



FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY



Diplomová práce

Generování jednotkových testů s využitím LLM

Milan Horínek





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.

© Milan Horínek, 2024.

Všechna práva vyhrazena. Žádná část tohoto dokumentu nesmí být reprodukována ani rozšiřována jakoukoli formou, elektronicky či mechanicky, fotokopírováním, nahráváním nebo jiným způsobem, nebo uložena v systému pro ukládání a vyhledávání informací bez písemného souhlasu držitelů autorských práv.

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.

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 existující literaturu ohledně generování 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

Tato diplomová práce zkoumá použití velkých jazykových modelů (LLM) pro automatizované generování softwarových testů, konkrétně GUI webových aplikací. Zvolený přístup zkoumá, zda vygenerované testy jsou nejen sémanticky smyslupné, aby odpovídali funkčním požadavkům softwaru, ale také zda jsou schopné odhalit i vložené chyby do softwaru. Výzkum zahrnuje návrh a vývoj automatizovaného nástroje, který využívá zpracování přirozeného jazyka k interpretaci popisů testů a generování odpovídajících testových skriptů. Tento nástroj byl extenzivně vyhodnocen na vzorové webové aplikaci za účelem posouzení přesnosti, úplnosti a spolehlivosti jednotlivých výstupních testů. Výsledky naznačují, že nástroje založené na LLM dokáží efektivně generovat softwarové testy a úspěšně identifikovat vložené chyby do aplikace.

Abstract

This thesis explores the use of large language models (LLMs) for automated generation of software tests, specifically GUI web applications. The chosen approach investigates whether the generated tests are not only semantically meaningful to match the functional requirements of the software, but also whether they are able to detect embedded bugs in the software. The research involves the design and development of an automated tool that uses natural language processing to interpret test descriptions and generate corresponding test scripts. This tool has been extensively evaluated on a benchmark web application to assess the accuracy, completeness and reliability of each test output. The results indicate that LLM-based tools can efficiently generate software tests and successfully identify bugs embedded in the application.

Klíčová slova

LLM • testing • unit • Robot Framework

Poděkování

Rád bych vyjádřil své hluboké poděkování vedoucímu mé diplomové práce, Ing. Richardu Lipkovi, Ph.D., za jeho neocenitelnou podporu, cenné rady a doporučení během celého procesu přípravy této práce. Velký dík patří také za čas, který mi byl ochotně věnován. Mé upřímné díky patří rovněž všem zaměstnancům Fakulty aplikovaných věd Západočeské univerzity v Plzni, kteří mi byli nápomocni během řešení problémů spojených s diplomovou prací a jejichž přednášky byly pro mne velkým přínosem a inspirací.

Zkratky

CNN Convolutional Neural Network - Konvoluční neuronová síť. 13

GGML GPT-Generated Model Language. 35

GGUF GPT-Generated Unified Format. 34, 35

GUI Graphical User Interface. 6, 21, 77, 78, 89

IR Information Retrieval - Získávání informací. 17

ML Machine Learning - strojové učení. 7, 35

NLP Natural Language Processing - Zpracování přirozeného jazyka. 7

PUT Parameterized Unit Test. 11

RNN Recurrent Neural Network - Rekurenční neuronová síť. 13

TDD Test Driven Development - Vývoj řízený testy. 3

VCS Version Control System. 38

Slovník pojmů

assert Příkaz používaný k ověření, zda je výsledek testu v souladu s očekávanými hodnotami, což umožňuje automatizovaně identifikovat chyby v testovaném kódu.. 88

benchmark Standardizovaný test nebo pokus určený k měření a srovnání výkonu systémů, komponent či aplikací.. 23

filtr Ve vývoji softwaru a testování se odkazuje na mechanismy nebo kritéria používaná k selektivnímu výběru testovacích vstupů nebo k rozhodování, které výstupy testů jsou relevantní pro další analýzu. Filtry mohou být použity k odstranění redundantních, nelegálních nebo jinak nežádoucích vstupů z procesu generování testů, což zvyšuje efektivitu testování tím, že se zaměřuje pouze na vstupy, které mohou odhalit chyby nebo porušení kontraktů. 12

kontext Rozsah textu nebo dat, který může model zpracovat při generování odpovědi. Funguje jako krátkodobá paměť modelu a určuje, kolik informací může model vzít v úvahu při tvorbě odpovědi.. 35, 36

kontrakt V kontextu jednotkových testů odkazuje na formálně definované podmínky nebo pravidla, které musí být dodrženy během vykonávání kódu. Kontrakty mohou specifikovat, co funkce očekává od svých vstupů a jaké výstupy nebo stavy by měla produkce kódu způsobit. Použití kontraktů pomáhá zajistit, že kód se chová podle očekávání a usnadňuje identifikaci chyb, když tyto podmínky nejsou splněny. 12

LLM Velký jazykový model (LLM - large language model) je typ modelu strojového učení, který je navržen tak, aby rozuměl a generoval text v přirozeném jazyce. Tyto modely jsou trénovány na obrovských datasetech a jsou schopny provádět různé úkoly, jako je textová klasifikace, generování textu, strojový překlad a další. 2, 7–9, 13–16, 23, 25, 34, 35, 38, 39, 79–82, 84, 88, 89

LSP Language Server Protocol. Protokol pro jazykový server používaný pro validaci syntaxe, sémantiky nebo formátování zdrojového kódu. Často se pod tímto označením myslí i samotný jazykový sever, který se o tuto analýzu stará..

42

mock objekt V kontextu testování jde o simulované objekty, které imituje chování reálných komponent v rámci systému a slouží k izolaci testované jednotky od jejích závislostí.. 4

RAG Retrieval-Augmented Generation (RAG) je metoda pro vylepšení generativních modelů získáním relevantních informací z velké datové sady a jejím využitím v kontextu pro generování přesnějších a relevantnějších výsledků..

17, 18

servlet Serverový programový modul napsaný v programovacím jazyce Java, který zpracovává požadavky klientů a implementuje rozhraní Servlet. Přestože servlety mohou reagovat na jakýkoli typ požadavku, nejčastěji se používají k rozšíření aplikací hostovaných webovými servery.. 43

teplota V terminologii strojového učení se jedná o hyperparametr, který ovládá relativitu a náhodnost vstupu modelu. Vyšší teplota vede k méně předvídatelným a kreativnějším výsledkům, kdežto nižší teplota produkuje více konzervativnější výstup.. 35, 38

testové pachy Unit Test Smells jsou špatné návyky nebo postupy, které se mohou objevit při psaní jednotkových testů. Tyto "smelly" často vedou k neefektivním nebo nespolehlivým testům a mohou ztížit údržbu testovacího kódu. Příklady zahrnují Duplicated Asserts, Empty Tests a General Fixture. 15

token V NLP oblasti se jedná o rozdělení sekvence textu na menší části (například slova, části slov, atd.).. 7, 35–37

Obsah

1	Úvod	3
1.1	Testy	4
1.2	Jazykové modely	8
1.3	Motivace	9
2	Rešerše	11
2.1	Provedená práce v problematice	11
2.1.1	Předchozí automatizovaná řešení	11
2.1.2	Vydané publikace	15
2.1.3	Modely	16
2.2	Testovací program	22
3	Návrh	24
3.1	Výběr cíle a metody testů	24
3.2	Navrhované řešení	26
4	Generování testů	28
4.1	Prerekvizity pro generování a spuštění testů	28
4.2	Výběr scénářů	29
4.3	Nahrávání scénářů	29
4.4	Výběr požadavků	33
4.4.1	Vytvoření nového testu	33
4.4.2	Vyplnění požadavků	35
4.5	Dotazování LLM	35
4.5.1	Modely a jejich spuštění	35
4.5.2	Dotazy	39
5	Spuštění testů	44
5.1	Spuštění testovaného programu a jeho orchestrace	44
5.1.1	Vytvoření kompozice	44
5.1.2	Orchestrace a automatizace nasazení	47

5.2	Ukládání výsledků	50
6	Vyhodnocení výsledků	53
6.1	Parametry pro spuštění	53
6.2	Metodologie vyhodnocení	54
6.2.1	Vyhodnocovací program	54
6.2.2	Legenda tabulek	55
6.2.3	Předpoklady	56
6.3	Výsledky pro jednotlivé modely	56
6.3.1	OpenAI	56
6.3.2	Anthropic	61
6.3.3	Meta	65
6.3.4	Google	71
6.3.5	Mistral	74
6.4	Kvalita výsledků	78
6.4.1	Úspěšnost LLM modelů v generování testů	79
6.4.2	Srovnání s lidsky psanými testy	80
6.5	Cena a časová náročnost generování	82
7	Budoucí vylepšení	85
7.1	Vlastní formát nahrávky	85
7.2	Komprese promptu	87
7.3	Úpravy promptu dle modelu	87
7.4	Fine-tuning	87
8	Závěr	89
A	Přiložené zdrojové kódy	91
	Bibliografie	95
	Seznam obrázků	101
	Seznam tabulek	103
	Seznam výpisů	104

Velké jazykové modely (LLM) jsou fenoménem posledních doby a staly se rychle nástrojem každodenního použití. Přinesly nám nástroje jako například *ChatGPT*, *GitHub Copilot* nebo i generativní obrazové modely jako DALL-E [1]. Existuje více druhů LLM modelů, ale všechny mají v základu stejnou funkci, a to generovat text podobný tomu, který by napsal člověk, nebo úryvky zdrojového kódu či v některých případech i generovat obrázky, vše na bázi textové instrukce v přirozeném jazyce, které se říká *prompt*. Některé společnosti již tyto modely implementovaly i do svých interních procesů a zaznamenali díky tomu zvýšenou produktivitu u svých zaměstnanců, případně i zvýšenou kvalitu zdrojového kódu programátorům. [2] I přes jejich široké využití mají velké jazykové modely stále své nedostatky, které jsou později diskutovány. Např. jsou schopny generovat pouze text, který již někdy viděli, či dochází k jevu zvanému *halucinace*.

