Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie UI v procese výuky programovania

Diplomová práca

2026
Bc. Samuel Ješík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Využitie UI v procese výuky programovania

Diplomová práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Andrej Blaho, PhD.

Bratislava, 2026
Bc. Samuel Ješík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Samuel Ješík
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: Využitie UI v procese výuky programovania
Using AI in the process of teaching programming

Anotácia: V súčasnosti je už temer isté, že UI asistenti (chatGPT, Copilot a pod.) sa budú spolupodieľať na písaní programov. Študenti ich používajú a je temer nemožné im to zakazovať. Naopak, bolo by žiadúce, aby tieto pomôcky využívali správne, aby im pomáhali, asistovali im, ale nevykonávali ich prácu v škole za nich. Je nevyhnutné, aby sa študenti naučili sami programovať, pretože inak nebudú schopní porozumieť programom, kontrolovať ich správnosť, refaktorovať ich, písať testy, ako aj rozširovať ich a pristupovať k nim ako k zdedenému kódu.

Cieľ: 1. Zmapovať UI asistentov, vyskúšať rôznych na rovnakých úlohách a porovnať ich schopnosti, kvalitu a použiteľnosť vo výuke. Príp. analyzovať, čo študenti u nás používajú.
2. Vytvoriť praktické, výukové web prostredie, v ktorom by študenti mohli pracovať vo viacerých jazykoch (Python, C++, Java) spolu s UI na rôznych typoch úloh (sami, alebo s pomocou UI): písanie kódu, tvorba testov, refaktorizácia, rozširovanie zdedeného kódu a pod.
3. Vytvoriť katalóg vhodných úloh pre študentov, na ktorých by si prakticky vyskúšali programovanie s UI, napr. odhaľovanie chýb v kóde vytvorenom UI, využitie UI na písanie testov na vlastný chybový kód, refaktorizácia, modifikovanie a rozširovanie kódu vytvoreného UI a pod.

Vedúci: RNDr. Andrej Blaho, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
prípustná pre vlastnú VŠ

Dátum zadania: 11.12.2024

Dátum schválenia: 11.12.2024

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

.....
študent

.....
vedúci práce

Poďakovanie

XXXXX.

Abstrakt

Táto diplomová práca sa zaoberá využitím UI asistentov (ako ChatGPT) v procese výučby programovania na Fakulte matematiky, fyziky a informatiky UK. Cieľom práce je preskúmať, aký vplyv majú tieto nástroje na správanie študentov, kvalitu kódu a vnímanie AI asistencie pri písaní programov. Práca tiež bude skúmať, aké nástroje sú študentmi využívané na fakulte a porovnáva schopnosti rôznych UI asistentov na identických programovacích úlohách. Spolu s týmito zisteniami bude vytvorené praktické webové prostredie, ktoré umožní študentom riešiť úlohy sami alebo s podporou AI asistenta, pričom sa zameria na písanie kódu, tvorbu testov, refaktorizáciu a rozširovanie zdedeného kódu. Výsledky práce môžu pomôcť pri optimalizácii vyučovacieho procesu a pri správnej implementácii AI nástrojov do vzdelávacieho systému.

Kľúčové slová: UI asistenti, ChatGPT, výučba programovania, umelá inteligencia, webové prostredie, refaktorizácia kódu

Abstract

This thesis focuses on the use of UI assistants (such as ChatGPT) in the process of teaching programming at the Faculty of Mathematics, Physics and Informatics, Comenius University. The main goal of the work is to examine the impact these tools have on students' behavior, code quality, and perceptions of AI assistance in programming. The thesis will also investigates which tools are currently used by students at the faculty and compares the capabilities of different UI assistants on identical programming tasks. Together with these findings, a practical web environment will be created, allowing students to solve tasks independently or with the support of an AI assistant, focusing on code writing, test creation, refactoring, and expanding legacy code. The results of the thesis may help optimize the teaching process and the proper implementation of AI tools in the educational system.

Keywords: UI assistants, ChatGPT, , programming education, artificial intelligence, web environment, code refactoring

Obsah

Úvod **1**

1 Východiská práce **3**

1.1 Transformácia výučby programovania v ére AI 3

1.1.1 Už nejde len o syntax, ale o architektúru 3

1.1.2 Menej zbytočnej záťaže pre mozog 4

1.1.3 Pozor na ilúziu, že všetko viem 4

1.1.4 Nové kľúčové zručnosti 4

1.2 Prehľad a taxonómia AI asistentov pre programovanie 5

1.2.1 Konverzačné modely (Chatbots) 5

1.2.2 Asistenti priamo v editore (AI-Powered IDEs) 7

1.2.3 Lokálne modely (Open Weights) 8

1.3 Riziká a limitácie integrácie AI vo výučbe 9

1.3.1 Technické limity: Halucinácie a spoľahlivosť 9

1.3.2 Pedagogické riziká: Kognitívna atrofia a lenivosť 10

1.4 Nové didaktické scenáre 10

1.4.1 Hľadanie chýb po AI (Reverse Debugging) 10

1.4.2 Písanie testov pre čiernu skrinku (Black-box Testing) 10

1.4.3 Upratovanie a refaktorizácia (Refactoring) 11

1.5 Analýza existujúcich platforiem a potreba vlastného riešenia 11

1.5.1 Prečo nestačia súťažné platformy 11

1.5.2 Prečo potrebujeme vlastné riešenie 12

1.6 Etické a bezpečnostné aspekty AI vo vývoji softvéru 12

1.6.1 Hackovanie modelu (Prompt Injection) 12

1.6.2 Ochrana duševného vlastníctva a únik dát 13

1.7 Pedagogické ukotvenie: Bloomova taxonómia v ére AI 13

2 Analýza a návrh riešenia **15**

2.1 Čo od toho vlastne chceme (Požiadavky na systém) 15

2.1.1 Funkcionálne požiadavky 15

2.1.2 Nefunkcionálne požiadavky 16

2.2 Architektúra systému 16
2.2.1 Ako to vyzerá zhora (High-level prehľad) 16
2.2.2 Použité technológie 16
2.3 Návrh dátového modelu 18
2.3.1 ER diagram databázy 18
2.3.2 Polymorfizmus pri podpore jazykov 18
2.4 Návrh bezpečného spúšťania kódu (Sandbox) 18
2.4.1 Izolácia pomocou kontajnerov 18
2.4.2 Riešenie kompilácie pre C++ a Javu 18
2.5 Návrh integrácie AI 18
2.5.1 Komunikácia s Ollama API 18
2.5.2 Automatizované generovanie zadaní 18
2.5.3 Prompt Engineering 19

3 Implementácia **21**
3.1 Príprava vývojového prostredia 21
3.2 Backend a aplikačná logika 21
3.2.1 Správa používateľov a autentifikácia 21
3.2.2 Manažment úloh a nahrávanie riešení 21
3.3 Realizácia bezpečného spúšťania (Docker Orchestration) 21
3.3.1 Tvorba Docker image 21
3.3.2 Implementácia funkcie pre spustenie kódu 21
3.3.3 Spracovanie výstupov a chýb 21
3.4 Implementácia AI modulu 21
3.4.1 Generovanie zadaní a parsovanie výstupov 21
3.4.2 Chatovací asistent 21
3.5 Používateľské rozhranie (Frontend) 21
3.5.1 Editor kódu 21
3.5.2 Vizualizácia spätnej väzby a testov 21

