FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Generování jednotkových testů s využitím LLM

Milan Horínek

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Generování jednotkových testů s využitím LLM

Bc. Milan Horínek

Vedoucí práce
Ing. Richard Lipka, Ph.D.

PLZEŇ **2024**

Citace v seznamu literatury:
HORÍNEK, Milan. *Generování jednotkových testů s využitím LLM*. Plzeň, 2024. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Richard Lipka, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

Studijní program: Softwarové a informační systémy
Forma studia: Prezenční

Podklad pro zadání DIPLOMOVÉ práce studenta

Jméno a příjmení:	**Bc. Milan HORÍNEK**
Osobní číslo:	**A23N0089P**
Adresa:	Česká 1024/6, Most, 43401 Most 1, Česká republika
Téma práce:	Generování jednotkových testů s využitím LLM
Téma práce anglicky:	LLM based unit test generator
Jazyk práce:	Čeština
Vedoucí práce:	Ing. Richard Lipka, Ph.D. Katedra informatiky a výpočetní techniky

Zásady pro vypracování:

1. Seznamte se s existujícími LLM, jejich technologií a možnostmi použití.
2. Prostudujte exitující literaturu ohledně genrování jednotkových testů, zejména s ohledem na využití neuronových sítí a LLM.
3. Seznamte se s existujícími ukázkovými programy s možností injekce chyb a vyberte vhodný testovací program.
4. Navrhněte a implementujte automatizovaný nástroj využívající popis testu v přirozené řeči, LLM a případně další informace, který vygeneruje sadu jednotkových testů.
5. Ověřte kvalitu automaticky vytvořených testů zejména s ohledem na přesnost a úplnost.
6. Zhodnoťte možnosti současných LLM pro generování testů.

Seznam doporučené literatury:

Dodá vedoucí práce.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum:

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Plzeň dne 14. května 2024

. . . . . .
Milan Horínek

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

TBD

Abstract

TBD

Klíčová slova

LLM • testing • unit • Robot Framework

Poděkování

TBD

Obsah

Úvod 1

TDB - Co jsou unit testy, LLM, a jejich předpoklady, GUI testování

Něco na úvod

1.1 Testy

1.2 Jazykové modely

1.3 Motivace

Rešerše 2

2.1 Provedená práce v problematice

2.1.1 Předchozí automatizovaná řešení

Jazykové modely nebyli prvními pokusy o automatizované generování jednotkových testů. Ještě před nimi existovala spousta metod zahrnující příklady jako *fuzzing, generování náhodných testů řízených zpětnou vazbou, dynamické symbolické exekuce, vyhledávvací a evoluční techniky, parametrické testování*. Zároveň také již na počátku století byli pokusy o vytvoření vlastní neuronové sítě sloužící právě čistě k úkolu testování softwaru. V této sekci je ukázka několika z nich.

2.1.1.1 Programatická řešení

Jedna z používaných programatických automatizovaných metod pro tvrobu jednotkových testů je tzv. *fuzzing*. V rámci těchto testů musí uživatel stále definovat jeho kódovou strukturu, resp. akce, které test bude provádět a jaký výstup očekávat. Automaticky generovaný je pouze vstup tohoto testu. Výhodou zde tedy je, že uživatel nemusí vytvářet maketu vstupních dat testu, která se zde vytvoří automatizovaně. Zůstává zde však problematika, že pro uživatele není kód *black-box*, ale celou jeho strukturu včetně požadovaného výstupu musí sám definovat. [**fuzzing**]

Pouze vstupy dokáže také generovat metoda *symbolické exekuce*, která postupně analyzuje chování větvení programu. Začíná bez předchozích znalostí a používá řešitel omezení k nalezení vstupů, které prozkoumají nové exekuční cesty. Jakmile jsou testy spuštěny s těmito vstupy, nástroj sleduje cestu, kterou se program ubírá, a aktualizuje svou znalostní bázi (q) s novými podmínkami cesty (p). Tento itera-

tivní proces se opakuje a nadále zpřesňuje sadu známých chování a snaží se maximalizovat pokrytí kódu. Nástroje běžně zvládají různé datové typy a respektují pravidla viditelnosti objektů. Snaží se také používat mock objekty a parametrizované makety k simulaci různých chování vstupů, čímž zlepšuje proces generování testů, aby odhalila potenciální chyby a zajistila komplexní pokrytí kódu testy. [**parizek_symbolic_execution**] Tato metoda je implementována například v nástroji IntelliTest v rámci IDE *Visual Studio*. Je používáná v kombinaci s *parametrickými testy*, také označovanými jako PUT. Ty na rozdíl od tradičních jednotkových testů, které jsou obvykle uzavřené metody, mohou přijímat libovolnou sadu parametrů. Nástroje se pak snaží automaticky generovat (minimální) sadu vstupů, které plně pokryjí kód dosažitelný z testu. Nástroje jako např. *IntelliTest* automaticky generují vstupy pro put, které pokrývají mnoho exekučních cest testovaného kódu. Každý vstup, který pokrývá jinou cestu, je "serializován"jako jednotkový test. Parametrické testy mohou být také generické metody, v tom případě musí uživatel specifikovat typy použité k instanci metody. Testy také mohou obsahovat atributy pro očekávané a neočekávané vyjímky. Neočekávané vyjímky vedou k selhání testu. put tedy do velké míry redukují potřebu uživatelského vstupu pro tvorbu jednotkových testů. [**IntelliTestInputGeneration2023**] [**microsoft2023testgen**]

Pokud zvolíme symbolické řešení vstupu společně s determinovanými vstupy a testovací cestou, vzniká tak hybridní řešení zvané jako *konkolické testovaní* nebo *dynamická symbolická exekuce*. Tento druh testů dokáží tvořit nástroje jako *SAGE, KLEE* nebo *S2E*. Problémem tohoto přístupu však je, když program vykazuje *nedeterministické* chování, kdy tyto metody nebudou schopny určit správnou cestu a zároveň tak ani zaručit dobré pokrytí kódu/větví. Velká míra používání stavových proměnných může vést k vysoké výpočetní náročnosti těchto metod a nenalezení praktického řešení. [**engler2006exe**] [**sen2005cute**] [**zhou2006safedrive**]

Další metodou je *náhodné generování testů řízené zpětnou vazbou*, která je vylepšením pro generování náhodných testů tím, že zahrnuje zpětnou vazbu získanou z provádění testovacích vstupů v průběhu jejich vytváření. Tato technika postupně buduje vstupy tím, že náhodně vybírá metodu volání a hledá argumenty mezi dříve vytvořenými vstupy. Jakmile dojde k sestavená vstupu, je provedena jeho exekuce a výsledek ověřen proti sadě kontraktů a filtry. Výsledek exekuce určuje, zda je vstup redundantní, proti pravidlům, porušující kontrakt nebo užitečný pro generování dalších vstupů. Technika vytváří sadu testů, které se skládají z jednotkových testů pro testované třídy. Úspěšné testy mohou být použity k zajištění faktu, že kódu jsou zachovány napříč změnami programu; selhávající testy (porušující jeden nebo více kontraktů) ukazují na potenciální chyby, které by měly být opraveny. Tato metoda dokáže vytvořit nejen vstup pro test, ale i tělo (kód) testu. Ovšem pro uživatele je

stále vhodné znát strukturu kódu. [**FeedbackDirectedRT**]

Z programatických metod se zdají být nejpokročilejší *evoluční algoritmy* pro generování sad jednotkových testů, využívající přístup založený na vhodnosti, aby vyvíjely testovací případy, které mají za cíl maximalizovat pokrytí kódu a detekci chyb. Tyto algoritmy mohou autonomně generovat testovací vstupy, které jsou navrženy tak, aby prozkoumávaly různé exekuční cesty v aplikaci. Uživatelé mohou interagovat s vygenerovanými testy jakožto s *black-boxem*. Testy se zaměřují na vstupy a výstupy, aniž by potřebovali rozumět vnitřní logice testovaného systému. Tento aspekt evolučního testování je zvláště výhodný při práci se složitými systémy nebo když zdrojový kód není snadno dostupný. Proces iterativně upravuje testovací případy na základě pozorovaných chování, upravuje vstupy pro efektivnější prozkoumání systému a identifikaci potenciálních defektů. Tato metoda podporuje vysokou úroveň automatizace při generování testů, snižuje potřebu manuálního vstupu a umožňuje komplexní pokrytí testů s menším úsilím. [**CAMPOS2018207**] [**abs-2111-05003**]

2.1.1.2 Neuronové sítě

Ke generování jednotkových testů lze využít i vlastní neuronové sítě. Takové se pokoušeli vytvářet například v práci "Unit test generation using machine learning"[**Saes2018UnitTestGenera**
kde byly testovány primárně RNN a experimentálně CNN sítě (ty však měli problém s větším množstvím tokenů). Modely byli testovány na jazyce Java. Metoda přistupovala k programům jakožto white box, tedy měli k dispozici celý zdrojový kód včetně zkompilovaného bytecodu. Při nejlepší konfiguraci dosáhl výsledek modelu 70.5% parsovatelného kódu (tedy takového bez chyb) natrénovaný z bezmála 10000 příkladů *zdrojový kód - test*. Výsledek práce je však stále jakýsi "proof of concept", protože zatímco vygenerují částečně použitelný výsledek, je vždy nutný zásah experta, aby mohlo dojít k vytvoření celého testovacího souboru. Takovéto sítě se však mohou silně hodit jako výpomoc programátorovi při psaní testů.

2.1.1.3 Nevýhody současných metod

Současné metody generování jednotkových testů, jako je *fuzzing* a *symbolická exekuce*, často vyžadují podrobnou znalost struktury kódu a očekávaných výstupů, což omezuje jejich efektivitu a zvyšuje složitost tvorby testů. Jen některé z těchto metod jsou schopné vygenerovat testy pouze na bázi specifikace (z black box pohledu) bez vnitřní znaloasti kódu. Většina z klasických metod je zároveň schopna generovat

pouze vstupy jednotkových testů, ale už ne samotné tělo (kód) testu nebo očekávané výstupy, a tedy pouze složí jako jakási konstra pro programátora, který musí test doimplementovat.

