

# Kapitola 1

## Analýza

### 1.1 Využitie umelej inteligencie v softvérovom inžinierstve

S príchodom vyvinutejších foriem umelej inteligencie (UI) dochádza k výrazným zmenám v tom, ako softvérové systémy vznikajú a ako sú udržiavané. Softvérové inžinierstvo (SE) sa postupne presúva od tradičných, ľudsky riadených procesov cez hybridné modely, v ktorých človek spolupracuje s inteligentnými systémami až k novému vývojovému bodu SE – označovanému aj ako SE 3.0, v ktorom UI už vystupuje ako aktívny účastník vývojového procesu [11, 18].

#### 1.1.1 Vývoj softvérového inžinierstva

Z historického hľadiska možno vývoj SE rozdeliť do štvorstupňového modelu. Tradičné SE 1.0 bolo založené na manuálnom písaní kódu, explicitných špecifikáciách a deterministických procesoch [11]. Neskôr v SE 1.5 sa objavili nástroje poskytujúce prediktívnu asistenciu, ako sú automatické dopĺňania kódu, ktoré však slúžili výhradne len ako podpora pri kódovaní [11, 18].

S nástupom veľkých jazykových modelov (LLM) sa SE dostáva do fázy SE 2.0, v ktorej UI dokáže generovať celé funkcie, testy či dokumentáciu na základe prirodzeného jazykového zadania [27, 18]. V tejto fáze si vývojár zachováva rozhodujúcu rolu pri definovaní úloh a hodnotení výstupov generovaných systémom.

Najnovší vývoj smeruje k tzv. agentickému SE 3.0, kde autonómne UI systémy dokážu plánovať kroky, využívať externé nástroje, iteratívne opravovať vlastné riešenia a dokonca samostatne vytvárať a odosielať zmeny do repozitárov [11, 3]. V tomto modeli sa úloha vývojára posúva od samotnej implementácie k orchestrácii, kontrole a hodnoteniu výsledkov práce UI.

### 1.1.2 Silné a slabé stránky umelej inteligencie

Analýza súčasných výskumov ukazuje, že UI dosahuje najlepšie výsledky v oblastiach, ktoré sú silno previazané s prirodzeným jazykom alebo majú vysokú mieru štruktúrálnej pravidelnosti.

Jednou z najvýraznejších oblastí je generovanie dokumentácie a sumarizácia kódu [11, 27]. LLM dokážu vytvárať zrozumiteľné popisy funkcií, vysvetlenia existujúceho kódu a technickú dokumentáciu s vysokou mierou akceptácie zo strany vývojárov. Ďalšou oblasťou, kde sa UI osvedčila, je rýchle prototypovanie a generovanie jednoduchého kódu. Modely sú schopné vytvárať syntakticky správne riešenia na základe stručného opisu problému, čím výrazne urýchľujú počiatočné fázy vývoja [27, 18].

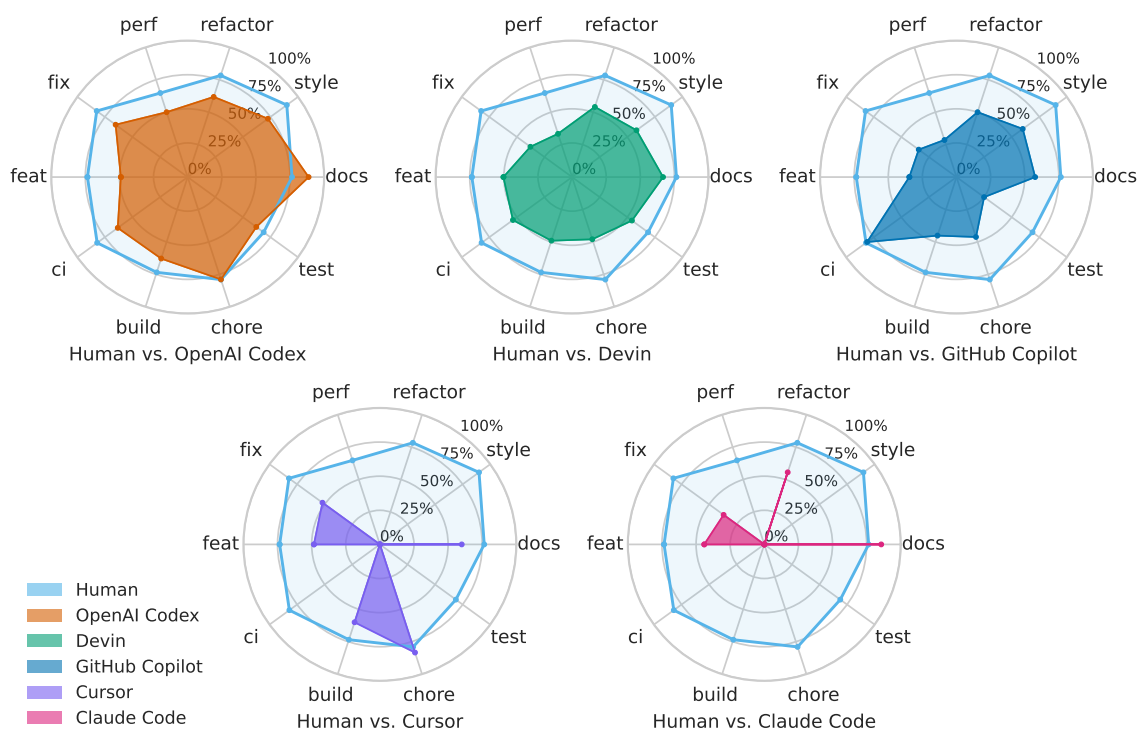
Napriek rýchlemu pokroku existujú oblasti SE, v ktorých súčasné UI systémy narážajú na zásadné limity. Ide najmä o úlohy vyžadujúce hlboké sémantické porozumenie, globálny kontext a dlhodobú konzistenciu [5, 27].

Jedným z hlavných problémov je porozumenie požiadavkám a návrh architektúry systému. LLM môžu nesprávne interpretovať neformálne alebo neúplné zadania a navrhovať riešenia, ktoré síce pôsobia uspokojivo, no nezodpovedajú skutočným potrebám systému [5]. Výrazné obmedzenia sa prejavujú aj pri vývoji komplexného aplikačného kódu a pri zásahoch do architektúry systému. Výskumy ukazujú, že UI generuje skôr jednoduché, lokálne zmeny a vyhýba sa úpravám, ktoré by výrazne menili štruktúru systému [11, 18]. Obmedzenia sa prejavujú aj pri testovaní a overovaní správnosti. Hoci LLM dokážu generovať testy, tieto testy často pokrývajú len bežné scenáre a nezachytávajú hraničné alebo neštandardné prípady [5, 27].