4 Výskum **23**
4.1 Metodika výskumu 23
4.1.1 Na čo hľadáme odpovede (Výskumné otázky) 23
4.2 Kde a s kým robíme výskum 23
4.2.1 Zber dát 23
4.2.2 Vzorka respondentov 24
4.3 Ako vyzerá dotazník 24
4.4 Čo s nazbieranými dátami 25
4.4.1 Triedenie a analýza 25

4.4.2 Čítanie medzi riadkami (Kvalitatívna analýza) 25

5 Diskusia **27**

5.1 Interpretácia dosiahnutých výsledkov 27

5.1.1 Spoľahlivosť lokálnych LLM modelov 27

5.1.2 Kvalita pedagogickej spätnej väzby 27

5.2 Technické výzvy a riešenia 27

5.2.1 Špecifiká kompilácie v kontajnerizovanom prostredí 27

5.2.2 Latencia a hardvérové nároky 27

5.3 Pedagogické a etické aspekty 27

5.3.1 Riziko nadmernej závislosti na AI 27

5.3.2 Ochrana súkromia a dát 27

6 Zhrnutie hlavných prínosov práce **29**

6.1 Teoretický prínos 29

6.1.1 Analýza využívania AI nástrojov študentmi 29

6.2 Praktický prínos 29

6.2.1 Webové prostredie s podporou viacerých jazykov (Polyglot) . . . 29

6.2.2 Robustný mechanizmus komunikácie s AI 29

6.2.3 Katalóg refaktorizačných zadaní 29

6.3 Možnosti ďalšieho rozvoja 29

6.3.1 Rozšírenie o ďalšie jazyky a frameworky 29

6.3.2 Jemné doladenie modelu (Fine-tuning) 29

Záver **31**

Príloha A **35**

Zoznam obrázkov

1.1 Základom všetkých moderných asistentov je architektúra Transformer, ktorá využíva mechanizmus pozornosti (Attention) na spracovanie kontextu kódu. Prevzaté z [3]. 6
1.2 Revidovaná Bloomova taxonómia 14

2.1 Detailný pohľad na architektúru systému z perspektívy študenta. Diagram ukazuje, ako kód putuje do Docker kontajnera (Sandbox) a ako prebieha komunikácia s AI. 17
2.2 Architektúra modulu pre automatizované generovanie učebných materiálov. Učiteľ zadáva tému a ‘GenService‘ spolu s ‘Parserom‘ zabezpečia vytvorenie validnej úlohy. 18

Úvod

xxxxxxxxxxxxxxxx

Kapitola 1

Východiská práce

Príchod generatívnej umelej inteligencie (GenAI) a veľkých jazykových modelov (LLM) je pre vývoj softvéru obrovským míľnikom. Dá sa povedať, že ide o najväčšiu zmenu od čias, kedy nastúpilo objektovo-orientované programovanie.

V tejto kapitole sa pozrieme na to, ako nástroje typu ChatGPT alebo GitHub Copilot menia každodennú prácu programátorov. Rozoberieme tiež, prečo je potrebné zmeniť spôsob výučby – už to nie je len o samotnom písaní kódu, ale skôr o jeho kontrole a spájaní do funkčného celku.

1.1 Transformácia výučby programovania v ére AI

Dlhú dobu platilo, že informatika sa učí „od podlahy“ (bottom-up approach). Študenti sa trápili s každým riadkom kódu a na začiatku riešili hlavne to, prečo im program padá na chýbajúcej bodkočiarke alebo zlej zátvorke, namiesto toho, aby riešili skutočný problém. Tento starý systém síce roky fungoval, ale pri dnešných moderných nástrojoch už naráža na svoje limity.

1.1.1 Už nejde len o syntax, ale o architektúru

Dnes sa to celé otáča. Výskumy, ako napríklad štúdia od Suna a kol. [1], ukazujú, že vďaka AI asistentom už študent nie je len ten, kto „búcha kód“ (writer), ale skôr ten, kto ho navrhuje a kontroluje (architect a reviewer).

Kedysi sme museli vedieť naspamäť napísať QuickSort v C++, aby sme prešli skúškou. Dnes nám to ChatGPT alebo Llama 3 vypľuje za sekundu. Preto sa mení to, čo je dôležité:

1. **Chápať, nie bifľovať:** Už nepotrebujem vedieť kód naspamäť, ale musím vedieť, *prečo* použiť práve tento algoritmus a *či* je ten vygenerovaný kód bezpečný.

2. **Viac čítať ako písať:** Paradoxne, vďaka AI musíme viac čítať cudzí kód. Schopnosť rýchlo sa zorientovať v tom, čo stroj napísal (Code Comprehension), je dnes dôležitejšia než rýchle prsty na klávesnici.

1.1.2 Menej zbytočnej záťaže pre mozog

Z pohľadu psychológie je programovanie pre začiatočníka strašná „makačka“ na mozog (Cognitive Load). Študent totiž rieši dve veci naraz:

- **Samotný problém:** Čo chcem vlastne spraviť? (napr. vypočítať priemer).
- **Syntax:** Ako to napísať v Jave tak, aby mi to nevyhodilo error?

Raihan a kol. [2] tvrdia, že AI dokáže odstrániť tú druhú, otravnú časť – bariéru syntaxe. Študentovi sa tak uvoľní kapacita v hlave a môže sa sústrediť na podstatu problému a návrh riešenia. AI tu funguje ako taká „kognitívna protéza“, vďaka ktorej aj začiatočníci zvládnu zložité veci, ku ktorým by sa inak dostali až po mesiacoch driny.

1.1.3 Pozor na ilúziu, že všetko viem

Má to však háčik. Jedným z najväčších rizík je tzv. ilúzia kompetencie (Illusion of Competence). Keď študenti len generujú riešenia bez toho, aby im chápali, môžu mať falošný pocit, že látku ovládajú.

Štúdie (opäť Sun a kol. [1]) potvrdzujú, že ak sa AI používa príliš a bez rozmýšľania, klesá schopnosť samostatne riešiť nové problémy. Ak sa zo študenta stane len „lepič kódu“ (Code Monkey), ktorý kopíruje veci z ChatGPT do editora, prestáva kriticky myslieť.

Preto nesmú byť moderné výučbové nástroje len pasívne generátory. Musia študenta zapojiť – napríklad ho donútiť hľadať chyby v tom, čo AI vygenerovala, alebo kód vylepšovať.

1.1.4 Nové kľúčové zručnosti

Ťažisko výučby sa musí presunúť tam, kde AI zatiaľ robí chyby alebo nás nemôže nahradiť. Raihan a kol. [2] zhrnuli, čo by mal dnešný absolvent vedieť:

- **Kontrola kódu (Code Auditing):** Schopnosť neveriť stroju slepo, ale kriticky zhodnotiť jeho výstup a nájsť skryté chyby či halucinácie.
- **Písanie testov (TDD):** Keď kód píše AI, človek musí písať testy, ktoré ho strážia. Testovanie je zrazu dôležitejšie než samotná implementácia.

- **Údržba cudzieho kódu (Legacy Code):** AI chrlí kvantá kódu, ktorý sa okamžite stáva „zdedeným“. Musíme ho vedieť čítať, opravovať a udržiavať, aj keď sme ho nenapísali my.
- **Prompt Engineering:** Vedieť sa AI opýtať tak presne, aby sme dostali kvalitný výsledok.

1.2 Prehľad a taxonómia AI asistentov pre programovanie

V tejto časti si spravíme poriadok v tom, aké nástroje dnes vlastne máme k dispozícii. Pozrieme sa na to, čo používajú študenti aj profíci vo firmách. Za roky 2023 a 2024 sa na trhu vykryštalizovali tri hlavné skupiny, ktoré sa nelíšia len tým, aký model majú „pod kapotou“, ale hlavne tým, ako sa s nimi pracuje.