Velké jazykové modely (LLM) mohou být atraktivní alternativou, protože pomocí nich lze potenciálně automatizovat generování jak vstupů pro testy, tak přidruženého testovacího kódu, čímž se snižuje potřeba hlubokého porozumění struktuře kódu. S takovým nástrojem není potřeba programátora, ale může s ním pracovat i méně zkušený uživatel (např. *tester*). Dále je zde možnost otestovat kód za pomocí slovní specifikace pro funkci nebo vlastnosti. Takové specifikace se používají například v *aero-space* nebo *automotive* sektoru. LLM také mohou objevit všechny možné stavy, které mohou u vstupu nebo výstupu nastat a pokusit se pro ně navrhnout test.

2.1.2 Vydané publikace

Jeden z poměrně nedávno vydaných článků (září 2023) nazvaný "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation"[**schafer2023empirical**] se zabývá využitím velkých jazykových modelů (LLM) pro automatizované generování jednotkových testů v jazyce JavaScript. Implementovali nástroj s názvem **TESTPILOT**, který využívá LLM *gpt3.5-turbo*, *code-cushman-002* od společnosti OpenAI a také model StarCoder, který vznikl jako komunitní projekt [**StarCoder2023**]. Vstupní sada pro LLM obsahovala signatury funkcí, komentáře k dokumentaci a příklady použití. Nástroj byl vyhodnocen na 25 balíčcích npm obsahujících celkem 1684 funkcí API. Vygenerované testy dosáhly pomocí gpt3.5-turbo mediánu pokrytí příkazů 70,2% a pokrytí větví 52,8%, čímž překonaly nejmodernější techniku generování testů v jazyce JavaScript zaměřenou na zpětnou vazbu, Nessie.

Zmíněný model *StarCoder* byl představen v článku "StarCoder: may the source be with you!"[**StarCoder2023**] z května 2023. Vytvořeny byly konkrétně 2 verze, *StarCoder* a *StarCoderBase*, s 15,5 miliardami parametrů a délkou kontextu 8K. Tyto modely jsou natrénovány na datové sadě nazvané *The Stack*, která obsahuje 1 bilion tokenů z permisivně licencovaných repozitářů GitHub. *StarCoderBase* je vícejazyčný model, který překonává ostatní modely open-source LLM modely, zatímco *StarCoder* je vyladěná verze speciálně pro Python, která se vyrovná nebo překoná stávající modely zaměřené čistě na Python. Článek poskytuje komplexní hodnocení, které ukazuje, že tyto modely jsou vysoce efektivní v různých úlohách souvisejících s kódem.

Článek "Exploring the Effectiveness of Large Language Models in Generating

Unit Tests"[**siddiq2023exploring**] z dubna 2023 hodnotí výkonnost tří LLM - *Codex*, *CodeGen* a *GPT-3.5* - při generování jednotkových testů pro třídy jazyka Java. Studie používá jako vstupní sady dva benchmarky, HumanEval a Evosuite SF110. Klíčová zjištění ukazují, že *Codex* dosáhl více než 80% pokrytí v datové sadě *HumanEval*, ale žádný z modelů nedosáhl více než 2% pokrytí v benchmarku *SF110*. Kromě toho se ve vygenerovaných testech často objevovaly tzv. testové pachy, jako jsou *duplicitní tvrzení* a *prázdné testy* [**testsmells**].

2.1.2.1 Srovnání výsledků

Výsledky diskutovaných studií v předchozím bodě jsme srovnali v tabulce 2.1. V rámci první práce dosahuje nejlepších výsledků model *gpt-3.5-turbo*, který dosáhl 70% pokrytí kódu testy a 48% úspěšnosti testů. U druhé studie má tento model na testovací sadě *HumanEval* velice podobný výsledek, ovšem model *Codex* dosáhl lepších výsledků. Může však také jít o rozdíl způsobený programovacím jazykem. Zatímco v práci [**schafer2023empirical**] se využívá jako benchmark sada balíčků jazyka *JavaScript*, který kvůli absenci explicitního typování, může být obtížnější pro strojové testování oproti jazyku Java, který je využit ve zbylých 2 pracích. [**jutai**] také využívá Javu a s modelem *gpt-3.5-turbo* dosahuje podobného pokrytí kódu a úspěšnosti jako *Codex* v práci [**siddiq2023exploring**].

2.1.3 Modely

Na LLM modelech nás konkrétně zajímá schopnost pozorumět progamovacím jazykům a ty poté také generovat na výstupu. Důležité pro nás také je, zda daný model je proprietární či otevřený a pod jakou licencí, tedy zda by byl vhodný pro naši práci. V případě analýzy zdrojového kódu může být také klíčovou vlastností délka kontextu daného modelu. Tyto vlastnosti jsou také zaneseny do tabulky 2.2.

StarCoder. Jedním z často používaných modelů v předchozích pracích je *StarCoder* a *StarCoderBase*, diskutovaný již v sekci 2.1. *Base* verze je schopna generovat kód pro více jak 80 programovacích jazyků. Model je navržen pro širokou škálu aplikací obsahující *generování*, *modifikaci*, *doplňování* a *vysvětlování* kódu. Jeho distribuce je volná a licence **CodeML OpenRAIL-M 0.1** [**BigCode2023**] umožňuje ho využívat pro množství aplikací včetně komerčních nebo edukačních. Jeho uživatel však má povinnost uvádět, že výsledný kód byl vygenerován modelem. Licence má své

restrikce z obavy tvůrců, protože by model mohl někoho při nepsrávném použití ohrozit. Tyto restrikce se aplikují na na všechny derivace projektů pod touto licencí. Zároveň není kompatibilní s Open-Source licencí právě kvůli těmto restrikcím.

Model StarCoder se také dočkal novější verze *StarCoder2*, který nepřímo navazuje na model *StarCoderBase*. Je naučen na archivu GitHub repoziářů archivovaných v rámci organizace *Software Heritage*, čítajících přes 600 programovacích jazyků a dalších pečlivě vybraných dat jako například *pull requesty*. Mimo toho také trénovací data obsahují staženou dokumentaci k vybraným projektům. Model se vyjímá tím, že se snaží udržet malou velikost. Je nabízen ve verzích s *3, 7* a *15* miliardy parametrů, i přesto však dle jejich úvodní studie [**lozhkov2024starcoder**] nabízí na sadě populárních programovacích jazyků shodné či lepší výsledky oproti modelu *CodeLlama* s 34 miliardy parametrů. Výhodou tohoto modelu nepochybně je, že ho lze spustit na spoustě dnešních počítačů s plným offloaded na grafickou kartu.

CodeLlama. Nedávno vydaným modelem je *Code Llama* od společnosti Meta. Jedná se o evoluci jejich jazykového modelu *Llama* specializovaný však čistě na úlohy kódování. Je postaven na platformě *Llama 2* a existuje ve třech variantách: *základní Code Llama*, *Code Llama - Python* a *Code Llama - Instruct*. Model podporuje více programovacích jazyků, včetně jazyků jako Python, C++, Java, PHP, Typescript, C nebo Bash. Je určen pro úlohy, jako je generování kódu, doplňování kódu a ladění. Code Llama je zdarma pro výzkumné i komerční použití a je uvolněn pod licencí MIT. Uživatelé však musí dodržovat zásady přijatelného použití, ve kterých je uvedeno, že model nelze použít k vytvoření služby, která by konkurovala vlastním službám společnosti Meta.

Copilot. Velmi populárním nástrojem pro generování kódu za pomocí LLM je GitHub Copilot, který je postaven na modelu *codex* od OpenAI. Původní model však byl přestal být zákazníkům nabízen a namísto něj OpenAI doporučuje ke generování kódu využívat chat verze modelů GPT-3.5 a GPT-4. Na architektuře GPT-4 je také postavený nástupce služby Copilot, Copilot X. Zmíněné modely chat GPT-3.5 a GPT-4 jsou primárně určeny pro generování textu formou chatu. Zvládají však zároveň i dobře generovat kód a jsou vhodné i úlohu generování jednotkových testů. Narozdíl od předchozích modelů však nejsou volně distribuovány a jsou poskytovány pouze jako služba společností OpenAI skrze API nebo je lze hostovat v rámci služby Azure společnosti Microsoft, která zajišťuje větší integritu dat. Jedná se tedy o uzavřený model a jeho uživatelé musí souhlasit s jeho podmínkami použití.

Práce	Model	Benchmark	Pokrytí testy	Úspěšnost
An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	*gpt-3.5-turbo*	Sada NPM balíčků	70.2%	48%
	code-cushman-002		68.2%	47.1%
	StarCoder		54%	31.5%
Exploring the Effectiveness of Large Language Models in Generating Unit Tests	*gpt-3.5-turbo*	HumanEval	69.1%	52.3%
		SF110	0.1%	6.9%
	CodeGen	HumanEval	58.2%	23.9%
		SF110	0.5%	30.2%
	Codex (4k)	HumanEval	87.7%	76.7%
		SF110	1.8%	41.1%
Java Unit Testing with AI: An AI-Driven Prototype for Unit Test Generation	*gpt-3.5-turbo*	JUTAI - Zero-shot, temperature: 0	84.7%	71%

Tabulka 2.1: Přehled a srovnání studií

Model	Otevřenost	Licence	Počet parametrů	Počet programovacích jazyků	Datum vydání
StarCoderBase	Volně dostupný	CodeML OpenRAIL-M 0.1	15.5 miliardy	80+	5/2023
Code Llama	Volně dostupný	Llama 2 Licence	34 miliard	8+	8/2023
gpt-3.5-turbo	Uzavřený	Proprietární	175 miliard?	?	5/2023
gpt-4	Uzavřený	Proprietární	1.76 bilionu	?	3/2023