Účelem této práce je ověřit, zda je možné sestavit postup, kterým by šlo LLM modely využít k automatizovanému generování *softwarových testů* (ať už jednotlivých tak integračních, apod.). Konkrétně byla zvolena k otestování webová aplikace, a cílem je, aby i nezkušený tester mohl použít takovýto automatizovaný nástroj k vytvoření testových skriptů, které později budou využity při následném vývoji či jiné validaci a verifikaci softwaru. Zároveň je cílem otestovat různé modely a zhodnotit kvalitu jimi vygenerovaných testů. Celá práce je určena jako výzkum tohoto tématu a slouží případně jako možnost proveditelnosti pro navazující projekty, které mohou využít zde získaných informací jako *vhodný LLM model* či *postup tvorby testů* a také napravit případné nedostatky, které tato práce objeví, a to jak na výstupu modelů, tak v samotném postupu.

1.1 Testy

Softwarové testování je proces, při kterém dochází k validaci (případně i k verifikaci) aplikace nebo systému a ověření, zda splňuje kladené požadavky, implementované funkce podávají předpokládaný výstup, nebo vyhovuje očekávání. Jejich účelem je pomáhat vývojářům k zjištění případných defektů a chyb, jak při prvotním vývoji tak úpravách. V některých případech také testy mohou sloužit jako reprezentace požadavku na software (např. TDD). Obecně je implementace testů v softwarovém inženýrství považována za krok vedoucí k vyšší spolehlivosti, bezpečnosti (např. dle ISO/IEC 9126), nebo efektivitě vývoje.

Testování může být prováděno jak *manuálně* tak *skrze automatizované nástroje*. Manuální testování zahrnuje lidské testery, kteří následují dané testové scénáře a ověřují tak software. Takové testování je však časově náročné a mohou se v něm objevovat lidské chyby. Z tohoto důvodu je běžně využito spíše v rámci verifikačního procesu softwaru. Častěji jsou proto softwarové testy automatizované, k čemuž jsou využity konkrétní nástroje a frameworky, poskytující rámec pro vytvoření a spuštění testů k danému druhu softwaru. [3]

Testy jsou běžně stavěny dle *testového případu* (anglicky *test case*), který je *specifikací*, poskytující jednoduše srozumitelné shrnutí jednotlivých funkcí softwarové aplikace. Ten může vycházet například z funkčního požadavku nebo případu použití. Měl by obsahovat: *unikátní identifikátor*, *jasný název*, *podrobnosti*, *prerekvizity* a případně *reference* na další dokumenty. Dále by měl obsahovat jak standardní průběh testového scénáře tak i alternativní scénáře (například *selhání*) a jejich podmínky (zejména jejich *vstupy a výstupy*). Každý ze scénářů by měl být detailně popsán a přesně určeny jejich místa výskytu a příčiny. Neměl by být opomenut žádný z možných stavů a alternativních scénářů, které mohou nastat, protože by to vedlo k *nejasnosti* specifikace. Součástí testového případu by také měl být očekávaný stav systému před a po dokončení *standardního průběhu* a popis metody, kterou lze úspěšné splnění testu ověřit. [4] Při *automatizovaném* testování se také využívají pojmy *testovací skript* (Test Script) a *testovací sada* (Test Suite), kdy skript je přepis *testového případu* do zdrojového kódu a soubor takových kódů poté tvoří právě *testovací sadu*. [5]

Testy se v softwarovém vývoji rozdělují na různé druhy dle jejich specifického zaměření a rozsahu. Zde je potřeba podotknout, že toto rozdělení testů je sémantické a v reálné implementaci může jeden test mít přesah i do jiné kategorie nebo sám spadat do více z nich. Mezi nejčastější druhy se řadí:

- **jednotkové (unit) testy** - Kontroluje správnou funkčnost nejmenších částí (tzv. *jednotek*) softwaru. Těmi může být např. *funkce, třída, modul, komponenta*, ... Testování jednotek probíhá bez jejich závislostí, které jsou většinou nahrazeny Mock objekty
- **integrační testy** - Mock objekty jsou nahrazeny reálnými závislostmi a je testována integrace softwaru do jejího kompletního běhového prostředí.
- **akceptační testy** - Scénáře, dle kterých probíhá převzetí softwaru do rukou zákazníka, předem domluvených oběma stranami.
- **kvalifikační testy** - Rigorózní testování, při kterém je ověřeno, zda produkt odpovídá návrhu, standardům a požadavkům.
- **kouřové (smoke) testy** - Zjednodušené a nenáročné testy, které jsou spuštěny při každém sestavení softwaru.

Pro demonstraci testů lze ukázat jednotkové testování na příkladu *kalkulačky*, kterou při objektové implementaci v jazyce Java může představovat třída obsahující metody se základními operacemi (viz obr. 1.1). Ukázkové příklady pro jednoduché testové požadavky jsou uvedeny v tabulce 1.1. Podle této specifikace a s využitím např. frameworku *JUnit* lze sestavit testovací skripty pro jednotlivé *jednotky*, za které se zde považují právě metody pro základní výpočetní operace. Z tabulky lze vidět, že pro každou funkci definujeme platné vstupy, výstupy a případně možné výjimky, které mohou nastat. Ty by při neplatných vstupech měly být odchyceny. Jak je zobrazeno v obrázku 1.2, pro *součet* by měl test projít s libovolnými celými čísly na vstupu. Lze zde také otestovat prohození vstupních hodnot *A B* a ověřit, zda se výsledek korektně mění či nemění (dle toho, co je předpokladem). Podobně je tomu i pro *rozdíl* nebo *součin*. Tím, že použitý Jazyk Java je *silně typovaný*, není zde potřeba kontrolovat, zda vstupní číslo je z oboru *celých čísel* (jinými slovy *integer*), protože pokud by se jednalo například o číslo v datovém typu *float*, nešel by takový kód ani spustit. Ovšem hodnota 0 je stále platná ve zde využitém datovém typu *integer* a v případě dělení nesmí být druhý argument nulový. V tomto případě je tady také potřeba otestovat, zda jednotka při nulovém argumentu korektně skončí výjimkou. K testování se také běžně využívá znalostí odborného *testera*, který by měl být schopen odhalit i možné nečekané chování nad rámec specifikace.

ID	Název	Vlastnost	Hodnota
TC-1	Součet	Popis	Matematický sočet dvou celých čísel
		Vstupy	Celá čísla A a B
		Výstupy	Celé číslo, které je výsledkem součtu čísel A a B
		Vyjímky	A nebo B nejsou validní celé číslo
TC-2	Rozdíl	Popis	Matematický rozdíl dvou celých čísel
		Vstupy	Celá čísla A a B
		Výstupy	Celé číslo, které je výsledkem rozdílu čísel A a B
		Vyjímky	A nebo B nejsou validní celé číslo
TC-3	Násobení	Popis	Matematický násobek dvou celých čísel
		Vstupy	Celá čísla A a B
		Výstupy	Výsledek násobení čísla A hodnotou B
		Vyjímky	A nebo B nejsou validní celé číslo
TC-3	Dělení	Popis	Matematické dělení dvou celých čísel
		Vstupy	Celá čísla A a B
		Výstupy	Výsledek dělení čísla A číslem B
		Vyjímky	A nebo B nejsou validní celé číslo Číslo B je rovno 0.

Tabulka 1.1: Zjednodušené testové požadavky pro ukáukový příklad kalkulačky.

```

public int add(int a, int b) {
    return a + b;
}

public int subtract(int a, int b) {
    return a - b;
}

public int multiply(int a, int b) {
    return a * b;
}

public int divide(int a, int b) {
    return a / b;
}

```

Obrázek 1.1: Ukázkové funkce kalkulačky vhodné pro jednotkové otestování.

```
@Test
public void testAdd() {
    assertEquals(5, add(2, 3));
    assertEquals(-1, add(-4, 3));
    assertEquals(0, add(0, 0));
}

@Test
public void testSubtract() {
    assertEquals(-1, subtract(2, 3));
    assertEquals(-7, subtract(-4, 3));
    assertEquals(0, subtract(0, 0));
}

@Test
public void testMultiply() {
    assertEquals(6, multiply(2, 3));
    assertEquals(-12, multiply(-4, 3));
    assertEquals(0, multiply(0, 10));
}

@Test
public void testDivide() {
    assertEquals(2, divide(6, 3));
    assertEquals(-2, divide(-6, 3));
    assertThrows(IllegalArgumentException.class, () -> divide(5, 0));
}
```

Obrázek 1.2: Ukázkové jednotkové testy pro funkce kalkulačky.