### 1.1.3 Riziká a dopad umelej inteligencie na softvérové inžinierstvo

Integrácia UI do softvérového inžinierstva so sebou prináša viacero zásadných problémov. Jedným z najčastejšie diskutovaných je rozdiel medzi výkonom na benchmarkoch a správaním v reálnych projektoch. Kým v kontrolovaných podmienkach dosahujú LLM vysokú úspešnosť, v otvorených softvérových ekosystémoch sú UI-generované príspevky výrazne menej často akceptované než ľudské [11]. Obr. 1.1 ilustruje rozdiely v miere akceptácie pull requestov medzi ľudskými vývojármi a autonómnymi kódujúcimi agentmi podľa typu úlohy, pričom údaje sú prevzaté zo štúdie Li et al. [11].

Ďalším významným problémom sú halucinácie, teda generovanie kódu alebo vysvetlení, ktoré pôsobia presvedčivo, no sú fakticky nesprávne [5, 18]. S rastúcou autonómiou UI agentov sa zvyšuje aj riziko nekontrolovaného správania. Autonómne systémy, ktoré plánujú a vykonávajú kroky bez priameho zásahu človeka, môžu vykonať nežiaduce zmeny, ak sú ich ciele nepresne špecifikované alebo ak zlyhá mechanizmus spätnej



Obr. 1.1: Miera akceptácie pull requestov podľa typu úlohy pre ľudských vývojárov a autonómnych kódujúcich agentov (prevzaté z [11])

väzby [3]. A s týmto súvisí aj zodpovednosť a autorstvo UI-generovaného kódu. V prípadoch, keď nie je jasne označené, že kód vznikol za pomoci UI, vznikajú problémy s auditovateľnosťou [11, 18].

Všetky analyzované práce sa zhodujú v tom, že UI softvérových inžinierov v dohľadnej budúcnosti nenahradí, ale zásadne zmení charakter ich práce [11, 18, 3]. Vývojár sa postupne presúva od manuálneho písania kódu k formulovaniu cieľov, hodnoteniu výstupov a riadeniu spolupráce s UI systémami. Tieto zistenia vytvárajú východisko pre detailnejšiu analýzu generovania zdrojového kódu pomocou LLM, ktorej sa venuje nasledujúca podkapitola.

## 1.2 Generovanie zdrojového kódu pomocou veľkých jazykových modelov

V súčasnom softvérovom inžinierstve (SE) predstavujú veľké jazykové modely (LLM) dominantnú triedu systémov umelej inteligencie (UI) využívaných na automatizované generovanie zdrojového kódu, pričom generovanie zdrojového kódu predstavuje jednu z najvýznamnejších aplikácií LLM v oblasti SE. Vďaka tréningu na rozsiahlych korpusoch kombinujúcich prirodzený jazyk a zdrojový kód dokážu tieto modely zachytávať syntaktické vzory a opakujúce sa štrukturálne schémy programov [8]. Súčasný výskum

Tabuľka 1.1: Porovnanie výkonu všeobecných a kódu-orientovaných jazykových modelov na open-domain a doménovo špecifických datasetoch (prevzaté z [6]).

Dataset	General-purpose		Code-oriented			
	ChatGPT-3.5-154B		CodeLlama-7B		PolyCoder-2.7B	
	BLEU	CodeBLEU	BLEU	CodeBLEU	BLEU	CodeBLEU
<i>Open-domain datasets</i>						
HumanEval	39.00	30.05	18.13	17.99	22.22	13.81
CodeSearchNet (Golang)	11.16	25.29	16.14	19.63	15.97	19.56
<i>Domain-specific datasets</i>						
Gin	5.81	17.73	8.47	18.07	8.13	17.05
Prometheus	1.75	12.17	1.79	13.13	1.77	12.11
gRPC-go	2.53	12.15	57.61	60.39	55.36	57.52
Unreal Engine	5.22	10.57	0.73	10.21	0.94	9.87
Cocos2d-x	3.71	12.92	16.76	25.87	13.83	23.83
Bgfx	0.86	8.09	0.55	7.72	0.76	7.16

ukazuje, že LLM možno efektívne využiť v širokom spektre úloh, od jednoduchého dopĺňania kódu až po generovanie celých funkcií, testov či častí aplikačnej logiky [8]. Tento potenciál je však zároveň sprevádzaný významnými obmedzeniami [6, 23, 2].

### 1.2.1 Modely a typy úloh generovania kódu

Existujúce práce rozlišujú medzi dvoma základnými triedami LLM používaných na generovanie kódu. Prvú skupinu tvoria všeobecné jazykové modely, ktoré sú primárne trénované na prirodzenom jazyku a až sekundárne aplikované na programátorské úlohy. Tieto modely vynikajú schopnosťou porozumieť zadaniu, no často vykazujú nižšiu presnosť pri dodržiavaní striktnej syntaxe a sémantiky programovacích jazykov. Druhú skupinu tvoria modely špecializované na kód, ktoré sú trénované alebo doladené na rozsiahlych dátach obsahujúcich zdrojový kód. Tieto modely dosahujú vyššiu syntaktickú správnosť a lepšie zvládajú dlhodobé závislosti medzi premennými, funkciami a dátovými štruktúrami [8]. Empirické porovnanie výkonu všeobecných a kódu-orientovaných jazykových modelov (tabuľka 1.1) ukazuje, že zatiaľ čo všeobecné modely dosahujú konkurencieschopné výsledky na open-domain benchmarkoch, pri doménovo špecifických datasetoch výrazne zaostávajú za modelmi trénovanými priamo na zdrojovom kóde [6].

Z pohľadu úloh možno generovanie kódu rozdeliť na viacero kategórií. Medzi najčastejšie patrí generovanie kódu z prirodzeného jazykového popisu, dopĺňanie existujúceho kódu, transformácia alebo refaktoring kódu a oprava chýb [8]. Samostatnú kategóriu tvoria špecializované úlohy, ako je generovanie optimalizačného kódu alebo kódu využívajúceho konkrétne doménové knižnice a aplikačné rozhrania [6, 19].