Tabulka 2.2: Přehled a srovnání modelů generujících kód

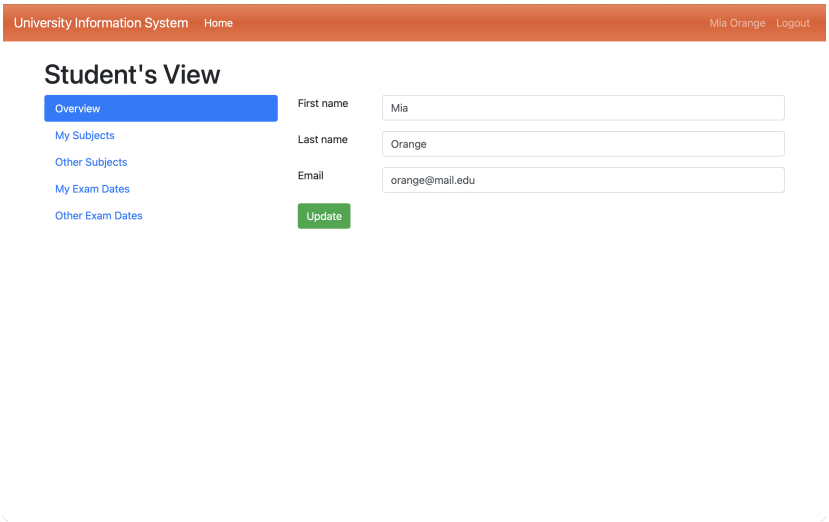
2.2 Testovací program

Pro účely této práce jsme se rozhodli vydat cestou GUI testů, konkrétně webových stránek / webové aplikace. Mezi požadavky na testovanou aplikaci a její výběr bylo mimo možnosti vytvořit pro ní automatiovaně sadu testů, také možnost zavedení (*injekce*) chyb do aplikace, díky kterým bude možno ověřit nejen fakt, že vygenerované testy jsou schopné detekovat *korektní* chování softwaru, ale také úspěšně detekovat *chyby*. Volitelnou podmínkou také byla existence již existujících testů vytvořených lidským programátorem, se kterými by bylo možné strojově generované testy porovnat.

Těmto požadavkům vyhovuje jeden z předešlých univerzitních projektů nazvaný **TbUIS** (*Testbed University Information System*), na který vytvořili Matyáš a Šmaus pod vedením doc. Herouta. [**Matyas2018**] [**Smaus2019**] Aplikace představuje *maketu* univerzitního informačního systému, avšak i tak implementuje většinu functionality, kterou bysme od podobného systému čekali a nabízí pohled jak ze strany *studenta* tak ze strany *vyučujícího* (viz obr. 2.1). Funkce tohoto systému vycházejí ze sady *use casů*, které popisuje web [1] aplikace. Výhodou tohoto systému primárně je, že nabízí nejen plně funkční a otestovanou variantu, ale také 27 dalších poruchových klonů, vždy porušující alespoň jeden *use case* a s ním potenciálně spojený test.

Projekt dále také obsahuje i sadu jak *funkčních* testů primárně pro backend vytvořených Poubovou tak také *akceptačních* testů držejících se pevně specifikací, vytvořených Vaisem. [**Poubova2019**] [**Vais2020**] Tyto testy využívají nejen pevné znalosti jednotlivých uživatelských scénářů, ale také jsou parametrizovány pomocí všech dostupných dat k aplikaci (*např. uživatelská jména, předměty, zkoušky, atd.*) k vyhodnocení chování softwaru v co nejvíce případech. Právě *akceptační* testy využívají námi zvolený **RobotFramework** pro testování a tedy vytvořené scénáře bude možné do určité míry přirovnat a případně zhodnotit jejich kvalitu.

[1] Aplikaci lze nalézt na adrese: `https://projects.kiv.zcu.cz/tbuis/web/`



Obrázek 2.1: Prostředí systému TbUIS z pohledu studenta.

Návrh 3

3.1 Co a jak testujeme

3.2 Navrhované řešení

Generování testů 4

4.1 Prerekvizity pro generování spuštění testů

4.2 Výběr scénářů

Pro vygenerování testů za pomocí LLM bylo nejdříve nutné vybrat ze sady *use caseů*[1] scénáře vhodné pro demonstraci nejen správné funkčnosti, ale také s možností ověření nesprávného chování na poruchových klonech (diskutováno v sekci 2.2). Požadavek tedy byl, aby většina z vybraných scénářů vycházející z *use casu* měl alespoň jeden poruchový klon, případně více. Vybráno bylo 10 scénářů, testovaných celkově na 14 variantách programu. Seznam vytvořených specifikací pro automatické generování testů se nachází v tabulkách 4.2 a 4.2. Scénáře budou dále také referovány jako *specifikace*. Každá specifikace má *číslo*, které odpovídá číslu *use casu*, ze kterého vychází (např. specifikace 04 vychází z use casu *UC.04*). Zde je nutné poznamenat, že ne všechny specifikace vychází pevně z jejich use casů, ale byli upravené tak, aby šli provést v jednom chodu bez nutných závislostí (*jako například namísto podmínky přihlášeného uživatele se uživatel vždy musí na začátku či během testu přihlásit do systému*). Popis specifikací v tabulce je pouze orientační a pro bližší upřesnění jednotlivých kroků, které se mají provést referujte web projektu. V tomto seznamu se také u každé specifikace nachází výčet klonů, na kterých lze očekávat poruchu. Celkový seznam klonů, a kterých se budou všechny vytvořené testy spouštět lze nalézt v tabulce 4.2. Podobně jako v případě specifikací, čísla klonů odpovídají číslům, jak jsou uvedena na oficiálním webu[2] projektu společně s vysvětlením jednolivých chyb

[1] Seznam dostupný na adrese: `https://projects.kiv.zcu.cz/tbuis/web/page/uis#use-cases`

[2] Seznam poruchových klonů společně s možností stažení: `https://projects.kiv.zcu.cz/tbuis/web/page/download`

varianty. Pro přehlednost jsme bezchybnou variantu označili číslem 00.

4.3 Nahrávání scénářů

Dle *specifikace* z předchozího bodu, vytvoří uživatel LLM generačního nástroje nahrávku jednotlivých kroků, které má test provést. V případě, že specifikace udává více uživatelských pohledů (*student / učitel*), nahraje uživatel tyto pohledy jednotlivě. Současný projekt pracuje s nahrávacím nástrojem v rámci vývojářských nástrojů webového prohlížeče *Google Chrome* (pro vývoj byla využita verze 124). Tento nástroj je stále experimentální funkce, takže nelze vyloučit možnost, že v následujících verzích již nemusí být dostupný. Díky této funkci lze nahrávat uživatelské vstupy a interakce s jednotlivými prvky v rámci webových stránek. Mimo jiné také umožňuje přidávat *asserty*, avšak tyto možnosti jsou nedostatečné a tedy v rámci této práce se omezíme pouze na údaje o interakci s prvky. Nahrávání zobrazeno na obr. 4.1. Nástroj je schopen vyexportovat výstup nahrávání v řadě formátů. Zde jsme zvolily JSON formát, který popisuje jednotlivé akce dle objektů. Nahrané scénáře uživatel vhodně pojmenuje a uloží do složky `input` programu, který je součástí tohoto projektu. Vytvořený program pro generování testů za pomocí LLM se stará o veškerou orchestraci generování i spouštění testů a také vytovření *šablon* pro testy, ze kterých se generují. Součástí těchto *šablon* je právě i uživatelova nahrávka. Ukázkovou strukturu `input` složky lze vidět ve výpisu 4.1, kde se nachází *šablona* pro *specifikace* 1 a 4. Význam *šablon* je diskutován v následující sekci.

Specifikace	Popis	Porucha na klonech
1	*Přihlášení do aplikace* Student i učitel se přihlásí do aplikace přihlašovacími údaji, dostupnými v databázi systému. Dále se zkontroluje, zda systém pro účet s neexistujícím uživatelským jménem nebo neplatným heslem vypíše chybovou hlášku.	02
4	*Odepsání předmětu* Student se přihlásí, odepíše předmět v patřičné sekci systému. Předmět by následně měl zmizet v ostatních sekcích a to jak v pohledu studenta tak učitele.	04, 19, 25, 26, 28
6	*Zapsání předmětu* Student se přihlásí, zapíše předmět v patřičné sekci, který se následně zobarazí i v ostatních částech systému. Z učitelského pohledu by se student měl zobrazit na seznamu studentů daného předmětu.	25, 26, 28
8	*Registrace na zkoušku* Student se přihlásí do systémů a zapíše se na jeden z možných zkouškových termínů. Tento termín by se měl přesunout mezi již zapsané termíny. V učitelském pohledu bude student na seznamu zapsaných na konkrétní zkoušku a také by mělo být možno studenta ohodnotit.	22, 25, 26, 28
9	*Zobrazení spolužáků u zkoušky* Student se přihlásí a u zkoušky si může rozkliknout seznam všech účastníků.	
10	*Zrušení předmětu* Učitel po přihlášení klikne na tlačítko *Remove* u předmětu, od jehož výuky se chce odhlásit. Ten se přestane zobrazovat ve všech ostatních sekcích systému z jeho pohledu, až na jeho znovuzapsání. Student by u předmětu neměl najít jméno daného učitele, který se odhlásil.	26, 28
11	*Zobrazení studentů u předmětu* Učitel se přihlásí do systému a u předmětu je schopen si zobrazit seznam zapsaných studentů.	26, 28