Testovat je možné nejen zdrojový kód softwaru, ale také jeho grafické rozhraní (dále pouze GUI). Zde je možné využít stejného manuálního testování jako bylo popsáno výše, ovšem existují i sofistikované automatizované metody. Účelem GUI testů nemusí být pouze základní ověření funkčnosti na testových scénářích, ale také zajištění, že GUI se správně chová například na různých zařízeních nebo velikostech obrazovky. Dále lze také ověřovat, zda nové aktualizace neporušily staré funkce (regresní testy) či zhodnotit, zda výkon aplikace odpovídá očekávání/předpokladům. Tyto testy lze automatizovat jak skriptem (příkladem zde je framework *Robot Framework* nebo *Selenium*, využívané pro testy webu a v tomto projektu) tak skrze nástroje umožňující nahrávání uživatelských akcí a interakcí se softwarem a jejich následné přehrání (podobně jako funguje například *makro* v sadě Microsoft Office). Mezi takové nástroje se řadí software jako např. *Ranorex*, *AutoHotKey* či *PyAutoGUI*. [6]

1.2 Jazykové modely

Velké jazykové modely (zkráceně LLM) jsou druhem modelu strojového učení (ML), který využívá techniky *hlubokého učení*, které se zaměřují na zpracování a generování přirozeného jazyka (také známé jako metody NLP). Tyto modely jsou navrženy tak, aby rozuměli kontextu a významu jazyka, a dokázali pro *textový vstup* odhadnout (vygenerovat) *textový výstup*. Tato schopnost jim umožňuje provádět široké spektrum operací nad přirozeným jazykem jako sumarizace, překlad textu, generování obsahu či dialogu a podobně. Příkladem takových modelů je například známý *GPT* (Generative Pre-trained Transformer) od společnosti OpenAI nebo *BERT* (Bidirectional Encoder Representations from Transformers). [7] [8] [9]

LLM modely získávají svá data při procesu *trénování*. Jejich trénink zahrnuje analýzu a zpracování obrovského množství textových dat, jejichž řetězce jsou rozděleny na jednotky zvané tokeny. Tokenizace je proces, při kterém je text rozdělen na menší části, které mohou být *slova*, *části slov* nebo pouze *jednotlivé znaky*. Toto rozdělení umožňuje lépe analyzovat a učit se závislosti mezi jednotlivými částmi textu. Každý token je pak převeden na numerický vektor pomocí techniky zvané *embedding*, při kterém jsou jednotlivým tokenům přiřazené váhy. Váhy v modelu jsou kritické, protože určují, jaký význam a důležitost jednotlivá slova a tokeny mají. Tyto váhy jsou dynamicky upravovány během trénování, aby co nejlépe odráželi vzory a závislosti v tréninkových datech. Model díky těmto hodnotám může model zpracovávat text v matematické formě. Modely jsou většinou předtrénovány na rozsáhlé sadě korpusů textových dat, které zahrnují články, knihy, webové stránky a další prameny obsahující psaný text. Cílem trénování je naučit model základní strukturu jazyka bez specifického cíle. Váhy jsou zároveň jedny z parametrů modelu. Dalšími parametry také mohou být zkreslení (bias) jednotlivých vrstev nebo jejich aktivace. Věškeré parametry modelu mohou být upravovány v rámci trénovacího procesu a mají zásadní vliv na výstup modelu. [10] [11]

Po natrénování modelu na obecné sadě dat je také možné přejít ke kroku zvanému „fine-tuning“, ve kterém dochází k jeho jemnému doladění, aby jeho schopnosti odpovídaly konkrétním úkolům nebo aplikacím, pro které by model měl být použit. K tomu je využita datová sada úzce zaměřená na tuto oblast. Během této fáze se upravují vnitřní parametry modelu tak, aby lépe odpovídali cílovým úkonům. Váhy v modelu, které jsou upravovány, určují, jak model reaguje na různé vstupy během interferenční fáze. Příkladem takového *fine-tuningu* může být úprava modelu pro generování zdrojového kódu specifického programovacího jazyka.

Pro funkci LLM je také důležitý kontext modelu, který je definován jako soubor relevantních informací, které předcházejí nebo následují za aktuálním tokenem a ovlivňují jeho význam nebo použití v dané situaci. LLM využívají kontext k lepšímu porozumění tomu, jak jsou slova používána ve větách, což umožňuje modelu generovat text, který je koherentní a relevantní k zadaným promptům. Kontext je tedy klíčový pro trénink modelu, protože pomáhá modelu pochopit nejen izolované tokeny, ale také celkovou strukturu a význam textu.

Jazykové modely se spouštějí během procesu zvaného *inference*. Jde o proces, při kterém model generuje výstupy (odpovědi, texty) na základě vstupních dat, které dostává. Těmto vstupním datům se říká *prompt*, kterým je podnět poskytnutý uživatelem jako *otázka*, *věta k dokončení*, nebo jiný typ zadání, které model používá jako východisko pro generování odpovídajícího textu. Znalosti a kontext nabrané při tréninku jako váhy jsou využity k predikci nejpravděpodobnějších tokenů, které by měli následovat po promptu. Interakce s LLM prostřednictvím promptu umožňuje uživatelům specifikovat, jaký typ informace nebo textu chtějí od modelu získat. Modely jsou běžně navrženy tak, aby rozpoznávali strukturu jazyka, tak kontext, ve kterém jsou jeho tokeny použity. To umožňuje modelům přizpůsobovat své odpovědi v závislosti na zadání a předchozím textu v konverzaci. [12] Při inferenci však také může docházet k fenoménu označovanému jako „halucinace“, kdy model generuje obsah, který je nesmyslný nebo nevěrný zdrojovým datům, jak definuje [13]. I přesto, že existují řešení pro vyřešení problémů spojených s halucinací jako např. *grounding* [14], tak se na ně nelze spoléhat a při práci s výstupy modelů je třeba počítat s těmito vedlejšími produkty inference.

1.3 Motivace

Tím, že tvorba softwarových testů ze specifikací je problém převodu *text na text* (přesněji zdrojový kód), souhlasí tak s funkcí LLM, která spočívá v predikci *textu*, na bázi *textového* vstupu (promptu). Jinými slovy jde o stejný problém. Tento předpoklad nám umožňuje převést vytváření testů na automatizovanou činnost s využitím strojového učení. Samotné vytváření testů v softwarovém vývoji je činnost, která je časově náročná a často vyžaduje i zkušeného uživatele (běžně programátora) pro jejich sestavení. Mimo jejich sestavení je také nutné i manuálně odhadovat chování softwaru, které v rámci specifikace nemusí být dáno. I přes nepopiratelnou důležitost testů jsou však z výše zmíněných důvodů často neúplné či naprosto opomenuté, což vede k neotestovanému softwaru.

Naším návrhem tedy je zautomatizovat samotný proces tvorby testů tak, aby je s pomocí jednoduchého nástroje zvládl vytvořit i méně zkušený (neprofesionální) uživatel jako například *tester*, *manažer apod.* Potenciálním scénářem může být, když zákazník určí specifikace (případně související požadavky), vývojář vytvoří prototyp softwaru, na kterém poté může tester dle dostupné specifikace definovat chování testu a dle něj je za pomoci automatizovaného procesu sestavena samotná sada testů. Výhodou generování s využitím LLM může být jejich schopnost do jisté míry porozumět kontextu textu. Mohou tak případně odhalit chyby (či stavy), které by nezkušený uživatel nemusel odhadnout. Zároveň jsou oproti člověku rychlé a tím, že jsou naučeny na širokém spektru dat (často čítající programovací jazyky, dokumentace, ...), tak je také možné, že spoustu vzorů ve zdrojových kódech a chování softwaru již viděli v učících datech.

2.1 Provedená práce v problematice

2.1.1 Předchozí automatizovaná řešení

Jazykové modely nebyli prvními pokusy o automatizované generování jednotkových testů. V současnosti existuje spousta metod, které se neustále rozvíjejí a nové vznikají. Mezi ně patří příklady jako *fuzzing*, *generování náhodných testů řízených zpětnou vazbou*, *dynamické symbolické vykonání*, *vyhledávací a evoluční techniky*, *parametrické testování*. Zároveň také již na počátku století byly pokusy o vytvoření vlastní neuronové sítě sloužící právě čistě k úkolu testování softwaru (například článek "Using a neural network in the software testing process"). [15] V této sekci je ukázka několika z nich.