Tabuľka 1.2: Typy chýb v kóde generovanom veľkými jazykovými modelmi na vybraných benchmarkoch. Hodnoty vyjadrujú percentuálne zastúpenie jednotlivých kategórií chýb (prevzaté z [2]).

Bug types	HumanEval plus			MBPP plus			APPS plus		
	GPT-4	GPT-3.5	Claude-3	GPT-4	GPT-3.5	Claude-3	GPT-4	GPT-3.5	Claude-3
A.1 Incomplete syntax structure	0.0	0.6	0.0	0.0	0.5	0.0	0.2	1.5	0.2
A.2 Incorrect indentation	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	0.0
A.3 Library import error	0.6	2.4	0.0	0.3	0.5	0.0	1.0	2.0	2.5
A Syntax bug	0.6	3.0	0.0	0.3	1.0	0.0	0.2	4.7	2.7
B.1 API misuse	0.0	2.4	0.6	2.4	1.0	1.8	0.5	1.3	1.0
B.2 Definition missing	0.0	5.5	0.0	0.0	0.0	0.0	0.5	2.2	1.8
B.3 Incorrect boundary condition check	0.0	3.0	0.0	1.5	0.5	1.5	2.8	4.3	1.7
B.4 Incorrect argument	0.0	0.0	0.0	1.0	0.0	0.3	1.5	0.3	1.0
B.5 Minors	0.6	0.6	0.6	0.5	0.0	2.3	1.8	1.7	2.2
B Runtime bug	0.6	11.6	1.2	4.0	4.0	2.3	7.2	9.8	7.7
C.1 Misunderstanding and logic error	12.8	20.7	17.1	12.0	15.5	13.8	31.3	46.5	37.5
C.2 Hallucination	0.6	0.0	1.2	1.8	5.8	3.0	7.0	3.0	4.5
C.3 Input/output format error	0.0	0.0	1.2	2.8	0.8	2.8	0.3	2.7	1.7
C.4 Minors	0.0	0.0	0.0	0.0	0.8	0.3	0.7	2.7	2.0
C Functional bug	13.4	20.7	19.5	16.5	22.8	19.8	39.3	54.8	45.7
D Ambiguous problem description	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.8	0.8

### 1.2.2 Silné a slabé stránky LLM pri generovaní kódu

Empirické štúdie potvrdzujú, že LLM sú obzvlášť efektívne pri generovaní krátkeho a stredne zložitého kódu, kde je riešenie možné odvodiť z lokálnych vzorov a bežných programátorských konštrukcií. Táto schopnosť robí z LLM vhodný nástroj na rýchle prototypovanie a podporu vývojára pri rutinných úlohách [8].

Na druhej strane viaceré rozsiahle empirické analýzy ukazujú, že generovaný kód často obsahuje netriviálne chyby. Medzi najčastejšie patria logické chyby, nesprávne ošetrovanie okrajových prípadov, chybné používanie aplikačných rozhraní a nekonzistentná práca so stavom programu [23, 2]. To ilustruje tabuľka 1.2, z ktorej vyplýva, že dominantnú časť tvoria funkčné a logické chyby, zatiaľ čo syntaktické chyby sa vyskytujú len zriedkavo. Významným problémom je aj skutočnosť, že mnohé z týchto chýb nie sú zachytiteľné bežnými automatickými metrikami ani jednoduchými unit testami, čo vedie k nadhodnocovaniu kvality generovaného kódu na benchmarkoch [2].

Osobitnou výzvou je generovanie doménovo špecifického kódu. Štúdie zamerané na túto oblasť ukazujú, že modely bez doménovej adaptácie často zlyhávajú pri používaní špecifických knižníc, porušujú doménové pravidlá alebo generujú neexistujúce rozhrania. Prenos znalostí medzi rôznymi doménami je obmedzený a úspech v jednej oblasti neznamená automaticky dobrý výkon v inej. Tieto zistenia poukazujú na limity univerzálneho prístupu k generovaniu kódu a zdôrazňujú potrebu cielenej adaptácie modelov [6].

Ďalším významným aspektom je bezpečnosť. Bez explicitných inštrukcií majú LLM tendenciu generovať kód s bežnými bezpečnostnými zraniteľnosťami, ako je nedostatočná validácia vstupov alebo nebezpečná práca s externými zdrojmi. [24].

### 1.2.3 Prístupy k zlepšeniu generovania kódu

S cieľom zmierniť identifikované limity sa výskum zameriava na rôzne techniky zlepšovania generovania kódu. Jednou z najdostupnejších stratégií je prompt engineering, pri ktorom sa formulácia vstupu využíva na explicitné usmernenie správania modelu [8]. Výskumy ukazujú, že bezpečnostne orientované alebo rolovo definované prompty môžu výrazne znížiť výskyt zraniteľností v generovanom kóde, hoci výsledky nie sú vždy konzistentné naprieč rôznymi úlohami [24].

Ďalším smerom je in-context learning, pri ktorom sú modelu poskytnuté ukážkové riešenia priamo v kontexte vstupu. Efektivita tejto techniky však silne závisí od výberu príkladov. Model-aware prístupy, ktoré berú do úvahy správanie konkrétneho modelu pri výbere kontextu, dosahujú stabilnejšie a presnejšie výsledky než náhodné alebo heuristické stratégie [13].

Významné zlepšenia prináša aj explicitné štruktúrovanie uvažovania modelu prostredníctvom štruktúrovaného chain-of-thought prompting. Rozdelenie riešenia na fázy porozumenia problému, návrhu algoritmu a samotnej implementácie vedie k konzistentnejšiemu kódu a znižuje výskyt logických chýb [12].

Pre náročnejšie alebo doménovo špecifické úlohy sa ako efektívne ukazujú techniky jemného doladenia. Parameter-efficient fine-tuning umožňuje adaptovať model na konkrétnu doménu s výrazne nižšími výpočtovými nárokmi než plný tréning, pričom dosahuje porovnateľný výkon [25]. Osobitným príkladom je instruction tuning pre generovanie optimalizačného kódu, ktorý preukázateľne zlepšuje správnosť a stabilitu riešení v tejto presnej a chybovo citlivej oblasti [19].

Analyzované práce sa zhodujú v tom, že žiadna z uvedených techník nepredstavuje univerzálne riešenie. Kvalita generovaného kódu je výsledkom kombinácie vlastností modelu, spôsobu jeho použitia a miery adaptácie na konkrétny kontext [8, 6, 2, 25].

## 1.3 Nástroje pre AI-asistovaný vývoj softvéru

Rýchly rozvoj veľkých jazykových modelov (LLM) viedol k vzniku celej triedy nástrojov určených na podporu vývojárov pri tvorbe softvéru. Táto sekcia sa zameriava na praktické nástroje, nie na všeobecné schopnosti modelov. Tieto nástroje integrujú generatívne modely priamo do vývojového prostredia a umožňujú automatizovať časti procesu, ktoré boli tradične vykonávané manuálne. Analýza týchto nástrojov slúži ako východisko pre identifikáciu ich spoločných obmedzení, ktoré motivujú návrh vlastného agentického prístupu v ďalšej časti práce.

### 1.3.1 Najpoužívanější nástroje pre AI-asistovaný vývoj

Súčasný nástroje pre UI-asistovaný vývoj softvéru možno rozdeliť podľa spôsobu interakcie s vývojárom a miery autonómie. Najrozšírenejšie sú konverzačné systémy a nástroje integrované priamo do vývojových prostredí, pričom v poslednom období sa objavujú aj agentické nástroje schopné vykonávať komplexnejšie pracovné postupy.

ChatGPT a Gemini patria medzi široko používané konverzačné nástroje, ktoré umožňujú generovanie zdrojového kódu na základe prirodzeného jazykového zadania. Ich hlavnou výhodou je flexibilita a schopnosť pracovať s neformálne definovanými požiadavkami. Vývojár môže prostredníctvom dialógu postupne spresňovať zadanie, žiadať alternatívne riešenia alebo vysvetlenie existujúceho kódu. Nevýhodou je absencia priamej integrácie do vývojového prostredia a obmedzený prístup ku globálnemu kontextu rozsiahlejších projektov.

GitHub Copilot sa využíva aj ako nástroj integrovaný do vývojových prostredí, ktorý poskytuje priebežné návrhy kódu počas písania. Je optimalizovaný na dopĺňanie existujúcej implementácie a zrýchlenie rutinných činností. Jeho návrhy sú v tomto prípade silne závislé od lokálneho kontextu súboru, čo zvyšuje plynulosť práce, no zároveň obmedzuje schopnosť riešiť úlohy vyžadujúce globálne uvažovanie nad architektúrou systému.

Medzi nástroje zamerané na profesionálne a cloudové prostredie patrí Amazon CodeWhisperer, ktorý je úzko previazaný s cloudovým ekosystémom a vývojom aplikácií využívajúcich vzdialené služby. Podobne Google Gemini Code Assist integruje generatívne modely do nástrojov určených pre vývoj aplikácií v rámci cloudových a mobilných platforiem. Tieto nástroje rozširujú koncept UI asistencie o podporu špecifických aplikčných rozhraní.

Osobitnú kategóriu tvoria nástroje orientované na príkazový riadok a agentické systémy. Gemini CLI umožňuje využívať generatívne modely mimo vývojového prostredia, napríklad pri analýze repozitárov, generovaní kódu alebo automatizácii vývojových úloh. Roo Code reprezentuje agentický prístup, pri ktorom systém dokáže iteratívne pracovať s projektom, vykonávať viacero krokov a upravovať kód bez neustáleho zásahu vývojára s možnosťou nastavenia roly pre rôzne druhy použitia.

### 1.3.2 Analýza kvality kódu generovaného AI nástrojmi

ChatGPT a GitHub Copilot patria medzi najvýznamnejšie nástroje v oblasti UI-asistovaného generovania zdrojového kódu, čo sa odráža aj v ich častom výskyte v odborných štúdiách a výskumoch. Preto je opodstatnené venovať pozornosť analýze nimi generovaného kódu so zameraním na porovnanie ich vlastností z hľadiska deterministickosti, funkčnej správnosti, komplexnosti a bezpečnosti generovaného kódu.

**Deterministickosť a stabilita výstupov** Pri opakovanom generovaní riešení pre rovnaké zadanie vykazuje ChatGPT výraznú variabilitu výstupov. Rozdiely sa prejavujú nielen v syntaktickej podobe kódu, ale aj v zvolenom algoritmickej postupe a správaní programu pri testovaní. Táto nedeterministickosť komplikuje reprodukovateľnosť výsledkov a znižuje predvídateľnosť správania systému [20].

GitHub Copilot produkuje stabilnejšie návrhy v rámci lokálneho kontextu súboru, keďže jeho generovanie je silne viazané na existujúci kód v bezprostrednom okolí kurzora. Napriek tomu sa pri opakovanej interakcii môžu objaviť rozdielne implementácie, najmä ak sa mení kontext alebo poradie úprav [14]. Stabilita Copilotu je teda vyššia na úrovni lokálnych návrhov, no neeliminuje variabilitu pri komplexnejších úlohách.

**Funkčná správnosť** Kód generovaný oboma nástrojmi je vo väčšine prípadov syntakticky korektný a priamo použiteľný. Pri dôkladnejšej validácii sa však často objavujú logické chyby, neúplná implementácia požiadaviek alebo nesprávne ošetrovanie okrajových prípadov [16, 17, 14].

Rozdiel medzi nástrojmi spočíva najmä v povahe chýb. ChatGPT má tendenciu generovať riešenia, ktoré pôsobia algoritmicke konzistentne, no môžu obsahovať jemné logické nezrovnalosti vyplývajúce z nesprávneho porozumenia zadania [20, 16]. GitHub Copilot častejšie produkuje kód, ktorý zapadá do lokálneho kontextu, avšak nezohľadňuje globálne požiadavky úlohy alebo širšiu architektúru systému [14].

**Komplexnosť a udržiavateľnosť** Generovaný kód v oboch prípadoch často vykazuje zvýšenú komplexnosť. Objavujú sa nadbytočné vetvenia, pomocné premenné alebo redundantné konštrukcie, ktoré nie sú nevyhnutné [16, 17, 14]. Tento jav sa môže ešte zosilniť pri iteratívnych úpravách, keď nové riešenia nadväzujú na pôvodný kód bez jeho zásadného zjednodušenia.

ChatGPT má tendenciu generovať samostatné, ucelené riešenia, ktoré môžu byť zbytočne rozsiahle alebo menej prispôsobené existujúcemu kódu [16, 17]. GitHub Copilot naopak vytvára fragmenty kódu úzko previazané s aktuálnym súborom, čo zvyšuje ich okamžitú použiteľnosť [14].

**Bezpečnostné vlastnosti** Bez explicitného zamerania na bezpečnosť majú výstupy oboch nástrojov tendenciu obsahovať bežné bezpečnostné slabiny, ako je nedostatočná validácia vstupov, nebezpečná práca s externými zdrojmi alebo nevhodné používanie kryptografických mechanizmov [17, 4]. Ako je znázornené na obr. 1.2, významná časť analyzovaných úryvkov kódu obsahovala aspoň jednu bezpečnostnú zraniteľnosť, pričom v nezanedbateľnom počte prípadov sa v jednom úryvku vyskytovalo viacero slabín súčasne.

Language	# Snippets	# Security weaknesses	# Snippets containing security weaknesses	# containing more than one security weakness
Python	419	387	124 (29.5%)	65 (15.5%)
JavaScript	314	241	76 (24.2%)	37 (11.8%)
Total	733	628	200 (27.3%)	102 (13.9%)

Obr. 1.2: Štatistika bezpečnostných slabín v kóde generovanom nástrojom GitHub Copilot pre jazyky Python a JavaScript. Tabuľka zobrazuje počet analyzovaných úryvkov kódu, celkový počet identifikovaných bezpečnostných slabín a podiel úryvkov obsahujúcich jednu alebo viac zraniteľností (prevzaté z [4]).

Iteratívne opravy na základe spätnej väzby dokážu časť bezpečnostných problémov odstrániť, avšak tento proces nie je konzistentný a často vedie k nárastu komplexnosti kódu [16, 17, 4]. Bezpečnostná kvalita generovaného kódu tak výrazne závisí od miery zapojenia vývojára a od použitých kontrolných mechanizmov.

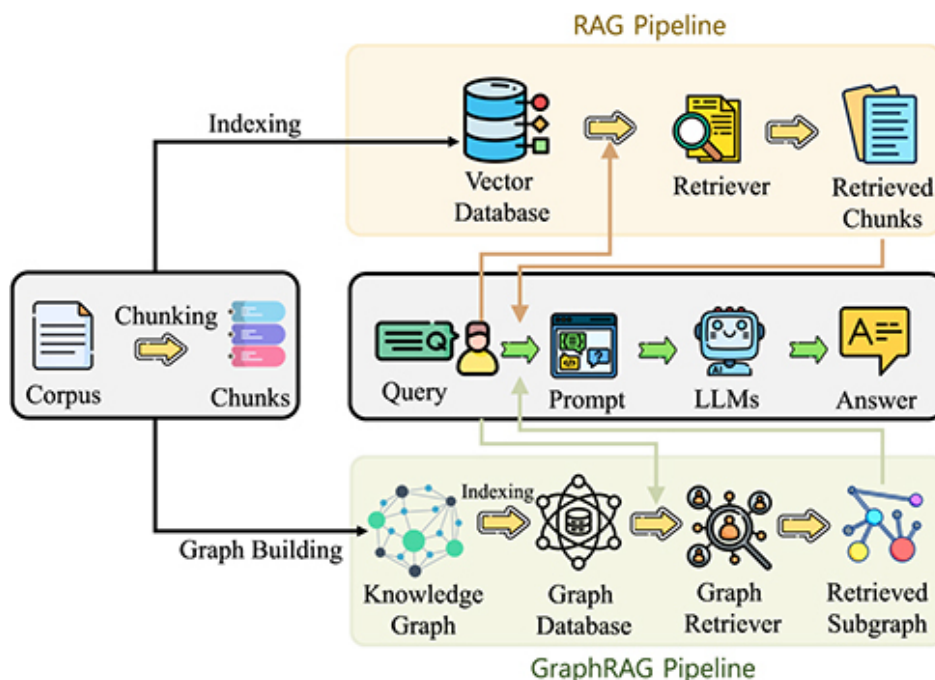
ChatGPT aj GitHub Copilot poskytujú významnú podporu pri vývoji softvéru, no ich výstupy nemožno považovať za plnohodnotnú náhradu manuálnej implementácie. Generovaný kód je vhodný ako východisko pre ďalší vývoj, no vyžaduje systematické testovanie, manuálnu revíziu a často aj bezpečnostnú analýzu [20, 16, 17, 4, 14].

## 1.4 GraphRAG a štruktúrované znalosti

Jedným z kľúčových smerov výskumu zameraného na prekonanie limitov veľkých jazykových modelov je *Graph Retrieval-Augmented Generation* (GraphRAG). Tento prístup rozširuje klasické schémy retrieval-augmented generation o využitie grafovo štruktúrovaných znalostí, ktoré explicitne modelujú entity a vzťahy medzi nimi. [21]. Cieľom tohto prístupu je zvýšiť presnosť, konzistentnosť a kontrolovateľnosť generovaných výstupov prostredníctvom explicitnej práce so znalosťami.

### 1.4.1 Základný princíp

Architektúra GraphRAG systémov sa typicky skladá z troch hlavných fáz: grafového indexovania, grafovo riadeného vyhľadávania a generovania odpovede obohatenej o grafové znalosti [21]. V prvej fáze sú vstupné znalosti transformované do grafovej reprezentácie, kde uzly reprezentujú entity a hrany ich vzájomné vzťahy. V druhej fáze je nad týmto grafom vykonané vyhľadávanie relevantných uzlov, ciest alebo podgrafov vzhľadom na vstupný dotaz. Výsledkom je štruktúrovaný kontext, ktorý je následne využitý v generatívnej fáze ako doplnkový vstup pre jazykový model.



Obr. 1.3: Porovnanie architektúr klasického retrieval-augmented generation (RAG) a GraphRAG. Zatiaľ čo RAG využíva vektorovú databázu a textové úryvky, GraphRAG pracuje s grafovo štruktúrovanou znalostnou bázou a vyhľadáva relevantné podgrafy.

Dôležitým aspektom GraphRAG je skutočnosť, že vyhľadávanie nie je obmedzené na jednorazový výber relevantných informácií. Viaceré prístupy umožňujú viac-krokové alebo iteratívne prechádzanie grafu, čím podporujú komplexné uvažovanie nad vzťahmi medzi entitami [26, 9].

### 1.4.2 Prínosy GraphRAG

Hlavnou výhodou GraphRAG je jeho schopnosť explicitne modelovať a využívať relačné znalosti. Kým klasický RAG je založený predovšetkým na podobnosti textových reprezentácií, GraphRAG umožňuje zachytiť viacstupňové závislosti a logické väzby, ktoré sú pre mnohé úlohy kľúčové [21]. To je obzvlášť dôležité v doménach, kde odpoveď závisí od reťazca vzťahov medzi viacerými entitami, a nie od jednej izolovanej informácie. Rozdiel medzi klasickým prístupom RAG a GraphRAG je schematicky znázornený na obrázku 1.3, kde je viditeľný posun od práce s textovými úryvkami k vyhľadávaniu relevantných podgrafov v znalostnej báze.

V kontexte generovania zdrojového kódu predstavuje GraphRAG prirodzenú odpoveď na viaceré identifikované limity LLM. Ako poukazujú existujúce štúdie, LLM majú obmedzený globálny kontext a nedisponujú explicitným modelom štruktúry softvérových systémov [8]. Grafové reprezentácie umožňujú modelovať závislosti medzi funkciami, triedami, modulmi alebo aplikačnými rozhraniami a poskytujú tak jazykovému

modelu konzistentný a overiteľný kontext.

Ďalším prínosom je flexibilita aktualizácie znalostí. Keďže grafová znalostná báza je oddelená od samotného modelu, je možné ju aktualizovať bez potreby opätovného tréningu alebo jemného doladenia LLM. [26].

### 1.4.3 Obmedzenia, riziká a praktické požiadavky

Napriek svojim výhodám prináša GraphRAG aj viaceré výzvy. Jedným z hlavných problémov je škálovateľnosť, keďže efektívne vyhľadávanie v rozsiahlych grafoch si vyžaduje sofistikované indexačné a vyhľadávacie mechanizmy. Ďalším rizikom je závislosť kvality generovania od kvality samotnej grafovej znalostnej bázy. Neúplné alebo nekonzistentné grafy môžu viesť k chybným alebo zavádzajúcim výstupom, a to aj v prípade silného generatívneho modelu [21, 26].

Použitie GraphRAG si zároveň vyžaduje dodatočné systémové komponenty, ako sú nástroje na konštrukciu a údržbu grafu, mechanizmy na mapovanie dotazov do grafového priestoru a stratégie integrácie grafových informácií do generatívneho procesu [21].

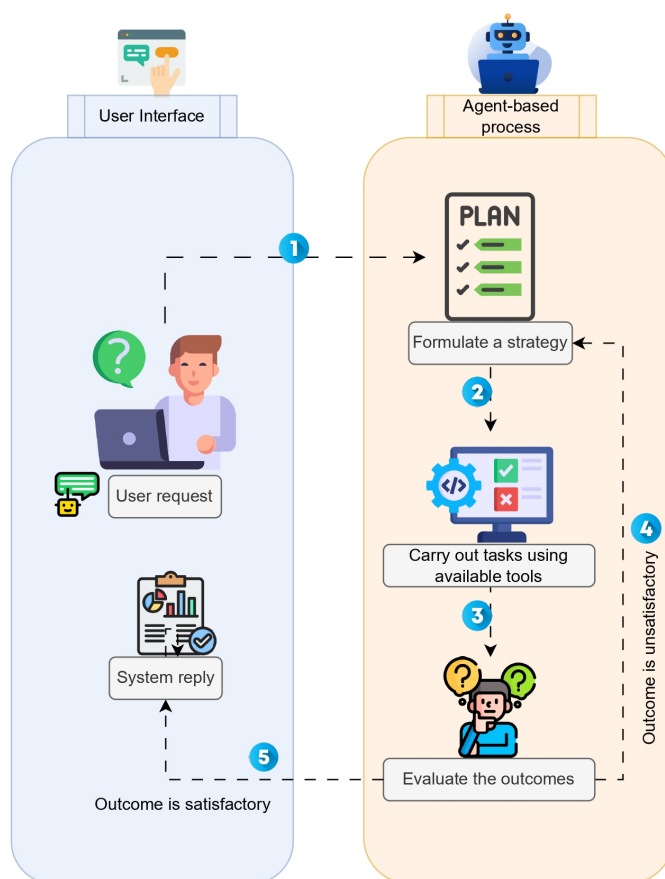
## 1.5 Agentické systémy

S postupným posunom veľkých jazykových modelov (LLM) od statického generovania výstupov k explicitnému uvažovaniu sa v softvérovom inžinierstve (SE) objavuje nová trieda systémov označovaných ako agentické systémy [7, 3]. Tieto systémy reprezentujú ďalší krok v evolúcii využitia umelej inteligencie (UI), v ktorom model nevystupuje len ako pasívny nástroj reagujúci na vstup, ale ako aktívny subjekt schopný samostatne plánovať, vykonávať a vyhodnocovať sekvencie krokov smerujúcich k dosiahnutiu stanoveného cieľa [3].

### 1.5.1 Od jazykových modelov k autonómnym agentom

Základným predpokladom vzniku agentických systémov je rozšírenie LLM o mechanizmy uvažovania, pamäte a spätnej väzby [3]. Samotné generovanie odpovedí, typické pre skoršie použitie LLM, sa ukazuje ako nedostatočné pre riešenie komplexných a dlhodobých úloh [3, 22]. Výskum preto smeruje k modelom, ktoré dokážu rozkladať problém na viacero krokov, iteratívne vyhodnocovať vlastné riešenia a adaptovať ďalšie správanie na základe výsledkov predchádzajúcich akcií [3].

V tomto kontexte sa autonómny agent definuje ako systém, ktorý disponuje cieľom, vnútorným stavom a schopnosťou vykonávať akcie v externom prostredí. Jeho správanie má cyklický charakter, zahŕňajúci vnímanie vstupu, uvažovanie, vykonanie akcie,



Obr. 1.4: Schéma agentického pracovného postupu ilustrujúca cyklus plánovania, vykonávania úloh, hodnotenia výsledkov a iteratívneho zlepšovania riešenia v autonómnych agentických systémoch (prevzaté z [3]).

pozorovanie výsledku a aktualizáciu pamäte. [3]. Agentický cyklus autonómnych systémov, pozostávajúci z plánovania, vykonávania akcií, hodnotenia výsledkov a iteratívnej spätnej väzby, je schematicky znázornený na obr. 2.2.

### 1.5.2 UI agenti a agentická umelá inteligencia

Súčasná literatúra rozlišuje medzi tradičnými UI agentmi a tzv. agentickou UI. Kým pojem UI agent sa historicky vzťahuje na systémy s vopred definovanými pravidlami a cieľmi, agentická UI označuje kvalitatívne vyšší stupeň autonómie [22]. Agentickosť preto nie je binárnou vlastnosťou, ale kontinuum, ktoré zahŕňa mieru iniciatívy, rozsah kontroly nad vykonávanými akciami a schopnosť riešiť dlhodobé úlohy [22].

Väčšina súčasných nástrojov využívajúcich LLM sa nachádza v strednej časti tohto kontinua [22]. Systémy síce dokážu navrhovať ďalšie kroky alebo reagovať na spätnú väzbu, no len zriedkavo fungujú ako plne sebariadené entity bez zásahu človeka [7, 22].

### 1.5.3 Multi-agentové systémy v softvérovom inžinierstve

Osobitnú kategóriu agentických systémov tvoria multi-agentové architektúry, v ktorých viacero agentov spolupracuje na riešení spoločného problému. Každý agent môže mať špecializovanú rolu, napríklad plánovanie úloh, generovanie zdrojového kódu, testovanie alebo validáciu výsledkov. Takýto prístup umožňuje paralelizáciu práce a lepšie rozdelenie zodpovedností, čím sa agentické systémy približujú organizácii ľudských vývojových tímov [7].

Výskum ukazuje, že multi-agentové systémy sú v SE použiteľné naprieč viacerými fázami životného cyklu softvéru, od analýzy požiadaviek cez implementáciu až po testovanie a revíziu kódu. Ich hlavnou výhodou je schopnosť riešiť komplexnejšie úlohy, ktoré presahujú možnosti jedného monolitického modelu [7]. Súčasne však rastie zložitosť koordinácie medzi agentmi a zvyšuje sa riziko šírenia chýb medzi jednotlivými komponentmi systému [7, 3].

### 1.5.4 Použiteľnosť agentických systémov

Agentické systémy nachádzajú uplatnenie najmä v úlohách, ktoré si vyžadujú viacstupňové uvažovanie, dlhodobú konzistenciu a interakciu s externými nástrojmi [3]. V SE ide predovšetkým o automatizované riešenie komplexných vývojových úloh, kde generovanie kódu predstavuje len jednu časť celého procesu. Tieto systémy sú schopné iteratívne upravovať riešenia, reagovať na zlyhania testov a prispôbovať ďalšie kroky na základe priebežnej spätnej väzby [7, 3].

Zároveň sa ukazuje, že efektívnosť agentických systémov nie je daná výlučne veľkosťou použitého jazykového modelu. Naopak, pre autonómne správanie sú často vhodnejšie menšie, špecializované jazykové modely, ktoré poskytujú vyššiu deterministickosť, nižšiu latenciu a lepšiu kontrolovateľnosť [1].

### 1.5.5 Výhody a obmedzenia agentických prístupov

Medzi hlavné výhody agentických systémov patrí schopnosť autonómneho plánovania, iteratívneho zlepšovania riešení a paralelného spracovania úloh [7, 3]. Tieto vlastnosti umožňujú efektívnejšie zvládanie komplexných problémov a znižujú potrebu neustáleho zásahu vývojára [7]. Agentické systémy tak podporujú posun roly človeka od priamej implementácie k dohľadu, hodnoteniu a riadeniu cieľov [7, 3].

Na druhej strane však takéto prístupy prinášajú aj významné riziká. Nekontrolovaná autonómia môže viesť k nežiadanejmu správaniu, najmä ak sú ciele systému nepresne špecifikované [3, 22]. Chyby vzniknuté v jednom kroku agentického cyklu sa môžu šíriť do ďalších fáz a znižovať kvalitu celého riešenia [3]. Ďalším problémom je absencia štandardizovaných metrík na hodnotenie agentického správania, čo sťažuje objektívne

porovnávanie jednotlivých prístupov [7, 22].

Agentické systémy preto nemožno vnímať ako univerzálne riešenie, ale ako ďalší krok v postupnej evolúcii využitia UI v softvérovom inžinierstve [22, 1]. Ich potenciál spočíva v rozšírení možností automatizácie a podpory vývojárov, nie v úplnej náhrade ľudského rozhodovania [7, 22].

## 1.6 Trendy a vízie budúceho vývoja

Analýza súčasných výskumných smerov naznačuje, že vývoj umelej inteligencie (UI) v softvérovom inžinierstve (SE) smeruje k zásadnej zmene charakteru vývojových procesov, roly človeka a samotnej podoby softvérových systémov. Budúce trendy nepredstavujú len postupné zlepšovanie kvality generovaného zdrojového kódu, ale poukazujú na prechod k autonómnym a kooperatívnym systémom [10, 11].

### 1.6.1 Modulárne agentické architektúry

Napriek rastúcej autonómii sa budúci vývoj neuberá smerom k monolitickým riešeniam založeným na jednom univerzálnom modeli. Naopak, čoraz väčší dôraz sa kladie na modulárne agentické architektúry, v ktorých spolupracuje viacero špecializovaných komponentov. Každý z týchto komponentov je zodpovedný za konkrétnu úlohu, ako je plánovanie, generovanie kódu, hodnotenie výsledkov alebo kontrola správnosti [1, 11].

Takýto prístup umožňuje vyššiu kontrolovateľnosť, lepšiu škálovateľnosť a efektívnejšie využitie výpočtových zdrojov. Menšie a špecializované modely zároveň poskytujú stabilnejšie a predvídateľnejšie správanie, čo je kľúčové pre systémy vykonávajúce autonómne rozhodnutia v dlhodobom horizonte [1].

### 1.6.2 Zmena charakteru zdrojového kódu a roly vývojára

S rastúcim podielom strojovo generovaného kódu dochádza aj k zmene jeho charakteru. Kód je čoraz častejšie optimalizovaný pre potreby strojového spracovania a automatickej validácie, nie primárne pre ľudskú čitateľnosť [10]. Táto transformácia má priamy dopad na rolu vývojára [11, 10]. Človek prestáva byť hlavným autorom implementácie a jeho úloha sa presúva k dohľadu nad autonómnymi procesmi, interpretácii výsledkov a riešeniu výnimočných situácií [11]. Budúci vývoj softvéru tak nie je charakterizovaný snahou o nahradenie vývojárov, ale o vytvorenie prostredia, v ktorom ľudia a autonómne systémy spolupracujú na dosahovaní spoločných cieľov [11, 10].

