Mendelova univerzita v Brně
Provozně ekonomická fakulta

Rozšíření pro testovací rámec ExTester do editoru Visual Studio Code

Diplomová práce

Vedoucí práce:
Ing. David Procházka, Ph.D.

Bc. Filip Pospíšil

Brno 2025

Poděkování

Rád bych poděkoval vedoucímu diplomové práce, panu Ing. Davidu Procházkovi, Ph.D., za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. V neposlední řadě děkuji své partnerce za její podporu, povzbuzení a trpělivost po celou dobu mého studia.

Čestné prohlášení

Prohlašuji, že jsem práci **Rozšíření pro testovací rámec ExTester do editoru Visual Studio Code** vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a v souladu s platnou Směrnicí o zveřejňování závěrečných prací.

Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

Brno, dne 21. prosince 2025 .....

podpis

Abstract

This master thesis focuses on the use of generative artificial intelligence for automating the testing of user interfaces for extensions in the Visual Studio Code editor. The aim of the thesis was to design and implement a prototype tool in the form of a Visual Studio Code extension that combines the existing ExTester testing framework with generative language models and can automatically generate user interface tests for a given extension. The proposed solution includes a user interface in the Visual Studio Code sidebar, loading the manifest of the tested extension to obtain context, and integrating a cloud AI service to generate test scenarios and test source code. The ExTester framework is then used to run the generated tests and analyze their results. The tool can also automatically suggest fixes when tests fail.

Developed plugin expands the capabilities of the ExTester framework and indicates the direction of future development of software testing tools.

Key words: user interface testing, large language model, generative AI, Visual Studio Code, VS Code, ExTester

Abstrakt

Tato diplomová práce se zaměřuje na využití generativní umělé inteligence pro automatizaci testování uživatelského rozhraní rozšíření v editoru Visual Studio Code. Cílem práce bylo navrhnout a implementovat prototyp nástroje ve formě Visual Studio Code rozšíření, který kombinuje existující testovací rámec ExTester s generativními jazykovými modely a dokáže automaticky generovat testy uživatelského rozhraní pro dané rozšíření. Navržené řešení zahrnuje uživatelské rozhraní v postranním panelu Visual Studio Code, načtení manifestu testovaného rozšíření pro získání kontextu a integraci cloudové AI služby pro generování testovacích scénářů a zdrojového kódu testů. Framework ExTester poté slouží ke spuštění vygenerovaných testů a analýze jejich výsledků. Nástroj rovněž dokáže automaticky navrhnout opravy při selhání testů.

Vyvinutý plugin rozšiřuje možnosti frameworku ExTester a naznačuje směr budoucího vývoje nástrojů pro testování softwaru.

Klíčová slova: testování uživatelského rozhraní, velký jazykový model, generativní AI, Visual Studio Code, VS Code, ExTester

Obsah

1 Úvod a cíl práce

Díky bohatému ekosystému zásuvných modulů (rozšíření) je *Visual Studio Code* (dále *VS Code*) jedním z nejrozšířenějších vývojových prostředí současnosti. Kvalita těchto rozšíření má přímý dopad na produktivitu vývojářů, a proto je důkladné testování každého pluginu nezbytné. Testování *VS Code* rozšíření ovšem představuje specifickou výzvu – zahrnuje totiž nejen kontrolu funkční logiky, ale i integraci s uživatelským rozhraním editoru. Tradiční přístup spoléhá převážně na ručně psané testy a manuální zkoušení funkcionality. Se stoupající komplexitou modulů a scénářů použití se ruční návrh testovacích případů stává obtížně škálovatelným. Proto existuje snaha tento proces automatizovat – ať už pomocí skriptů simulujících akce uživatele, nebo využitím specializovaných testovacích frameworků.

V posledních letech se objevuje nová možnost pro zvýšení efektivity testování softwaru: využití generativní umělé inteligence. Moderní nástroje umělé inteligence (Artificial Intelligence, dále AI) dokáží na základě textového zadání generovat kód nebo celé testovací sekvence. V kontextu *VS Code* pluginů to znamená, že AI může například z popisu scénáře („uživatel klikne na tlačítko rozšíření, očekává se otevření dialogu…“) přímo navrhnout kód testu ověřující dané chování. První studie naznačují, že takový přístup může výrazně ušetřit čas. V [21] ukazují, že generování *end-to-end* [1] testů pomocí modelu *ChatGPT* zkrátilo dobu vývoje testovacích skriptů oproti čistě manuálnímu postupu a to v některých případech statisticky významně. Umělá inteligence tak slibuje urychlení tvorby testů a rozšíření pokrytí, aniž by tester musel vše programovat ručně.

Zároveň však zkušenosti ukazují, že AI nenahradí testery úplně a bez omezení. Modely jako *ChatGPT* mohou vygenerovat základ testovacího skriptu, ale výsledný kód často vyžaduje dopracování a kontrolu ze strany vývojáře. Například u testů interagujících s uživatelským rozhraním je nutné pečlivě navrhnout testovací scénáře a ověřit, že vygenerované aserce[2] skutečně ověřují správné chování aplikace. Obecně platí, že AI usnadňuje vytvoření návrhu testu, ale finální doladění a ověření správnosti zůstává na člověku. Kromě toho existují případy, kdy generativní model nedokáže správně vyřešit složitou logiku očekávaných výsledků. Dle [22] s rostoucí obtížností úlohy výrazně klesá přesnost velkých jazykových modelů (Large Language Model, dále LLM) při generování korektních testovacích případů.

Motivací této práce je prozkoumat oblast využití umělé inteligence pro automatizaci testování pluginů na úrovni uživatelského rozhraní ve *VS Code* a současně ověřit, jaké jsou limity a požadavky takového řešení v praxi.

Cílem práce je navrhnout a realizovat efektivní postup automatizovaného testování uživatelského rozhraní rozšíření pro *VS Code* pomocí frameworku *ExTester* s podporou generativní AI a následně ověřit jeho přínosy. Práce usiluje o vytvoření prototypu nástroje (formou *VS Code* rozšíření), který na základě popisu scénářů automaticky vygeneruje testy uživatelského rozhraní (User Interface, dále UI) daného pluginu a umožní jejich spuštění.

[1] Testy, které ověřují funkčnost celého systému od začátku až do konce, simuluji reálné uživatelské interakce a zajišťují, že všechny komponenty spolu správně fungují.

[2] Tvrzení používané v rámci testování softwaru, které ověřuje splnění definované podmínky.

Tím má být dosaženo zrychlení a zjednodušení testovacího procesu – omezení rutinní manuální práce testerů při zachování schopnosti odhalit závažné chyby v rozšíření.

K naplnění tohoto cíle bude postupováno následovně: nejprve bude provedena rešerše přístupů k automatizovanému generování testů s využitím generativní AI a analýza možností jejich integrace s testovacím frameworkem *ExTester*. Na základě těchto poznatků bude navrženo řešení, které kombinuje generativní tvorbu UI testů s mechanismy jejich validace, iterativní opravy a kontroly sémantické správnosti v rámci frameworku *ExTester*. Po implementaci prototypu bude provedeno experimentální vyhodnocení na vybraném rozšíření pro *Visual Studio Code*. Vyhodnocení se zaměří především na posouzení přínosu AI asistence při tvorbě UI testů, konkrétně na rychlost návrhu testovacích scénářů, použitelnost vygenerovaných testů a jejich sémantickou kvalitu ve vztahu k reálnému chování testovaného rozšíření. Zvláštní pozornost bude věnována identifikaci případů, kdy generované testy sice technicky procházejí, avšak neověřují deklarovanou funkcionalitu, a tedy vyžadují manuální zásah vývojáře. Výsledkem práce tak bude ověření, zda a do jaké míry může AI asistence v kombinaci s frameworkem *ExTester* zefektivnit proces tvorby UI testů pro *VS Code* rozšíření, a současně vymezení omezení tohoto přístupu, zejména v oblasti sémantické správnosti a nutnosti lidské kontroly.

2 Současný stav testování rozšíření

V současné době se rozšíření testuje na několika úrovních, aby byla pokryta jak vnitřní logika, tak interakce s hostitelskou aplikací. Základem jsou jednotkové testy, které ověřují izolované části kódu nezávislé na *VS Code* API. Následují integrační testy zaměřené na spolupráci s platformou *VS Code* v rámci speciálního *Extension Development Host* režimu. Pro co nejrealističtější simulaci uživatelských scénářů pak slouží testy uživatelského rozhraní, které rozšíření ovládají podobně, jako by ho ovládal skutečný uživatel. Tato vícestupňová strategie pomáhá odhalit chyby v logice i integraci a zajišťuje vysokou kvalitu rozšíření před jeho nasazením k reálným uživatelům.

2.1 Typy testů v rámci rozšíření

2.1.1 Jednotkové testy

Zaměřují se na izolované funkční celky rozšíření. Typicky testují jednotlivé funkce nebo moduly implementované v kódu. Cílem testu je ověřit, že vnitřní logika funguje správně v různých scénářích a to bez závislosti na *VS Code* API. Jednotkové testy běží přímo v *Node.js* bez nutnosti spouštění instance *VS Codu*. Místo skutečných objektů z *VS Code* se používají mocky[3] nebo *stuby*[4]. Jednotkové testy díky tomu mohou běžet v libovolném prostředí a lze je snadno kontinuálně integrovat. Příkladem jednotkového testu může být ověření, že funkce pro zpracování textu vrací očekávaný výstup pro daný vstup. Z pohledu logiky *VS Code* sem spadá většina logiky, která není závislá na instanci editoru. Takovou logiku je možné vyčlenit do samostatných modulů a testovat je klasickými nástroji jako je *Mocha* či *Jest*.

2.1.2 Integrační testy

Testují rozšíření v kontextu běžící instance *VS Code*. Ověřují tedy interakci kódu rozšíření s *VS Code* API a případně dalšími rozšířeními či systémem. *VS Code* umožňuje spouštět integrační testy ve speciálním režimu, tzv. *Extesnion Development Host*, což představuje izolovanou instanci *VS Code* určenou pro vývoj a testování doplňků. Tato instance načte testované rozšíření a testy pak mohou vyvolávat *VS Code* API. Testy přesahují rámec čistě izolované jednotky kódu a zahrnují plnou integraci s platformou *VS Code*. Typicky se pro ně využívá stejný framework jako pro testy jednotkové s příslušným nastavením prostředí pro běh uvnitř instance *VS Code*. Již oficiální *scaffolding*[5] v *Yeoman* šabloně pro *VS Code* rozšíření generuje příklad integračních testů a skripty pro jejich spuštění – ty zajistí stažení odpovídající verze *VS Code* a spuštění testů uvnitř něj pomocí *Mocha*.[13] Integrační testy ověřují funkčnost jako je registrace příkazů, reakce na události editoru či správná manipulace s *VS Code* API. Příkladem takového testu je například otevření virtuálního

[3] Simuluje chování závislosti a umožňuje ověřovat interakce (např. volání metod).

[4] Simuluje chování závislosti a vrací předem definované hodnoty.

[5] Automatické generování základní struktury projektu, včetně potřebných souborů a příkladů pro rychlý start vývoje.

dokumentu, vložení obsahu přes příkaz a následná kontrola, že byl očekávaný text vložen do dokumentu. Integrační testy tedy pokrývají správnou spolupráci rozšíření s hostitelskou aplikací. Nezahrnují však samotné manuální interakce s UI – akce vyvolávají programově přes API. Některé výsledky jdou obtížně ověřit, pokud je *VS Code* API přímo neposkytuje – např. nelze získat text zobrazený v notifikačním okně, protože *VS Code* API neumožňuje načíst obsah takové notifikace. Podobně není jednoduché ověřit obsah *WebView* nebo stav kontextových nabídek. Tyto mezery pak musí pokrýt testy na úrovni uživatelského rozhraní.

2.1.3 Testy uživatelského rozhraní

Jedná se o *end-to-end* testy simulující skutečné uživatelské scénáře. Rozšíření běží ve vlastní instanci *VS Code* a test automaticky ovládá editor podobně, jako by to dělal uživatel. Kliká na prvky, píše do vstupů, čte zobrazený obsah. Testy uživatelského rozhraní (UI) tedy ověřují celý tok (*flow*) od uživatelské akce (například klik na ikonu rozšíření, stisk klávesové zkratky, výběr položek menu) až po finální odezvu v UI (zobrazení výsledků, otevření panelu atd.). Technicky jde o běh obdobný integračnímu testování, navíc je potřeba ale nástroj, který umí automatizovaně pracovat s DOM stromem[6] a prvky rozhraní aplikace. *VS Code* je postaveno na *Electronu[14] (Chromiu)*[39], takže jeho UI je vnitřně webová stránka. UI testy jsou nejblíže reálnému používání doplňků a umí odhalit chyby, které se v izolovaných podmínkách neprojeví. Takovou chybou může být špatná integrace do menu, nefunkční reakce na kliknutí nebo kolize s jinými částmi rozhraní. Takový scénář by jednotkový ani integrační test nezachytil. Nevýhodou UI testů je jejich jejich složitost a křehkost – musí se vypořádat s tím, že UI *VS Code* je komplexní (např. obsahuje mnoho zanořených vrstev elementů) a může se měnit mezi verzemi. Běh UI testů je pomalejší (start celé aplikace, čekání na vykreslení prvků). Proto se v praxi často počet UI testů omezuje jen na kritické scénáře a mnoho méně zásadních detailů se spoléhá na integrační či manuální testování. Pro zajištění vysoké kvality rozšíření jsou UI testy nezastupitelné – odhalí problémy, které jinak skončí až u koncového uživatele. Moderní přístupy a nástroje (viz další část) se snaží psaní a údržbu UI testů zjednodušit, například pomocí konceptu *Page Objects*[7], a částečně tak zmírnit jejich komplikovanost.

2.2 Existující nástroje a frameworky

Microsoft poskytuje základní podporu pro psaní a spouštění testů prostřednictvím tzv. *Extension Test Runner* utilit. Konkrétně existují moduly `@vscode/test-electron` a `@vscode/test-web`, které dokáží připravit prostředí pro integrační testy v desktopové nebo webové verzi *VS Code*. Tyto nástroje se používají v kombinaci s tradičními testovacími frameworky, přičemž výchozí (a oficiálně doporučovaný) je *Mocha*.

[6] Document Object Model je hierarchická struktura reprezentující *HTML* nebo *XML* dokument, kde každý prvek, atribut a textový obsah je uzel v tomto stromě.

[7] Abstrakcí pro práci s konkrétními částmi *VS Code* UI.

2.2.1 Mocha

Mocha[32] je osvědčený *JavaScriptový* testovací framework, který umožňuje psát testy formou popisných bloků (`describe/it`) a podporuje asynchronní testy, což je důležité pro operace jako spouštění příkazů *VS Code*. Při použití *Yeoman* generátoru kódové kostry rozšíření se automaticky nastaví skript `npm test` využívající *Mocha* a zmíněné `@vscode` testovací moduly k spuštění základních testů. Testy se spouští přes CLI příkaz `vscode-test` (dříve součást balíčku vscode) nebo nově `@vscode/test-cli`, který stáhne potřebný runtime *VS Code* a vykoná uvnitř něj *Mocha* testy. *Mocha* tedy tvoří jádro pro psaní testů – jak jednotkových (běžících čistě v *Node.js*), tak integračních (běžících v *VS Code*). K psaní asercí se běžně využívá knihovna *Chai*.[5] *Mocha* se používá zejména pro testování logiky a chování kódu v *Node.js* nebo ve specifickém prostředí, jakým je *VS Code*. Nejedná se však o framework určený pro testování uživatelského rozhraní. K tomu slouží specializované nástroje, které dokážou simulovat nebo přímo interagovat s prohlížečem (např. *Selenium*[45] či *ExTester*). Tyto nástroje nabízejí pokročilé funkce pro práci s DOM, ověřování prvků a simulaci uživatelských akcí. Naproti tomu *Mocha* zůstává lehkým a flexibilním řešením pro automatizaci testů čistě na aplikační nebo integrační úrovni.

2.2.2 ExTester

ExTester (*vscode-extension-tester*)[41] je nástroj vyvinutý komunitou (konkrétně týmem společnosti *Red Hat*) speciálně pro automatizované testování UI *VS Code* rozšíření. Jedná se o framework postavený nad *Selenium WebDriverem*, který zjednodušuje psaní a spouštění *end-to-end* testů pro doplňky.

Jeho název „*Extension Tester*“ vystihuje účel – umožnit vývojářům spustit UI testy jejich rozšíření snadno jedním příkazem. Implementačně řeší veškerou přípravu prostředí, která by jinak byla na vývojáři: stáhne požadovanou verzi *VS Code*, najde a stáhne odpovídající *ChromeDriver* (podle verze *Chromia* v *Electronu*), spustí novou instanci *VS Code* s testovaným rozšířením (aby se neovlivnila běžná instalace), a pak přes *WebDriver* ovládá UI tohoto *VS Code*. Tím odbourává obrovské množství ruční práce – autoři *ExTesteru* v [42] popisují, že jinak by člověk musel ručně řešit až 7 kroků přípravy (výběr *VS Code* dle OS, nastavení prostředí bez titulkové lišty, instalace rozšíření, atd.) předtím, než by vůbec mohl začít psát testy. *ExTester* tyto kroky skrývá a poskytuje jednoduché rozhraní.

Po instalaci balíčku `vscode-extension-tester` z `npm` stačí napsat testy (v *TypeScriptu*), nakonfigurovat `package.json` a spustit je přes dodaný CLI příkaz `extest`. Framework interně využívá *Mocha*, takže testy se píší pomocí bloků `describe` a `it`, stejně jako v klasických *JavaScript* testech. *ExTester* však poskytuje vysokoúrovňové API (*Page Objects*) pro práci s *VS Code* UI – má předdefinované třídy a metody reprezentující hlavní části rozhraní (např. `SideBar`, `EditorView`, `Workbench`, `InputBox` atd.) a umožňuje např. `workbench.openCommandPalette()` místo ruční simulace stisku kláves. Tím zjednodušuje skripty a zvyšuje odolnost testů vůči změnám (lokátory elementů jsou udržovány v rámci projektu).

Možnosti *ExTesteru* zahrnují například automatické pořízení snímku obrazovky při

selhání testu, podporu různých verzí *VS Code* (včetně *Insiders*[8] buildů), a běh testů v různých prostředích OS (lokálně i v CI prostředí s displejem či virtuálním *framebufferem*).

Stručně shrnuto, *ExTester* je „komplexní řešení“ pro UI testy *VS Code*: „Umožňuje bez námahy spouštět UI testy pro *VS Code* rozšíření se *Selenium WebDriverem* a tím zvyšovat kvalitu uživatelského rozhraní.“. Byl to jeden z prvních nástrojů, který prolomil bariéru obtížné automatizace *VS Code* UI – jak poznamenal jeho autor v [42], před jeho vznikem bylo automatické *end-to-end* testování *VS Code* doplňků považováno za velmi obtížné a takřka neřešené.

Dnes je *ExTester* (*vscode-extension-tester*) dostupný jako open-source (*npm* balíček) a používá ho např. *Red Hat* pro testy svých rozšíření pro *VS Code*. Interně spoléhá na *Mocha* a její dobře známou logiku `describe/it`, což usnadňuje přechod vývojářů, kteří už s *Mocha* testy pracovali.

2.2.3 WebDriverIO

Dalším nástrojem pro *end-to-end* testy *VS Code* rozšíření je integrace do populárního testovacího frameworku *WebdriverIO*[49]. *WebdriverIO* je moderní automatizační framework primárně pro webové aplikace (postavený nad *WebDriver* protokolem, nově podporující i *Chrome DevTools protokol*) a nabízí mnoho rozšiřujících servis.

Jedna z nich, `wdio-vscode-service`, byla představena v roce 2022 a umožňuje psát *VS Code* UI testy v ekosystému *WebdriverIO*. Principiálně se podobá *ExTesteru* – také dokáže automaticky nainstalovat požadovanou verzi *VS Code*, stáhnout odpovídající *Chromedriver* a spustit *VS Code* s testovaným rozšířením. Navíc umožňuje přístup k *VS Code* API přímo z testů (což je zajímavá vlastnost – test tak může kombinovat ovládání UI s voláním interních API). *WebdriverIO VS Code service* rovněž umí spustit *VS Code* jako webovou aplikaci na serveru, což se hodí pro testování webových verzí rozšíření (v prohlížeči). Dále automaticky generuje *page objecty* s lokátory pro danou verzi *VS Code* – v tom se inspiroval právě projektem *ExTester*, od kterého přebírá myšlenku a přizpůsobuje ji do světa *WebdriverIO*.

Výhodou integrace do *WebdriverIO* je, že vývojář může využít veškerý ekosystém kolem (reportování, integrace s Cloud službami jako *BrowserStack*, psaní testů v jazyce dle výběru – *TypeScript*, *Node.js*, dokonce možnost *BDD*[9] syntaxe přes *Cucumber*[10], atd.). Christian Bromann (autor *WebdriverIO*) ve svém blogu [5] doporučuje investovat do *end-to-end* testů a zmiňuje, že s novou *VS Code service* je možné *VS Code* automatizovat jako každou jinou *Electron*/web aplikaci. *WebdriverIO* dokáže pokrýt i mezery oficiálních nástrojů – například testování `webview` panelů, které oficiální *VS Code test runner* nepodporuje, je s pomocí UI automatizace (*Selenium/DevTools*) možné. Díky tomu lze otestovat

[8] Vývojářské verze *VS Code*, které obsahují nejnovější funkce ještě před jejich oficiálním vydáním.

[9] Behavior-Driven Development je metodika vývoje softwaru, která definuje chování aplikace pomocí scénářů psaných srozumitelným jazykem z pohledu uživatele.

[10] Nástroj, který umožňuje psát a automaticky spouštět testovací scénáře metodikou BDD pomocí srozumitelného přirozeného jazyka.

i rozšíření, která mají vlastní UI ve `webview` (např. nástěnky, grafy, formuláře) – test může otevřít `webview` a použít selektory v jeho DOM ke kontrole obsahu.

WebdriverIO VS Code service je tedy alternativou k *ExTesteru* – oba nástroje mají podobný cíl a funkce. Volba může záviset na preferencích: pokud někdo již používá *WebdriverIO*, může snadno zařadit testy *VS Code* do svého testovacího běhu. *ExTester* na druhou stranu představuje standalone řešení cílené přímo na *VS Code* a nepotřebuje další rámec.

2.2.4 Další přístupy a nástroje

Kromě výše zmíněných existují i jiné cesty, jak automatizovat testy *VS Code* rozšíření. Například někteří vývojáři experimentovali s *Cypress.io*[10] – oblíbeným nástrojem pro *end-to-end* testy webových aplikací. *VS Code* jako takový sice není čistě webová aplikace (běží v *Electronu*), ale projekt *code-server*[8] umožňuje provozovat *VS Code* jako webovou aplikaci v prohlížeči. Juan Manuel Allo v roce 2020 ukázal, že je možné spustit *code-server* s testovaným rozšířením uvnitř *Dockeru*[12] a poté pomocí *Cypress* simulovat interakce v UI *VS Code* na webu. Tímto způsobem napsal funkcionální testy (BDD styl) blízké akceptačním kritériím pro své rozšíření. Tento přístup vyžaduje více infrastruktury (běh *VS Code* na serveru, integrace s *Cypress*), ale přináší možnost využít bohaté rozhraní *Cypressu* a jeho časovou osu testů. Podobně lze teoreticky použít i *Playwright*[35] (framework od *Microsoftu* pro automatizaci prohlížečů) k ovládání *VS Code – Playwright* dokáže spouštět i desktopové aplikace v *Electronu*. V komunitě se to zatím více ujalo u projektu *Eclipse Theia* (open-source IDE založené na principu *VS Code*), kde běží UI testy právě na *Playwrightu*. To dokazuje, že moderní headless prohlížečové nástroje jsou schopné UI *VS Code* (nebo jemu podobné) ovládat, a že volba konkrétního nástroje může vycházet z preferencí týmu. Ještě starším přístupem, používaným dříve u *Electron* aplikací, byl *Spectron* (framework postavený na *WebDriverIO* určený speciálně pro *Electron*). *Spectron* uměl spustit *Electron* aplikaci a ovládat ji, a existovaly pokusy jej využít i pro *VS Code* doplňky. Nicméně *Spectron* byl oficiálně ukončen a nahrazen právě modernějšími přístupy (např. kombinací *Playwright/DevTools*).

3 Jak by se mělo testovat

3.1 Návrhy v literatuře a doporučení expertů

V odborné literatuře a praxí mezi vývojáři panuje shoda na několika postupech, které v obecné rovině usnadňují testování rozšíření *VS Code* na úrovni uživatelského rozhraní.

Prvním doporučením je navrhovat rozšíření s ohledem na jeho testovatelnost. To zahrnuje oddělení aplikační logiky od závislostí na *VS Code* API. *VS Code* API neposkytuje žádnou přímou podporu pro čisté jednotkové testy rozšíření. Jako vhodné řešení je možné zabalení volání *VS Code* API pomocí vlastních tříd. [3] Díky tomu lze tyto třídy snadno nahradit (mockovat) a izolovat testovat logiku rozšíření izolovaně od samostatného *VS Code*. V těchto testech je vhodné simulovat události a vstupy programově (například vyvolat falešnou událost místo skutečné interakce) a ověřit reakce systému, čímž se sníží potřeba komplexního spouštění celého *VS Code*. [11]

Pro samotné integrační UI testy se doporučuje zaměřit na hlavní uživatelské scénáře. Není nutné vyčerpávajícím způsobem automatizovat každý okrajový případ, místo toho je vhodné testovat především klíčové workflow, které uživatelé v rozšíření provádějí. [11] Tedy pokud dané rozšíření přidává do nový příkaz do palety příkazů (*command palette*), který otevírá například otevírá nový panel, test by měl ověřit tento primární tok – vyvolání a otevření panelu. Drobné detaily či okrajové situace je efektivnější pokrýt jinou formou. Tento *risk-based*[11] přístup zohledňuje nákladnost UI testů, které jsou pomalé a neúčinné na údržbu. [11]

Z literatury rovněž plyne důraz na využití vysokoúrovňových abstrakcí pro UI testy jako jsou například *Page Object* modely. Framework *ExTester* posykutje připravené objekty reprezentující hlavní částí *VS Code* UI jako jsou u postranní bannery, editory, příkazové paleta a jiné, což výrazně zjednodušuje psaní testů. [40] Místo pracného prohledávání DOM stromů a simulace kliknutí či stisků kláves může tester volat připravené metody. Toto abstrahované API vede k vyšší odolnosti testů vůči změnám UI – selektory prvků jsou udržovány v rámci frameworku. Obecně lze danou vlastnost označit jako znovupoužitelnost komponent.

