Budapesti Corvinus Egyetem

A generatív mesterséges intelligencia szerepe és használatának kihívásai a nagyvállalati szoftverfejlesztésben

Informatikai menedzsment szak

Készítette
Szkupien Péter

Konzulens
dr. Drótos György

2025

NYILATKOZAT
GENERATÍV MESTERSÉGES INTELLIGENCIA HASZNÁLATÁRÓL

Szkupien Péter hallgató kijelentem, hogy az *Informatikai menedzsment* szakon 2025/2026/1 félévben beadott szakdolgozatom elkészítéséhez generatív mesterségesintelligencia-rendszert vagy -szolgáltatást használtam az alábbiak szerint:

- A rendszer neve: *OpenAI ChatGPT*, célja: források keresése, összefoglalása
- A rendszer neve: *Anthropic Claude*, célja: források BibTeX bejegyzésének generálása
- A rendszer neve: *Google NotebookLM*, célja: interjúk hanganyagának összefoglalása
- A rendszer neve: *Google Gemini*, célja: szakdolgozat szövegének utólagos szerkesztése

Kijelentem, hogy minden generált tartalmat kritikus szemmel ellenőriztem, és meggyőződtem azok megfelelőségéről, ténybeli pontosságáról és jogszerűségéről.

Tartalomjegyzék

Ábrajegyzék

Táblázatjegyzék

1. fejezet

Bevezetés

Az emberiség fejlődésének történetében megannyi vívmány tekinthető mérföldkőnek. Ha a kellően távoli múltba tekintünk, nagyon különböző mérföldköveket találunk, mint pl. a tűz használata, a kerék feltalálása, a könyvnyomtatás, vagy éppen a gőzgép feltalálása. Noha egyik jelentőségéhez sem férhet kétség, mégis szinte lehetetlen összehasonlítani őket, annyira különböző területeken hoztak áttörést.

Az elmúlt évtizedekben azonban a technológiai fejlődés drámaian felgyorsult: a mérföldkövek már nem évszázadonként, hanem évtizedenként, vagy akár csupán néhány évenként követik egymást, témájukban pedig egyre inkább az informatika köré összpontosulnak. A számítógépek megjelenésétől datált időszak – amelyet nevezhetünk a digitalizáció korának vagy a negyedik ipari forradalomnak is – számos innovációjáról már most biztosan kijelenthető, hogy valóban megváltoztatta az életünket, ilyen pl. a személyi számítógép (PC), az internet, és az okostelefonok elterjedése. Rengeteg olyan is van azonban, amik ugyan hasonló változásokat ígértek, a máról még nem lehet eldönteni, valóban be is váltják-e majd ezeket a várakozásokat, mint pl. a virtuális valóság vagy éppen a blokklánc.

Napjainkban a *mesterséges intelligencia* (MI) az az újdonság, amely lázban tartja a világot (ez nem puszta személyes megfigyelés, a vonatkozó cégek részvényárfolyamainak szárnyalása elég erős indikátor). Ezen belül is a *generatív mesterséges intelligencia* (GMI) a leglátványosabb, hiszen új tartalmakat képes létrehozni. Legyen szó akár szövegről, akár képről, akár videóról, ezek a gombnyomásra, akár ingyenesen létrejövő tartalmak jellegüket tekintve ugyanolyanok, mint azok, amiket eddig kizárólag emberek állítottak elő. Míg a korábbi innovációk döntően csupán az emberi erőt cserélték gépi erőre, vagy a monoton, repetitív feladatokat automatizálták, a generatív mesterséges intelligenciával *látszólag* intellektuális, kreatív vagy akár művészi folyamatokban is lecserélhetővé válik az ember. Adódik tehát a kérdés, hogy hogyan hat ennek a technológiának a megjelenése olyan emberi tevékenységekre, amelyek esetében korábban fel sem merülhetett, hogy automatizáljuk.

Ebbe a sorba tartozik a szoftverfejlesztés is, amely éppen az az intellektuális tevékenység, amely segítségével eddig a többi repetitív folyamatot automatizáltuk. Hiába szoftverfejlesztés eredményei maguk a generatív mesterséges intelligencia szoftverek is, azok nemcsak célként, hanem eszközként is szolgálhatnak a szoftverfejlesztésben, felvetve

a kérdést, hogy hogyan is változtatja meg a generatív mesterséges intelligencia magát a szoftverfejlesztési folyamatot.

Ezt illetően szélsőséges véleményekkel találkozhatunk. Míg egyesek szerint a mesterséges intelligencia megjelenése (és különösen az általa generált kódok) miatt csak még nagyobb szükség lesz szoftverfejlesztőkre, mások szerint a nem is olyan távoli jövőben a mesterséges intelligencia már a szoftverfejlesztők munkáját is el fogja venni, hiszen olcsóbban fog jobb kódokat generálni.

Mivel mérnökinformatikusként magam is szoftverfejlesztőként dolgozom, különösen foglalkoztat a kérdés. Az elmúlt években az az érzés alakult ki bennem, hogy nem lehet nem találkozni ezzel a technológiával, mindenki erről beszél, és öngerjesztő módon mindenki attól fél, hogy kimarad belőle. Az új, látványos eszközök mögött ugyanakkor gyakran nem látszik kristálytisztán a valódi hozzáadott érték, így könnyen megkérdőjelezhető, megalapozottak-e a témát övező hatalmas várakozások. Ennek a kérdésnek a megválaszolása (már ha egyáltalán lehetséges) természetesen jóval túlmutat egy szakdolgozat keretein, mégis egyértelmű volt számomra, hogy ezt a témát szeretném körüljárni.

Szakdolgozatomban azt vizsgálom, hogyan hat a generatív mesterséges intelligencia a szoftverfejlesztésre. A *Software Development Lifecycle* (SDLC) fázisain keresztül elemzem, az egyes fázisokban mi lehet a szerepe ennek az új technológiának, valamint, hogy ehhez képest hol tart a gyakorlatban az alkalmazása. Külön figyelmet fordítok azokra a körülményekre, amelyek relevánsak lehetnek a technológia alkalmazhatóságát illetően, ideértve a potenciális nehézségeket, amelyek meggátolhatják az elméleti felhasználási lehetőségek gyakorlati megvalósulását.

Munkám több ponton is kapcsolódik az Informatikai menedzsment posztgraduális képzés tanmenetéhez. A *Szervezeti információrendszerek* kurzuson külön előadás foglalkozott a mesterséges intelligencia hatásaival (*„Mesterséges intelligencia: Revolution vagy Hype?”*), míg a szoftverfejlesztés a *Software engineering* tárgynak volt témája. A mesterséges intelligencia jogi és biztonsági aspektusait az *Infokommunikációs jog* és az *Informatikai biztonság* kurzusok tárgyalták.

A dolgozat 2. fejezetében áttekintem a téma releváns elméleti ismereteit, a szoftverfejlesztést és a generatív mesterséges intelligenciát, valamint a kettő találkozását, illetve röviden áttekintem a szoftverfejlesztés iparágát. A 3. fejezetben vázolom kutatásom módszertanát, ideértve a kutatási kérdéseket és az adatgyűjtés módját. A 4. fejezetben kutatási kérdésenként részletesen ismertetem az eredményeket. Az 5. fejezetben következtetéseket vonok le a kutatási eredményekből, valamint megfogalmazok egy akciótervet a technológia bevezetésére. Végül a 6. fejezetben összegzem a dolgozatot.

2. fejezet

Elméleti áttekintés

Hosszú út vezetett az első szoftverek megjelenésétől a generatív mesterséges intelligencia térhódításáig. Az elmúlt évtizedekben a szoftverfejlesztés területén drámai változások zajlottak le, amelyek alapjaiban formálták át a szakmát és a fejlesztési folyamatokat. A technológiai fejlődés következtében egyre összetettebb rendszerek épültek ki, amelyek kezelése és fejlesztése új megközelítéseket és eszközöket igényelt.

Ebben a fejezetben összefoglalom a szakdolgozat témájához kapcsolódó elméleti ismereteket. A 2.1. alfejezetben áttekintem a szoftverfejlesztési folyamat lépéseit és fontosabb módszertanait. A 2.2. alfejezetben bemutatom a mesterséges intelligencia főbb területeit, majd a 2.3. alfejezetben ismertetem, hogyan hat a mesterséges intelligencia a szoftverfejlesztésre. Végül a 2.4. alfejezetben röviden áttekintem a szoftveripar főbb jellegzetességeit.

2.1. Szoftverfejlesztés

A számítógépek fejlődésével egyre bonyolultabb problémák kerültek a programozók látókörébe, hiszen a növekvő számítási kapacitás egyre több esetben volt már elegendő. Ez a potenciál nem is maradt kihasználatlanul, ami a szoftverrendszerek komplexitásának drámai növekedéséhez vezetett (Briggs & Nunamaker Jr., 2020). A kezdeti komplexitást jól mutatja, hogy az első elektronikusan tárolt program, amelyet Tom Kilburn írt 1948-ban egy szám legnagyobb valódi osztójának megkeresésére, mindössze 17 utasításból állt (Lavington & Society, 1998). Ezzel szemben a Google összes szoftverének együttes kódbázisát 2 milliárd sorosra becsülik (Potvin & Levenberg, 2016), ami jól mutatja a robbanásszerű fejlődést.

A kevesebb, mint egy évszázad alatt ilyen meredeken növekvő bonyolultságot látva nem szabad azonban szem elől tévesztenünk, hogy az emberi agy kapacitása nem változott. Vagyis a szoftveresen megoldott problémák komplexitása bőven átlépte már azt a határt, amelyet egy ember még részleteiben képes átlátni. Ennek a kezelését az újabb és újabb absztrakciós szintek bevezetése tette lehetővé, hiszen a magasabb absztrakciós szinteken már nem nehezítik a tisztánlátást az alacsonyabb szintek részletei. A növekvő komplexitás, és az abból következő különböző absztrakciós szintek összhangban tartása szükségessé tette

strukturált fejlesztési módszertanok kidolgozását, amelyek segítségével a projektek kézben tarthatók és a csapatok hatékonyan tudnak együttműködni.

2.1.1. Szoftverfejlesztési életciklus

A szoftverfejlesztés során már a korai évtizedekben nyilvánvalóvá vált, hogy a növekvő komplexitás és a hatékony csapatmunka strukturált megközelítést igényel. A kezdeti spontán, ad hoc fejlesztés gyakran vezetett időbeli csúszásokhoz, költségtúllépéshez és minőségi problémákhoz. Ezek a nehézségek hívták életre a szoftverfejlesztés első modelljét (vízesésmodell), amely strukturált fázisokra bontotta a folyamatot (Royce, 1970). Később ez alapján dolgozták ki a *szoftverfejlesztési életciklus* (Software Development Life Cycle, SDLC) koncepcióját, amely általános keretrendszert ad a szoftverek tervezéséhez, fejlesztéséhez, teszteléséhez és karbantartásához (Boehm, 1988; Sommerville, 2015). Az SDLC célja, hogy a fejlesztés folyamata átlátható, megismételhető, hatékony és mérhető legyen, hozzájárulva ezzel ahhoz, hogy az elkészült termék végül megfeleljen a megrendelői és felhasználói elvárásoknak.

A szoftverfejlesztési életciklus modellje tehát nemcsak technikai irányú, hanem menedzsment eszköz is: közös nyelvet biztosít a szoftverfejlesztés folyamatához, elősegítve a kommunikációt a különböző szerepkörök között, támogatja a tervezést és a minőségbiztosítást, valamint csökkenti a projektkockázatokat (Hossain, 2023). A jól definiált fázisok segítenek abban, hogy a fejlesztési folyamat logikusan épüljön fel, és minden lépésnek világos bemenetei és kimenetei legyenek. Bár az egyes szervezetek és módszertanok eltérően valósítják meg, az SDLC alapvetően a következő lépésekből áll (IBM, é.n.-b).

1. **Projekttervezés (Planning).** Célja a projekt céljainak, hatókörének, erőforrásigényének és kockázatainak meghatározása. Ebben a szakaszban történik a projekt ütemezése és a kezdeti költségbecslés is, amik alapot adnak a további fejlesztési döntésekhez. Eredménye a projektterv és a *kezdeti* szoftverkövetelmény specifikáció (Software Requirement Specification, SRS).

2. **Elemzés (Analysis).** A fejlesztendő rendszer funkcionális és extrafunkcionális (nem funkcionális) követelményeinek összegyűjtése, elemzése és dokumentálása. A cél, hogy minden érintett fél számára egyértelmű legyen, mit kell a rendszernek teljesítenie. Eredménye a követelmények részletes dokumentációja.

3. **Rendszertervezés (Design).** A rendszer logikai és technikai architektúrájának kialakítása, beleértve az adatmodelleket, a komponensek közötti kapcsolatokat, interfészeket és a felhasználói felület alapvető struktúráját. Az átgondolt tervezés biztosítja, hogy az implementáció során már egyértelmű legyen, mit is kell csinálni. Eredménye a szoftverterv dokumentáció (Software Design Document, SDD).

4. **Fejlesztés (Development).** A szoftver tényleges megvalósítása (kódolása) a korábbi fázisok során keletkezett dokumentumok alapján, vagyis a forráskód elkészítése, a komponensek integrálása és bizonyos előzetes egységtesztek végrehajtása. Eredménye a szoftver egy funkcionális (működő) prototípusa.

5. **Tesztelés (Testing).** A fejlesztett rendszer validálása, amely során ellenőrzik, hogy az a tervezett követelményeknek megfelelően működik-e. Számos különböző módszer szolgál a hibák azonosítására, mint pl. statikus kódanalízis, code review, különböző manuális/automata tesztek (egységteszt, integrációs teszt, rendszerteszt), sérülékenységvizsgálat. A hibák azonosítása és dokumentálása után természetesen a javításuk következik, egészen addig, amíg az újratesztelés sikerrel nem jár. Eredménye egy javított, jobb minőségű (ideális esetben akár hibamentes) szoftver.

6. **Bevezetés (Deployment).** A kész rendszer éles környezetbe helyezése, ahol már hozzáférnek a tényleges végfelhasználók. A technikai bevezetésen túl ide tartozik annak a biztosítása is, hogy a felhasználók valóban értsék, hogyan kell használniuk az új rendszert, illetve, hogy a bevezetés a lehető legkevésbé akassza meg a meglévő folyamatokat. Eredménye egy olyan szoftver, amely már a cégfelhasználók számára is elérhető.

7. **Karbantartás (Maintenance).** A rendszer hosszú távú támogatása garantálja a szoftver folyamatos működőképességét és alkalmazkodását a változó üzleti igényekhez. Ez magában foglalja frissítések és hibajavítások biztosítását, valamint akár új funkciók fejlesztését is. Eredménye egy frissebb, javított szoftver.

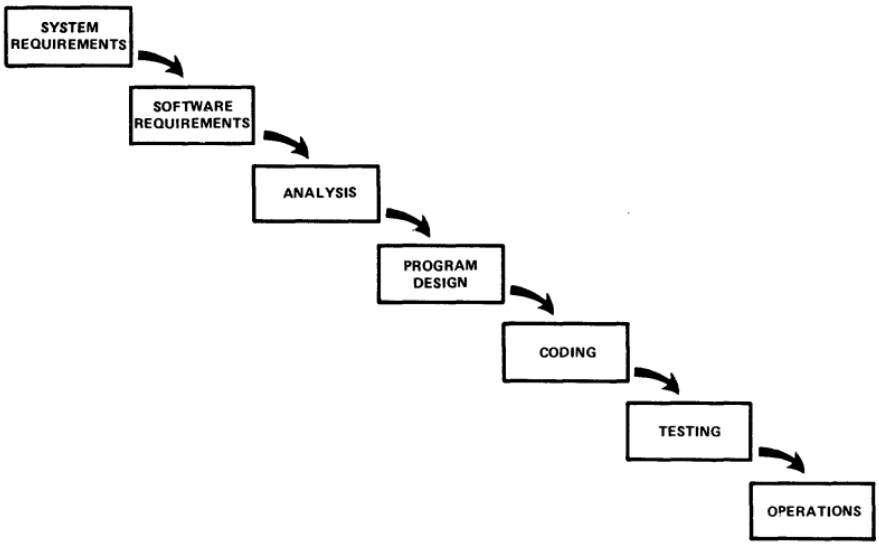
A fázisok egymásra épülnek, a különböző fejlesztési modellek azonban már eltérően értelmezhetik a fázisok közti átmeneteket. Míg a legegyszerűbb megközelítés szerint a fázisok lineárisan követik egymást, más modellek szerint iteratívan is végrehajthatók. Látható tehát, hogy az SDLC csupán testre szabható közös alapot teremt a számos különböző modell számára, amelyek így egységesen elemezhetők és összehasonlíthatók (Alazzawi és mtsai., 2023).

2.1.2. Klasszikus modellek

A szoftverfejlesztés első modellje a *vízesésmodell* (2.1. ábra, Royce, 1970), amelyben a diszjunkt fázisok szigorúan szekvenciálisan követik egymást. Ez a modell egyszerű és átlátható, a fázisok közti átmenetek, valamint a be- és kimenetek egyértelműek. Fontos azonban megjegyezni, hogy a vízesésmodell nem tudja hatékonyan kezelni a követelmények utólagos változását, hiszen ekkor elölről kellene kezdeni az egész folyamatot.

Noha a vízesésmodellt tekintjük az első formálisan leírt modellnek, az *iteratív* megközelítés már ennél korábban is megfogalmazódott (Zurcher és Randell, 1968). Ebben a szemléletben a fázisok között visszacsatolás van, vagyis a rendszer több iteráció során válik egyre részletesebbé, míg el nem nyeri végleges formáját. (Valójában Royce (1970) már a vízesésmodellt bemutató cikkében is írt visszacsatolásról (*„do it twice”*), de ez a modell mégis szigorúan szekvenciálissá egyszerűsítve terjedt el.)

A szekvenciális és iteratív megközelítések ötvözéséből született meg a *spirálmodell*, amely iteratív ciklusokban dolgozik és külön hangsúlyt fektet a kockázatelemzésre minden egyes fázisban (2.2. ábra, Boehm, 1988). A spirálban minden „gyűrű” egy újabb fejlesztési ciklust jelöl, amelyben célokat határoznak meg, értékelik a rizikófaktorokat, prototípu-



2.1. ábra. A vízesésmodell lépései. (Royce, 1970)

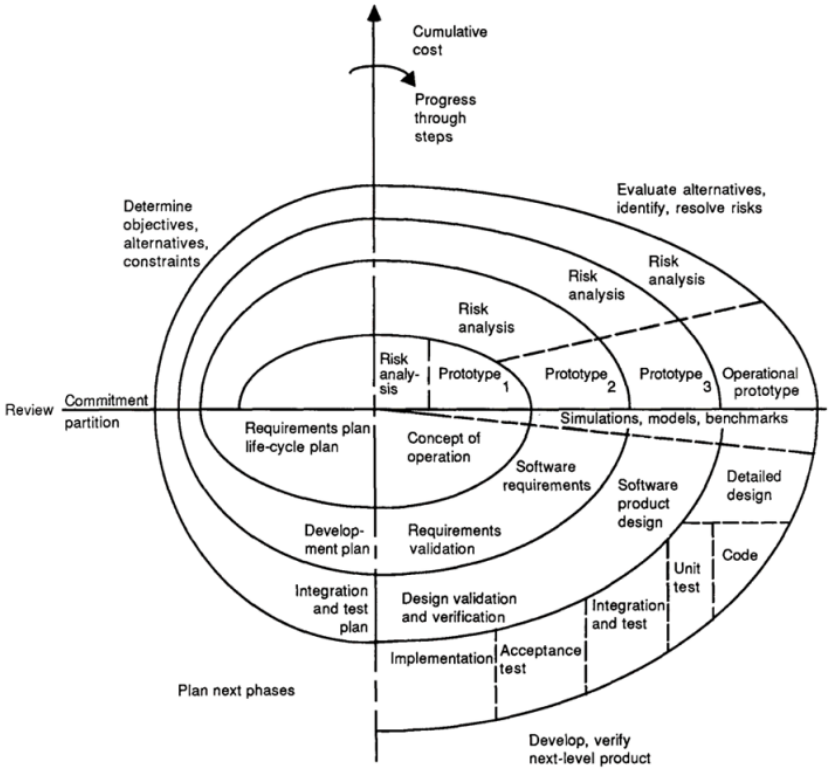
sokat fejlesztenek, majd megtervezik a következő ciklust. Ez a modell különösen olyan összetett projektekben alkalmazható, ahol a követelmények gyakran változnak, vagy magas a kockázat.