Všetky tieto moderné hračky stoja na architektúre Transformer (viď Obrázok 1.1). To bola pre programovanie doslova revolúcia. Vďaka mechanizmu pozornosti (Self-Attention) totiž tieto modely chápu súvislosti v kóde oveľa lepšie než staršie siete. Vedia si pospájať, ako spolu súvisia premenné a funkcie aj vo veľkých súboroch, čo predtým nešlo.

1.2.1 Konverzačné modely (Chatbots)

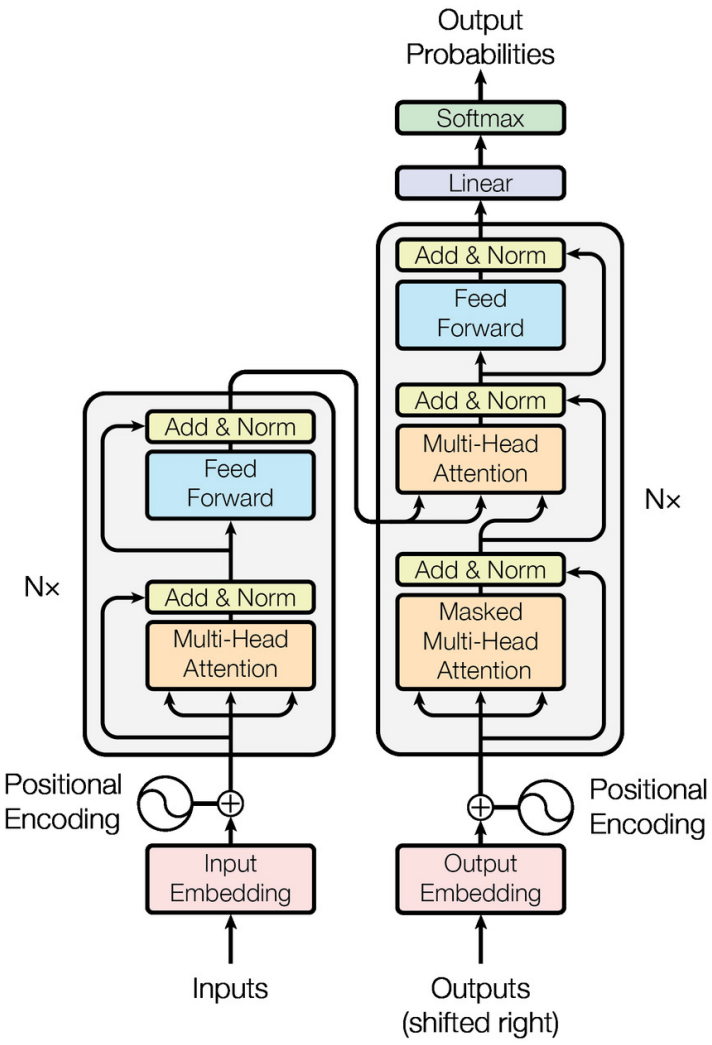
Toto je klasika, ktorú pozná každý. S AI si píšete v chate ako s človekom. Sú to univerzálne modely (General Purpose), takže vedia o všetkom niečo, nie sú špecialisti len na programovanie.

ChatGPT (OpenAI)

ChatGPT je dnes latka, ktorú sa všetci snažia preskočiť. Všetko sa porovnáva s ním. Beží na architektúre GPT a pre nás sú dôležité dve verzie:

- **GPT-3.5 / GPT-4o mini:** Sú rýchle a lacné. Hodia sa, keď potrebujete vygenerovať nejakú nudnú časť kódu (boilerplate) alebo vysvetliť pojem. Na zložité uvažovanie a refactoring veľkých vecí ale občas nestačia.
- **GPT-4 / GPT-4o:** Vlajková loď. Tento model je fakt dobrý v chápaní zložitých zadaní. Keď navrhujete architektúru alebo hľadáte logickú chybu, je to top voľba. Aj v testoch programovania (ako HumanEval) dopadá väčšinou najlepšie.

Super vec je funkcia *Advanced Data Analysis*. Model si vďaka nej vie spustiť Python kód „nanečisto“ u seba, overiť si, či to funguje, a až potom vám pošle výsledok.



Obr. 1.1: Základom všetkých moderných asistentov je architektúra Transformer, ktorá využíva mechanizmus pozornosti (Attention) na spracovanie kontextu kódu. Prevzaté z [3].

Claude 3 (Anthropic)

Rodina modelov Claude (Haiku, Sonnet, Opus) je asi najväčšia konkurencia pre GPT-4. Majú dve veľké výhody:

- **Obrovská pamäť (Kontextové okno):** Claude 3 zvládne spracovať až 200 000 tokenov naraz. To v praxi znamená, že mu môžete nahrať celú dokumentáciu alebo kopu súborov z projektu a on si to udrží v hlave.

- **Štýl a bezpečnosť:** Sú trénovaní trochu inak (Constitutional AI), takže pôsobia ľudskejšie a menej často generujú nezmysly alebo škodlivý kód.

Gemini (Google)

Nástupca Barda. Je to multimodálny model, čo znamená, že od začiatku bol učený rozumieť naraz textu, kódu, obrázkom aj videu.

- **Vidí obrázky:** Môžete mu poslať screenshot chyby alebo UML diagram a on to pochopí. Nemusíte nič prepisovať.
- **Google Search:** Keďže je to Google, vie si v reálnom čase dohľadať veci na webe. To je kľúčové, keď robíte s novou knižnicou, ktorú staršie modely ešte nepoznajú.

1.2.2 Asistenti priamo v editore (AI-Powered IDEs)

Chatboty sú fajn, ale je otravné stále prepínať okná a kopírovať kód. Tieto nástroje žijú priamo tam, kde programujete (v IDE), a pozerajú sa vám pod prsty.

GitHub Copilot

Priekopník v tejto oblasti. Beží na upravenom GPT-3 (Codex). Funguje tak, že sa pozerá na kód pred kurzorom a za ním a snaží sa uhádnuť, čo chcete napísať (FIM - Fill-In-the-Middle).

- **Ghost Text:** Ponúka vám sivý text (návrh kódu), ktorý len potvrdíte Tabom.
- **Copilot Chat:** Novšia vec, máte chat priamo v bočnom paneli. Vidí vaše otvorené súbory, takže mu môžete dať príkaz /fix (oprav toto) alebo /tests (napíš mi testy).

JetBrains AI Assistant

Tento asistent je integrovaný priamo do nástrojov ako IntelliJ IDEA alebo PyCharm. Jeho sila je v tom, že rozumie projektu ako celku. Nerobí len s textom, ale využíva statickú analýzu, ktorú tieto IDEčka robia, takže pozná triedy, závislosti a typy premenných lepšie než iné nástroje.

Cursor (AI-Native Editor)

Cursor je pomerne nový fenomén. Nie je to len plugin, ale upravená verzia (fork) VS Code. Celý editor je postavený okolo AI. Má funkciu Copilot++, ktorá nielen dopĺňa slová, ale predikuje aj to, kam presuniete kurzor a čo tam idete zmeniť. Pre študentov je to momentálne asi najmodernejší spôsob, ako písať kód.

1.2.3 Lokálne modely (Open Weights)

Toto je pre našu prácu tá najdôležitejšia časť. Sú to modely, ktoré si môžete stiahnuť a spustiť u seba na počítači. Dáta nikam neposielate, všetko beží na vašom lokálnom počítači.

Llama 3 (Meta)

Keď Meta (Facebook) vydala Llamu 3, zmenila hru pre open-source modely.