Integrace do vývojového cyklu je další best pracite. To znamená pravidelné spouštění testů lokálně během vývoje a zároveň jejich automatické spouštění během c*ontinous integration pipeline*. *Microsoft* poskytuje oficiální *Test Runner* (balíček `@vscode/test-electron` a CLI `vscode-test`) pro snadné spouštění testů na různých platformách.[25] Tento nástroj lze využit v CI, například prostřednictvím *Azure Pipelines* které podporují paralelní běh testů na *Windows*, *Linux* i *macOS*. K dispozici jsou také ukázkové konfigurace. UI testy je vhodné v rámci CI spouštět jen při určitých podmínkách, protože mohou brzdit rychlé prototypování.

Využití AI jako asistenta testování lze chápat jako *emerging best practice*. například v [21] doporučují nasadit generativní AI pro urychlení tvorby testů, ale ponechat člověka v roli kontrolora. Jejich studie ukázala, že pokud tester měl k dispozici jasně popsané

[11] Přístup k řízení, kdy jsou priority a rozhodnutí určovány na základě posouzení rizik.

scénáře (v podobě *Gherkin* specifikace) a použil nástroje jako *ChatGPT* k vygenerování testovacího skriptu, došlo vždy ke zrychlení vývoje testů – v některých případech i statisticky významnému. [21] Zároveň však platí, že AI generované testy je nutné revidovat a dopracovat. Tester s dostatečnými zkušenostmi by měl výsledný kód projít, upravit nepřesnosti a doplnit chybějící kontrolní body. [21] Best practice tedy je brát výstupy AI jako návrh či draft, který ušetří část rutinní práce, ale finální zodpovědnost za kvalitu testu nese vývojář. Podobně [4] ve svém experimentu s *ChatGPT* konstatují, že více než třetina asercí vygenerovaných modelem byla nesprávných. [4] Jako doporučení se ukazuje i tzv. prompt engineering – správné formulování vstupů pro AI. Pokud je prompt dostatečně konkrétní a zahrnuje požadavky na okrajové případy, model dokáže pokrytí kódu významně zlepšit. [4]

3.2 Problémy stávajícího přístupu

Navzdory uvedeným doporučením je testování *VS Code* rozšíření v současné praxi stále problematické. Nejzásadnějším problémem je nutnost ručního psaní testů, absence nástrojů pro automatické generování testovací kostry a slabá standardizace napříč CI/CD procesy.

3.2.1 Ruční psaní testů

Ruční psaní testů je neefektivní a náchylné k opomenutím. Tvorba testů vyžaduje čas a často bývá vývojáři odsouvána či zanedbávána. Na rozdíl od produkčního kódu, který vývojář píše s jasným cílem funkčnosti, psaní testů může být vnímáno jako dodatečná zátěž neprodukující nové funkce. V prostředí *VS Code* rozšíření se navíc tester potýká s komplikacemi popsanými výše (nutnost *mockování*, nastavování *VS Code* hostu atd.), což ještě zvyšuje bariéru. Mnoho open-source rozšíření proto obsahuje jen triviální demo testy vygenerované *scaffoldingem* nebo vůbec žádné. [3]

3.2.2 Chybějící nástroje pro generování testovací kostry z kódu

V jiných ekosystémech existuje vyšší míra podpory při vytváření testů – například frameworky generující výchozí testovací třídy či utility, které dle funkcí připraví kostru testů. Pro *TypeScript* a *VS Code* žádné podobné rozšíření neexistovalo. Vývojář je tak nucen projít celý kód rozšíření a ručně si vytipovat co a jak otestovat. U rozsáhlejších pluginů využívajících více komponent tento přístup představuje slabé místo. Tato mezera začíná být vyplňována nástroji využívajícím AI. Příkladem je třeba *GitHub Copilot*, který umí na základě zdrojového kódu navrhnout sadu testů.[28] Ačkoliv jde o velký posun, jedná se o relativně novou technologii a ne každý projekt ji využívá do své workflow. Mnoho vývojářů navíc váhá nad důvěrou dané technologii, viz výše. Specializované generátory testů existují převážně pro jiné jazyky, například *Pynguin* pro *Python* nebo *EvoSuite* pro *Javu*. Pro oblast *VS Code* UI testování zatím chybí nástroj, který by vývojářům usnadnil generování kostry a počátek testování,

3.2.3 Slabá standardizace testování UI v CI/CD pipelines

Po vytvoření automatizovaných UI testů je vhodná integrace do nástrojů kontinuálního sestavení a nasazení. Hlavní komplikací je, že UI testy vyžadují spuštění grafického prostředí *VS Code*. Na serverech pro CI je tedy nutné emulovat prostředí a zajistit dostupnost potřebných závislostí.[25] Neexistuje jednotný postup pro integraci na všech CI službách. V praxi je díky tomu integrace UI testů často odložena nebo se spouští pouze v omezeném množství. Chybí zde standard podobný jednotkovým testům, kde je například `npm test` funkční všude. Nedostatek standardizace byl identifikován i v širším kontextu – studie uvádějí, že v průmyslové praxi chybí dostatek souhrnných důkazů podporujících přijetí automatického generování testů, což zpomaluje jeho rozšíření. [20] Než se UI testy stanou samozřejmou součástí CI/CD, je třeba překonat technické překážky a vytvořit vzorová řešení, která komunita přijme. Dokud se tak nestane, zůstane implementace UI testů do *pipeline* každý řešit po svém, s různou mírou úspěšnosti.

3.3 Které kroky lze automatizovat a jaká jsou omezení

Pokroky v oblasti umělé inteligence – zejména velkých jazykových modelů (LLM) typu *GPT-4* – otevírají nové možnosti, jak automatizovat vybrané části procesu testování. Na základě výše popsaných mezer identifikujeme několik oblastí, kde by AI mohla výrazně pomoci, ale zároveň vymezujeme její limity.

3.3.1 Generování testovacího kódu ze zdrojového kódu rozšíření

LLM modely dokážou analyzovat zdrojový kód a navrhnout pro něj testy v přirozeném jazyce či přímo ve formě kódu. *GitHub Copilot* či podobné nástroje již umí vygenerovat základní testy (včetně asercí) pro danou funkci nebo třídu.[26] V kontextu *VS Code* rozšíření by AI mohla automaticky vytvořit testy například pro každou veřejnou příkazovou funkci rozšíření, s různými scénáři volání. To zahrnuje navržení vstupních parametrů a očekávaných výstupů. Výhodou je zrychlení – AI zvládne navrhnout desítky testů během pár vteřin, zatímco člověku by to trvalo mnohonásobně déle. [20]

3.3.2 Automatizace tvorby uživatelských scénářů (sekvencí akcí) v UI

lze využít k vygenerování *end-to-end* scénářů na základě slovního popisu funkčnosti. Například vývojář popíše: *"Otevři VS Code, nainstaluj mé rozšíření, stiskni Ctrl+Shift+P, z nabídky vyber příkaz X, vlož text Y do vstupního pole a ověř, že se zobrazí oznámení Z."* a model (jako *ChatGPT*) navrhne skript v *ExTester* syntaxi provádějící tyto kroky. V [21] ukázali, že z dobře definovaných scénářů v jazyce *Gherkin* dokáže *ChatGPT* vygenerovat odpovídající testovací skripty (pro webové aplikace) a ušetřit tím testerovi práci. Obdobný princip by šel aplikovat na *VS Code*: AI převede specifikaci scénáře na kód volající příslušné *Page Object* metody *ExTesteru*. To by zautomatizovalo nejnáročnější část – sestavení sekvence uživatelských interakcí. Navíc AI může navrhnout i varianty scénáře, čímž generuje další testy.

3.3.3 Analýza pokrytí a návrh doplňujících testů

AI může asistovat při tzv. *gap analysis* testů – projde stávající testy a kód a identifikuje části kódu, které nebyly volány nebo podmínky, které nebyly ověřeny. Následně navrhne testy zaměřené právě na tyto mezery. Například pokud rozšíření obsahuje příkaz, který za určitých podmínek loguje varování do konzole, a žádný test dosud neověřil tuto větev, AI může vygenerovat test simulující onu podmínku a kontrolující výskyt varování. Výzkum v této oblasti naznačuje, že kombinace různých nástrojů může pokrytí výrazně zlepšit – v [4] zjistili, že *ChatGPT* a tradiční generátor *Pynguin* mají jen minimální překryv v neotestovaných částech kódu, takže jejich kombinací lze dosáhnout lepšího společného pokrytí. [4]

3.3.4 Automatická aktualizace testů při změně kódu.

Pokud dojde ke změně rozhraní či chování rozšíření, AI by teoreticky mohla upravit existující testy namísto vývojáře. Například když se změní text zprávy, kterou rozšíření zobrazuje, AI může při porovnání staré a nové verze kódu zrefaktorovat i očekávání v testech. Tato oblast zatím představuje spíše možnou oblast budoucího vývoje.

3.4 Limitace a výzvy AI pro generování testů

Navzdory lákavým možnostem má nasazení AI v testování i svá omezení. Zaprvé, AI postrádá spolehlivé porozumění záměru kódu. Generované testy mohou syntakticky vypadat dobře, ale není zaručeno, že ověřují skutečně to, co autor zamýšlel. Studie ukázaly, že značná část asercí odvozených čistě z modelu může být nesprávná nebo nelogická. [4] Model totiž nemusí správně pochopit obchodní logiku – například může očekávat jinou návratovou hodnotu funkce, než jaká je správná, protože se opírá jen o pravděpodobný vzor v datech, ne o skutečný požadavek. V kontextu V*S Code UI* je problém ještě zvýrazněn tím, že některé správné výstupy testů nejsou explicitně v kódu – např. správné zobrazení nějakého prvku v UI. LLM může takový test navrhnout, ale nemůže ho úspěšně vyhodnotit.

Další limitací je omezená schopnost AI provádět komplikované výpočty a dedukce při generování testů. V [22] zjistili, že současné špičkové modely (včetně *GPT-4*) selhávají, pokud má test zahrnovat složitější logické odvození nebo více kroků, protože model nemá interní paměť stavu a nedokáže spolehlivě provádět *multi-step* výpočty. [22] To je relevantní například pro testy, které by ověřovaly sekvence akcí závislých na předchozím kontextu (např. otevření jednoho okna mění obsah jiného). Model může vytvořit test, který přehlédne nějakou nezbytnou inicializaci nebo kontext.

Související problém je neznalost interního stavu aplikace při generování očekávání. AI může navrhnout, že po provedení akce X má být ve zvolené proměnné hodnota Y, ale pokud je tato hodnota výsledkem komplikovaného výpočtu nebo volání API, model to spíše odhadne, než aby měl jistotu. V praxi to vede k testům, které projdou syntaktickou kontrolou, ale při spuštění selžou kvůli nesouladu očekávaných výsledků. [22] Leckteré nástroje se snaží tuto mezeru řešit zapojením runtime feedbacku – například framework

TestChain používá víceagentní přístup, kdy nechá LLM vygenerovat test, spustí ho v *Python* interpretru a na základě skutečného výstupu upraví očekávání. Tím se výrazně zvýší přesnost asercí (v případě *GPT-4* o téměř 14 % na obtížných úlohách). [22] Takový přístup kombinuje AI s tradičním vykonáním kódu, což je slibné. Nicméně v prostředí *VS Code* by obdobný mechanismus byl složitější, neboť zahrnuje spuštění celého editoru a pluginu v automatu a interpretace výsledků.

AI má také omezení kontextu a škálovatelnosti. LLM mají limit, kolik kódu lze naráz zpracovat (tzv. *context window*). U většího rozšíření nemusí být možné nahrát celý projekt do modelu a nechat si vygenerovat kompletní sadu testů. Muselo by se postupovat po částech, což může opomenout interakce mezi komponentami. Dále, pokud se rozšíření výrazně změní, AI generování by se muselo znovu aplikovat – dnešní nástroje neumějí inkrementální generování.[12]. To vede buď k přegenerování (a možnému vytvoření duplicit či nekonzistence), nebo k nutnosti ruční údržby stejně jako u klasických testů.

Nakonec je tu aspekt integrace a přijatelnosti. Zavedení AI do procesu testování vyžaduje, aby vývojáři změnili svou pracovní rutinu, učili se nové nástroje a důvěřovali jim. Dokud nebude jasně prokázáno, že AI generované testy skutečně zvyšují kvalitu softwaru, mnozí mohou váhat je plně přijmou. [20]

Současným limitem tedy není jen technologie samotná, ale i lidský faktor – ochota spolehnout se na umělou inteligenci v kritické části vývoje, jakou je testování.

[12] Doplnění testů pouze pro nově přidaný kód.

4 Existující projekty a řešení

4.1 Přehled již existujících projektů

4.1.1 ChatGPT

ChatGPT[34] je pokročilý velký jazykový model vyvinutý společností *OpenAI*, který dokáže na základě přirozeného jazyka generovat kód a to včetně testů. Model převádí textové popisy požadavků do konkrétních testovacích skriptů, což umožňuje rychlé prototypování. Jeho schopnost analyzovat kontext a navrhovat odpovídající aserce výrazně urychluje počáteční fázi testování. Výstupy jsou však generovány na základě pravděpodobnostních modelů, a proto se občas objeví tzv. halucinace. Navržené testy často slouží jako základní návrh, který je nutné dále revidovat a upravit. Díky rozsáhlému tréninku na různorodých datech je *ChatGPT* schopen navrhnout testy pro široké spektrum scénářů, včetně neobvyklých *edge-case*[13] situací. Jeho nedostatkem je nedeterministický výstup, což znamená, že stejný vstup může vést k odlišným výsledkům.

4.1.2 GitHub CoPilot

GitHub Copilot[17] je AI asistent integrovaný do vývojových prostředí, například ve *VS Code*, a využívá model *OpenAI Codex* pro generování kódu v reálném čase. Sleduje kontext aktuálního souboru a na základě komentářů či pojmenování funkcí poskytuje návrhy, včetně návrhů testů. Díky tomu výrazně zrychluje tvorbu testovací kostry a eliminuje nutnost psaní boilerplate[14] kódu. *Copilot* je schopen automaticky navrhnout strukturu testovacích tříd, importy a dokonce i aserce, což usnadňuje rutinní práci vývojářů. Je kompatibilní s řadou programovacích jazyků a testovacích frameworků, což z něj činí univerzální nástroj. Nicméně generované návrhy je často třeba manuálně ověřit a doladit, neboť se může stát, že kód sice syntakticky vyhovuje, ale logicky neodpovídá potřebám projektu. Interaktivní integrace do IDE umožňuje vývojářům snadno přizpůsobit navržené řešení.

4.1.3 Pynguin

Pynguin[44] je nástroj určený pro automatizované generování unit testů v *Pythonu*. Využívá algoritmy k prohledávání možných vstupních hodnot. Automaticky vytváří sekvence volání funkcí s cílem maximalizovat pokrytí testovaného kódu, čímž odhaluje různé *edge-case* situace. Generované testy slouží zejména jako regresní testy, kde se aktuální výstupy uloží jako očekávané hodnoty. Díky systematickému přístupu dokáže *Pynguin* rychle vytvořit rozsáhlou sadu testů, což výrazně šetří čas v porovnání s ručním psaním. Výstupy však mohou mít problémy s čitelností a údržbou, neboť generované sekvence volání ne-

[13] Situace je výjimečný případ, kdy se systém chová neočekávaně při extrémních nebo neobvyklých vstupních hodnotách.

[14] Také šablonový kód nebo rutinní kód, označuje opakující se části kódu v různých souborech nebo projektech, přestože jsou stejné nebo téměř stejné.

musí být intuitivní. Pokud je v programu skrytá chyba, může se stát, že *Pynguin* zachytí nesprávné chování a vloží jej do testů, což vyžaduje následnou revizi. Jeho využití je tedy vhodné zejména v projektech, kde je třeba rychle vytvořit základní testovací sadu, kterou následně doplní vývojář.

4.1.4 EvoSuite

EvoSuite[15] je pokročilý automatizovaný generátor testů pro *Javu*, který využívá evoluční algoritmy k vytváření *JUnit* testů. Automaticky generuje testovací sady, které systematicky prozkoumávají různé větve kódu, a tím maximalizují pokrytí testů. Generované aserce vycházejí z aktuálního chování testovaného kódu, což umožňuje detekci regresních chyb při budoucích změnách. Nástroj zlepšuje kvalitu testů tím, že využívá evoluční techniky k optimalizaci celé testovací sady najednou. *EvoSuite* je schopen odhalit chyby i v okrajových případech, čímž zvyšuje robustnost testování. Generované testy mohou být neintuitivní a obtížně čitelné, což znesnadňuje jejich údržbu a případné opravy.

4.1.5 Další nástroje

Kromě výše popsaných nástrojů existuje řada dalších řešení, jako jsou *Tabnine*[46], *CodeWhisperer*[1], *Randoop*[37], *TestForge*[48] a *CodiumAI*[9], které rovněž přispívají k automatizaci generování testů. Tyto nástroje se liší v metodách generování – od kontextového doplňování kódu až po evoluční a *random-based* techniky. Nabízejí alternativní přístupy a mohou být využity jako doplněk k hlavním nástrojům.

4.2 Analýza dostupných řešení

Při pohledu na dostupné nástroje pro generování a automatizaci testů je patrné rozdělení na otevřené (open-source) projekty a komerční produkty. Obě kategorie mají své silné i slabé stránky.

4.2.1 Open-source řešení

Většina akademických nástrojů a komunitních frameworků je dostupná jako open-source. Patří sem *ExTester*, *wdio-vscode-service*, *Pynguin*, *EvoSuite*, *Randoop* a další. Jejich výhodou je volná dostupnost a často aktivní komunita, která je vylepšuje. Například *ExTester* je vyvíjen otevřeně pod hlavičkou *Red Hat* a vývojáři rozšíření mohou přispívat vlastními vylepšeními nebo reportovat problémy přímo na *GitHubu*. Podobně nástroje jako *EvoSuite* jsou výsledkem akademického výzkumu, mají publikované zdrojové kódy a lze je upravovat či integrovat dle potřeby. Pro uživatele to znamená nulové licenční náklady a možnost přizpůsobení nástroje specifickým požadavkům. Na druhou stranu open-source projekty nemusejí poskytovat profesionální podporu. U specializovaných generátorů testů se také stává, že po skončení projektu ztratí aktivní údržbu.

V kontextu *VS Code* rozšíření je drtivá většina nástrojů open-source (v souladu s duchem *VS Code* platformy). *ExTester* i *wdio-vscode-service* integrace jsou volně k dispozici.

I nové AI rozšíření jako *TestForge* či *CodiumAI* lze z Marketplace[27] nainstalovat zdarma.

Obecně open-source nabízí možnost transparentnosti – u generovaných testů může být důležité vědět, jak byly vytvořeny, co přesně dělají. U otevřených nástrojů lze auditovat jejich logiku (např. u *Pynguin* si lze prohlédnout, jak generuje vstupy). To může zvýšit důvěru v jejich používání.

4.2.2 Komerční řešení

Komerční sféra začala nabízet „AI for testing" relativně nedávno. Patří sem hlavně *GitHub Copilot*[17], dále enterprise řešení[15] jako *Diffblue Cover* pro *Javu* nebo služby některých startupů (např. již zmíněný *CodiumAI*). Jejich hlavním tahákem je uživatelská přívětivost a integrace do firemních procesů. *GitHub Copilot* je např. integrován do *VS Code* a podporuje řadu jazyků bez nutnosti cokoli nastavovat. Za cenu poplatku získá vývojář nepřetržitou asistenci AI. Komerční nástroje často také nabízejí oficiální podporu. To je důležité zejména pro firmy, které chtějí spoléhat na automatizované testování u velkých projektů – potřebují garance.

Na druhou stranu, komerční řešení mohou být uzavřená (*closed-source*), což přináší otázky ohledně bezpečnosti a transparentnosti. Například *GitHub Copilot* využívá kód uživatele k vytvoření návrhů a tyto informace procházejí skrz cloud *OpenAI*. Některé společnosti mohou váhat s nasazením takového nástroje kvůli obavám o únik citlivého kódu. Také generované testy zůstávají černou skříňkou – není vidět, na základě jaké logiky byly navrženy. V open-source komunitě se to řeší recenzemi a sdílením zkušeností, u komerčních je nutno věřit dodavateli.

4.2.3 Srovnání

Z hlediska kvality se nedá jednoznačně říct, která skupina vede. Například vygenerované testy z *GitHub Copilot* mohou působit velmi elegantně, protože model se učil z lidsky psaných testů – tedy komerční AI může produkovat čitelnější testy. Pozoruhodný je i ekonomický aspekt: open-source nástroj může firmě ušetřit náklady za licence, ale může vyžadovat investice do zaškolení a údržby *in-house*. Komerční nástroj má přímý finanční náklad, ale slibuje úsporu času vývojářů – je tedy na konkrétní organizaci zvážit, co se vyplatí.

4.3 Inspirace z jiných oblastí

Automatické generování testů není zdaleka problémem unikátním pro *VS Code* či *TypeScript*. Lze se poučit z nástrojů a postupů v jiných jazycích a prostředích a zvážit jejich adaptaci pro náš kontext.

[15] Nástroje nebo služby cíleně vyvíjeny a nabízeny s ohledem na potřeby velkého firemního prostřední.

4.3.1 Java a staticky typované jazyky

V *Javě* existuje bohatá tradice nástrojů pro generování testů (*EvoSuite*, *Randoop* aj.). Tyto nástroje těží z toho, že jazyk je staticky typovaný a reflektivní – lze programově získat seznam tříd, metod, jejich parametrů a návratových hodnot, a automat pak zkouší volat metody s různými hodnotami. Podobný princip by šel aplikovat v *TypeScriptu*, který je též typovaný (byť za běhu se typová informace ztrácí). Při kompilaci či prostřednictvím *TypeScript Compiler API*[29] by však bylo možné získat strukturu kódu rozšíření: jaké příkazy registruje, jaké exportuje funkce, jaké má závislosti. Takový model může sloužit jako základ pro generátor testů – analogicky k *EvoSuite* by mohl zkoušet volat každou funkci s různými variantami argumentů (vygenerovanými např. z typů).

4.3.2 Model-based testing a genrování scénářů

V některých doménách (zejm. u GUI aplikací a webů) se osvědčily modelově řízené přístupy. Například nástroj *Skyfire* generuje z modelu aplikace (stavového diagramu) automaticky scénáře v jazyce *Gherkin* pro testování webové aplikace, čímž nahrazuje ručně psané scénáře. [20] Pro rozšíření *VS Code* by analogicky šlo vytvořit model interakce uživatele s rozšířením – např. stavový automat, kde stavy reprezentují určité konfigurace editoru a přechody akce uživatele (otevření příkazu, úprava textu, uložení souboru atd.). Generátor testů by se pak pokusil automaticky projít všechny dosažitelné sekvence akcí. Tím by vznikly testovací scénáře, které by mohly odhalit nečekané kombinace událostí vedoucí k chybám.

4.3.3 Dynamická analýza a fuzzing

U jazyků jako *JavaScript*/*TypeScript*, které umožňují dynamičtější přístup, by mohl být efektivní také *fuzzing* – náhodné či generativní zkoušení různých vstupů. Například pokud rozšíření reaguje na určité typy souborů nebo obsah souboru, mohl by *fuzz* tester generovat různé varianty těchto souborů a kontrolovat, zda rozšíření nevyhazuje runtime chyby.

4.3.4 Adaptace do prostředí VS Code a TypeScript

Adaptace výše zmíněných technik na *VS Code* rozšíření naráží na unikátní aspekty: rozšíření běží v hostovaném prostředí, komunikují s jinými částmi *VS Code* a často pracují asynchronně. Jakákoli automatizace musí umět instancovat toto prostředí a monitorovat je – což je přesně to, co dělá *ExTester* či *vscode-test*. Proto se nabízí cesta rozšířit stávající frameworky o AI funkce. Například zabudovat do *ExTesteru* modul, který přečte manifest[16] rozšíření a pomocí AI vygeneruje testovací kód pokrývající hlavní příkazy.

[16] Soubor `package.json`, který popisuje *metadata* rozšíření pro *VS Code*, jako jsou příkazy, aktivace nebo nastavení.

4.3.5 Spojení s dokumentací

Některé nástroje umí z komentářů či specifikací odvodit testy. Pokud by autoři rozšíření psali např. ve `README` souboru příklady použití jejich rozšíření (např. sekvence kroků, co udělat a co se má stát), AI by mohla takové příklady transformovat na automatizované testy. Tím by se využila stávající znalost domény v lidsky srozumitelné formě.

5 Využití AI v testování: přínosy, rizika a kritika

5.1 Přínosy využití AI

5.1.1 Rychlejší prototypování a tvorba testů

Zapojením AI lze výrazně zrychlit počáteční fáze tvorby testů. Generativní modely dokáží navrhnout základní testy pro daný kód na základě popisu požadavků a vývojář tedy místo zdlouhavého ručního psaní každého testovacího případu zvlášť může získat návrh celé sady testů během krátké chvíle. To podporuje rychlé prototypování – vývojář si nechá vygenerovat sadu testů a okamžitě je může spustit proti aplikaci, aby získal představu o aktuálním pokrytí a potenciálních chybách. V [21] empiricky ověřili, že použití AI při vytváření testovacích skriptů zkracuje dobu vývoje testů oproti čistě manuálnímu psaní. Rychlé prototypování testů se hodí zejména v agilním vývoji – AI vygeneruje hrubé testy paralelně s vývojem funkce, takže ještě před dokončením funkčnosti lze testy spustit a iterovat nad obojím současně. Důsledkem je zkrácení cyklu vývoj-testování, což pomáhá udržovat vysoké tempo vývoje bez kompromisů v kvalitě.

5.1.2 Zkrácení doby potřebné pro psaní základní testovací kostry

AI výrazně urychluje psaní *boilerplate* kódu testů. Mnoho testů má podobnou strukturu – například inicializace testovaného objektu, volání metody s určitými parametry a ověření výsledku. Tyto opakující se vzory může generativní nástroj obstarat automaticky, čímž vývojáři ušetří rutinní práci. *GitHub Copilot* je schopný doplňovat testovací kód na základě kontextu – když vývojář začne psát test, AI navrhne pokračování (např. při otevření závorky metody `testXYZ` ihned nabídne několik řádků volání a *asercí*). AI umí generovat i celé opakující se bloky: pokud například potřebujeme obdobnou sadu testů pro různé vstupy, stačí naznačit první a model nabídne další varianty. Tím odpadá práce „*copy-paste a uprav*“, kterou by vývojář opakovaně vykonával. Zrychlení se netýká jen samotného psaní, ale i nastavování testovacího prostředí: AI může automaticky vložit potřebné importy a výchozí konfigurace aniž by tester musel vše dohledávat. V důsledku se programátor může soustředit na logiku testů, místo aby ztrácel čas technickými detaily.