A vízesésmodell továbbfejlesztése a *V-modell* (2.3. ábra, Rook, 1986), amelyben különös hangsúlyt kap a tesztelés, hiszen minden fejlesztési szinthez kapcsolódik egy tesztelési fázis is. A V bal szára az egyre részletesebb tervezési lépésekből áll (top-down), amelyeket alul a tényleges implementáció követ. A V jobb szára pedig az egyre magasabb szintű validációs fázisokat tartalmazza (bottom-up). Előnye, hogy az egyes fejlesztési lépésekkel párhuzamosan tervezhetők a kapcsolódó tesztek, ugyanakkor nem elég rugalmas ahhoz, hogy kezelni tudja a változó követelményeket.

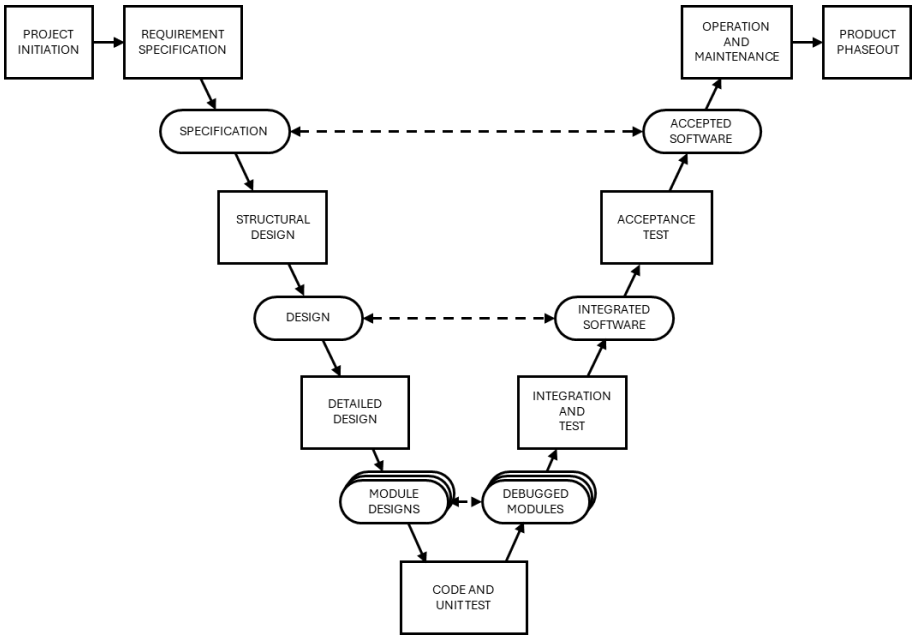
2.1.3. Agilis módszertanok

A klasszikus, szekvenciális fejlesztési modellek (például a vízesés- és a V-modell) jól strukturáltak, de a gyakorlatban gyakran bizonyultak túlságosan merevnek a gyorsan változó üzleti és technológiai környezetben. A követelmények ritkán maradnak változatlanok egy teljes projektidőszakon át, és a felhasználói igények sokszor csak a fejlesztés előrehaladtával tisztulnak ki. Ezt a problémát a korai modellek nehezen tudták kezelni, mivel a folyamat lineáris jellege miatt egy későbbi fázisban felmerülő változás az egész fejlesztési ciklust visszavethette. Ebből a felismerésből született meg az *agilis szoftverfejlesztés* gondolata (előzményének tekinthető a *Rapid Application Development (RAD)* és a *software prototyping*), amely a rugalmasságot, az iterativitást és a folyamatos visszacsatolást helyezi a középpontba (Beck és mtsai., 2001; Highsmith, 2002).

Az agilis szemlélet 2001-ben vált formálisan meghatározottá, amikor 17 szoftverfejlesztő megfogalmazta az *Agile Manifestót* (*Manifesto for Agile Software Development*, Beck és mtsai., 2001). A dokumentum négy alapértéket és tizenkét elvet rögzít, amelyek célja a fejlesztés emberközpontúbbá, együttműködőbbé és gyorsabban reagálóvá tétele.



2.2. ábra. A spirálmodell lépései. (Boehm, 1988)



2.3. ábra. A V-modell lépései. (Rook (1986) ábrája feljavított minőségben, saját szerkesztés.)

A négy alapérték a következő:

- az **egyének és interakciók** fontosabbak, mint a folyamatok és eszközök,
- a **működő szoftver** fontosabb, mint az átfogó dokumentáció,
- a **megrendelővel való együttműködés** fontosabb, mint a szerződéses tárgyalás,
- a **változásra való reagálás** fontosabb, mint a terv követése.

Ezek az elvek nem a dokumentáció vagy a tervezés elhagyását jelentik, hanem azoknak az emberi tényezők és a gyors visszacsatolás mögé rendelését. Az agilitás tehát nem a folyamatok hiányát, hanem azok ésszerű minimalizálását és adaptivitását jelenti.

Az agilis módszertanok közül a legismertebb a *Scrum*, amely iteratív fejlesztési ciklusokat (ún. sprinteket) alkalmaz, jellemzően 2–4 hetes időtartamban. Minden sprint végén egy működő szoftververzió (inkrementum) kerül bemutatásra, amelyet a csapat retrospektív megbeszélésen értékel, így a következő iterációban azonnal érvényesíthetők a tapasztalatok (Schwaber & Sutherland, 1997).

A *Kanban* módszer ezzel szemben a feladatok folyamatos elvégzésére és a vizuális feladatkövetésre épít: a munkaelemek egy táblán haladnak végig a „to do” – „in progress” – „done” állapotokon, elősegítve az átláthatóságot és a szűk keresztmetszetek felismerését (Anderson, 2010).

Az *Extreme Programming* (XP) a kódminőség és a fejlesztői gyakorlatok javítását helyezi előtérbe, például páros programozással (pair programming), tesztvezérelt fejlesztéssel (Test Driven Development, TDD) valamint folyamatos integrációval és folyamatos szállítással (Continuous Integration / Continuous Delivery, CI/CD) (Beck, 2004).

Az agilis módszertanok nem önmagukban állnak, hanem az SDLC iteratív megvalósításai: az életciklus fázisai itt nem szigorú sorrendben követik egymást, hanem folyamatosan ismétlődnek kisebb körökben. A hangsúly a folyamatos értékteremtésen, a csapat autonómiáján és a visszajelzések gyors beépítésén van. Ennek köszönhetően az agilis fejlesztés különösen jól illeszkedik a gyorsan változó üzleti környezethez és a modern technológiai ökoszisztémákhoz.

2.1.4. Automatizációs trendek

Az agilis fejlesztés térnyerésével párhuzamosan a szoftverfejlesztésben megjelent egy új szemlélet, amely a fejlesztési és üzemeltetési tevékenységek szoros integrációjára épül: ez a *DevOps*. A kifejezés a *Development* és *Operations* szavak összevonásából származik, és egy olyan kulturális és technológiai megközelítést takar, amely az együttműködést, az automatizációt és a folyamatos visszajelzést helyezi előtérbe (Humble & Farley, 2010; Kim és mtsai., 2016). A DevOps célja, hogy megszüntesse a fejlesztői és az üzemeltetési csapatok közti hagyományos szakadékot, ezáltal gyorsabb, megbízhatóbb és skálázhatóbb szoftverszállítást tegyen lehetővé.

A DevOps egyik legfontosabb alapelve a *folyamatos integráció és folyamatos szállítás* (CI/CD), amely az automatizációs eszközök segítségével biztosítja, hogy a kódmódosítá-

sok rendszeresen, automatizált módon épüljenek be a központi kódbázisba, majd tesztelés után akár éles környezetbe is kerülhessenek (Fowler, 2006). Az automatizált build- és tesztfolyamatokat gyakran olyan eszközök valósítják meg, mint a *Jenkins*[1], a *GitLab CI/CD*[2] vagy a *GitHub Actions*[3], amelyek lehetővé teszik a pipeline-ok vizuális konfigurálását, a verziókezeléssel való integrációt és a különböző környezetekbe történő automatikus telepítést.

A konténerizáció és az infrastruktúra automatizálása szintén kulcsszerepet játszanak a modern DevOps-gyakorlatban. A *Docker*[4] a fejlesztők számára biztosít egységes futtatási környezetet, amely minimalizálja a „works on my machine" típusú hibákat, míg a *Kubernetes*[5] a konténerizált alkalmazások automatikus ütemezését, skálázását és monitorozását végzi el (Hightower és mtsai., 2019). Az infrastruktúra leírását kód formájában (Infrastructure as Code, IaC) olyan eszközök támogatják, mint a *Terraform*[6] és az *Ansible*[7], amelyek deklaratív módon teszik lehetővé a rendszerek konfigurációját és újrakonstruálását (Brikman, 2022). Ezek az automatizációs trendek együttesen nemcsak a fejlesztés sebességét növelik, hanem hozzájárulnak a hibák korai felismeréséhez, a rendszerek megbízhatóságának növeléséhez és a *folyamatos fejlesztési ciklus* (Continuous Improvement) megvalósításához.

Összességében a DevOps és a CI/CD megközelítések az agilis elvek technológiai kiterjesztései, amelyek a gyors alkalmazkodást és a folyamatos értékteremtést támogatják. Az automatizáció ma már nem csupán kényelmi eszköz, hanem versenyképességi tényező a vállalati szoftverfejlesztésben, különösen a komplex rendszerek és a felhőalapú architektúrák korában.

2.1.5. Low-code, no-code paradigma

A szoftverfejlesztés folyamatosan az automatizáció irányába mozdul el: a DevOps- és CI/CD-megközelítések az üzemeltetés és a szállítás folyamatát egyszerűsítik, míg a legújabb trendek magát a fejlesztési munkát is igyekeznek automatizálni. Ennek kulcsa az *absztrakciós szint* növelése, amiben az *objektumorientált programozás* (object-oriented programming, OOP) tekinthető a *low-code* és *no-code* paradigma előzményének. Utóbbiak célja, hogy a fejlesztők – vagy akár fejlesztői háttérrel nem rendelkező üzleti felhasználók – vizuális, deklaratív eszközök segítségével hozhassanak létre alkalmazásokat (Richardson & Rymer, 2014). A *low-code* megközelítés még igényel bizonyos mértékű kódolási tevékenységet, míg a *no-code* platformok teljesen grafikus, drag-and-drop alapú környezetet biztosítanak.

A low-code platformok tipikusan előre definiált komponenseket, adatkapcsolatokat és felhasználói felületi elemeket kínálnak, amelyekből gyorsan összeállíthatók alkalmazások.

[1] https://www.jenkins.io/
[2] https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/
[3] https://github.com/features/actions
[4] https://www.docker.com/
[5] https://kubernetes.io/
[6] https://www.terraform.io/
[7] https://www.ansible.com/

Az ilyen környezetek célja a fejlesztés felgyorsítása, a hibák csökkentése és az üzleti oldalon jelentkező igények gyorsabb kielégítése. Az olyan megoldások, mint az *OutSystems*[8], a *Mendix*[9] vagy a *Microsoft Power Apps*[10], jól példázzák ezt a tendenciát: mindhárom platform lehetővé teszi alkalmazások gyors prototípusának elkészítését, adatkapcsolatok beállítását, valamint integrációt külső rendszerekkel. A no-code megközelítés ezt továbbviszi azáltal, hogy fejlesztői tudás nélkül is használható környezetet kínál, ilyen például a *Bubble*[11] vagy a *Google AppSheet*[12].

A low-code és no-code rendszerek jelentősége a vállalati környezetben folyamatosan növekszik, különösen ott, ahol az informatikai osztályok kapacitása korlátozott, de az üzleti igények gyors megvalósítást követelnek. Bár ezek a platformok nem alkalmasak minden fejlesztési feladatra, hatékonyan kiegészítik a hagyományos fejlesztést: lehetővé teszik az egyszerűbb alkalmazások és belső eszközök gyors előállítását, így a fejlesztők nagyobb figyelmet fordíthatnak az összetettebb problémákra. A low-code/no-code irányzat ezért a szoftverfejlesztés demokratizálásának egyik kulcstényezője, és előkészíti a terepet a generatív mesterséges intelligencián alapuló kódgenerátorok által támogatott fejlesztési megoldások számára (Matvitskyy és mtsai., 2023).

2.2. Mesterséges intelligencia

A *mesterséges intelligencia* (MI, artificial intelligence, AI) fogalma eltér a köznyelvben és a tudományos diskurzusban. A hétköznapi kommunikációban az MI gyakran bármilyen automatizált vagy „okos” működésű rendszert jelöl – például egy alkalmazást, egy chatbotot vagy egy autóban működő navigációs rendszert. A tudományos definíciók ezzel szemben szűkebbek és pontosabbak: mesterséges intelligenciának azokat a számítógépes rendszereket tekintjük, amelyek képesek olyan kognitív funkciókat utánzó műveletekre, mint az érvelés, a tanulás, az érzékelés vagy a döntéshozatal (Russell & Norvig, 2021).

A definíció tisztázatlansága régóta problémát jelent az információrendszerek kutatásában is. Egy 2005 és 2020 között megjelent cikkeket feldolgozó szisztematikus irodalomáttekintés szerint a mesterséges intelligencia kifejezést az információrendszerek szakirodalma sokszor homályosan használja, és a kutatások egy része nem tesz különbséget az MI, a gépi tanulás és az adatelemzés között (Collins és mtsai., 2021). A szerzők szerint a jövőbeni kutatások egyik kulcsfeladata az MI pontos fogalmi kereteinek meghatározása, hogy az empirikus vizsgálatok jobban összehasonlíthatók legyenek.

2.2.1. Történeti áttekintés

A mesterséges intelligencia története szorosan összefonódik a számítástechnika fejlődésével. Az MI kifejezést először McCarthy és mtsai. (1955) használta a híres Dartmouth-konferencián, ahol a kutatók célul tűzték ki az „intelligens gépek” megalkotását. Az 1950-es

[8] https://www.outsystems.com/
[9] https://www.mendix.com/
[10] https://powerapps.microsoft.com/
[11] https://bubble.io/
[12] https://www.appsheet.com/

és 1960-as években az MI kutatásokat a *szimbolikus, szabályalapú rendszerek* uralták: az olyan korai projektek, mint az *ELIZA* (Weizenbaum, 1966) vagy a *SHRDLU* (Winograd, 1972) egyszerű természetesnyelv-feldolgozási feladatokat oldottak meg szabályrendszerek segítségével.

A hetvenes évek közepétől az úgynevezett *MI-tél* (AI winter) időszaka következett, mivel a technológiai korlátok és a túlzott várakozások miatt a kutatások lassultak. Az 1980-as években a *szakértői rendszerek* (expert systems) jelentették az új áttörést, amelyek például az orvosi diagnosztikában vagy az ipari hibadetektálásban használták a mesterséges intelligenciát (Feigenbaum, 1977). A *gépi tanulás* (machine learning) koncepciója ekkor kezdett erősödni, de a valódi forradalom a 2010-es években következett be, amikor a nagy mennyiségű adat (Big Data), a megnövekedett számítási teljesítmény és a grafikus processzorok (GPU-k) elérhetősége lehetővé tette a *mélytanulás* (deep learning) gyakorlati alkalmazását.

A 2010-es évektől a mesterséges intelligencia a mindennapi technológiák részévé vált. A Google, az Amazon és a Microsoft felhőalapú MI-szolgáltatásokat kínál, miközben az önvezető járművek, az orvosi diagnosztikai rendszerek és a pénzügyi döntéstámogató algoritmusok is mind mesterséges intelligencián alapulnak. A ChatGPT[13] és más *nagy nyelvi modellek* (Large Language Models, LLM) megjelenése a 2020-as évek elején újabb paradigmaváltást hozott, amit már sokan a *generatív mesterséges intelligencia* korszakának neveznek (Cao és mtsai., 2023). Ezt a fejlődési ívet ma az *agentic AI* irányzat folytatja, amelynek célja, hogy a mesterséges intelligencia ne csupán tartalmat generáljon, hanem autonóm módon képes legyen döntéseket hozni és tevékenységeket végrehajtani (Acharya és mtsai., 2025).

2.2.2. Főbb irányok

A mesterséges intelligencia története során több irányzat különíthető el, amelyek részben egymást részben kiegészítve, részben felváltva dominálták a kutatást és a gyakorlati alkalmazásokat.

Szimbolikus mesterséges intelligencia (Szakértői rendszer). A szimbolikus MI, más néven „klasszikus" vagy „Good Old-Fashioned AI" (GOFAI), logikai szabályok és explicit tudásreprezentációk alapján működik. A rendszerek előre definiált tudásbázisokra és következtetési szabályokra épülnek. A megközelítés előnye, hogy a döntéshozatal átlátható és magyarázható, hátránya viszont a korlátozott tanulási képesség és a nagymértékű manuális tudásbevitel igénye (Newell & Simon, 1976).

Gépi tanulás (Machine Learning). A gépi tanulás a mesterséges intelligencia egyik legfontosabb alterülete, amelyben a rendszerek adatmintákból, nem pedig előre programozott szabályokból tanulnak. Ezzel a módszerrel kiváltható, hogy embereknek kelljen explicit leírni komplex szabályrendszereket, ehelyett az MI „magától" képes megtanulni

[13] https://chatgpt.com/

az adatok mögötti összefüggéseket. A gépi tanulás három fő kategóriája: a felügyelt (supervised), a nem felügyelt (unsupervised) és a megerősítéses (reinforcement) tanulás. A leggyakrabban alkalmazott technikák közé tartoznak a döntési fák, a támogatott vektorgépek (Support Vector Machine, SVM), a neurális hálók és a Bayes-féle modellek (Mitchell, 1997).

Mélytanulás (Deep Learning). A mélytanulás a neurális hálózatok többrétegű architektúráira épül, amelyek képesek hierarchikus jellemzők automatikus kinyerésére az adatokból. Ezzel a módszerrel jelentős áttörést sikerült elérni a gépi látás, a beszédfelismerés és a természetesnyelv-feldolgozás (natural Language Processing, NLP) területén. A mélytanulás különösen hatékony, ha nagy mennyiségű címkézett adat és nagy számítási kapacitás áll rendelkezésre (LeCun és mtsai., 2015).

Generatív mesterséges intelligencia. A generatív MI (GMI) az utóbbi évek legjelentősebb innovációs irányzata, amely képes új tartalmak – például szöveg, kép, zene vagy programkód – előállítására a tanulási adatok mintázatai alapján. A generatív modellek legismertebb típusai a generatív adverszárius hálók (GAN), a variációs autoenkóderek (Variational Autoencoder, VAE) és a transzformer alapú nagy nyelvi modellek (Large Language Model, LLM) (Banh & Strobel, 2023; Cao és mtsai., 2023). Ezek az architektúrák nemcsak mintákat ismernek fel, hanem képesek új, koherens és kontextushoz illeszkedő kimeneteket előállítani. A legmodernebb modellek már több százmilliárd paraméterrel rendelkeznek, és bizonyos feladatokban képesek megközelíteni az emberi szintű teljesítményt (Hadi és mtsai., 2023).

Agentikus mesterséges intelligencia. Az agentikus MI (agentic AI) egy feltörekvő paradigma, amelyben az MI-t már nem is puszta tartalom-előállításra (mint a generatív MI-t), hanem autonóm cselekvésre használják. Az MI ágensek előre definiált feladatokat teljesítenek önállóan (vagy minimális emberi felügyelettel). Működésük alapja a tervezési (planning) és következtetési (reasoning) képesség, amelyhez gyakran nagy nyelvi modelleket (LLM) használnak. Az agentikus rendszerek képesek egy komplex feladatot lépésekre bontani, külső eszközöket használni információgyűjtésre vagy műveletek végrehajtására, és a visszajelzésekből tanulva dinamikusan adaptálódni a változó körülményekhez (Acharya és mtsai., 2025). Ez a megközelítés megkísérli még közelebb vinni az MI rendszerek működését az emberi cselekvéshez: az MI reaktív eszközből proaktív szereplővé válik.

2.2.3. Erősségek, korlátok és kockázatok

A mesterséges intelligencia legnagyobb ereje az adatokból való tanulás képessége: az MI-rendszerek hatalmas adathalmazokban is képesek felfedezni rejtett mintázatokat, amelyek emberi elemzők számára láthatatlanok maradnának. Ugyanakkor az MI megbízhatósága nagymértékben függ az adatok mennyiségétől és minőségétől – ha az adathalmazok torzítottak, az algoritmusok is torz eredményeket fognak produkálni (Mehrabi és mtsai., 2021).