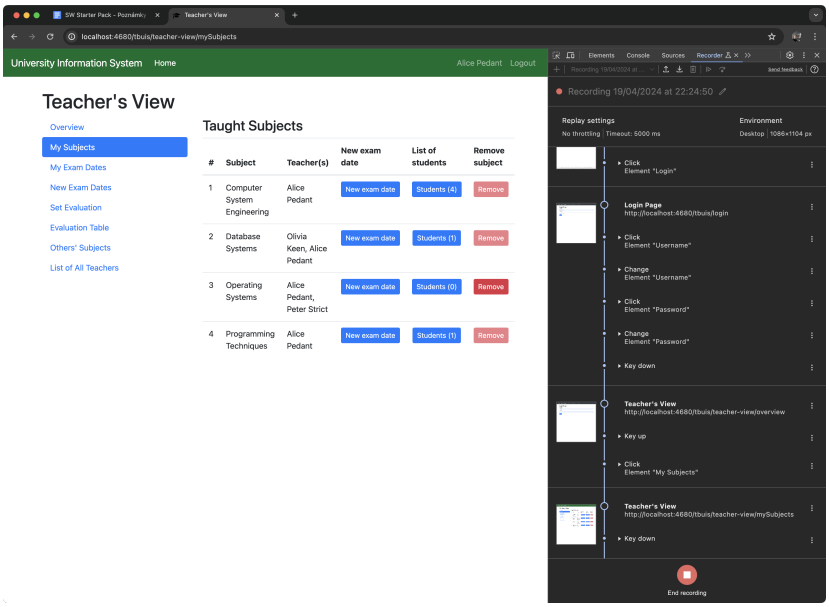
Tabulka 4.1: Specifikace pro generované testy - část 1

Specifikace	Popis	Porucha na klonech
12	*Zrušení zkoušky* Učitel se přihlásí a v sekci jeho přidělených zkoušek odstraní konkrétní termín. Tento termín by nyní neměl být vidět jak v učitelském tak studentském pohledu do systému.	20, 21, 23, 26, 28
17	*Přihlášení se k výuce předmětu* Učitel se přihlásí do systému a v seznamu předmětů se přihlásí k výuce daného předmětu. Ten by se poté měl zobrazit ve zbytku systému v rámci patřičných sekcí. Zároveň studenti by nyní měli u předmětu vidět jméno tohoto vyučujícího.	18, 24, 25, 26, 27, 28
18	*Zobrazení seznamu učitelů a předmětů, které vyučují* Přihlášený učitel je schopen si zobrazit seznam všeech učitelů.	27, 28

Tabulka 4.2: Specifikace pro generované testy - část 2

Číslo klonu	Porucha
00	Bez defektu
02	Překlep v nadpisu
04	Návrat na špatnou stránkau
18	Chybějící sloupec v tabulce
19	Náhodně chybějící tlačítko
20	Nefunkční tlačítko
21	Změna se nepropíše do UI
22	Nefunkční tlačítko
23	Smazání se nepropíše do DB
24	Přidání se nepropíše do DB
25	Tabulka studentů prázdná
26	Tabulka učitelů prázdná
27	Nesprávný výběr z DB
28	Mix chyb (včetně interní chyby systému)

Tabulka 4.3: Seznam poruchových klonů využitých pro testování.



Obrázek 4.1: Nahrávání scénáře za pomocí nástroje v Google Chrome

Výpis 4.1: Ukázková struktura input složky

```
milan:dp\program\input$ ls -l
milan     448 Apr 21 20:52 ..
milan    3815 Mar 27 20:24 rec-spec-1-student.json
milan    3820 Mar 27 20:28 rec-spec-1-teacher.json
milan    6043 Mar 30 19:42 recording-spec-4-student.json
milan   10635 Mar 31 09:18 recording-spec-4-teacher.json
milan    1370 Apr 21 20:52 spec-1.txt
milan    1328 Mar 31 09:28 spec-4.txt

```

4.4 Výběr požadavků

4.4.1 Vytvoření nového testu

Primárním programem celé práce je soubor `test.py`, který se nalézá v kořenové složce programu (`/program`) a má konrétně 3 pracovní režimy:

1. Vyvtvoření šablony pro test
2. Vygenerování testu dle šablony

3. Spuštění sady testů

Právě první režim je v současném kroce důležitý. Jednotlivé režimy programu jsou spuštěny za pomocí argumentů (argumenty jednotlivých režimů ukázané ve výpisu 4.2). V případě *vytvoření* nového testu se za argument přidává název šalony testu. Tento název musí být unikání a využívá se později ve zbytku programu jakožto identifikátor daného testu. Po potvrzení příkazu se ve vstupní složce vytvoří nová šablona pro test se zvoleným názvem a příponou `.txt`. Tuto šablonu může uživatel může dále upravit. Celý příkaz je ukázaný ve výpisu 4.3. Vytvořenou šablonu uživatel zobrazí a upraví v *textovém editoru*.

Výpis 4.2: Hlavní režimy programu

```
milan:dp\program\input$ python3 test.py -n   #Nový test
milan:dp\program\input$ python3 test.py -i   #Generování
milan:dp\program\input$ python3 test.py -r   #Spuštění testů

```

Výpis 4.3: Vytvoření nového testu (šablony)

```
milan:dp\program\input$ python3 test.py -n specficication-1

```

Zdrojový kód 4.4: Vzor pro vyplnění šablony testu

```
Write Robot Framework scanario. Open page like in this JSON
    recording and then when you execute all the steps in the
    recording, do this:

- //TODO

{% include 'recording.json' %}

```

Zdrojový kód 4.5: Vyplněná šablona testu pro specifikaci 18

```
Write Robot Framework scanario. Open page like in this JSON
    recording and then when you execute all the steps in the
    recording, do this:

- Check if there are these names present on the page: Julia
    Easyrider, Olivia Keen, John Lazy, Alice Pedant, Thomas
    Scatterbrained, Peter Strict
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-0"]/td[3] has text matching "Numerical
    Methods"
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-1"]/td[3] has text matching "Database
    Systems, Fundamentals of Computer Networks, Introduction
    to Algorithms, Mobile Applications, Web Programming"
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-2"]/td[3] should not contain text
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-3"]/td[3] has text matching "Computer
    System Engineering, Database Systems, Operating Systems,
    Programming Techniques"
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-4"]/td[3] has text matching "Computation
    Structures"
- Check if element with path //*[@id="tea.listOfAllTeachers.
    table.teacherRow-5"]/td[3] has text matching "Operating
    Systems, Programming in Java, Software Engineering,
    Software Quality Assurance"

{% include 'recording-spec-18.json' %}

```

4.4.2 Vyplnění požadavků

Do šablony je vložen základní prompt společně s požadavky, které uživatel může vyplnit (viz ukázka 4.4). Pod požadavky je defaultně vložena nahrávka `recording.json`. Tuto hodnotu nahradí uživatel za název souboru vložené nahrávky. Formát šablony jakožto vstupu pro prompt testu byl zvolen pro jednodušší parametrizaci. Šablona využívá jazyk *Jinja2*. V rámci vytvořených šablon v této práci byly tyto parametrizační vlastnosti využity například pro vytvoření pohledu *studenta* a *učitele*. Místo, které je v šabloně označeno jako `\\TODO` slouží pro napsání požadavků testu. Od uživatele se čeká, že tyto požadavky vypíše v odrážkách, čemuž odpovídá i formát *promptu*. Ukázka vyplněných požadavků pro konkrétní test je součástí výpisu 4.5,

Tvůrce	Model	Použitá verze	Runtime
OpenAI	GPT-4	gpt-4-32k	API
	GPT-4 Turbo	gpt-4-turbo-2024-04-09	
	GPT-3 Turbo	gpt-3-turbo-0125	
Mistral	Mistral v0.2 7B	TBD	Lokální
	Mistral Large	mistral-large-2402	API *La Plateforme*
Anthropic	Claude 3 Opus	claude-3-opus-20240229	API
Meta	Codellama	TBD	Lokální
Google	Gemini 1.5 Pro	gemini-1.5-pro-preview-0409	API *Google Cloud*

Tabulka 4.4: Použité LLM modely

který ukazuje vypsané vlastnosti pro specifikaci 18, jak bylo popsáno v sekci 4.2.

4.5 Dotazování LLM

4.5.1 Modely a jejich spuštění

Jak již bylo řečeno v kapitole 2, velké jazykové modely existují jak v prorietární formě, které jsou dostupné pouze přes API poskytovatele nebo webové rozhraní, ale také se dají najít modely dostupné v *public* či *open source* formě dále referovány jako *otevřené* modely). V rámci tohoto projektu byly využity obě varianty, tedy *proprietární* modely srkze API poskytovatele a otevřené lokálně nebo také srze API u některého z poskytovatelů a to z důvodu vysokých hardwarových nároků některých modelů.Konkrétní seznam použitých modelů v práci včetně typu přístupu k nim lze nalézt v tabulce 4.4. Pro dotazování vzdálených modelů využívá většina koncových bodů API od OpenAI, tedy jsou kompatibilní s jejich knihovnou pro různé jazyky. Některé společnosti však pro své modely využívají vlastní definici API, mezi ně se řadí například *Anthropic* nebo *Google*. V projektu tedy využíváme pro veškeré kompatibilní modely (resp. jejich běhová prostředí) knihovnu pro OpenAI API, se kterým jsou kompatibilní, a pro zbytek jejich vlastní knihovny.

4.5.1.1 Prostředí pro lokální modely

Pro *lokální* modely a jejich spuštění byl využit software *LM Studio*, který je postaven nad *C++* knihovnou `llama.cpp`. Ta umožňuje jednoduché spuštění LLM modelů ve formátu GPT-Generated Unified Format (GGUF). Jedná se o *binární* for-

mát souborů určený pro rychlé načítání a ukládání modelů využíváných pro účely interferenčních úloh. Jeho návrh umožňuje snadnou úpravu při zachování zpětné kompatibility. [**ggerganov_gguf**] [**huggingface_gguf**] Modely mohou být vyvíjené za pomoci různých frameworků (např. *PyTorch*) a poté převedeny do GGUF formátu pro použití v rámci GGML knihovny, která umožňuje efektivní spuštění modelů na CPU a GPU. Knihovna také umožňuje *kvantizaci*, tedy techniku využívanou ke snížení paměťových nároků ML úloh. Zahrnuje reprezentaci vah a aktivací za pomoci datakových typů s nižší přesností (např. *int8*) oproti obvyklým 32 bitovým číslům (např. *float32*), ve kterých bývají reprezentována originální data modelu. Jejím cílem je snížit počet bitů, což vede k nižší velikosti modelu a to k nižším HW nárokům a s tím spojených vlastností (*snížená energetická spotřeba, nižší latence, ...*). Kvantizace má však i nežádoucí dopady a to *ztrátu přesnosti* nebo případně *výkonu* modelu. [**huggingface_quantization**] GGUF tedy tvoří jednotný formát, ve kterém se spousta otevřených LLM modelů (případně jejich konverzí) distribuuje. Jednou z platforem pro jejich distribuci je například HuggingFace.