5.1.3 Možnost objevování neočekávaných scénářů testování

AI může přispět k nalezení *edge case* situací, které by lidský tester nemusel předem promyslet. Generativní model totiž není omezen provozní slepotou či zažitými postupy – navrhuje testy podle vzorů z trénovacích dat, v nichž mohou být zahrnuty i méně obvyklé případy. Například *ChatGPT* může k funkci pro výpočet nějaké hodnoty navrhnout test s extrémním vstupem, protože rozpozná, že takové situace by mohly nastat, i když na ně vývojář nepomyslel. Takové nepředvídané kombinace vstupů mohou odhalit chyby robustnosti. Automatizované nástroje (jako evoluční testování či random testing) zase prozkoumávají prostor vstupů systematičtěji a často narazí na scénáře, které lidé nezkouší – například volání metod ve zvláštním pořadí, konkurenční přístup, hraniční velikosti

struktur apod. Studie kombinující AI a klasické generování testů ukazují zajímavý efekt: různé techniky nacházejí odlišné scénáře. V [4] zjtili, že testy generované *ChatGPT* a testy od nástroje *Pynguin* měly jen malý překryv – každý našel jiné části kódu, které druhý nepokryl. To vede k tomu, že AI doplňuje tradiční přístupy a ve společné kombinaci pak pokryjí širší spektrum situací. Nasazením AI tak lze objevit i neočekávané chyby: například generativní model může vymyslet test, který volá API sekvencí kroků, již vývojář nepředpokládal, a ta odhalí skrytou chybu. Dalším příkladem je využití AI pro generování mutací vstupních dat – model může navrhnout bizarní, ale syntakticky platné vstupy, které odhalí nedostatky ve validaci.

5.2 Rizika a důvody proč se AI někdy odmítá

5.2.1 Algoritmicky správné, přeložitelné, ale nevalidní testy

Generativní AI modely mají tendenci vytvářet testovací případy, které jsou na první pohled algoritmicky správné a syntakticky přeložitelné, ale při bližším zkoumání nevalidní, protože testují nesprávné věci. Tento problém, často označovaný jako halucinace, znamená, že model generuje kód, který vypadá věrohodně, ale jeho logická podstata je chybná. Například AI může navrhnout test, který ověřuje neexistující funkcionalitu, volá metodu API, jež v dané verzi softwaru není dostupná, nebo zcela ignoruje klíčové hraniční případy. Takové testy mohou projít kompilací a dokonce úspěšně běžet, ale ve skutečnosti neodhalí chyby v kódu, protože jsou založeny na nesprávných předpokladech. Výsledkem je falešný pocit bezpečí – test prochází, ale chyba v aplikaci zůstává skrytá, protože testuje špatnou podmínku.

Tento typ nekonzistence se projevuje zejména při komplexnějších úlohách, kde je potřeba více kroků logického uvažování. Výsledky [22] ukazují, že i špičkové modely, jako *GPT-4*, selhávají při generování bezchybných testů, pokud úloha vyžaduje hlubší pochopení kontextu nebo složitější výpočty. Například model může vytvořit test, který vypadá konzistentně, ale ve skutečnosti ověřuje nepodstatný aspekt systému nebo zcela opomíjí kritické scénáře. Dalším problémem je nedostatek reprodukovatelnosti – mírná změna ve formulaci požadavku může vést k vygenerování odlišného testu, který je stejně nevalidní, ale jiným způsobem. Tato nestabilita znamená, že tester nemůže spoléhat na konzistentní kvalitu výstupů a každý test musí být důkladně analyzován. V praxi to může vést k odmítání AI, protože validace a oprava takových testů může být časově náročnější než jejich ruční vytvoření.

5.2.2 Validní logická struktura testů se syntaktickými chybami

Druhou kategorií problémů při generování testů pomocí AI je situace, kdy testy mají správnou strukturu a zaměřují se na validní aspekty systému, ale jsou nepřeložitelné kvůli syntaktickým chybám. Tyto testy jsou logicky smysluplné – například správně ověřují očekávané chování systému, pokrývají klíčové scénáře nebo zohledňují hraniční případy – ale kvůli chybám v syntaxi je nelze spustit. Příkladem může být použití neexistujících funkcí,

špatně definovaných proměnných, nesprávné syntaxe pro daný programovací jazyk nebo odkazy na knihovny, které nejsou v projektu dostupné.

Tento problém pramení z toho, že generativní AI modely nevykonávají kód, který vytvářejí, a nemají tedy zpětnou vazbu o jeho funkčnosti. Absence ověření syntaxe znamená, že i přes správnou logickou strukturu testu může být výsledek nefunkční. Oprava těchto chyb vyžaduje pečlivou kontrolu a úpravu kódu, což zvyšuje časovou náročnost. Pokud AI generuje velké množství testů, z nichž značná část obsahuje syntaktické chyby, může proces jejich opravy paradoxně převýšit čas potřebný na ruční psaní testů. Tento typ nespolehlivosti zdůrazňuje potřebu robustních mechanismů pro automatickou validaci syntaxe AI výstupů, aby bylo možné minimalizovat riziko nepřeložitelných testů a zvýšit efektivitu jejich využití v praxi.

5.2.3 Bezpečnostní a licenční otázky (autorské právo, důvěrnost kódu)

Nasazení AI při vývoji přináší i právní a bezpečnostní rizika, kvůli nimž některé organizace využití těchto nástrojů přímo zakazují. Jedním z hlavních problémů je důvěrnost kódu: mnoho AI služeb (např. veřejné API *ChatGPT*, cloudová verze *Copilotu*) zpracovává vstupní kód na svých serverech. To znamená, že se potenciálně citlivý či proprietární kód odesílá třetí straně. Firmy se obávají úniku obchodního tajemství – a ne bezdůvodně.

Obava, že jednou zadaný kód může „prosáknout“ ven (například že model při odpovědi jinému uživateli vygeneruje část vašeho kódu), vedla řadu velkých společností (*Apple*, *Goldman Sachs*, *Verizon* aj.) [33] k plošnému zákazu používání *ChatGPT* a podobných nástrojů na pracovních počítačích. Místo toho firmy volí *on-premise* řešení nebo nástroje, které slibují neuchovávat data (například lokálně běžící modely). S důvěrností souvisí i právní odpovědnost za trénovací data: modely jako *Copilot* byly natrénovány na open-source kódu, včetně kódu pod licencemi jako GPL[17]. Model může občas vygenerovat úryvek, který se téměř shoduje s kusem kódu z tréninku, aniž by uvedl licenci či původ. To vyvolává diskuzi, zda používání takového výstupu nepředstavuje porušení autorských práv. Z právního hlediska je situace nejasná – je-li výstup dostatečně originální, pak ne, ale pokud by model reprodukoval např. 20 řádků doslova z nějakého GPL projektu, mohlo by jít o porušení licence. *Microsoft* (*GitHub*) reagoval zavedením *Copilot Copyright Commitment* v roce 2023, což je příslib, že uhradí náklady spojené s případnými žalobami z porušení copyrightu kvůli použití *Copilotu*.[47] Další aspekt je bezpečnost kódu: studie odhalily, že modely občas generují kód zranitelný vůči útokům, protože v tréninku mohly získat i špatné návyky. Naslepo přijatý AI návrh testu či kódu by tak mohl zanést do aplikace bezpečnostní díru. Právní nejistota a riziko úniku dat jsou tak hlavními důvody, proč někteří AI při vývoji odmítají. Raději obětují možné zvýšení produktivity, než aby ohrozili duševní vlastnictví nebo bezpečnost.

[17] GNU General Public License je copyleftová open-source licence, která vyžaduje, aby každý projekt využívající GPL kód byl také zveřejněn pod GPL.

6 AI služby a LLM řešení

Automatizace testování rozšíření pro *Visual Studio Code* pomocí umělé inteligence nabývá na významu s rozvojem velkých jazykových modelů (LLM). Tradiční ruční psaní testů je časově náročné a může opomenout některé scénáře. Naopak automatické generování testů s využitím AI slibuje rychlejší pokrytí kódu a odhalení i netriviálních chyb. [20] V rámci diplomové práce je rozšiřován framework *ExTester* a uvažuje se o integraci AI pro generování testů. Následující část proto hodnotí dostupné AI služby a platformy LLM z hlediska jejich vhodnosti pro generování testovacího kódu (včetně UI testů), možnosti integrace, kvality výstupu a nutnosti revize, a přínosů oproti ručnímu testování. Porovnání využití vlastního LLM modelu vs. externích služeb se zaměří výhody, nevýhody, náklady, licence a dlouhodobou udržitelnost.

6.1 Možné AI služby a jejich přehled

6.1.1 ChatGPT

ChatGPT[34] patří k nejpokročilejším AI nástrojům pro generování zdrojového kódu. Zvláště model *GPT-4.5* dosahuje vynikající kvality výstupu a poradí si i s komplexnějšími scénáři oproti starším modelům *GPT-4* nebo *GPT-3.5*. Studie ukazují, že *ChatGPT* (konkrétně model *GPT-4*) dokáže generovat unit testy s pokrytím kódu srovnatelným se specializovanými nástroji pro automatické testování. [4] V současné době je k dispozici i odlehčený model *3o-mini-high* vhodný pro programování. Disponuje však omezeným výkonem – hodí se spíše pro jednodušší úlohy či rychlé interaktivní použití, zatímco pro generování složitějších testů (např. sekvence UI kroků) je vhodnější plnohodnotný *GPT-4.5*.

OpenAI nabízí API rozhraní, takže *ChatGPT* lze snadno zapojit do stávajících vývojových nástrojů a workflow. V kontextu generování testů v rámci frameworku *ExTester* by bylo možné například automaticky vygenerovat testovací scénáč na základě modelu popisu scénáře (např. „`Otevři paletu příkazů, spusť příkaz X, očekává se zobrazení oznámení Y`“) nebo na základě analýzy zdrojového kódu. Výstupem by byl kód testu využívající API *ExTesteru*.

Model sice nemusí mít předchozí detailní znalost *ExTester* API, ale díky obecným schopnostem doplňování kódu a kontextu (zejména modely s podporou velkého kontextu) lze do promptu zahrnout ukázky *ExTester* syntaxe nebo dokumentaci. *ChatGPT* vygeneruje test s voláním příslušných metod *ExTesteru* (např. otevření okna, vyhledání prvku, simulace kliknutí apod.), který pak lze spustit běžným postupem. Integrace tedy probíhá formou „AI-asistenta“, jenž na požádání navrhne testy – buď přímo v editoru (např. formou *VS Code* rozšíření volající API *OpenAI*), nebo v rámci CI skriptu, který vygenerované testy zařadí do sady testů.

Přestože *GPT-4* dosahuje vysoké kvality, výstupy nutně vyžadují kontrolu a případné dopracování vývojářem. *ChatGPT* často správně pokryje různé scénáře, ale přibližně třetina asercí může být nesprávná či nepřesná. [4]

Model může navrhnout očekávané hodnoty, které neodpovídají skutečné funkčnosti kódu. U testů interagujících s UI se také může stát, že vygenerovaný skript opomene synchronizační kroky (např. počkat na vykreslení prvku) nebo použije nesprávné selektory. Vývojář musí takový test ověřit, spustit a upravit případné chyby. Kombinace AI a lidské revize vede k dobrým výsledkům – například v [4] měl *ChatGPT* a tradiční generátor testů *Pynguin* jen minimální překryv v neotestovaných částech aplikace, takže každý odhalil jiné mezery.

6.1.2 Grok

Grok[50] od *xAI*, projektu pod vedením Elona Muska, je pokročilý velký jazykový model zaměřený na *truth-seeking* odpovědi s důrazem na logické uvažování, programování a zpracování vizuálních vstupů. Ve verzi *Grok 3* byl trénován na superpočítači *Colossus*[18], což mu umožňuje excelovat v matematických, kódovacích a analytických úlohách díky intenzivnímu trénování s posilovaným učením a technikou chain-of-thought [19]. *Grok 3* přemýšlí nad řešeními iterativně – při dotazu analyzuje více přístupů, kontroluje své odpovědi a zpřesňuje je, což trvá od několika sekund po minuty. Celkově *Grok* generuje vysoce kvalitní kód, který je často funkční, ale může vyžadovat drobné úpravy pro okrajové případy.[16]

Díky důrazu na hluboké uvažování *Grok* exceluje v analýze požadavků a odvozování posloupnosti akcí, což je klíčové pro tvorbu testovacích scénářů. Integrovaný mechanismus *Code Interpreter* umožňuje spouštět kód a ověřovat jeho funkčnost v reálném čase. *Grok* tak může při generování testů „přemýšlet nahlas“, například ověřit, zda daný selektor UI elementu existuje, pokud má přístup k instanci prostředí. Pro UI testy v *ExTesteru* by to teoreticky mohlo zjednodušit validaci, ale v praxi by vyžadovalo integraci s nástroji jako *VS Code* API, což *Grok* zatím nativně nepodporuje. API *xAI* je plně dostupné, což usnadňuje integraci do vlastních nástrojů, i když API stále zaostává za rozšířením *OpenAI* nebo *Anthropic Claude*.

Grok 3 produkuje kód srovnatelný s nejlepšími modely jako je *GPT-4o*, díky iterativnímu přístupu, který minimalizuje logické chyby. Generované testy obvykle pokrývají většinu požadavků (nastavení, akce, očekávání) a jsou konzistentní, ale vývojáři by měli kontrolovat okrajové podmínky, syntaxi a asynchronní operace. Kvalita je dostatečná pro rychlé prototypování i produkční použití po standardní revizi.

6.1.3 Anthropic Claude

Claude[2] od firmy *Anthropic* je dalším špičkovým LLM zaměřeným na bezpečnou a kvalitní konverzaci. Ve verzi *Claude 2* uvedené v roce 2023 a novějších iteracích se výrazně zlepšily schopnosti modelu v programování a analýze kódu. *Claude* je trénován s důrazem

[18] Výpočetní *cluster* s *200 000* GPU *NVIDIA H100*, který umožnil obrovský výpočetní výkon v podobě více než 200 milionů GPU hodin.

[19] Technika, při níž jazykový model řeší problém krok za krokem s explicitním vysvětlením úvah, což zvyšuje přesnost a srozumitelnost výsledků.

na srozumitelnost a dodržování instrukcí, což v praxi znamená, že často produkuje velmi čitelný a dobře komentovaný kód. Má také mimořádně velké okno kontextu – až 100 tisíc tokenů ve verzi *Claude 2* a 200 tisíc ve verzi 3. (*Grok 3* nabízí 131072 tokenů a *GPT4.5* 128 000 tokenů.) To je výhoda pro zpracování dlouhých zadání nebo při práci s rozsáhlými kódy (například generování testů z dlouhého specifikačního dokumentu nebo analýza většího projektu).

Claude si vede velmi dobře v generování kódu a zvládá i komplexní úlohy. Pro generování testovacích scénářů je *Claude* vhodný zejména tam, kde je potřeba pracovat s velkým množstvím vstupního textu – například při generování testů z obsáhlé specifikace chování rozšíření, nebo pokud chceme modelu poskytnout dokumentaci *ExTester* a API rozhraní celého rozšíření najednou. *Claude* rovněž umožňuje skriptovat testy v jednom dlouhém dialogu – lze mu předložit několik scénářů naráz a požádat o vygenerování testovací sady.

Anthropic zpřístupňuje *Claude* přes vlastní API, podobně jako *OpenAI*. Integrace do vývojových nástrojů je tedy možná. Pro *ExTester* by postup byl obdobný jako u *ChatGPT* – poslat dotaz s popisem testu a relevantním kontextem, získat kód testu.

Claude je známý tím, že produkuje obsáhlé vysvětlení a komentáře spolu s kódem. To může testerovi pomoci pochopit návrh testu. Kvalitou je kód srovnatelný s jinými špičkovými modely – drobné chyby se mohou vyskytnout, ale v praxi uživatelé hlásí vysokou spolehlivost. Některé benchmarky uvádějí, že *Claude 3* (označovaný též *Claude 3.5 Sonnet*) v určitých testech kódování drží úspěšnost blízko *GPT-4*.[19]

Výhodou je, že *Claude* má tendenci dodržet zadání velmi přesně a méně často odbíhá od tématu. Pokud například specifikujeme očekávaný výstup testu, *Claude* se toho drží. Přesto je nezbytné vygenerované testy otestovat – jako u jiných modelů platí, že AI může nesprávně pochopit okrajové požadavky. Revize by ale nemusela být tak rozsáhlá neboť *Claude* se profiluje jako model, který „vkládá do odpovědí opatrnost“ a vyhne se zjevným nesmyslům.

6.1.4 LLaMA 2

LLaMA 2[23] je rodina velkých jazykových modelů vyvinutých společností *Meta AI* jako nástupce původního modelu *LLaMA*. Jde o modely dostupné ve variantách o 7, 13 a 70 miliardách parametrů. *Meta LLaMA 2* byla uvolněna v červenci 2023 pod relativně otevřenou licencí, která umožňuje i její komerční využití (s omezením až pro extrémně velké podniky). K dispozici jsou jak základní předtrénované modely, tak verze jemně vyladěné pomocí instrukcí a zpětné vazby pro konverzační použití (*LLaMA 2-Chat*). Model *LLaMA 2* byl trénován na rozsáhlém množství textových dat a má kontextovou kapacitu 4096 tokenů, tedy dvojnásobnou oproti *LLaMA 1*. Díky efektivní architektuře poskytuje vysoký výkon v poměru ke své velikosti a patří k nejvýkonnějším otevřeným LLM modelům současnosti.

Z hlediska generování kódu však *LLaMA2* (v základní podobě) nedosahuje takové úrovně jako specializované modely. V benchmarku *HumanEval* pro generování funkcí z popisu dosáhl *LLaMA 2* (*13B*) úspěšnosti pouze kolem 30 %, zatímco nejmodernější

modely jako *GPT-4* dosahují až 67 %. [18] Pro potřeby generování testů to znamená, že *LLaMA 2* sice dokáže na základě textového zadání navrhnout jednoduché testy, ale může častěji produkovat méně přesný či nekompletní kód. Vyšší varianty (*70B*) mohou nabídnout lepší výsledky díky větší znalostní kapacitě, avšak za cenu výrazně vyšších nároků na hardware. Omezením je také menší vyladění na instrukce týkající se programování oproti specializovaným modelům – *LLaMA 2* byla primárně trénována na obecný text. Na druhou stranu, otevřenost modelu umožňuje jeho další doladění (*fine-tuning*) na konkrétní doménu či úlohy.

Pro potřeby generování testů to znamená, že *LLaMA 2* sice dokáže na základě textového zadání navrhnout jednoduché testy, ale může častěji produkovat méně přesný či nekompletní kód. Vyšší varianty (*70B*) mohou nabídnout lepší výsledky díky větší znalostní kapacitě, avšak za cenu výrazně vyšších nároků na hardware. Omezením je také menší vyladění na instrukce týkající se programování oproti specializovaným modelům – *LLaMA 2* byla primárně trénována na obecný text. Na druhou stranu, otevřenost modelu umožňuje jeho další doladění na konkrétní úlohy.

6.1.5 Code LLaMA

Code LLaMA[24] je specializovaná verze modelu *LLaMA 2* zaměřená na generování programového kódu. Vydána byla společností *Meta* v srpnu 2023 jako open-source s totožnou licencí jako *LLaMA 2*. *Code LLaMA* vychází z architektury *LLaMA 2*, ale je dále dotrénována na rozsáhlém počtu zdrojových kódů a technických textů. Existuje ve třech velikostech: *7B, 13B a 34B* parametrů, přičemž větší modely poskytují lepší kvalitu výstupu za cenu vyšších nároků na výkon. Oproti základnímu *LLaMA* 2 si *Code LLaMA* výrazně lépe vede v úlohách programování – například v zmíněném testu *HumanEval* dosáhla *34B* verze úspěšnosti kolem 53 %, což výrazně překonává základní *LLaMA 2* (30 %), i když na nejmodernější *GPT-4* (67 %) stále ztrácí.

Hlavní výhodou *Code LLaMA* pro generování testů je schopnost produkovat syntakticky i logicky korektní kód ve více programovacích jazycích. Model *Instruct* navíc rozumí instrukcím v přirozeném jazyce, takže lze zadat například popis funkčnosti pluginu a nechat model vygenerovat odpovídající testovací skripty. Otevřenost modelu znamená, že jej lze provozovat lokálně a integrovat do nástrojů. Limitem menších variant může být nižší hloubka porozumění složitějším požadavkům – *7B* model může generovat jednodušší či částečně neoptimální testy, zatímco *34B* varianta již vyžaduje výkonný hardware (desítky *GB VRAM*), který nemusí být běžně k dispozici.

V kontextu *VS Code* je *Code LLaMA* velmi relevantní, neboť byl navržen přímo pro podporu vývojářských činností. Lze si představit jeho integraci do rozšíření tak, že při zadání popisu testovacího scénáře či funkce pluginu *Code LLaMA* vygeneruje strukturu testů. Díky podpoře tzv. *infilling režimu* umí model doplňovat kód i doprostřed souboru, což se hodí při generování testů do existující testovací sady.

6.1.6 Mistral

Mistral 7B[31] je otevřený jazykový model představený startupem *Mistral AI* v září 2023, který vzbudil pozornost svou efektivitou. S pouhými 7 miliardami parametrů dokázal *Mistral 7B* překonat tehdy nejlepší otevřený model *LLaMA 2* s 13 miliardami parametrů na všech hodnocených benchmarcích. Zejména vyniká v logickém uvažování a porozumění instrukcím, což je pro generování testů klíčové. V oblasti generování kódu dosahuje výkonu srovnatelného s *Code LLaMA 7B*. [36]

Pro generování testů poskytuje *Mistral 7B* zajímavou kombinaci výkonu a efektivity. Umožňuje v rámci *VS Code* rozšíření realizovat základní AI asistenci při tvorbě testů i na slabším stroji, přesto s kvalitou lepší, než jakou by dříve nabídly obdobně velké modely. Samozřejmě, ve srovnání s modely jako *Code LLaMA 13B* může *Mistral 7B* generovat o něco jednodušší nebo méně spolehlivé testy – vzhledem k omezené parametrové kapacitě nemusí vždy porozumět velmi komplikovaným scénářům nebo obsáhlému kontextu. Na druhou stranu, díky otevřenosti a očekávatelné komunitní podpoře není problém model dále doladit na konkrétní typ vstupů, čímž by se jeho přesnost mohla ještě zvýšit.

6.2 Vlastní LLM vs. využití existujících služeb

Při zvažování nasazení AI pro generování testů stojí organizace před rozhodnutím: využít existující službu/model (komerční API nebo open-source model bez úprav), anebo vytrénovat a doladit vlastní model speciálně pro své potřeby. Obě cesty mají své výhody a nevýhody v několika rovinách – kvalita a výkon, náklady, právní aspekty a dlouhodobá udržitelnost.

6.2.1 Kvalita a efektivita řešení

Přesnost a schopnosti: Komerční LLM jako *GPT-4* (*OpenAI*) či *Claude* bývají trénovány na obrovských datech s nejmodernějšími architekturami, což jim dává náskok v obecné inteligenci a porozumění složitým úlohám. Vlastní model by takovou kvalitu dosáhl jen těžko, pokud by nebyl srovnatelně velký a trénovaný na podobném množství dat. V praxi proto nasazení existujících špičkových modelů zaručuje vyšší kvalitu generovaných testů hned od začátku – model pravděpodobně pokryje více okrajových případů a udělá méně chyb v logice testu. Naproti tomu menší, vlastní LLM může v určitých doménách selhávat (např. špatně vyhodnotí, co je očekávaný výstup funkce v komplikované situaci). [22]

Rychlost dosažení výsledků: Použití hotové služby znamená, že téměř ihned můžeme generovat testy – stačí integrace API. Trénování vlastního modelu je zdlouhavý proces (dny až měsíce, v závislosti na rozsahu) a vyžaduje iterativní ladění. Pokud je cílem rychle zvýšit pokrytí testy, externí AI služba přinese hodnotu okamžitě, zatímco vlastní model by byl spíše dlouhodobá investice.

Kontrola nad výstupem: Vlastní model lze potenciálně lépe přizpůsobit specifickým požadavkům. Například může generovat testy přesně dle interních stylů a konvencí hned jak ho to naučíme. U veřejné služby jsme odkázáni na to, co se model naučil obecně –

někdy může generovat příliš obecné nebo stylově nejednotné testy. Je však možné tyto aspekty řídit v promptu, takže rozdíl se dá zmenšit.

6.2.2 Náklady časové a finanční

Využití existujících služeb: Typicky funguje na bázi *pay-as-you-go*[20] nebo paušální předplatné. Krátkodobě jsou náklady nízké – například vygenerování jednoho testovacího souboru může stát jen haléře v přepočtu na tokeny. Není nutná žádná počáteční investice, hardware zajišťuje poskytovatel. Pro projekty, které nepotřebují generovat obrovské množství kódu nepřetržitě, je ekonomicky výhodnější využít cloudovou AI. Pokud by se ale generovalo velmi mnoho testů, může se celková cena za využití API nasčítat. Nicméně i tak se pohybujeme v částkách, které jsou ve srovnání s náklady na vývoj vlastního modelu zanedbatelné. Z časového hlediska je nasazení služby rychlé, jak je zmíněno výše.

Trénování vlastního LLM: Tato varianta může být velmi nákladná. Vytvoření modelu srovnatelného s *GPT-3* se odhaduje na jednotky až desítky milionů dolarů v nákladech na výpočet [6], nemluvě o potřebě expertního týmu a iterativního ladění. To je pro běžnou organizaci nerealistické. Reálnější je vzít open-source model a doladit ho (*fine-tune*) na vlastní data.

Například *fine-tuning 7B–13B* modelu může stát v řádu nižších tisíců dolarů (za GPU čas), u *70B* modelu to mohou být desítky tisíc. Dále je ale nutné investovat čas lidí: připravit trénovací data (např. sebrat dostatek příkladů testů, scénářů, dokumentace), nastavit experimenty, ověřovat kvalitu modelu. Celkově se implementace vlastního modelu může protáhnout na týdny či měsíce vývoje. Finančně kromě samotného jednorázového trénování musíme počítat i s provozními náklady – hardware (GPU servery) buď vlastnit, nebo platit za pronájem cloudových instancí. Tyto náklady běží stále, na rozdíl od *pay-per-use* modelu u externí služby. Proto pro organizaci, která nemá AI vývoj jako svoji hlavní činnost, bývá ekonomičtější využít hotové služby. Výjimkou může být situace, kdy firma již disponuje potřebnou infrastrukturou (např. volné GPU servery) a experty, takže trénování využije existující zdroje.

Jednorázově je tedy investiční výdaj u vlastního modelu vysoký, zatímco provozní výdaje mohu být pak nižší pro velké objemy použití. Pro diplomovou práci či menší tým je prakticky jisté, že levnější a rychlejší cestou je využít existující AI službu (např. přes API *GPT-4*), neboť náklady na vybudování vlastního LLM by nebyly úměrné přínosu.