A technológia másik komoly korlátja az *állapottér-robbanás* (state space explosion). Ez azt jelenti, hogy a lehetséges megoldási kombinációk száma exponenciálisan növekszik a bemeneti paraméterek számával. Emiatt a legtöbb MI-modell nem képes garantálni a globálisan optimális megoldást, hanem heurisztikus, közelítő stratégiákat alkalmaz, lokális optimumokat adva eredményül. A mély neurális hálók további nehézsége a tesztelhetőség és a helyességbizonyítás kérdése: mivel a döntéshozatal a belső súlyok millióin alapul, a rendszerek gyakorlatilag „feketedobozként" működnek, azaz nem tudhatjuk pontosan, miért az adott eredményt adja (Raji és mtsai., 2020).

Az MI bevezetése adatvédelmi és etikai kockázatokat is hordoz. A nagy nyelvi modellek például képesek lehetnek privát információk tárolására vagy visszaidézésére, ami adatszivárgási veszélyt jelenthet. További problémát okoz, hogy az algoritmusok gyakran nem adnak lehetőséget a felhasználói kontrollra vagy az „értesítés és választás" alapelv érvényesítésére, ami adatvédelmi jogi aggályokat vet fel (Plant és mtsai., 2022).

A vállalati környezetben különösen fontos a *responsible AI* (felelős mesterséges intelligencia) elveinek betartása, ideértve a modellek átláthatóságát, auditálhatóságát és az emberi döntéshozatal fenntartását kritikus folyamatokban (Floridi & Cowls, 2019). Noha a mesterséges intelligencia rendszerek képesek önálló döntéseket hozni és javaslatokat generálni, a felelősség végső soron mindig az emberi felhasználót terheli. A vállalatoknak ezért egyértelmű felelősségi kereteket kell kialakítaniuk az MI-alkalmazások használatakor, hogy világos legyen, ki viseli a következményeket egy hibás döntés, félrevezető javaslat vagy adatkezelési incidens esetén. Az MI tehát nem váltja ki az emberi felelősséget, csak a formáját változtatja meg: a technológiai döntések etikai és jogi következményeit továbbra is az emberi szereplőknek kell viselniük.

Az MI hosszú távú hatásait tekintve több kutató felveti az emberi kompetenciák fokozatos eróziójának kockázatát is. Ha a szakemberek mindennapi munkájukban egyre inkább az MI-re támaszkodnak, fennáll a veszélye, hogy a döntéshozatal és a problémamegoldás képessége fokozatosan csökken (Gerlich, 2025). Ugyanakkor az MI nem az ember helyettesítője, hanem eszköze lehet: a legnagyobb potenciál a hibrid rendszerekben rejlik, ahol az emberi intuíció és a mesterséges analitika egymást erősítve működik együtt (Akinnagbe, 2024).

2.3. Mesterséges intelligencia a szoftverfejlesztésben

A szoftverfejlesztés komplex mérnöki feladat, ahol a siker nemcsak technikai tudáson, hanem kreativitáson, együttműködésen és folyamatos tanuláson is múlik. Mivel a fejlesztés specifikus szaktudást igénylő tevékenység, költségei is magasak, így a hatékonyság növelésének kérdése gyakorlatilag egyidős magával a szoftverfejlesztéssel. A különböző fejlesztési módszertanok, programnyelvek, integrált fejlesztői környezetek és automatizációs megoldások mind ugyanazt a célt szolgálták: rövidebb idő alatt, vagyis hatékonyabban és alacsonyabb költségen lehessen egyre komplexebb szoftvereket létrehozni, egyre magasabb minőségben.

A mesterséges intelligencia megjelenése új fejezetet nyitott ebben a folyamatban. Az MI széleskörű alkalmazhatósága miatt gyakorlatilag a szoftverfejlesztési életciklus minden szakaszában képes értéket teremteni, a követelményfeltárástól kezdve a kódoláson, tesztelésen és dokumentáláson át egészen az üzemeltetésig (IBM, é.n.-a). Az MI-alapú eszközök ma már nemcsak kiegészítő segédeszközök, hanem a fejlesztési folyamat integrált (sőt, gyakran elengedhetetlen) részei, amelyek az emberi tudást és tapasztalatot kiegészítve támogatják a fejlesztést.

2.3.1. Nemgeneratív mesterséges intelligencia a szoftverfejlesztésben

A mesterséges intelligencia már a generatív modellek megjelenése előtt is fontos szerepet játszott a szoftverfejlesztésben. Már a 2000-es évek elejétől kezdve alkalmaztak gépi tanulási és adatbányászati módszereket a hibák előrejelzésére, a kódminőség javítására, valamint a tesztelés és üzemeltetés automatizálására. Ezeket a megoldásokat nem tekintjük „generatívnak", hiszen nem új kódot vagy szöveget hoznak létre, hanem elemzéssel, előrejelzéssel vagy döntéstámogatással támogatják a fejlesztési folyamatot.

Ez az alfejezet csupán néhány jellegzetes felhasználási módot mutat be példaként az SDLC különböző fázisaiból, ez a felsorolás azonban semmiképpen sem tekinthető szisztematikus áttekintésnek. A bemutatott felhasználási módokat a 2.1. táblázat foglalja össze.

Fejlesztési időtartam előrejelzése. A projektmenedzsment folyamatokban az MI-t gyakran használják becslésre. A gépi tanulási modellek képesek korábbi projektadatokból (pl. story pointok, commit mennyiség, csapatteljesítmény) tanulni, és ezek alapján előrejelezni a jövőbeli feladatok időigényét (Ali & Gravino, 2019).

Erőforrás-allokáció optimalizálása. Az erőforrás-allokációs problémák – például hogy melyik fejlesztők, melyik feladatokon, milyen sorrendben dolgozzanak – jól formalizálhatók optimalizációs problémaként. A modern MI-eszközök prediktív modelleket alkalmaznak a csapatteljesítmény és a feladatbonyolultság elemzésére. Az ilyen döntéstámogató rendszerek alkalmasak a menedzsment folyamatok támogatására, hosszabb távon akár automatizálására is (Nabeel, 2024).

Követelményellenőrzés. Mivel a követelményekre épül később a teljes szoftverfejlesztési folyamat, kiemelt fontosságú, hogy precízen legyenek megfogalmazva. A természetesnyelv-feldolgozás (NLP) segítségével a szöveges követelmények is elemezhetők, és ellenőrizhető, hogy teljesítik-e a követelményekkel szemben támasztott általános elvárásokat, mint pl. egyértelműség, teljesség és ellentmondásmentesség (L. Zhao és mtsai., 2021).

Architektúraelemzés. A szoftverarchitektúra alapvetően meghatározza a rendszer hosszú távú karbantarthatóságát. Egyes MI-eszközök képesek a rendszer forráskódjából és függőségi gráfjából automatikusan következtetni bizonyos architekturális tulajdonságokra, mint például a rétegsértések, ciklikus függőségek, instabil komponensek vagy más

úgynevezett *architecture smell*-ek (hibára utaló mintázatok) jelenléte. Az ilyen eszközök statisztikai és gépi tanulási módszerekkel azonosítják a tipikus mintázatokat (Cunha és mtsai., 2020).

Statikus kódanalízis. A statikus kódanalízis célja, hogy a forráskódot futtatás nélkül vizsgálja, és azonosítsa a hibalehetőségeket, kódstílusbeli problémákat vagy biztonsági sebezhetőségeket. A modern eszközök a hagyományos szintaktikai ellenőrzésen túl már gépi tanulási modelleket is használnak a hibák mintázatainak felismerésére. Az ilyen modellek korábbi hibajavítási adatokat használnak tanításhoz, és képesek a fejlesztői szokásokból tanulni (Amalfitano és mtsai., 2023).

Hibapredikció. A hibapredikció célja annak előrejelzése, hogy egy adott kódrészletben vagy modulban mekkora a hibák megjelenésének valószínűsége. Míg a statikus kódanalízis a kód állapotát vizsgálja, addig a hibapredikció időbeli adatokat (pl. commit history, fejlesztői aktivitás, hibajegyek) elemez, így képes megtalálni azokat a komponenseket, amelyek a jövőben nagyobb karbantartási kockázatot hordoznak (Y. Zhao és mtsai., 2023).

Tesztadat-generálás. Fontos különbséget tenni a kombinatorikus és a generatív megközelítések között: a nemgeneratív (kombinatorikus) modellek meglévő adatok vagy specifikációk alapján generálnak új bemeneteket — például keresési algoritmusok, genetikus optimalizáció vagy constraint-solver technikák segítségével (J. Zhang és mtsai., 2014). Ezek a rendszerek nem „találnak ki" új tesztadatokat, hanem a lehetséges bemeneti tér lefedettségét optimalizálják, növelve a tesztelés hatékonyságát.

Regressziósteszt-priorizálás. A regressziós tesztelés során a szoftver új verzióinak ellenőrzése történik, hogy a módosítás nem rontotta el azt, ami eddig már működött. Ez nagyméretű rendszerek esetén rendkívül időigényes lehet. A regressziósteszt-priorizálás célja, hogy a teszteket olyan sorrendben futtassa, ami minimalizálja az időráfordítást és maximalizálja a hibák detektálását. Erre azért van szükség, mert a tesztelésre rendelkezésre álló idő rendre kevesebb, mint az összes teszt teljes futási ideje (Sawant, 2024).

Anomáliafelismerés. Az üzemeltetési és karbantartási szakaszban az anomáliafelismerés a normál működéstől eltérő viselkedés automatikus azonosítását jelenti, amire szintén használhatók MI-alapú megoldások. A rendszerlogok, teljesítménymutatók és hálózati forgalmi adatok alapján működő AIOps-rendszerek képesek proaktívan jelezni egy esetleges meghibásodást vagy teljesítményromlást (Jeyarajan és mtsai., 2025).

2.1. táblázat. Nemgeneratív MI felhasználások és eszközök az SDLC fázisaiban. (Saját szerkesztés.)

SDLC fázis	Felhasználás típusa	Eszközök
Projekttervezés	Fejlesztési időtartam előrejelzése	Atlassian Forecast[14]
	Erőforrás-allokáció optimalizálása	Monday AI[15], ClickUp Brain[16]
Elemzés	Követelményellenőrzés	IBM Engineering Requirements Quality Assistant[17], ScopeMaster Requirements Analyser[18]
Rendszertervezés	Architektúraelemzés	Designite[19], Embold[20]
Fejlesztés	Statikus kódanalízis	SonarQube[21]
	Hibapredikció	Bugspots[22]
Tesztelés	Tesztadat-generálás	Microsoft PICT[23], Hexawise[24]
	Regresszióteszt-priorizálás	Leapwork[25]
Karbantartás	Anomáliafelismerés	Datadog Watchdog[26], Dynatrace Davis AI[27]

2.3.2. Generatív mesterséges intelligencia a szoftverfejlesztésben

A nemgeneratív és a generatív mesterséges intelligencia közti alapvető különbség a szoftverfejlesztésbeli felhasználásukban is tetten érhető: míg előbbi az emberek által előállított különféle artefaktumokat (dokumentum, programkód, konfiguráció stb.) elemzi, utóbbi már konkrétan előállítja ezeket. Vagyis a generatív mesterséges intelligencia – emberi utasítások (promptok) alapján – képes a szoftverfejlesztési folyamat során előállítandó kimenetek közvetlen létrehozására.

A generatív MI modellek tehát tekinthetők emberi nyelven („emberi nyelvi interfészen") irányítható fejlesztői asszisztenseknek. A kódolásban ez olyan, mint a páros programozás félig gépi megvalósítása, ahol az MI egy mindig elérhető és soha el nem fáradó „pair programmer". Gyorsan képes ötleteket és megoldási javaslatokat adni, kódvázlatokat előállítani, meglévő kódokat elemezni, ugyanakkor nem tévedhetetlen, ezért emberi kontrollt igényel (De Siano és mtsai., 2025).

[14] https://marketplace.atlassian.com/vendors/1213598/forecast
[15] https://monday.com/
[16] https://clickup.com/brain
[17] https://www.ibm.com/docs/en/erqa
[18] https://www.scopemaster.com/solutions/requirements-analyser/
[19] https://www.designite-tools.com/
[20] https://embold.io/
[21] https://www.sonarsource.com/products/sonarqube/
[22] https://github.com/igrigorik/bugspots
[23] https://github.com/microsoft/pict
[24] https://hexawise.com/
[25] https://www.leapwork.com/
[26] https://www.datadoghq.com/product/platform/watchdog/
[27] https://www.dynatrace.com/platform/artificial-intelligence/

A 2020-as években megjelenő nagy nyelvi modellek (LLM) alapozták meg a generatív MI ezirányú felhasználását. Az óriási mennyiségű szövegen (így többek között programkódokon is) tanított modellek képesek kontextusba illő válaszokat adni – programozási kérdésekre is. Az általános LLM-ek mellett azonban megjelentek a célspecifikus modellek is, mint pl. a kifejezetten programkódokra finomhangolt megoldások, az ún. code LLM-ek. Ezeket a modelleket az általános szövegek mellett nagy mennyiségű forráskódon, dokumentáción, pull requesten és hibajegyen tanították. Ennek eredményeként a modellek megtanulják a programkódok tipikus mintázatait, a programozási nyelvek szintaktikai szabályaitól kezdve egészen a kódok magas szintű szerkezetéig (Q. Zhang és mtsai., 2024).

Fejlesztői eszközök. A generatív mesterséges intelligencia integrációja rohamosan terjed a fejlesztői eszközökben (Walsh és mtsai., 2025). Ennek egyik formája az *integrált fejlesztői környezetekbe* (integrated development environment, IDE) épített kódolási asszisztens. Ezek az eszközök nemcsak egyszerű kódkiegészítést biztosítanak (mint már a nemgeneratív eszközök is), hanem egész sorokat, sőt, teljes függvényeket vagy osztályokat is legenerálnak. A modern eszközök képesek figyelembe venni a kód kontextusát is: átlátják a szerkesztett fájlt vagy akár a teljes projektet, és ennek megfelelő javaslatokat adnak.

A másik jellemző eszköz a chat alapú fejlesztői asszisztens, amely lehetővé teszi, hogy a fejlesztő párbeszédet folytasson az MI modellel a kódról. Ez használható bonyolult függvények magyarázatára, alternatív megoldások keresésére, hibakeresésre. Az agentikus megközelítés ennél is tovább megy, azok a rendszerek már képesek közvetlenül végrehajtani a feladatokat (pl. kódot generálni, projektet refaktorálni, fordítani).

Hallucináció. A nyelvi modellek kapcsán fontos megemlíteni a *hallucináció* problémáját, vagyis amikor a modell olyan választ állít elő, amely kontextusba illőnek és helyesnek tűnik, vagyis hihető, ugyanakkor mégis hibás.[28] Ez egyaránt jelenthet egy téves tényadatot (pl. egy egyszerű matematikai számítás eredménye) vagy hivatkozást valamire, ami valójában nem létezik (pl. szolgáltatás, tudományos mű). A hallucináció a modellek jelenlegi tanítási és kiértékelési módszereiből ered, amelyek leegyszerűsítve a találgatást jutalmazzák a bizonytalanság beismerésével szemben (Kalai és mtsai., 2025).

Kontextus. A modellek gyakorlati hasznosíthatósága szempontjából kulcskérdés a *kontextus* mérete és kezelése. Minél komplexebb problématerekről van szó, annál inkább elengedhetetlen a modellek széles látóköre, pl. egy több ezer fájlból álló projekt esetén nagyon korlátosan használható egy olyan modell, amely csak az adott fájlt képes átlátni, hiszen egy közelről helyesnek tűnő megoldás lehet, hogy valójában több új problémát okoz, mint ahányat megold.

A kontextus tágítása azonban nemcsak mennyiségi kérdés. Nagyvállalati környezetben azok a tudáselemek, amelyek egy probléma megoldásához szükségesek, gyakran nem egy helyen és azonos formában állnak rendelkezésre, hanem széttöredezetten. Vagyis pl.

[28] Egyes szakértők felhívják a figyelmet, hogy a hallucináció elnevezés egy félrevezető eufemizmus, hiszen ez valójában az LLM-ek működésének a lényege. Amit hallucinációnak hívunk, az a gyakorlatban egyszerű tévedés (Barroca, 2024).

egyaránt kellene értelmezni különféle szoftver projekteket, hibajegyeket, dokumentumokat, adatbázisokat és tudásmegosztó oldalakat (Yang és mtsai., 2025).

Ahhoz tehát, hogy nagyvállalati környezetben is hatékonyan tudjuk használni a generatív MI-t, kulcsfontosságú az MI és a vállalati rendszerek mély integrációja. Fontos látni, hogy ez az integráció kétirányú tudásmegosztást feltételez, és mindkét irány számos kihívást és kockázatot hordoz. Egyrészt a vállalati rendszerekben felhalmozott tudást az MI modell számára hozzáférhetővé kell tenni, hogy az képes legyen beemelni azt a saját kontextusába. Másrészt az MI modelleket úgy kell integrálni a vállalati folyamatokba, hogy azok ne szigetszerű eszközök legyenek a szervezeten belül, hanem a vállalati működés szerves részeivé váljanak – miközben a kimenetük nem kerül a szervezeten kívülre.

2.4. Iparági áttekintés

A szoftveripar a globális gazdaság egyik leggyorsabban növekvő és legnagyobb hatású ágazata. A digitális transzformáció évtizedeiben szinte minden iparág szoftvervezéreltté vált, ami drasztikusan megnövelte a szoftverfejlesztés iránti keresletet. Napjainkban az szoftveriparon belül is külön említést érdemelnek a felhőalapú szolgáltatások, a SaaS (Software as a Service) modellek, a kibervédelem és az üzleti folyamatok digitalizációja. A szoftveripar a kezdetekben támogató funkciót töltött be, napjainkra azonban már számos szervezet alapvető versenyképességi tényezőjévé vált. Ezen szervezetek esetében a gyors fejlesztés, a megbízhatóság és a skálázhatóság közvetlenül teremtenek üzleti értéket.

Az iparági elemzések évek óta folyamatos, jelentős növekedést jeleznek a vállalati szoftverpiacon, a várakozások szerint 2025-ben a piac éves bevétele meghaladhatja a 730 milliárd dollárt (Grand View Research, 2025). Egy 2023-as statisztika 2024-re 28,7 millió szoftverfejlesztőt prognosztizált (Statista, 2023), ez becslések szerint 2030-ra elérheti a 45 millió főt is (Uspenskyi, 2025). A piaci szereplőket az alábbi négy fő csoportba sorolhatjuk:

1. **Technológiai óriások (Big Tech).** A piac domináns szereplői (pl. Microsoft, Google, Meta, Amazon), akik nemcsak használják, hanem *szolgáltatják* is az alapvető infrastruktúrát (pl. operációs rendszer, irodai szoftvercsomag, felhőszolgáltatások). Számukra az MI a platformszolgáltatások (PaaS/SaaS) fontos eleme, így a GMI technológiáknak nemcsak felhasználói, hanem szolgáltatói és aktív kutatói is.

2. **Szoftverfejlesztő ügynökségek.** Ide tartoznak a globális tanácsadók (pl. Accenture, Cognizant, Epam) és a regionális fejlesztőházak. Üzleti modelljük az emberi erőforrás „bérbeadásán” vagy projektalapú elszámoláson alapul. Számukra a legnagyobb versenytényező a hatékonyság és a fejlesztői produktivitás növelése, aminek érdekében többek között GMI-t is felhasználhatnak.

3. **Nagyvállalatok.** A nem technológiai fókuszú szektorok (pénzügyi szektor, autóipar, gyógyszeripar) belső fejlesztőrészlegei. Itt a legjellemzőbbek az alapműködést támogató *örökölt rendszerek*, és a velük járó *technikai adósság*, amely leküzdésében nagy segítséget jelenthet a GMI.

4. **Startupok és KKV-k.** A gyors innovációra építő cégek, ahol gyakori az erőforráshiány (tőke és munkaerő). Ebben a szegmensben kulcskérdés a gyors piacra lépés (time-to-market), amihez nagy segítséget jelenthet a GMI.