LM Studio umožňuje spuštění právě modelů ve formátu GGUF, včetně jejich stažení z repozitářů a funkcemi s tím spojených (například *vypsání seznamu kvantizací, spojení rozdělených modelů, atd.*). Jeho primární funkcí je spouštět lokální LLM modely jako *chat* nebo *lokální server* nabízející API kompatibilní s definicí dle OpenAI. Pro každý druh modelů (jako *Llama, Mistral, Command R, ...*) také obsahuje předdefinované nastavení, které si uživatel může upravit. Mezi tímto nastavením jsou hyperparametry jako *teplota, Min P, Top P, délka kontextu, maximálního počtu vygenerovaných tokenů a dalších*. Mimo toho obsahuje i nastavení pro hardware jako *počet vláken CPU, počet GPU vrstev, GPU framework* a podobně. *Vrstvou* je zde myšlena vrstva neutronů, která provádí určit výpočty a zpracování vstupních dat. Vrstvy mohou být ku příkladu *vstupní, skryté, výstupní*. Klíčovou vlastností nastavitelnou v rámci runtimu tohoto softwaru je možnost nastavit chování kontextového okna. Mezi možnosti se řadí:

1. *Klouzavé okno* - Model si pamatuje posledních x tokenů z konverzace, které využívá ke generování výstupu a predikci.
2. *Začátek a konec* - Model si pamatuje první prompt (případně i systémový) a zbylé tokeny na konci konverzace.
3. *Zastavení* - Při dosažení počtu tokenů v rámci konverzace rovnající se x, je generování odpovědi zastaveno.

Proměnnou x je zde myšlena celková délka kontextu, kterou *model* nebo *runtime* podporuje. V případě *runtimu* v rámci *LM Studio* (resp. `llama.cpp`) může uživatel

Typ	Komponenta
CPU	AMD Ryzen 8700F
Základní deska	Asus B650-PLUS TUF
CPU Chlazení	BOX
RAM	Kingston FURY Renegade DDR5 64GB 6400 MHz (4x16GB)
VGA	Radeon RX 7900 XT 20GB Asus TUF
PSU	ADATA XPG CYBERCORE 1300W
Case	BeQuiet! Dark Base Pro 901

Tabulka 4.5: Seznam komponent testovací PC sestavy

délku kontextu zvolit až do délky 16 384 tokenů. Pokud však tato délka bude větší, jak podporovaný kontext samotným modelem, délka kontextu, který model má v paměti, bude dán hodnotou z modelu. Například modely vycházející z architektury *Llama-2* disponují kontextem 4096 tokenů, jak bylo diskutováno v sekci 2.1.3. Pokud tedy runtime bude nastaven s větším kontextovým oknem, bude rozhodující právě tato hodnota modelu.

Pro spuštěnou instanci modelu lze také nastavit *systémový prompt*. Jedná se o první zprávu, která se modelu předává a měla by obsahovat pokyny pro model definující jeho chování, roli a další relevatní informace, které mohou zlepšit přesnost jeho výsledků nebo mu umožnit lépe pochopit kontext. Tento prompt je běžně vložen pouze na začátku konerzace a poté uložen do paměti modelu. Ukázku takového systémového promptu lze vidět ve výpisu 4.6. Tento konkrétní příklad pochází právě ze softwaru *LM Studio*.

Zdrojový kód 4.6: Příklad systémového promptu

```
1 You're an intelligent programming assistant. Output code only
    , no text around.
2
```

4.5.1.2 Testovací sestava a nastavení

Pro spouštění lokálních modelů byla využita PC sestava, jejíž komponenty jsou popsané v tabulce 4.5. Runtime programu *LM Studio*, popsaného v minulé sekci, byl nastaven, aby využíval 12 z 16 dostupných CPU vláken. GPU akcelerace fungovala v režimu *OpenCL*. Počet GPU vrstev se však liší model od modelu. Pro správnou funkčnost je potřeba udržet celý model v paměti RAM, tedy v případě naší sestavy jsme mohli pracovat s modely pouze cca do 60GB, s tím, že do VRAM grafické karty lze nahrát pouze vrstvy o celkové velikosti přibližně 20GB. Na GPU tedy byl akcele-

rován jen takový počet vrtev, který z jejich celkového počtu odpovídal této velikosti. Délka *kontextového okna* byla nastavena na 8 192 tokenů, avšak pro některé modely je tato délka zkrácena samotným modelem (viz sekce 4.5.1.1). Jednotlivé hyperparametry byly poté nastaveny jako:

1. **teplota**: 0.7
2. **Top P**: 1
3. **Maximum vygenerovaných tokenů**: 3000
4. **režim kontextového okna**: klouzavé okno

Zbytek *hyperparametrů* zůstal na defaultních hodnotách daného modelu, který byl testovaný. Toto nastavení bylo využito jak pro *lokální* modely tak pro *API* dotazy na poskytovatele proprietáních či jiných vzdáleně spuštěných modelů. Použité hodnoty těchto parametrů vycházejí z prvotních testovacích experimentů, kdy se osvědčili jako vhodné pro generování jednotkových testů, protože by měli zaručovat *nižší determiničnost* výsledku a jeho *vyšší přesnost*.

4.5.2 Dotazy

Samotné vygenerování testů za pomoci dotazování LLM je řešeno skrze *generující* režim programu, jako je ukázáno ve výpisu 4.2. Tento režim jako argument vyžaduje *název šablony*, která funguje jako *prompt* pro model a byla vytvořena v předchozím kroku (viz sekce 4.4.1). Tento režim poporuje i více nepovinných argumentů jako:

- `count` - Počet variací testů, které se mají vygenerovat. Celé číslo. Defaultní hodnota 1.
- `manual` - Zkopírovat prompt do schránky pro manuální vložení do rozhraní modelu. Vlajka. Vhodné pro debug.
- `cmd` - Vypsat prompt do standardního výstupu. Vlajka.
- `compress` - Vlajka vyjadřující použití promptové komprese.

Mimo argumentů je *generování testů* také konfigurováno *proměnnými prostředí*, které pokud nejsou definovány, program se snaží načíst soubor `.env` (pokud je přítomný), který tyto proměnné obsahuje. Konfigurace generování za pomocí těchto

proměnných byla zvolena, protože se většinou nepřidávají do VCS nebo je jejich přidání ošetřeno, protože obsahují citlivá data, která by měla zůstat pouze na stroji uživatele. Mezi základní proměnné prostřední patří:

- `API_URL` - Adresa URL, na kterou program dělá API dotazy.
- `API_KEY` - Klíč pro přístup ke vzdálenému API (pro lokální modely stačí zadat libovolnou hodnotu nebo nedefinovat).
- `API_MODEL` - Název modelu, na kterém programu bude v rámci API požadovat vygenerování výstupu (hodnota dána dokumentací daného API).
- `MAX_TOKENS` - Maximální počet tokenů, které má model vygenerovat. V závislosti na dokumentaci API je potřeba nastavit hodnotu v určiném rozsahu.

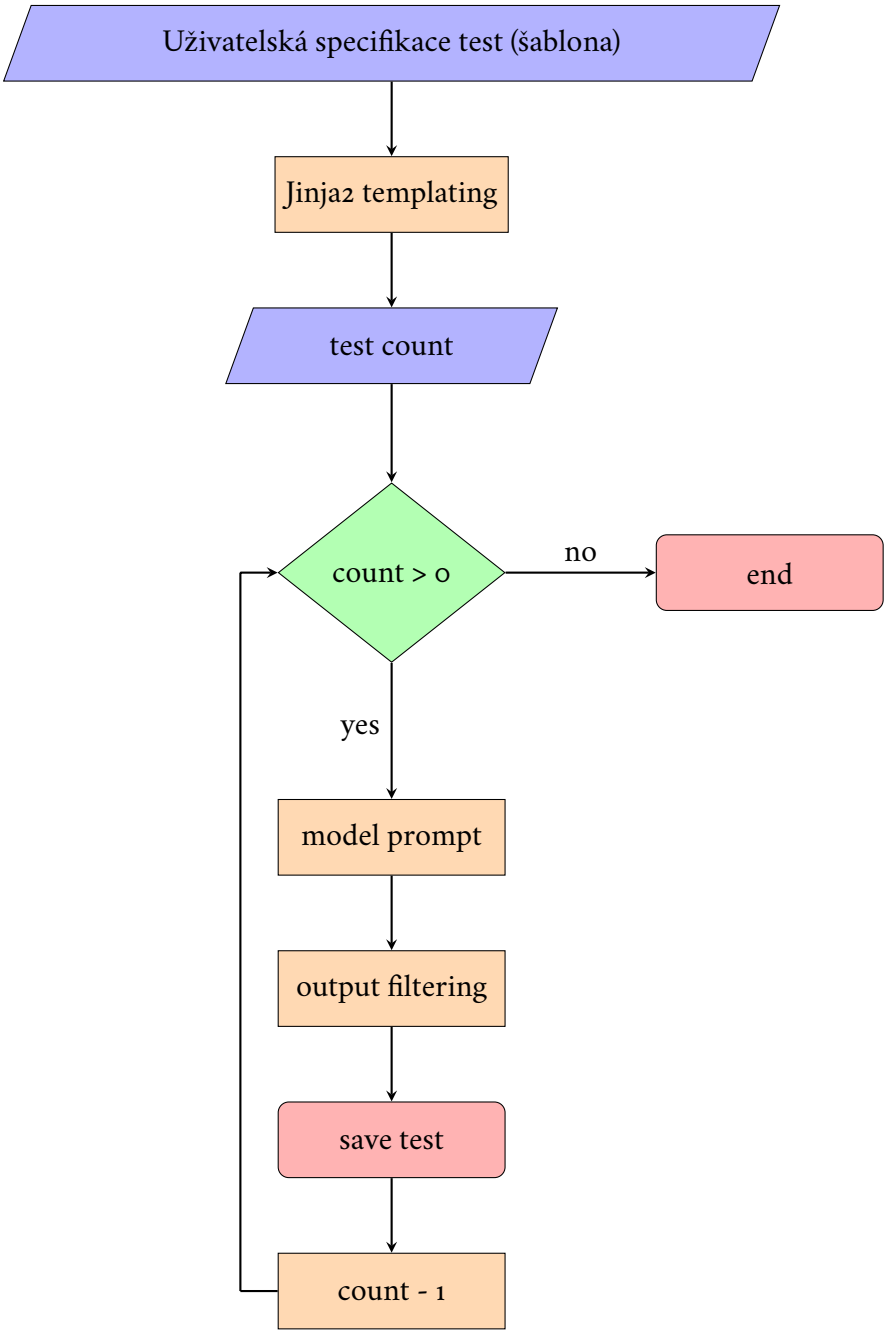
Mimo parametrů pro generování může `.env` soubor obsahovat i další parametry (resp. proměnné) určené pro jiné operace programu. Ukázková podoba tohoto souboru je zobrazena ve výpisu 4.9. Soubor je umístěn v kořenové složce programu.