2.1.1.1 Programatická řešení

Jedna z používaných programatických automatizovaných metod pro tvorbu 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. [16]

Pouze vstupy dokáže také generovat metoda *symbolického vykonání*, která postupně analyzuje chování větvení programu. Začíná bez předchozích znalostí a používá řešitel omezení (anglicky označovaný jako *constraint solver*) k nalezení vstupů,

kteřé prozkoumají nové vykonávací cesty (anglicky *execution path*) v rámci řízení toku programu. 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 iterativní proces se opakuje a nadále zpřesňuje sadu známých chování a snaží se maximalizovat pokrytí kódu. *Znamé chování* lze definovat jako sadu identifikovaných vykonávacích cest programu, včetně podmínek a stavů, které byly analyzovány a uloženy do znalostní báze nástroje (q). Umožňuje efektivně generovat nové testovací vstupy a předvídat chování programu na základě již známých dat. 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. [17] Tato metoda je implementována například v nástroji IntelliTest v rámci IDE *Visual Studio*. Je používána v kombinaci s *parametrickými testy*, také označovanými jako PUT. Na rozdíl od tradičních jednotkových testů, které jsou typicky uzavřené a nezměnitelné metody, parametrické testy mohou přijímat libovolné množství parametrů a adaptují se tak testovacím scénářům. Software pro jejich vytváření 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 vykonávacích cest testovaného kódu. Každý vstup, který pokrývá jinou cestu, je "serializován" jako jednotkový test. Parametrické testy mohou být také generické metody (funkce schopná pracovat s různými druhy datových typů bez nutnosti psát pro každý typ specifický kód zvlášť), v tom případě musí uživatel specifikovat typy použité k instanci metody. Testy také mohou obsahovat atributy pro *očekávané* (např. dělení nulou) a případně *neočekávané* vyjimky (např. předání reference na nulový objekt). Neočekávané vyjimky vedou k selhání testu. PUT tedy do velké míry redukuje potřebu uživatelského vstupu pro tvorbu jednotkových testů. [18] [19]

Pokud zvolíme symbolické řešení vstupu (některé vstupní argumenty testovaného programu nebudou reprezentovány konkrétními hodnotami, ale zastoupené symbolickými proměnnými) společně s determinovanými vstupy (využity konkrétní hodnoty) a vykonávací cestou, vzniká tak hybridní řešení zvané jako *konkolidé testování* nebo *dynamické symbolické vykonání*. Tento druh testů dokáže 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. Nedeterministické chování není tolik obvyklé a objevuje se zejména v případě *paralelizmu*, *externího přerušení* či u algoritmů postavených na *pseudonáhodě*. [20] Tento problém není specifický pouze pro tento typ automatizovaného testování, ale i pro předchozí varianty. Zatímco *symbolická* část vykonání objeví veškeré možné cesty, *konkrétní* vykonání poté určí

sled jednotlivých událostí při spuštění, aby nedocházelo k redundantnímu testování. [21] Dalším problémem této metody také může být výběr stavových proměnných. Problémem je zejména jejich rozsah. Například více stavových proměnných s malým celočíselným rozsahem povede pouze k omezenému množství cest, kdežto pouze jednotky takových proměnných o celém rozsahu typu `double` může vést k daleko vyššímu počtu cest, což také zvýší výpočetní náročnost těchto metod a tedy se i snižuje pravděpodobnost úspěšného nenalezení praktického řešení (v případě, že takové řešení vůbec existuje nebo je to účelem vyhledávání). [22] [23] [24] [25]

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 (jedná se o dostupnou funkci nebo metodu v rámci programu) a hledá hodnoty argumentů, které náhodně generuje nebo upravuje vstupy vytvořenými v předchozích iteracích tohoto procesu. Jakmile dojde k sestavení vstupu, je provedeno jeho vykonání a výsledek ověřen proti sadě kontraktů a filtryů. Výsledek vykonání jeho srovnáním s očekávanými výstupy, definovanými v kontraktu, a filtry určuje, zda je vstup redundantní (definován tak filtry), proti pravidlům (jak filtrů tak kontraktů), porušující kontrakt (např. nedefinované výstupy) nebo užitečný pro generování dalších vstupů (splňuje kontrakt a prochází filtry). Technika vytváří sadu testů, které se skládají z jednotkových testů pro testované třídy. Úspěšné testy mohou být použity k zajištění faktu, že kontrakty 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. [26]

Z programatických metod se zdají být nejpokročilejší *evoluční algoritmy* pro generování sad jednotkových testů, využívající přístup založený na vhodnosti (fitness funkce), aby vyvíjely testovací případy, které mají za cíl maximalizovat pokrytí kódu a detekci chyb. Na rozdíl od náhodného generování, které vytváří testy bez specifického cíle, evoluční algoritmy používají fitness funkci k iterativnímu zdokonalování testů na základě jejich schopnosti odhalovat chyby a efektivně prozkoumávat kód. Tyto algoritmy mohou autonomně generovat testovací vstupy, které jsou navrženy tak, aby prozkoumávaly různé vykonávací cesty v aplikaci a dynamicky se upravují na základě pozorovaných výsledků, což zvyšuje jejich účinnost a snižuje potřebu manuálního vstupu. 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. [27] [28]

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"[29], kde byly testovány primárně RNN a experimentálně CNN sítě (ty však měli problém s větším množstvím tokenů). Modely byly testovány na programech napsaných v jazyce Java. Metoda se zaměřila na programy s plným přístupem k jejich zdrojovému kódu a zkompilevanému bytkódu, což umožnilo detailní analýzu z pohledu white box. Při nejlepší konfiguraci dosáhl výsledek modelu 70.5% parsovatelného kódu (tedy takového bez chyb, který lze spustit) 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 *symbolické vykonávání*, 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 znaloosti 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 pomoci slovní specifikace pro funkci nebo vlastnosti. Takové specifikace se používají napří-

lad v *aero-space* nebo *automotive* sektoru. [30] LLM také mohou objevit jiné možné stavy, které mohou u vstupu nebo výstupu nastat a pokusit se pro ně navrhnout test, případně nedokonalosti ve specifikaci, které mohou obejít.

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"[31] 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 [32]. 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!"[32] 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 permissivně 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"[33] 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*¹ a *Evosuite SF110*². 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* [34].

¹<https://ai.google.dev/gemma/terms>

²<https://paperswithcode.com/dataset/evosuite-sf110-benchmark>

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 [31] se využívá jako benchmark sada balíčků v jazyce *JavaScript*, který kvůli absenci explicitního typování, může být obtížnější pro strojové testování oproti jazyku *Java*, který je využit ve zbylých 2 pracích. [35] také využívá *Java* a s modelem *gpt-3.5-turbo* dosahuje podobného pokrytí kódu a úspěšnosti jako *Codex* v práci [33].

2.1.3 Modely

Na LLM modelech nás konkrétně zajímá schopnost porozumět programovacím jazykům a ty poté také generovat na výstupu. Důležité pro nás také je, zda daný model je proprietární či otevřený a pod jakou licenci, 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 [31] 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** [36] 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 restriktce z obavy tvůrců, protože by model mohl někoho při neoprávněném použití ohrozit. Tyto restriktce se aplikují na všechny derivace projektů pod touto licenci. Zároveň není kompatibilní s Open-Source licenci právě kvůli těmto restrikcím.

Model *StarCoder* se také dočkal novější verze *StarCoder2*, který nepřímo navazuje na model *StarCoderBase*. Je naučen na archivu GitHub repozitářů 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. [37] I samotný model *LLama 2*, případně novější verze *LLama 3* by měli být schopné generovat zdrojový kód a oproti *CodeLlama* modelu mohou mít v některých ohledech lepší predikci pro tokeny v obecném kontextu. [38]

Copilot. Velmi populárním nástrojem pro generování kódu za pomoci LLM je GitHub Copilot³, 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 X⁴. 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. [39] 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.

Anthropic. Velmi schopným modelem, který jak jeho tvůrci⁵ tak i nezávislé strany

³<https://github.com/features/copilot>

⁴<https://github.com/features/preview/copilot-x>

⁵<https://www.anthropic.com/news/claude-3-family>

pusuzují jakožto stejně přesný a inteligentní, ne-li v některých případech přesnější a schopnější jak model GPT-4, je Claude 3 od společnosti Anthropic. [40] Ten je nabízen ve 3 varintách: *Opus*, *Sonnet* a *Haiku*, kdy verze *Opus* je s počtem 137 miliard parametrů a kontextovým oknem 200 tisíc tokenů největší z nich a ostatní využívají méně parametrů. Znamená to ale také, že mají nižší cenu, protože rodina modelů Claude 3 je pouze proprietární a dostupná buďto skrze API společnosti Anthropic či na službě AWS. Mimo EU je také dostupný v rámci chatovací aplikace. [41]

Cohere. Podobně jako společnost Meta, vydává otevřené modely i startup *Cohere*, který stojí za modely jako *Command*, *Command R* a *Command R+*. Většina z nich je otevřena⁶ pro *nekomerční* použití (pod licencí CC-BY-NC-4.0) a tedy volně použitelné např. pro výzkumné a neziskové účely. Mimo základních schopností tyto modely oficiálně podporují více jak 13 mluvených jazyků. Zároveň modely *Command R(+)* byli navrženy se schopností využívat zdrojování metodou RAG za pomoci konektorů, kterými může být například *internetový vyhledávač* nebo *lokální soubory*. Sám tvůrce však upozorňuje na fakt, že větší modely jsou vhodné spíše pro účely IR, zatímco pro běžné úlohy sumarizace je vhodný menší a proprietární model *Command*. I přes fakt, že tyto modely lze provozovat *lokálně*, je zde problém ve velké náročnosti na hardware. Modely jsou však dostupné i skrze nejen oficiální API, ale i jako služba u cloudových poskytovatelů. [42] [43]

Google. Společnost Google v současnosti nabízí 2 rodiny LLM modelů a to víceúčelové *Gemini*⁷, které jsou proprietární a dostupné ve verzi 1.0 Pro a Ultra a novější 1.5 Pro. Google také vypustil malé modely pro generování zdrojového kódu s názvem *Gemma* a pod upravenou Apache 2.0 licencí⁸. Podobně jako *Command R* modely mají schopnost podkládat svůj výstup na bázi webového vyhledání nebo poskytnutých dokumentů. V případě modelu *Gemini 1.5 Pro* je tato schopnost navíc rozšířena tím, že podporuje kontext s velikostí až 1 milion tokenů. Tato schopnost může přijít vhod například pro návrh zdrojového kódu v rámci větší kódové základny. *Gemini* je dostupné v základním modelu 1.0 Pro zdarma (dříve Bard), ovšem služba *Gemini Studio* nabízející ostatní modely není v současnosti dostupná v EU a tedy jediný přístup k modelům je skrze API v rámci služby Google Cloud⁹.

