Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Podpora pochopenia softvéru

Diplomová práca

2026
Bc. Teodor Fuček

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Podpora pochopenia softvéru

Diplomová práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	doc. Ing. Ivan Polášek, PhD.

Bratislava, 2026
Bc. Teodor Fuček

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Teodor Fuček
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Podpora pochopenia softvéru
Supporting software comprehension

Anotácia: Každá softvérová spoločnosť si cení svojich kvalitných expertov a ich čas, ktorý musia venovať novoprijatým zamestnancom a začiatočníkom na vysvetlenie konkrétneho obsahu a kontextu zdrojového kódu. Vysvetľovanie neznámeho systému a jeho jednotlivých elementov a modulov, metód a tried pomocou AI pre začiatočníka v tíme až po zobrazenie modelu (napr. UML diagramu tried alebo UML sekvenčného diagramu) cez PlantUML by mohol finančne aj časovo tento proces zefektívniť.

Cieľ: Navrhnite postup využitia AI pri generovaní dokumentácie alebo interaktívnom vysvetľovaní softvéru.

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 06.11.2024

Dátum schválenia: 08.11.2024

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Abstrakt

Táto diplomová práca sa zameriava na podporu pochopenia existujúceho softvéru pomocou nástrojov umelej inteligencie. Motiváciou je znížiť čas a náklady, ktoré skúsení vývojári venujú vysvetľovaniu zdrojového kódu novým členom tímu. V práci navrhujeme a implementujeme nástroj, ktorý automaticky spracuje repozitár so zdrojovým kódom, vykoná statickú analýzu a pomocou jazykového modelu vygeneruje prehľadnú dokumentáciu a vizuálne výstupy vo formáte PlantUML. Riešenie cieli na dve skupiny používateľov, začiatočníka, ktorý potrebuje rýchlu orientáciu v projekte, a skúseného programátora, ktorý vyžaduje technicky presné a konzistentné zhrnutie. Výsledkom je praktický postup a prototyp nástroja, ktorý zrýchľuje onboarding a zlepšuje porozumenie architektúre a kľúčovým častiam kódu.

Kľúčové slová: pochopenie softvéru, dokumentácia, veľké jazykové modely, UML diagramy, PlantUML

Abstract

This diploma thesis focuses on supporting the understanding of existing software using artificial intelligence tools. The motivation is to reduce the time and cost that experienced developers spend explaining source code to new team members. In this thesis, we design and implement a tool that automatically processes a source-code repository, performs static analysis, and uses a language model to generate clear documentation and visual outputs in PlantUML format. The solution targets two user groups: a beginner who needs quick orientation in the project, and an experienced programmer who requires a technically accurate and consistent summary. The result is a practical workflow and a prototype tool that speeds up onboarding and improves understanding of the architecture and key parts of the code.

Keywords: software comprehension, documentation, large language models, UML diagrams, PlantUML

Obsah

Úvod

Kapitola 1

Motivácia

Kapitola 2

Ciele práce

Kapitola 3

Východiská práce

3.1 Využitie veľkých jazykových modelov (LLM) pri podpore pochopenia softvéru

Veľké jazykové modely, ako napríklad ChatGPT, GitHub Copilot alebo Bard, predstavujú významný posun v oblasti asistencie pri sofvérovom inžinierstve. Vedia generovať zrozumiteľné, gramaticky správne a syntakticky korektné výstupy, majú potenciál výrazne podporiť proces porozumenia existujúcich softvérových systémov. [7]

V rámci onboardingu nových vývojárov alebo analýzy neznámeho systému môžu LLM plniť viaceré úlohy:

- **Generovanie prirodzenojazyčného popisu kódu:** LLM môžu automaticky vysvetliť význam tried, metód a modulov, čím znižujú potrebu manuálneho mentorovania.
- **Automatizované dopĺňanie komentárov a dokumentácie:** Vývojári môžu získať relevantnú dokumentáciu ku kódu bez nutnosti jej ručného písania, čo šetrí čas a zvyšuje čitateľnosť.
- **Transformácia kódu do vizuálnych modelov:** V spojení s nástrojmi ako PlantUML je možné automaticky generovať triedne alebo sekvenčné UML diagramy, ktoré pomáhajú pri pochopení štruktúry systému.
- **Poskytovanie spätnej väzby v reálnom čase:** LLM môžu asistovať pri vývoji prostredníctvom návrhov ďalších krokov alebo odporúčaní na opravu chýb v kóde.

Takéto použitie LLM umožňuje vývojárom pýtať sa otázky typu „Čo robí táto metóda?“, „Na čo slúži tento modul?“ alebo „Ako spolu súvisia jednotlivé časti systému?“. Model tak môže slúžiť ako mentor, ktorý zefektívni proces zaučenia nových členov tímu a zníži záťaž skúsených vývojárov. [7]

Na druhej strane, využitie LLM prináša aj isté riziká. Ako upozorňuje Ozkaya, výstupy modelov môžu byť síce gramaticky správne, ale nie vždy aj semanticky korektné. Z dôvodu náhodnej povahy generovania textu môže dôjsť k šíreniu nesprávnych alebo zavádzajúcich informácií. Rovnako je nutné dbať na kvalitu a zloženie trénovacích dát, ktoré môžu obsahovať chyby. Z týchto dôvodov je potrebné doplniť využitie LLM o kontrolné mechanizmy [7].