Kutatásom fókuszában a nagyvállalati szoftverfejlesztés áll, amely lényegesen eltér a kisebb projektek vagy startupok világától. A nagyvállalati rendszerek gyakran több évtizedes múlttal, sokszor különböző generációk technológiáira épülve működnek tovább. A több millió soros monolitikus rendszerek, az üzletmenet szempontjából kritikus szoftverek (pl. core pénzügyi rendszerek, telekommunikációs platformok) és a heterogén infrastruktúrák mind növelik a komplexitást, amelyet emberi kapacitásokkal már egyre nehezebb teljes mértékben átlátni. A folyamatosan szaporodó compliance követelmények tovább bonyolítják a rendszereket, miközben a vállalatok változatlanul rövid fejlesztési ciklusokat és folyamatos innovációt várnak el. A rendszerek és komponenseik közötti erős kölcsönhatások miatt egy látszólag egyszerű változtatás is nagy kockázatokkal járhat, így gyakran a látszólag egyszerű fejlesztői döntések előtt sem hagyható ki az előbb felsorolt komplexitás áttekintése.

A fenti tényezők mellett az iparágban kritikus kérdés a technikai adósság (technical debt) és az örökölt (legacy) rendszerek fenntartása is. Számos szervezet informatikai infrastruktúrájának jelentős része még mindig olyan technológiákon fut, amelyeket nehéz modernizálni, mert a kritikus belső folyamatok ezekre épülnek. A legacy rendszerek frissítése gyakran magas kockázattal jár, a dokumentáció nem teljes, a tudás pedig sokszor személyekhez kötött (tribal knowledge). A minőségbiztosítás, a tesztelés és a hibajavítás így a nagyvállalatok egyik legköltségesebb és legmunkaigényesebb tevékenységévé vált (Monaghan & Bass, 2020).

Mindez jól magyarázza, miért vált a generatív mesterséges intelligencia az iparág egyik legígéretesebb technológiájává. A rendszerek növekvő mérete, a tudás szétszórtsága és a dokumentáció hiánya olyan környezetet eredményez, ahol különösen nagy értéke van annak, ha egy modell képes nagy mennyiségű szöveget, kódot és információt egyszerre értelmezni. A generatív MI nem csupán végrehajtási automatizálást tesz lehetővé, hanem a várakozások szerint a fejlesztési folyamat szellemi terheit is képes csökkenteni: kódolási mintázatokat ismer fel, alternatív megoldásokat javasol, összefoglalja a kontextust, és segít átlátni a komplex rendszerek működését. Ez a képesség teszi relevánssá és stratégiai jelentőségűvé a generatív MI-t a nagyvállalati szoftverfejlesztésben, különösen olyan környezetekben, ahol a gyorsaság, a minőség és a szabályozási megfelelőség egyszerre kritikus elvárás.

3. fejezet

Kutatásmódszertan

Ebben a fejezetben összefoglalom a dolgozatban alkalmazott kutatásmódszertant. A 3.1. alfejezetben vázolom a kutatás hátterét és motivációját, majd a 3.2. alfejezetben meghatározom a kutatási célokat. A 3.3. alfejezetben megfogalmazom a kutatási célokból következő kutatási kérdéseket, majd a 3.4. alfejezetben bemutatom az alkalmazott adatgyűjtési módokat. Végül a 3.5. alfejezetben összegzem a bemutatott módszertant.

3.1. Kutatás háttere

Ahogyan szinte az egész világot, a szoftverfejlesztést is alaposan felforgatta a generatív mesterséges intelligencia. Mivel magam is szoftverfejlesztőként dolgozom, személyesen is találkozom a téma dilemmáival. Egyik vállalat sem szeretne semmiről sem lemaradni, ezért kifejezetten támogatják ezen eszközök használatát, de ennek a gyakorlatban számos nehézsége is adódik. Ez a személyesen is megtapasztalt kettősség motivált arra, hogy tágabb kontextusban is megvizsgáljam a technológia felhasználását.

Számos tanulmány foglalkozik a generatív mesterséges intelligenciával, azon belül is a szoftverfejlesztésben való felhasználhatóságával. Én azonban ezen belül is azt szeretném megvizsgálni, hogy milyen potenciális ellentmondások húzódnak az elméleti felhasználhatóság és a gyakorlati felhasználás között, vagyis mik azok a körülmények és nehézségek, amelyek éket vernek a potenciális lehetőségek és a rögvalóság közé.

3.2. Kutatási célok

Kutatásom célja a generatív mesterséges intelligencia szoftverfejlesztésbeli felhasználásának körüljárása mind elméleti, mind gyakorlati oldalról, majd az elméleti tudás és a gyakorlati tapasztalatok szintetizálása. Mindezt egy olyan rendszerbe foglalva elemzem, amely újrafelhasználható modellként szolgálhat potenciális további kutatások számára is, azaz alkalmas arra, hogy felmérje más szervezet(ek) viszonyulását ehhez a technológiához.

Kutatásomat a szoftverfejlesztési életciklus (SDLC) strukturálja, amely egy régóta használt modell a szoftverfejlesztés fázisainak leírására. Az SDLC univerzális fázisain végighaladva elemzem a generatív mesterséges intelligencia felhasználását.

Minden fázis kapcsán áttekintem a szakirodalomban leírt főbb felhasználási lehetőségeket, valamint az iparági és akadémiai interjúalanyok álláspontját a lehetséges felhasználásokról. Összegzem az interjúkon elhangzott, valamint személyesen átélt felhasználási tapasztalatokat, amelyeken keresztül tetten érhető, hol is tart valójában az elméleti lehetőségek gyakorlati megvalósulása. Végül az összegyűjtött tudás és tapasztalat alapján megkísérlem feltárni a gyakorlati megvalósulás mozgatórugóit, szükséges feltételeit, potenciális kihívásait, akár ellehetetlenítő körülményeit.

3.3. Kutatási kérdések

Munkám során egyszerre igyekeztem a téma technikai és emberi aspektusait is megvizsgálni. Noha maga a szoftverfejlesztés és a generatív mesterséges intelligencia kifejezetten technikai témák, ezek mellett igyekeztem kidomborítani a szoftverfejlesztésben dolgozók személyes megéléseit is, legyenek azok akár általános gondolatok a témában, akár konkrét felhasználási tapasztalatok. Mindezek alapján az alábbi kutatási kérdéseket fogalmaztam meg, és kerestem rájuk a választ.

RQ1. Hogyan vélekednek a szoftverfejlesztésben dolgozók általánosságban a generatív mesterséges intelligenciáról?

RQ2. Hogyan használható a generatív mesterséges intelligencia a szoftverfejlesztési életciklus (SDLC) különböző fázisaiban?

1. Projekttervezés
2. Elemzés
3. Rendszertervezés
4. Fejlesztés
5. Tesztelés
6. Bevezetés
7. Karbantartás

RQ3. A tapasztalatok alapján mik a generatív mesterséges intelligencia szoftverfejlesztésbeli felhasználásának kihívásai és korlátai?

3.4. Adatgyűjtés

Kutatási témámat több eltérő nézőpontból is szerettem volna megvizsgálni, így munkám során különböző módon gyűjtött adatokat szintetizáltam. Áttekintettem a releváns szakirodalmat, iparági panelbeszélgetéseket szerveztem, interjúkat készítettem egyetemi oktatókkal, valamint felhasználtam személyes tapasztalataimat is.

3.4.1. Szakirodalmi áttekintés

Kutatásom egyik fő részét a vonatkozó szakirodalom feldolgozása adja (2. fejezet). Számos nemzetközi publikációt tekintettem át, többek között konferenciacikkeket, preprint publikációkat, iparági jelentéseket és technológiai leírásokat. Ez egyrészt megalapozta munkám elméleti hátterét, másrészt rávilágított arra is, hogy mennyire újszerű témáról van szó, hiszen számos területen még a tudománynak is csak kérdései vannak.

3.4.2. Iparági panelbeszélgetések

A személyes vélemények és tapasztalatok becsatornázására személyes jelenlétű, kiscsoportos panelbeszélgetéseket szerveztem egy globális pénzügyi szolgáltató budapesti irodájának dolgozói között. A sok évtizedes múlttal rendelkező cég működésében a kezdetektől fogva kulcsszerepet játszik a saját fejlesztésű szoftveres megoldásokra épülő innováció, így kiváló terepet szolgáltatott kutatásomhoz.

Összesen 11 résztvevővel zajlottak a beszélgetések, akiket a kezelhető létszám érdekében kettéosztottam. A beosztás során igyekeztem hasonló szakmaitapasztalat-eloszlást kialakítani a két csoportban. A panelbeszélgetések adatait a 3.1. táblázat tartalmazza.

3.1. táblázat. Iparági panelbeszélgetések adatai. (Saját szerkesztés.)

Dátum	Időtartam	Résztvevő	Szakmai tapasztalat	Beosztottak száma
2025. 11. 13.	90 perc	Szoftverfejlesztő 1.	10 év	0
		Szoftverfejlesztő 2.	20 év	0
		Szoftverfejlesztő 3.	15 év	1
		Szoftverfejlesztő 4.	18 év	2
		Szoftverfejlesztő 5.	18 év	9
2025. 11. 14.	75 perc	Szoftverfejlesztő 6.	20 év	0
		Szoftverfejlesztő 7.	14 év	0
		Szoftverfejlesztő 8.	16 év	0
		Szoftverfejlesztő 9.	15 év	1
		Szoftverfejlesztő 10.	32 év	0
		Szoftverfejlesztő 11.	27 év	0

A beszélgetéseket rávezetésképp az MI-vel kapcsolatos általános vélemények megvitatásával kezdtük. Ezt követte egy brainstorming a generatív MI szoftverfejlesztésbeli felhasználását illetően, amely során az elhangzott ötleteket cédulákra írtuk, és az SDLC fázisai szerint helyeztük el a táblán. Az ötletelést követően részletesen megvitattuk valamennyi ötletet, konkretizálva, hogy pontosan mit értettek alatta, továbbá kitérve a releváns körülményekre, kihívásokra és korlátokra. Esetenként a múltbeli tapasztalatokon túlmenően arról is kialakultak izgalmas beszélgetések, hogy vajon mit hozhat még a jövő egy-egy felhasználási lehetőséget illetően.

3.4.3. Akadémiai interjú

Az iparágban dolgozók tapasztalatai mellett szerettem volna bevonni az akadémiai szféra meglátásait is, ezért egyetemi oktatókkal is szerettem volna interjúkat készíteni. Az egyetemi nézőpont beemelését több okból is fontosnak tartottam: egyrészt a legmodernebb technológiákat illetően mindenképp releváns a kutatásban élenjáró egyetemi szféra nézőpontja, másrészt a téma a leendő szoftverfejlesztő munkaerőt illetően is felvet kérdéseket, amelyekre előbb-utóbb az egyetemi oktatásnak kell válaszokat adnia.

Végül sajnos csak egy ilyen interjút sikerült elkészítenem, személyes jelenléttel. Ezt az interjút kevésbé strukturáltam, mint az iparági panelbeszélgetéseket, hiszen itt kevésbé az SDLC szerinti konkrét tapasztalatokon, inkább az eltérő nézőpont általános meglátásain volt a hangsúly. A Budapesti Műszaki és Gazdaságtudományi Egyetem (BME) Villamosmérnöki és Informatikai Karának (VIK) egyetemi docensével készítettem interjút, amelynek adatait a 3.2. táblázat tartalmazza.

3.2. táblázat. Akadémiai interjú adatai. (Saját szerkesztés.)

Dátum	**Időtartam**	**Résztvevő**	**Szakmai tapasztalat**	**Tudományos fokozat**
2025. 11. 20.	30 perc	Egyetemi docens 1.	14 év	PhD

3.4.4. Személyes tapasztalatok

A kutatásom során saját szakmai tapasztalataimat is felhasználtam. Mivel több éve szoftverfejlesztőként dolgozom nagyvállalati környezetben, közvetlenül találkoztam azokkal a működési sajátosságokkal és mindennapi kihívásokkal, amelyek a szoftverrendszerek fejlesztését és üzemeltetését övezik. A compliance környezet, a komplex örökölt rendszerek, a dokumentáció hiányosságai és az időnyomás mind olyan tényezők, amelyek közvetlenül befolyásolják a generatív mesterséges intelligencia alkalmazhatóságát is. Ezek a tapasztalatok így értékes kontextust adtak ahhoz, hogy a többi adatot értelmezni tudjam.

Az elmúlt években a generatív MI megjelenését nemcsak hírfogyasztó állampolgárként, hanem a munkám során is megtapasztaltam, így első kézből láthattam, milyen elvárások, kérdések és bizonytalanságok merülnek fel a fejlesztői közösségben. Ez a személyes perspektíva természetesen nem helyettesíti a formális adatgyűjtést, de fontos kiegészítő nézőpontot biztosít: segít megérteni, hogyan jelennek meg a technológiai újítások a mindennapi gyakorlatban, és milyen tényezők határozzák meg azok reális alkalmazhatóságát.

3.5. Összegzés

A különböző adatgyűjtési módszerek eltérő nézőpontokból világítják meg kutatásom témáját. A dolgozat során a *triangulációt* elvét követve kombináltam a szakirodalmi áttekintést, iparági panelbeszélgetéseket, akadémiai interjút és saját szakmai tapasztalataimat, hogy minél árnyaltabb és megbízhatóbb képet nyújtsak a generatív mesterséges intelligen-

cia szoftverfejlesztésre gyakorolt hatásáról. Az így integrált adatokat a kutatási kérdések mentén rendszerezve és értelmezve elemeztem.

Kutatásom egyik egyértelmű korlátját a válaszadók köre jelenti. Egyrészt a minta mérete csak korlátosan teszi lehetővé általános következtetések levonását, másrészt a résztvevők összetétele sem tekinthető reprezentatívnak a teljes iparágra nézve. Mindezek ellenére az eredmények jól érzékeltetik azokat a trendeket, dilemmákat és tapasztalatokat, amelyek a vizsgált témában jelenleg relevánsak lehetnek. A kutatás így értelmezhető egy kvalitatív feltáró vizsgálatként, amely alapot adhat későbbi, nagyobb mintán végzett kutatásokhoz.

A konkrét megállapításokon túl a dolgozatban alkalmazott strukturált, SDLC-fázisok mentén szervezett megközelítés és a panelbeszélgetéseken alapuló módszertan újrafelhasználható keretet biztosíthat további vizsgálatokhoz. Egy hasonló kutatás nagyobb mintán átfogó képet adhat arról, hogyan gondolkodnak a szoftverfejlesztők pl. egy szervezeten belül a generatív MI lehetőségeiről és korlátairól, illetve hol tartanak annak gyakorlati alkalmazásában.

4. fejezet

Kutatási eredmények

Ebben a fejezetben kutatási kérdésenként haladva összefoglalom kutatásom eredményeit. A 4.1. alfejezetben bemutatom a generatív mesterséges intelligenciával kapcsolatos általános vélekedéseket (**RQ1**). A 4.2. alfejezetben az SDLC fázisai szerint haladva áttekintem a generatív mesterséges intelligencia lehetséges felhasználásait (**RQ2**), valamint a vonatkozó tapasztalatokat, kihívásokat és korlátokat (**RQ3**).

4.1. A generatív mesterséges intelligencia megítélése

Ebben az alfejezetben a generatív mesterséges intelligenciával kapcsolatos általános vélekedéseket összegzem, vagyis az **RQ1** kutatási kérdést vizsgálom: *Hogyan vélekednek a szoftverfejlesztésben dolgozók általánosságban a generatív mesterséges intelligenciáról?*

Újdonság varázsa. A generatív mesterséges intelligenciát körülövezi az újdonság varázsa. Sokan izgalmasnak tartják a technológiát és szeretnek kísérletezni vele, vagyis a használata nemcsak a hatékonyságra, hanem az általános fejlesztői közérzetre (wellbeing) is jó hatással lehet (Karaci Deniz és mtsai., 2023). Az újdonság varázsa ugyanakkor nem tart örökké, és a sikertelen használat (pl. sok körös ismételt promptolás egy hiba javításáért) frusztráló is tud lenni, ahogyan az is, ha egy szervezeten belül túlságosan erőltetik az MI használatát.

Átalakuló munkakörök. A megkérdezettek általánosságban nem féltik a munkájukat, azt gondolják, továbbra is szükség lesz szoftverfejlesztői és szakterületi tudással rendelkező emberekre, de a munkakörök átalakulása már most elkezdődött (Muratović és mtsai., 2024). Aki nem használ AI-t, nem lesz versenyképes, így kulcsfontosságú a tanulási képesség, pl. a helyes promptok megfogalmazásához (prompt engineering). Egyelőre nem alakult ki iparági konszenzus azt illetően, hogyan hat majd az MI a szükséges fejlesztői létszámra: vannak, akik szerint a technológia fejlődés miatt még több szakemberre lesz szükség, míg mások szerint épp ellenkezőleg (Singla és mtsai., 2025). Jelenleg az sem egyértelmű még, hogy ahol történtek már leépítések, ott valóban az MI használat sikere állt-e mögöttük, nem pusztán a korábbi túlzott bővülés korrekciója (HOLD Alapkezelő, 2025).

Junior fejlesztők. Egyre szélesebb körben terjed az a vélekedés, hogy a junior fejlesztőknek nem szabad engedni a generatív MI eszközök használatát. A vonatkozó érvelés szerint a tudás akkor rögzül igazán, amikor mi magunk jövünk rá a megoldásra, akkor nem, ha megmondják nekünk. Vagyis maga a gondolkodás, a megoldás keresése, a „szenvedés'' rögzíti a tudást[1] – ami nélkül egy hasonló, de nem pontosan ugyanolyan problémát már nem fogunk tudni megoldani. Rövid távon ugyanakkor más a helyzet, hiszen a legnagyobb produktivitásnövekedést pont a junioroknál okozzák a generatív MI eszközök (Ziegler és mtsai., 2024). Létező vélekedés az is, hogy az MI teljesen kiváltja a junior fejlesztőket, kérdés ugyanakkor, hogy a jelen junior fejlesztői nélkül kik lesznek a jövő senior fejlesztői.

Felesleges szóáradat. A megkérdezettek közül többen is felvetették azt a nehézséget, hogy a generatív MI modellek feleslegesen hosszan fogalmazzák meg a válaszokat. Olyan ez, mintha ezt az új technológiát a nem hatékony emberi kommunikációhoz igazítanánk, ami olyan abszurd helyzetekhez vezethet, hogy egy MI modell által hosszasan megfogalmazott szöveget egy másik MI modellel foglaltatnak össze, holott semmi garancia nincs arra, hogy eközben nem torzul az eredeti információ.

Kreativitás hiánya. Mivel az MI modelleket a meglévő adatokon tanítják, sokak szerint nem lehetnek képesek igazi kreativitásra, valóban újszerű megoldások megalkotására (out of the box thinking). Vannak, akik szerint az embert valójában az *absztrahálás* képessége tette naggyá, és erre az MI igazán sosem lesz képes.

Kilátások. Azt illetően egyetértés uralkodik a megkérdezettek között, hogy a generatív MI az elmúlt években hatalmasat fejlődött. Arról ugyanakkor már megoszlanak a vélemények, hogy mi várható a jövőben: vannak, akik a fejlődés további gyorsulását várják, míg mások lassulásra számítanak a hatalmas energiaigény és a valós tanító adatok kimerülése miatt.

4.2. A generatív mesterséges intelligencia felhasználása

Ebben az alfejezetben bemutatom a generatív mesterséges intelligenciával szoftverfejlesztésbeli felhasználhatóságát, valamint áttekintem az azokat övező tapasztalatokat, kihívásokat és korlátokat, vagyis az **RQ2** (*Hogyan használható a generatív mesterséges intelligencia a szoftverfejlesztési életciklus (SDLC) különböző fázisaiban?*) és **RQ3** (*A tapasztalatok alapján mik a generatív mesterséges intelligencia szoftverfejlesztésbeli felhasználásának kihívásai és korlátai?*) kutatási kérdéseket vizsgálom. Először általános megállapításokat fogalmazok meg, majd a továbbiakban az SDLC fázisai mentén mutatom be az eredményeket.