6.2.3 Regulační a licenční aspekty

Důvěrnost a soulad s předpisy: Jak již bylo zmíněno, při využití veřejných cloud AI služeb je nutné zvážit důvěrnost dat. Odesláním zdrojového kódu (např. neveřejného rozšíření) do služby jako *OpenAI* API teoreticky hrozí, že data uvidí provozovatel nebo by mohla být uložena. *OpenAI* sice pro API klientů garantuje, že data nepoužije k trénování bez svolení a smaže je po určité době, přesto některé organizace z právního hlediska toto využití nepovolují. Řešením může být *Azure OpenAI*, kde je smluvně ošetřeno uklá-

[20] Platba za využité tokeny či volání API.

dání dat podobně, nebo nasazení open-source modelu interně, kde data vůbec neopouští firmu. U vlastního LLM tedy otázka důvěrnosti odpadá (pokud se trénuje čistě na interních/povolených datech).

Autorské právo a licence generovaného kódu: Výstupy z LLM nejsou dosud jednoznačně právně klasifikované. Většina poskytovatelů (*OpenAI*, *Anthropic*) ve svých podmínkách uvádí, že uživatel je vlastníkem výstupu, s výjimkou případů, kdy by výstup byl přímo z chráněného díla v trénovacích datech. To znamená, že např. testovací kód vygenerovaný AI může vývojář normálně použít a licencovat dle libosti. Riziko nastává, pokud model vygeneruje větší část kódu shodnou s existujícím licencovaným kódem. U testů to není příliš pravděpodobné. Přesto došlo ke zjištění, že *Copilot* občas emitoval i 1:1 kopie licencovaných úryvků, pokud měl specifický prompt. Z právního hlediska by pak použití takového výstupu mohlo zavazovat k dodržení licence původního díla. Pro minimalizaci rizika by AI generované testy měly být originalistické – spíše nové kombinace volání API než opsané konkrétní implementace. U vlastního modelu trénovaného na firemním kódu zase může nastat opačný problém: model je derivátem tohoto kódu, ale výstupy mohou obsahovat části, které firma považuje za tajné.

Licence modelu a servisní smlouvy: Při použití komerčního API musí uživatel dodržovat podmínky služby. Ty typicky zahrnují zákaz použití k ilegálním účelům, někdy omezení na určitý obsah apod. V kontextu generování testů nejsou tato omezení problém. Důležitější je, že některé služby mají omezení množství či rychlosti – například *GPT-4* API má limit na počet požadavků za minutu. U vlastního řešení žádné takové smluvní limity nejsou – výkon škáluje jen s hardware. Ale zase pozornost vyžaduje licence open-source modelu: například *LLaMA 2* je zdarma pro komerční použití, ale pokud by náš produkt využívající LLM měl přes 700 milionů uživatelů, Meta vyžaduje uzavření jiné dohody. Většina open-source modelů (*Falcon, Mistral*) má přívětivou licenci (*Apache 2.0*), takže je lze použít bez obav.

6.2.4 Udržitelnost a správa modelu

Aktualizace a vylepšování: AI modely se rychle vyvíjejí – *OpenAI* či *Anthropic* průběžně uvádějí vylepšené verze, které poskytují lepší výkon nebo větší kontext. Pokud využíváme službu, máme výhodu, že poskytovatel za nás inovuje model – my jen přepneme na novější verzi a hned těžíme z lepší kvality. Například přechod z *GPT-3.5* na *GPT-4* přinesl dramatické zlepšení v přesnosti generovaných testů (méně chybných asercí atd.), a to bez naší vlastní práce. U vlastního modelu bychom museli investovat do průběžného trénování, abychom drželi krok s dobou – jinak model morálně zastará a jeho znalosti mohou být rychle zastaralé. Zvlášť v doméně vývoje software: model natrénovaný v roce 2023 nebude znát nové API *VS Code* z roku 2025, nové verze *ExTesteru* atd., pokud ho nebudeme adaptovat. U externí služby je vysoká šance, že novější model už tyto znalosti bude mít.

Údržba infrastruktury: Provoz vlastního LLM znamená starat se o servery: instalovat aktualizace, monitorovat výkon, řešit pády procesů, optimalizovat kód atd. V dlouhodobém horizontu to může zatěžovat vývoj a odvádět pozornost od jádra byznysu (pokud jím není přímo AI). U služeb tuto starost přebírá poskytovatel – zajišťuje škálování, zálo-

hy, v případě *Azure* i integritu dat. Pro tým soustředící se na kvalitu softwaru je obvykle výhodné nechat tyto starosti na odbornících mimo firmu a využívat AI jako službu.

Dlouhodobé náklady: Je třeba zvážit, jestli bude využití AI pro generování testů spíše konstantní potřeba, nebo jednorázová akce. Pokud jde o jednorázové vygenerování sady testů (např. při vývoji nového rozšíření vytvořit počáteční testy), pak nemá smysl budovat vlastní model – využije se krátce. Pokud by naopak aplikace vyžadovala neustálé interakce (např. inteligentní asistent v editoru, který denně generuje stovky testů pro každého uživatele), pak by se mohlo vyplatit investovat do optimalizace nákladů přes vlastnictví modelu. Většinou ale i kontinuální použití lze uřídit finančně přes vhodný tarif služby. U vlastního modelu navíc může nastat, že morálně zastará: nové algoritmy či architektury mohou náš model překonat a museli bychom od nuly začít s novým. Naproti tomu při odběru služby se flexibilně můžeme rozhodnout přejít k jinému poskytovateli nebo jiný model, pokud se objeví lepší. Tato flexibilita je cenná – investice do vlastního modelu je fixní a ztrácí hodnotu, pokud model přestane být špičkový.

7 Metodika

Metodika popisuje postup, jakým byly naplněny cíle práce, a představuje navržené řešení pro automatizované generování UI testů. V úvodu této části jsou shrnuty klíčové kroky od návrhu až po implementaci prototypu: nejprve je nastíněna architektura rozšíření *VS Code*, které kombinuje framework *ExTester* s generativní AI pro tvorbu testů, následně jsou rozebrány použité technologie a integrační mechanismy. Metodika dále popisuje, jak je zajištěna validace a udržovatelnost vygenerovaných testů a jakým způsobem bude ověřena účinnost navrženého postupu v praxi. Tato kapitola tak přemosťuje teoretické poznatky z rešerše a analýzy s praktickou realizací prototypu a připravuje půdu pro následné experimentální vyhodnocení přínosů AI při testování *Visual Studio Code* rozšíření.

7.1 Volba AI řešení

V kontextu generování testů pro *VS Code* rozšíření dává současný stav technologií většinou výhodu existujícím službám. Ty nabízejí vyšší okamžitou kvalitu a nižší počáteční náklady. Vlastní LLM řešení může být opodstatněné jen v případě striktních požadavků na soukromí nebo velmi specifických potřeb, které komerční modely nesplňují. Často se volí kompromis: využít open-source model, ale nebudovat ho od nuly – místo toho vzít již natrénovaný (např. *Code Llama*) a jen ho hostovat interně, případně lehce doladit. Tím se získá slušná kvalita a zachová kontrola nad daty, ovšem s podstatně menším úsilím než trénovat nový model. Nevýhodou oproti komerčním modelům tu ale zůstává, že kvalita nebude srovnatelná. Proto, pokud nejsou překážky využití cloudu, je ekonomicky i prakticky rozumnější využít model typu *GPT-4/Claude* přes API a soustředit se spíše na nastavení vhodného workflow kolem (validace testů, ukládání, začlenění do projektu) než na samotné budování AI.

Výběr řešení založeného na *OpenAI ChatGPT* vychází z podrobné analýzy v rešerši. Ta ukázala, že největší kvalitu generovaného kódu poskytují špičkové modely jako *GPT-4*, zatímco open-source LLM buď vykazují nižší přesnost, nebo vyžadují značné hardwarové nároky. Navíc *OpenAI* nabízí dobře zdokumentované API, což umožňuje snadnou integraci do vývojového workflow. Konkurenční modely (např. *Claude*, *Grok*) sice dosahují podobné inteligence, avšak *ChatGPT* má díky enormní uživatelské základně a rozsáhlé komunitní podpoře nejvyspělejší ekosystém. Z toho plyne, že volba *ChatGPT* znamenala nejlepší kompromis mezi kvalitou výstupu (generování rozumných a komplexních testů), dostupností rozhraní a jazykovými schopnostmi i podporou komunity. Pokud neexistují zásadní požadavky na soukromí, je ekonomicky rozumnější využít veřejné modely typu *ChatGPT* prostřednictvím API a soustředit se na okolní workflow místo vlastní výstavby AI.

7.2 Vhodnost pro generování UI testů

Z pohledu diplomové práce lze závěry výše aplikovat následovně: Pro generování UI testů v *TypeScriptu*, které využívají *ExTester* API, se jako nejefektivnější jeví nasadit někte-

rou z výkonných AI platforem (např. *ChatGPT GPT-4*, *Grok* či *Claude*) prostřednictvím jejich API. Tyto modely mají dostatečné schopnosti pochopit popisy uživatelských scénářů a přeložit je do sekvence volání *ExTester* metod. Vývojář může interaktivně doladit prompt, případně doplnit kontext (ukázkové volání *ExTester*) a model rychle navrhne test. Díky své inteligenci navrhne i různé varianty vstupů a akcí, čímž zvýší pokrytí. *ExTester* sám o sobě slouží k běhu testů, ale nenabízí jejich generování – zde AI doplňuje chybějící článek automatizace. LLM lze integrovat přímo do *VS Code* v podobě rozšíření. Pro zajištění kvality by měl být proces doplněn o review: vygenerované testy projde vývojář, spustí je *ExTesterem* proti instanci *VS Code* a zkontroluje se, že testy prochází a skutečně testují zamýšlené chování. Synergie AI a člověka se ukazuje jako optimální – studie doporučují kombinovat AI generované testy s lidským zásahem, protože dohromady odhalí více problémů než každý přístup sám. [20]

7.3 Definice funkčních a nefunkčních požadavků.

Definice funkčních a nefunkčních požadavků vychází z analýzy stavu v předchozích kapitolách. Funkční požadavky vycházejí z identifikovaných limitujících faktorů stávajícího přístupu k testování, které negativně ovlivňují efektivitu, škálovatelnost a udržitelnost testovacího procesu. V našem kontextu to byly zejména problémy popsané v kapitole 3 – např. nutnost ručního psaní testů, absence nástrojů pro automatické generování kostry testů. Požadavek na automatické generování testů explicitně reaguje na zjištěnou mezeru v absenci nástroje pro generování *TypeScript* testů pro framework *ExTester*. Nefunkční požadavky (např. přenositelnost, použitelnost, výkon nebo kompatibilita s CI/CD) byly formulovány na základě stanovených kritérií v kapitolách 2–5. Rešerše například upozornila na slabou standardizaci UI testů v CI/CD pipelines, která dělá jejich integraci problematickou. Požadavky na kompatibilitu s CI/CD tedy reflektují tuto potřebu, zatímco požadavky na použitelnost a intuitivní ovládání reagují na nutnost zjednodušit workflow vývojáře.

7.3.1 Funkční požadavky

Automatické generování testů: Nástroj musí umožnit automaticky generovat UI testy na základě již existujícího zásuvného modulu. Hlavním cílem je, aby uživatel (vývojář rozšíření) poskytl již naimplementovaný zásuvný modul, ze kterého nástroj s využitím AI vytvoří odpovídající testovací skript. Generované testy by měly využívat strukturu a API frameworku *ExTester* (*Mocha* `describe/it` bloky a předpřipravené *š* metody) tak, aby byly srozumitelné pro vývojáře a v optimálním případě i přeložitelné a spustitelné.

Integrace s ExTesterem: Nástroj musí podporovat syntaktické konstrukce *ExTesteru* a hladce se integrovat s tímto frameworkem. To zahrnuje generování testů využívajících *Page Object* API *ExTesteru* (např. volání metod jako `workbench.openCommandPalette()` namísto nízkoúrovňové simulace kláves) a respektování požadavků *ExTesteru* na strukturu testů. Výstupem generování budou testy kompatibilní s *ExTester CLI* (příkaz `extest`), které lze bez úprav spustit v rámci testovacího běhu.

Integrace do vývojového prostředí: Zásuvný modul musí být plně integrován do prostředí *VS Code*, aby zapadl do běžného workflow vývojářů. To znamená, že nabídne uživatelsky přívětivé rozhraní (např. příkazy v *Command Palette* nebo vlastní panel) pro generování testů. Vývojář by měl být schopen vytvářet testy přímo z *VS Code* bez nutnosti přepínat do externích nástrojů.

7.3.2 Nefunkční požadavky

Přenositelnost: Řešení by mělo být použitelné na všech hlavních platformách (*Windows*, *Linux*, *macOS*) a kompatibilní s různými verzemi *VS Code* i *ExTesteru*. Nástroj musí fungovat konzistentně (v rámci možností AI) bez ohledu na operační systém, aniž by vyžadoval specifickou konfiguraci pro jednotlivé platformy.

Použitelnost: Důraz je kladen na snadné použití nástroje. Uživatelské rozhraní rozšíření by mělo být intuitivní, aby i vývojáři bez velkých zkušeností dokázali nástroj úspěšně využít. Dokumentace a nápověda by měly být součástí řešení. Z hlediska UX je cílem minimalizovat počet kroků potřebných k vygenerování testů a poskytnout srozumitelné výstupy (např. jasná hlášení o průběhu či chybách testů).

Rozšířitelnost: Architektura nástroje by měla umožňovat snadné rozšiřování o další funkce či úpravy. Mělo by být možné relativně jednoduše aktualizovat použitý model generativní AI nebo pravidla pro generování testů, aniž by bylo nutné přepisovat celé řešení. Rozšiřitelnost také znamená, že kód bude čistý a modulární, aby jej bylo možné snadno udržovat a vylepšovat v budoucnu.

7.4 Možnosti empirického vyhodnocení

Po implementaci prototypu následuje empirické vyhodnocení, které ověří splnění výše definovaných požadavků a celkový přínos nástroje. Pro objektivní zhodnocení byly navrženy měřitelné charakteristiky zaměřené na hlavní aspekty procesu generování testů a kvality výsledné testovací sady. Plánované empirické hodnocení navazuje na podobné studie a metodiky uvedené v literatuře. Studie [21] například doporučují explicitní porovnání efektivity a přesnosti AI-generovaných testů oproti manuálnímu postupu – ukázaly, že generování end-to-end testů pomocí *ChatGPT* významně ušetřilo práci testerům při definovaných scénářích.

Naše metodika proto bude testovat rozšíření v reálném provozu a porovnávat výsledky generované AI s ručně psanými testy na stejných scénářích. Hodnocení se bude zaměřovat jak na technickou správnost a stabilitu vygenerovaných testů, tak na jejich věcnou shodu se zadáním scénářů a míru opory v reálných funkcionalitách testovaného rozšíření. Součástí vyhodnocení bude rovněž posouzení výskytu nepodložených či nereálných kroků v testech a čitelnosti výsledného kódu z pohledu vývojáře. Do empirického hodnocení budou zahrnuty jak objektivní metriky (např. spustitelnost testů, chybovost či pokrytí kódu), tak subjektivní hodnoticí škály reflektující kvalitu a použitelnost výstupů. Cílem je kvantifikovat, nakolik automatizovaná generace testů skutečně zrychluje proces testování, aniž by negativně ovlivnila přesnost, srozumitelnost nebo věrohodnost výsledných testovacích

scénářů. Tento přístup přímo vychází z doporučení zmíněných studií (např. [21]) a je dále rozpracován v příslušné kapitole.

Kvalita generovaných testů: Kvalita automaticky vytvořených testů je posuzována z několika vzájemně provázaných hledisek. Zahrnuje především správnost testovacích scénářů vzhledem k zadání, jejich technickou bezchybnost a stabilitu při opakovaném spouštění, ale také míru, s jakou se generované testy opírají o reálně existující funkcionalitu testovaného rozšíření. Důraz je kladen rovněž na identifikaci případů, kdy generovaný test obsahuje nepodložené nebo nereálné kroky, a na celkovou srozumitelnost a udržovatelnost výsledného kódu. Kvalitní test by měl nejen korektně ověřovat požadované chování aplikace, ale měl by být také čitelný a prakticky použitelný pro vývojáře. Konkrétní metriky a způsob jejich vyhodnocení jsou detailně popsány v následující kapitole.

Pokrytí kódu: Míra, do jaké generované testy pokrývají kód testovaného rozšíření (*code coverage*). Vyšší pokrytí obecně znamená, že testy ověřují větší část funkčnosti aplikace. Pokrytí bude měřeno pomocí nástroje pro měření *code coverage* a udáváno v procentech pokrytých řádků kódu. Tento ukazatel kvantifikuje, nakolik široké spektrum scénářů dokáže generovaný balík testů obsáhnout.

Uživatelská spokojenost: Zahrnuje subjektivní hodnocení vývojářů, kteří nástroj používají, a jejich důvěru v generované testy. Bude hodnoceno, jak snadno se plugin používá v praxi, zda vývojářům skutečně usnadňuje práci a zda by jej byli ochotni nasadit do svého běžného procesu testování rozšíření. Tato charakteristika může být měřena kvalitativně (např. formou zpětné vazby od uživatelů) na vybraném vzorku uživatelů.

Ačkoli metriky jako tyto byly v literatuře zmiňovány, v závěrečném hodnocení nebudou zahrnuty. Důvodem je jejich omezená vypovídací hodnota v kontextu našeho výzkumu a v některých případech také chybějící předpoklady pro smysluplné využití těchto metrik. Pokrytí kódu sice měří podíl zdrojového kódu vykonaného generovanými testy, avšak i vysoké procento pokrytí samo o sobě nezaručuje účinnost testů při odhalování chyb. Vyhodnocení navíc bude pracovat pouze se vzorkem z navržených testů.

Metrika správnosti testů hodnotí, nakolik generované testy (zejména jejich aserce) skutečně ověřují zamýšlenou funkčnost programu a mají správné výsledky. Její aplikace však vyžaduje znalost očekávaných výstupů či referenční implementace pro každou testovanou funkci, což v rámci této práce nebude k dispozici.

Uživatelská spokojenost jako metrika nebude do hodnocení zařazena, protože hodnoceným nástrojům chybí přímý uživatelský kontakt. Vzhledem k tomu, že testy budou generovány automatizovanými nástroji, nebude možné v dané fázi získat relevantní zpětnou vazbu od koncových uživatelů. Zatímco uživatelská spokojenost je běžně využívána při hodnocení interaktivních systémů nebo vývojových nástrojů s uživatelským rozhraním, v tomto kontextu bude chybět potřebná interakce a měřitelná odezva. Proto by případné zařazení této metriky do hodnocení bylo spekulativní a nepřineslo by objektivně interpretovatelné výsledky.

Další často zmiňovanou charakteristikou je robustnost generovaného výstupu. V literatuře je však robustnost typicky definována jako odolnost modelu vůči změnám vstupních parametrů (např. parafrázím či nepřesnostem zadání). Tento typ robustnosti je relevantní především v situacích, kdy je vstup proměnlivý nebo nejednoznačný. V posuzovaném pří-

padě jsou však vstupy do generativního procesu jednoznačně definované a neměnné, což činí hodnocení vstupní robustnosti metodicky nepřínosným. Robustnost ve smyslu konzistence výstupů při opakovaných bězích se shodným zadáním by byla relevantní spíše v případě nedeterministického nastavení modelu. V této práci je však generativní komponenta konfigurována deterministicky, a proto tato vlastnost nebude samostatně vyhodnocována.

8 Implementace

Tato kapitola podrobně popisuje realizaci prototypu rozšíření pro *Visual Studio Code* (*VS Code*). Nejprve je představena architektura řešení a hlavní komponenty. Následně jsou rozebrány rozebrány klíčové implementační kroky: generování testů pomocí jazykového modelu, uložení a struktura testovacích souborů, integrace s prostředím *VS Code* a *ExTesterem* pro spuštění testů a detekci chyb. Důraz je kladen na problémy a na jejich řešení – od volby vhodného AI modelu pro různé fáze, přes technická omezení prostředí, až po zajištění spolehlivého zachycení a analýzy výsledků testů. V závěru jsou diskutovány další implementační rozhodnutí.

8.1 Architektura a komponenty

Implementované řešení je navrženo modulárně a s ohledem na případnou rozšiřitelnost a udržovatelnost. Architektura nástroje integruje generativní AI a zahrnuje několik základních komponent.

Uživatelské rozhraní: Rozšíření poskytuje jednoduché UI formou vlastního zobrazení v postranním panelu *VS Code*. Toto zobrazení obsahuje položky pro jednotlivé kroky procesů – *Generování Test Proposals*, *Fix Compilation Issues* a *Fix Runtime Failures*. Uživatel tak jedním kliknutím spustí generování testů, zkontroluje a případně opraví chyby při kompilaci nebo samotném běhu. Díky využití integrovaného UI tak nástroj zapadne do prostředí *VS Code*..

Modul analýzy rozšíření: Rozšíření extrahuje základní informace o testovaném vstupním pluginu. Pomocí parsování základního souboru definujícího rozšíření `package.json` získává relevantní části pluginu. Díky tomu získává identifikátory rozšíření, seznam registrovaných příkazů, aktivačních událostí, položek menu, uvítacích obrazovek či konfiguračních klíčů. Tyto informace charakterizují účel rozšíření a jsou využity jako kontext pro generování testů. Cílem je poskytnout generativní AI přehled o tom, jaké funkcionality plugin nabízí (například jaké příkazy implementuje), aby navržené testy byly smysluplné a pokrývaly podstatné scénáře.

Komunikační modul: Zajišťuje volání jazykového modelu pro generování testů a návrhy oprav. Implementace využívá *OpenAI* API – uživatel zadává svůj API klíč v nastavení rozšíření. Nástroj využívá dva režimy modelu:

- *ChatGPT* režim pro úlohy a analýzu.
- *Codex* režim specializovaný na generování zdrojového kódu.

Toto oddělení vychází z pozorování, že nejpokročilejší model *GPT-5* exceluje v porozumění zadání a návrhu testovacích scénářů, zatímco specializovaný model *GPT-5-Codex* produkuje syntakticky přesný kód. Komponenta obsahuje funkce, které asynchronně odešlou dotaz s připraveným promptem do příslušného modelu a obdrží odpovědi.

Generátor testovacích souborů: Stará se o vytvoření nové sady testů v projektu testovaného rozšíření. Zajišťuje vytvoření adresářové struktury a vygenerování testovacích souborů ve formátu očekávaném *ExTesterem*. Generátor pracuje s výstupem AI (návrhy

testů) – výsledek prvotního dotazu AI je parsován a převeden na interní reprezentaci testovacích případů. Pro každý navržený test pak modul vytvoří odpovídající soubor s testem: nejprve připraví adresář (pokud jsou testy členěny dle kategorií) a poté zapojí AI k vygenerování samotného obsahu testu. Zde se využívá schopností modelu generovat kód – nástroji je předán detailní prompt s popisem konkrétního testu (ze scénáře navrženého v předchozím kroku) a relevantním kontextem. Modul následně přijme od AI text zdrojového kódu testu a zapíše jej do souboru. V případě, že by generování selhalo (například model nevrátí použitelný výstup), implementace ošetřuje tuto situaci vytvořením prázdné kostry testu jako záložního řešení. Tím je zaručeno, že pro každý navržený scénář vznikne alespoň soubor testu – buď s vygenerovaným obsahem, nebo prázdný, který může vývojář později ručně doplnit. Tento postup zvyšuje robustnost: zamezí se úplnému výpadku generování při dílčím selhání AI a umožní pokračovat v procesu.

Modul spouštění testů a kolekce výsledků: Klíčovou součástí je integrace s frameworkem *ExTester* pro automatické spuštění vygenerovaných testů. *ExTester* interně využívá *Mocha test runner* a po zavolání příkazu `extest` si připraví samostatnou instanci *VS Code* ve které provede testy. Implementace obsahuje *runner*, jenž dokáže testy spustit a získat jejich výstup pro další analýzu. Bylo zvoleno řešení na bázi *VS Code Tasks* – rozšíření programově vytvoří a spustí úlohu typu *Shell*, která spustí testy v integrovaném terminálu. Při standardním spuštění testů by výstup proběhl pouze v terminálu, proto nástroj využívá přesměrování výstupu do souboru: testy se spouští s parametry tak, aby veškerý jejich konzolový výstup (*stdout* i *stderr*) byl zapsán do dočasného log souboru. Po dokončení běhu testů modul tento soubor načte a celý výpis (včetně případných chyb kompilace, selhání testů a *stack trace*) programově zpracuje. Pro parsování výstupu byl implementován pomocný parser: odstraní *ANSI* sekvence (barevné formátování) a na základě známé struktury výstupů *Mocha* identifikuje jednotlivé neúspěšné testy, jejich názvy, chybové hlášky, případně dotčený soubor a řádek.

Modul opravy testů: Posledním stavebním blokem je komponenta, která na základě zachycených výsledků umí opět využít AI k návrhu oprav neúspěšných testů. Integruje se s parserem výstupů – převezme informace o první nalezené chybě (nebo postupně o více chybách) a připraví odpovídající prompt pro LLM s žádostí o úpravu testu. Součástí promptu jsou detailní údaje: text chybové zprávy či *stack trace*, obsah problematického testovacího souboru (představující aktuální verzi kódu, který je třeba opravit) a kontext pluginu. AI model na tomto základě navrhne úpravu kódu testu. Návratový formát odpovědi je opět přímo zdrojový kód – implementace počítá s tím, že z odpovědi získaný kód přímo zapíše do souboru testu, čímž aplikuje AI opravu. Následně modul znovu spustí daný test a ověří, zdali úprava problém vyřešila, nebo zda přetrvávají další chyby.

Architektura tedy pokrývá celý cyklus: analýza vstupního rozšíření, návrh testovací sady, vytvoření a spouštění těchto testů v prostředí *VS Code*, vyhodnocení výsledků a případné automatizované opravy chyb. Jak uvádí metodika, jedná se o kombinaci generativního přístupu s mechanismy validace a úprav testů v rámci *ExTester* frameworku. V následujících podsekcích jsou detailně rozebrány nejdůležitější aspekty implementace a řešení problémů, které se objevily.

8.2 Generování testů

8.2.1 Analýza rozšíření

Proces generování testů začíná shromážděním informací o testovaném rozšíření. Rozšíření načte obsah `package.json` projektu a extrahuje z něj klíčové sekce, jež charakterizují funkcionalitu pluginu (viz výše). Tyto údaje následně strukturalizuje do objektu `relevantParts` – například: identifikátor pluginu, seznam příkazů a jejich popisů, položky nabídek, aktivační události, konfigurační volby atd. Cílem je poskytnout AI modelu kontext, jaké funkce a scénáře má vygenerovanými testy pokrýt. Na základě těchto podkladů je sestaven první prompt. Tento prompt modelu vysvětluje, o jaké rozšíření jde a jaké testy má navrhnout – zahrnuje shrnutí účelu pluginu a jeho rozhraní. Výstupem očekávaným od modelu je návrh sady testů, ideálně ve strukturované formě (např. pole objektů v *JSON*, kde každý objekt představuje jeden test se jménem, kategorií a popisem kroků).