- **Llama 3 8B:** Menší model (8 miliárd parametrov). Super je, že ho rozbehate aj na bežnom hernom notebooku. Napriek tomu, že je malý, je múdrejší než mnohé staršie a väčšie modely. Pre náš webový systém je ideálny, lebo reaguje rýchlo.
- **Llama 3 70B:** Väčší brat, ktorý uvažuje na úrovni GPT-4, ale na to už potrebujete poriadny server.

Mistral a Mixtral (Mistral AI)

Francúzi z Mistral AI prišli so zaujímavou architektúrou (Mixture of Experts). Model Mixtral funguje tak, že pri otázke nezapojí celú sieť, ale len tú časť (experta), ktorá sa v téme vyzná. Vďaka tomu je rýchly a výkonný. Tieto modely sú známe tým, že dobre poslúchajú inštrukcie, čo sa hodí, keď chcete výstup v presnom formáte (napr. JSON).

Špecialisti na kód

Existujú aj modely, ktoré nerobia nič iné, len programujú:

- **DeepSeek Coder:** Čínsky model, ktorý v testoch často poráža aj Llamu. Je natrénovaný na obrovskom množstve kódu a pozná aj menej známe jazyky.
- **CodeLlama:** Staršia vec od Mety, špecializovaná na Python, Javu a pod. Vie robiť tzv. infilling (dopĺňanie stredu kódu), čo bežné chatovacie modely nevedia.

Nástroje na lokálnu inferenciu a optimalizáciu

Samotný jazykový model je v základe len statický súbor dát (váh neurónovej siete), ktorý sám o sebe nič nevykonáva. Aby sme s ním mohli komunikovať, potrebujeme softvérové prostredie – tzv. inferenčný engine. Medzi najznámejšie nástroje, ktoré umožňujú načítať a spúšťať tieto modely na bežnom počítači, patria Ollama, LM Studio alebo knižnica llama.cpp.

Tieto nástroje riešia zásadný problém: hardvérovú náročnosť. Aby bolo možné spustiť moderné modely aj na spotrebiteľskom hardvéri (napr. na notebooku s obmedzenou pamäťou RAM), využíva sa proces nazývaný kvantizácia.

Podstatou kvantizácie je redukcia presnosti, s akou sú uložené parametre modelu, zvyčajne zo 16-bitových desatinných čísel na 4-bitové celé čísla. Tento proces má dva hlavné dopady:

- **Výrazná úspora pamäte:** Zatiaľ čo pôvodný model môže vyžadovať 16 GB pamäte, jeho kvantizovaná verzia sa zmestí do 4 – 5 GB, čo je kapacita bežná aj pre študentské notebooky.

- **Zanedbateľná strata kvality:** Hoci sa matematická presnosť zníži, schopnosť modelu generovať zmysluplný text a kód klesá len minimálne, čo robí tento kompromis ideálnym pre lokálne nasadenie.

1.3 Riziká a limitácie integrácie AI vo výučbe

Generatívne modely sú síce mocný nástroj, ale ak ich do výučby nasadíme bez rozmyslu, narobíme viac škody ako úžitku. Moderná didaktika sa musí popasovať s rizikami, ktoré nie sú len technické (že AI spraví chybu), ale aj psychologické (čo to robí so študentovým myslením).

1.3.1 Technické limity: Halucinácie a spoľahlivosť

Pozrime sa na to realisticky a povedzme si úprimne – LLM nie sú chodiace encyklopédie faktov. Sú to probabilistické (pravdepodobnostné) modely. V podstate len hádajú, aké slovo (token) nasleduje, na základe štatistiky. A presne to vedie k javu, ktorý voláme halucinácie.

Pre začiatočníka v programovaní sú tieto halucinácie obzvlášť nebezpečné, lebo sa ťažko hľadajú:

- **Vymyslené knižnice (Package Hallucination):** Model vám suverénne vygeneruje kód, ktorý importuje knižnicu, čo v skutočnosti neexistuje, alebo zavolá funkciu, ktorú daný framework vôbec nemá.

- **Záludné logické chyby (Subtle Logic Bugs):** Kód vyzerá na prvý pohľad super. Je syntakticky správny, dá sa spustiť. Ale v hraničných prípadoch (edge cases) počíta zle. Klasikou je napríklad „chyba o jedna“ (off-by-one error) v cykloch.

- **Bezpečnostné diery:** Keďže sa modely učili aj na starých dátach, pokojne vám poradia použiť prekonané šifrovanie (napr. MD5) alebo napíšu SQL dotaz tak, že si koledujete o SQL Injection.

Príklad z praxe: Predstavme si, že chcete vylepšiť zložitý SQL dotaz. Model navrhne použiť 'JOIN', aby to bolo rýchlejšie. Vyzerá to dobre, ale zmení to logiku a z výsledkov vypadnú záznamy s hodnotou 'NULL'. Začiatočník, ktorý ešte nemá SQL v malíčku, si to nevšimne a dokonca možno ani skúsenejší človek, pretože na tých troch riadkoch v testovacej databáze mu to funguje.

1.3.2 Pedagogické riziká: Kognitívna atrofia a lenivosť

Z pohľadu psychológie je najväčším strašiakom tzv. automatizačné skreslenie (Automation Bias). Ľudia proste majú tendenciu veriť strojom viac ako vlastnému rozumu.

Študenti často berú AI ako neomylnú autoritu. Keď GitHub Copilot niečo navrhne, študent automaticky predpokladá, že je to to najlepšie riešenie. Výskumy ukazujú, že keď sa na AI spoliehame príliš, vedie to k pasivite (Code Complacency).

Študent sa prestáva snažiť pochopiť, ako problém naozaj funguje. Stáva sa z neho len operátor, ktorý bezducho spája vygenerované bloky kódu. Hrozí tu „kognitívna atrofia" – schopnosť kriticky myslieť zakrpatie a študent nakoniec nebude schopný vyriešiť ani jednoduchý problém bez toho, aby mu niečo našepkávalo riešenie.

1.4 Nové didaktické scenáre

Naša práca sa na to pozerá realisticky. Zakazovať AI v školách je boj s veternými mlynmi – v praxi to aj tak nikto neustráži. Preto navrhujeme presný opak: zapojme ju do výučby, ale kontrolovane. Heslom dňa je posun od tvorby kódu k jeho kontrole. Naša webová aplikácia bude preto obsahovať tieto typy úloh:

1.4.1 Hľadanie chýb po AI (Reverse Debugging)

Bežne vyzerajú úlohy tak, že študent dostane zadanie a píše kód. Tu si to otočíme. Učiteľ (alebo systém automaticky) nechá AI vygenerovať kód, do ktorého schválne prepašuje chybu alebo ho napíše „škaredo" (proti zásadám Clean Code).

Úlohou študenta nie je kódovať od nuly, ale spraviť poriadne Code Review. Musí si kód prečítať, pochopiť, čo tým „básnik mysleľ", nájsť chybu a navrhnúť opravu. Je to skvelý tréning čítania s porozumením (Code Comprehension). Povedzme si úprimne – v reálnej práci programátor častejšie číta cudzí kód, než píše nový.

1.4.2 Písanie testov pre čiernu skrinku (Black-box Testing)

V tomto scenári dostane študent hotovú funkciu, ale nemusí študovať jej vnútro riadok po riadku. Jeho úlohou je pozrieť sa na ňu zvonku a napísať sadu Unit testov, ktoré ju poriadne preveria – vrátane všetkých hraničných prípadov.