⁶Např. model *Command R+*: <https://huggingface.co/CohereForAI/c4ai-command-r-plus>

⁷<https://deepmind.google/technologies/gemini/>

⁸<https://ai.google.dev/gemma/terms>

⁹<https://cloud.google.com/vertex-ai>

Rodina modelů Gemma je poté volně dostupná a to v provedení se 2 nebo 7 miliardy tokenů a ve verzích *Base* a *Instruct*. Dle tvůrců by model měl být pro účely psaní zdrojového kódu, matematického zápisu a logiky přesnější jak model *Llama 2* ve verzi se 13 miliardy paramterů. [44] V rámci upravené licence se společnost *Google* vzdává odpovědnosti za kód vygenerovaný těmito modely.

Mistral. Open-source modely schopné generovat kód také nabízí společnost *Mistral* společně s jejich proprietárními modely jako *Mistral Large* modelem. Jedním z jejich prvních otevřeně vydaných modelů byl *Mistral 7B*, který jak název vypovídá obsahuje pouze 7 miliard parametrů, ale i přes tento fakt je schopen v některých kategoriích překonat větší modely jako například *Llama 2*. [45] Výhodou je, že může být jednoduše vyladěn pro potřeby specifické aplikace. Jejich dalším modelem je také *Mixtral 8x7B* cílící na dosažení stejných kvalit jako např. model *GPT-3.5 Turbo* v běžně používaných benchmarcích. Jeho větší varianta *Mistral 8x22B* pak přidává podporu pro další mluvené jazyky. [46] Všechny jejich otevřené modely jsou uvolněny pod licencí Apache 2.0. Mezi jejich proprietární modely se pak řadí model *Mistral Small* a zejména *Mistral Large*¹⁰, který podobně jako *Mistral 8x22B* zvládá více jazyků tak i podporuje RAG. Dostupný je jako API na jejich vlastní platformě *Le Plateforme* či u cloudových poskytovatelů.

¹⁰<https://mistral.ai/news/mistral-large/>

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

Tabulka 2.1: Přehled a srovnání studií

Model	Otevřenost	Licence	Počet parametrů	Programovacích jazyků	Datum vydání
<i>StarCoderBase</i>	Volně dostupný	CodeML OpenRAIL-M 0.1	15.5 miliardy	80+	5/2023
<i>CodeLlama</i>	Volně dostupný	Llama 2 Licence	34 miliard	8+	8/2023
<i>gpt-3.5-turbo</i>	Uzavřený	Proprietární	175 miliard?	?	5/2023
<i>gpt-4</i>	Uzavřený	Proprietární	1.76 bilionu	?	3/2023
<i>claude-3-opus</i>	Uzavřený	Proprietární	137 miliard	?	3/2024
<i>command</i>	Uzavřený	Proprietární	52 miliard	?	?
<i>command-r</i>	Volně dostupný	CC-BY-NC-4	35 miliard	?	3/2024
<i>command-r-plus</i>	Volně dostupný	CC-BY-NC-4	104 miliard	?	4/2024
<i>mistral-7b</i>	Volně dostupný	Apache 2.0	7 miliard	?	9/2023
<i>mistral-8x7b</i>	Volně dostupný	Apache 2.0	7 miliard	?	2/2024
<i>mistral-8x22b</i>	Volně dostupný	Apache 2.0	22 miliard	?	4/2024
<i>mistral-large</i>	Uzavřený	Proprietární	?	?	2/2024
<i>gemma</i>	Volně dostupný	Gemma	7 miliard	?	2/2024

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 automatizovanou 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. [47] [48] 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 ¹¹ 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 držících se pevně specifikací, vytvořených Vaisem. [49] [50] 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.

¹¹Aplikaci lze nalézt na adrese: <https://projects.kiv.zcu.cz/tbuis/web/>

University Information System Home

Mia Orange Logout

Student's View

- Overview
- My Subjects
- Other Subjects
- My Exam Dates
- Other Exam Dates

First name:

Last name:

Email:

Obrázek 2.1: Prostředí systému TbUIS z pohledu studenta.

3.1 Výběr cíle a metody testů

V rámci této práce se chceme zaměřit nejen na možnost generování softwarových testů za pomoci LLM a jeho proveditelnosti, ale také na jejich schopnost odhalit možné problémy, které se mohou objevit a jsou proti specifikaci daného testu nebo mimo ni. Toho chceme docílit nejen vygenerováním testů a srovnáním jejich vlastností a výsledků s testy napsanými člověkem, ale také narozdíl od v současnosti dostupných prací na podobné téma máme v plánu sputit testy nejen na validním kódu testovaného softwaru (aplikace), ale také zkusit do zdrojového kódu injektovat chyby a ověřit, zda výstupy testů z LLM jsou schopné tyto chyby detekovat a také analyzovat tyto výstupy a jejich případné nedostatky. Předpokladem zde je, že testy predikované modelem budou více verbózní a pravděpodobně komplexnější než lidsky napsané testy, avšak funkcionalita (resp. její ověření na testované aplikaci) by měla v nejlepším případě být jak mezi *strojovými* tak *lidsky napsanými* testy shodná. Jeden rozdíl, který v základu volíme je, že naše *strojové* testy jsou parametrizovány v rámci promptu a nepředpokládá se od nich (ani nevyžaduje) možnost načítat parametry za chodu jako mohou mít *lidsky napsané* testy. Za parametry se mohou považovat například vstupní hodnoty, které testy využívají, případně hodnoty assertů (například v kombinaci se sadou vstupních parametrů) a podobně.

Jako druh testů jsme zde zvolili GUI testování webových stránek (resp. aplikace) s využitím nástroje *RobotFramework* a knihovny *SeleniumLibrary*. Nejedná se tedy o klasický případ *jednotkových* testů, ale stále má podobný rozsah, i když se dá říci, že vygenerované testy již mají přesah do integrační fáze. Zvolený druh testování byl vybrán i z důvodu, protože díky němu může tato práce navazovat na předchozí univerzitní projekt *TbUIS*, popsáný v sekci 2.2, který přesně vyhovuje našemu požadavku pro *injekci chyb* a vznikl přesně pro tyto případy benchmarkingu

softwarových testů. Robot Framework je rozhraní postavené nad Pythonem, využívající vlastní jazyk pro zápis testů, který využívá příkazy v podobě tzv. *klíčových slov*, které poskytují buďto poskytnuté knihovky nebo je lze definovat i v rámci testového souboru. Umožňuje také importovat a spustět klasické Python moduly či podle nich definovat objekty. Jednotlivé testovací *scénáře* se poté definují jako „Test Cases“. Jak pro *klíčová slova* tak *scénáře* či *nastavení* je potřeba tuto sekci kódu oddělit, jak lze vidět v příkladu 3.1. Lze také importovat např. klíčová slova či testovací data z externích skriptů jako je ukázáno na řádce 3. Podobně jako v běžných programovacích jazycích lze u klíčových slov definovat *argumenty*. Stejně tak jsou podporovány proměnné, které mohou být deklarovány např. jako konstanty (viz řádka 5) nebo použity přímo v kódu pro uložení výstupu klíčového slova (řádka 16 ukázky). Jednotlivé výrazy jsou od sebe odděleny *tabulátorem*, případně jiným podporovaným znakem.

Zdrojový kód 3.1: Příklad struktury RobotFramework testu.

```
1 *** Settings ***
2 Library      SeleniumLibrary
3 Resource     keywords.resource

5 *** Variables ***
6 ${URL}       https://example.com
7 ${BROWSER}   Chrome

9 *** Test Cases ***
10 Open Website and Check for Text
11     [Setup]    Open Browser      ${URL}      ${BROWSER}
12     Page Should Contain    Welcome to Example Domain
13     [Teardown]  Close Browser

15 Check User Name
16     ${name}=    Get User Name
17     Should Be Equal    ${name}      ${exp_full_name}

19 *** Keywords ***
20 Open Browser
21     [Arguments]    ${url}      ${browser}
22     Open Browser    ${url}      ${browser}
23     Maximize Browser Window
```

Výsledným cílem práce je tedy předpokládán nástroj pro vygenerování jednotkových testů využívající Robot Framework na bázi uživatelské specifikace. Tento nástroj by také měl být schopen otestovat vygenerované testy dle daných specifikací na požadovaných variantách testované aplikace, mezi kterými budou klony u kterých je i není očekávané selhání testu. Tyto výsledky budou na konci vyhod-