3.2 Podpora porozumenia softvérových modelov pomocou LLM

Porozumenie softvérových modelov, teda UML diagramov, patrí medzi kľúčové výzvy v softvérovom inžinierstve. Pre nových členov vývojového tímu býva často náročné zorientovať sa v komplexnej architektúre existujúcich systémov. Práve v tomto smere môžu veľké jazykové modely výrazne pomôcť.

Podľa štúdie [2], ktorá hodnotila schopnosť LLM asistovať pri modelovaní a pochopení softvérových artefaktov, môžu tieto modely slúžiť ako interaktívni asistenti, ktorí sú schopní:

- vysvetľovať časti UML modelu v ľudskej reči [2],
- odpovedať na otázky ohľadom štruktúry a správania systému [1, 2],
- generovať alebo dopĺňať časti modelu na základe textových požiadaviek alebo existujúcich diagramov [1, 2],
- transformovať medzi rôznymi reprezentáciami ako je napríklad ľudská reč a UML diagramy [1, 2].

3.3 Význam návrhu promptov pri využívaní LLM

Účinnosť veľkých jazykových modelov do veľkej miery závisí od kvality vstupov, ktoré im používateľ poskytne. Tento proces je veľmi dôležitý pri využívaní generatívnej umelej inteligencie v oblasti softvérového inžinierstva [4].

Správne navrhnutý prompt – teda textová požiadavka alebo otázka zadaná modelu – má zásadný vplyv na kvalitu, presnosť a relevantnosť výstupu. V kontexte podpory porozumenia softvéru to znamená, že:

- prompt musí jednoznačne špecifikovať, akú časť kódu alebo modelu chceme vysvetliť,

- požiadavka by mala byť formulovaná čo najkonkrétnejšie, aby sa minimalizovala generalizácia výsledku,

- je možné využívať few-shot príklady – ukážky očakávaného výstupu, ktoré modelu naznačujú formu alebo štruktúru odpovede [4].

Prompt engineering nie je len technika, ale aj forma dizajnu komunikácie medzi človekom a umelou inteligenciou [4]. V prípade vysvetľovania zdrojového kódu alebo generovania UML diagramov zo špecifikácie je kvalitný prompt nevyhnutný na to, aby model porozumel kontextu a požiadavke vývojára.

Pre potreby nástrojov zameraných na podporu nových vývojárov v tíme tak návrh vhodných promptov predstavuje dôležitú súčasť celého systému. Automatizácia niektorých častí promptov môže zefektívniť interakciu a zvýšiť presnosť výstupov generovaných LLM.

3.4 Generovanie dokumentácie ku kódu pomocou LLM

Jednou z významných aplikácií veľkých jazykových modelov v softvérovom inžinierstve je generovanie dokumentácie ku kódu. Tento proces môže výrazne zefektívniť onboarding nových vývojárov, znížiť záťaž skúsených kolegov a zlepšiť udržiavateľnosť softvéru.

Štúdia Guelmana a kol. [3] ukazuje, že LLM sú schopné generovať dokumentáciu ku kódu s kvalitou porovnateľnou s ľudskou, najmä v prípadoch, keď je kód dobre štruktúrovaný. Autori realizovali rozsiahlu analýzu a zistili, že:

- modely ako ChatGPT dosahujú vysokú presnosť pri popise účelu funkcií a premenných,

- generované komentáre bývajú často detailnejšie ako tie, ktoré píšu vývojári manuálne,

- vývojári hodnotili výstupy modelov ako užitočné a niekedy dokonca preferovali automaticky generovanú dokumentáciu pred ľudskou.

Zároveň však štúdia upozorňuje na limity: v prípadoch, kde je kód zložitejší alebo neobsahuje dostatočne výrečné pomenovania, kvalita generovaných komentárov klesá. Dôležitou súčasťou celého procesu je preto schopnosť modelu rozpoznať kontext a pomenovania symbolov. [3]

3.5 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) je moderný prístup, ktorý rozširuje možnosti LLM tým, že im pri generovaní odpovedí „dohľadáva“ dodatočné informácie z externých zdrojov. Klasické LLM pracujú len s dátami, na ktorých boli vytrénované, čo znamená, že nemusia vedieť poskytnúť úplne aktuálne informácie alebo presné odborné detaily mimo svojho trénovacieho datasetu. RAG tento problém rieši tak, že ešte pred samotným generovaním doplní model o kontext získaný z vyhľadávacieho mechanizmu. Tým znižuje chybovosť odpovedí a zlepšuje ich presnosť [5].

Základný princíp RAG je založený na dvoch krokoch: *retrieval* (vyhľadávanie) a *generation* (generovanie). Najskôr sa používateľská query prevedie do vektorového priestoru pomocou embeddingov. Takto reprezentovaná query sa porovná s obsahom uloženým vo vektorovej databáze. Tá môže obsahovať dokumenty, technické špecifikácie, interné materiály alebo iné textové zdroje. Systém vyberie najrelevantnejšie dokumenty a vloží ich ako dodatočný kontext do promptu. Ten je následne odoslaný jazykovému modelu. Model potom negeneruje odpoveď len z vlastnej „pamäte“, ale aj na základe práve načítaných informácií [5].