8.2.2 Využití modelu GPT-5 vs GPT-5-Codex

Pro tuto fázi byl zvolen pokročilý jazykový model *GPT-5* (volaný prostřednictvím API) z důvodu jeho vysoké schopnosti porozumění kontextu a generování konzistentních, smysluplných návrhů i pro komplexnější scénáře. Model je instruován tak, aby výsledkem jeho odpovědi byl strojově čitelný seznam testů. Zde se v praxi ukázala jedna z výzev generativního přístupu – model často vracel výstup ve formátu obsahujícím *markdown* či vysvětlující text, přestože byl požadován čistý *JSON*. Implementace proto obsahuje post-processing krok, který přijatý text pročistí: odstraňuje případné *markdown syntaxe*, komentáře nebo neformální úvody, a snaží se izolovat platný *JSON*. Teprve poté je provedeno parsování *JSON* řetězce na datové struktury. Tento postup byl nezbytný, aby bylo možné automatizovaně zpracovat návrhy testů od AI – bez něj by sebemenší odchylka formátu (například uvozující fráze či chybějící uvozovky) způsobila selhání při parsování. V případě, že by se *JSON* nepodařilo rozpoznat, rozšíření chybu zaloguje a uživatele upozorní, že generování testů selhalo.

Jakmile se podaří získat strukturovaný seznam navržených testů, přistupuje nástroj k vlastnímu generování zdrojového kódu testů. V této fázi se uplatňuje druhý model – označený jako *GPT-5-Codex*. Jedná se v principu o použití API specializovaného na doplňování kódu. Model typu *GPT-5* výborně sumarizuje a navrhuje, ale při generování rozsáhlejšího kódu někdy sklouzává k nesprávné syntaxi nebo opomenutí detailů. Naproti tomu model trénovaný na kód produkuje syntakticky preciznější výsledky, zvláště pokud má dobře definované instrukce. Implementace tedy předává každou navrženou testovací situaci druhému modelu s podrobným promptem obsahujícím: popis konkrétního scénáře (co má test ověřit, jaké kroky v UI provádí), potřebný kontext (část manifestu s příkazem či funkcionalitou, které se scénář týká) a požadavek na vytvoření kompletního testu v syntaxi *ExTester* (tj. použití *Mocha* `describe/it` bloků a volání metod *Page Object* API z *ExTesteru*). Model v ideálním případě vrátí přímo blok zdrojového kódu *TypeScript* s testem – ten rozšíření přijme, opět z něj odstraní případné formátovací značky a zapíše do odpovídajícího souboru.

8.2.3 Generování jednoho vs. více testů

Původní záměr byl vygenerovat najednou celou sadu testů pokrývajících různé scénáře. Během implementace se však ukázalo jako výhodnější (z hlediska spolehlivosti a kontroly) generovat a skládat testy postupně. Rozšíření aktuálně zpracovává navržené testy sekvenčně. Tato volba souvisí jak s omezeními API (např. délka a komplexita promptu roste s počtem testů) tak i s praktickou zkušeností, že je snazší a výhodnější generovat jeden test po druhém. Každý test se generuje nezávisle stejným postupem: pro každý návrh se zavolá *Codex* model a výsledek se zapíše do vlastního souboru.

8.2.4 Ukládání testovacích souborů

Vygenerované testy jsou ukládány přímo do projektu testovaného rozšíření, aby je bylo možné ihned spustit v kontextu daného kódu. Implementace vytváří (pokud ještě neexistuje) složku `src/ui-test` v kořenovém adresáři projektu. Tato složka byla zvolena na základě konvence a výchozího nastavení *ExTesteru*. Pokud byly testy navržené AI rozděleny do kategorií (např. podle funkčnosti), nástroj vytvoří v rámci `ui-test` odpovídající podadresáře pro každou kategorii a do nich uloží testovací soubory. Samotné názvy testů (a souborů) se odvozují z názvů scénářů navržených AI – rozšíření zajistí, že výsledný soubor ponese například název `NazevTestu.test.ts`. Každý soubor obsahuje jeden nebo více `it` bloků v rámci jednoho `describe` bloku, dle povahy testu.

V případě, že generování obsahu testu AI modelem selže (např. API nevrátí odpověď v časovém limitu, nebo model odpoví neplatným kódem), implementace tuto chybu zachytí. Do logu se zaznamená informace o neúspěchu a daný test se vytvoří jako prázdný soubor s minimální kostrou. Tento *fallback* mechanismus byl přidán, aby nástroj vždy produkoval výstup pro každý scénář a vývojář měl možnost ručně doplnit chybějící část, místo aby byl celý proces kvůli jednomu selhání způsobil zásadní problémy. Z pohledu *ExTesteru* prázdný testový soubor nepředstavuje problém – může být přeskočen nebo projde (pokud neobsahuje žádnou aserci). Tato vlastnost tedy zajišťuje kontinuitu procesu generování.

Po úspěšném vygenerování testů rozšíření zobrazí uživateli shrnutí – například kolik testovacích souborů bylo vytvořeno a ve které složce. Uživatel má možnost nahlédnout do vygenerovaných testů, upravit je či spustit. V této fázi vstupuje do procesu další významná část implementace: automatická validace testů.

8.3 Spouštění testů a detekce chyb

Jakmile jsou testy vygenerovány, je třeba ověřit, zda jsou funkční – tedy zda jdou zkompilovat a úspěšně spustit proti testovanému rozšíření. Nástroj toto ověřuje automaticky pomocí integrace s *ExTesterem*. Klíčovou výhodou *ExTesteru* je, že dokáže kompletně obstarat běh UI testů ve *VS Code* – stáhne potřebnou verzi *VS Code*, spustí novou instanci s nainstalovaným testovaným pluginem a uvnitř ní provede testy. Výstup z testů (*log*) vypisuje do konzole.

8.3.1 Řešení konfliktu více běhů a prostředí Mocha

Z počátku se nabízelo spustit celý testovací balík jedním příkazem a čekat na výsledek. V praxi se ale objevily komplikace s opakovaným spouštěním testů a s interpretací jejich výsledků. Při vývoji se ukázalo, že opakované volání *ExTester* (zejména rychle po sobě) může vést ke konfliktům – např. pokud předchozí instance *VS Code* ještě neběží nebo není správně ukončena, nová se nespustí, nebo dojde k chybě při pokusu navázat *WebDriver* spojení. Kritickým problémem bylo, že *ExTester* při opakovaném spouštění narážel na konflikt v parametru `-user-data-dir`, protože nová instance *VS Code* se pokoušela využít již obsazený adresář uživatelských dat, což vedlo k chybě. Tato kolize zcela znemožnila běh testu a vedla k selhání. Rovněž se projevila určitá nestabilita *Mocha test runneru* při opakovaných bězích: *ExTester* spouští testy v novém *VS Code*, ale výsledky poskytuje do stejného terminálu. Při více testovacích souborech se stávalo, že selhání jednoho testu ovlivnilo následující (například pokud test zhavaroval na neošetřené výjimce, další testy se už nespustily a celý běh byl ukončen s chybou). Tyto faktory komplikovaly jednoznačnou detekci, které testy neprošly a proč.

8.4 Automatická oprava chyb

Jednou z hlavních inovací řešení je zavedení iterativního cyklu generování a oprav, který zajišťuje, že výsledkem jsou funkční testy. Tato smyčka sestává ze dvou kroků: nejprve řešení případných kompilačních chyb (syntaktické a obdobné problémy bránící spuštění testu) a poté řešení runtime chyb (chyby logiky testů při běhu). Oba kroky využívají LLM k automatizovaným opravám, ale liší se přístupem v detailu.

8.4.1 Oprava syntaktických chyb

Rozšíření má pro tento účel definován příkaz `Fix Compilation Issues`, který uživatel může spustit. Pokud parser výstupu testů odhalí nějaké selhání, nástroj nejprve zjišťuje, zda jde o chybu kompilace. Typickým příznakem je, že testy se ani nespustily nebo selhaly hned na začátku s chybou syntaxe (např. `Unexpected token`, `Cannot find module` apod.). Taková chyba se obvykle projeví u prvního testovacího souboru a zastaví běh celé sady. Tento příkaz opět běží v kontextu *VS Code* s indikačním pruhem průběhu a provádí následující:

1. **Spuštění testů:** Nejprve dojde k pokusu o kompilaci testů s cílem získat aktuální výstup. To je užitečné, pokud mezitím uživatel například testy upravil ručně – nástroj tak pracuje s čerstvými informacemi. Pokud tentokrát testy lze zkompilovat bez chyby, rozšíření oznámí, že žádné kompilační chyby nejsou a situace je vyřešena. V opačném případě parser opět dodá seznam selhání.
2. **Získání detailů o chybě:** Nástroj vezme první nalezenou chybu v seznamu a shromáždí kontext k ní. Kromě samotné chybové zprávy a názvu testu se pokusí načíst i obsah testovacího souboru, ve kterém k chybě došlo. Dále znovu připraví objekt se širším kontext o pluginu.

3. **Sestavení promptu pro opravu:** Pro vygenerování opravy se využívá speciální prompt. Tento prompt definuje roli modelu jako „asistent pro opravu testů v *TypeScriptu* pro *ExTester*“ a zahrnuje: text chybové hlášky či výňatek *stack trace*, aktuální obsah testovacího souboru a související kontext. AI je požádána, aby navrhla úpravu kódu tak, aby kompilační chyba byla vyřešena.
4. **Aplikace opravy:** Rozšíření obdrženou odpověď (typicky blok kódu) pročistí a výsledný kód se pak zapíše zpět do zdrojového souboru testu.
5. **Opětovné spuštění testů:** Po zapsání úpravy se automaticky spustí znovu celý balík testů. Pokud nyní testy lze zkompilovat, uživatel je informován o úspěchu. Pokud se stále objevuje další kompilační chyba, celý cyklus se vrací do kroku 2 a opakuje až do situace, kdy po poslední iteraci již vygenerované testy neobsahují syntaktické chyby.

Tímto způsobem je zajištěno, že výsledné testy jsou syntakticky správné a připravené k běhu. Tento krok v podstatě implementuje doporučovaný mechanismus automatické validace syntaxe AI výstupů – diplomová práce už v analytické části zdůraznila, že bez takové validace může množství syntaktických chyb znehodnotit přínos AI generování testů. Náš nástroj tyto kontroly plně automatizuje a šetří tak čas, který by vývojář strávil ručním hledáním překlepů či špatných importů.

8.4.2 Oprava sémantických chyb

Druhou úrovní problémů, které mohou nastat, jsou chyby v logice testů – tedy situace, kdy test sice běží, ale neprojde správně, typicky kvůli nesplněné aserci nebo nečekané výjimce během interakce s UI. Tyto runtime chyby znamenají, že test byl syntakticky v pořádku, ale buď špatně ověřuje výsledky, nebo nesprávně zachází s prostředím. Příklady mohou být: AI špatně odhadla očekávaný text v UI, vynechala nějaký krok (test pak narazí na prvek, který není připraven), nebo testovaná akce proběhla jinak, než model předpokládal.

Pro automatizovanou opravu runtime chyb slouží příkaz *Fix Runtime Failures*. Implementace tohoto kroku se oproti kompilačním chybám ukázala složitější. Jednoduchý přístup využitý při opravě syntaktických chyb se příliš neosvědčil. Jednak při více selhávajících testech AI oprava jednoho nemusela vyřešit další a bylo nutné spouštět testy opakovaně, jednak více současných chyb v jedné sadě pletlo model (smíchaly se kontexty různých testů). Rovněž technicky nastal problém s opakovaným spouštěním celé testovací sady v jedné instanci *VS Code*. Instance mohla zůstat v nekonzistentním stavu po předchozím neúspěšném testu (např. neuzavřený dialog či nevrácené UI do výchozího stavu). To by ovlivnilo následné běhy.

8.4.3 Izolované spouštění po jednotlivých testech

Řešením bylo změnit strategii – detekovat a opravovat chyby po jednom testovacím souboru. Nástroj nejprve zjistí, jaké testovací soubory v projektu existují (vygenerované testy). Poté iteruje jeden soubor po druhém a pro každý provede samostatný běh. Tím pádem

v každém běhu běží izolovaně pouze jeden testovací scénář (resp. skupina v jednom souboru). To přináší dvě výhody:

- Případný pád jednoho testu neovlivní ostatní – po skončení testu *ExTester* ukončí instanci *VS Code* a pro další test se otevře nová, čistá instance,
- výstup je jednodušší a parser se soustředí jen na chyby z jednoho testu.

Pro každý testovací soubor se tedy provede běh a vyparsuje výstup. Pokud test prošel, není třeba nic dělat. Pokud v testu nastaly selhání, filtrujeme z nich pouze ty relevantní k samotnému běhu. Může se stát, že v jednom testovacím souboru je více `it` bloků a selhalo jich několik. Implementace v aktuální verzi řeší vždy jen první nalezenou chybu z daného souboru, případně postupně několik, ale po každé opravě opět test spustí a ověří, zda už celý soubor projde. Konkrétně: Pro každý soubor vezme první selhání, zavolá AI na opravu (podobně jako u kompilačních – poskytne prompt s informacemi o chybě, kódem testu a kontextem pluginu) a zapíše upravený kód. Následně ihned spustí znovu tentýž testovací soubor k ověření. Pokud test nyní celý projde, považuje se oprava za úspěšnou a nástroj pokračuje na další soubor. Pokud stále některá část souboru selhává, cyklus se zopakuje – vezme další (resp. na listu chyb první) selhání a pošle opět modelu k opravě. Tímto iterativním procesem se u jednoho testu může provést více oprav, dokud není daný testovací soubor bez chyby, anebo dokud nedojdou detekované chyby (pak by zůstalo na uživateli zkontrolovat, co ještě není v pořádku).

Tato strategie se ukázala být stabilnější. Izolované spuštění každého testu zamezilo konfliktům mezi testy a také vyřešilo problém s více instancemi *Mocha/VS Code* – každá oprava startuje s čistým stavem.

8.4.4 Generativní opravy vs. manuální zásah

Během vývoje se ukázalo, že model *GPT-5* často dokáže správně navrhnout úpravu testu na základě jediné chybové zprávy. Nicméně ne vždy je oprava kompletní – v některých případech model opraví první problém, ale v testu je skryta další logická chyba, kterou odhalí až další běh. Proto je iterativní postup s opakováním testů po každé úpravě nezbytný. V metodice práce bylo avizováno, že AI nenahradí plně lidskou kontrolu a že finální doladění má dělat vývojář. Implementované rozšíření se snaží tuto mezeru co nejvíce zúžit tím, že opakovaně aplikuje AI opravy, dokud testy celé neprojdou. Přesto je vývojářům doporučeno výsledné testy zkontrolovat. Nástroj jim v tom pomáhá i tím, že veškeré kroky loguje do výstupního kanálu a v případě, že se ani po více pokusech nepodaří test opravit, upozorní je na nutnost ručního zásahu.

Na závěr této fáze rozšíření uživateli oznámí, kolik chyb bylo automaticky opraveno a zda všechny testy prošly. Pokud by i poté ještě nějaké testy selhávaly, nástroj to oznámí varováním a vývojář může buď spustit další iteraci generování/opravy, nebo chyby dořešit ručně.

8.5 Implementační výzvy a přijatá řešení

Při vývoji se objevila řada detailnějších problémů, které bylo nutné překonat, aby nástroj fungoval spolehlivě v běžných podmínkách. Následuje výčet nejdůležitějších z nich spolu s popisem řešení.

8.5.1 Volba a kombinace jazykových modelů

Důležitým rozhodnutím bylo, jaký AI model (či modely) použít pro generování testů. Teoretická analýza naznačovala výhody *GPT-5* oproti starším modelům z hlediska kvality generovaného kódu. V praxi se potvrdilo, že *GPT-5* dokáže navrhnout realistické scénáře testů a poradí si i se složitějším zadáním. Nicméně pro samotné generování kódu testů bylo vhodné využít model optimalizovaný pro kód (*Codex*). Proto byl implementován hybridní přístup: *GPT-5* pro návrh a *GPT-5-Codex* pro generování kódu. Toto rozhodnutí zlepšilo syntaktickou správnost výstupů, ale vyžadovalo více volání na API. V metodice se původně uvažovalo i o možnosti natrénovat vlastního menšího modelu či použít open-source LLM (např. *Code LLaMA*) – realizace však ukázala, že integrace s dostupnou službou (*OpenAI ChatGPT*) je nejrychlejší cestou k funkčnímu prototypu, což je v souladu se závěry srovnání nákladů a přínosů vlastního modelu vs. cloudové služby. Do budoucna by šlo implementaci doplnit o možnost volby modelu v nastavení (např. přepnout na levnější *GPT-3.5-Turbo* pro méně náročné generování, nebo umožnit integraci jiné API).

8.5.2 Získávání výstupu testů

Jedna z největších výzev byla spolehlivá komunikace mezi běžícím testem a rozšířením. *VS Code* neumožňuje jednoduše číst obsah integrovaného terminálu, proto nebylo možné prostě spustit testy viditelně a číst jejich výstup z terminálu API. Řešením bylo zmíněné přesměrování do souboru. Tomu musela být přizpůsobena i parser logiky: původně se uvažovalo o použití *problem matcheru VS Code* (který umí zachytávat chyby z výstupu při běhu *VS Code Tasks*). Proto si rozšíření implementuje vlastní čtení a parsing. Při vývoji došlo k situacím, kdy *ExTester* proces běžel, ale log soubor zůstal prázdný (např. při havárii procesu). Nyní je ošetřeno i to – pokud je výstup prázdný nebo nečitelný, uživatel dostane chybové hlášení s informací, aby nahlédl do logu (příp. do konzole vývojáře *VS Code*) pro více detailů.

8.5.3 Stabilita a ladění parseru

Zpočátku parser nerozlišoval přesně typy chyb a bral každý neúspěch stejně. To vedlo k situacím, kdy AI byla vyzvána k „opravení runtime chyby“, i když ve skutečnosti šlo o syntaktický překlep – model pak navrhoval zbytečně složité změny, nebo naopak při syntaktické chybě dostal příliš stručný prompt. Zavedení klasifikace chyb (kompilace vs. runtime) tento problém vyřešilo – prompt pro opravu nyní vždy odpovídá povaze chyby. Například při kompilační chybě obsahuje prompt celý kód souboru a žádá opravu syntaxe, zatímco u runtime chyby obsahuje i text aserce a soustředí se na logiku testu. Dále bylo

nutné ošetřit některé okrajové formáty výstupu *Mocha*: například když test spadne ještě před tím, než se zaregistruje (chyba v *before hooku* apod.), výstup vypadá jinak. Parser byl proto rozšířen, aby pokryl i tyto varianty (např. hledá klíčová slova „*Unhandled Exception*“ atd. a přiřadí je k runtime chybám s odpovídající zprávou).

8.5.4 Uživatelská kontrola a zásahy

Ačkoli nástroj se snaží maximum práce obstarat automaticky, obsahuje několik interakcí od uživatele. Například po prvotním generování testů se testy nespouštějí zcela bez vědomí uživatele – ten musí aktivně zvolit spuštění dalších kroků. To dává vývojáři možnost zkontrolovat a upravit generované testy před kompilací či spuštěním. Také při aplikaci AI oprav, pokud by nástroj nevěděl, kam změnu zapsat, vyzve uživatele k zadání cesty. V logu jsou všechny AI návrhy viditelné, takže zkušený vývojář může posoudit, zda změnu přijme, případně ji dodatečně editovat. Tyto mechanismy přispívají k použitelnosti a bezpečnosti – nástroj se snaží asistovat, ale finální kontrolu ponechává na programátorovi, jak ostatně doporučují i odborné studie kombinovat AI asistenci s lidským dohledem.

9 Výsledky a diskuse

Tato kapitola prezentuje dosažené výsledky práce, zejména prototyp vytvořeného nástroje a návrh metodiky jeho vyhodnocení. Nejprve je představen vyvinutý plugin pro *Visual Studio Code*, který umožňuje automatické generování testů uživatelského rozhraní pomocí generativní AI. Následně je popsán typický workflow jeho použití v praxi. Na závěr jsou definována kritéria hodnocení kvality generovaných testů včetně metrik, podle nichž bude efektivita navrženého řešení posuzována.

9.1 Popis vytvořeného nástroje

Hlavním výstupem práce je funkční rozšíření integrované pro *Visual Studio Code (VS Code)*, které staví na testovacím frameworku *ExTester* a přidává generativní schopnosti. Rozšíření načte manifest cílového *VS Code* rozšíření (`package.json`), navrhne strukturované scénáře testů uživatelského rozhraní, a pro každý z nich vygeneruje spustitelné *TypeScript* testy pomocí modelů *OpenAI*. Tyto testy dále umí kompilovat, spouštět a opravovat. Zachytí překladové i běhové chyby, zanalyzuje jejich výstup a navrhne potřebné úpravy testů. Ty následně zapíše zpět testovacích scénářů a ověří jejich správnost novým během. Celý proces běží přímo ve *VS Code* – v kartě `ExTester Test Generator` spustíte jednotlivé kroky z panelu či *Command Paletty* (generování návrhů, opravu kompilace, opravu runtime chyb), průběh vidíte v notifikacích a detailní log v dedikovaném output kanálu.

9.1.1 Generování testů

Nástroj využívá velké jazykové modely (konkrétně *GPT-5* a *GPT-5-Codex*) k návrhu scénářů testů uživatelského rozhraní a generování jejich zdrojového kódu. Model vytváří testy v jazyce *TypeScript* nad *ExTester* API, tedy nad sadu vyšší úrovně metod pro ovládání *VS Code* (otevírání panelů, spouštění příkazů, interakce s prvky rozhraní apod.). Aby bylo generování co nejrelevantnější, rozšíření částečně aplikuje princip RAG (*Retrieval-Augmented Generation*): před dotazem na model analyzuje manifest testovaného rozšíření, konkrétně `package.json`, ze kterého získá seznam dostupných příkazů, aktivačních událostí, konfiguračních voleb a dalších metadat. Tato data se promítnou do promptu jako kontext, takže generativní AI navrhuje testy, které odpovídají reálným schopnostem daného pluginu a snižuje se riziko, že bude testovat neexistující nebo nerelevantní chování.

9.1.2 Funkcionality a integrace do VS Code

Rozšíření poskytuje uživatelsky přívětivé rozhraní přímo v prostředí *VS Code*. Po instalaci se objeví v kartě `ExTester Test Generator` v postranním panelu, kde nabízí vlastní stromové zobrazení s položkami pro jednotlivé kroky procesu. Tyto akce lze spouštět jak z panelu, tak z *Command Paletty*.

Mezi hlavní funkce rozšíření patří:

1. **Návrh a generování testů jedním kliknutím**: Akce `Generate Test Proposals` načte workspace `package.json`, připraví souhrn manifestu a pošle jej modelu *GPT-5*, který vrátí strukturované návrhy testů uživatelského rozhraní. Následně je pro vybrané návrhy vygenerován zdrojový kód testů v *TypeScriptu* nad *ExTester* API.
2. **Automatické vytvoření a zpřístupnění výsledku**: Vygenerované testovací skripty jsou automaticky uloženy do projektu do testové struktury. Uživatel s nimi pracuje jako s běžnými soubory v projektu, může je ihned otevřít v editoru, upravovat a verzovat.
3. **Spouštění testů přes ExTester**: Rozšíření integruje volání `ExTester stacku` a umí testy spouštět[21] v rámci *VS Code* úloh. Průběh běhu je dostupný v integrovaném terminálu a výstup slouží jako vstup pro následnou analýzu chyb.
4. **Iterativní vylepšení a automatické opravy testů**: Rozšíření dokáže zachytit chyby vzniklé při kompilaci i běhu testů a využít AI k návrhu jejich automatické opravy. Uživatel má k dispozici akci `Fix Compilation Issues` pro automatické opravení typových, syntaktických či API chyb a akci `Fix Runtime Failures` pro úpravu testu při selhání za běhu (např. doplnění čekání, úprava asercí či navigace v uživatelském rozhraní). Nástroj změny aplikuje do příslušných testovacích souborů, test znovu spustí a celý proces doprovází průběžnými notifikacemi a detailním logem v dedikovaném output kanálu, což výrazně zvyšuje robustnost a snižuje nutnost ručního ladění.

9.1.3 Workflow použití

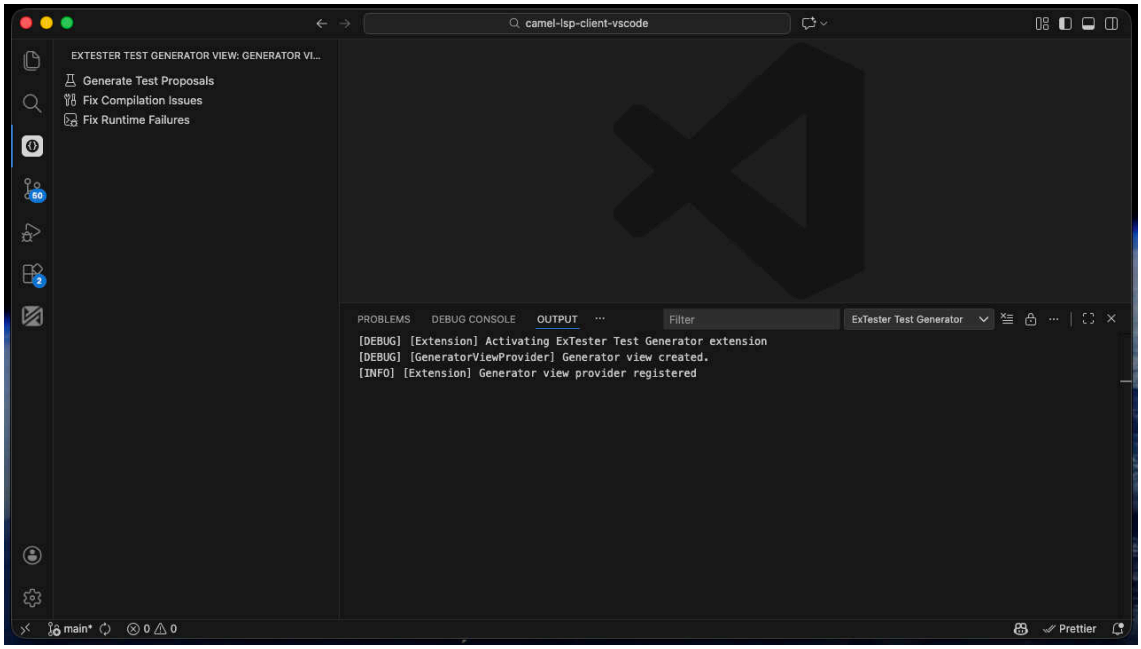
Typický způsob použití rozšíření lze shrnout do následujícího pracovního postupu:

1. **Příprava prostředí**: Uživatel nejprve otevře v prostředí *Visual Studio Code* projekt testovaného rozšíření, provede instalaci závislostí (`npm install`) a v nastavení *VS Code* nakonfiguruje hodnotu `extester-test-generator.apiKey`, aby bylo možné využívat služeb generativní AI.
2. **Generování počáteční sady testů**: Poté uživatel otevře v postranním panelu kartu `ExTester Test Generator` a spustí příslušný příkaz pro vygenerování návrhů a zdrojových kódů testů. Rozšíření na základě informací z projektu vytvoří nové testovací soubory, které jsou dále dostupné k manuální kontrole a úpravám.
3. **Ladění kompilace testů**: Následuje fáze ověření, zda testy bezchybně procházejí kompilací. Uživatel spustí testovací skript definovaný v projektu. Pokud dojde k chybám při překladu či typové kontrole, využije akci `Fix Compilation Issues`, která na základě chybového výstupu navrhne a aplikuje úpravy testů. Tento krok se opakuje do doby, než testy kompilují bez chyb.
4. **Ladění běhového chování testů**: Jakmile jsou testy kompilovatelné, zaměří se uživatel na jejich běhové chování. Testy opět spustí pomocí *ExTester* frameworku a v pří-

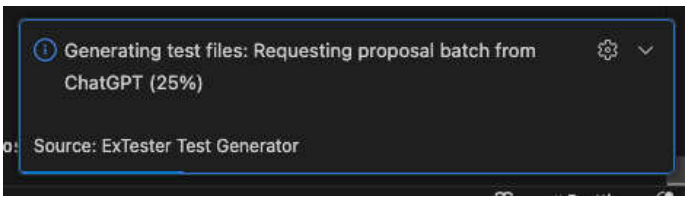
[21] Rozšíření umí spouštět testy pro svoje účely, nikoliv na základě požadavku uživatele.

padě selhání využije akci `Fix Runtime Failures`, která na základě výsledků běhu navrhne úpravy kroků testu (např. synchronizace s uživatelským rozhraním, aserce). Po aplikaci změn jsou testy znovu spuštěny a proces se iterativně opakuje, dokud nedojde k ustálení chování.