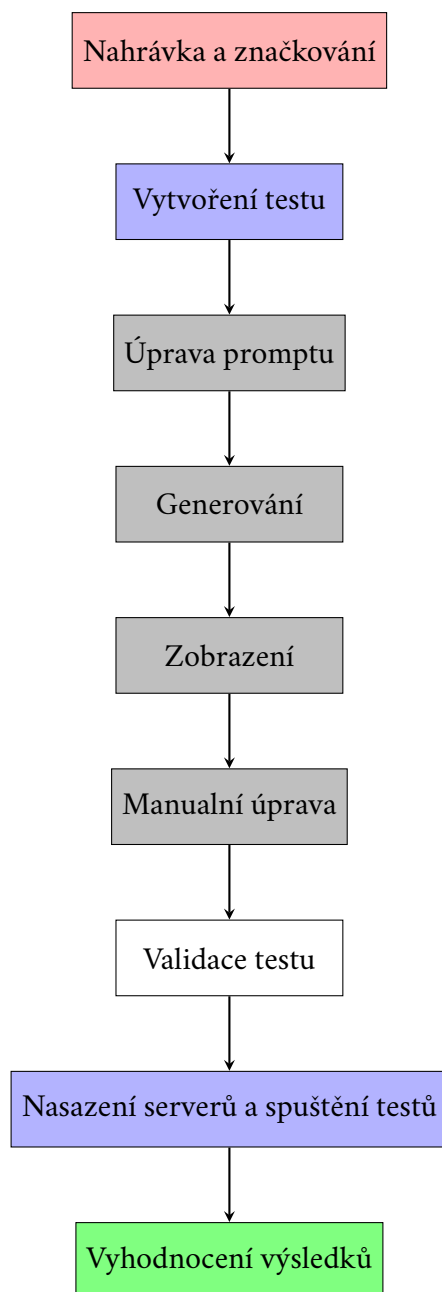
noceny pro různé LLM modely a bude určeno, které z modelů dokáží generovat testy s co nejvyšší úspěšností detekce chyb a také je možno takovýto přístup využít v navazujících pracích, které by popisovanou generaci testů využívali.

3.2 Navrhované řešení

Řešení by v návrhu mělo fungovat jako softwarová *pipeline*, tedy řetězec úkonů, které na sebe v rámci automatizovaného pracovního postupu musí navazovat, aby docílili určitého výsledku, za který v tomto případě považujeme *otestovaný software*, zatímco jednotlivé vygenerované testy a jejich varinaty se berou jako mezivýsledky. Celý návrh pipeline je zobrazen na obr. 3.1. Kompletní proces začíná tím, že uživatel vytvoří s pomocí nástroje záznam prostupu webovou stránkou, tedy dokument, který obsahuje, na které prvky kliká či s nimi jinak interaguje a v jakém pořadí či časovém intervalu. Zároveň je od uživatele očekáváno, že označí nebo si poznamená prvky (např. dle jejich identifikátoru) od kterých je očekávána nějaká hodnota nebo chování (assert). Poté uživatel za pomoci orchestračního programu vytvoří nový test, nebo spíše šablonu podle které ho bude model generovat. Vyvořená šablona bude uživatelem doplněna o záznam (přesněji *odkaz na něj*) a také doplněna o podmínky pro testování, které byli během záznamu poznamenány nebo vycházejí ze specifikace, dle které je testovací scénář (nebo přesněji test) tvořen. Doplněná šablona poté tvoří *prompt* pro model.

Po přesném specifikování podoby testu lze přistoupit k jeho *generování*, při kterém se LLM model dotazuje vytvořeným *promptem* a je očekáváno vygenerování x tokenů, které lze považovat za vytvořený test, případně při detekci nevhodných tokenů lze přistoupit k případné validaci či filtrování. Samozřejmostí také je, že uživateli je do vytvořeného testu umožněno nahlédnout a při viditelné chybě generovaného kódu i připustit manuální zásah a úpravu (nebude využito v této práci, jde jen o diskutovanou možnost). Samotné vytvoření testu a jeho generování je možné opakovat pro každý potřebný scénář či testový případ, který uživatel potřebuje vytvořit, než nastane další krok.

Tímto dalším krokem je již samotné *spuštění testů*, kterému však musí předcházet *nasazení testované aplikace* a to v různých variantách. V rámci tohoto procesu bude vždy nasazena jedna varianta aplikace, poté spuštěny veškeré požadované testy, uloženy výsledky a program přejde k další variantě testovaného programu. Zmíněné kroky je také možno parametrizovat dle požadavků uživatele. Na konci pipeline by měl uživatel být schopen vyhodnotit výsledky a to jak programaticky tak manuálně.



Obrázek 3.1: Návrh pipeline projektu.

Generování testů

4

4.1 Prerekvizity pro generování a spuštění testů

Pro spuštění projektových programů jsou vyžadovány následující softwarové a hardwarové prerekvizity:

- **Spuštění testů**

- **Operační systém** - Program je navržený pro „UNIX-like“ operační systémy (Linux, MacOS), které jsou pro jeho spuštění vyžadovány. V případě operačního systému Windows je doporučeno využít technologii **WSL2**.
- **Docker Desktop** (včetně Docker Compose).
- **Python 3.11** nebo vyšší (z důvodu kompatibility některých frameworků).
- **Robot Framework** společně s knihovnou *SeleniumLibrary* jako Python moduly.
- V případě, že se společně s Robot Framework nenainstaluje, je také potřeba přidat Chrome nebo Chromium driver.
- Přítomný display server (nelze spouštět na čistě CLI systémech).

- **Lokální spouštění LLM** (generování testů)

- **Operační systém** - Libovolný.
- **Hardware** - CPU s podporou AVX2 instrukční sady, minimum 16GB RAM, v případě běhu na VGA doporučeno alespoň 6GB VRAM.
- Nainstalovaný software **LM Studio**.

4.2 Výběr scénářů

Pro vygenerování testů za pomoci LLM bylo nejdříve nutné vybrat ze sady *use case*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 webu² projektu společně s vysvětlením jednotlivých chyb 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.

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

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

Specifikace	Popis	Porucha na klonech
1	<i>Přihlášení do aplikace</i> 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	<i>Odepsání předmětu</i> 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	<i>Zapsání předmětu</i> Student se přihlásí, zapíše předmět v patřičné sekci, který se následně zobrazí i v ostatních částech systému. Z učitelského pohledu by se student měl zobrazit na seznamu studentů daného předmětu.	25, 26, 28
8	<i>Registrace na zkoušku</i> 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	<i>Zobrazení spolužáků u zkoušky</i> Student se přihlásí a u zkoušky si může rozkliknout seznam všech účastníků.	
10	<i>Zrušení předmětu</i> Učitel po přihlášení klikne na tlačítko <i>Remove</i> 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	<i>Zobrazení studentů u předmětu</i> Učitel se přihlásí do systému a u předmětu je schopen si zobrazit seznam zapsaných studentů.	26, 28

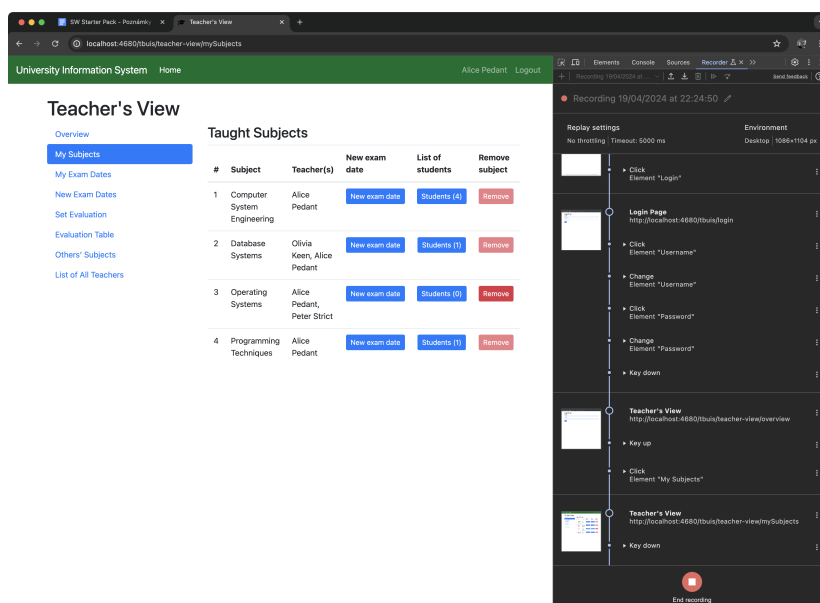
Tabulka 4.1: Specifikace pro generované testy - část 1

Specifikace	Popis	Porucha na klonech
12	<i>Zrušení zkoušky</i> Učitel se přihlásí a v sekci jeho přidělených zkoušek odstraní konkrétní termín. Tento termín by nyní neměl být vidět jak v učitel-ském tak studentském pohledu do systému.	20, 21, 23, 26, 28
17	<i>Přihlášení se k výuce předmětu</i> 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	<i>Zobrazení seznamu učitelů a předmětů, které vyučují</i> Přihlášený učitel je schopen si zobrazit seznam všech 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 pomoci nástroje v Google Chrome

Nástroj je schopen vyexportovat výstup nahrávání v řadě formátů. Zde jsme zvolili 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 pomoci 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 uživatelská 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.