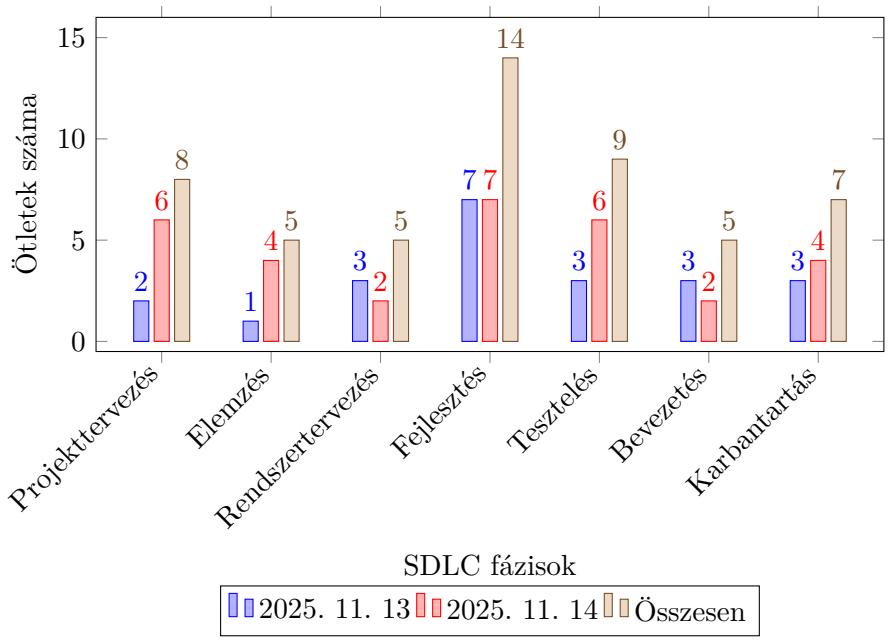
[1] Az alapszakos egyetemi tanulmányaim legelején elsőként kezembe került jegyzet (Szeszlér, 2014) előszava idézte az anekdotát, amely szerint amikor I. Ptolemaiosz király megkérdezte Euklidészt, a kor nagy matematikusát, hogy nem lehetne-e a geometriát az Elemek (Euklidész, é.n.) áttanulmányozásánál könnyebben elsajátítani, Euklidész azt válaszolta: „A geometriához nem vezet királyi út. [...] Munka nélkül nincs kenyér, sem geometria."

Emberi kontroll. *„Lustaságból használni jó, tudatlanságból használni veszélyes''* – foglalja össze a generatív mesterséges intelligenciában (nagy nyelvi modellekben) rejlő lehetőségeket és az azokat övező veszélyeket *Egyetemi docens 1.*, rámutatva ezzel a hallucináció problémájára, és arra, hogy mennyire veszélyes vakon megbízni az MI által generált válaszokban. Mivel a modellek esetenként helyesnek tűnő, de valójában teljesen hibás válaszokat adnak, elengedhetetlen az emberi utóellenőrzés, amelyhez emberi tudásra van szükség. Iparági ajánlások is felhívják a figyelmet az MI emberi kontrolljának fontosságára (Mann & O'Connor, 2023).

Specifikusság, pontosság. A generatív mesterséges intelligencia számos olyan fejlesztési feladat megoldására használható, amelyekre már korábban is léteztek kizárólag algoritmikus (a mai köznyelv MI meghatározása alapján „MI-mentes") megoldások. Jelentős különbség, hogy az algoritmikus módszerek determinisztikusak, belső működésük pontosan leírható, így eredményük is megmagyarázható, helyességük pedig bizonyítható. Ezek az eszközök tehát biztonságosan használhatók, cserébe csak egy-egy konkrét részfeladatot képesek megoldani. Ezzel szemben a generatív MI alapú eszközök belső működésüket tekintve *fekete dobozok (black box)*, azaz nem tudjuk megmagyarázni, az adott bemenetre miért pont az adott kimenetet adják. A belső működés átláthatatlansága bizonytalanságot és pontatlanságot hordoz, cserébe ezek az eszközök univerzálisak, hiszen különböző jellegű kérdésekre egyaránt adnak *valamilyen* választ. A problémák orvoslására megjelentek a generatív MI-t használó specifikus céleszközök, amelyek az univerzalitás feláldozásáért cserébe pontosságot ígérnek. Fontos látni, hogy a dilemma valójában a pontosság (feladatspecifikus eszköz, biztosan helyes eredmény) és a kényelmesség (univerzális eszköz, bizonytalan eredmény) között húzódik, az igazi áttörést pedig az olyan integrált eszközök jelentenék, amelyek *megbízhatóan* képesek támogatni a teljes szoftverfejlesztési folyamatot (Gnanasambandam és mtsai., 2025).

Megspórolt idő. A generatív mesterséges intelligencia potenciálisan hatékonyabbá teszi a szoftverfejlesztést, vagyis a fejlesztők bizonyos feladatokat rövidebb idő alatt tudnak elvégezni, mint korábban. Mivel így kevesebb időt kell egyes technikai részekkel eltölteniük, több idejük marad magasabb szintű tervezésre, munkatársakkal együtt gondolkodásra, új technológiák elsajátítására (Finley, 2024).

Kódoláson túl. A technológia felhasználásának fókuszában a kódolás áll (Walsh és mtsai., 2025), amit a panelbeszélgetések során elhangzott ötletek eloszlása is alátámaszt (4.1. ábra). Fontos azonban látni, hogy a szoftverfejlesztők valójában a munkaidejük jóval kisebb részét töltik kódolással, mint azt akár ők, akár a menedzsereik gondolják (Meyer és mtsai., 2021; Software, 2021). Éppen ezért a kódolási asszisztensek produktivitásnövelő hatása a teljes szoftverfejlesztési folyamatra vetítve nagyon korlátos. Olyan eszközökre van szükség, amelyek a generatív mesterséges intelligencia segítségével az SDLC valamennyi fázisát hatékonyabbá teszik (Valdes és mtsai., 2025).



4.1. ábra. A panelbeszélgetéseken elhangzott ötletek eloszlása SDLC fázisok szerint. (Saját szerkesztés.)

4.2.1. Projekttervezés

A projekttervezés során határozzuk meg a szoftverprojekt célját, hatókörét, erőforrásait és kockázatait. Másképp megfogalmazva meg kell határoznunk, hogy *mit* (cél, hatókör) és *mivel* (erőforrások, kockázatok) fogunk csinálni. Noha a szakirodalom kevés konkrét, valós felhasználási esetet említ ebből a fázisból, a megkérdezettek a jövőbe tekintő spekulációk mellett beszámoltak konkrét tapasztalatokról is, jóllehet, azok gyakran inkább még csak kísérletezésnek tekinthetők.

Helyzetfelmérés. A projekttervezés első lépése rendszerint az aktuális helyzet feltérképezése, amely egyaránt jelentheti a szervezet belső működését és a külső környezetet. A belső működés kapcsán fel kell mérni a használt folyamatokat, a meglévő rendszereket, azok főbb komponenseit, valamint a mindezek közt húzódó kapcsolatokat. A küldő környezetnek meghatározó elemei többek között a versenytársak és a szabályozók. Mindezekről rendszerint sokféle, töredezett, heterogén információ áll rendelkezésre, amelyek feldolgozása és összegzése időigényes emberi feladat. A GMI ugyanakkor hatékonyan képes feldolgozni és összefoglalni az ilyen széttagolt tudáshalmazokat.

A GMI ez irányú hatékony használatához elengedhetetlen a belső vállalati működést leíró dokumentumok beemelése a modell kontextusába. Mivel ezek általában érzékeny üzleti adatokat tartalmaznak, felmerülnek adatvédelmi aggályok az adatok sorsát illetően (Banh és mtsai., 2025). Amennyiben az adatok nem oszthatók meg az MI modellel, az nagyban korlátozza a hatékonyságot. Megoldás lehet az is, ha egy vállalat saját maga futtat on-premise MI modellt, de ennek komoly hardverberuházási költsége van (a nagyméretű modellek futtatásához olyan nagy hardverkapacitásra van szükség, amelyet egyetlen vállalat reálisan nem tud hatékonyan kihasználni). Ennek alternatívája, amikor a modellek

virtuális privát felhőben futnak, vagyis a hardvert nem kell megvásárolni, mégis biztonsági mechanizmusok gátolják meg a vállalati adatok kiszivárgását.

Brainstorming. A projekttervezés potenciális eleme az ötletgyűjtés, vagyis a lehetséges fejlesztési irányok, megoldások, kiegészítő funkciók összegyűjtése. A brainstorming célja nem a mély technikai elemzés, hanem a lehetőségek minél sokszínűbb feltérképezése. A generatív MI ebben kifejezetten hatékony, hiszen képes a megadott célok és korlátok mentén nagyszámú ötletet és hivatkozást összegyűjteni.

Az ilyen típusú ötletgenerálás különösen hasznos akkor, amikor a projekt kezdeti szakaszában „vakfoltok" merülhetnek fel, illetve amikor a csapatnak nincs kellő kapacitása szisztematikus kutatásra. Fontos ugyanakkor kiemelni, hogy a GMI nem helyettesíti a mély domain-szakértelmet, inkább csak széleskörű, de gyakran felszínes irányokat tud mutatni, amelyek közül már az emberi tudás és intuíció alapján érdemes választani.

Értékelés. Az ötletelést követően elengedhetetlen az egyes koncepciók részletes értékelése, vagyis annak vizsgálata, hogy az adott javaslat mennyire életképes, milyen előnyei és hátrányai vannak, illetve milyen hozzáadott értéket képvisel. Ebben a lépésben a GMI képes strukturált érveket generálni (pro-kontra listák), rámutatva ezáltal olyan szempontokra, amelyek a fejlesztőcsapat számára elsőre nem kézenfekvőek, különösen, ha a csapat összetétele nem fedi le az összes releváns szakterületet. A panelbeszélgetések résztvevői is megemlítették, hogy a modellek gyakran váratlan, de releváns szempontokat hoznak fel egy adott ötlettel kapcsolatban.

Az értékelés fontos része lehet a rendelkezésre álló adatokra épülő elemzés is, pl. hogy egy új funkciót valóban használna-e a felhasználói bázis. A GMI képes rövid szöveges összefoglalókat generálni nagy mennyiségű analitikai adatból és felhasználói visszajelzésből, ezzel hatékonyan támogatva a döntés-előkészítési folyamatot. Ennek a felhasználásnak természetesen előfeltétele, hogy megfelelő mennyiségű és minőségű adat áll rendelkezésre, amelyekből az MI modell dolgozni tud. Általánosságban is elmondható, hogy adat nélkül hatékony MI használat sincs.

Kockázatelemzés. A projekttervezés során elengedhetetlen a kockázatok feltárása. A potenciális kockázatok tárháza szinte végtelen: technológiai, erőforrásbeli, üzemeltetési, biztonsági vagy megfelelőségi (compliance) kockázatokra egyaránt fel kell készülni. A GMI ebben a szakaszban jó kiegészítő eszköz lehet, hiszen rengeteg korábbi példát, iparági esetleírást, legjobb gyakorlatot (best practice) és dokumentációt használtak fel a modell tanítása során, amelyekből kiindulva képes lehet releváns kockázatokat azonosítani.

Az MI modellek ugyanakkor egyelőre nem helyettesíthetik a szakértők munkáját, hiszen csak a múltbeli példákból és az azokon felismert mintázatokból tud dolgozni, holott a kockázatok kimerítő feltárása a gyakorlatban gyakran igényel kreativitást. Ráadásul az adott szervezet egyedi kontextusát sem szabad figyelmen kívül hagyni, amely nem biztos, hogy kimerítően beemelhető a modell kontextusába. A GMI tehát ebben a szakaszban inkább a gondolkodási horizont szélesítésében, mintsem a döntéshozatalban játszik szerepet.

Becslés. Egy projekt sikeressége többek között az idő- és erőforráskeretek betartásában mérhető, amelyek alapja a reális idő- és erőforrásbecslés. A generatív MI erre is használható: a modell a meglévő kódbázis, az érintett komponensek, korábbi hasonló projektek és fejlesztői becslések alapján képes előzetes idő- és munkaigénybecslést kalkulálni (Nguyen Duc és mtsai., 2023). A panelbeszélgetések során is említették, hogy egy új funkció fejlesztéséhez szükséges időt a specifikáció és a forráskód fontosabb részei alapján a GMI modellek képesek meglepően pontosan megbecsülni.

Dokumentumgenerálás. A projekttervezés végső kimenete általában formális dokumentum, mint pl. projektterv, kezdeti specifikáció, követelménylista, kockázatelemzés vagy becslési táblázat. A generatív MI jelentősen felgyorsíthatja ezeknek a dokumentumoknak az előállítását. A modell képes a rendelkezésre álló adatok, feljegyzések, részeredmények alapján jól strukturált, koherens dokumentumokat generálni, amelyek és formai szempontból illeszkednek korábbi referenciamunkákhoz (Zeichick, 2024).

A gyakorlatban ez nemcsak időt takarít meg, hanem javíthatja a dokumentumok minőségét is, mivel a modell képes konzisztens terminológiát, egységes szerkezetet és átlátható formátumot biztosítani. Ugyanakkor fontos látni, hogy a generált dokumentumokat minden esetben emberi utóellenőrzésnek kell alávetni, különösen tartalmi szempontból, hiszen a projekt további sikeressége szempontjából elengedhetetlen a pontosság garantálása.

4.2.2. Elemzés

Az elemzés során határozzuk meg és dokumentáljuk a szoftverrel szemben támasztott követelményeket. A követelmények célja, hogy a projekt minden résztvevője számára teljesen egyértelmű legyen, hogy a készítendő megoldásnak pontosan mire kell *képesnek lennie* (funkcionális követelmények) és *milyennek* kell lennie, vagyis milyen minőségi jellemzőkkel kell rendelkeznie a megoldásnak (extrafunkcionális követelmények). MI alapú követelményelemző eszközök léteznek ugyan a gyakorlatban is, de a GMI ilyen irányú felhasználása egyelőre inkább tűnik jövőbeli elképzelésnek, mint múltbeli gyakorlatnak.

Követelményszármaztatás. A követelmények jelentős része származhat különböző dokumentumokból, amelyek tartalmát adottságként kell kezelni. Ezek egyaránt lehetnek belső vállalati (pl. illesztendő meglévő rendszerek tulajdonságai, belső információbiztonsági szabályzatok) és vállalaton kívüli (pl. szabályozói előírások – különösen szigorúan szabályozott nagyvállalati környezetben) források. A GMI képes lehet kinyerni ezekből a forrásokból a lényeges információkat, majd strukturálni és formális követelményekké alakítani azokat (Rico & Öberg, 2025), vagy pontosítani a már meglévő követelményeket, esetleg rámutatni hiányzó követelményekre. Az MI modell felismeri a követelményként értelmezhető mondatokat, és képes azokból egységes szerkezetű, pontos követelményeket származtatni. Ez különösen hasznos olyan projekteknél, ahol nagy mennyiségű kiinduló forrás tartalmaz potenciális követelményeket, ezek a források pedig heterogének, eltérő formátumúak.

Fontos ugyanakkor rámutatni, hogy egy szoftverprojekt szempontjából a követelmények összegyűjtése kritikus fontosságú. Ennek a fázisnak a precizitásán nagyban múlik a

később szükséges iterációk száma, így az MI által származtatott követelményeket mindenképp szükséges emberi utóellenőrzés alá vetni. Kritikus fontosságú továbbá a követelmények teljességének az ellenőrzése, vagyis a megbizonyosodás arról, hogy valóban minden releváns követelmény összegyűjtésre került.

Követelményosztályozás. A követelményeket általában osztályozzuk a jellegük szerint, vagyis az alapján, hogy mire vonatkoznak. Beszélhetünk funkcionális és extrafunkcionális követelményekről, utóbbi kategórián belül többek között teljesítményre, megbízhatóságra, biztonságra, megfigyelhetőségre, kompatibilitásra, karbantarthatóságra, használhatóságra vonatkozó követelmények. A GMI segíthet az elkészült követelmények osztályozásában (Nguyen Duc és mtsai., 2023), ami elősegíti a rendezett követelménystruktúrát, és átláthatóbb dokumentációt eredményez.

Követelményminőség. Bár a követelmények természetes nyelven kerülnek megfogalmazásra, a későbbiekben viszont egzaktan kell tudnunk ellenőrizni a megvalósulásukat, ezért kritikus fontosságú a precíz megfogalmazás. Ennek biztosítására különböző szabályok léteznek a megfogalmazására, amelyek célja a követelmények egyediségének, azonosíthatóságának, egyértelműségének és tesztelhetőségének biztosítása. Kerülni kell tehát a túl általános, homályos, félreérthető megfogalmazásokat, amelyekről nem dönthető el egyértelműen, hogy megvalósultak-e. A nagy nyelvi modellek képesek lehetnek ellenőrizni ezeket a megfogalmazásbeli elvárásokat, különösen, ha az elvárások minél részletesebben megfogalmazva állnak rendelkezésre.

Megvalósíthatóság. A követelmények összegyűjtése után meg kell vizsgálni, hogy azok megvalósíthatók-e. A megvalósíthatóság több síkon is értelmezhető, mint pl. technikailag, üzletileg és erőforrások szempontjából. A megvalósíthatóság szükséges, de nem elégséges alapfeltétele a követelmények ellentmondás-mentessége. A generatív MI segíthet a potenciális ellentmondások, egymással nem összhangban lévő követelmények azonosításában, ami főleg nagy számú/terjedelmű követelmények esetében lehet különösen hasznos. A modellek akár képesek is lehetnek megoldási javaslatokat tenni az ellentmondások feloldására. Ugyanakkor az értelmezés és döntéshozatal továbbra is emberi feladat, hiszen a megvalósíthatóság végső megítélése mély szakterületi tudást igényel.

4.2.3. Rendszertervezés

A rendszertervezés során definiáljuk a rendszer logikai és technikai architektúráját, az adatmodelleket, valamint a komponensek közti kapcsolatokat és interfészeket. Gondos tervezéssel lehetséges az esetleges hibák korai feltárása, ami nagyban elősegíti a későbbi folyamatok (különösen a fejlesztés és a tesztelés), így összességében az egész SDLC hatékonyságát. A megkérdezettek tapasztalatai alapján a részletes rendszertervezést nem adják ki emberi kézből, de prototípusok generálására már most is aktívan használják a GMI-t.

Technológiaválasztás. A technológiaválasztás során a fejlesztőcsapatnak meg kell határoznia, hogy a projekt mely eszközökre, keretrendszerekre, könyvtárakra és infrastruktúrára támaszkodik. Mivel vállalati környezetben a rendszerek jellemzően hosszútávra készülnek, a kiválasztás során nemcsak a jelen körülményeit kell figyelembe venni, hanem azt is, hogy a választott technológiák támogatása hosszútávon biztosított-e. Sőt, az is fontos mérlegelési szempont lehet, hogy mennyire közismert a technológia, hiszen ez közvetlenül befolyásolja, hogy mennyire könnyen (és milyen költségen) lehet fejlesztőt találni a projektre. A generatív MI képes lehet feltérképezni a potenciális megoldásokat (ideértve a *state of the art*-ot is), összefoglalni a főbb előnyöket és hátrányokat, és mindezek alapján javaslatokat megfogalmazni (Banh és mtsai., 2025; Rico & Öberg, 2025). Ezek az ötletek segíthetnek ugyan a döntéshozók perspektívájának tágításában, de az MI modell javaslatainak kritikátlan felhasználása helyett nem szabad szem elől téveszteni a döntések súlyát, hiszen a technológiaválasztás hosszútávú következményekkel járhat az üzemeltetésre, a teljesítményre, a biztonságra és a bővíthetőségre nézve.

Prototipizálás. A rendszertervezés fontos támogató tevékenysége a prototípusok készítése, amelyek célja a koncepciók gyors validálása és a magas szintű megoldási irányok gyakorlati kipróbálása. A generatív MI ebben a fázisban képes gyorsan létrehozni egyszerű proof-of-concept (PoC) vagy minimális életképes termék (minimal viable product, MVP) jellegű megoldásokat, beleértve az alapvető infrastruktúra-komponenseket vagy egyszerű felhasználói felületeket. Ez jelentős időmegtakarítást eredményezhet, különösen akkor, ha a projekt több lehetséges architektúrát is vizsgál, lehetővé téve azok összehasonlítását. Az összehasonlításon túl ez a módszer egy új technológia megismerésekor is hasznos lehet, hiszen egy egyszerű, ám mégis működő példán keresztül sokszor könnyebben elsajátítható az új tudás. Fontos azonban hangsúlyozni, hogy a prototípusokat a helyükön kell kezelni, azok semmiképp sem tekinthetők végleges terveknek. Az MI modellek által generált kód és architektúra sok esetben csak demonstrációs célokra alkalmas, és az sem garantált, hogy hűen reprezentálja az adott technológia használhatóságát.