Významnou výhodou RAG je, že umožňuje pracovať s novými a špecifickými údajmi bez potreby model opätovne trénovať. Stačí zmeniť alebo doplniť vstupnú databázu a kvalita aj obsah výstupov sa tomu okamžite prispôsobia. Systém je teda veľmi flexibilný a vhodný najmä v prostrediach, kde sa informácie rýchlo menia – napríklad pri technickej dokumentácii, právnych textoch, zákazníckej podpore, podnikových dátach alebo vedeckých publikáciách [5].

RAG zároveň pomáha znižovať riziko tzv. halucinácií, pretože generovaný text sa opiera o reálne a overiteľné zdroje. Používateľ navyše môže zistiť, z ktorých dokumentov model čerpal, čo zvyšuje transparentnosť aj dôveryhodnosť celého systému. Tento prístup je do istej miery podobný akademickému citovaniu, kde možno každé tvrdenie spätne overiť v pôvodnom zdroji [5].

V širšom kontexte vývoja umelej inteligencie predstavuje RAG dôležitý posun v tom, ako sú generatívne modely prepojené s externým svetom. Model už nie je iba statický generátor textu na základe historických dát, ale stáva sa dynamickým, kontextovo citlivým systémom. Vďaka tomu dokáže poskytovať presné, odborné a zároveň aktuálne informácie v takmer reálnom čase [5].

3.6 LangChain a LangGraph

V tejto práci používame dve úzko príbuzné technológie z ekosystému vývoja aplikácií nad veľkými jazykovými modelmi – *LangChain* a *LangGraph*. Ich kombinácia nám umožňuje navrhovať a implementovať komplexné, viacagentové systémy, ktoré je možné

relatívne jednoducho rozširovať. Keďže jadrom nášho riešenia je orchestrácia viacerých agentov, prirodzene sa ponúka využitie oboch nástrojov.

LangChain predstavuje všeobecný framework na budovanie aplikácií nad LLM. Poskytuje sadu znovupoužiteľných komponentov (reťazce, nástroje, pamäť, retrievery, integrácie na modely a vektorové databázy) a umožňuje ich skladať do sekvenčných alebo mierne vetvených workflowov. [6] V praxi to znamená, že vývojár namiesto priameho volania modelu rieši vyššiu úroveň logiky, napríklad „vezmi vstup, doplň kontext z dokumentov, zavolaj model, následne odpoveď uprac a zaloguj“. Takýto prístup výrazne skracuje čas potrebný na prototypovanie a zároveň podporuje znovupoužiteľnosť častí riešenia.

LangGraph naopak stavia na základoch LangChainu a rozširuje ich o výslovne grafový model riadenia. Jednotlivé kroky workflowu sú reprezentované ako uzly (*nodes*) a prechody medzi nimi ako hrany (*edges*). Namiesto lineárneho reťazenia krokov tak vzniká orientovaný graf, v ktorom je možné modelovať vetvenie, slučky, opakované rozhodovanie a zdieľaný stav medzi agentmi. [6] V kontexte tejto práce je to kľúčové najmä pre viacagentový scenár, v ktorom nad jedným repozitárom kódu postupne pracuje „generátor textu“ a „reviewer“.

Porovnanie týchto prístupov ukazuje, že LangChain sa typicky odporúča pre jednoduchšie, prevažne lineárne alebo len mierne vetvené workflowy. Tam stačí rýchlo poskladať sekvenciu volaní modelu a pomocných nástrojov. LangGraph je naopak vhodnejší pre robustnejšie prostredia, kde je potrebné udržiavať stav medzi krokmi, koordinovať viacerých agentov a umožniť im opakovane sa rozhodovať na základe priebežných výsledkov. To nám pomáha k tomu, aby nástroj vedel iteratívne analyzovať zdrojový kód, rozhodovať o ďalších krokoch (napr. či zapojiť kontrolného agenta) a v prípade potreby sa vrátiť k predchádzajúcim fázam spracovania. [6]

Napokon je vhodné zdôrazniť, že LangChain a LangGraph nevystupujú izolovane, ale sú súčasťou širšieho ekosystému nástrojov pre budovanie agentových LLM aplikácií. Často sa používajú spolu so sprievodnými nástrojmi, ako sú LangSmith a LangFlow. Vzniká tak technologické prostredie, ktoré umožňuje nielen implementáciu samotného nástroja, ale aj jeho testovanie, ladenie a potenciálne nasadenie v produkčnom prostredí. [6]

Kapitola 4

Výskum

4.1 Popis vyvinutého nástroja

Náš softvér je určený na automatizovanú analýzu, dokumentovanie a vizualizáciu zdrojového kódu Python projektov. Návrh vychádza z dvoch kľúčových prípadov použitia, ktoré odrážajú rozdielne potreby a očakávania užívateľov pri práci so zdrojovým kódom:

- **Newcomer (začiatočník vo firme)** – používateľ, ktorý sa prvýkrát stretáva s existujúcim kódom a potrebuje rýchlo pochopiť architektúru projektu, účel jednotlivých modulov a význam najdôležitejších funkcií. Pre takéhoto užívateľa nástroj poskytuje ucelenú technickú dokumentáciu, vizuálne UML diagramy, sumarizáciu tried a funkcií, a taktiež príručky ku softvéru.
- **Senior programátor** – skúsený vývojár, ktorý vyžaduje efektívne nástroje na reverzné inžinierstvo, vyhľadávanie kritických častí systému, či analýzu architektúry a závislostí. Pre túto skupinu poskytuje systém metriky z AST analýzy, identifikáciu najdôležitejších funkcií na základe vlastného indexu dôležitosti, detailné UML diagramy celého systému a nástroje umožňujúce orientáciu v rozsiahlych projektových štruktúrach. Takisto skúsenému vývojaťovi systém poskytne rýchle spravenie dokumentácie pre zákazníka.