5. **Stabilizace a další využití**: Výsledkem popsaného workflow je stabilní sada testů uživatelského rozhraní, která pokrývá klíčové scénáře testovaného rozšíření. Tuto sadu může uživatel dále udržovat a rozšiřovat (ručně i s pomocí AI funkcí rozšíření) a začlenit ji do kontinuální integrace jako standardní součást testovacího procesu.



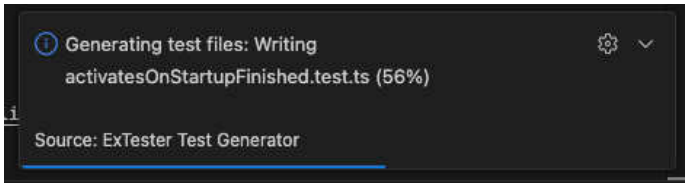
Obrázek 1: Rozhraní implementovaného zásuvného modulu a výstup loggeru.



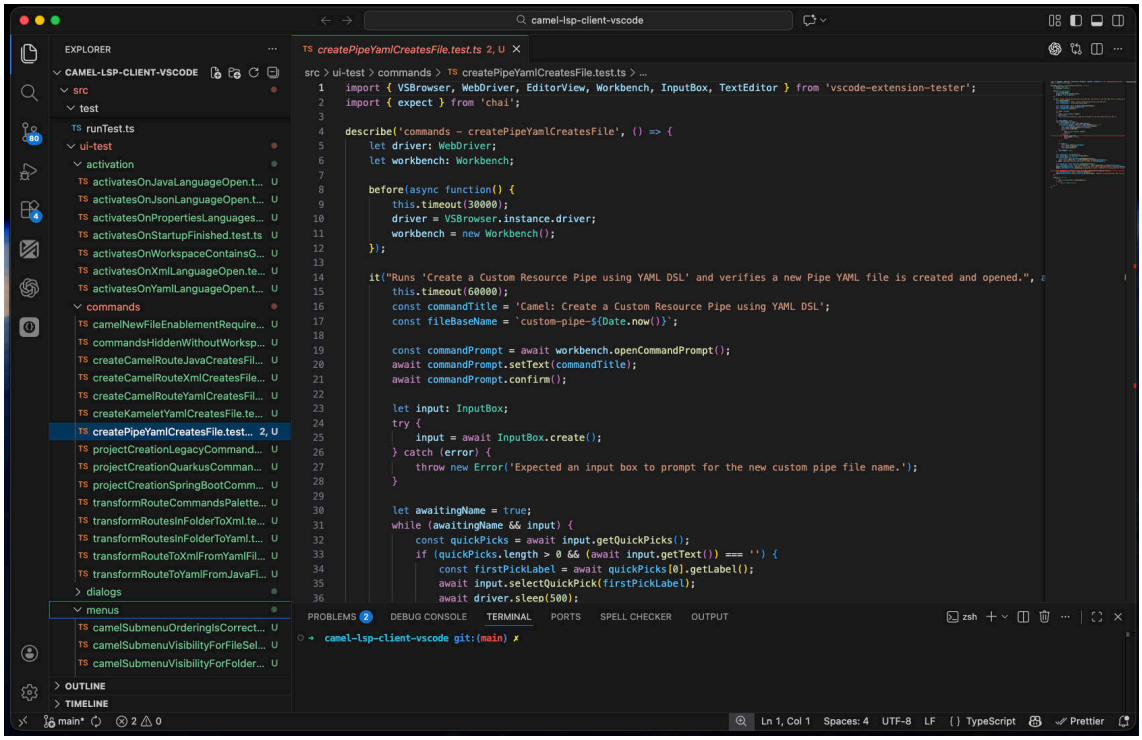
Obrázek 2: Generování návrhu testovacích scénářů.

9.2 Metodika hodnocení

Jak bylo popsáno v sekci 7.4, vyhodnocení se zaměřuje především na generativní část RAG systému – tedy na kvalitu a přínos automaticky generovaných testů. Cílem je posoudit,



Obrázek 3: Generování specifického testu.



Obrázek 4: Vygenerovaný testovací scénář.

nakolik navržené řešení splňuje očekávání a do jaké míry dokáže AI asistence zefektivnit proces testování. Pro systematické zhodnocení byly definovány následující metriky:

Věrnost (faithfulness): Tato metrika hodnotí, do jaké míry obsah vygenerovaného testu odpovídá zadání scénáře. Věrnost znamená, že testovací kroky přesně reflektují požadované uživatelské akce a ověřují stanovené očekávané výsledky, aniž by opomíjely podstatné části scénáře nebo naopak přidávaly něco navíc.[43] Hodnocení bude prováděno na Likertově škále 0–4, kde 0 představuje zcela neodpovídající test (výstup se výrazně liší od zadaného scénáře) a 4 značí plně věrný test (výstup přesně pokrývá celý scénář bez odchylek).

Podloženost (groundedness): Metrika podloženosti posuzuje, nakolik je generovaný test podložen dostupnými informacemi o systému a využívá pouze relevantní funkcionality. Jinými slovy jde o to, zda všechny akce a aserce v testu dávají smysl vzhledem k existujícím vlastnostem testovaného rozšíření. V ideálním případě se AI “drží při zemi”

– testuje jen to, co daný plugin skutečně umí (např. volá reálně existující příkazy, otevírá skutečně dostupné prvky uživatelského rozhraní).[4] Hodnocení proběhne opět na škále 0–4: 0 = nulová podloženost (test obsahuje převážně nereálné či neexistující prvky), 4 = výborná podloženost (všechny kroky testu odpovídají reálným funkcím a dokumentaci pluginu).

Míra halucinací (hallucination rate): Tato kvantitativní metrika doplňuje podloženost a udává podíl nerelevantního či fakticky nesprávného obsahu vygenerovaného testu. Konkrétně se sleduje, kolik kroků nebo tvrzení v testu lze označit za halucinaci (tj. výmysl modelu bez opory v kontextu nebo funkčnosti pluginu).[43] Míra halucinací bude vyjádřena procentuálně – např. pokud ze 20 kroků testovacího skriptu 2 kroky odkazují na neexistující prvky, míra halucinace činí 10 %. Nižší hodnoty jsou samozřejmě lepší; ideálním cílem je míra halucinací 0 %, což by znamenalo, že AI nevygenerovala žádný nesmyslný či nepodložený krok. Tato metrika se vyhodnotí analýzou testu.[38]

Technická správnost výstupu (answer correctness): Tato metrika ověřuje, zda jsou generované testy po technické stránce správné a spustitelné. Zahrnuje to jak syntaktickou správnost kódu (test se musí zkompilovat/interpretovat bez chyb), tak logickou správnost sekvence kroků (test by na správně fungující aplikaci měl projít a na aplikaci s chybou adekvátně selhat).[22] Technická správnost se bude hodnotit binárně (vyhovuje / nevyhovuje): každý vygenerovaný test buď splňuje požadavky (jde spustit a chová se očekávaně), nebo nesplňuje (např. obsahuje chybnou syntaxi, volá neexistující metodu, případně vždy selže i na bezchybné aplikaci).

Čitelnost kódu: Tato metrika se zaměřuje na kvalitu formy generovaných testů – tedy jak přehledný a srozumitelný je výsledný testovací kód. I správně fungující test může být napsán nejasně nebo zbytečně komplikovaně, což by snižovalo jeho praktickou použitelnost pro tým.[4][20] Čitelnost bude hodnocena subjektivně na škále 0–4, kde 0 znamená velmi nečitelný kód (např. zmatená struktura, špatné pojmenování, absence komentářů) a 4 značí výbornou čitelnost (kód je strukturovaný, dobře okomentovaný, snadno pochopitelný). Tato doplňková metrika pomůže zhodnotit, zda výstupy AI nejsou jen funkční, ale také udržovatelné z pohledu lidského čtení.

Kromě výše uvedených hodnoticích metrik bude v rámci experimentu sledována také kvalita výstupu v jednotlivých fázích generativního procesu. Celý pipeline RAG systému totiž neprodukuje finální test okamžitě – typický tok zahrnuje tři po sobě jdoucí kroky (prvotní generování, opravu kompilace a opravu běhových chyb), z nichž každý má odlišnou charakteristiku a může odhalit jiné typy nedostatků. Proto je součástí metodiky i vícestupňové hodnocení, které posuzuje kvalitu v jednotlivých fázích a jejich příspěvek ke konečnému výsledku.

9.3 Volba testovaného rozšíření

Pro účely experimentálního vyhodnocení byl zvolen open-source plugin *camel-lsp-client--vscode* dostupný ve veřejném repozitáři[7]. Jedná se o oficiální *VS Code* klientskou implementaci *Language Server Protocolu* (LSP) pro technologii *Apache Camel*. Rozšíření je v rámci ekosystému *VS Code* široce používáno, je aktivně vyvíjeno komunitou pod hlavič-

kou projektu *Camel Tooling* a představuje vhodný zástupný příklad středně komplexního reálného pluginu.

Výběr tohoto rozšíření nebyl náhodný – splňuje několik důležitých kritérií relevantních pro evaluaci generovaných testů uživatelského rozhraní:

Reálná komplexita: Plugin využívá LSP komunikaci, registruje několik příkazů, pracuje s kontextovými nabídkami, notifikacemi a interaguje s editorem. Díky tomu poskytuje dostatek scénářů, které nejsou triviální a vyžadují více kroků interakce v rámci uživatelského rozhraní.

Stabilní a přehledná codebase: Projekt má dobře strukturovaný kód v *TypeScriptu*, včetně rozdělení na aktivační logiku, registrace příkazů, synchronizaci s jazykovým serverem a podporu různých DSL pro *Camel*. To je ideální pro sledování toho, jak si AI poradí s analýzou kódu a návrhem testů.

Dostupnost zdrojů a otevřenost projektu: Jako open-source plugin umožňuje detailní inspekci implementace, snadné instrumentování a bezproblémové začlenění do testovacího prostředí. To eliminuje nutnost řešit licenční omezení a usnadňuje opakovatelnou reprodukci experimentu.

Interakce uživatelského rozhraní v rámci *VS Code*: Rozšíření obsahuje typické uživatelské scénáře (např. otevření *Camel* souboru, zobrazení diagnostiky, nabídka code-actions, aktivace příkazů), které jsou vhodné pro testování prostřednictvím frameworku *ExTester*. Tyto scénáře zároveň poskytují ideální základ pro měření věrnosti, relevance a správnosti AI-generovaných testů.

Předchozí znalost a doménová zkušenost: Vzhledem k tomu, že se jedná o rozšíření, na jehož vývoji jsem se v minulosti podílel, jsou jasně známy funkční požadavky, běžné scénáře chování i okrajové případy. To umožňuje objektivněji posoudit, nakolik AI generuje smysluplné testy, které odpovídají skutečným očekáváním.

Plugin tak představuje vhodný kompromis mezi realistickou složitostí a možností přesně vyhodnotit kvalitu výsledků. Zároveň umožňuje testovat různé úrovně interakce — od jednoduchých příkazů až po komplexní workflow zahrnující jazykový server — což činí experiment reprezentativním i pro širší kategorii *VS Code* rozšíření, která pracují s LSP, strukturovanými dokumenty nebo generují diagnostiku.

9.4 Generate test proposals

9.4.1 Návrh testovacích scénářů

První fáze workflow představuje analýzu samotného rozšíření, návrh testovacích scénářů a jejich generování. Seznam navrhnutých testovacích scénářů je uveden v příloze A. Z pohledu vývojáře dává seznam smysl, výjimku tvoří kategorie `activation`. Ačkoliv návrh dobře reflektuje dostupná data o aktivačních událostech samotného zásuvného modulu, většina testů v téhle kategorii nedává smysl. Testovaný modul aktivuje spolu s událostí `On Startup Finish`. S ohledem na vnitřní logiku *VS Code* tato událost vždy nastane dříve, než některá z dalších definovaných událostí. To znamená, že i naprosto korektně vygenerovaný test pro událost `On Java Language Open` (či pro jinou událost) by mohl končit

falešně pozitivně, neb samotné rozšíření je ve stavu `Activated` již ve chvíli, kdy test teprve inicializuje své rozhraní a chystá se otevřít soubor typu *Java* či jiný. V některých případech můžeme pozorovat částečnou duplicitu testovaného obsahu, kdy by bylo možné některé testy například parametrizovat nebo zcela vynechat.

- **Věrnost**, hodnocení: 3. Vygenerovaný návrh dobře reflektuje potřeby modulu. Výjimku tvoří kategorie `activation`, viz popis výše.
- **Podloženost**, hodnocení: 4. Veškeré navrhnuté testovací scénáře vycházejí z dostupného kontextu a jsou tedy plně podloženy.
- **Míra halucinací**, hodnocení: 0 %. Žádný z navrhnutých testovacích scénářů neodpovídá halucinaci.
- **Technická správnost výstupu**, hodnocení: nelze hodnotit.
- **Čitelnost kódu**: hodnocení: nelze hodnotit

9.4.2 Generování testovacích scénářů

Zásuvný modul v této fázi vygeneruje obsah jednotlivých testovacích scénářů. Celkem v této fázi bylo vygenerováno 41 testů. 11 testů bylo vygenerováno bez kompilačních chyb, 30 testů obsahovalo nejméně jednu kompilační chybu. Celkem testy obsahovaly 118 kompilačních chyb různého charakteru. Těmto chybám se bude více věnovat následující sekce.

Pro další kroky a vyhodnocování byl vybrán jeden test z každé kategorie. To celkem tvoří vzorek 6 testovacích scénářů. Jedná se o následující testy, které budou dále reprezentovány pouze svoji kategorií.

- **`activation`**`/activatesOnJsonLanguageOpen.test.ts`
- **`commands`**`/projectCreationQuarkusCommandAvailable.test.ts`
- **`dialogs`**`/multiFileTransformXmlOpensFilePicker.test.ts`
- **`menus`**`/camelSubmenuOrderingIsCorrect.test.ts`
- **`settings`**`/camelCatalogVersionSettingPersists.test.ts`
- **`views`**`/welcomeCreateProjectLinkInteraction.test.ts`

Všechny hodnocené testy velmi dobře reflektují věrnost jejich očekávanému scénáři. Samotné scénáře jsou vhodně podložené na základě analyzovaného manifestu. Test v kategorii *dialogs* je v současné fázi přeložitelný a spustitelný, nicméně vykazuje běhové chyby. Ostatní testy nejsou přeložitelné a ani spustitelné.

[22] Žádný z testů není v dané fázi kompilovatelný bez chyby.

[23] Aserce testu jsou ve své podstatě korektní, s ohledem na výše popsanou situaci s aktivací ale žádná tato aserce reálně nebude otestována validně.

Oblast	Věrnost	Podloženost	Halucinace	Technická správnost[22]	Čitelnost
Activation	4	3	100 %[23]	0	4
Commands	4	4	0 %	0	2
Dialogs	4	3	0 %	0	3
Menus	3	4	0 %	0	3
Settings	4	4	0 %	0	3
Views	3	3	0 %	0	2

Tabulka 1: Hodnocení generovaných výstupů po prvním kroku

9.5 Oprava kompilačních chyb

Cílem tohoto kroku je úspěšně opravit všechny kompilační chyby. V optimálním případě by mělo dojít k opravě tak, jak by ji vykonal vývojář.

9.5.1 activations/activatesOnJsonLanguageOpen.test.ts

Vygenerovaný test vykazuje dvě typově shodné kompilační chyby. Jedná se o chybu typu `Property does not exist on type`, tedy kód se snaží využívat neexistující funkce. Opravený kód vykazuje větší množství změn. Kromě samotné opravy kompilačních chyb je změněná i logika v podobě dosažení aktivačního bodu – otevření souboru typu *JSON*. Zatím co původně vygenerovaný test se snaží otevřít soubor `tasks.json`[24] a v případě že neexistuje si jej vytvoří i s validním obsahem, opravený kód k dosažení aktivačního eventu otevírá soubor `settings.json`[25], ke kterému lze přistoupit pomocí command paletty a existuje vždy. Kompilační chyby jsou tedy opraveny validně. Test je v této fázi přeložitelný a spustitelný. Test ovšem neprochází a končí běhovou chybou typu `Unable to locate element`[26].

- **Věrnost**, hodnocení: 4. Oprava reflektuje očekávané řešení.
- **Podloženost**, hodnocení 4: I přes větší změny v logice testu vše vychází z dostupného kontextu a reflektuje potřeby testu.
- **Míra halucinací**, hodnocení: 0 %. Nevykazuje žádný zbytečný kódu.
- **Technická správnost výstupu**, hodnocení: 0. Vykazuje běhové chyby.
- **Čitelnost**, hodnocení: 4. Kód je dobře strukturovaný a i nezkušený uživatel z něj pochopí základní scénáře a principy.

[24] Soubor, který definuje vlastní spustitelné úlohy (např. kompilaci, testy nebo skripty).

[25] Soubor, ve kterém si *VS Code* ukládá konfiguraci editoru a rozšíření.

[26] *ExTester* nedokázal najít požadovaný prvek v uživatelském rozhraní. Pravděpodobně ještě není zobrazený nebo má jiný selektor než test očekává.

9.5.2 commands/projectCreationQuarkusCommandAvailable.test.ts

Vygenrovaný test vykazuje tři stejné chyby jako v případě testu z kategorie *activations*. Jde tedy o snahu využívat neexistující funkce. Oprava kompilačních chyb kromě samotného řešení kód zjednodušuje. Z pohledu *flow* u daného testu nedochází k žádné významné změně a test si tak zachovává svoji logiku. Test je přeložitelný a spustitelný, jeho běh ovšem končí chybou typu `Test assertion failure` kdy test očekává přítomnost položky v `command palette`, ale položka se tam v danou chvíli nevyskytuje.

- **Věrnost**, hodnocení: 4. Kompilační chyby jsou opraveny, kód je lehce zjednodušen.
- **Podloženost**, hodnocení: 4. Vše vychází z dostupného kontextu a reflektuje potřeby testu.
- **Míra halucinací**, hodnocení: 0 % – Nevykazuje žádný zbytečný kódu.
- **Technická správnost výstupu**, hodnocení: 0. Vykazuje běhové chyby.
- **Čitelnost**, hodnocení: 3. Kód je dobře strukturovaný a je dobře čitelný.

9.5.3 menus/camelSubmenuOrderingIsCorrect.test.ts

Vygenerovaný test vykazuje dvě kompilační chyby typu `Property does not exist on type`. Opravu kompilačních chyb pojal jazykový model tak, že `await camelMenuItem!.getSubmenu()` nahradil `(camelMenuItem as any)`. K jiné změně v této logice testu nedošlo. Je tedy jasné, že tato oprava není validní, neb `camelMenuItem` bude stále reprezentovaný stejným typem, který funkcí `getSubmenu()` nedisponuje. Dále došlo ke změně typu vytvářeného souboru, který je potřebný pro některé kroky testu. Tuto změnu lze považovat za naprosto zbytečnou, ale v principu neškodlivou. Ačkoliv jde test nyní přeložit a spustit, vykazuje běhovou chybu typu `Unable to locate element`.

- **Věrnost**, hodnocení: 0. Tato forma opravy kompilačních chyb není opravou.
- **Podloženost**, hodnocení: 0. Oprava nerespektuje koncept.
- **Míra halucinací**, hodnocení: 0 %. Test po opravě nevykazuje žádné nové zbytečné funkce.
- **Technická správnost výstupu**, hodnocení: 0. Vykazuje běhové chyby.
- **Čitelnost**, hodnocení: 3. Kód je dobře strukturovaný a je dobře čitelný.

9.5.4 settings/camelCatalogVersionSettingPersists.test.ts

Vygenerovaný test obsahuje 8 kompilačních problémů. V 7 případech jde již o známou chybu v podobě volání neexistujících funkcí, v posledním případě jde o pokus o importování neexistující třídy, tedy o chybu typu `Module has no exported member`. Oprava kompilačních chyb odpovídá očekávanému přístupu, nedochází k žádnému generování zbytečných funkcí nebo zásahu do sémantiky testu. Opravený test je přeložitelný a spustitel-

ný, jeho běh však končí běhovou chybou typu `ElementNotInteractableError: element not interactable`. To znamená, že požadovaný element byl sice nalzen, ale nelze s ním provádět žádné interakce.

- **Věrnost**, hodnocení: 4. Oprava odpovídá očekávanému přístupu.
- **Podloženost**, hodnocení: 4. Způsob opravy odpovídá kontextu.
- **Míra halucinací**, hodnocení: 0 % Neobsahuje žádné zbytečné úpravy.
- **Technická správnost výstupu**, hodnocení: 0. Vykazuje běhové chyby.
- **Čitelnost**, hodnocení: 3. Kód je dobře strukturovaný a je dobře čitelný.

9.5.5 views/welcomeCreateProjectLinkInteraction.test.ts

Vygenerovaný test vykazuje dvě totožné chyby spočívající v nekorektním volání funkce. Funkce očekává vstupní argument, test je ovšem volá bez jakéhokoliv argumentu. Oprava kompilačních chyb spočívá ve změně přístupu k požadované komponentě. Tento přístup nelze považovat za nejvhodnější, pro opravu kompilačních chyb by pouze stačilo doplnit požadovaný argument. V tomto případě chybí argument specifikující typ očekávané notifikace. Test je přeložitelný a spustitelný. Vykazuje běhovou chybu typu `Waiting for element to be located`. To znamená, že test požadovaný element zatím nenašel a jen čeká, až se objeví v uživatelském rozhraní.

- **Věrnost**, hodnocení: 1. Očekávané řešení spočívá v doplnění argumentu funkce.
- **Podloženost**, hodnocení: 1. Ačkoliv nebyla poškozená sémantika testu, kontext k této opravě nebyl validně využitý.
- **Míra halucinací**, hodnocení: 0 % Žádný zbytečný kód nebyl vygenerován.
- **Technická správnost výstupu**, hodnocení: 0. Vykazuje běhové chyby.
- **Čitelnost**, hodnocení: 2. Kód je hůře čitelný, jeho členění je dostatečné.

9.5.6 Vyhodnocení

Oblast	Věrnost	Podloženost	Halucinace	Technická správnost	Čitelnost
Activation	4	4	0 %	0	4
Commands	4	4	0 %	0	3
Dialogs[27]	–	–	–	–	–
Menus	0	0	0 %	0	3
Settings	4	4	0 %	0	3
Views	1	1	0 %	0	2

Tabulka 2: Hodnocení generovaných výstupů po druhém kroku

V rámci výsledků můžeme pozorovat silnou korelaci mezi věrností a podložeností výsledků. V jednom případě test vykazoval silnou míru halucinovaného kódu. Tento zbytečný a přidaný kód měl zároveň negativní dopad na čitelnost kódu. Sémantika jednotlivých testů nebyla výrazně narušena. Navzdory tomu že žádný z testů po této fázi neproběhne bez běhové chyby, je nutno brát v potaz skutečnost, že hlavním cílem této fáze bylo kód zbavit kompilačních chyb a vytvořit spustitelný kód. O opravu běhových chyb se stará další fáze celé workflow.

S ohledem na to, že všechny testy byly na konci této fáze přeložitelné a bez kompilačních chyb, fáze je považována za **funkční**.

9.6 Oprava běhových chyb

Cílem tohoto kroku je úspěšně opravit všechny běhové chyby. Mezi běhovými chybami po předchozí fázi lze pozorovat skutečnost, že všechny chyby vyzařují obdobný charakter. Z pohledu zkušeného vývojáře lze říci, že odstraňování chyb tohoto charakteru není jednoduchá a přímočará záležitost. Příčin, které mohou způsobit podobnou běhovou chybu je nespočet a jen část z nich je přímo způsobena chybnou implementací. Mezi takové chyby může patřit například pokus o přístup k elementu v době, kdy ještě nebyl vykreslen nebo nese jiné než očekávané identifikátory. Obdobné chyby mohou být způsobeny i změnami mezi jednotlivými verzemi *ExTesteru* či *VS Code*. Dále je potřeba vzít v úvahu i fakt, že některé schopnosti *ExTesteru* jsou limitovány v závislosti na platformě. Obecně lze říci, že v případě spouštění testů na operačním systému *macOS* existuje větší množství limitací než v případě jiných operačních systémů. Odhalování tohoto typu chyb nezřídka kdy vyžaduje kromě analytického přístupu i nutnost řešení formou pokus-omyl.

9.6.1 activation/activatesOnJsonLanguageOpen.test.ts

Pro opravu chyby typu `Unable to locate element` se jazykový model rozhodl využít přístupu v podobě rozgenerování kódu a vytvoření nových funkcí. Výsledkem je tedy výrazně delší kód obsahující velkou míru kódové halucinace. Kód přináší nové funkce, které jsou zbytečné a vykazují velkou míru duplicity již s existujícími funkcemi frameworku *ExTester*. V daném kontextu nejde o vhodný přístup. Test po této fázi končí stále běhovou chybou v podobě chyby `ElementNotInteractableError: element not interactable`.

- **Věrnost**, hodnocení: 1. Nevhodný způsob přístupu k opravě vedoucí k nefunkčnímu řešení.
- **Podloženost**, hodnocení: 1. V daném kontextu není tento přístup vhodný.
- **Míra halucinací**, hodnocení: 80 %. Velké množství zbytečně vygenerovaného kódu.
- **Technická správnost výstupu**, hodnocení: 0. Test končí běhovou chybou.

[27] Testovací scénář neobsahuje kompilační chyby.

- **Čitelnost**, hodnocení: 4. Ačkoliv je nově vygenerovaný kód zbytečný, je přehledný a dobře organizovaný.

9.6.2 commands/projectCreationQuarkusCommandAvailable.test.ts

Oprava chyby typu `Test assertion failure` byla jazykovým modelem pojmuta zvláštně. Nově opravený kód oproti své předchozí části prošel značným zjednodušením. Sémantika testu tímto ovšem nebyla významně narušena a nelze tedy toto zjednodušení považovat nutně za chybné. Při sledování děje na obrazovce během testu lze pozorovat, že se odehrávají očekávané akce. Navzdory tomu test na konci této fáze stále končí běhovou chybou, tentokrát typu `ElementNotInteractableError: element not interactable`.

- **Věrnost**, hodnocení: 3. Spíše než o opravu samotné chyby šlo o přegenerování celého scénáře.
- **Podloženost**, hodnocení: 1. Značné zjednodušení znamená, že samotný kontext nebyl příliš využit.
- **Míra halucinací**, hodnocení: 0 %. Neobsahuje žádné zbytečné funkce.
- **Technická správnost výstupu**, hodnocení: 0. Test končí běhovou chybou.
- **Čitelnost**, hodnocení: 3. Kód je dobře čitelný a přehledný.

9.6.3 dialogs/newCamelFileSubmenuAccessibleFromFileMenu.test.ts

Oprava původní chyby, která vznikla na úrovni aserce očekávající platnou funkci `setDialogHandler`, byla jazykovým modelem pojata jako snaha obejít nevyhovující prostředí – místo skutečného dialogového chování byl test upraven tak, aby akceptoval i situaci, kdy je tato funkce nepřítomná. Zatímco tím došlo k výraznému zjednodušení a technickému zpřehlednění kódu, věrnost původnímu záměru testu tím částečně utrpěla. Přestože nový test již nepřichází do konfliktu s touto asercí, selhává jiným typem chyby, konkrétně `ElementNotInteractableError: element not interactable`.

- **Věrnost**, hodnocení 2: Test stále ověřuje dostupnost příkazu, ale již nepracuje s reálnými soubory a neověřuje samotnou logiku multi-file transformace.
- **Podloženost**, hodnocení 4: Oprava stojí na stejných pomocných funkcích a běžném API *ExTesteru* dostupném z kontextu.
- **Míra halucinací**, hodnocení 0 %: Neobsahuje žádné zbytečné funkce.
- **Technická správnost výstupu**, hodnocení: 0. Test končí běhovou chybou.
- **Čitelnost**, hodnocení: 4. Struktura testu je mírně kratší a dostatečně přehledná.

9.6.4 menus/camelSubmenuOrderingIsCorrect.test.ts

Původní test selhával na chybě `NoSuchElementError: Unable to locate element`. Jazykový model se pokusil tuto situaci vyřešit přístupem spočívajícím v obcházení problémového místa – přegeneroval scénář tak, aby pracoval pouze s hlavičkou sekce a kontextovým menu nad ní. Výsledkem je tedy značně přepracovaná struktura testu, která se oproti původní verzi výrazně zjednodušuje v hlavním těle, ale zároveň přináší řadu nových pomocných funkcí, jež často duplikují funkcionalitu již dostupnou v rámci *ExTesteru*. Navzdory těmto úpravám však test končí novou chybou `TypeError: section.openContextMenu is not a function`.

- **Věrnost**, hodnocení: 2. Test neřeší rozdíl mezi kontextovým menu složky a souboru, tím nepokrývá původní záměr testu.
- **Podloženost**, hodnocení: 3. Část chování je spíše odhad než přímo odvozené důsledky z původního testu.
- **Míra halucinací**, hodnocení: 30 %. Přibily pomocné funkce, které nejsou nutné a vykazují duplicitu.
- **Technická správnost výstupu**, hodnocení: 0. Test končí běhovou chybou.
- **Čitelnost**, hodnocení: 3. Hlavní test je kratší, logika je dobře rozčleněná.

9.6.5 settings/camelCatalogVersionSettingPersists.test.ts

Původní test selhával na chybě `ElementNotInteractactableError`, která vznikla při úpravě nastavení přes grafické rozhraní `SettingsEditor`. AI se rozhodla problém obejít tím, že UI část zcela odstranila a nahradila ji přímým přepisem souboru `settings.json`. Tím se však odklonila od původního účelu testu, který měl ověřovat skutečné chování rozšíření v prostředí *VS Code*.

Nově vzniklá verze je výrazně složitější, zavádí vlastní čekací mechanismy a silně spoléhá na otevření *User Settings* přes *Command Palette*, což vede k nové chybě: `Error: Failed to open User Settings (JSON) via command palette`. Oprava tak sice odstranila původní problém s neinteraktivním prvkem, ale nahradila jej jiným, který test zastaví ještě dříve.

- **Věrnost**, hodnocení: 1. Opravený test už vůbec nepracuje UI nastavením ale pracuje se `settings.json`, tím se vzdaluje původnímu záměru.
- **Podloženost**, hodnocení: 2. Využívá existující příkazy a klíče, ale ignoruje zavedený způsob práce a opírá se spíše o domněnky ohledně názvů a struktur.
- **Míra halucinací**, hodnocení: 40 %. Test zavádí vlastní, duplicitní logiku pro čekání. Vytváří navíc složitou metodu pro otevírání souboru s nastavením, i když už je pokrytá *ExTesterem*.
- **Technická správnost výstupu**, hodnocení: 0. Test končí běhovou chybou.

- **Čitelnost**, hodnocení: 3. Scénář je přehledný, funkce jsou dobře pojmenované. Celý test je ale zbytečně komplexní.

9.6.6 views/welcomeCreateProjectLinkInteraction.test.ts

Chybu typu `Waiting for element to be located` jazykový model opravuje dalším pokusem o kompletní přegenerování. Nově vygenerovaný kód vykazuje drobnou míru kódové halucinace v podobě funkce, kterou by pokryla již existující funkce *ExTesteru*. Na konci této fáze testový scénář nevykazuje žádnou běhovou chybu, aserce je vyhodnocena kladně. Testovací scénář je tím pádem považován za funkční.

- **Věrnost**, hodnocení: 2. Kód je prakticky celý přegenerovaný, nelze se na něj tedy dívat jako na opravu dané chyby.
- **Podloženost**, hodnocení: 1. Přegenerování celého scénáře znamená, že daný kontext pro opravu specifické chyby nebyl použit.
- **Míra halucinací**, hodnocení: 20 %. Kód vykazuje drobnou míru halucinace.
- **Technická správnost výstupu**, hodnocení: 1. Test probíhá bez běhové chyby.
- **Čitelnost**, hodnocení: 4. Kód je dobře čitelný a členěný.

9.6.7 Vyhodnocení

Oblast	Věrnost	Podloženost	Halucinace	Technická správnost	Čitelnost
Activation	1	1	80 %	0	4
Commands	3	1	0 %	0	3
Dialogs	2	4	0 %	0	4
Menus	2	3	30 %	0	3
Settings	1	2	40 %	0	3
Views	2	1	20 %	1	4

Tabulka 3: Hodnocení generovaných výstupů po třetím kroku

Ačkoliv výsledky třetí fáze zdánlivě vykazují spíš neúspěch, z pohledu vývojáře se na ně nelze dívat zcela jednoznačně. V případě jednoho testu máme na konci celé workflow spustitelný test, který vykazuje funkční aserce. Vyhodnocení sémantické smysluplnosti těchto testů se věnuje další sekce.

9.7 Manuální zásah vývojáře

Cílem tohoto kroku je manuální zásah do vytvořených testovacích scénářů. U testů, které na konci workflow neprochází bude provedena oprava zkušeným vývojářem. Cílem této opravy bude minimalistická snaha o opravení testovacího scénáře tak, aby proběhl co

nejmenší zásah do vygenerované sémantiky testu – tedy aby samotný scénář byl narušen co nejméně. Tímto přístupem bude nejméně ovlivněno výsledné vyhodnocení celkového přínosu testovacího scénáře v následující podsekci. Před samotným provedením manuální opravy bylo u každého testu individuálně vyhodnoceno, zda je vhodnější použít kód po fázi 2 – opravě kompilačních chyb, či po fázi 3 – opravě běhových chyb.

9.7.1 activation/activatesOnJsonLanguageOpen.test.ts

K opravě byl zvolen test po druhé fázi – opravě kompilačních chyb. Bylo nutné zásadně přepracovat způsob zjišťování aktivace rozšíření, protože původní přístup nefungoval. Také bylo potřeba opustit práci s textovým výpisem `Running Extensions`, který je nespolehlivý a nedeterministický. Namísto toho byl použit panel *Extensions*, který poskytuje stabilnější a uživatelsky viditelný zdroj informací o aktivovaném rozšíření. Bylo nutné přidat práci s `ActivityBar`, otevřít konkrétní `View Control` a vyhledat sekci `Installed`. Dále bylo třeba použít metadata z `package.json`, zejména `displayName`, aby se odstranily natvrdo zapsané řetězce a test byl robustnější vůči změnám ID. V sekci nainstalovaných rozšíření bylo nutné vyhledat konkrétní položku pomocí filtru `@installed`. Následně bylo potřeba pracovat přímo s DOM prvky rozšíření a vyhledat element `activationTime`, který *VS Code* zobrazuje pouze u aktivovaných pluginů. Aserce se změnila na jednoduché ověření přítomnosti prvku, což je jasnější indikátor aktivace než full-textové hledání.

9.7.2 commands/projectCreationQuarkusCommandAvailable.test.ts

K opravě byl zvolen test po třetí fázi – opravě běhových chyb. Kód byl výrazně zjednodušen odstraněním přímé práce s `WebDriverem` i čekáním přes `sleep`, takže bylo možné spolehnout se pouze na API *ExTesteru*. Byla odstraněna konstrukce s `try/finally` a kontrolou `undefined`, protože *QuickOpenBox* je nyní získáván přímo a bezpečně. Otevření `Command Palette` texttt zkráceno na jediné volání a odpadlo přetypovávání i zbytečná pomocná logika. Zůstala pouze podstata testu – zadání textu, získání `QuickPicks`, vyhodnocení přítomnosti příkazu a následné uzavření vstupního boxu. Díky tomu byl původní nefunkční koncept převeden do čistší, stabilnější a funkční podoby, která je snadněji udržovatelná.

9.7.3 dialogs/multiFileTransformXmlOpensFilePicker.test.ts

V tomto případě nebylo možné dosáhnout funkčního výsledku ani po důkladné revizi logiky testu po obou fázích. Problém nespočívá v chybné implementaci či nesprávném použití API *ExTesteru*, ale v samotné povaze interakce s dialogy v *VS Code*. Test se snaží ověřit otevření nativního dialogu pro vícenásobný výběr souborů, nicméně tento prvek není skrze *WebDriver* ani prostředky *ExTesteru* interagovatelný. Využití interního jednoduchého dialogu *VS Code* rovněž není možné, protože tento dialog nepodporuje výběr více položek, což činí požadovaný scénář technicky neproveditelným. Jelikož je tato limitace známá a dlouhodobě neřešená, byl test vyhodnocen jako neopravitelný z důvodů ležících mimo rámec zásuvného modulu. Situace je navíc zdokumentována ve veřejně dostupném issue[30], které potvrzuje, že nativní dialogy nelze v současném stavu automatizovat ani

obejít. Výsledkem je, že test nelze převést do stabilní či spustitelné podoby, a jeho správná funkce závisí na budoucích změnách samotného *VS Code*. Z pohledu metodiky jde o scénář, který překračuje možnosti frameworku, a nejvhodnějším řešením je test dočasně vyřadit.

9.7.4 menus/camelSubmenuOrderingIsCorrect.test.ts

K opravě byl zvolen test po 2. fázi – opravě kompilačních chyb. Bylo nutné zásadně zjednodušit jeho strukturu. Původní verze spoléhala na dynamické vytváření souboru, komplikované procházení stromu `Exploreru`, práci s `DefaultTreeItem` a opakované dotazování přes `WebDriver`. Tento postup byl rozsáhlý, nestabilní a silně závislý na konkrétní podobě pracovního prostoru. Opravená verze využívá předpřipravené testovací zdroje otevřené přes `openResources`, což poskytuje konzistentní a plně kontrolované prostředí. Díky tomu mohly být odstraněny nadbytečné kroky související s vytvářením souborů i složité prohledávání stromu — konkrétní testovací položky jsou dostupné přímo a spolehlivě. Zároveň byl použit modernější přístup založený na `ViewItem`, který umožňuje jednodušší a robustnější práci s kontextovým menu bez nutnosti externích čekání nebo opakovaného dotazování. Aserce zůstaly beze změny a test nadále přímo porovnává skutečné pořadí položek v submenu *New Camel File* s očekávaným výsledkem. Úpravy tak směřovaly především ke zjednodušení přípravných kroků a odstranění technických obtíží, aniž by byl ovlivněn způsob, jakým test vyhodnocuje správnost nabídky. Výsledkem je kratší, stabilnější a lépe udržovatelný test, který vychází z původní struktury, ale zbavuje se jejích problematických částí.

9.7.5 settings/camelCatalogVersionSettingPersists.test.ts

K opravě byl zvolen test po druhé fázi – opravě kompilačních chyb. V testu bylo vyhledávání nastavení upraveno tak, že se `findSetting` používá s kombinací `"Camel catalog version" + "Camel"`, aby bylo nastavení spolehlivě nalezeno i bez plného názvu rozšíření. Bylo změněno volání příkazu pro `JSON` nastavení na `Preferences: Open User Settings (JSON)`, takže je nyní čtena uživatelská konfigurace `settings.json` ve správném scope. Logika práce se `settings.json` byla ponechána, ale byla doplněna o pevně daný příkaz a následné otevření editoru přes `EditorView`, čímž byl zpřesněn scénář, který test ověřuje. Celý `after` blok byl přepracován tak, aby se nejprve vždy zavřely všechny editory pomocí `closeAllEditors()`, a test tak nezačínal obnovu nastavení z nekonzistentního stavu UI. Obnovení původní hodnoty nastavení bylo změněno tak, že je nastavení znovu explicitně otevřeno přes příkaz `'Preferences: Open Settings (UI)'` a následné `openSettings()`, což zaručuje, že se původní hodnota zapisuje do stejného místa, ze kterého byla čtena. Podmínka pro obnovu hodnoty `(originalValue !== undefined && newValue !== undefined && originalValue !== newValue)` byla zachována, ale byla přesunuta do jednoduššího toku bez `try/finally`, čímž byl úklid testu zpřehledněn. Tímto přeuspořádáním kroků před a po testu byla odstraněna závislost na implicitním stavu edi-

torů a nastavení a test byl stabilizován tak, aby konzistentně ověřoval jak UI nastavení, tak zápis v `settings.json`.

9.7.6 views/welcomeCreateProjectLinkInteraction.test.ts

Test nebylo nutné opravovat, protože po třetí fázi workflow procházel bez jakýchkoliv kompilačních či běhových chyb.

9.7.7 Vyhodnocení

Z celkového počtu šesti ověřovaných testů jich pět potřebovalo opravu. Všechny testy bylo možné úspěšně opravit do stavu, kdy jsou spustitelné a vykazují známky funkčních asercí. Každý vyžadoval odlišný typ zásahu a nelze proto zobecnit jednotný postup ani dobu potřebnou k opravě. Některé opravy byly poměrně jednoduché a spočívaly pouze v úpravě několik řádků kódu či zpřesnění práce s API *ExTesteru*, zatímco jiné vyžadovaly zásadní přepracování celého testovacího přístupu – například změnu strategie vyhledávání prvků, opuštění nespolehlivých metod či nahrazení celých bloků kódu robustnější variantou. U všech testů bylo nutné rozhodnout, zda je vhodnější vycházet z výsledku po opravě kompilačních chyb (fáze 2), nebo až po opravě běhových chyb (fáze 3), protože různé generované verze poskytovaly odlišnou míru použitelného základu pro manuální zásah. Míra chybovosti se rovněž lišila – některé návrhy vykazovaly jen drobné nepřesnosti, zatímco jiné obsahovaly zásadní omyly v logice testu či v práci s uživatelským rozhraní, které bránily jakémukoli úspěšnému provedení. Přesto bylo možné ve většině případů test stabilizovat při zachování původní sémantiky tak, aby byl zachován smysl generovaného scénáře a nedošlo k jeho neúměrnému přepsání. Celkově lze tedy konstatovat, že manuální oprava byla realizovatelná, avšak její náročnost byla proměnlivá a jednoznačný, univerzální postup neexistoval.

9.8 Diskuze

9.8.1 activation/activatesOnJsonLanguageOpen.test.ts

Testový scénář dokáže technicky ověřit, zda je rozšíření aktivováno, avšak jeho sémantika je zásadně chybná. Test předpokládá, že otevření *JSON* souboru je tím okamžikem, který zapříčiní aktivaci rozšíření, a že tato aktivace bude ještě v daný moment detekovatelná přes *activationTime* v seznamu nainstalovaných rozšíření. Ve skutečnosti však tento scénář nikdy nenastane, protože rozšíření *Camel LSP Client* se aktivuje již při spuštění *VS Code* na základě aktivační události *onStartupFinished*, tedy podstatně dříve, než dojde k interakci s jakýmkoli *JSON* souborem. Test tak sice formálně ověřuje přítomnost prvku *activationTime*, ale neověřuje to, co deklaruje – nedokazuje, že otevření *JSON* souboru aktivaci vyvolalo, nýbrž pouze potvrzuje, že *extension* je již aktivováno z jiného důvodu. Sémanticky jde tedy o **neplatný** scénář, který nemá oporu v reálném chování rozšíření a nesprávně modeluje jeho aktivační podmínky. Přítomnost *activationTime* je tudíž výsledkem startovací logiky *VS Code*, nikoliv důsledkem akce simulované testem. Test lze

tedy považovat za technicky funkční, stabilní a formálně proveditelný, avšak z hlediska věcného obsahu nedává smysl a neodpovídá skutečné aktivační strategii rozšíření. Finálně zhodnoceno, tento test neověřuje relevantní ani realistickou situaci a jeho používání by vedlo k falešnému pocitu pokrytí aktivační logiky.

9.8.2 commands/projectCreationQuarkusCommandAvailable.test.ts

Tento test je technicky funkční, stabilní a přesně ověřuje to, co deklaruje – tedy přítomnost příkazu *Create a Camel Quarkus project* v *Command Palette*. Z formálního hlediska jde o čistý, stručný a sémanticky konzistentní scénář, který správně pracuje s API *ExTesteru* a neobsahuje zbytečnou logiku ani problematické konstrukce. Test přímo vyvolá *Command Palette*, vyfiltruje položky podle zadaného textu a pomocí jednoduché aserce ověří, že mezi výsledky existuje odpovídající příkaz, což odpovídá běžným očekáváním uživatele i typickému workflow v *VS Code*. Zároveň je tento test věcně smysluplný – přítomnost příkazu v paletě skutečně představuje realistickou a důležitou funkční vlastnost rozšíření, kterou lze hodnotit i z hlediska uživatelské použitelnosti. Scénář tedy není pouze technicky správný, ale i významový: validuje existenci základního vstupního bodu rozšíření, bez něhož by uživatel nemohl zahájit pracovní postup. Na rozdíl od jiných testů, kde generativní model vytvořil nepravdivou premisu nebo nereálný kontext, zde sémantika odpovídá reálnému chování aplikace a test skutečně ověřuje relevantní výsek funkcionality. Lze tedy uzavřít, že test nejen prochází, ale také dává smysl a je vhodný jako součást celkové sady UI testů.

9.8.3 dialogs/multiFileTransformXmlOpensFilePicker.test.ts

Tento testový scénář představuje technicky komplexní pokus ověřit, zda příkaz *„Transform Camel Routes in multiple files to XML DSL“* nejen existuje v *Command Palette*, ale také po spuštění otevře multi-file výběrový dialog. Formálně je test vystavěn korektně: pracuje s *Workbench* API, vyhledá cílový příkaz, spustí jej a následně se pokouší zachytit otevření dialogu prostřednictvím dostupných hooků (*setDialogHandler*). Logika detekce je navržena robustně – test pracuje s více možnými strukturami dialogových objektů, ověřuje podporu *multi-select* voleb a snaží se bezpečně potvrdit nebo odmítnout dialog. Po stránce implementace tedy nejde o chybný či nesémantický návrh. Test skutečně provádí to, co deklaruje, a správně reflektuje zamýšlený uživatelský scénář: spuštění příkazu má vést k dialogu umožňujícímu výběr více souborů. Zároveň byla provedena analýza běhových chyb a všechny opravitelné problémy byly odstraněny – scénář byl tedy posuzován až v jeho finální podobě po pokusu o stabilizaci. Zásadní problém však spočívá mimo samotnou implementaci: test je z principu neproveditelný. *VS Code* používá pro výběr souborů nativní systémové dialogy operačního systému, které nejsou přístupné přes *WebDriver* ani žádný z podpůrných mechanismů *ExTesteru*. Tato omezení jsou dlouhodobě známá a zdokumentovaná, a neexistuje způsob, jak pomocí standardních API takový dialog spolehlivě zachytit nebo s ním interagovat. Alternativní cesta – využití interního *VS Code* dialogu – rovněž není proveditelná, protože tento typ dialogu nepodporuje multi-file výběr, což je

podstatou testovaného scénáře. I přes důkladnou revizi dvou generací generovaného kódu tedy nebylo možné dosáhnout funkčního výsledku. Test není nestabilní či chybně napsaný – je neopravitelný, protože se snaží validovat chování, které leží mimo možnosti dostupné automatizační vrstvy. Zachycení nativního *file pickeru* není v současné architektuře *VS Code* podporováno, a jak uvádí i veřejně dostupná issue dokumentace, neexistují žádné známé *work-aroundy*, které by umožnily tento scénář spolehlivě automatizovat. Z metodického hlediska tak nejde o test s chybnou sémantikou, ale o test simulující situaci, kterou není možné automatizovaně ověřit. Jeho zachování by vedlo k trvale selhávajícím nebo zavádějícím výsledkům a neposkytovalo by žádnou reálnou záruku kvality. Nejvhodnějším řešením je proto test dočasně vyřadit a v budoucnu znovu zvážit jeho implementaci až ve chvíli, kdy bude *VS Code* podporovat interakci s nativními dialogy, případně nabídne jejich alternativu dostupnou testovacím frameworkům. Výsledkem hodnocení tedy je, že tento test nelze uvést do funkční podoby – nikoli kvůli jeho konstrukci, ale proto, že jeho cíl přesahuje technické možnosti nástroje.

9.8.4 menus/camelSubmenuOrderingIsCorrect.test.ts

Tento test je technicky čistý, stabilní a přesně provádí to, co deklaruje – ověřuje strukturu podnabídky *New Camel File* v kontextu souboru i složky. Postupně prováděné kroky (nalezení položky v *Exploreru*, otevření kontextového menu, vyvolání submenu a porovnání položek) odpovídají běžnému *use-case* a využívají standardní API *ExTesteru* bez problematických konstrukcí. Test je tedy z implementačního hlediska bez výrazných nedostatků. Z hlediska sémantické hodnoty je však jeho přínos omezenější. Ověřované pořadí položek sice odráží současný návrh UI, nicméně představuje spíše detail uživatelského rozhraní než zásadní vlastnost pluginu. Změna pořadí položek nemusí znamenat funkční chybu ani narušení doménové logiky; jde spíše o preferenční a vizuální aspekt, který může být v budoucnu změněn bez dopadu na samotnou funkcionalitu. Test tedy nevaliduje „správnost“ chování rozšíření v užším slova smyslu, ale spíše striktně hlídá konzistenci konkrétní konfigurace menu. Současně je vhodné zdůraznit, že tento typ *„golden ordering“* scénáře je ze své podstaty křehký. Jakákoli drobná úprava – změna textu, přidání ikony, zavedení lokalizace, nebo přidání nové položky do nabídky – povede k selhání testu, aniž by šlo o reálnou chybu z pohledu uživatele nebo systému. Test tak velmi těsně váže implementaci na současnou podobu UI, která není nutně konečná a může se vyvíjet. Na druhou stranu právě tato striktnost může být záměrná: pokud je cílem udržovat stabilní, dokumentované pořadí položek, poskytuje test silnou regresní kontrolu proti nechtěným změnám nebo chybným úpravám group indexů. Pozitivní je také to, že test správně rozlišuje kontext nad souborem a nad složkou. V první větvi ověřuje nabídku směřující k tvorbě jednotlivých entit či transformaci jedné route, zatímco v kontextu složky test kontroluje přítomnost hromadných transformačních příkazů, které by nad souborem dávaly jen malý smysl. Test tak neověřuje pouze pořadí položek, ale i jejich kontextovou relevanci, což přidává scénáři určitou míru sémantické hodnoty. Celkově lze test považovat za technicky správný a formálně konzistentní, avšak jeho skutečná věcná hodnota je spíše okrajová – neověřu-

je funkční logiku rozšíření, ale především detailní stav UI. Jde tedy o užitečný, ale méně zásadní test, který spíše chrání stabilitu prezentace než stabilitu samotné funkcionality.

9.8.5 settings/camelCatalogVersionSettingPersists.test.ts

Tento test je technicky funkční a zároveň sémanticky poměrně dobře navržený – ověřuje realistický scénář, kdy je uživatelské nastavení změněno v UI a následně zkontrolováno jak v grafickém prostředí, tak v `settings.json`. Scénář odpovídá reálnému workflow uživatele: hodnota „*Camel catalog version*“ je v nastavení vyhledána, změněna na jinou platnou hodnotu a poté je ověřeno, že se změna skutečně propsala a po opětovném otevření nastavení přetrvá. Druhá část testu navíc ověřuje konzistenci mezi UI a textovou konfigurací – kontrolou hodnoty v souboru `settings.json` je potvrzeno, že extension pracuje s uživatelským nastavením tak, jak *VS Code* očekává. Z pohledu sémantiky je však třeba zdůraznit, že tento scénář ověřuje primárně vlastnost samotného *VS Code* (schopnost udržet a propsat hodnotu nastavení), nikoliv chování testovaného zásuvného modulu. Vhodnější testovací scénář by ověřoval, zda změna verze katalogu skutečně ovlivňuje chování pluginu – například dostupnost konkrétních komponent v *Camel katalogu*[28]. Ideální by bylo najít komponentu, která je přítomna v jedné verzi katalogu a v jiné již nikoliv, otevřít relevantní soubor, zjistit dostupné komponenty, přepnout verzi katalogu a znovu ověřit, zda se nabídka komponent změnila v souladu s očekáváním. Takový test by jednoznačně prokazoval, že nastavení není jen uloženo v konfiguraci, ale že je i aktivně využíváno pluginem a promítá se do jeho funkcionality. Současný test je tedy užitečný jako dílčí kontrola perzistence nastavení, ale sémanticky se pohybuje spíše na úrovni editoru než na úrovni doménové logiky rozšíření.