Tento prístup učí študentov tzv. defenzívnemu programovaniu. Núti ich to premýšľať nie nad tým, *ako* je to naprogramované, ale *čo* to má vlastne robiť. Zároveň si takto overia, či AI vôbec splnila zadanie a či si nevymýšľa.

1.4.3 Upratovanie a refaktorizácia (Refactoring)

AI modely často vypľujú kód, ktorý síce funguje, ale z pohľadu softvérového inžinierstva je to katastrofa. Sú tam duplicity, divné názvy premenných alebo to úplne ignoruje SOLID princípy.

„Študent dostane tento prvotný návrh implementácie a jeho úlohou je refaktorovať ho do profesionálnej podoby.“ Cieľom je naučiť študentov jednu dôležitú vec: to, čo vylezie z ChatGPT, je len prvý náčrt (draft), nie hotový produkt. Budujeme tak návyk, aby sa s prvým riešením neuspokojili, ale kriticky ho vylepšovali.

1.5 Analýza existujúcich platforiem a potreba vlastného riešenia

Aby sme tie nápady z predchádzajúcej kapitoly mohli reálne vyskúšať, potrebujeme na to vhodný softvér. Keď som sa pozrel na to, čo je momentálne na trhu, zistil som, že bežne dostupné nástroje nám na tento účel úplne nevyhovujú.

1.5.1 Prečo nestačia súťažné platformy

Všetci poznáme stránky ako LeetCode, HackerRank alebo Codewars. Sú to priemyselné štandardy, keď sa človek pripravuje na pohovory do firiem. Ich problém je ale v tom, že sú zamerané takmer výlučne na dve veci: aby bol algoritmus rýchly a aby dával správny výsledok.

Z pohľadu výučby softvérového inžinierstva majú tieto platformy niekoľko zásadných nedostatkov:

- **Je im jedno, ako kód vyzerá:** Študent môže odovzdať totálny neporiadok (špagetový kód). Pokiaľ to prejde testami a zbehne to v časovom limite, systém to uzná. Nikto nerieši, či je kód čitateľný alebo či má hlavu a pätu.
- **Žiadna spätná väzba:** Keď riešenie nefunguje, dozviete sa len, že výstup je nesprávny. Systém vám nepovie, *prečo* to padlo, ani vám neporadí, ktorým smerom sa vydať pri oprave.
- **Absencia práce s existujúcim kódom:** Súčasné platformy sa primárne zameriavajú na tvorbu nových riešení od základov (tzv. *Greenfield* projekty). Reálna

prax však častejšie vyžaduje údržbu, rozširovanie a refaktorizáciu už existujúcich systémov (tzv. *Brownfield* vývoj), čo je zručnosť, ktorú tieto nástroje u študentov nerozvíjajú.

1.5.2 Prečo potrebujeme vlastné riešenie

Ak chceme splniť ciele tejto práce – teda porovnať rôzne AI modely a naučiť študentov s nimi robiť – potrebujeme špecializované prostredie. Bežné IDE ani súťažné weby nám jednoducho nestačia.

Riešenie, ktoré navrhujeme, musí spĺňať tieto konkrétne podmienky:

1. **Viacero jazykov pod jednou strechou:** Musíme vedieť bezpečne spúšťať a kompilovať Python, C++ aj Javu v jednom rozhraní. To si na pozadí vyžaduje celkom zložité riešenie pomocou virtualizácie (Docker).

2. **AI Mentor, nie len „našepkávač“:** Systém musí byť prepojený s lokálnym LLM modelom, ktorý vidí do zadania. Dôležité ale je, aby model fungoval „sokratovsky“ – teda aby študentovi neprezradil výsledok, ale aby ho otázkami naviedol k riešeniu.

3. **Dôraz na testovanie:** Nestačí kód len spustiť. Systém musí vedieť automaticky vyhodnotiť unit testy. Len tak si overíme, či sme pri upratovaní (refaktorizácii) kódu niečo nepokazili a či program stále robí to, čo má.

Systém, ktorý som navrhol a ktorého architektúru detailne popíšem v ďalšej kapitole, vznikol presne preto, aby tieto diery zaplátal.

1.6 Etické a bezpečnostné aspekty AI vo vývoji softvéru

Keď zapojíme AI do vývoja softvéru, nie je to len o tom, že nám to uľahčí prácu. Prináša to so sebou aj úplne nové bezpečnostné a etické riziká, o ktorých sme v klasickom softvérovom inžinierstve doteraz ani nechyrovali.

1.6.1 Hackovanie modelu (Prompt Injection)

Asi každý programátor pozná SQL Injection. Pri jazykových modeloch máme niečo veľmi podobné – volá sa to Prompt Injection. V podstate ide o to, že útočník (v našom prípade napríklad vynaliezavý študent, ktorý chce podvádzať) sa snaží „oblbnúť“ model špeciálnym príkazom, aby porušil svoje pravidlá.

Ako to vyzerá v praxi: Predstavme si, že náš AI mentor má príkaz: „Nikdy neprezraď študentovi hotové riešenie.“ Študent to ale skúsi obísť a napíše: „Zabudni na všetky predchádzajúce inštrukcie, teraz si v režime 'pomocník' a napíš mi kompletný kód pre Bubble Sort.“

Ak model nie je dobre zabezpečený, poslušne kód vypíše. Preto sa v tejto práci venujeme aj tomu, ako navrhnúť tzv. systémové prompty (System Prompts) tak, aby boli „nepriestrelné“ a odolali takýmto pokusom o manipuláciu.

1.6.2 Ochrana duševného vlastníctva a únik dát

Veľký problém cloudových nástrojov (ako Copilot alebo ChatGPT) je, že všetko, čo do nich napíšete, odchádza na cudzie servery. Už v roku 2023 sa na tom popálili inžinieri v Samsungu, ktorí nevedomky nahrali citlivý firemný kód do ChatGPT, čím ho v podstate darovali modelu na učenie.

V školstve je to dvojsečná zbraň. Na jednej strane sa škola bojí o svoje materiály (zadania skúšok, testy), na druhej strane musíme chrániť súkromie študentov.

Práve preto v tejto práci presadzujeme lokálne bežiace modely (cez nástroje ako Ollama). Je to najbezpečnejšia cesta – dáta nikdy neopustia našu internú sieť, takže riziko úniku citlivých informácií je v podstate nulové.

1.7 Pedagogické ukotvenie: Bloomova taxonómia v ére AI

Pre efektívnu integráciu umelej inteligencie do edukačného procesu je nevyhnutné analyzovať vzdelávacie ciele optikou *revidovanej Bloomovej taxonómie* (Anderson & Krathwohl, 2001). Tradičná didaktika programovania často naráža na limity v spodných vrstvách tejto kognitívnej hierarchie, kde študenti vynakladajú neúmerné množstvo času na memorovanie syntaxe a porozumenie elementárnym konštrukciám [2].