Vďaka týmto dvom perspektívam je nástroj flexibilný a univerzálny: nových členov tímu rýchlo uvedie do problematiky, zatiaľ čo skúseným vývojárom poskytne podklady pre technické rozhodovanie a hlbšiu analýzu projektu.

4.1.1 Hlavná charakteristika

Nástroj predstavuje modulárny systém, v ktorom jednotlivé komponenty (ďalej *agenti*) vykonávajú samostatné kroky spracovania zdrojového kódu. Každý agent je zodpo-

vedný za konkrétnu úlohu od analýzy a extrakcie štruktúry programu cez generovanie technickej dokumentácie až po tvorbu UML diagramov pomocou LLM.

Hlavnou myšlienkou vývoja bola potreba automatizovať a zjednodušiť proces analýzy existujúceho kódu, ako aj poskytnúť konzistentné výstupy vhodné pre technickú dokumentáciu a prezentáciu architektúry systémov.

4.1.2 Získavanie zdrojového kódu

Prvým krokom spracovania je automatizované stiahnutie repozitára zo služby GitHub. O túto časť systému sa stará modul `RepositoryReader`, ktorý zabezpečuje:

- korektné klonovanie repozitára pomocou knižnice `GitPython`,
- vytvorenie dočasnej pracovnej kópie v lokálnom adresári,
- vyhľadanie všetkých súborov typu `.py`,
- načítanie ich obsahu do pamäťovej štruktúry pre ďalšie spracovanie.

Týmto spôsobom je možné analyzovať aj väčšie projekty bez toho, aby bolo nutné zadefinovať presnú štruktúru adresárov alebo manuálne prechádzať jednotlivé súbory.

4.1.3 Statická analýza kódu

Po získaní zdrojových súborov nasleduje fáza statickej analýzy založenej na abstraktnom syntaktickom strome (AST). Analýza zahŕňa:

- extrakciu definícií funkcií a tried,
- identifikáciu atribútov a metód v rámci tried,
- analýzu importov a závislostí medzi modulmi,
- získavanie metrických údajov, ako sú počet riadkov, cyklomatická zložitosť či počet volaní.

4.1.4 LLM agenty pre dokumentáciu a vizualizáciu

Kľúčovým prvkom nášeho softvéru je integrácia veľkého jazykového modelu prostredníctvom platformy Ollama a knižnice LangChain/Langgraph. Výsledkom je :

- generovať technickú dokumentáciu pre jednotlivé súbory,
- sumarizovať funkcionalitu tried a modulov,

- vytvárať inštalačné príručky s detailným návodom na použitie,
- tvoriť UML diagramy (komponentový diagram a diagram tried) na základe analýzy zdrojového kódu.

LLM agenti nepracujú izolovane – mnohé výstupy sú vzájomne prepojené. Napríklad class diagram pre celý systém je kombináciou *deterministickej* analýzy (identifikácia atribútov a metód v AST) a *generatívnej* analýzy (odvodzovanie vzťahov medzi triedami prostredníctvom LLM).

4.1.5 Hlavná riadiaca logika

Celý proces orchestruje skript `main.py`, ktorý poskytuje používateľovi interaktívne menu. Na základe výberu používateľa je spustená príslušná úloha:

- generovanie dokumentácie k zdrojovým súborom,
- zobrazenie desiatich najdôležitejších funkcií podľa vlastného indexu dôležitosti,
- vytvorenie komponentového UML diagramu,
- vytvorenie diagramu tried pre celý projekt,
- generovanie komplexnej inštalačnej príručky.

Takýto prístup umožňuje flexibilné rozširovanie funkcionality o ďalších agentov a spracovateľské kroky bez zásahu do existujúceho API.

4.2 Funkcionálne požiadavky

Funkcionálne požiadavky znamenajú očakávané správanie systému , ktoré sú nevyhnutné pre automatizovanú analýzu, dokumentáciu a vizualizáciu Python projektov. Požiadavky sú formulované s ohľadom na dve hlavné skupiny používateľov: nových členov projektov, ktorí potrebujú rýchlo porozumieť existujúcemu kódu, a seniorných programátorov, ktorí očakávajú podrobnú technickú analýzu a podporu pri práci so zložitejšími architektúrami.

4.2.1 Požiadavky na prácu s repozitárom

Systém musí umožniť načítanie verejného zo stránky GitHub. Používateľ zadá iba URL a nástroj vykoná:

- automatické stiahnutie repozitára do dočasného adresára,

- identifikáciu všetkých súborov typu `.py`,
- načítanie ich obsahu pre ďalšie spracovanie,
- bezpečné odstránenie dočasného adresára po ukončení analýzy.

