F. Ali, „GPT-1 to GPT-4: Each of OpenAI's GPT Models Explained and Compared,” 11 4 2023. [Online]. Available: https://www.makeuseof.com/gpt-models-explained-and-compared/. [Hozzáférés dátuma: 30 10 2023].

[8] R. Goutham Golla, „Here Are Six Practical Use Cases for the New Whisper API,” 6 3 2023. [Online]. Available: https://slator.com/six-practical-use-cases-for-new-whisper-api/. [Hozzáférés dátuma: 30 10 2023].

[9] J. Gill, „FEATURE-As AI-generated art takes off - who really owns it?,” *Reuters,* 7 9 2022.

[10] „GPT models,” 2023. [Online]. Available: https://platform.openai.com/docs/guides/gpt. [Hozzáférés dátuma: 30 10 2023].

[11] T. Adhikary, „React Hooks Fundamentals for Beginners,” 15 3 2022. [Online]. Available: https://www.freecodecamp.org/news/react-hooks-fundamentals/. [Hozzáférés dátuma: 30 10 2023].

[12] J. Hrisho, „securingsincity/react-ace,” 23 5 2020. [Online]. Available: https://github.com/securingsincity/react-ace/commits/master/docs/Ace.md. [Hozzáférés dátuma: 30 10 2023].

[13] „Machine Learning Glossary: Generative AI,” 8 8 2023. [Online]. Available: https://developers.google.com/machine-learning/glossary/generative#prompt. [Hozzáférés dátuma: 30 10 2023].

[14] „Prompt Engineering for Generative AI,” 8 8 2023. [Online]. Available: https://developers.google.com/machine-learning/resources/prompt-eng. [Hozzáférés dátuma: 30 10 2023].

[15] „Universal Sentence Encoder,” 11 10 2023. [Online]. Available: https://www.tensorflow.org/hub/tutorials/semantic_similarity_with_tf_hub_universal_encoder. [Hozzáférés dátuma: 30 10 2023].