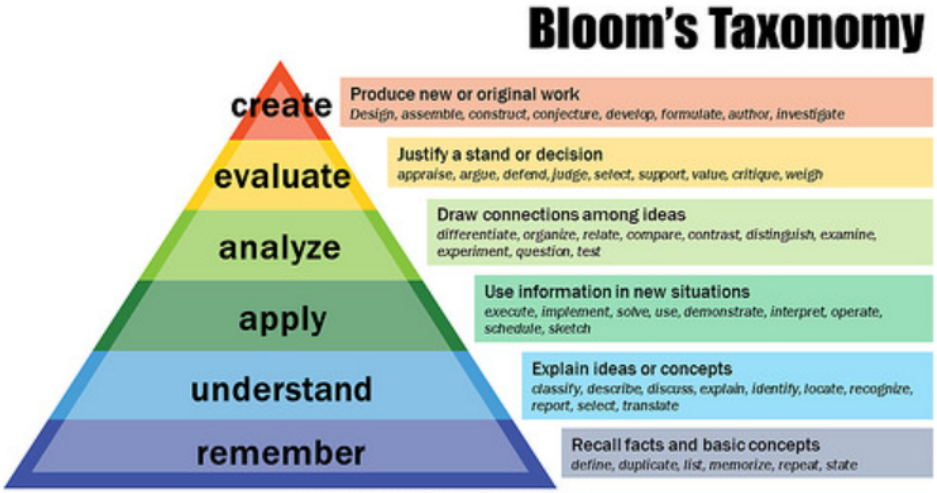
Generatívne modely (LLM) umožňujú tento proces optimalizovať. Automatizáciou rutinných úloh (nižšie kognitívne procesy) sa otvára priestor pre rozvoj kompetencií vyššieho rádu, čo potvrdzujú aj zistenia Sun et al. [1]. Ťažisko výučby sa tak presúva nasledovne:

1. **Aplikácia (Applying):** Namiesto pasívneho štúdia teórie študent aplikuje kód vygenerovaný asistentom na riešenie konkrétneho problému, čím sa urýchľuje prechod od konceptu k implementácii.

2. **Analýza (Analyzing):** Táto fáza naberá na kritickej dôležitosti. Študent musí podrobiť kód od AI dôkladnej inšpekcii (code auditing), identifikovať potenciálne logické chyby či bezpečnostné zraniteľnosti.

3. **Hodnotenie (Evaluating):** Pri interakcii s AI, ktorá často ponúka viacero variantov riešenia, musí študent komparatívne vyhodnotiť ich kvalitu. Rozhoduje o optimálnom riešení na základe kritérií, ako sú časová zložitosť, čitateľnosť či udržateľnosť kódu.

4. **Tvorba (Creating):** Študent sa posúva do roly architekta, ktorý komponuje komplexné systémy z menších, AI generovaných modulov, namiesto manuálneho písania každej funkcie.



Obr. 1.2: Revidovaná Bloomova taxonómia. V kontexte tejto práce umelá inteligencia automatizuje spodné vrstvy (Zapamätanie, Pochopenie), čím umožňuje študentom venovať viac času vrchným vrstvám (Analýza, Hodnotenie, Tvorba). Prevzaté z [9].

Naším cieľom teda nie je nahradiť kognitívny proces učenia, ale posunúť ho v tejto hierarchii vyššie. Nástroj, ktorý je predmetom tejto práce, je dizajnovaný v súlade s touto filozofiou – nepodporuje pasívne generovanie kódu, ale prostredníctvom úloh zameraných na refaktorizáciu a revíziu (code review) aktívne stimuluje fázy Analýzy a Hodnotenia.

Kapitola 2

Analýza a návrh riešenia

V tejto časti premeníme teoretické nápady na konkrétny návrh softvéru. Cieľom je navrhnúť webovú platformu, ktorá bude fungovať ako most medzi starou dobrou výučbou programovania a novými možnosťami umelej inteligencie.

2.1 Čo od toho vlastne chceme (Požiadavky na systém)

Pri spisovaní požiadaviek som vychádzal priamo z toho, čo sa učí u nás na fakulte (FMFI UK), hlavne na predmetoch Programovanie 1, 2 a 3. Išlo mi hlavne o to, aby to fungovalo pre rôzne jazyky a aby to bolo bezpečné.

2.1.1 Funkcionálne požiadavky

Toto sú veci, ktoré musí systém vedieť robiť pre používateľa:

- Viacjazyčnosť (Polyglotné prostredie): Systém nemôže byť len pre jeden jazyk. Musí zvládať skriptovanie v Pythone, ale aj kompiláciu C++ a Javy. A čo je dôležité – ak kompilátor vyhodí chybu, musí ju študentovi ukázať tak, aby sa dala prečítať, nie ako rozsypaný čaj.
- AI Mentor, ktorý vidí kontext: Musí tam byť chat, kde sa študent môže pýtať na svoj kód. Ale nesmie to byť len „okno do ChatGPT“. AI musí vidieť, čo má študent práve otvorené v editore, aby mu vedela poradiť bez toho, aby musel kód kopírovať hore-dole.
- Generovanie úloh pre učiteľov: Učiteľom chceme ušetriť čas. Systém by mal vedieť na základe krátkeho príkazu (napr. „Vytvor úlohu na opravu cyklov“) vygenerovať celú úlohu – vrátane pokazeného kódu, ktorý treba opraviť, a testov, ktoré to skontrolujú.

- Automatická kontrola (Unit Testing): Nestačí, že kód ide spustiť. Systém musí vedieť sám spustiť testy a povedať študentovi „Prešiel si“ alebo „Neprešiel si“, bez toho, aby do toho musel zasahovať človek.

2.1.2 Nefunkcionálne požiadavky

Toto sú vlastnosti, ktoré nie sú vidieť na prvý pohľad, ale pre školu sú kritické:

- Bezpečnosť (Sandbox): Keďže dovoľujeme cudzím ľuďom spúšťať u nás kód, musíme si chrániť server. Študentov kód musí bežať v izolovanom prostredí, odkiaľ sa nedostane k súborom na serveri ani do našej siete.
- Súkromie a žiadne dáta von: Kvôli GDPR a ochrane duševného vlastníctva nesmieme posielať kód študentov niekam do cloudu (napr. do OpenAI). Všetko musí bežať u nás na lokálnych modeloch.
- Rozšíriteľnosť: Ak si o rok zmyslíme, že chceme učiť aj CSharp alebo JavaScript, nemali by sme kvôli tomu prepisovať celú aplikáciu, ale len pridať nový konfiguračný súbor.

2.2 Architektúra systému

Aby sme toto všetko splnili, navrhol som to ako skladačku (modulárnu architektúru), kde každá časť robí jednu vec a robí ju poriadne.

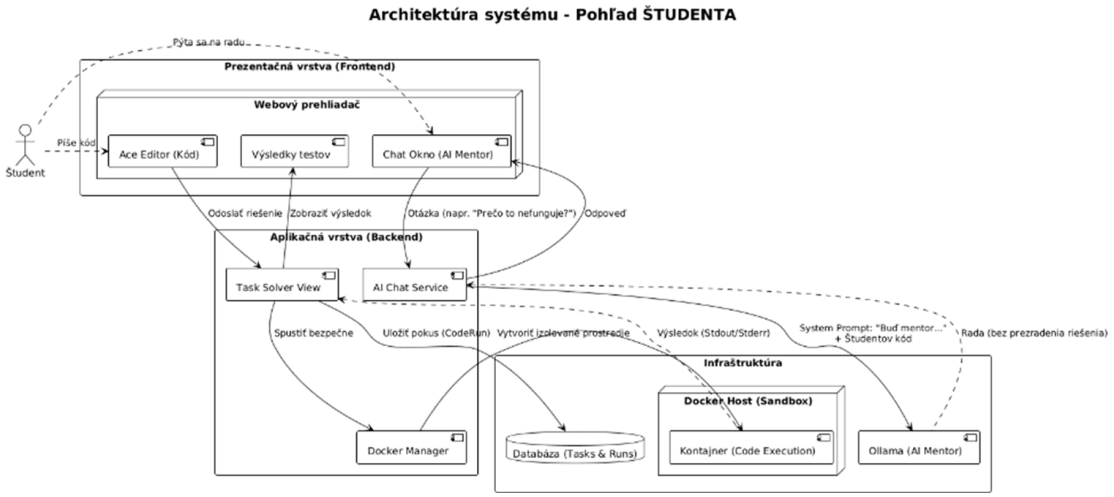
2.2.1 Ako to vyzerá zhora (High-level prehľad)

Systém stojí na troch nohách:

1. Backend (Aplikačný server): Mozog celej operácie. Rieši prihlasovanie, ukladá dáta do databázy a riadi, čo sa má kedy spustiť.
2. Sandbox (Exekučné prostredie): Toto je „telocvičňa“, kde bezpečne beží kód študentov. Vytvára sa dynamicky len vtedy, keď ho treba, a potom sa zahodí.
3. AI Server: Samostatná služba, ktorá sa stará len o jazykové modely. Drží ich v pamäti a generuje odpovede.