Tieto funkcie umožňujú používateľovi pracovať s projektmi bez predchádzajúcej lokálnej prípravy a zaručujú konzistentný proces analýzy.

4.2.2 Požiadavky na statickú analýzu zdrojového kódu

Po načítaní projektu musí systém vykonať statickú analýzu pomocou abstraktného syntaktického stromu (AST). Systém musí byť schopný:

- identifikovať všetky triedy a ich atribúty,
- identifikovať metódy a funkcie definované v súboroch,
- extrahovať importy a vzťahy medzi modulmi,
- získavať rôzne metrické údaje, ako počet riadkov kódu, počet volaní funkcií alebo cyklomatickú zložitosť.

Tieto údaje tvoria základ pre ďalšie formy analýzy a vizualizácie.

4.2.3 Požiadavky na vyhodnocovanie dôležitosti funkcií

Systém musí vypočítať index dôležitosti pre každú funkciu alebo metódu v projekte. Tento index musí zohľadňovať výsledky statickej analýzy, čo umožňuje identifikovať najkritickejšie časti aplikácie. Systém musí vedieť:

- vypočítať dôležitosť na základe viacerých metrík,
- zoradiť funkcie podľa významu,
- zobraziť používateľovi najdôležitejšie funkcie v prehľadnej podobe.

Táto funkcionalita je dôležitá najmä pre seniorných programátorov, ktorí potrebujú rýchlo identifikovať kľúčové prvky architektúry.

4.2.4 Požiadavky na generovanie dokumentácie pomocou LLM

Náš systém musí priamo zo zdrojového kódu generovať technickú dokumentáciu. Generovanie sa deje kvôli LLM agentovi. Systém musí:

- vytvoriť popisy pre každý analyzovaný súbor,
- vysvetliť účel súborov, funkcií a tried,
- vytvárať konzistentnú a technicky správnu dokumentáciu,
- umožniť jej revíziu pomocou sekundárneho kontrolného agenta,
- uložiť výsledky do jedného výstupného dokumentu.

Tento proces poskytuje novým členom tímu rýchly prehľad o logike projektu.

4.2.5 Požiadavky na generovanie UML diagramov

Požiadavkou nášeho softvéru je umožňovať automatickú vizualizáciu architektúry projektu pomocou UML diagramov. Musí poskytovať:

- komponentový diagram založený na vzťahoch medzi modulmi,
- diagram tried pre celý projekt s atribútmi a metódami,
- generovanie vzťahov medzi triedami pomocou LLM,
- export diagramov do formátu kompatibilného s PlantUML.

Diagramy sú dôležitým nástrojom pri orientácii v architektúre rozsiahlych projektov.

4.2.6 Požiadavky na generovanie inštalačnej príručky

Systém musí byť schopný vytvoriť kompletnú inštalačnú príručku pre daný projekt. Príručka musí obsahovať:

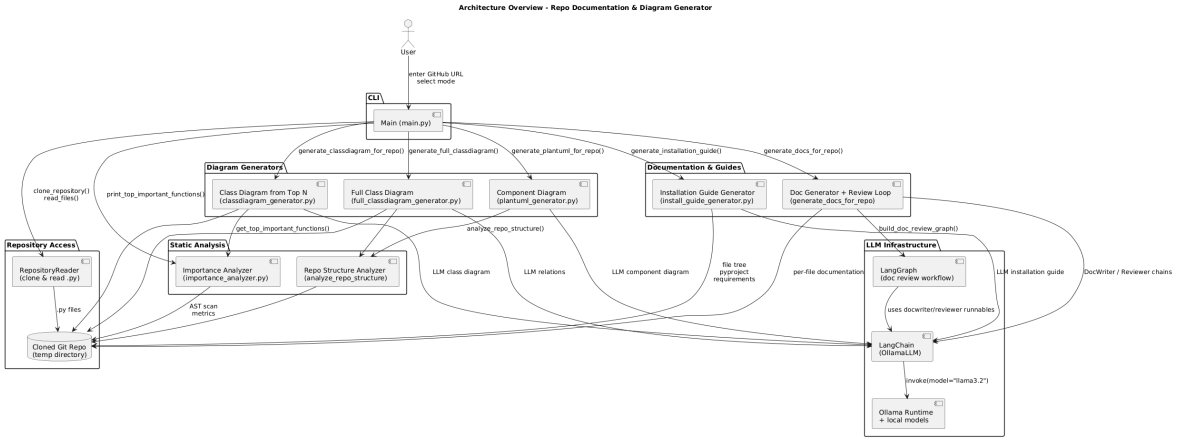
- systémové požiadavky,
- postup klonovania repozitára,
- konfiguráciu virtuálneho prostredia,
- inštaláciu závislostí,
- konfiguráciu LLM modelov (ak sú potrebné),

- návod na spustenie,
- riešenie najčastejších problémov.

Tento výstup slúži ako rýchly manuál pre nových používateľov projektu.

4.3 Návrh riešenia a architektúra

Na základe definovaných cieľov a požiadaviek som náš systém navrhol ako modulárnu aplikáciu. Každý modul má jasne definované zodpovednosti a je dostatočne oddelený od ostatných modulov. Architektúra tohoto systému je znázornená v obrázku 4.1.



