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.

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 dolgozat elméleti alapját adó háttérismereteket, a szoftverfejlesztést és a generatív mesterséges intelligenciát, valamint a kettő találkozását. 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. Végül a 6. fejezetben összegzem a dolgozatot.

# 2. fejezet

# Háttérismeretek

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 háttérismereteket. 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 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.).

1. **Tervezés (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. **Tervezés (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.

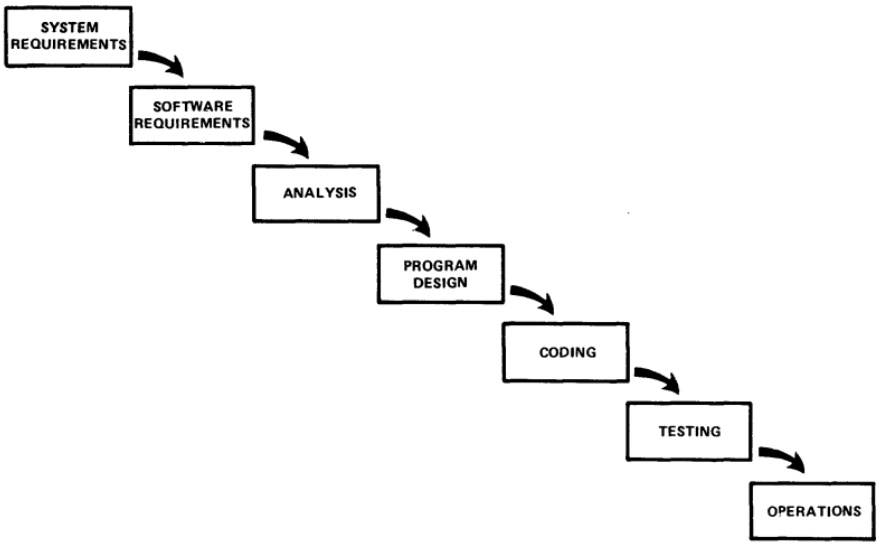
A fázisok egymásra épülnek, a különböző fejlesztési modellek azonban már eltérően értelmezhetik a fázisok közti átmeneteket. Míg a legegyszerűbb megközelítés szerint a fázisok lineárisan követik egymást, más modellek szerint iteratívan is végrehajthatók. Látható tehát, hogy az SDLC csupán testre szabható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

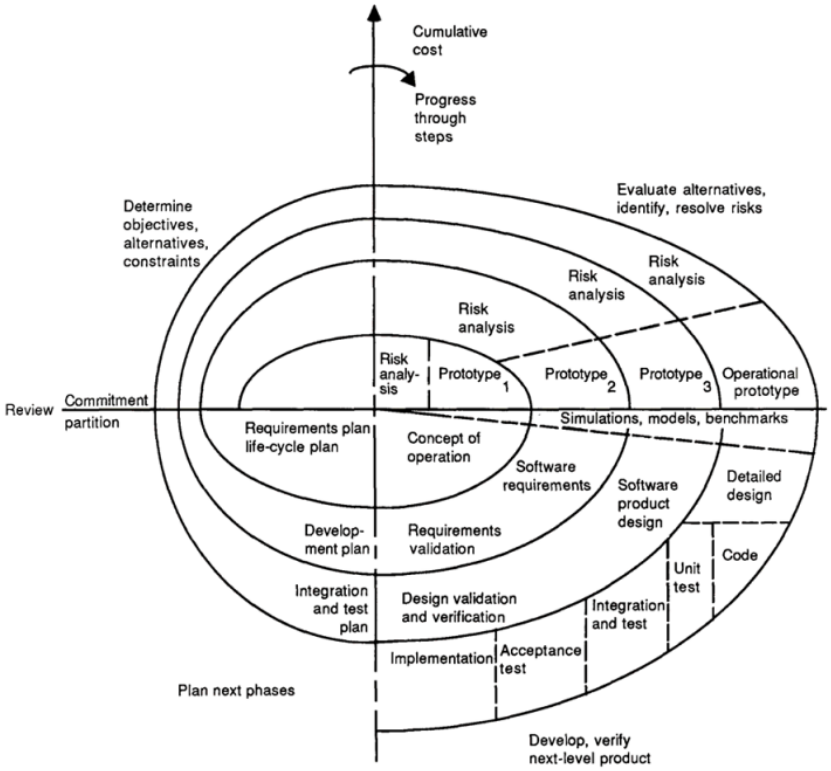
fejlesztenek, majd megtervezik a következő ciklust. Ez a modell különösen olyan összetett projektekben alkalmazható, ahol a követelmények gyakran változnak, vagy magas a kockázat.

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.

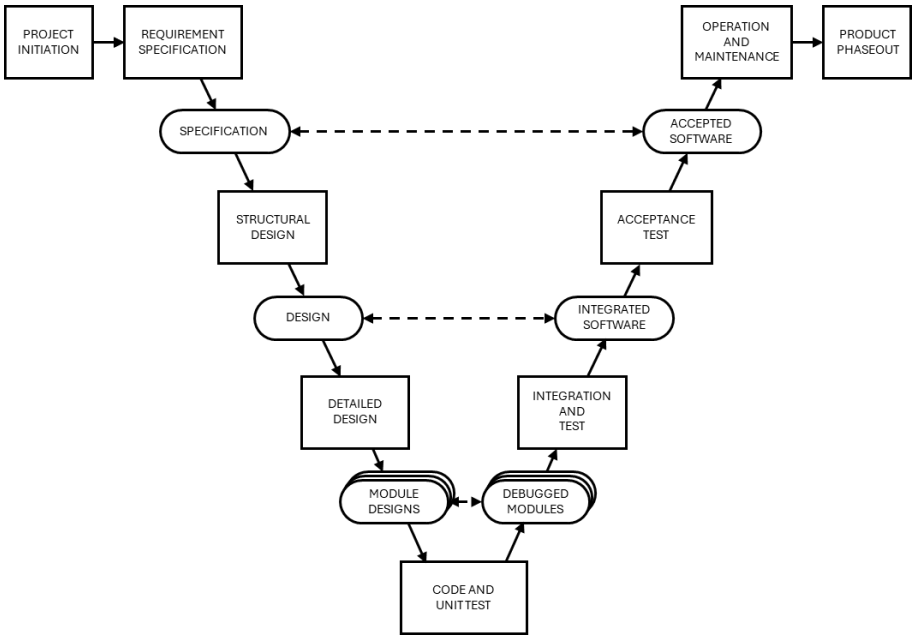
### 2.1.3. Agilis módszertanok

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, 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

## 2.3. Mesterséges intelligencia a szoftverfejlesztésben

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

## 3. fejezet

# Kutatásmódszertan

# 4. fejezet

# Kutatási eredmények

# 5. fejezet

# Következtetések

# 6. fejezet

# Összegzés

# Köszönetnyilvánítás

# Irodalomjegyzék


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

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

Beck, K. (2004). *Extreme Programming Explained: Embrace Change* (2nd 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* (3rd kiad.). O'Reilly Media.

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

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* (2nd 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.). *What is the software development lifecycle (SDLC)?* [Letöltve: 2025-11-02]. IBM. https://www.ibm.com/think/topics/sdlc

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.

Matvitskyy, O., Iijima, K., West, M., Davis, K., Jain, A., & Vincent, P. (2023). *Magic Quadrant for Enterprise Low-Code Application Platforms* [Document ID: G00793788, Published 17 October 2023]. Gartner Research. https://www.gartner.com/en/documents/4843031

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

Richardson, C., & Rymer, J. R. (2014). *New Development Platforms Emerge for Customer-Facing Applications* (Forrester Report) [Ez a jelentés alkotta meg a "low-code" kifejezést]. 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.

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

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

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