Ukázkové volání programu v režimu generace testů lze vidět v ukázce 4.7. Program je schopen vygenerovat pro jeden test více variant na bázi stejného promptu (vytváření nových dotazů na LLM). Detailní popis algoritmu je nastíněn v obrázku 4.2. *Šablona*, kterou uživatel vytvořil je použita pro *render* finálního promptu za pomocí šablonovacího jazyka *Jinja*. Tato metoda byla zvolena z důvodu *jednoduché a předhledné modifikace* vstupů (zde nazvaých jako *šablon* nebo *uživatelských specifikací*), *možnosti nezávislého vložení nahrávky* a také především případné *parametrizace* vstupu, díky které by mohla jít měnit *uživatelská jména*, *názvy prvků* a další možné údaje v šabloně vhodné pro parametrizaci.

Výpis 4.7: Ukázka volání generace testu dle šablony

```
milan:dp\program$ python3 test.py -i spec-4 --count 10
```

Finální podoba promptu je poté skrze *konektor* pro příslušné LLM (například zmíněná *OpenAI* knihovna, *HTTP* dotaz či jiný druh konektoru) předána jako dotaz modelu. Společně s ním je i modelům předán v rámci těchto konektorů i *systémový prompt*. V případě, že se je aktivní vstupní příznak programu `--manual`, systémový prompt se přidá před vyrenderovaný prompt a tento spojený text je zkopírován do *schránky*. Program načítá systémový prompt ze souboru `templates/system.txt`. Jeho znění (viz výpis 4.8) vychází poažadavků a nedostatků, které byli při testovacím generování zpozorovány.



Obrázek 4.2: Zjednodušené schema algoritmu pro dotazování jazykového modelu při generování testu.

Zdrojový kód 4.8: Použitý systémový prompt

```
You're an intelligent programming assistant. You write Robot Framework scenarios and scripts using the Selenium Library. Insert delays between steps. Output code only, no text around. Use Chrome as browser. Locate elements using XPath. Close browser between scenarios.

```

Zdrojový kód 4.9: Ukázka ".env"souboru

```
API_URL="https://api.mistral.ai/v1"
API_KEY="Zca6nB87qmijn4"
API_MODEL="mistral-large-latest"

MAX_TOKENS=3000
DEVICE="mps"

```

I přes důraz na nutnost pouze *programového* výstupu v rámci systémového promptu, mají modely tendeci tuto podmínku ignorovat a generovat text okolo kódu. Protože jsou modely naučené tak, aby jejich výstup byl text v **Markdown** formátu, obsahují i značky se začátkem kódového bloku a značením jazyka, ale také prvky jako odrážky, apod. Tyto prvky se pak objevují i ve výstupech, kde je požadován pouze *čistý text* (jako například v ukázce 4.10). V našem případě je toto chování nevhodné, protože vyžadujeme na výstupu pouze *kód testu*. Díky struktuře *Markdown* formátování však lze lze kód z tohoto výstupu jednoduše vyparsovat. Toto parsování má právě na starosti funkce *filtrování výstupu* ve schematu 4.2. V rámci programu byla implementována tak, že ze vstupního textu vyjme kód v prvním nalezeném kódovém bloku, označeného pomocí znaků ‘‘‘. Pokud blok nenajde, vrátí původní text bez změny (předpokládá, že model se držel systémového promptu). V případě více kódových bloků přítomných ve výstupu, kdy požadovaný kód nebude v prvním z bloků, dojde k nesprávnému parsování, avšak jedná se o očekávaný případ, protože detekce správného kódu je netriviální a zároveň *subjektivní* problematika.

Zdrojový kód 4.10: Výstup modelu s nadbytečným textem.

```
Here's a Robot Framework scenario that follows the recorded JSON steps for interacting with the web page, and then performs the checks as specified:

‘‘‘robotframework
*** Settings ***
Library           SeleniumLibrary

*** Variables ***
```

```
8  ${URL}              http://localhost:4680/tbuis/index.jsp
9  ${USERNAME}         lazy
10 ${PASSWORD}         pass
11 ${BROWSER}          Chrome

13 *** Test Cases ***
14 Open and Verify Webpage Content
15 ...
16 '''

18 In this Robot Framework script, I've incorporated commands
       that:
19 - Open the specified page and set the viewport size.
20 - Perform the login process using the provided username and
       password.
21 - Navigate to the "List of All Teachers" section.
22 - Check for the presence of specified names on the page.
23 - Validate the text content of specific table cells related
       to the teachers' courses.
24 Adjust the locators (xpath) as necessary to match the exact
       structure and identifiers used in your application's HTML.
25
```

Vyfiltrovaný výstupní text modelu je poté zapsán do souboru ve výstupní složce nazvané `generated`, nacházející se v kořenové složce programu. Test je uložen s názvem ve formátu `{nazev-vstupu}-{cislo-varianty}.robot` jakožto *RobotFramework* soubor. První část tohoto názvu souhlasí s názvem *promptové šablony*, pro kterou uživatel vytváří testy. Druhá část poté čísluje vygenerované varianty. Pokud již ve výstupní složce existuje vygenerovaný test pro danou specifikaci, program nové variantě přiřadí číslo o 1 vyšší, tedy pokračuje v číselné řadě. Názvy vygenerovaných testů jsou postupně vypisovány do konzole. Od uživatele je očekáváno, že vygenerované testy ve výstupní složce zkontroluje a ověří, že obsahují validní (*např. obsahuje text, připomínající RobotFramework test*). Je zde totiž stále možnost, že model nevygenerovat žádný kód a vypsat pouze text nebo *filtrace* neproběhla validně. Tento krok by v ideálním případě bylo možné automatizovat za pomocí *analyzátoru* pro *RobotFramework*, který se nachází v rámci LSP pro *RobotFramework* `rebotframework-lsp` nachází. Bohužel však neumí validovat zápis pro knihovnu `SeleniumLibrary`, která je v rámci systémového promptu vyžadována a očekávána, že bude přítomná ve vygenerovaném testu.

Spuštění testů 5

5.1 Spuštění testovaného programu a jeho orchestrace

Testovaný program *TbUIS* (popsaný v sekci 2.2) je distribuován jakožto `.WAR` soubory, tedy komprimovaný webový archiv jazyka Java, spustitelný jakožto servlet. Z dokumentace projektu a předchozích diplomových prací vyplývá, že pro nasatení je určený aplikační server *Tomcat* verze 7 až 9. S vyššími verzemi není aplikace kompatibilní. Každá varianta aplikace (*poruchové klony*) je distribuována jako samostatný *WAR* soubor. Pro spuštění konkrétní varinaty a spuštění vygenerovaných testů na této variantě bylo nutné vytvořit *automatizované nasazení* aplikace *TbUIS*. Pro nasazení je nutné brát v úvahu nejen samotnou webovou aplikaci, ale i její závislost, kterou je *databáze* MySQL (resp. MariaDB), která musí být inicializována dodaným skriptem.

5.1.1 Vytvoření kompozice

Pro účely *automatizovaného nasazení* byla zvolena technologie *Docker*, tedy *kontejnerové* řešení s možností vytvářet obrazy a kompozice. Právě kompozice je vhodným nástrojem pro jednoduché *lokální* nasazení aplikace *TbUIS*, protože potřebujeme 2 samostatné kontejnery; první s *aplikačním servervem* a druhý s *databází*.

Před samotnou integrací je nutné v každé z variant aplikace (*WAR* soubory) změnit *adresu* databáze. Docker kompozice umoňují přidělit jednotlivým kontejnerům názvy, za pomocí kterých lze v rámci interní DNS tyto kontejnery adresovat. V rámci archivů bylo nutné modifikovat adresu v souborech *WEB-INF/classes/META-INF/persistence.xml* a *WEB-INF/classes/applicationContext.xml*. Pro zjednodušení

vlastního nasazení jsou modifikované webové archivy součástí repozitáře této práce. Pro kontejner *databáze* byl zvolen název `tbuis-db`. Veškeré potřebné soubory pro spuštění testovaného programu se nachází ve složce `tbuis` v rámci programové složky.

K vytvoření *kompozice* je potřeba mít také obrazy, ze kterých se budou vytvářet kontejnery. Zatímco pro databázi stačí *MariaDB* kontejner z veřejného repozitáře, tak pro aplikační server je potřeba vytořit vlastní obraz, který bude vycházet z obrazu pro *Tomcat* aplikační server, ale vždy do něj budou nahrána data jiné varianty aplikace. Pro vytoření obrazu byl do složky přidán *Dockerfile* sloužící k jeho sestavení. Obsah tohoto souboru lze vidět v ukázce 5.1. Skript přebírá cestu k *WAR* souboru za pomocí argumentu a ten poté vkládá do obrazu na předem definovanou cestu a s definovaným názvem. V rámci Tomcat kontejneru slouží právě složka `webapps` pro servlety a jejich název udává cestu, které v rámci HTTP dotazů jsou dostupné. Pro zvolený název `tbuis` tedy odpovídá cesta `/tbuis`.