Tervezés. A GMI képes lehet egyszerűbb komponensek konkrét megtervezésére is, mint pl. egy egyszerű API (application programming interface, a szoftverkomponensek közti kommunikáció interfésze), valamint a vonatkozó metrikák (pl. szükséges tárhely, memória, sávszélesség) becslésében is segítséget nyújthat. Fontos azonban kiemelni, hogy még ha elsőre hívogatónak is tűnhet egy komplett rendszerterv legenerálása, valójában nagyon veszélyes. Egy komplex rendszerterv, különösen a hozzá kapcsolódó dokumentációk olyan összetettek és terjedelmesek, hogy azoknak a kellően tüzetes emberi utóellenőrzése egész egyszerűen irreális. A megkérdezettek ráadásul egyetértettek abban, hogy egy nem általuk készített komplex megoldás kellő részletességű átnézése valójában gyakran nehezebb és nagyobb szellemienergia-befektetést igényel, mint ha ők maguk készítették volna el, így pedig nagyon könnyű átsiklani az MI által vétett hibák felett.

4.2.4. Fejlesztés

A fejlesztés során ténylegesen implementáljuk a szoftvert, vagyis megvalósítjuk a rendszertervet. A megvalósításnak teljesítenie kell a korábban meghatározott követelményeket. Ebben a fázisban általában többnyire kódolás zajlik, amely eredményeként előáll a szoftver forráskódja. A GMI alkalmazása itt a legkézenfekvőbb és legelterjedtebb, amit több dolog is alátámaszt: egyrészt ehhez a fázishoz áll rendelkezésre a legtöbb GMI céleszköz, másrészt ezt (és a tesztelést) taglalja a legtöbb publikáció, harmadrészt a megkérdezetteknek is a kódolást illetően volt a legtöbb ötlete és tapasztalata.

Kódgenerálás. A GMI kódgenerálásra (természetes nyelvi utasításokból programkód előállítása) való felhasználása jelentős időmegtakarítást eredményezhet, különösen a kevés tényleges gondolkodást igénylő, repetitív feladatok esetében. Ilyenek lehetnek többek között sablon (boilerplate) kódok, integrációs megoldások vagy a felhasználói felület (user interface, UI) egyszerű komponensei. A tapasztalatok szerint minél kevésbé komplex a feladat, annál hasznosabb a GMI, összetett üzleti logika vagy komplex technikai megoldások esetén viszont a modellek gyakran hibás vagy félrevezető kódot generálnak, és a javítás (vagy a modell rábírása a javításra) több erőfeszítést igényel, mint az önálló megírás (Karaci Deniz és mtsai., 2023; Liang és mtsai., 2024; Modi, 2024). A panelbeszélgetések résztvevői úgy foglalták össze ezt a jelenséget, hogy komplex esetekben hiába kezdik el lelkesen használni az MI-t a gondolkodás megspórolására, végül általában kiderül, hogy az *igazán* bonyolult problémák végiggondolása, vagyis az *igazi* gondolkodás nem spórolható meg.

A GMI-alapú kódgenerátorok alkalmazhatósága szempontjából kulcskérdés az adott szoftverkomponens szerepe: mivel a kódba kerülhetnek nehezen észrevehető hibák, a kevésbé fontos, nem kritikus részeken érdemes alkalmazni a technológiát. Azt is érdemes szem előtt tartani, hogy programnyelvenként eltérő garanciák vannak arra vonatkozóan, hogy milyen hibák milyen korán derülnek ki: míg egyes nyelvek lehet, hogy már le sem fordulnak, bizonyos szkriptnyelvek esetén elképzelhető, hogy csak jóval a bevezetés után, élesben derül majd fény egy hibára.

Megmagyarázás. A generatív MI jól használható a meglévő kódrészek megértésének támogatására, ami különösen fontos nagy, rosszul dokumentált kódbázisok esetén. A modell képes összefoglalni, hogy egy adott függvény, osztály vagy modul mi célt szolgál, milyen bemeneteket és kimeneteket kezel, milyen kapcsolatban áll más komponensekkel, és mi a szerepe az egész rendszerben. Ez nagyban gyorsíthatja a fejlesztők munkáját, különösen új belépők vagy a projektbe frissen bekapcsolódó fejlesztők számára. A GMI megmagyarázó képessége újonnan írt kódok esetén is használható, többek között az új forráskód kommentezésével (kód ellátása magyarázó megjegyzésekkel) vagy a változások összefoglalásával (pl. verziókezelő számára).

Refaktorálás. Refaktorálásnak nevezzük egy programkód átalakítását, a viselkedés módosítása nélkül, pusztán azért, hogy javuljon a kód struktúrája, olvashatósága, karbantart-

hatósága. A GMI hatékonyan támogathatja kisebb, jól körülhatárolt refaktorálási feladatok elvégzését, mint pl. változóátnevezés, kisebb logikai letisztázások, duplikációk eltávolítása, függvények szétbontása. Nagyobb léptékű, átfogó módosítások esetén azonban komoly kockázatot jelent, hogy a modell által generált változtatások olyan nagyok, hogy a fejlesztők már nem látják át teljesen, így végül vakon megbíznak az MI modellben. Ez magában hordozza a viselkedés akaratlan módosítását, ami olyan hibákat injektálhat a rendszerbe, amelyekre potenciálisan csak később, élesben derül fény. A megkérdezettek tapasztalatai is azt támasztották alá, hogy az MI által végzett refaktorálásokat mindenképp le kell ellenőrizni, mert könnyen végezhet nem várt, helytelen módosításokat is.

Hibakeresés. A fejlesztési folyamat szerves részét képezi a készülő kód folyamatos kipróbálása és az iteratív hibakeresés, amelyben a generatív MI szintén nagy segítséget nyújthat. A modellek képesek értelmezni a kapott hibaüzeneteket, és a hibás kódban rámutatni a hibák tényleges okaira. A hibák szöveges magyarázatán túl képesek javaslatot tenni a javításra vagy akár konkrétan legenerálni a javított változatot. Ez még az egyszerű szintaktikai hibák esetén is hasznos lehet (pl. egy hosszú lekérdezésben könnyű apró hibákat véteni, amelyeket egyesével kijavítani jóval tovább tart, mint beadni az egészet az MI modellnek), de különösen hasznos, amikor a hiba valódi oka nem egyértelmű, egyszerűen csak egy váratlan viselkedésre keresünk magyarázatot.

Szkriptek. A fejlesztés során gyakran adódnak olyan feladatok, amelyek nem tartoznak a fejlesztő szigorúan értelmezett szakterületéhez, de mégis meg kell őket oldania. Ez jelentheti akár bizonyos fejlesztői lépések automatizálását, akár adatok kinyerését fájlokból, a közös az bennük, hogy bár a feladatok egyszerűek, a praktikusan használt nyelvben (általában szkriptnyelv, pl. bash, awk) gyakran kevésbé magabiztos a fejlesztő, mert csak ritkán kényszerül használni. Ezek a feladatok tipikusan nem hordoznak jelentős kockázatot, és a generált megoldás is elég tömör és egyszerű ahhoz, hogy „ránézésre" vagy első kipróbálásra megállapítható a helyessége, ugyanakkor jelentős időmegtakarítást eredményezhet, hogy nem kell egy kontextusváltás árán belemélyedni a ritkán használt nyelvbe. Bár nem szorosan tartozik ide, de mégis itt érdemes megemlíteni a reguláris kifejezések írását és értelmezését is, amiben szintén nagyon hasznos segítség lehet a GMI.

Konvertálás. A fejlesztés során esetenként szükség van különböző fájlformátumok (pl. JSON, XML, CSV) közötti átalakításra, amit a GMI magabiztosan képes elvégezni. Az is előfordul, hogy egy adott programnyelven rendelkezésre álló logikát egy másik programnyelvben is elő kell állítani. Ez ugyan már jóval komplexebb feladat, mint egy fájlkonverzió, a tapasztalatok azt mutatják, hogy erre is használhatók a generatív MI modellek (Panetta, 2023; Zeichick, 2024). A programnyelvek közti fordítás esetén is igaz, ami általánosságban is elmondható az MI által generált éles rendszerbe szánt kódokról: fontos megbizonyosodni a helyességről, akár emberi ellenőrzéssel, akár alapos teszteléssel.

4.2.5. Tesztelés

A tesztelés a szoftverfejlesztési életciklus kritikus fontosságú szakasza, melynek célja a szoftver minőségének biztosítása, a hibák feltárása, valamint annak igazolása, hogy az elkészült megoldás megfelel az előre meghatározott funkcionális és extrafunkcionális követelményeknek. A tesztelés hagyományosan erőforrásigényes és gyakran monoton tevékenység, amely azonban elengedhetetlen a későbbi, éles környezetben felmerülő, sokkal költségesebb hibák elkerülése érdekében. Mivel a tesztelés jelentős részét a tesztek megírása (kódolása) teszi ki, a fejlesztéshez hasonlóan itt is hatékony segítséget jelenthetnek a generatív MI megoldások. Ez a potenciál megmutatkozik a céleszközök és vonatkozó publikációk számában, és a megkérdezettek beszámolóiban is.

Tesztgenerálás. A fejlesztők körében az egyik legnépszerűbb felhasználási mód a tesztek, különösen az egységtesztek (unit test) generálása (amely egy atomi funkcionális egység működését tesztelik). A GMI modellek képesek a forráskód alapján automatikusan teszteseteket írni, lefedve számos bemeneti kombinációt és végrehajtási ágat (Modi, 2024; Petry & Carlson, 2024). Ez jelentősen növelheti a kódlefedettséget és csökkentheti a tesztírásra fordított időt.

Ugyanakkor fontos tisztában lenni egy súlyos módszertani korláttal: ha a teszteket a forráskódból származtatjuk (különösen, ha ugyanazzal a GMI modellel generáljuk őket, amely a forráskódot is generálta), az valójában *fehér dobozos* tesztelést (white box testing) eredményez, ahol a tesztkészlet a kód belső működését tükrözi, nem pedig a specifikációt. Ha a kód hibás logikát vagy struktúrát tartalmaz, a generált tesztkészlet potenciálisan ehhez a hibás logikához fog illeszkedni, így hiába fut le sikeresen egy teszt, a tesztelni kívánt, de valójában nem tesztelt hiba mégis rejtve marad. A valódi minőségbiztosításhoz elengedhetetlen, hogy a tesztek a kódtól függetlenül, az elvárások (követelmények) alapján (is) készüljenek (*fekete dobozos* tesztelés, black box testing).

Tesztadat-generálás. A tesztelés során gyakori probléma a megfelelő minőségű és mennyiségű tesztadat előállítása. A valós éles adatok használata szigorú adatvédelmi és biztonsági kockázatokat vet fel, a kézi adatgyártás pedig lassú és nem skálázható. A generatív MI képes szintetikus, de a valós adatok statisztikai jellemzőit és struktúráját hűen tükröző tesztadatokat előállítani (Petry & Carlson, 2024). Ez lehetővé teszi a fejlesztők számára, hogy biztonságosan teszteljenek határértékeket, kivételes eseteket vagy nagy adathalmazokat igénylő teljesítménytesztet végezzenek. Korlátot jelenthet ugyanakkor, hogy a generálás nem feltétlenül szisztematikus: a modell tanult minták alapján dolgozik, nem pedig algoritmikusan, így előfordulhat, hogy bizonyos speciális, de kritikus adatkombinációkat magától nem állít elő, hacsak a promptban erre külön nem utasítjuk.

Integrációs tesztelés. Míg a unit tesztek a kód legkisebb funkcionális egységeit vizsgálják, az integrációs tesztelés a komponensek közötti együttműködés ellenőrzésére szolgál. Ez a terület komplexebb, mivel gyakran külső függőségeket, adatbázisokat vagy hálózati kommunikációt igényel. A fejlettebb, agentikus MI megoldások képesek lehetnek arra, hogy

autonóm módon elindítsák a szükséges kódokat, felépítsék a tesztkörnyezetet, és szkriptek segítségével szimulálják és ellenőrizzék a komponensek közötti interakciókat (Petry & Carlson, 2024). A GMI segíthet a komponenseket összekötő „ragasztó" kódok (glue code) és a teszteléshez szükséges konfigurációk elkészítésében is, ami jelentős terhet vesz le a tesztautomatizáló mérnökök válláról.

Kimenetellenőrzés. A regressziós tesztelés során – amikor azt vizsgáljuk, hogy egy módosítás nem rontott-e el korábban már működő funkciókat – gyakran keletkeznek nagyméretű naplófájlok (logok) és kimeneti állományok. Ezek manuális átnézése rendkívül időigényes és az emberi figyelmet próbára tevő feladat. A generatív MI modellek hatékonyan használhatók a tesztfutások eredményeinek összehasonlítására. A modell képes összevetni az elvárt és a kapott kimenetet, kiemelni az eltéréseket, sőt, a hibaüzenetek alapján akár előzetes diagnózist is felállíthat a hiba lehetséges okáról, felgyorsítva ezzel a hibajavítási ciklust.

Kompatibilitásellenőrzés. A szoftverek fejlődésének természetes velejárója a változás, azonban bizonyos helyzetekben fontos szempont a visszafelé kompatibilitás (backward compatibility) biztosítása, különösen API-k vagy mások (akár szervezete belül, akár azon kívül) által is használt könyvtárak esetén. A generatív MI segíthet a tervezett változtatások elemzésében, összevetve a régi és az új verzió interfészét és belső működését. A modell képes azonosítani azokat a módosításokat (pl. egy függvény szignatúrájának megváltozását vagy a belső működés nem kívánt megváltozását), amelyek problémát okozhatnának a szoftvert használóknál, még azelőtt, hogy a módosítás éles környezetbe kerülne.

Kódátvizsgálás. A kódátvizsgálás (code review) kettős szerepet tölt be a GMI kontextusában. Egyrészt az ember által írt kód MI általi ellenőrzése értékes eszköz: a modellek *fáradhatatlanul* képesek kiszűrni a szintaktikai hibákat, a biztonsági sérülékenységeket, vagy javaslatokat tenni a kód stílusának javítására (Modi, 2024; Petry & Carlson, 2024; Zeichick, 2024), míg emberi ellenőrzés esetén számolni kellene a figyelem lankadásának veszélyével.

Másrészt viszont komoly kihívást jelent az MI által generált kód emberi felülvizsgálata. A kutatások és a gyakorlati tapasztalatok azt mutatják, hogy ha a kódot nagyrészt az MI generálja, az emberi fejlesztő hajlamos a felületes ellenőrzésre. Egyrészt pszichológiailag nehezebb és unalmasabb más (vagy egy gép) kódját olvasni és értelmezni, mint sajátot írni. Másrészt az MI által generált kód szintaktikailag általában helyes és „szépnek" tűnik, ami hamis biztonságérzetet ad. Így a fejlesztők könnyen átsiklanak olyan logikai vagy szemantikai hibák felett, amelyeket saját kódolás közben nem vétettek volna el, vagy azonnal észrevettek volna. Paradox módon tehát az MI által generált kód ellenőrzése esetenként nagyobb kognitív terhelést és fegyelmet igényel, mint a kódolás maga.

4.2.6. Bevezetés

A bevezetés fázis képez hidat a fejlesztés és az üzemeltetés között, amely során az elkészült szoftver kikerül a védett, elszeparált fejlesztői környezetből és élesben elérhetővé válik a végfelhasználók számára. Ez a szakasz különösen kritikus, hiszen bármilyen hiba közvetlen hatással van a szolgáltatás elérhetőségére és az *üzletmenet folytonosságára*. Bizonyára ezzel is magyarázható, hogy ebben a fázisban a GMI használata egyelőre inkább csak spekulációnak tűnik, valós ipari tapasztalatok nemigen állnak rendelkezésre.

Infrastruktúrakód. A modern felhőalapú környezetekben az infrastruktúra kezelése már nem manuális kattintgatással, hanem kód formájában történik (Infrastructure as Code, IaC). A generatív MI modellek képesek legenerálni vagy módosítani ezeket a deklaratív konfigurációs fájlokat. Mivel ezek a leíró nyelvek szigorú struktúrával rendelkeznek, az MI képes a természetes nyelven megfogalmazott igényeket szintaktikailag helyes konfigurációs kóddá alakítani.

Függőségelemzés. A bevezetés előtt álló szoftvercsomagok gyakran tartalmaznak külső könyvtárakat és modulokat. A generatív MI segíthet a függőségek (dependency) elemzésében, feltárva az esetleges verzióütközéseket vagy ismert biztonsági sérülékenységeket. Ez azért lehet különösen hasznos, mert ezek a hibák gyakran akár hosszú ideig is rejtve maradhatnak, és csak később, éles futás közben, váratlanul okoznak problémát.

Dokumentáció. A bevezetés elengedhetetlen része a változások dokumentálása (enélkül ugyanis többek között elképzelhetetlen a hatékony változásmenedzsment). A fejlesztés és tesztelés során keletkező technikai információkból (commit üzenetek, hibajegy, CI/CD logok) a generatív MI képes meghatározott célú dokumentációt generálni. Ez egyaránt jelentheti a fejlesztőknek és üzemeltetőknek szóló technikai leírásokat, vagy a végfelhasználóknak szánt közérthető verzióinformációkat (release notes) és felhasználói kézikönyveket (Petry & Carlson, 2024). Ez azért is lehet hiánypótló a gyakorlatban, mert a fejlesztők általában nem szívesen, csak muszájból írnak dokumentációt, így annak a mennyisége, minősége és naprakészsége gyakran nem kielégítő.

Végrehajtás. Az autonóm ágensek megjelenésével megnyílt a lehetőség arra, hogy az MI hajtsa is végre a telepítési folyamatot. Ez azonban rendkívüli kockázatot hordoz: ahhoz, hogy egy ágens telepíteni tudjon, írási hozzáférést kell kapnia az éles (production) környezethez. Mivel a nagy nyelvi modellek működése nem determinisztikus, a közvetlen hozzáférés katasztrofális következményekkel járhat. Ezt támasztja alá az a közelmúltbeli eset is, amikor egy MI alapú kódoló eszköz véletlenül törölte egy vállalat éles adatbázisát, majd kérdésre „letagadta”, és azt állította, hogy nem történt törlés (Chong Ming, 2025). Az eset rávilágít arra, hogy az emberi felügyelet nélküli közvetlen hozzáférés az éles környezethez jelenleg elfogadhatatlan kockázatot jelent.

Folyamat definiáltsága. A bevezetéssel kapcsolatban kirajzolódik egy érdekes paradoxon a generatív MI szerepét illetően. A sikeres automatizáció kulcsa a folyamat pontos definiálása. Ha egy szervezetnél a telepítési folyamat nincs pontosan definiálva (ad-hoc jellegű), akkor az MI ágens sem fogja tudni sikeresen végrehajtani, hiszen hiányzik a kontextus és a szabályrendszer. Ha viszont a folyamat lépései (pipeline) precízen, egzaktan definiálva vannak, akkor azok leprogramozhatók hagyományos, determinisztikus eszközökkel (szkriptek, CI/CD eszközök), amelyek utána 100%-os megbízhatósággal működnek. Ebben az esetben a nemdeterminisztikus, potenciálisan hibázó GMI használata a végrehajtásra felesleges kockázatot visz a rendszerbe. A GMI helye tehát nem a folyamat *futtatásában*, hanem a folyamatot leíró automatizációs kódok *létrehozásában* van.

4.2.7. Karbantartás

A szoftverek életciklusa nem ér véget a sikeres bevezetéssel, az ezt követő karbantartási fázis időben gyakran a leghosszabb az összes fázis közül, hiszen különösen nagyvállalati környezetben a szoftverek gyakran évtizedekig üzemelnek. Ebben a fázisban a cél a rendszer stabil működésének biztosítása, a felhasználók támogatása, az esetlegesen felmerülő hibák javítása, valamint a potenciális továbbfejlesztési igények kezelése. A GMI vonatkozó felhasználása értelemszerű lenne, de többségében egyelőre mégis inkább csak vágyálom – kivéve az ügyféltámogatást, hiszen ilyen chatbotokkal már bizonyára mindenki találkozott.

Hibajegy-előfeldolgozás. Az üzemeltetés során beérkező hibajegyek (ticket) kezelése jelentős emberi erőforrást igényel, különösen azért, mert a bejelentések gyakran pontatlanok, hiányosak vagy duplikáltak. A generatív MI képes a beérkező természetes nyelvű bejelentések előfeldolgozására: osztályozza a hibákat súlyosság és típus szerint, valamint összeveti azokat más esetekkel, hogy azonosítsa az ismétlődéseket. Amennyiben a bejelentésből hiányoznak nélkülözhetetlen adatok, a modell automatikusan bekérheti ezeket a bejelentőtől még azelőtt, hogy a jegy egy fejlesztőhöz kerülne. Ezen felül az MI képes lehet egy hibajegy szempontjából releváns naplófájlok beazonosítására is, sőt, megfelelő jogosultságokkal akár közvetlenül csatolhatja is azokat a hibajegyhez.

Hibaokfeltárás. Egy hiba részletes kivizsgálása, különösen a tényleges kiváltó ok megtalálása gyakran nehéz fejlesztői feladat, hiszen nagy mennyiségű információ (forráskód, naplófájlok, stack trace, hibajegy) között kell megtalálni az összefüggéseket. A generatív MI modellek hatékonyan támogatják a kiváltóok-elemzést (root cause analysis, RCA): a rendelkezésre álló információk alapján képesek hipotéziseket felállítani a hiba lehetséges okairól (Petry & Carlson, 2024; Zeichick, 2024). Bár a modell által javasolt diagnózis nem minden esetben pontos, mégis gyakran jelent hasznos kiindulási pontot, csökkentve ezzel a hiba kijavításához szükséges időráfordítást.

Teljesítményelemzés. A karbantartás nemcsak a hibajavításról, hanem a rendszer hatékonyságának fenntartásáról is szól. A rendszerekbe az élettartamuk során jellemzően egyre több adat kerül, aminek a hatékony kezelése pl. skálázódási kihívásokat jelenthet

a későbbiekben. A generatív MI segíthet a teljesítménybeli szűk keresztmetszetek (bottleneck) azonosításában. A kód statikus elemzésével vagy a futásidejű profilozási adatok értelmezésével a modell rámutathat azokra a kódrészletekre (pl. nem hatékony adatbázislekérdezések, felesleges fájlműveletek, cache-elés hiánya), amelyek lassítják a rendszer működését, és optimalizációs javaslatokat tehet azok javítására.

Ügyféltámogatás. A generatív MI egyik leglátványosabb felhasználási területe a felhasználókkal való közvetlen kapcsolattartás. A nagy nyelvi modellekre épülő chatbotok képesek intelligens ügyfélszolgálati (customer service) feladatokat ellátni, válaszolva a gyakori kérdésekre vagy segítve a felhasználókat a szoftver használatában (Gartner, 2025). Ez tehermentesíti az emberi ügyfélszolgálatot, akik így a komplexebb problémákra fókuszálhatnak. Emellett a GMI felhasználható a felhasználói visszajelzések (feedback) elemzésére és összegzésére is, segítve a termékmenedzsereket abban, hogy a nagy mennyiségű szöveges értékelésből kiszűrjék a leggyakoribb igényeket és panaszokat. Fontos azonban a kockázatok szem előtt tartása és kezelése: a közvetlen ügyfélkommunikációban az MI tévedései (pl. hallucináció) reputációs károkat okozhat.

5. fejezet

Következtetések

Ebben a fejezetben kutatási eredményeim összefoglalásául következtetéseket fogalmazok meg: az 5.1. alfejezetben a generatív mesterséges intelligencia megítélését illetően, az 5.2. alfejezetben a technológia szerepéről és lehetséges felhasználásairól, az 5.3. alfejezetben a kapcsolódó kihívások és tapasztalatok vonatkozásában. Az 5.4. alfejezetben bemutatok egy akciólistát a generatív MI sikeres bevezetéséhez, végül az 5.5. alfejezetben értékelem az alkalmazott kutatásmódszertant.

5.1. A generatív mesterséges intelligencia megítélése

A kutatás során kirajzolódott kép alapján a szoftverfejlesztők a generatív mesterséges intelligenciára (GMI) alapvetően mint egy nagy tudású, univerzális, fáradhatatlan, ám korántsem tévedhetetlen asszisztensre tekintenek. A technológia megítélése kettős: miközben elismerik a benne rejlő kényelmet és potenciált, a szakemberek tisztán látják annak (jelenlegi) korlátait is.

Univerzális. A GMI lett a modern szoftverfejlesztés kalapácsa: *akinek kalapács van a kezében, az hajlamos mindent szögnek nézni* (Maslow, 1966). Mivel a nagy nyelvi modellek univerzálisak és rendkívül kényelmesen használhatók természetes nyelven, a fejlesztők hajlamosak olyan feladatokra is ezt használni, amelyekre léteznek pontosabb, megbízhatóbb céleszközök. A GMI „mindenhez ért”, de speciális technikai feladatokban (pl. statikus kódanalízis, refaktorálás) gyakran elmarad a determinisztikus algoritmusok pontosságától. A gyakorlatban a kényelem azonban sokszor felülírja a racionalitást, ami kockázatot jelent.

Absztrahálás hiánya. A megkérdezettek szerint a generatív MI jelenlegi formájában nem képes az emberi gondolkodás egyik legfontosabb elemére, az *absztrahálásra*. Mivel a modellek a meglévő adatokon, a múltban keletkezett mintázatokon tanulnak, működésükből hiányzik a valódi kreativitás, az igazi „out of the box” gondolkodás. A szoftverfejlesztés lényege gyakran éppen az egyedi, korábban nem látott (üzleti) problémák leképezése absztrakt modellekre, amit a jelenlegi GMI eszközök önállóan nem tudnak elvégezni. Így pedig

a szoftverfejlesztés legértékesebb és legfárasztóbb része, azaz a problémák *megértése* és az *„igazi" gondolkodás* továbbra sem spórolhatók meg.

Megbízhatóság. A technológia megítélését negatívan befolyásolják a megbízhatósági problémák, különösen a hallucináció jelensége. A fejlesztők számára frusztráló, hogy az eszközök gyakran magabiztos stílusban közölnek teljesen téves információkat. Szintén negatív jelenségként értékelhető a tartalom felesleges felduzzasztása. Abszurd helyzeteket teremt, amikor az egyik MI modell által generált, információban ritka, de terjedelmes szöveget egy másik modellel kell összefoglaltatni a hatékony feldolgozáshoz. Ez rávilágít arra, hogy a technológia jelenleg a redundáns emberi kommunikációt utánozza, ahelyett, hogy a szoftverfejlesztésben elvárt precizitást és tömörséget célozná meg.

Munkaerőpiac. A munkaerőpiaci kilátásokat illetően a kutatás nem igazolta vissza a fejlesztők munkájának megszűnésével kapcsolatos félelmeket. A megkérdezettek nem féltik a munkájukat, sőt, egyetértés mutatkozik abban, hogy a jövőben is szükség lesz olyan szakemberekre, akik képesek a valós világ problémáit digitális megoldásokra fordítani. Ehhez ugyanis mindkét világ – az üzleti domain és a technológia – mélyreható ismerete szükséges. A következtetés szerint a szoftverfejlesztő munkakör átalakul, de nem szűnik meg: az MI nem kiváltja az embert, hanem a segítőjévé válik a mindennapi munkában.

Kilátások. Végezetül megállapítható, hogy a technológia megítélése széles skálán mozog attól függően, hogy az egyén mennyire mélyedt el a használatában. Arról azonban ettől függetlenül majdhogynem egyetértés uralkodik, hogy a technológiát övező várakozások gyakran inkább a fantázia, semmint a tapasztalatok szüleményei.

5.2. A generatív mesterséges intelligencia szerepe

Kutatásom alapján a generatív mesterséges intelligencia a szoftverfejlesztési életciklus (SDLC) számos pontján képes értéket teremteni, azonban a jelenlegi felhasználás eloszlása egyenletlen. A GMI felhasználásának szempontjait az 5.1. táblázat tartalmazza.

5.1. táblázat. A GMI felhasználásának szempontjai. (Saját szerkesztés.)

Szempont	Pozitív	Negatív
Felhasználási terület	Fejlesztésben, tesztelésben hatékony segítség	Többi SDLC fázisba még nincs megfelelően integrálva
Hibatűrés	Könnyen javítható dolgokra alkalmas	Kritikus feladatokra kockázatos
Felhasználási mód	Ellenőrzésre kiváló	Generálás eredményét nehéz utóellenőrizni
Kezdeti tapasztalatok	5% siker	95% kudarc

Felhasználási terület. A legszembetűnőbb következtetés a kódolási és tesztelési fázisok túlreprezentáltsága. Mind a szakirodalmi áttekintés, mind a panelbeszélgetések tapasztalatai azt mutatják, hogy a fejlesztők és az eszközgyártók fókusza elsősorban a kódgenerálásra irányul. Itt áll rendelkezésre a legtöbb tapasztalat, ezzel szemben az SDLC többi fázisában inkább csak kísérletezés zajlik. Ez egyben rámutat egy kihasználatlan potenciálra is: a valódi hatékonyságnöveléshez a generatív MI-t a teljes életciklusban alkalmazni kellene, nem csak a fejlesztési idő kisebb részét kitevő kódolásban (Valdes és mtsai., 2025).

A 4. fejezetben bemutatott felhasználási lehetőségeket a *Kísérletezés/Spekuláció*, *Kezdeti tapasztalatok* és *Aktív használat* kategóriákba soroltam, amit a 5.2. táblázat tartalmaz. A besorolás nem reprezentatív (hiszen elsősorban a panelbeszélgetéseken alapul, mivel a szakirodalomban bemutatott lehetőségekről gyakran nem állapítható meg, mennyire tartoznak hozzájuk valós ipari felhasználások is), de ennek ellenére is jó áttekintést ad az eredményeimről.

5.2. táblázat. A GMI használati lehetőségeinek besorolása SDLC fázisok és elterjedtség szerint. (Saját szerkesztés.)

SDLC fázis	**Kísérletezés/ Spekuláció**	**Kezdeti tapasztalatok**	**Aktív használat**
Projekt-tervezés	Helyzetfelmérés Brainstorming Értékelés Kockázatelemzés	Dokumentum-generálás Becslés	
Elemzés	Követelmény-származtatás Követelményosztályozás Követelményminőség Megvalósíthatóság		
Rendszer-tervezés	Tervezés	Technológiaválasztás Prototipizálás	
Fejlesztés			Kódgenerálás Megmagyarázás Refaktorálás Hibakeresés Szkriptek Konvertálás
Tesztelés	Kimenetellenőrzés Kompatibilitásellenőrzés	Integrációs tesztelés	Tesztgenerálás Tesztadat-generálás Kódátvizsgálás
Bevezetés	Infrastruktúrakód Függőségelemzés Végrehajtás	Dokumentáció	
Karbantartás	Hibajegy-előfeldolgozás Teljesítményelemzés	Hibaokfeltárás	Ügyféltámogatás

Hibatűrés. A GMI szerepének meghatározásakor kulcskérdés a feladat kritikussága és a hibatűrés mértéke. A technológia kiválóan alkalmazható olyan területeken, ahol a hiba megengedhető, mert „olcsó" vagy könnyen javítható (pl. prototípus készítése, egy UI elem stílusának generálása), de kockázatos a kritikus rendszerek (pl. közlekedés, energetika, pénzügy) esetében. A következtetés az, hogy a GMI szerepe a kockázatmentes kísérletezés és a gyors prototipizálás területén a legjelentősebb, míg az éles, biztonságkritikus rendszerekben a szerepe korlátozottabb kell, hogy maradjon.

Felhasználási mód. Kutatásom rávilágított egy fontos emberi aspektusra a tartalomlétrehozás kapcsán. Bár a GMI képes nagy mennyiségű kódot vagy dokumentációt generálni, az emberi természetből fakadó lustaság miatt fennáll a veszélye, hogy ezeket a kimeneteket a felelős személyek nem ellenőrzik megfelelő alapossággal. Ezzel szemben a technológia sokkal biztonságosabban használható az ellenkező irányban: ember által írt kódok vagy szövegek ellenőrzésére (review), kontextus magyarázatára vagy hibakeresésre. Ebben az esetben az MI „fáradhatatlan" figyelme kiegészíti az emberi kreativitást.

Kezdeti tapasztalatok. Végezetül fontos következtetés, hogy a GMI vállalati integrációja jelenleg gyerekcipőben jár. Ahogy arra az MIT friss kutatása is rámutatott, a projektek jelentős része, akár 95%-a kudarcot vall vagy nem hoz átütő sikert, mert a megoldások szigetszerűek maradnak (Challapally és mtsai., 2025). A technológia nem illeszkedik szervesen a meglévő vállalati folyamatokba, nem éri el a belső tudásbázisokat. Ez arra enged következtetni, hogy a GMI sikeres alkalmazása ma már nem elsősorban technológiai, hanem vezetési, szervezési és folyamatmenedzsment kihívás.

5.3. A generatív mesterséges intelligencia használatának kihívásai

A kutatási eredmények alapján a generatív MI szoftverfejlesztésbeli alkalmazását számos, nem tisztán technológiai kihívás nehezíti. Ezek a gátló tényezők gyakran erősebbnek bizonyulnak, mint maga a technológiai korlát. A GMI használatának főbb kihívásait az 5.3. táblázat tartalmazza.

Jogi és vállalati környezet. A nagyvállalati környezetben a legfőbb akadályt a jogi bizonytalanság és az adatbiztonsági szigor jelenti. A cégek érthető módon védik szellemi tulajdonukat és ügyféladataikat, ami szélsőséges esetben ahhoz vezet, hogy a biztonsági megoldások gyakorlatilag kiüresítik az MI eszközöket, feláldozva a potenciális hatékonyságnövelést. A fejlesztők így sokszor nem a legmodernebb eszközökhöz férnek hozzá, vagy ha igen, akkor nem tudják azokat a releváns adatokkal és rendszerekkel használni.

Tudásintegráció. A GMI hatékony működésének feltétele a kontextus ismerete. Egy szoftverfejlesztési projekt tudásanyaga azonban széttöredezett: a követelmények és dokumentumok, a kód és az adatok mind más rendszerben vagy adatbázisban találhatók. Ezen

5.3. táblázat. A GMI használatának főbb kihívásai. (Saját szerkesztés.)

Szint	Kihívás	Adottság	Kulcsszavak
Ország	Jogi környezet	Jogszabályok	Törvény Compliance
Szervezet	Vállalati környezet	Belső szabályok	Policy Compliance
Szervezet	Tudásintegráció	Szervezeti tudás	Heterogén Fragmentált
Ember	Emberi tényezők és kompetenciák	Személyes tudás	Prompt engineering Deskilling
Ember	Pszichológiai csapda	Emberi természet	Utóellenőrzés Unalom
Technológia	Műszaki korlátok és költségek	Technológiai fejlettség	Nemdeterminizmus Hallucináció Token

heterogén források biztonságos és naprakész integrálása a nyelvi modellek kontextusába komoly architekturális kihívás, ám enélkül a modell csak általános válaszokat tud adni, nem pedig specifikus segítséget.

Emberi tényezők és kompetenciák. A technológia használata új kompetenciákat igényel, mint például a *prompt engineering*. Ez nem csupán technikai tudás, hanem egy újfajta kommunikációs készség, amelyet tanulni kell. A szervezeteknek időt és teret kell biztosítaniuk ennek elsajátítására. Ugyanakkor felmerül az „elbutulás” (deskilling) kockázata is: ha a fejlesztők túlságosan hozzászoknak az MI segítségéhez, elveszíthetik rutinjukat az alapszintű problémamegoldásban, és előfordulhat, hogy MI támogatás nélkül már nem lesznek képesek hatékonyan dolgozni (Banh és mtsai., 2025).

Pszichológiai csapda. A szoftverfejlesztésben az MI ígérete az, hogy átveszi az unalmas, repetitív munkát. Ezzel azonban egy új, még unalmasabb feladatot teremt: a generált kimenetek átnézését és validálását. Az emberi természetből fakadóan a monoton ellenőrzési feladatok során lankad a figyelem. Ez a helyzet veszélyes hibákhoz vezethet: a fejlesztők hajlamosak átsiklani olyan MI által generált hibák felett, amelyeket ők maguk talán el sem követtek volna, vagy saját kódolás közben azonnal észrevettek volna.

Műszaki korlátok és költségek. A modellek használata nem ingyenes; a tokenek (számítási egységek) fogyása közvetlen költséget jelent, melyet menedzselni kell. Emellett technikai kihívást jelent a reprodukálhatóság hiánya: a nemdeterminisztikus modellek ugyanarra a bemenetre eltérő kimenetet adhatnak, ami megnehezíti a tesztelést és a validálást (Sallou és mtsai., 2024). A hallucináció kiküszöbölésére megoldást jelenthetne az LLM válaszok automatizált validálása, ehhez azonban szükség lenne a validálandó tudás

formalizálására, hiszen jelenleg a tudás természetes nyelven áll rendelkezésre, nem pedig matematikai formalizmusok által leírva.

5.4. Akciólista a generatív mesterséges intelligencia bevezetéséhez

A kutatási eredmények és a feltárt kihívások alapján az alábbi lépésekből álló akciótervet javaslom a technológia sikeres szervezeti bevezetéséhez:

1. **Jogi és szabályozási környezet tisztázása.** Az első és legfontosabb lépés egy világos, mindenki számára elérhető szabályrendszer megalkotása. A fejlesztőknek tudniuk kell, mely eszközöket használhatják, milyen adatokat oszthatnak meg ezekkel, és hol húzódnak a tilalmak határai. A bizonytalanság megszüntetése kulcsfontosságú a technológia használatának széleskörű elterjedéséhez.

2. **Mérhető keretrendszer kialakítása.** A bevezetés sikerességének nyomon követéséhez mérőszámokra van szükség. Definiálni kell, mit tekintünk sikernek (pl. fejlesztési ciklusidő csökkenése, hibák számának csökkenése), és mérni kell a ráfordításokat is (pl. licenszköltségek, tokenhasználat). A megtérülés (return of investment, ROI) nyomon követése segíti a tisztánlátást, és alapot ad a technológia skálázását illető döntések előkészítéséhez.

3. **Vállalati tudásbázis integrálása.** A valódi hatékonyságnöveléshez meg kell teremteni a technikai feltételeit annak, hogy az MI modellek (biztonságos keretek között) hozzáférjenek a vállalat belső tudásanyagához (dokumentumok, forráskódok, adatbázisok). Enélkül az eszközök csak általános tudásúak lesznek, de nem válnak specifikus szakértőkké.

4. **Kultúra és tudásmegosztás.** A technológia bevezetése nem ér véget a licenszek megvásárlásával. Támogatni kell a belső tudásmegosztást (pl. rendszeres fórumok, legjobb gyakorlatok (best practice) megosztása), és dedikált időt kell biztosítani a kísérletezésre. A fejlesztők érdeklődésének felkeltése és fenntartása kulcsfontosságú a sikeres adoptációhoz (Ahlawat és mtsai., 2024).

5.5. Kutatásmódszertan értékelése

Kutatásom lezárásaként érdemes értékelni az alkalmazott módszertant is, amely újrafelhasználható a generatív mesterséges intelligenciával kapcsolatos szervezeti vélemények és szoftverfejlesztésbeli tapasztalatok strukturált vizsgálatára. Az SDLC fázisaira épülő struktúra megfelelőnek bizonyult: segített a szerteágazó téma rendszerezésében, és mankót adott a panelbeszélgetések résztvevőinek is a gondolkodáshoz.

A panelbeszélgetések résztvevőinek száma megfelelő volt, eltérő tapasztalati szintjük hasznosnak bizonyult, de szerencsés lett volna, ha nemcsak fejlesztőket sikerült volna megszólaltatni, hanem a szoftverfejlesztéshez kapcsolódó más munkakörök képviselőit is.

A résztvevők közötti interakciók olyan szempontokat is felszínre hoztak, amelyek egyéni interjúk során rejtve maradtak volna. Tapasztalatom szerint a fejlesztők szívesen beszélnek a témáról, az eredetileg tervezett 60 perces időkeret szűkösnek is bizonyult, a 90 perc azonban már elegendő volt a mélyebb szakmai viták kibontakozásához is.

Az ötletek SDLC fázisok szerinti rendszerezését követően eredetileg terveztem egy olyan kört is, amikor az ötleteket aszerint értékeljük, hogy a résztvevők szerint azokat milyen mértékben lehet az MI-re bízni, amihez a *human only*, *AI in the loop*, *human in the loop*, *AI only* osztályozást terveztem használni. Az első panelbeszélgetésen kipróbáltam ezt, azonban nem váltotta be a hozzá fűzött reményeket: az derült ki, hogy az eredeti felhasználási ötleteket szinte minden esetben tovább lehet bontani még kisebb részekre, ami megnehezíti az eredeti ötletek osztályozását (hiszen a különböző alrészek eltérő osztályokba esnének). Ezért ezt a részt a második panelbeszélgetésen már el is hagytam, és a dolgozatban sem szerepeltettem.

A kutatás legfőbb korlátja a reprezentativitás hiánya. Bár a feltárt trendek és dilemmák *vélhetően* iparági szinten is érvényesek, a következtetések mégsem általánosíthatók, ehhez további, nagyobb mintán elvégzett kutatásokra lenne szükség. A kidolgozott módszertan – a strukturált SDLC alapú panelbeszélgetés – ugyanakkor egy könnyen adaptálható, újrafelhasználható eszköz, amellyel bármely szoftverfejlesztő szervezet hatékonyan felmérheti saját érettségét és viszonyulását a generatív mesterséges intelligenciához.

6. fejezet

Összegzés

Szakdolgozatomban azt vizsgáltam, hogyan hat a generatív mesterséges intelligencia a szoftverfejlesztésre. Ehhez elsőként áttekintettem a releváns elméleti ismereteket. Bemutattam a szoftverfejlesztési életciklust, valamint a fejlesztési folyamat legismertebb modelljeit és módszertanait. Áttekintettem a mesterséges intelligencia fejlődésének főbb állomásait, majd a mesterséges intelligencia szoftverfejlesztésre gyakorolt hatásait. Példákkal demonstráltam, hogy már a nemgeneratív MI is komoly hatást gyakorolt a fejlesztési folyamatra, valamint felvillantottam a generatív MI-vel kapcsolatos legfontosabb kihívásokat. Végül röviden áttekintettem a szoftveripar sajátosságait.

Kidolgoztam és bemutattam kutatásom módszertanát. A kutatás hátteréből és motivációjából kiindulva kutatási célokat határoztam meg, majd azokból megfogalmaztam a kutatási kérdéseket. Kutatásom során számos különböző adatgyűjtési módot használtam, amelyeket részletesen bemutattam.

Kutatásom eredményei bemutatták, hogyan ítélik meg a szoftverfejlesztők a generatív mesterséges intelligenciát, mint új technológiát. A szoftverfejlesztési életciklus fázisai mentén haladva részletesen bemutattam, hogy milyen felhasználási potenciál rejlik a technológiában és milyen vonatkozó valós tapasztalatok állnak rendelkezésre. Sorra vettem a felhasználás nehézségeit, elméleti és gyakorlati korlátait.

A kutatás eredményeiből következtetéseket vezettem le. Összegeztem a technológia megítélését, valamint körvonalaztam a szoftverfejlesztésen belüli szerepét. Azonosítottam a felhasználás legfőbb kihívásait, és megfogalmaztam egy rövid akciólistát, amely segítségül szolgálhat a technológia bevezetése előtt álló szervezetek számára. Végül értékeltem az alkalmazott kutatásmódszertant, és kijelöltem a kutatás lehetséges folytatását.

Köszönetnyilvánítás

Köszönöm konzulensemnek, *Drótos Györgynek*, hogy az első féléves *Szervezeti információrendszerek* tárgy oktatása során olyan szemléletet adott át, amely azóta is elkísér. Köszönöm továbbá, hogy vállalta a konzulensi szerepet, és észrevételeivel segítette a szakdolgozat elkészültét.

Köszönöm a megkérdezett egyetemi szakértőnek és a panelbeszélgetések résztvevőinek, hogy idejüket nem kímélve megosztották velem gondolataikat és tapasztalataikat.

Köszönöm mindenkinek, aki támogatott a képzés elvégzésében. Az elmúlt másfél év is úgy volt, mint mindig e világi élet: *egyszer fázott, másszor lánggal égett.* Hálás vagyok a barátaimért, akik mellettem voltak, amikor a legnagyobb szükségem volt rájuk: *Körte, Zsuzsi, Miki* és persze *Ági, Anna, Dóri, Hanga, Szander, Viktor, Virág, Zsófi* – köszönöm!

> – *Vége? – kérdezte Róbert Gida.*
> – *Ennek. De van más mesém.*
>
> A. A. Milne: *Micimackó*

Irodalomjegyzék

Acharya, D. B., Kuppan, K., & Divya, B. (2025). Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey. *IEEE Access*, *13*, 18912–18936. https://doi.org/10.1109/ACCESS.2025.3532853

Ahlawat, P., Bedard, J., Damani, S., Feldman, B., Lucero, E., Samdaria, A., & Sipperly, G. (2024). *The Art of Scaling GenAI in Software* [Letöltve: 2025-11-23]. Boston Consulting Group. https://www.bcg.com/publications/2024/the-art-of-scaling-genai-in-software

Akinnagbe, O. (2024). Human-AI Collaboration: Enhancing Productivity and Decision-Making. *International Journal of Education, Management, and Technology*, *2*, 387–417. https://doi.org/10.58578/ijemt.v2i3.4209

Alazzawi, A., Yas, Q., & Rahmatullah, B. (2023). A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions. *Iraqi Journal for Computer Science and Mathematics*, *4*, 173–190. https://doi.org/10.52866/ijcsm.2023.04.04.014

Ali, A., & Gravino, C. (2019). A systematic literature review of software effort prediction using machine learning methods. *J. Softw. Evol. Process*, *31*(10). https://doi.org/10.1002/smr.2211

Amalfitano, D., Faralli, S., Hauck, J. C. R., Matalonga, S., & Distante, D. (2023). Artificial Intelligence Applied to Software Testing: A Tertiary Study. *ACM Comput. Surv.*, *56*(3). https://doi.org/10.1145/3616372

Anderson, D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

Banh, L., Holldack, F., & Strobel, G. (2025). Copiloting the future: How generative AI transforms Software Engineering. *Information and Software Technology*, *183*(100). https://doi.org/10.1016/j.infsof.2025.107751

Banh, L., & Strobel, G. (2023). Generative Artificial Intelligence. *Electronic Markets*, *33*, 1–17. https://doi.org/10.1007/s12525-023-00680-1

Barroca, E. (2024). *LLMs don't hallucinate, they make mistakes* [Letöltve: 2025-11-22]. https://vertesiahq.com/blog/llm-hallucinations-vs-mistakes

Beck, K. (2004). *Extreme Programming Explained: Embrace Change* (2. kiad.). Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin,

R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for Agile Software Development [Letöltve: 2025-11-02]. https://agilemanifesto.org/

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement. *Computer*, *21*(5), 61–72. https://doi.org/10.1109/2.59

Briggs, R. O., & Nunamaker Jr., J. F. (2020). Special Section: The Growing Complexity of Enterprise Software. *Journal of Management Information Systems*, *37*(2), 313–315. https://doi.org/10.1080/07421222.2020.1759339

Brikman, Y. (2022). *Terraform: Up & Running: Writing Infrastructure as Code* (3. kiad.). O'Reilly Media.

Cao, Y., Li, S., Liu, Y., Yan, Z., Dai, Y., Yu, P. S., & Sun, L. (2023). A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT. *arXiv preprint*. https://arxiv.org/abs/2303.04226

Challapally, A., Pease, C., Raskar, R., & Chari, P. (2025). *State of AI in Business 2025: The GenAI Divide*. MIT NANDA.

Chong Ming, L. (2025). *Replit's CEO apologizes after its AI agent wiped a company's code base in a test run and lied about it* [Letöltve: 2025-11-26]. Business Insider. https://www.businessinsider.com/replit-ceo-apologizes-ai-coding-tool-delete-company-database-2025-7

Collins, C., Dennehy, D., Conboy, K., & Mikalef, P. (2021). Artificial Intelligence in Information Systems Research: A Systematic Literature Review and Research Agenda. *International Journal of Information Management*, *60*, 102383. https://doi.org/10.1016/j.ijinfomgt.2021.102383

Cunha, W., Angulo, G., & Camargo, V. (2020). InSet: A Tool to Identify Architecture Smells Using Machine Learning, 760–765. https://doi.org/10.1145/3422392.3422507

De Siano, G. D., Fasolino, A. R., Sperlí, G., & Vignali, A. (2025). Translating code with Large Language Models and human-in-the-loop feedback. *Information and Software Technology*, *186*, 107785. https://doi.org/https://doi.org/10.1016/j.infsof.2025.107785

Euklidész. (é.n.). *Elemek* [Eredeti kiadás: i.e. 300 körül].

Feigenbaum, E. A. (1977). The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering. *International Joint Conference on Artificial Intelligence*.

Finley, K. (2024). *How developers spend the time they save thanks to AI coding tools* [Letöltve: 2025-11-23]. GitHub. https://github.blog/ai-and-ml/generative-ai/how-developers-spend-the-time-they-save-thanks-to-ai-coding-tools/

Floridi, L., & Cowls, J. (2019). A Unified Framework of Five Principles for AI in Society. *Harvard Data Science Review*, *1*(1). https://hdsr.mitpress.mit.edu/pub/l0jsh9d1

Fowler, M. (2006). *Continuous Integration* [Letöltve: 2025-11-02]. https://martinfowler.com/articles/continuousIntegration.html

Gartner. (2025). *Gartner Experts Answer the Top Generative AI Questions for Your Enterprise* [Letöltve: 2025-11-23]. Gartner. https://www.gartner.com/en/topics/generative-ai

Gerlich, M. (2025). AI Tools in Society: Impacts on Cognitive Offloading and the Future of Critical Thinking. *Societies*, *15*(1). https://doi.org/10.3390/soc15010006

Gnanasambandam, C., Harrysson, M., Singh, R., & Chawla, A. (2025). *How an AI-enabled software product development life cycle will fuel innovation* [Letöltve: 2025-11-22]. McKinsey & Company. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/how-an-ai-enabled-software-product-development-life-cycle-will-fuel-innovation

Grand View Research. (2025). Software Market (2025 – 2030) Size, Share & Trends Analysis Report. https://www.grandviewresearch.com/industry-analysis/software-market-report/toc

Hadi, M. U., Al-Tashi, Q., Qureshi, R., Shah, A., Muneer, A., Irfan, M., Zafar, A., Shaikh, M., Akhtar, N., Wu, J., & Mirjalili, S. (2023). *Large Language Models: A Comprehensive Survey of its Applications, Challenges, Limitations, and Future Prospects* (tech. rep.). TechRxiv. https://doi.org/10.36227/techrxiv.23589741.v6

Highsmith, J. (2002). *Agile Software Development Ecosystems*. Addison-Wesley.

Hightower, K., Burns, B., & Beda, J. (2019). *Kubernetes: Up and Running: Dive into the Future of Infrastructure* (2. kiad.). O'Reilly Media.

HOLD Alapkezelő. (2025). Hold After Hours: Megfejtjük az AI hypeot [Letöltve: 2025-11-23]. https://www.youtube.com/watch?v=dwytNPD5fAo

Hossain, M. (2023). Software Development Life Cycle (SDLC) Methodologies for Information Systems Project Management. *International Journal For Multidisciplinary Research*. https://doi.org/10.36948/ijfmr.2023.v05i05.6223

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

IBM. (é.n.-a). *AI in Software Development* [Letöltve: 2025-11-05]. IBM. https://www.ibm.com/think/topics/ai-in-software-development

IBM. (é.n.-b). *What is the software development lifecycle (SDLC)?* [Letöltve: 2025-11-02]. IBM. https://www.ibm.com/think/topics/sdlc

Jeyarajan, B., Murugan, A., Pandy, G., & Pugazhenthi, V. J. (2025). AI for Predictive Monitoring and Anomaly Detection in DevOps Environments. *SoutheastCon 2025*, 450–455. https://doi.org/10.1109/SoutheastCon56624.2025.10971552

Kalai, A. T., Nachum, O., Vempala, S. S., & Zhang, E. (2025). Why Language Models Hallucinate. https://arxiv.org/abs/2509.04664

Karaci Deniz, B., Gnanasambandam, C., Harrysson, M., Hussin, A., & Srivastava, S. (2023). *Unleashing developer productivity with generative AI* [Letöltve: 2025-11-23]. McKinsey. https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/unleashing-developer-productivity-with-generative-ai

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.

Lavington, S., & Society, B. C. (1998). *A History of Manchester Computers*. British Computer Society.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Liang, J. T., Yang, C., & Myers, B. A. (2024). A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. https://doi.org/10.1145/3597503.3608128

Mann, K., & O'Connor, F. (2023). *How to Use Generative AI to Boost Developer Productivity* [Letöltve: 2025-11-23]. Gartner. https://www.gartner.com/en/software-engineering/insights/how-to-use-gen-ai-to-boost-developer-productivity

Maslow, A. H. (1966). *The Psychology of Science: A Reconnaissance*. Harper & Row.

Matvitskyy, O., Iijima, K., West, M., Davis, K., Jain, A., & Vincent, P. (2023). *Magic Quadrant for Enterprise Low-Code Application Platforms*. Gartner Research. https://www.gartner.com/en/documents/4843031

McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (1955). A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence. *Dartmouth AI Conference*.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., & Galstyan, A. (2021). A Survey on Bias and Fairness in Machine Learning. *ACM Computing Surveys*, *54*(6), 1–35. https://doi.org/10.1145/3457607

Meyer, A. N., Barr, E. T., Bird, C., & Zimmermann, T. (2021). Today Was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering*, *47*(5), 863–880. https://doi.org/10.1109/TSE.2019.2904957

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Modi, M. D. B. (2024). Transforming Software Development Through Generative AI: A Systematic Analysis of Automated Development Practices. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, *10*(6), 536–547. https://doi.org/10.32628/CSEIT24106197

Monaghan, B. D., & Bass, J. M. (2020). Redefining Legacy: A Technical Debt Perspective. In M. Morisio, M. Torchiano, & A. Jedlitschka (Szerk.), *Product-Focused Software Process Improvement* (o. 254–269). Springer International Publishing.

Muratović, F., Gill, J., Kearns-Manolatos, D., & Alibage, A. (2024). *How can organizations engineer quality software in the age of generative AI?* [Letöltve: 2025-11-23]. Deloitte Center for Integrated Research. https://www.deloitte.com/us/en/insights/industry/technology/how-can-organizations-develop-quality-software-in-age-of-gen-ai.html

Nabeel, M. (2024). AI-Enhanced Project Management Systems for Optimizing Resource Allocation and Risk Mitigation: Leveraging Big Data Analysis to Predict Project Outcomes and Improve Decision-Making Processes in Complex Projects. *Asian Journal of Multidisciplinary Research & Review*, *5*, 53–91. https://doi.org/10.55662/AJMRR.2024.5502

Newell, A., & Simon, H. A. (1976). Computer Science as Empirical Inquiry: Symbols and Search. *Communications of the ACM*, *19*(3), 113–126. https://doi.org/10.1145/360018.360022

Nguyen Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., Rafiq, U., Melegati, J., Kemell, K.-K., Saari, M., Zhang, Z., Le, H., Quan, T., & Abrahamsson, P. (2023). *Generative Artificial Intelligence for Software Engineering - A Research Agenda*. https://doi.org/10.2139/ssrn.4622517

Panetta, K. (2023). *Set Up Now for AI to Augment Software Development* [Letöltve: 2025-11-23]. Gartner. https://www.gartner.com/en/articles/set-up-now-for-ai-to-augment-software-development

Petry, S., & Carlson, A. (2024). *10 ways GenAI improves software development* [Letöltve: 2025-11-23]. PwC. https://www.pwc.com/us/en/tech-effect/ai-analytics/generative-ai-for-software-development.html

Plant, R., Giuffrida, V., & Gkatzia, D. (2022). You Are What You Write: Preserving Privacy in the Era of Large Language Models. https://arxiv.org/abs/2204.09391

Potvin, R., & Levenberg, J. (2016). Why Google stores billions of lines of code in a single repository. *Commun. ACM*, *59*(7), 78–87. https://doi.org/10.1145/2854146

Raji, I., Smart, A., White, R., Mitchell, M., Gebru, T., Hutchinson, B., Smith-Loud, J., Theron, D., & Barnes, P. (2020). Closing the AI accountability gap: defining an end-to-end framework for internal algorithmic auditing, 33–44. https://doi.org/10.1145/3351095.3372873

Richardson, C., & Rymer, J. R. (2014). *New Development Platforms Emerge for Customer-Facing Applications* (Forrester Report). Forrester Research.

Rico, S., & Öberg, L.-M. (2025). Challenges and Opportunities for Generative AI in Software Engineering: A Managerial View. *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 1338–1344. https://doi.org/10.1145/3696630.3728718

Rook, P. (1986). Controlling Software Projects. *IEEE Software*, *3*(5), 7–16. https://doi.org/10.1109/MS.1986.231364

Royce, W. W. (1970). Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*, 1–9.

Russell, S., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4. kiad.). Pearson Education.

Sallou, J., Durieux, T., & Panichella, A. (2024). Breaking the Silence: the Threats of Using LLMs in Software Engineering. *2024 IEEE/ACM 46th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 102–106. https://doi.org/10.1145/3639476.3639764

Sawant, P. D. (2024). Test Case Prioritization for Regression Testing Using Machine Learning. *2024 IEEE International Conference on Artificial Intelligence Testing (AI-Test)*, 152–153. https://doi.org/10.1109/AITest62860.2024.00027

Schwaber, K., & Sutherland, J. (1997). *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. https://scrumguides.org/

Singla, A., Sukharevsky, A., Yee, L., Chui, M., Hall, B., & Balakrishnan, T. (2025). *The state of AI in 2025: Agents, innovation, and transformation* [Letöltve: 2025-11-23]. QuantumBlack, AI by McKinsey. https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai

Software. (2021). *Global Code Time Report: Understanding engineering friction, productivity, and flow* [Letöltve: 2025-11-23]. https://www.software.com/reports/code-time-report

Sommerville, I. (2015). *Software Engineering* (10th kiad.). Pearson Education.

Statista. (2023). Number of software developers worldwide in 2018 to 2024 [Letöltve: 2025-12-03]. https://www.statista.com/statistics/627312/worldwide-developer-population/

Szeszlér, D. (2014). *Bevezetés a számításelméletbe 1.* https://cs.bme.hu/bsz1/jegyzet/bsz1_jegyzet.pdf

Uspenskyi, S. (2025). How Many Software Engineers Are There in 2025? [Letöltve: 2025-12-03]. https://springsapps.com/knowledge/how-many-software-engineers-are-there-in-2025

Valdes, R., Walsh, P., Khandabattu, H., & Bhat, M. (2025). *Emerging Tech: AI Developer Tools Must Span SDLC Phases to Deliver Value* (tech. rep.). Gartner. https://www.gartner.com/en/documents/report/emerging-tech-ai-developer-tools-must-span-sdlc-phases-to-deliver-value

Walsh, P., Khandabattu, H., Brasier, M., Holloway, K., & Batchu, A. (2025). *Magic Quadrant for AI Code Assistants* (tech. rep.). Gartner. https://www.gartner.com/doc/reprints?id=1-2LVTG7RP%5C&ct=250915

Weizenbaum, J. (1966). ELIZA—A Computer Program for the Study of Natural Language Communication between Man and Machine. *Communications of the ACM*, *9*(1), 36–45. https://doi.org/10.1145/365153.365168

Winograd, T. (1972). *Understanding Natural Language* (Doctoral dissertation). Academic Press, Inc.

Yang, W., Some, L., Bain, M., & Kang, B. (2025). A comprehensive survey on integrating large language models with knowledge-based methods. *Knowledge-Based Systems*, *318*, 113503. https://doi.org/https://doi.org/10.1016/j.knosys.2025.113503

Zeichick, A. (2024). *7 Ways GenAI Can Help Improve Software Development* [Letöltve: 2025-11-23]. Oracle. https://www.oracle.com/asean/artificial-intelligence/generative-ai/generative-ai-software-development/

Zhang, J., Zhang, Z., & Ma, F. (2014). *Automatic generation of combinatorial test data*. Springer.

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., & Chen, Z. (2024). A Survey on Large Language Models for Software Engineering. https://arxiv.org/abs/2312.15223

Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E.-V., & Batista-Navarro, R. T. (2021). Natural Language Processing for Requirements

Engineering: A Systematic Mapping Study. *ACM Comput. Surv.*, *54*(3). https://doi.org/10.1145/3444689

Zhao, Y., Damevski, K., & Chen, H. (2023). A Systematic Survey of Just-in-Time Software Defect Prediction. *ACM Comput. Surv.*, *55*(10). https://doi.org/10.1145/3567550

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2024). Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM*, *67*(3), 54–63. https://doi.org/10.1145/3633453

Zurcher, F., & Randell, B. (1968). Iterative Multi-Level Modeling - A Methodology for Computer System Design, 867–871.