Obr. 4.1: Architektonický diagram systému

4.4 Použité technológie

Táto kapitola opisuje technologický základ vyvinutého nástroja. Nasledujúci zoznam sumarizuje všetky hlavné technológie použité pri vývoji nášho softvéru:

- Python 3 – primárny programovací jazyk projektu.
- GitPython – knižnica na automatizovanú prácu s Git repozitármi.
- AST (Abstract Syntax Tree) modul – nástroj na statickú analýzu Python kódu.
- LangChain – framework pre orchestráciu LLM agentov.
- Ollama – lokálna platforma pre spúšťanie veľkých jazykových modelov.
- Llama 3.2 (resp. iné LLM modely) – jazykový model používaný na generovanie dokumentácie a UML diagramov.

- PlantUML – štandardný formát pre textovú definíciu UML diagramov.
- Markdown – formát pre výstupné dokumenty (napr. inštalačná príručka).
- GitHub – zdroj projektových repozitárov.

V nasledujúcej časti sú jednotlivé technológie popísané detailnejšie, avšak bez formálneho členenia na podsekcie, aby bol text plynulejší a vhodný do naratívnej časti práce.

Python 3 tvorí základ celého projektu a predstavuje hlavný programovací jazyk nástroja. Využíva sa na prácu s dátovými štruktúrami, spracovanie textu a zdrojových súborov, statickú analýzu prostredníctvom modulu AST, implementáciu jednotlivých agentov aj komunikáciu s jazykovými modelmi. Python sme zvolili pre svoju flexibilitu a aaj pre veľký výber knižníc. Tak isto pre integráciu ollama sme potrebovali zvoliť python.

GitPython je knižnica umožňujúca komunikáciu s Git repozitármi priamo z prostredia Pythonu. V projekte zabezpečuje klonovanie vstupných repozitárov na základe URL, prácu s dočasnými adresármi a ošetrenie chýb pri klonovaní napríklad neplatná URL, neexistujúci repozitár alebo konfliktný stav cieľového priečinka. Vďaka tomu môže nástroj bez manuálneho zásahu používateľa načítať zdrojový kód ľubovoľného projektu hostovaného na GitHube či v inom Git serveri.

AST modul (`ast`) je súčasťou štandardnej knižnice Pythonu a umožňuje previesť zdrojový kód do podoby abstraktného syntaktického stromu. Projekt tento modul využíva na identifikáciu tried, metód a funkcií, extrakciu atribútov tried, analýzu importov a výpočet metrických údajov, ako je počet volaní funkcií či približná cyklomatická zložitosť. Slúži ako input pre generovanie dokumentácie aj UML diagramov a zároveň minimalizuje riziko halucinácií .

LangChain predstavuje framework pre orchestráciu veľkých jazykových modelov a ich integráciu do komplexnejších pipeline. V rámci nástroja umožňuje vytvárať tzv. LLM agentov, ktorí kombinujú prompt, konkrétny model a doplnkovú logiku. Týmto spôsobom vznikajú samostatní agenti zodpovední za tvorbu dokumentácie, za revíziu (review) vygenerovaných textov, za generovanie UML diagramov či za tvorbu inštalačnej príručky. LangChain poskytuje jednotné rozhranie pre prácu s modelmi a uľahčuje kompozíciu viacerých krokov spracovania do jedného konzistentného pracovného toku.

Ollama je lokálna platforma, ktorá umožňuje spúšťanie veľkých jazykových modelov priamo na počítači používateľa. V našom nástroji slúži ako infraštruktúrna vrstva pre beh LLM, takže všetky dotazy smerujú do lokálneho runtime prostredia namiesto do cloudových služieb. To umožňuje spracovanie zdrojového kódu bez jeho odosielania na externé servery. Týmto sa posilňuje bezpečnosť a súlad s internými bezpečnost-

nými politikami. Zároveň to znižuje prevádzkové náklady pri opakovanom používaní nástroja.

Model Llama 3.2 slúži ako primárny jazykový model. V projekte je nasadený na generovanie technickej dokumentácie zdrojového kódu, sumarizáciu architektúry modulov, návrh PlantUML diagramov tried a komponentov a tvorbu inštalačných príručiek. Tento model bol zvolený pre dobrý pomer medzi kvalitou výstupom, výkonom a možnosťou lokálneho spúšťania cez Ollamu.

PlantUML je textový formát a nástroj určený na tvorbu UML diagramov pomocou jednoduchého textového zápisu. V nástroji sa využíva na generovanie komponentových diagramov na základe importných vzťahov medzi modulmi a diagramov tried, ktoré vznikajú kombináciou AST analýzy a LLM interpretácie. Textová povaha PlantUML umožňuje jednoduché verzovanie diagramov v Gite, ich integráciu do dokumentácie a automatizovanú aktualizáciu pri opakovanej analýze projektu.

Markdown je značkovací jazyk, ktorý sa v projekte používa najmä na generovanie inštalačných príručiek a ďalších textových výstupov. Jeho výhodou je dobrá čitateľnosť v surovej textovej podobe, široká podpora na platforme GitHub a možnosť jednoduchého exportu do HTML či PDF. Vygenerovaná dokumentácia tak môže byť priamo vložená do repozitára ako README alebo doplnkový dokument bez potreby ďalšieho spracovania.