Zdrojový kód 5.1: Dockerfile pro sestavení obrazu varianty aplikačního serveru.

```
1 FROM tomcat:9.0
2 ARG WAR_FILE
3 COPY ${WAR_FILE} /usr/local/tomcat/webapps/tbuis.war
4
```

Obraz z *Dockerfile* lze vytvořit samostatně, ale také lze jeho sestavení zavolat při vytváření kompozice. To se provádí za pomocí nástroje *Docker Compose*. Defaultní název souboru, který příkaz pro vytváření kompozice `docker-compose up` využívá je `docker-compose.yml`. Takovýto defaultní soubor je vytvořen v podsložce programu `tbuis` (jeho obsah se nachází v ukázce 5.2). Jak přípony souboru vyplývá, jedná se o textový soubor ve formátu *YAML*. V rámci něj definujeme dvojici služeb potřebných pro nasazení aplikace. Službou v kompozici je myšlen kontejner. Při vytváření služby lze definovat i její *proměnné prostředí*, *porty* a jejich předávání, *připojené* složky, *sítě* a další vlastnosti běžně nastavované pro kontejner. Pro *databázi* je potřeba předat port 3306, ale také připojit složku s inicializačními skripty (v našem případě dostupná ve stejné složce jako kompoziční soubor). V ní se nachází soubor `init-db.sql`, který obsahuje příkazy pro vytvoření databáze s daným názvem a uživatele, za kterého se aplikační server připojuje. U služby aplikačního serveru je potřeba sestavit obraz. Pro toto sestavení využíváme vytvořený Dockerfile, nacházející se ve stejné složce. Také je potřeba předat argument s cestou pro *WAR* soubor, který bude do serveru nahraný. Pro přístup do webového rozhraní aplikace z venčí kompozice byl zvolen port 4680 předaný z defaultního portu 8080, který aplikace používá. Tento port bude sloužit vygenerovaným testům pro přístup. V případě potřeby může uživatel tento port změnit dle svých požadavků. Oba kontejnery jsou

také připojeny do stejné *virtuální sítě*, který zajišťuje komunikaci mezi nimi.

Zdrojový kód 5.2: Docker Compose soubor pro sestavení kompozice

```
version: '3.8'

services:

  db:
    image: mariadb
    container_name: tbuis-db
    environment:
      MARIADB_ROOT_PASSWORD: testtest
    ports:
      - "3306:3306"
    volumes:
      - ./db:/docker-entrypoint-initdb.d/
    networks:
      - tbuis

  tomcat:
    build:
      context: .
      args:
        WAR_FILE: ${WAR_FILE_PATH}
    container_name: tbuis-tomcat
    depends_on:
      - db
    ports:
      - "4680:8080"
    networks:
      - tbuis

networks:
  tbuis:
    driver: bridge

```

Pro vytvoření a spuštění této kompozice je potřeba zavolat příkaz z ukázky 5.3, kde příkaz `up` vyjadřuje právě *vytvoření* kompozice. Pro povolení sestavení obrazu v rámci vytváření kompozice, je také potřeba přidat argument `--build`. Aby se kontejnery spustili na pozadí a ne v terminálu, je také vyžadován argument `-d`. Zároveň z důvodu, abychom nástroj `docker-compose` nemuseli volat pouze ze složky, ve které se kompoziční soubor nachází, lze také přidat argument `-f` a hodnotou cesty ke konfiguračnímu souboru. Před příkazem je také vyexportována proměnná prostředí `WAR_FILE_PATH`, která určuje variantu serverletu, který bude nasazen do webové aplikace. Jedná se o cestu relativní vůči *docker compose* souboru. V případě

nutnosti *smazat* kompozici, je také možnost zavolat příkaz `docker-compose down` (viz ukázka 5.4). Protože při vytváření kompozic byli sestaveny i obrazy, které mohou na počítači zbírat nemalé místo, je také k příkazu přidán argument `--rmi all`, který mimo kotejnerů vytvořených kompozicí smaže i nevyužívané obrazy.

Výpis 5.3: Vytvoření Docker kompozice z připravené konfigurace

```
milan:dp\program$ WAR_FILE_PATH=./defect-00_free.war
    docker-compose -f tbuis/docker-compose.yml up -d --build

```

Výpis 5.4: Smazání Docker kompozice společně s vytvořenými obrazy

```
milan:dp\program$ docker-compose -f tbuis/docker-compose.yml
    down --rmi all

```

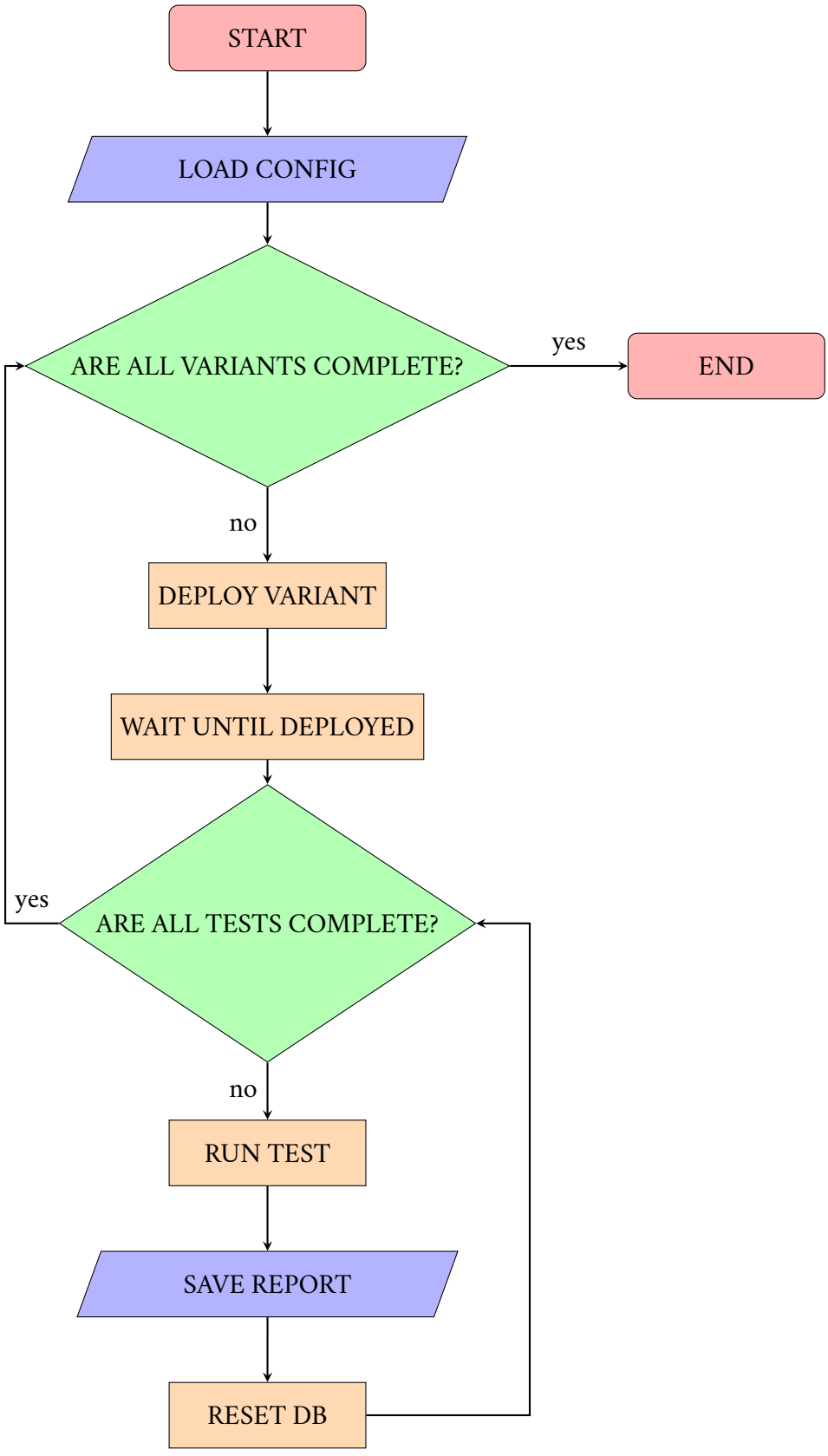
5.1.2 Orchestrace a automatizace nasazení

Výše vypsané příkazy automatizují nasazení jedné z variant webové aplikace. Namísto jejich manuálního volání však bude tento úkon provádět orchestrační program `test.py` v režimu `-r` (*run*). Příklad takového volání se nachází v ukázce 5.5. Samotná orchestrace spuštění vygenerovaných testů spočívá v tom, že program nasadí jednu z požadovaných variant aplikace, poté postupně spustí jednotlivé testy vyhovující zadanému kritériu v rámci hodnoty pro argument režimu `-r`. Po dokončení všech testů je načtena další z požadovaných verzí aplikace pro nasazení a otestování. Zvolené testy jsou tedy spuštěny na všech vybraných variantách (orchestrace nastíněna v obr. 5.1). Po dokončení testování je poté vytvořená kompozice společně se všemi jejími sestavenými a staženými obrazy smazána. K těmto úkonům je využit právě nástroj *Docker* a *Docker Compose*, popsané v sekci 5.1.1. Mezi jednotlivými testy spouštěnými na jedné z variant aplikace je také potřebna provést *obnovení databáze*, protože některé z testovacích scénářů databázi modifikují (viz tabulky 4.2 a 4.2). Pro toto obnovení nabízí aplikace *HTTP* endpoint nebo tlačítko na úvodní stránce. Pro zjednodušení byl mezi statické soubory přidán RobotFramework scénář, který na toto tlačítko na úvodní stránce klikne.