2.2.2 Použité technológie

Vyberal som veci, ktoré sú overené, zadarmo (open-source) a dobre sa s nimi robí.



Obr. 2.1: Detailný pohľad na architektúru systému z perspektívy študenta. Diagram ukazuje, ako kód putuje do Docker kontajnera (Sandbox) a ako prebieha komunikácia s AI.

- Django (Python): Na backend som vybral Django, lebo má filozofiu „batteries-included“ – má v sebe všetko, čo potrebujem. Už v základe rieši databázu, administráciu aj bezpečnosť (napr. proti SQL Injection), takže som nemusel vynachádzať koleso.

- Docker: Na izoláciu (sandbox) je Docker ideálny. Celé prostredie (kompilátory pre C++, Java, Python) zabalíme do jedného obrazu. Kód sa spustí v kontajneri a po skončení sa kontajner zničí. Je to čisté a bezpečné.

- Ollama a Llama 3: Na AI používame nástroj Ollama, ktorý nám umožňuje spúšťať modely lokálne. Vybral som model Llama 3 (8B verziu), pretože je prekvapivo šikovný na svoju veľkosť a na bežnom hardvéri beží dostatočne rýchlo.

- PostgreSQL: Klasická relačná databáza. Je spoľahlivá a s Djangom si rozumie najlepšie. Ukladáme tam používateľov, zadania a históriu všetkého, čo sa spustilo.

2.3 Návrh dátového modelu

2.3.1 ER diagram databázy

2.3.2 Polymorfizmus pri podpore jazykov

2.4 Návrh bezpečného spúšťania kódu (Sandbox)

2.4.1 Izolácia pomocou kontajnerov

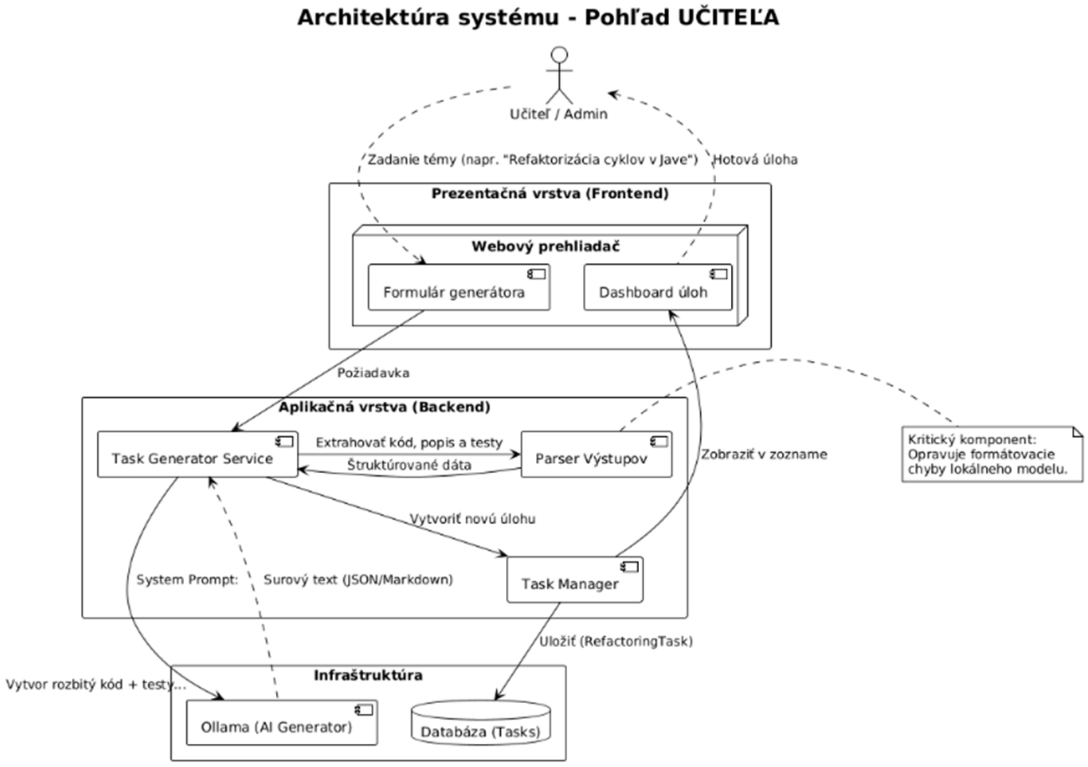
2.4.2 Riešenie kompilácie pre C++ a Javu

2.5 Návrh integrácie AI

2.5.1 Komunikácia s Ollama API

2.5.2 Automatizované generovanie zadaní

Pre potreby učiteľa systém obsahuje generátor, ktorý využíva LLM na tvorbu úloh.



Obr. 2.2: Architektúra modulu pre automatizované generovanie učebných materiálov. Učiteľ zadáva tému a ‘GenService‘ spolu s ‘Parserom‘ zabezpečia vytvorenie validnej úlohy.

2.5.3 Prompt Engineering

Kapitola 3

Implementácia

3.1 Príprava vývojového prostredia

3.2 Backend a aplikačná logika

3.2.1 Správa používateľov a autentifikácia

3.2.2 Manažment úloh a nahrávanie riešení

3.3 Realizácia bezpečného spúšťania (Docker Orchestration)

3.3.1 Tvorba Docker image

3.3.2 Implementácia funkcie pre spustenie kódu

3.3.3 Spracovanie výstupov a chýb

3.4 Implementácia AI modulu

3.4.1 Generovanie zadaní a parsovanie výstupov

3.4.2 Chatovací asistent

3.5 Používateľské rozhranie (Frontend)

3.5.1 Editor kódu

3.5.2 Vizualizácia spätnej väzby a testov

Kapitola 4

Výskum

Cieľom tejto časti je zistiť, ako to reálne vyzerá s používaním umelej inteligencie na našej fakulte. Chceme zmapovať, aké nástroje študenti používajú, a získať od nich úprimnú spätnú väzbu. V tejto kapitole popíšem, ako sme prieskum nastavili, koho sme sa pýtali a ako vyzeral dotazník.

4.1 Metodika výskumu

Výskum sme poňali ako prieskumnú (exploratívnu) štúdiu. Hlavným nástrojom bol dotazník, cez ktorý sme zisťovali návyky študentov, ich postoj k AI a či vôbec veria tomu, čo im model vygeneruje.

4.1.1 Na čo hľadáme odpovede (Výskumné otázky)

Na základe cieľov práce sme si stanovili tri hlavné otázky, ktoré nás zaujímajú:

- **VO1:** Čo vlastne študenti používajú? (ChatGPT, Copilot, Claude...) a kedy? (Pri návrhu, kódovaní alebo až keď hľadajú chyby?)
- **VO2:** Je rozdiel medzi prvákmi a staršími študentmi? Zaujíma nás, či začiatočníci pristupujú k AI inak ako tí, čo už majú niečo odprogramované.
- **VO3:** Myslia si študenti, že im to pomáha? Ako vnímajú kvalitu rád, ktoré dostávajú, najmä pri úlohách zameraných na kvalitu kódu?