Výpis 4.1: Ukázková struktura input složky

```

1 milan:dp/program/input$ ls -l
2 milan      448 Apr 21 20:52 ..
3 milan      3815 Mar 27 20:24 rec-spec-1-student.json
4 milan      3820 Mar 27 20:28 rec-spec-1-teacher.json
5 milan      6043 Mar 30 19:42 recording-spec-4-student.json
6 milan     10635 Mar 31 09:18 recording-spec-4-teacher.json
7 milan      1370 Apr 21 20:52 spec-1.txt
8 milan      1328 Mar 31 09:28 spec-4.txt
9

```

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 konkrétně 3 pracovní režimy:

1. Vytvoření 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 pomoci argumentů (argumenty jednotlivých režimů ukázané ve výpisu 4.2). V případě *vytvoření nového testu* se za argument přidává název šablony testu. Tento název musí být unikátní 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

```

1 milan:dp/program/input$ python3 test.py -n #Nový test
2 milan:dp/program/input$ python3 test.py -i #Generování
3 milan:dp/program/input$ python3 test.py -r #Spuštění testů
4

```

Výpis 4.3: Vytvoření nového testu (šablony)

```

1 milan:dp/program/input$ python3 test.py -n specification-1
2

```

Zdrojový kód 4.4: Vzor pro vyplnění šablony testu

```

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

3 - //TODO

5 {% include 'recording.json' %}
6

```

Zdrojový kód 4.5: Vyplněná šablona testu pro specifikaci 18

```

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

3 - Check if there are these names present on the page: Julia
  Easyrider, Olivia Keen, John Lazy, Alice Pedant, Thomas
  Scatterbrained, Peter Strict
4 - Check if element with path //*[@id="tea.listOfAllTeachers.
  table.teacherRow-0"]/td[3] has text matching "Numerical
  Methods"
5 - 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"
6 - Check if element with path //*[@id="tea.listOfAllTeachers.
  table.teacherRow-2"]/td[3] should not contain text
7 - 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"
8 - Check if element with path //*[@id="tea.listOfAllTeachers.
  table.teacherRow-4"]/td[3] has text matching "Computation
  Structures"
9 - 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"

11 {% include 'recording-spec-18.json' %}
12

```

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, 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 proprietární 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

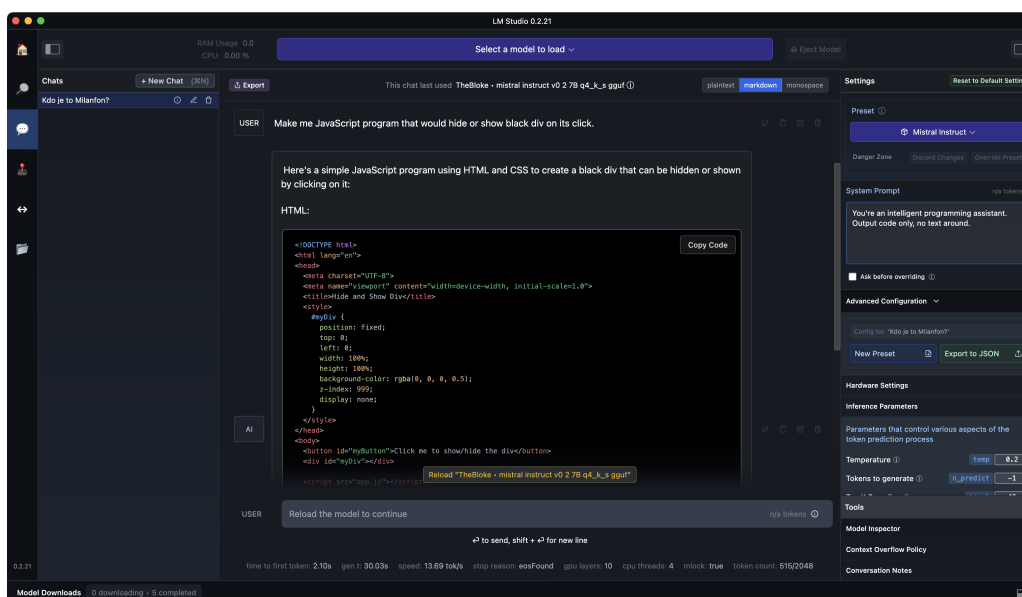
Pro *lokální* modely a jejich spuštění byl využit software *LM Studio*, který je postaven nad C++ knihovnou `llama.cpp`. Ta umožňuje jednoduché spuštění LLM modelů ve formátu GPT-Generated Unified Format (GGUF). Jedná se o *binární* formát souborů

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	Mixtral 7B	TheBloke/Mistral-7B-Instruct-v0.2	Lokální
	Mistral Large	mistral-large-2402	API <i>La Plateforme</i>
Anthropic	Claude 3 Opus	claude-3-opus-20240229	API
Meta	Codellama	TheBloke/Phind-CodeLlama-34B-v2	Lokální
	Llama 3 70B	MazyarPanahi/Meta-Llama-3-70B-Instruct	
	WizardCoder	TheBloke/WizardCoder-Python-34B-V1.0	
Google	Gemini 1.5 Pro	gemini-1.5-pro-preview-0409	API <i>Google Cloud</i>

Tabulka 4.4: Použité LLM modely

určený pro rychlé načítání a ukládání modelů využívaných pro účely interferenčních úloh. Jeho návrh umožňuje snadnou úpravu při zachování zpětné kompatibility. [51] [52] 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. [53] 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



Obrázek 4.2: LM Studio

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 runtime 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 *runtime* v rámci *LM Studio* (resp. *llama.cpp*) může uživatel 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

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

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 relevantní informace, které mohou zlepšit přesnost jeho výsledků nebo mu umožnit lépe pochopit kontext. Tento prompt je běžně vložen pouze na začátku konverzace 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 spuštění lokálních modelů byla využita PC sestava, jejíž komponenty jsou popsány 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 akcelerován jen takový počet vrstev, 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 proprietárních 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 podporuje 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** - Vypsát 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 pomoci 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 prostředí 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 určitém 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 4.3. Šablona, kterou uživatel vytvořil je použita pro *render* finálního promptu za pomoci šablonovacího jazyka *Jinja*. Tato metoda byla zvolena z důvodu *jednoduché a přehledné modifikace* vstupů (zde nazvaných 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.

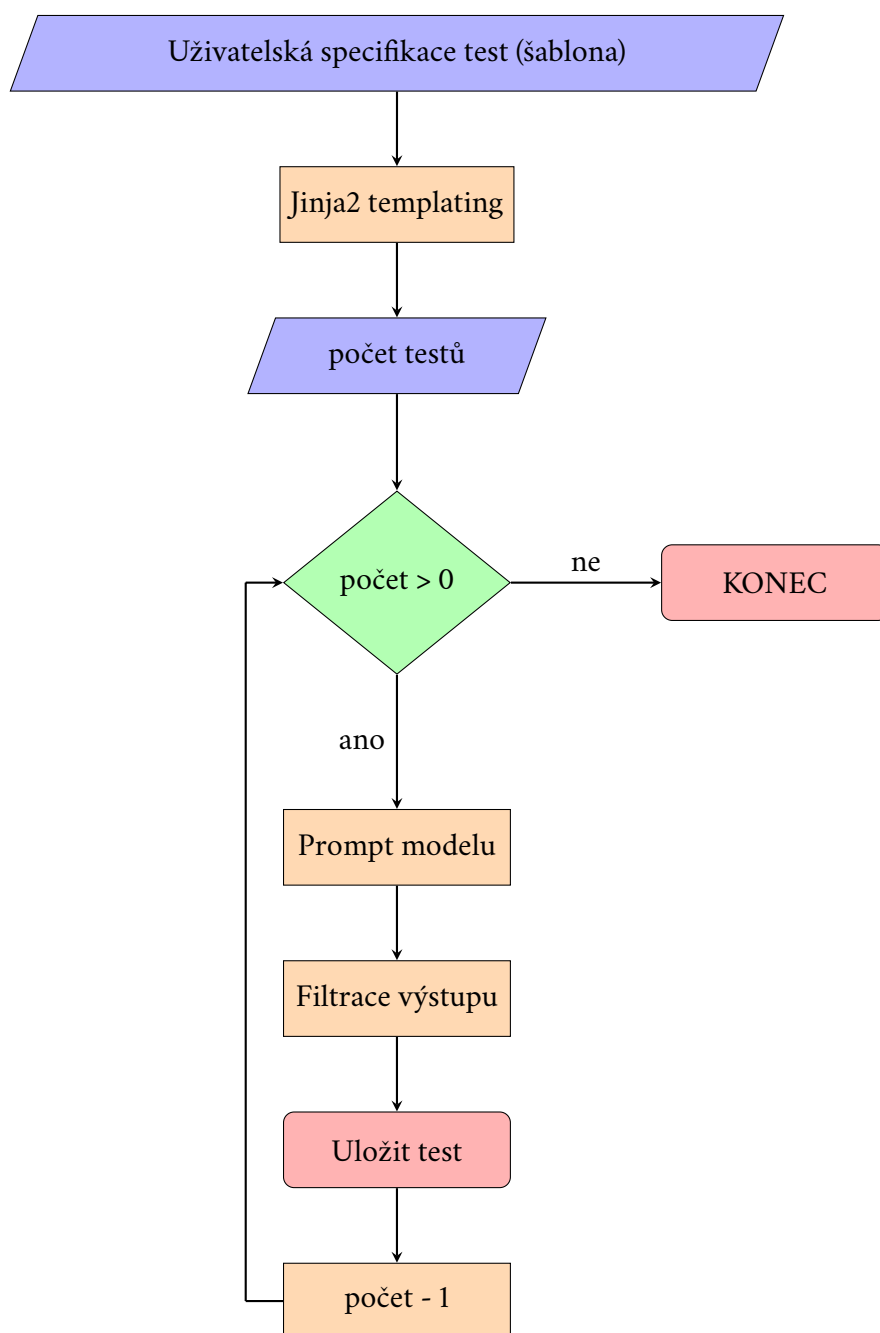
Výpis 4.7: Ukázka volání generace testu dle šablony

```
1 milan:dp/program$ python3 test.py -i spec-4 --count 10
2
```

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 modelu 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 požadavků a nedostatků, které byli při testovacím generování upozorovány.