GitHub plní v nástroji úlohu základného zdroja analyzovaných projektov. GitHub tak predstavuje prirodzené prostredie pre získavanie vstupného kódu moderných softvérových projektov, na ktorých je demonštrovaná funkcionalita nástroja.

4.5 Generovanie klasickej dokumentácie

Jednou z najdôležitejších funkcionalít nášho nástroja je schopnosť automaticky generovať dokumentáciu zdrojového kódu. Táto dokumentácia na presné a technicky korektné zhrnutie architektúry. Celý proces je realizovaný kombináciou statickej analýzy kódu a kooperáciou dvoch nezávislých LLM agentov: **dokumentačného agenta** a **reviewer agenta**.

4.5.1 Statická analýza ako vstup pre dokumentáciu

Pred samotným generovaním dokumentácie prebieha statická analýza zdrojového kódu. Jej cieľom je poskytnúť LLM modelu štruktúrované a presné informácie o obsahu súborov. Pre každý Python súbor sa identifikujú:

- názvy a umiestnenie funkcií a tried,
- atribúty a metódy tried,

- importované moduly,
- základná štruktúra súboru a jeho zodpovednosť.

Výstup statickej analýzy výrazne znižuje pravdepodobnosť nesprávnych interpretácií zo strany modelu a zároveň zaručuje konzistentnosť vygenerovaných textov.

4.5.2 Dokumentačný agent

Dokumentačný agent predstavuje primárny LLM modul zodpovedný za vytvorenie prvej verzie technickej dokumentácie. Tento agent dostáva ako vstup:

- obsah zdrojového súboru,
- výsledky AST analýzy,
- doplnkové metadáta (cesta súboru, modulová štruktúra, ap.).

Jeho úlohou je:

- vysvetliť účel súboru a jeho komponentov,
- popísať triedy a funkcie v kontexte celého systému,
- zhrnúť dôležité mechanizmy fungovania modulu,
- generovať konzistentný a technický text vhodný do dokumentácie.

Agent pracuje deterministicky, s dôrazom na technickú presnosť a štruktúrovaný výstup.

4.5.3 Reviewer agent

Po vytvorení pôvodnej verzie dokumentácie prechádza výstup ďalším stupňom – kontrolou pomocou **reviewer agenta**. Tento agent má charakter nezávislého hodnotiteľa, ktorý vykonáva:

- kontrolu technickej presnosti,
- hľadanie nekonzistencií medzi kódom a dokumentáciou,
- návrhy na doplnenie nepopísaných častí,
- kontrolu správnosti terminológie,
- zlepšovanie čitateľnosti a odbornosti textu.

```python
def review_node(state: DocReviewState) -> Dict[str, Any]:
    review = reviewer.invoke(
        {
            "path": state["path"],
            "code": state["code"],
            "doc": state.get("documentation", ""),
        }
    )
    review_text = str(review)
    decision, feedback = parse_decision(review_text)

    updates: Dict[str, Any] = {
        "review_text": review_text,
        "decision": decision,
        "feedback": feedback,
    }

    # ak treba pokračovať, pripravíme feedback pre ďalší write krok
    if decision != "APPROVE":
        updates["feedback_for_writer"] = feedback or (
            "Please fix coverage, correctness, clarity, and parameter notes as needed."
        )
        updates["round"] = int(state.get("round", 0)) + 1

    return updates
```

Obr. 4.2: Kód pre ďalšie zapojenie DocMaker agenta

Po spracovaní údajov sa rozhoduje v systéme ako je demonštrované v kóde 4.2. Reviewer agent dostáva ako vstup:

- pôvodnú dokumentáciu od hlavného agenta,
- plný obsah zdrojového súboru,
- výsledky statickej analýzy.

Týmto spôsobom kontrolná fáza nie je založená na heuristickom porovnávaní textu, ale na hlbokom porozumení samotnému kódu. Výsledkom je oveľa kvalitnejšia dokumentácia, ktorá odstraňuje nepresnosti alebo príliš všeobecné formulácie.

4.5.4 Pipeline generovania dokumentácie

Celý proces generovania dokumentácie prebieha v nasledovných krokoch:

1. Používateľ zvolí možnosť „Generovať dokumentáciu“.
2. Systém automaticky naklonuje cieľový GitHub repozitár.
3. Statická analýza pomocou AST identifikuje štruktúru kódu.
4. Dokumentačný agent vytvorí prvý návrh dokumentácie.
5. Reviewer agent skontroluje a prípadne upraví text.
6. Výsledný dokument sa uloží ako jeden súhrnný výstup.

Proces podporuje tvorbu dokumentácie pre:

- všetky súbory v projekte,
- jediný súbor podľa výberu používateľa.