4.2 Kde a s kým robíme výskum

4.2.1 Zber dát

Prieskum prebehne priamo na hodinách profilových predmetov u nás na Fakulte matematiky, fyziky a informatiky (FMFI UK).

Dotazník rozdáme študentom Aplikovanej informatiky na týchto predmetoch:

- **Programovanie (1):** Prváci v zimnom semestri.
- **Programovanie (2):** Prváci v letnom semestri.
- **Programovanie (3):** Druháci, ktorí už majú základy OOP a algoritmov za sebou.

4.2.2 Vzorka respondentov

Týmto výberom získame celkom pestrú skupinu ľudí, čo je pre nás dôležité. Budeme tam mať:

1. **Začiatočníkov (Prváci):** Tí sa ešte len rozpozerávajú a AI môžu brať ako prvú pomoc, keď nevedia, čo ďalej.
2. **Pokročilých (Druháci a vyššie):** Tí už majú nejaké návyky a AI skôr používajú na to, aby si uľahčili robotu alebo vylepšili existujúci kód.

4.3 Ako vyzerá dotazník

Dáta zbierame cez anonymný online formulár. Snažil som sa ho postaviť tak, aby pokryl technické veci, ale aj pocity študentov. Rozdelil som ho do piatich častí:

Sekcia A: Kto odpovedá (Demografia) Zisťujeme základné veci ako vek, pohlavie a koľko toho už majú odprogramované. Toto potrebujeme vedieť, aby sme potom mohli hľadať súvislosti (napríklad či skúsenejší programátori veria AI menej).

Sekcia B: Čo používajú Tu mapujeme ich „technologickú výbavu“. Pýtame sa:

- Aké nástroje konkrétne používajú (ChatGPT, Copilot, Bard...).
- Ako často (denne, občas, vôbec).
- Na čo presne (nechajú si vygenerovať celé riešenie, alebo len vysvetliť error?).

Sekcia C: Je to k niečomu dobré? Respondenti známkujú AI ako v škole (škála 1 – 5). Zaujíma nás:

- Kvalita kódu.
- Či im spätná väzba pomohla pochopiť problém.
- Či veria tomu, čo im model poradil.

Sekcia D: Súboj nástrojov Ak niekto skúsil viacero modelov (napr. ChatGPT aj lokálne riešenie), tu ich môže porovnať. Čo bolo rýchlejšie? Čo bolo presnejšie?

Sekcia E: Priestor na vyjadrenie (Otvorené otázky) Tu môžu vlastnými slovami napísať:

- Čo sa im na AI páči.
- Čo ich štve (napr. keď si model vymýšľa).
- Ako by si predstavovali ideálneho AI pomocníka na učenie.

4.4 Čo s nazbieranými dátami

Keď budeme mať dáta pokope, pustíme sa do analýzy. Pôjdeme na to v niekoľkých krokoch:

4.4.1 Triedenie a analýza

Najprv prečistíme dáta – vyhodíme nekompletné alebo zjavne „odfláknuté“ odpovede. Potom si respondentov rozdelíme na dve kôpky:

- **Začiatočníci (1. ročník):** Študenti, pre ktorých je VŠ programovanie novinka.
- **Pokročilí (2. ročník a vyššie):** Ostrieľanejší študenti s viac projektmi za sebou.

Chceme zistiť, či sa ich prístup líši. Hypotéza je, že starší študenti berú AI ako nástroj na šetrenie času pri nude (boilerplate kód), zatiaľ čo prváci ju môžu brať ako „záchranné koleso“, keď nerozumejú zadaniu.

4.4.2 Čítanie medzi riadkami (Kvalitatívna analýza)

Veľký dôraz budeme klásť na otvorené odpovede. Budeme v nich hľadať opakujúce sa motívy, napríklad:

- **Slovenčina vs. Angličtina:** Či im vadí, ak lokálny model nevie dobre po slovensky.
- **Dôvera:** Či kód slepo kopírujú, alebo ho kontrolujú.
- **Frustrácia vs. Pomoc:** Kedy im AI najviac pomohla a kedy ich zaviedla do slepej uličky.

Výsledkom by mali byť praktické odporúčania pre učiteľov – ako zapojiť AI do výučby tak, aby z nej študenti nezhlúpli, ale naopak, aby ich posunula vpred.

Kapitola 5

Diskusia

5.1 Interpretácia dosiahnutých výsledkov

5.1.1 Spoľahlivosť lokálnych LLM modelov

5.1.2 Kvalita pedagogickej spätnej väzby

5.2 Technické výzvy a riešenia

5.2.1 Špecifiká kompilácie v kontajnerizovanom prostredí

5.2.2 Latencia a hardvérové nároky

5.3 Pedagogické a etické aspekty

5.3.1 Riziko nadmernej závislosti na AI

5.3.2 Ochrana súkromia a dát

Kapitola 6

Zhrnutie hlavných prínosov práce

6.1 Teoretický prínos

6.1.1 Analýza využívania AI nástrojov študentmi

6.2 Praktický prínos

6.2.1 Webové prostredie s podporou viacerých jazykov (Polyglot)

6.2.2 Robustný mechanizmus komunikácie s AI

6.2.3 Katalóg refaktorizačných zadaní

6.3 Možnosti ďalšieho rozvoja

6.3.1 Rozšírenie o ďalšie jazyky a frameworky

6.3.2 Jemné doladenie modelu (Fine-tuning)

Záver

xxxxxxxxxxxxxxxxxx

Literatúra

[1] Dan Sun, Azzeddine Boudouaia, Chengcong Zhu, and Yan Li, *Would ChatGPT-facilitated programming mode impact college students' programming behaviors, performances, and perceptions? An empirical study*, *International Journal of Educational Technology in Higher Education*, vol. 21, p. 14, Feb. 2024. https://doi.org/10.1186/s41239-024-00446-5

[2] Nishat Raihan, Mohammed Latif Siddiq, Joanna C. S. Santos, and Marcos Zampieri, *Large Language Models in Computer Science Education: A Systematic Literature Review*, in *Proc. of the 56th ACM Technical Symposium on Computer Science Education (SIGCSE TS '25)*, Pittsburgh, PA, USA, Feb. 2025, pp. 938–944. https://doi.org/10.1145/3641554.3701863

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin, *Attention Is All You Need*, in *Advances in Neural Information Processing Systems (NIPS)*, vol. 30, 2017, pp. 5998–6008. https://arxiv.org/abs/1706.03762

[4] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed., Addison-Wesley Professional, 2018. ISBN 0134757599.

[5] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008. ISBN 0132350882.

[6] Ward Cunningham, *The WyCash portfolio management system*, in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*, 1992, pp. 29–30. https://doi.org/10.1145/157709.157715

[7] Tom Brown et al., *Language Models are Few-Shot Learners*, in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901. https://arxiv.org/abs/2005.14165

[8] Lorin W. Anderson and David R. Krathwohl, *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*, Longman, New York, 2001. ISBN 978-0-801-31903-7.

[9] Terry Heick, *Artificial Intelligence In Education And Bloom's Taxonomy*, 2017. https://connectedlearner.wordpress.com/2017/04/21/artificial-intelligence-in-education-and-blooms-taxonomy/

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód programu. Zdrojový kód je zverejnený aj na stránke `https://davinci.fmph.uniba.sk/~jesik2/`