### 1.6.3 Umelá inteligencia ako člen vývojového tímu

S rastúcou mierou autonómie sa mení aj spôsob, akým je UI integrovaná do vývojových tímov. Budúce systémy sú koncipované ako aktívni účastníci tímovej spolupráce, ktorí preberajú zodpovednosť za konkrétne úlohy v rámci životného cyklu softvéru. Takéto systémy dokážu iniciovať implementačné kroky, navrhovať úpravy existujúceho kódu, analyzovať výsledky testov a reagovať na zistené nedostatky [11].

Vývojové tímy sa tak postupne menia na hybridné zoskupenia pozostávajúce z ľudí a autonómnych systémov [11, 10]. Efektivita takéhoto tímu nezávisí len od schopností jednotlivých komponentov, ale najmä od kvality ich koordinácie, jasne definovaných rolí a mechanizmov spolupráce [11].

### 1.6.4 Autonómny návrh architektúr a akcelerácia vývoja

Významným trendom presahujúcim samotné generovanie kódu je automatizácia návrhu architektúr softvérových a modelových systémov. Návrh architektúry, ktorý bol tradične založený na skúsenostiach a intuícii expertov, je čoraz častejšie formulovaný ako optimalizačný problém riešiteľný autonómnymi systémami. Tieto systémy dokážu skúmať rozsiahle priestory možných riešení a objavovať architektúry, ktoré nie sú obmedzené ľudskými dizajnovými vzormi [15].

Autonómny návrh architektúr má zásadné dôsledky pre tempo vývoja [10, 15]. Ak systémy dokážu samostatne navrhovať a zlepšovať vlastné komponenty, vzniká mechanizmus spätnej väzby, ktorý vedie k urýchľovaniu ďalšieho pokroku [10]. Vývoj sa tak postupne presúva od lineárneho zlepšovania k dynamickému procesu, v ktorom je značná časť inovácií generovaná samotnými systémami [10, 15].

## 1.7 Cieľ práce, výskumné otázky a hypotézy

### 1.7.1 Cieľ práce

Cieľom tejto práce je preskúmať možnosti využitia UI pri generovaní zdrojového kódu v kontexte moderného SE a analyzovať obmedzenia súčasných prístupov založených na LLM. Na základe tejto analýzy je cieľom práce navrhnúť a experimentálne overiť koncept agentického prístupu ku generovaniu zdrojového kódu, ktorý kombinuje generatívne schopnosti LLM so systematickým využitím externých, štruktúrovaných znalostí. Cieľom navrhovaného prístupu je zlepšiť kvalitu generovaného kódu prostredníctvom riadeného pracovného postupu, iteratívneho spracovania úloh a zapojenia mechanizmov spätnej väzby.

Súčasťou práce je experimentálne overenie navrhnutého prístupu na praktickom scenári, ktoré umožňuje posúdiť jeho prínosy a obmedzenia v porovnaní s bežnými

spôsobmi využitia LLM pri vývoji softvéru.

### **1.7.2 Výskumné otázky a hypotézy**

# Literatúra

- [1] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025.
- [2] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Muling Wu, Yunbo Tao, Ming Zhang, Mingxu Chai, Jessica Fan, Zhiheng Xi, Rui Zheng, Yueming Wu, Ming Wen, Tao Gui, Qi Zhang, Xipeng Qiu, and Xuanjing Huang. What is wrong with your code generated by large language models? an extensive study. *Science China Information Sciences*, 69(1):112107, January 2026.
- [3] Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. From llm reasoning to autonomous ai agents: A comprehensive review, 2025.
- [4] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. Security weaknesses of copilot-generated code in github projects: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 34(8), October 2025.
- [5] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. The current challenges of software engineering in the era of large language models. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025.
- [6] Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(3), February 2025.
- [7] Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025.
- [8] Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications, 2025.

- [9] Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, Yang Song, Chen Zhu, Hengshu Zhu, and Ji-Rong Wen. KG-agent: An efficient autonomous agent framework for complex reasoning over knowledge graph. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9505–9523, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [10] Daniel Kokotajlo, Scott Alexander, Thomas Larsen, Eli Lifland, and Romeo Dean. AI 2027. <https://ai2027.com/>, 2024. Online report, accessed 2025-10-13.
- [11] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering, 2025.
- [12] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2), January 2025.
- [13] Jia Li, Chongyang Tao, Jia Li, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. Large language model-aware in-context learning for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(7), August 2025.
- [14] Shuang Li, Yuntao Cheng, Jinfu Chen, Jifeng Xuan, Sen He, and Weiyi Shang. Assessing the performance of ai-generated code: A case study on github copilot. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, 2024.
- [15] Yixiu Liu, Yang Nan, Weixian Xu, Xiangkun Hu, Lyumanshan Ye, Zhen Qin, and Pengfei Liu. Alphago moment for model architecture discovery, 2025.
- [16] Yue Liu, Thanh Le-Cong, Ratnadira Widayarsi, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Trans. Softw. Eng. Methodol.*, 33(5), June 2024.
- [17] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024.
- [18] Michael R. Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. Automatic programming: Large language models and beyond. *ACM Trans. Softw. Eng. Methodol.*, 34(5), May 2025.

- [19] Zeyuan Ma, Yue-Jiao Gong, Hongshu Guo, Jiacheng Chen, Yining Ma, Zhiguang Cao, and Jun Zhang. Llamoco: Instruction tuning of large language models for optimization code generation. *IEEE Transactions on Evolutionary Computation*, pages 1–1, 2026.
- [20] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2), January 2025.
- [21] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey. *ACM Trans. Inf. Syst.*, 44(2), December 2025.
- [22] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges. *Information Fusion*, 126:103599, February 2026.
- [23] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: an empirical study. *Empirical Software Engineering*, 30(3):65, February 2025.
- [24] Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhif, and Riccardo Scandariato. Prompting techniques for secure code generation: A systematic investigation. *ACM Trans. Softw. Eng. Methodol.*, 34(8), October 2025.
- [25] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Trans. Softw. Eng. Methodol.*, 34(7), August 2025.
- [26] Qinggang Zhang, Shengyuan Chen, Yuanchen Bei, Zheng Yuan, Huachi Zhou, Zijin Hong, Hao Chen, Yilin Xiao, Chuang Zhou, Junnan Dong, Yi Chang, and Xiao Huang. A survey of graph retrieval-augmented generation for customized large language models, 2025.
- [27] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianhong Guo, Weicheng Wang, and Yanlin Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, December 2024.