4.5.5 Výhody zapojenia dvoch agentov

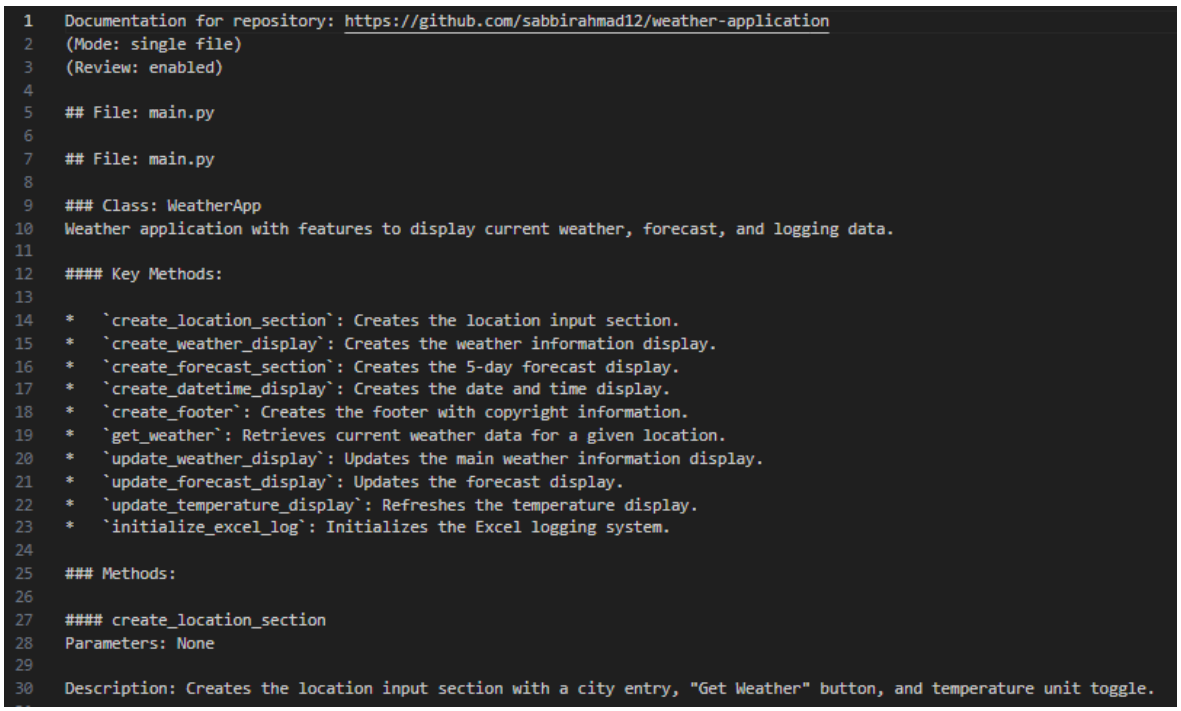
Použitie dvojice kooperujúcich agentov prináša významné výhody:

- **Vyššia presnosť** – každý text je automaticky skontrolovaný.
- **Eliminácia halucinácií** – reviewer porovnáva dokumentáciu s reálnym kódom.
- **Kvalitnejší text** – text prechádza dvoma fázami spracovania.
- **Lepšia konzistentnosť** – systém udržiava rovnaký tón a štruktúru naprieč projektom.

4.5.6 Zhrnutie

Výslednú podobu generovanej dokumentácie ilustruje obrázok 4.3. Na ukážke je zobrazený výstup pre jeden z analyzovaných modulov, kde nástroj automaticky vytvoril súvislý technický text. Dokumentácia obsahuje:

- stručný úvodný popis modulu a jeho zodpovednosti v rámci systému,
- prehľad hlavných tried a funkcií vrátane ich účelu,
- vysvetlenie väzieb na iné časti projektu (napríklad na pomocné utility alebo LLM agenta),
- zhrnutie typických scenárov použitia danej časti kódu.



```
Documentation for repository: https://github.com/sabbirahmad12/weather-application
(Mode: single file)
(Review: enabled)

## File: main.py

## File: main.py

### Class: WeatherApp
Weather application with features to display current weather, forecast, and logging data.

#### Key Methods:

*   `create_location_section`: Creates the location input section.
*   `create_weather_display`: Creates the weather information display.
*   `create_forecast_section`: Creates the 5-day forecast display.
*   `create_datetime_display`: Creates the date and time display.
*   `create_footer`: Creates the footer with copyright information.
*   `get_weather`: Retrieves current weather data for a given location.
*   `update_weather_display`: Updates the main weather information display.
*   `update_forecast_display`: Updates the forecast display.
*   `update_temperature_display`: Refreshes the temperature display.
*   `initialize_excel_log`: Initializes the Excel logging system.

### Methods:

#### create_location_section
Parameters: None

Description: Creates the location input section with a city entry, "Get Weather" button, and temperature unit toggle.
```

Obr. 4.3: Ukážka výslednej generovanej dokumentácie pre vybraný modul projektu.

Kapitola 5

Výsledky a diskusia

Záver

Literatúra

[1] F. J. Alcaide, J. R. Romero, and A. Ramírez. Can explainable artificial intelligence support software modelers in model comprehension? *Software and Systems Modeling*, 2025.

[2] J. Cámara, J. Troya, L. Burgueño, et al. On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, 22:781–793, 2023.

[3] Ian Guelman, Arthur Gregório Leal, Laerte Xavier, and Marco Tulio Valente. Using large language models to document code: A first quantitative and qualitative assessment, 2024.

[4] IBM. What is prompt engineering?, 2025. Accessed: 2025-05-12.

[5] Rick Merritt. What is retrieval-augmented generation, aka rag?, 2025. Accessed: YYYY-MM-DD.

[6] Ryan Ong. Langchain vs langgraph vs langsmith vs langflow: Key differences explained, September 2025. Accessed: 2025-12-11.

[7] Ipek Ozkaya. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software*, 40(3):4–8, 2023.

Príloha A: