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

Tartalomjegyzék

Ábrajegyzék

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 – amit 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, amelyek 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, ami 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, ami éppen az az intellektuális tevékenység, ami 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. Végül a 2.3. alfejezetben ismertetem, hogyan hat a mesterséges intelligencia a szoftverfejlesztésre.

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, amit 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, amit 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 iránytű, 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, ami alapot ad a további fejlesztési döntésekhez. Eredménye 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 (implementálása) 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, ami 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.

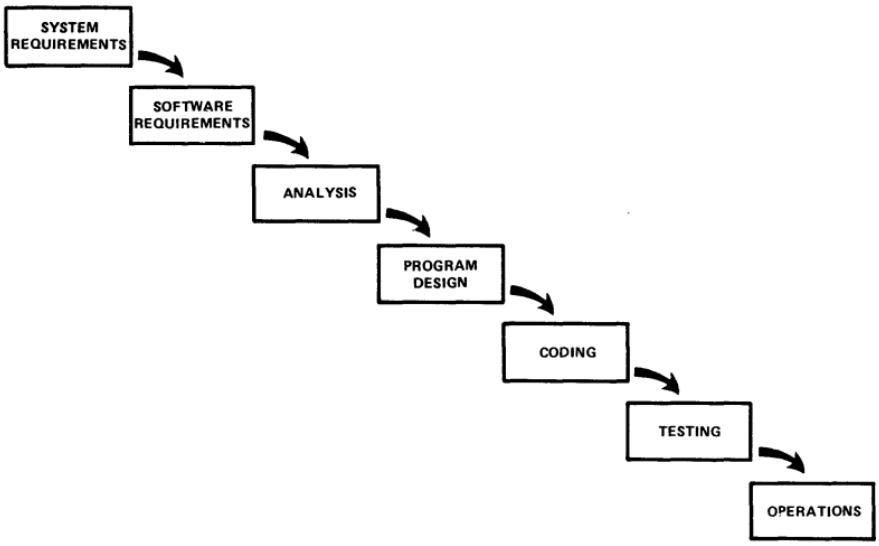
A fázisok egymásra épülnek, a különböző fejlesztési modellek azonban már eltérően értelmezhetik a fázisok közti átmeneteket. Míg a legegyszerűbb megközelítés szerint a fázisok lineárisan követik egymást, más modellek szerint iteratívan is végrehajthatók. Látható tehát, hogy az SDLC csupán testre szabható 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), amiben 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, amiben célokat határoznak meg, értékelik a rizikófaktorokat, prototípusokat



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

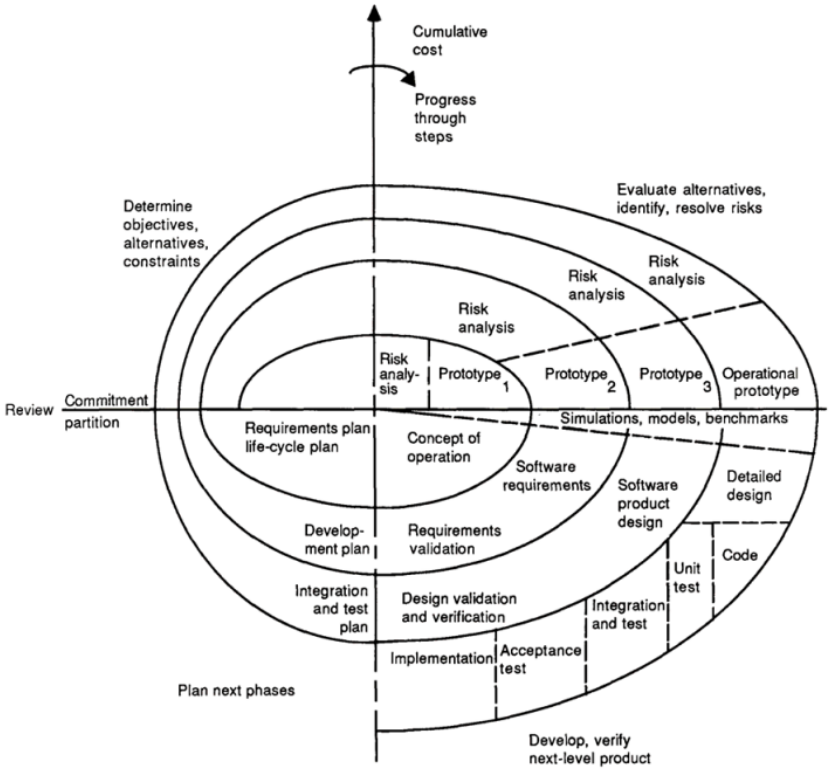
fejlesztenek, majd megtervezik a következő ciklust. Ez a modell különösen olyan összetett projektekben alkalmazható, ahol a követelmények gyakran változnak, vagy magas a kockázat.

A vízesésmodell továbbfejlesztése a *V-modell* (2.1. 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), amiket 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.

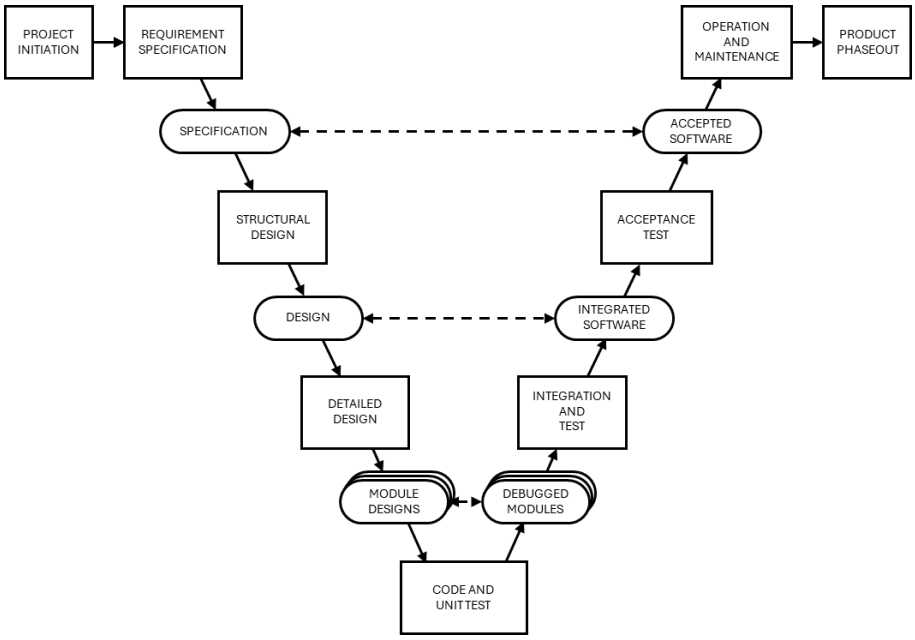
2.1.3. Agilis módszertanok

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, 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. A négy alapérték a következő:



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



2.3. ábra. A V-modell lépései. (Boehm (1988) ábrája feljavított minőségben, saját szerkesztés.)

- 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 kódmódosítások 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 egyik legfontosabb irányzata a *low-code* és *no-code* paradigma, amelynek 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 programozói 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. 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],

[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/
[8] https://www.outsystems.com/

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 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,

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

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, amelyet 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 (Good Old-Fashioned AI). 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 az adatok mögötti összefüggéseket. A gépi tanulás három fő kategóriája: a felügyelt (supervised), a nem felügyelt (unsupervised) és a megerősítéses (reinforcement) tanulás. A

[13] https://chatgpt.com/

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), ami 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, teszte-

lésen 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 regressziós tesztprioritás célja, hogy a teszteket olyan sorrendben futtassa, amely minimalizálja az időráfordítást és maximalizálja a hibák korai detektálását. Erre azért van szükség, mert a tesztelésre rendelkezésre álló idő mindig 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).

[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/

2.1. táblázat. Nemgeneratív MI felhasználások és eszközök az SDLC fázisaiban

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]
	Regressziósteszt-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" (jóllehet nem támogató, csak autonóm). Gyorsan képes ötleteket és megoldási javaslatokat adni, kódvázlatokat előállítani, meglévő kódokat elemezni, ugyanakkor nem tévedhetetlen, ezért emberi kontrollt igényel (De Siano és mtsai., 2025).

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. 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 a generatív MI modellek kimeneteit úgy kell integrálni a vállalati folyamatokba, hogy ezek a megoldások ne szigetszerű eszközökként jelenjenek meg a szervezeten belül, hanem a vállalati működés szerves, beágyazott elemeivé váljanak.

[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).

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. Az iparági elemzések évek óta folyamatos, jelentős növekedést jeleznek a vállalati szoftverpiacon (Grand View Research, 2025). Ezen 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.

A nagyvállalati szoftverfejlesztés lényegesen eltér a kisebb projektek vagy startupok világától. Ezek a 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, amit 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

Az elmúlt évek egyik legforróbb témája a generatív mesterséges intelligencia. Életünk szinte minden részére hatással van, sőt, talán azt sem túlzás mondani, hogy felforgatja azt. A tudományos kutatástól kezdve az orvosláson keresztül az oktatáson át a művészetig sorra nyíltak meg olyan lehetőségek, amelyek korábban elképzelhetetlenek voltak, és ezzel számos korábbi alapvetés is megkérdőjeleződött azt illetően, hogy miben rejlik az emberi tevékenységek emberi mivolta.

Ez a hullám természetesen a szoftverfejlesztést is elérte, sőt, mivel a technológia maga is ide köthető, talán érthető, hogy ezen a területen különösen meghatározó témává vált. A szoftverfejlesztés hatékonyabbá tételének lehetősége joggal hozta lázba a tágan értelmezett szakmát, ami a generatív mesterséges intelligencia eszközök gyors elterjedéséhez vezetett, gyakran alapvetően megváltoztatva ezzel a korábban megszokott munkafolyamatokat.

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 egy sokaság (pl. egy szervezet) 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. 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 amerikai 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. A panelbeszélgetések adatait a 3.1. táblázat tartalmazza.

3.1. táblázat. Iparági panelbeszélgetések adatai

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 ötletelek 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. 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

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 intelligencia 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

...

4.1. A generatív mesterséges intelligencia megítélése

RQ1.

lustaságból használni jó, tudatlanságból használni veszélyes

junioroknak, betanulóknak egyre gyakrabban tiltják, mert ha az AI mondja meg a megoldást, nem rögzül igazán. a szenvedés (gondolkodás, megoldás megtalálása) rögzíti a tudást (a matematikához nem vezet királyi út) – enélkül hasonló problémáknál nem fogjuk tudni használni ugyanezt a tudást[1]

kiváltja a juniorokat –> de akkor honnan lesznek később seniorok?

4.2. A generatív mesterséges intelligencia felhasználása

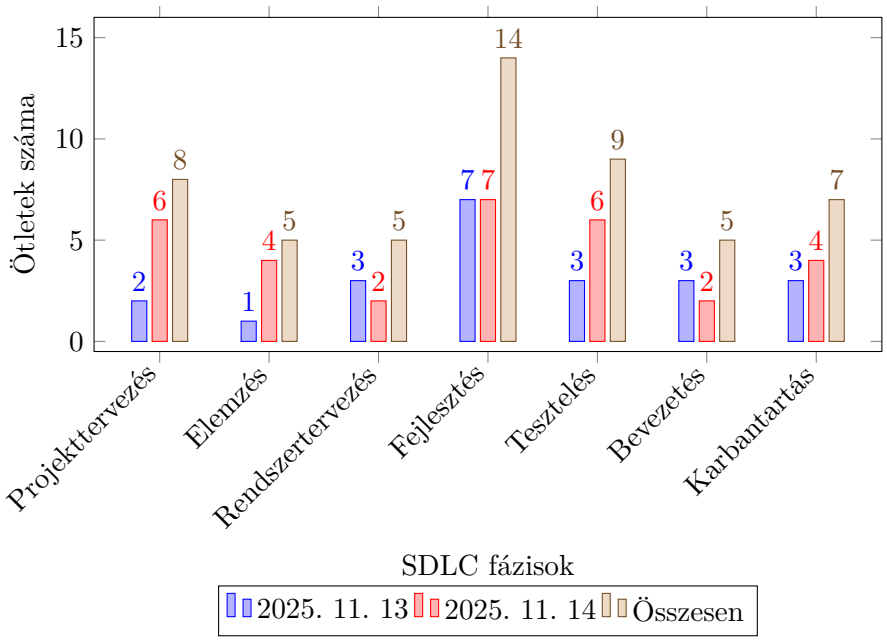
RQ2 és RQ3 egyben, SDLC fázisonként

brainstorming postitek eloszlása

számos felhasználás már nemgeneratív MI-vel is volt, de generatívval kényelmesebb (még ha nem is feltétlenül jobb) a sok pontosabb céleszköz vs. egy kevésbé pontos általános eszköz dilemma a pontosság és a hatékonyság között, lásd (Gnanasambandam és mtsai., 2025)

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 bizonyos technikai részekkel eltölteniük, több idejük marad magasabb szintű tervezésre, munkatársakkal együtt gondolkodásra, új technológiák elsajátítására Finley, 2024.

[1] Az alapszakos egyetemi tanulmányaim legelején elsőként kezembe került jegyzet (Szeszlér, 2014) előszava idézte az anekdotát, ami 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."



4.1. ábra. Az ötletek eloszlása SDLC fázisok szerint

A technológia felhasználásának fókuszában a kódolás áll (Walsh és mtsai., 2025), amit az ötletek eloszlása is alátámaszt (4.1. ábra). Fontos azonban látni, hogy a szoftverfejlesztők 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).

4.2.1. Projekttervezés

széles, de nem mély research, csak pointerek keresése, hogy merre érdemes indulni

ötletgyűjtés: mit lenne érdemes csinálni

aktuális helyzet/működés felmérése

adatok elemzése, pl. fontos-e egyáltalán a feature? (használják-e a rendelkezésre álló adatok alapján)

kockázatok elemzése

ötlet challenge-elése, pro-kontra

erőforrás, ETA becslés

tervdokumentum generálás

4.2.2. Elemzés

megvalósíthatóság elemzése

külső dokumentumok elemzése (azokból származó követelmények): adatok alapján követelmények meghatározása

követelmény megfogalmazás ellenőrzése, hiányzó követelmények keresése

4.2.3. Rendszertervezés

eszköz/technológia választás: lehetőségek keresése, megoldási lehetőségek összehasonlítása

prototípusok elkészítése: POC/MVP, infrastruktúra, egyszerű UI generálása

komplett rendszerterv legenerálása nagyon veszélyes: ki fogja azt tényleg olyan részletesen átnézni, mint amennyire ahhoz át kellett volna gondolni, hogy ő maga megcsinálja? túl nagy mennyiségű információ

4.2.4. Fejlesztés

kódgenerálás: nem kritikus, kevés mentális energiát igénylő dolgok (pl. boilerplate, UI) komplex dolgokra hibás eredményt ad, a sok javítási iteráció fárasztóbb, mint megírni magát a kódot, végső soron nem spórolható meg a probléma végiggondolása (Liang és mtsai., 2024)

refaktor:

syntax error fix (pl. hosszú sql query), auto complete

kód megértése, elmagyarázása, dokumentálása (mit csinál)

scriptek (egyszerű, egyértelmű feladatok, olyan nyelven, amivel ritkán dolgozunk, tehát nem ismerjük, pl. awk, bash)

reguláris kifejezés (regular expression, regex) írás/értelmezés

fájlformátumok közti konvertálás

AI generált kód: fordítás+CI/CD sok hibát megtalálhat, de script nyelveknél még rizikósabb, hogy valami csak futtataáskor élesben derül ki

4.2.5. Tesztelés

unit test generálás – gond: white box test, ha a kódból generálja, nem igazi független teszt

backward compatibility check

regressziós teszt output elemzés

integration test: AI agent elindítja a komponenseket, teszteli a kommunikációt

tesztadat generálás – korlát: nem szisztematikus

AI generált kód reviewja: ember nem vonódik be, ha AI írta a kódot, nehezebb és unalmasabb átnézni, főleg ha sok kódot generál

vannak hibák, amiket nem követnénk el, de ha elénk tesznek egy kódot, amiben ott van, nem vesszük észre (nehezebb észrevenni, mint nem elrontani)

de az ember által írt kód AI általi reviewja, ellenőrzése jó felhasználás

4.2.6. Bevezetés

review

dependency analysis

deploy config generálás

utódokumentálás: ha van deploy dokumentálás, technical/user guide generálás

folyamat definiáltsága a kulcs: ha nem egy jól meghatározott folyamat, az AI se fogja tudni ha jól megahtározott folyamat, akkor viszont már eleve automatizálható, nem kell igazán AI

agentic AI elvégezheti a telepítést, de ahhoz hozzáférés kell éles rendszerhez, ami nagyon veszélyes

példa: AI letörölt egy éles adatbázist, amit utána le is tagadott https://www.businessinsider.com/replit-ceo-apologizes-ai-coding-tool-delete-company-database-2025-7

4.2.7. Karbantartás

bottleneck analysis

find error cause

ticket preprocess (hiányzó adat kitöltése, hasonló hibák keresése, logok keresése)

LLM customer service, customer feedback kérés

5. fejezet

Következtetések

kb. 5 oldal

5.1. A generatív mesterséges intelligencia megítélése

mindenhez ért, de semmiben nem a legjobb, viszont nagyon kényelmes mindenre használni (még arra is, amire lenen jobb (pl. algoritmikus, biztosan helyes) céleszköz) – akinek kalapács van a kezében, az mindent szögnek néz

meglévő mintákon tanul –> nincs igazi out of the box thinking, igazi kreativitás

hallucináció

bullshitelés (igazi adatok elrejtése, hogy aztán egy másik AI-jal visszafejtsék, pedig nem pontos inverzek)

nem féltik a munkájukat, szükség lesz rájuk (mindegy, hogy konkrétan szoftverfejlesztőnek hívjuk-e, de szükség lesz olyanra, aki valós problémákat digitális megoldásokra tud fordítani, mert ehhez mindkét világot ténylegesen érteni kell)

AI használat fontos, de nem váltja ki az embert. inkább társa/okos asszisztens

absztrahálás képessége (emberi gondolkodás sajátja)

igazi gondolkodás nem spórolható meg

hype

5.2. A generatív mesterséges intelligencia szerepe

megengedhető hiba? mennyire kritikus a feladat?

szoftverfejlesztésben jól használható

AI létrehozás veszélyes, mert úgyse nézi át, akinek át kéne

de reviewra, kontextus magyarázatra, nem kritikus dolgokra hasznos és hatékony

kódolás túlreprezentáltság:

MIT 95%-ban kudarc: szigetszerűek

5.3. A generatív mesterséges intelligencia használatának kihívásai

jogi környezet

vállalati környezet (adatbiztonság, biztonsági megoldások kiherélik a toolokat)

céges tudásbázis integrálás (JIRA, Confluence, kódok, DB-k)

elfogy a token

túlságosan hozzászokunk (elbutulás)

unalmas munka elvégzése helyett unalmas átnézni az AI outputot –> emberi természet, luistaság –> át se nézzük igazán –> veszély (vannak hibák, amiekt elkövetni nem követünk el, de mégsem vesszük észre)

Vince: LLM válasz validálása: ehhez szükséges a tudás formalizálása –> kihívás: emberiség tudását formalizálni (ontológia?)

5.4. Akciólista a generatív mesterséges intelligencia bevezetéséhez

jogi környezet tisztázása: mit lehet?

mérhető keretrendszer: ROI, usage, tokenek, költségek

céges tudásbázis integrálása (kockázatok felmérése)

rendszeres dolgozók közti tudásmegosztás, dedikált idő a tanulásra, kísérletezésre

5.5. Kutatásmódszertan értékelése

újrafelhasználható módszer a céges AI használat/gondolkodás felmérésére

SDLC jó keretrendszer

diverz csapatok (tapasztalat, beosztás, cégnél töltött évek, cégen belül min dolgoznak)

szeretnek ilyenekről beszélgetni az emberek

AI only, ..., human only besorolás nem működött

panel létszám OK, 1 óra kevés, 1,5 óra OK

nem reprezentatív

6. fejezet

Összegzés

kb. 2 oldal

Köszönetnyilvánítás

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

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., & 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

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]. 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

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.

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

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

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.

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.

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

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.

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.

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/

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.

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

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&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

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

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