9.8.6 views/welcomeCreateProjectLinkInteraction.test.ts

Tento test jako jediný dospěl na konec celé workflow bez nutnosti manuálního zásahu vývojáře. Test je technicky proveditelný a při běhu prochází, ale z hlediska sémantiky jde o téměř úplný nesmysl. Deklarovaným cílem je ověřit, že kliknutí na odkaz *„Create a Camel project“* na úvodní obrazovce je v případě zakázaného příkazu korektně ošetřeno, ale test ve skutečnosti nic takového neověřuje. Pokud se odkaz vůbec nenajde, test se spokojí s tím, že je stále dostupný *window handle* a rovnou skončí – tedy situace, kdy se scénář vůbec neodehrál, je považována za úspěšný průběh. I v případě, že se odkaz najde a kliknutí selže, je závěrečná aserce `expect(clicked || true).to.be.true` tautologií, která projde vždy, a nezávisí ani na tom, zda se skutečně kliklo, ani na tom, jak UI zareagovalo. Test tak v praxi neověřuje ani stav „neselhalo to po kliknutí“, protože stejný výsledek by byl dosažen i bez jakéhokoli pokusu o kliknutí nebo bez existence odkazu. Kontrola `getWindowHandle()` představuje jen velmi slabou kontrolu, která je navíc od pluginu prakticky odtržená – spíše ověřuje, že celý *VS Code* nespadl, což je extrémně nízká a nereálná laťka. Poměrně složitá pomocná logika *ensureEmptyExplorer*, která

[28] Camel katalog je strukturovaný seznam všech dostupných komponent, konektorů a jejich verzí, který používaný plugin využívá k poskytování inteligentních návrhů a validace v editoru.

zkouší různé varianty *„Close Folder/Workspace“*, vnáší do testu další šum, ale nepřispívá k ověření chování samotného příkazu `camel.jbang.project.new`. Výsledkem je scénář, který je sice komplikovaný na údržbu, ale aplikačně nic zásadního nevaliduje a nepřináší smysluplnou jistotu ohledně robustnosti nebo použitelnosti rozšíření. Sémanticky tak test spíše potvrzuje, že „něco se možná stalo a *VS Code* ještě žije“, než že by skutečně ověřoval korektní ošetření zakázaného nebo nefunkčního příkazu v rámci pluginu. Lze ho tedy hodnotit jako technicky fungující, ale významově prázdný – poskytuje falešný pocit pokrytí scénáře s *disabled* příkazem, aniž by tento scénář reálně a měřitelně ověřoval.

9.8.7 Vyhodnocení

Vyhodnocení se vrací k cílům metodiky a závěrům rešerše, zejména kapitol 5 a 7, a hodnotí dosažené výsledky navrženého řešení. Volba generativního jazykového modelu se v praxi ukázala jako přínosná především v počáteční fázi testování, kdy umožnila velmi rychlé vytvoření základních testovacích scénářů a kostry UI testů. Tento přínos odpovídá závěrům uvedeným v literatuře, která zdůrazňuje schopnost AI výrazně urychlit tvorbu testů a snížit manuální zátěž vývojářů. Automatizovaný návrh testů tak skutečně vedl ke zrychlení počáteční fáze testovacího procesu a usnadnil pokrytí klíčových funkcionalit rozšíření.

Současně se však potvrdila i rizika identifikovaná v rešeršní části práce. Na základě provedené sémantické analýzy lze říci, že generované testy vykazují velmi různorodou úroveň věcné správnosti, což potvrzuje, že technická průchodnost testu nemusí automaticky znamenat jeho skutečnou užitečnost. U části testů se ukázalo, že generativní model dokáže vytvořit sémanticky smysluplné scénáře, které odpovídají reálnému chování rozšíření a mají praktickou hodnotu pro jeho ověřování. Jiné testy však byly založeny na chybných předpokladech o chování systému, čímž sice technicky procházely, ale neověřovaly deklarovanou funkcionalitu.

Tento jev vede k falešnému pocitu pokrytí, který je v praxi nebezpečnější než test selhávající, protože vytváří iluzi, že určitá oblast aplikace je otestována, i když tomu tak není. U části scénářů se potvrdilo, že generativní model umí zachytit správnou strukturu a workflow testu, ale nerozumí plně doménové logice či očekávaným aktivačním podmínkám pluginu. V některých případech testy ověřovaly vlastnosti samotného editoru místo chování rozšíření, čímž se míjely účelem.

Analýza rovněž ukázala, že sémantická kvalita testů není konzistentní. Test, který původně selhával, mohl po cílené úpravě nabídnout hodnotné ověření funkce, zatímco test, který bez problémů prošel celým workflow, mohl být z hlediska ověřované funkcionality zcela irelevantní. Tyto výsledky potvrzují závěry uvedené v odborné literatuře, podle nichž generativní modely vyžadují lidský dohled a jejich výstupy nelze bez další kontroly považovat za plnohodnotné testy.

Z hlediska metodiky lze konstatovat, že funkční požadavky byly naplněny – rozšíření je schopno automaticky generovat UI testy nad frameworkem *ExTester* a integrovat je přímo do prostředí Visual Studio Code. Nefunkční požadavky týkající se použitelnosti, přenositelnosti a rozšiřitelnosti byly rovněž splněny, neboť nástroj nabízí intuitivní ovládání a konzistentní chování napříč operačními systémy. Přetrvávající výzvou zůstává oblast

integrace UI testů do CI/CD pipeline, která není v současné praxi standardizována a nebyla cílem této práce.

Celkově lze shrnout, že generativní přístup představuje významnou podporu při tvorbě UI testů, zejména ve fázi návrhu a rychlého prototypování. Zároveň se však jednoznačně ukazuje, že generativní AI nemůže nahradit lidský úsudek při posuzování smysluplnosti a relevance testovacích scénářů. Manuální zásah vývojáře zůstává nezbytný nejen pro opravu chyb, ale především pro ověření, zda test skutečně testuje to, co deklaruje. Výsledný proces tak potvrzuje nutnost kombinace AI asistence a lidské expertízy, což odpovídá závěrům současného výzkumu v oblasti AI-asistovaného testování.

10 Závěr

Tato práce se zabývala využitím generativní umělé inteligence pro automatizaci tvorby testů uživatelského rozhraní pro rozšíření do *Visual Studio Code*. Cílem bylo navrhnout a implementovat prototyp zásuvného modulu, který kombinuje framework *ExTester* s generativními modely a dokáže automaticky generovat testy, spouštět je, analyzovat jejich výsledky a případně opravovat zjištěné chyby. Prototyp byl úspěšně realizován a výsledný nástroj demonstruje základní koncepty automatického generování testů uživatelského rozhraní.

Navržené řešení přináší potenciál pro několik přínosů. Nástroj automatizuje rutinní tvorbu testů a tím zrychluje testovací cyklus vývoje rozšíření. Dokáže navrhovat neobvyklé scénáře a hraniční vstupy, na které by vývojář nemusel pomyslet. Pomáhá odhalovat neočekávané chyby a zvyšuje pokrytí kódů. Celé řešení je integrováno do *Visual Studia Code*, díky tomu je možné nástroj snadno zapojit do workflow vývojáře.

Prototyp má zároveň své limity a neobejde se bez lidského dohledu. Ne vždy model správně rozumí záměru – generovaný test lze sice úspěšně zkompilovat, ale následně může selhat za běhu. Implementace odhalila výrazný rozdíl mezi opravou kompilačních a běhových chyb. Syntaktické a typové chyby dokázal generativní jazykový model často opravit automaticky bez výraznější tendence narušit smysluplnost testovacího scénáře. Řešení běhových chyb je ovšem složitější a nástroj si s nimi neumí adekvátně poradit – opakované pokusy o jejich řešení vedly k narušení sémantiky testů a k chybné rekonfiguraci celého scénáře. Nástroj je rovněž limitován kontextovým oknem modelu a charakterem vstupních dat. Z těchto důvodů je nezbytné, aby generované testy vždy prošly kontrolou vývojáře.

Z provedeného vyhodnocení také vyplynulo, že při manuálním zásahu se bylo možné dostat k poměrně uspokojivým výsledkům. Cílem testování bylo co nejméně zasahovat do sémantiky testů, nicméně pokud by se vývojář zaměřil i na její systematickou opravu, bylo by možné dosáhnout velmi kvalitních a dobře strukturovaných testovacích scénářů. Nelze však jednoznačně říci, zda je tento postup v praxi skutečnou úsporou času – může být přínosný, ale zároveň může vývojář strávit opravováním chyb více času, než kdyby test napsal ručně od začátku. Určitým paradoxem je, že jediný test, který prošel plným workflow bez nutnosti zásahu programátora, sice formálně procházel, ale netestoval deklarovanou funkcionalitu – takový výsledek dokládá, že lidská kontrola není jen doplňkem, ale nutností.

I přes uvedená omezení prokázal vytvořený plugin značný potenciál. Moderní nástroje umělé inteligence dokážou doplnit stávající testovací nástroje a urychlit vybrané fáze vývoje. Pro spolehlivé výsledky je nutné kombinovat asistenci umělé inteligence s lidským dohledem.

Zásuvný modul je k dispozici jako open-source projekt na *GitHubu*[29], což umožní jeho další využití a vývoj. Zároveň tento přístup umožní případný komunitní vývoj. Na tento prototyp lze navázat a rozšířit jej o další funkce.

Závěrem lze konstatovat, že cíle diplomové práce byly splněny. Nástroj prokázal smysluplnost i limity navrženého přístupu. Umělá inteligence dokáže urychlit vývojářskou práci, avšak není všemocná a slouží jen jako asistent vyžadující odborný dohled zkušené-

[29] `https://github.com/pospisilf/extester-code-generator`

ho vývojáře. Výsledný plugin rozšiřuje možnosti frameworku *ExTester* a naznačuje směr budoucího vývoje nástrojů pro testování softwaru.

11 Přehled literatury

Odkazy

[1] Amazon Web Services. *Amazon CodeWhisperer*. Official page for the Amazon CodeWhisperer AI coding assistant. [cit. 2025-11-19]. URL: https://aws.amazon.com/codewhisperer.

[2] Anthropic. *Claude – AI Assistant by Anthropic*. Official site for Claude, next-generation AI assistant by Anthropic. [cit. 2025-11-19]. URL: https://claude.ai/.

[3] A. Bassey a J. Reifer. *Testing VSCode Extensions with TypeScript*. Microsoft Dev Blogs [online]. 2024-04-12 [cit. 2025-03-04]. URL: https://devblogs.microsoft.com/ise/testing-vscode-extensions-with-typescript.

[4] Shreya Bhatia et al. "Unit Test Generation using Generative AI : A Comparative Performance Analysis of Autogeneration Tools". In: *Proceedings of the 1st International Workshop on Large Language Models for Code*. LLM4Code '24. Lisbon, Portugal: Association for Computing Machinery, 2024, s. 54–61. ISBN: 9798400705793. DOI: 10.1145/3643795.3648396.

[5] Christian BROMANN. *Stateful. A Complete Guide to VS Code Extension Testing*. A Complete Guide to VS Code Extension Testing [online]. 2022-07-26 [cit. 2025--03-04]. URL: https://stateful.com/blog/a-complete-guide-to-vs-code-extension-testing.

[6] Katharina Buchholz. *The Extreme Cost Of Training AI Models*. Forbes [online]. 2024-08-23 [cit. 2025-04-15]. URL: https://www.forbes.com/sites/katharinabuchholz/2024/08/23/the-extreme-cost-of-training-ai-models/.

[7] camel-tooling. *camel-tooling/camel-lsp-client-vscode*. GitHub repository – VS Code extension for Apache Camel language support (client for Camel Language Server). [cit. 2025-12-10]. URL: https://github.com/camel-tooling/camel-lsp-client-vscode.

[8] coder. *coder/code-server*. GitHub repository – VS Code in the browser (runs Visual Studio Code as a remote web app). [cit. 2025-10-16]. URL: https://github.com/coder/code-server.

[9] CodiumAI. *CodiumAI – AI-assisted test generation*. Official site of CodiumAI platform for generating meaningful tests. [cit. 2025-11-19]. URL: https://www.codium.ai/.

[10] Cypress. *Cypress – Frontend Testing Framework*. Cypress end-to-end testing tool for web applications. [cit. 2025-10-16]. URL: https://www.cypress.io/.

[11] Dan Čermák. *Testing Visual Studio Code extensions*. Talk at FOSDEM 2021 — Testing and Automation devroom. [online]. 2021-02-06 [cit. 2025-03-04]. URL: https://archive.fosdem.org/2021/schedule/event/testing_vscode_extensions/.

[12] Docker. *Docker – Empowering App Development for Developers*. Official site of Docker container platform. [cit. 2025-10-16]. URL: https://www.docker.com.

[13] Microsoft Docs. *Visual Studio Code Extension API – Testing Extensions*. Microsoft Docs [online]. [cit. 2025-03-04]. URL: https://code.visualstudio.com/api/working-with-extensions/testing-extension.

[14] Electron. *Electron*. Framework for building desktop applications with web technologies. [cit. 2025-12-10]. URL: https://www.electronjs.org.

[15] EvoSuite. *EvoSuite – Automatic test generation for Java*. Official site of the EvoSuite automatic test generation tool. [cit. 2025-12-10]. URL: https://www.evosuite.org.

[16] David Gewirtz. *X's Grok did surprisingly well in my AI coding tests*. ZDNet [online]. 2025-01-06 [cit. 2025-04-15]. URL: https://www.zdnet.com/article/xs-grok-did-surprisingly-well-in-my-ai-coding-tests/.

[17] GitHub. *GitHub Copilot Features*. Official GitHub page outlining Copilot AI coding assistant features. [cit. 2025-12-10]. URL: https://github.com/features/copilot.

[18] Diana Cheung. *Meta Llama 2 vs. OpenAI GPT-4: A Comparative Analysis of an Open-Source vs. Proprietary LLM*. Codesmith Blog [online]. 2023-11-09 [cit. 2025-05-11]. URL: https://www.codesmith.io/blog/meta-llama-2-vs-openai-gpt-4-a-comparative-analysis-of-an-open-source-vs-proprietary-llm.

[19] Anita Kirkovska. *Comparison Analysis: Claude 3.5 Sonnet vs GPT-4o*. Vellum AI Blog [online]. 2024-06-25 [cit. 2025-04-15]. URL: https://www.vellum.ai/blog/claude-3-5-sonnet-vs-gpt4o.

[20] Ted Kurmaku, Eduard Paul Enoiu a Musa Kumrija. "Human-based Test Design versus Automated Test Generation: A Literature Review and Meta-Analysis". In: *Proceedings of the 15th Innovations in Software Engineering Conference*. ISEC '22. Gandhinagar, India: Association for Computing Machinery, 2022. ISBN: 9781450396189. DOI: 10.1145/3511430.3511433.

[21] Maurizio Leotta et al. "AI-Generated Test Scripts for Web E2E Testing with ChatGPT and Copilot: A Preliminary Study". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, s. 339–344. ISBN: 9798400717017. DOI: 10.1145/3661167.3661192.

[22] Kefan Li a Yuan Yuan. *Large Language Models as Test Case Generators: Performance Evaluation and Enhancement*. 2024. arXiv: 2404.13340 [cs.SE]. URL: https://arxiv.org/abs/2404.13340.

[23] Meta AI. *Llama (family of large language models) – Meta AI*. Official site for Meta Llama large language models, including LLaMA2. [cit. 2025-11-19]. URL: https://www.llama.com/.

[24] Meta AI. *Meta Code Llama – AI Tool for Coding*. Official page for Code Llama, a code-specialized large language model. [cit. 2025-11-19]. URL: https://www.llama.com/code-llama/.

[25] Microsoft. *Continuous Integration – Visual Studio Code Extension API*. Official documentation on setting up Continuous Integration for VS Code extensions. [cit. 2025-12-10]. URL: https://code.visualstudio.com/api/working-with-extensions/continuous-integration.

[26] Microsoft. *Testing – Visual Studio Code Documentation*. Official documentation on testing support in Visual Studio Code. [cit. 2025-12-10]. URL: https://code.visualstudio.com/docs/editor/testing.

[27] Microsoft. *Visual Studio Marketplace*. Official marketplace for extensions for Visual Studio, VS Code, Azure DevOps and related tools. [cit. 2025-12-10]. URL: https://marketplace.visualstudio.com/.

[28] Microsoft. *Write tests with AI – Visual Studio Code Documentation*. Official documentation on writing tests with AI (e.g., using GitHub Copilot) in Visual Studio Code. [cit. 2025-12-10]. URL: https://code.visualstudio.com/docs/editor/testing#_write-tests-with-ai.

[29] Microsoft / TypeScript contributors. *Using the Compiler API – TypeScript Wiki*. Documentation on how to use the TypeScript Compiler API from the official TypeScript Wiki. [cit. 2025-12-10]. URL: https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API.

[30] Microsoft / VS Code contributor. *Simple file dialog: support multi-select · Issue #186637 · microsoft/vscode*. GitHub issue requesting support for multi-select in the simple file dialog in VS Code. [cit. 2025-12-10]. URL: https://github.com/microsoft/vscode/issues/186637.

[31] Mistral AI. *Mistral AI*. Official site of Mistral AI – frontier AI models and platform. [cit. 2025-11-19]. URL: https://mistral.ai/.

[32] MochaJS. *Mocha – simple, flexible, fun JavaScript test framework*. [cit. 2025-09-23]. URL: https://mochajs.org.

[33] Felicity Nelson. *Many Companies Are Banning ChatGPT. This Is Why*. ScienceAlert [online]. 2024-06-16 [cit. 2025-04-15]. URL: https://www.sciencealert.com/many-companies-are-banning-chatgpt-this-is-why.

[34] OpenAI. *ChatGPT*. Official site for the ChatGPT AI language model. [cit. 2025--12-10]. URL: https://chatgpt.com.

[35] Playwright. *Playwright – Testing for Modern Web Apps*. End-to-end testing framework supporting multiple languages. [cit. 2025-10-16]. URL: https://playwright.dev.

[36] Prompt Engineering Guide. *Mistral 7B LLM*. Prompt Engineering Guide [online]. [cit. 2025-05-11]. URL: https://www.promptingguide.ai/models/mistral-7b.

[37] Randoop Project. *Randoop – Automatic unit test generation for Java*. Official site of the Randoop unit test generator for Java. [cit. 2025-11-19]. URL: https://randoop.github.io/randoop/.

[38] Keshav Rangan a Yiqiao Yin. *A Fine-tuning Enhanced RAG System with Quantized Influence Measure as AI Judge*. 2024. arXiv: 2402.17081 [cs.IR]. URL: https://arxiv.org/abs/2402.17081.

[39] Red Hat. *New tools for automating end-to-end tests for VS Code extensions*. Red Hat Developer Blog [online]. 2019-11-18 [cit. 2025-03-28]. URL: https://developers.redhat.com/blog/2019/11/18/new-tools-for-automating-end-to-end-tests-for-vs-code-extensions.

[40] Red Hat. *vscode-extension-tester (ExTester): README and docs*. GitHub repository [online]. [cit. 2025-03-04]. URL: https://github.com/redhat-developer/vscode-extension-tester/.

[41] redhat-developer. *redhat-developer/vscode-extension-tester*. GitHub repository – ExTester for Visual Studio Code extensions. [cit. 2025-10-10]. URL: https://github.com/redhat-developer/vscode-extension-tester.

[42] Josef Richter. *New tools for automating end-to-end tests for VS Code extensions*. Red Hat Developer Blog [online]. 2019 [cit. 2025-03-04]. URL: https://developers.redhat.com/blog/new-tools-for-automating-tests-vscode-extensions.

[43] Binita Saha, Utsha Saha a Muhammad Zubair Malik. "QuIM-RAG: Advancing Retrieval-Augmented Generation With Inverted Question Matching for Enhanced QA Performance". In: *IEEE Access* 12 (2024), s. 185401–185410. DOI: 10.1109/ACCESS.2024.3513155.

[44] se2p. *se2p/pynguin*. GitHub repository – Pynguin automatic unit test generation for Python. [cit. 2025-12-10]. URL: https://github.com/se2p/pynguin.

[45] Selenium. *Selenium – Browser Automation*. Official site of the Selenium browser automation project. [cit. 2025-12-10]. URL: https://www.selenium.dev.

[46] Tabnine. *Tabnine – AI Code Completion*. Official site of the Tabnine AI coding assistant. [cit. 2025-12-10]. URL: https://www.tabnine.com.

[47] TechTarget. *Microsoft Copilot Copyright Commitment explained*. Explanation of Microsoft's Copilot Copyright Commitment addressing intellectual property concerns. [cit. 2025-12-10]. URL: https://www.techtarget.com/searchenterprisedesktop/tip/Microsoft-Copilot-Copyright-Commitment-explained.

[48] TestForge. *TestForge*. Official site of the TestForge AI-powered Python test case development tool. [cit. 2025-11-19]. URL: https://www.testforge.ai/.

[49] WebdriverIO. *WebdriverIO*. Framework for browser and automation testing. [cit. 2025-10-16]. URL: https://webdriver.io.

[50] xAI. *Grok – AI Assistant by xAI*. Official site for Grok AI assistant. [cit. 2025-11--19]. URL: https://grok.com/.

Přílohy

A Navrhnuté soubory s testy

Soubor
activation/activatesOnJavaLanguageOpen.test.ts
activation/activatesOnJsonLanguageOpen.test.ts
activation/activatesOnPropertiesLanguagesOpen.test.ts
activation/activatesOnStartupFinished.test.ts
activation/activatesOnWorkspaceContainsGlobs.test.ts
activation/activatesOnXmlLanguageOpen.test.ts
activation/activatesOnYamlLanguageOpen.test.ts
commands/camelNewFileEnablementRequiresResource.test.ts
commands/commandsHiddenWithoutWorkspace.test.ts
commands/createCamelRouteJavaCreatesFile.test.ts
commands/createCamelRouteXmlCreatesFile.test.ts
commands/createCamelRouteYamlCreatesFile.test.ts
commands/createKameletYamlCreatesFile.test.ts
commands/createPipeYamlCreatesFile.test.ts
commands/projectCreationLegacyCommandHidden.test.ts
commands/projectCreationQuarkusCommandAvailable.test.ts
commands/transformRouteCommandsPaletteVisibilityDependsOnActiveEditor.test.ts
commands/transformRouteToXmlFromYamlFile.test.ts
commands/transformRouteToYamlFromJavaFile.test.ts
commands/transformRoutesInFolderToXml.test.ts
commands/transformRoutesInFolderToYaml.test.ts
dialogs/createRouteFromOpenApiYamlPromptsAndCreates.test.ts
dialogs/multiFileTransformXmlOpensFilePicker.test.ts
dialogs/multiFileTransformYamlOpensFilePicker.test.ts
dialogs/newCamelFileSubmenuAccessibleFromFileMenu.test.ts
dialogs/transformRoutesInFilesToXmlMultiSelectFlow.test.ts
dialogs/transformRoutesInFilesToYamlMultiSelectFlow.test.ts
menus/camelSubmenuOrderingIsCorrect.test.ts
menus/camelSubmenuVisibilityForFileSelection.test.ts
menus/camelSubmenuVisibilityForFolderSelection.test.ts
menus/explorerContextShowsCamelSubmenu.test.ts
menus/fileNewMenuShowsNewCamelFileWithWorkspace.test.ts
settings/camelCatalogRuntimeProviderEnumOptions.test.ts
settings/camelCatalogVersionSettingPersists.test.ts
settings/camelExtraComponentsArrayPersists.test.ts
settings/jbangVersionDefaultAndOverrideUsed.test.ts
settings/languageServerJavaHomeSettingHandled.test.ts
settings/telemetryTogglePersists.test.ts
views/welcomeCreateProjectLinkInteraction.test.ts
views/welcomeViewShownInEmptyWorkspace.test.ts

Tabulka 4: Návrh testovacích scénářů na základě anylýzy manifestu.

B Kompilační chyby

Soubor	Počet chyb
activation/activatesOnJavaLanguageOpen.test.ts	1
activation/activatesOnJsonLanguageOpen.test.ts	2
activation/activatesOnPropertiesLanguagesOpen.test.ts	4
activation/activatesOnStartupFinished.test.ts	1
activation/activatesOnWorkspaceContainsGlobs.test.ts	2
activation/activatesOnXmlLanguageOpen.test.ts	4
activation/activatesOnYamlLanguageOpen.test.ts	5
commands/camelNewFileEnablementRequiresResource.test.ts	4
commands/createCamelRouteXmlCreatesFile.test.ts	1
commands/createCamelRouteYamlCreatesFile.test.ts	1
commands/createKameletYamlCreatesFile.test.ts	4
commands/createPipeYamlCreatesFile.test.ts	2
commands/projectCreationLegacyCommandHidden.test.ts	2
commands/projectCreationQuarkusCommandAvailable.test.ts	3
commands/projectCreationSpringBootCommandAvailable.test.ts	1
commands/transformRouteToXmlFromYamlFile.test.ts	2
commands/transformRouteToYamlFromJavaFile.test.ts	3
dialogs/newCamelFileSubmenuAccessibleFromFileMenu.test.ts	6
dialogs/transformRoutesInFilesToXmlMultiSelectFlow.test.ts	5
menus/camelSubmenuOrderingIsCorrect.test.ts	2
menus/camelSubmenuVisibilityForFileSelection.test.ts	5
menus/camelSubmenuVisibilityForFolderSelection.test.ts	5
menus/fileNewMenuShowsNewCamelFileWithWorkspace.test.ts	4
settings/camelCatalogRuntimeProviderEnumOptions.test.ts	9
settings/camelCatalogVersionSettingPersists.test.ts	8
settings/camelExtraComponentsArrayPersists.test.ts	5
settings/jbangVersionDefaultAndOverrideUsed.test.ts	10
settings/languageServerJavaHomeSettingHandled.test.ts	9
settings/telemetryTogglePersists.test.ts	6
views/welcomeCreateProjectLinkInteraction.test.ts	2

Tabulka 5: Přehled rozložení 118 kompilačních chyb mezi 30 testů.