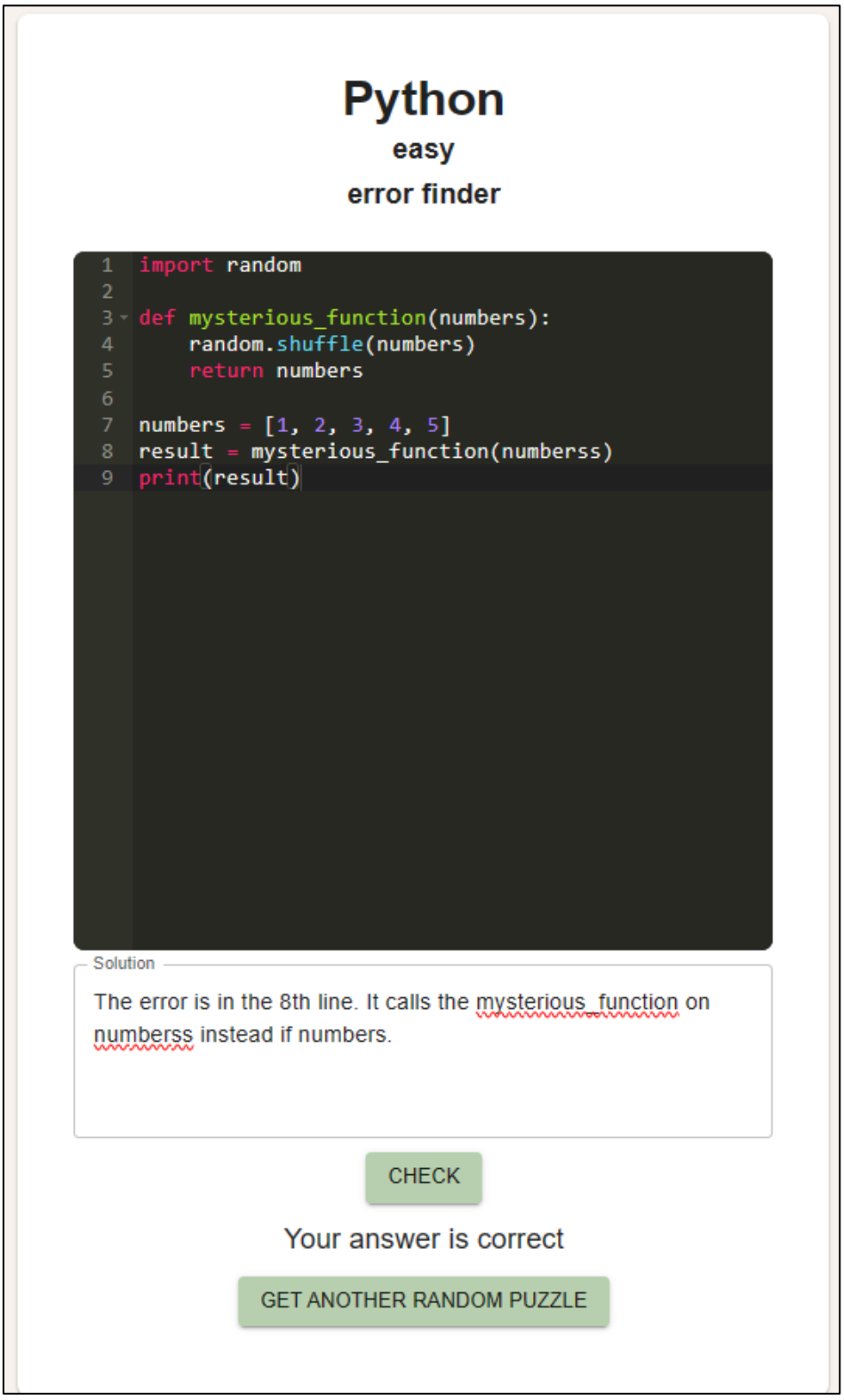
10 Mellékletek

A mellékletek között a játékot demonstráló képernyőképek nagy méretű, jól látható változatai vannak.

10.1 *Kódelemzős játékmód*

20. ábra A kódelemzős játékmód

10.2 *Hibakeresős játékmód*

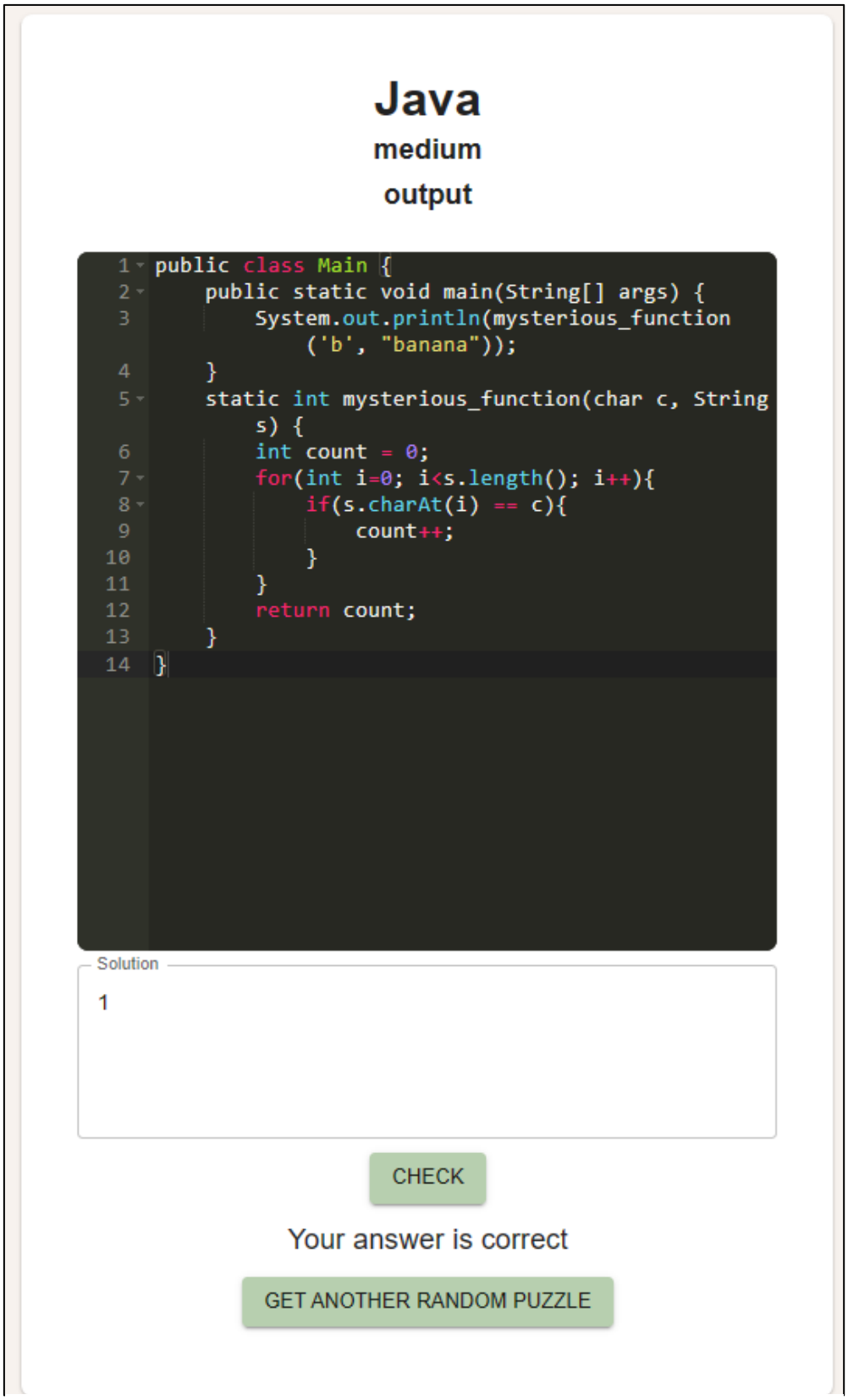


21. ábra A hibakeresős játékmód

10.3 *Promptírós játékmód*

22. ábra A *promptírós* játékmód

10.4 *„Mit ír ki?” játékmód*



23. ábra A *„Mit ír ki?”* játékmód