Výpis 5.5: Spuštění orchestračního programu v režimu spouštění testů

```
milan:dp\program$ python3 test.py -r "codellama/spec-*" --name
    "codellama-runs"

```



Obrázek 5.1: Znázornění orchestrace spouštění testů

Spuštící režim programu nabízí možnost hned několika argumentů a jejich hodnot. Požadovaná hodnota je zde pro *režimový* argument `-r`, která musí obsahovat název nebo vzor názvu testů, které se v rámci složky *generated* mají spustit. Pro přehlednost si uživatel může ve výstupní složce testů rozdělovat vygenerované testy do podsložek, případně držet vhodnou konvenci pro jejich označení. Hodnota pro vzor názvu testů může být jakýkoliv validní *UNIX wildcard* pro označení souborů. Příklad použití takovéto *wildcard* lze vidět v ukázce 5.5, kde se spouští testy (resp. soubory) z podsložky `codellama`, které vyhovují vzoru `spec-*`, což znamená všechny soubory začínající řetězcem "spec-"v této podsložce. Tento přístup pro zvolení cesty a souborů umožňuje nejen právě dělit výstupní složku testů na podsložky, ale také uživateli spustit jen konkrétní vybrané testy. Spuštění pouze omezené množiny testů je ukázáno ve výpisu 5.6, kde uživatel požaduje spustit testy vyhovující specifikaci 1, 4 a 6 ve všech jejich varintách (značení vygenerovaných testů popsáno v sekci 4.5.2) ve složce "openai-gpt-4"(tedy například testy vygenerované modelem GPT-4 od OpenAI). Hodnoty pro tento argument je vhodné psát v *uvozovkách*.

Výpis 5.6: Alternativní volání režimu spuštění testů

```
milan:dp\program$ python3 test.py -r
    "openai-gpt-4/spec-{1,4,6}-*" --cont_count 2 --name
    "gpt-4 test runs"

```

Další argumenty pro *spouštěcí* řežim jsou již *nepovinné*, avšak alespoň argument `--name` je doporučený. Pomocí toho argumentu lze nastavit jméno aktuálního běhu testů, pomocí kterého lze poté jednoduše identifikovat s ním související výsledky. Samotné výsledky, jejich ukládání a struktura jsou vysvětleny v následující sekci kapitoly. Pro hodnotu tohoto argumentu je také doporučené využít uvozovky (ukázkové volání jak v příkladu 5.5 a 5.6). Pro zkušební běhy testů lze také využít argument `--cont_count`, jehož hodnota omezí počet volaných variant aplikace, dostupných z *konfigurace* (číselná hodnota vyšší než 0).

Samotný běh aplikace, testů a jejich případného vyhodnocení neovlivňují pouze *argumenty*, ale také komplexnější *konfigurace*, která se nachází v rámci konfiguračního souboru `configuration.py`. Jednotlivé varinaty aplikace, které se budou spouštět v rámci běhu lze zvolit tak, že se názvy jejich *WAR* souborů ze složky `tbuis` přidají do pole `RUN_CONTAINERS` v rámci konfigurace.

Zdrojový kód 5.7: Zjednodušená ukázka konfiguračního souboru

```
RUN_CONTAINERS = [
    "defect-00_free.war",
    "defect-02-C0.H0.M0.L1_S_S_03.war",
    "defect-04-C0.H0.M0.L1_S_S_04.war",
```

```
    ...
    "defect-26-C1.H0.M0.L0_T_S_01.war",
    "defect-27-C1.H0.M0.L0_U_D_01.war",
    "defect-28-C2.H2.M1.L0_M_CR.war"
]

POSITIVE_FAILS = {
    "1": ["defect-02-C0.H0.M0.L1_S_S_03.war"],
    "4": ["defect-04-C0.H0.M0.L1_S_S_04.war", "defect-19-C0.H1.M0.L0_S_S_10.war", "defect-25-C1.H0.M0.L0_S_S_01.war", "defect-26-C1.H0.M0.L0_T_S_01.war", "defect-28-C2.H2.M1.L0_M_CR.war"],
    "6": ["defect-25-C1.H0.M0.L0_S_S_01.war", "defect-26-C1.H0.M0.L0_T_S_01.war", "defect-28-C2.H2.M1.L0_M_CR.war"],
    ...
}

```

5.2 Ukládání výsledků

Při spuštění testu vyváří *RobotFramework* hned několik výstupních souborů, mezi kterými je i *report* jakožto soubor `output.xml`. V rámci něj se nacházejí veškeré klíčové informace o běhu scénáře. Konkrétně jde o *log* provedených úkonů, *výsledky* jednotlivých testovacích případů a také případné *errory*. Protože tento výstup obsahuje spoustu redundantních informací, je v rámci programu přeparsován a uložen na 2 samostatná místa. Z původního výstupu jsou parsovány hodnoty: *počet úspěšných*, *neúspěšných* a *chybových* testovacích případů pro každý ze spuštěných testovacích scénářů. Pro ten je vždy také vyhodnocen koncový stav (*FAIL, PASS* nebo *ERROR*). Protože *RobotFramework* nerozlišuje mezi testem, který *selhal* a testem, který neměl spustit nebo při jeho běhu došlo k chybě nesouvisející se scénářem, bylo potřena toto rozlišení implementovat na bázi přítomných *errorů* v rámci výstupního souboru.

Pro označení výsledku testu (dále také jako *report*) je využito *časové razítko* doby spuštění série testů ve formátu `YYYYMMDDhhmmss` společně s vybraným názvem v rámci argumentu (popsáno v předchozí sekci 5.1.2) oddělených znakem "-". V případě, že samostatný název běhu testů není uživatelem vybrán, zůstává pro plné označení reportu pouze časové razítko.

Prvním z míst pro uložení výsledků je textový soubor uložený ve podsložce `reports` v rámci kořenové složky programu. V rámci tohoto souboru jsou vykresleny textové tabulky, vykreslené pro každou nasazenou variantu aplikace (varianty

```
Report name: 20240411140043-GPT-4-Turbo run


Container name: defect-00_free.war
+-----------------------------------+----------+----------+----------+----------+
| Name                              | Status   |   Passed |   Failed |   Errored |
+===================================+==========+==========+==========+==========+
| openai-gpt45-turbo/spec-12-10.robot | FAIL     |        1 |        1 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-1-8.robot   | FAIL     |        0 |        3 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-11-7.robot  | PASS     |        1 |        0 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-1-3.robot   | PASS     |        3 |        0 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-18-4.robot  | PASS     |        1 |        0 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-12-1.robot  | FAIL     |        1 |        1 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-10-7.robot  | FAIL     |        0 |        2 |         0 |
+-----------------------------------+----------+----------+----------+----------+
| openai-gpt45-turbo/spec-12-4.robot  | FAIL     |        0 |        2 |         0 |
+-----------------------------------+----------+----------+----------+----------+
```

Obrázek 5.2: Ukázka textové tabulky výsledků

Obrázek 5.3: Ukázka databázové tabulky výsledků

jsou v rámci programu také označovány jako kontejnery, protože odpovídají kontejneru Dockeru). V rámci tabulky jsou pro každý spuštěný test vypsány výsledky jednotlivých sledovaných údajů a koncový výsledek testu. Ukázka tabulky je vyzobrazena na obr. 5.2. Název tabulky odpovídá právě testovanému kontejneru (resp. varianty testu). Soubor report je pojmenovaný jeho celým názvem, jak bylo popsáno v předchozím odstavci.

Ačkoliv tabulková reprezentace výsledků v textové podobě je jednoduchá pro uživatele a jeho čtení, tak již není vhodná pro strojové čtení a zpracování, proto jako druhá možnost pro uložení výsledů byla zvolena *SQLite* databáze, resp. soubor `report_db.sqlite` umístěný ve stejné podložce `reports` jako textové reporty. Tento druh formátu byl zvolen z důvodu jeho jednoduchosti a široké podpoře napříč programovacími jazyky. V rámci této databáze je vytvořena tabulka "runs", která obsahuje řádky s hodnotami výsledků jednotlivých testovacích scénářů, podobně jako v případě tabulek textového reportu. Narozdíl od nich se zde pro veškeré údaje využívá jedna tabulka a mezi jednotlivými běhy jde rozlišit primárně dle jejich názvu. Ukázka tabulky je vyzobrazena na obr. 5.3. V rámci malé lokální databáze je poté možné data efektivně vyhledávat, filtrovat a nadále s nimi pracovat.

Vyhodnocení výsledků

6

6.1 Parametry pro spuštění

Vygenerované testy z kroku 4 (konkrétně scénáře vypsané v tabulkách 4.2 a 4.2) byly v rámci výstupní složky `generated` rozděleny do podsložek s odpovídajícím názvem, ze kterého je patrné, jakým modelem (případně od jaké společnosti) byly vygenerovány. Možnost takového rozdělení byla popsán v sekci 5.1.2. Tyto vygenerové testy, využité k následnému testování poruchových klonů, jsou součástí programového repozitáře projektu. Do konfiguračního souboru byli přidány varianty aplikace dle tabulky 4.2 (ukázka konfiguračního souboru ve výpisu 5.7). Z tabulky scénářů také víme, pro které varianty testovacího programu také můžeme očekávat chybu v daném scénáři. Proto je do konfiguračního souboru přidáno i *mapování*, které každému scénáři (označené pouze číslem) přiřazuje pole názvů *WAR* souborů aplikace, ve kterých je chybovost očekávána. Na bázi tohoto mapování dále vyhodnocovací program určí správnost výsledku testu (tj. v případě očekávaného selhávání jde o správný výsledek).Tento program a použité metody jsou popsány v následující sekci.

6.2 Metodologie vyhodnocení

6.3 Výsledky pro jednotlivé modely

6.3.1 OpenAI

6.3.2 Anthropic

6.4 Cena a časová náročnost generování

Budoucí vylepšení 7

Závěr 8

Seznam obrázků

Seznam tabulek

Seznam výpisů