Zdrojový kód 4.8: Použitý systémový prompt

```
1 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.
```



Obrázek 4.3: Zjednodušené schema algoritmu pro dotazování jazykového modelu při generování testu.

2

Zdrojový kód 4.9: Ukázka ".env"souboru

```

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

5 MAX_TOKENS=3000
6 DEVICE="mps"
7

```

I přes důraz na nutnost pouze *programového* výstupu v rámci systémového promptu, mají modely tendenci 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 kód z tohoto výstupu jednoduše vyparsovat. Toto parsování má právě na starosti funkce *filtrování výstupu* ve schématu 4.3. 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.

```

1 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:

3 '''robotframework
4 *** Settings ***
5 Library                SeleniumLibrary

7 *** 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 vypsát pouze text nebo *filtrace* neproběhla validně. Tento krok by v ideálním případě bylo možné automatizovat za pomoci *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

V této kapitole je diskutováno spuštění vygenerovaných testů a jde o implementaci kroku popisovaného v návrhu (3.2). Samotnému spuštění vybrané sady testů musí nejdříve předcházet nasazení testovaného softwaru, pro které zde byla vytvořena automatizovaná orchestrace využívající technologii *Docker*. Dále bylo také potřeba spustit samotné testy na vybraných varinátách testované aplikace. I o toto spuštění se stará orchestrační software. Popsána zde je také *konfigurace*, která je pro jednotlivé kroky potřeba, a jak jsou ukládány výsledky testů společně s jejich formátem.

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 nasazení 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 serverem* a druhý s *databází*.

Před samotnou integrací je nutné v každé z variant aplikace (WAR soubory) změnit *adresu* databáze. Docker kompozice umožňují přidělit jednotlivým kontejnerům názvy, kterými 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 vytvořit vlastní obraz, který bude vycházet z obrazu pro *Tomcat* aplikační server, ale vždy do něj budou nahrána data jiné varianty aplikace. Pro vytvoření 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 pomoci 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 nástrojem *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 přípona souboru naznačuje, 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

```
1 version: '3.8'

3 services:

5   db:
6     image: mariadb
7     container_name: tbuis-db
8     environment:
9       MARIADB_ROOT_PASSWORD: testtest
10    ports:
11      - "3306:3306"
12    volumes:
13      - ./db:/docker-entrypoint-initdb.d/
14    networks:
15      - tbuis

17   tomcat:
18     build:
19       context: .
20       args:
21         WAR_FILE: ${WAR_FILE_PATH}
22     container_name: tbuis-tomcat
23     depends_on:
24       - db
25     ports:
26       - "4680:8080"
27     networks:
28       - tbuis

30 networks:
31   tbuis:
32     driver: bridge
33
```

Pro vytvoření a spuštění této kompozice je potřeba zavolat příkaz z ukázky 5.3, kde příkaz `up` vyjadřuje právě *vytvoření* kompozice. Pro povolení sestavení ob-

razu 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

```
1 milan:dp/program$ WAR_FILE_PATH=./defect-00-free.war
   docker-compose -f tbuis/docker-compose.yml up -d --build
2
```

Výpis 5.4: Smazání Docker kompozice společně s vytvořenými obrazy

```
1 milan:dp/program$ docker-compose -f tbuis/docker-compose.yml
   down --rmi all
2
```

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 potřebná 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 spuštění testů

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

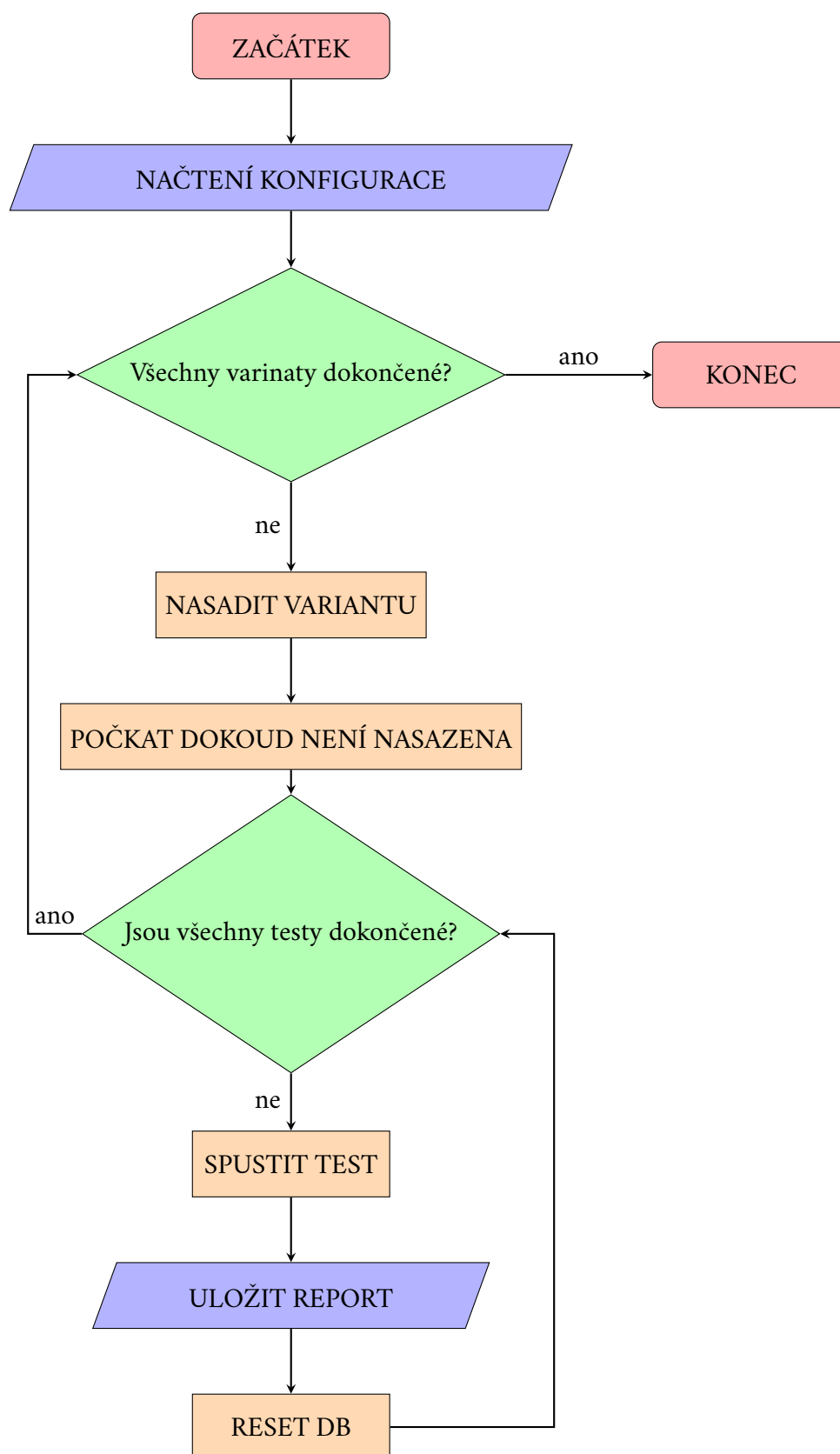
Spuštěcí režim programu nabízí možnost hned několika argumentů a jejich hodnot. Požadovaná hodnota je zde pro *režimový* argument `-r`, která musí obsahovat název nebo vzor názvu testů, které se v rámci složky *generated* mají spustit. Pro přehlednost si uživatel může ve výstupní složce testů rozdělovat 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

```
1 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* režim jsou již *nepovinné*, avšak alespoň argument `--name` je doporučený. Tímto argumentem lze nastavit jméno aktuálního běhu testů, kterým 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 doporučeno 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 *tbu*is přidají do pole *RUN_CONTAINERS* v rámci konfigurace.



Obrázek 5.1: Znázornění orchestrace spouštění testů

Zdrojový kód 5.7: Zjednodušená ukázka konfiguračního souboru

```

1 RUN_CONTAINERS = [
2     "defect-00-free.war",
3     "defect-02-C0.H0.M0.L1_S_S_03.war",
4     "defect-04-C0.H0.M0.L1_S_S_04.war",
5     ...
6     "defect-26-C1.H0.M0.L0_T_S_01.war",
7     "defect-27-C1.H0.M0.L0_U_D_01.war",
8     "defect-28-C2.H2.M1.L0_M_CR.war"
9 ]

11 POSITIVE_FAILS = {
12     "1": ["defect-02-C0.H0.M0.L1_S_S_03.war"],
13     "4": ["defect-04-C0.H0.M0.L1_S_S_04.war", "defect-19-C0.
14         H1.M0.L0_S_S_10.war", "defect-25-C1.H0.M0.L0_S_S_01.war",
15         "defect-26-C1.H0.M0.L0_T_S_01.war", "defect-28-C2.H2.M1.
16         L0_M_CR.war"],
17     "6": ["defect-25-C1.H0.M0.L0_S_S_01.war", "defect-26-C1.
18         H0.M0.L0_T_S_01.war", "defect-28-C2.H2.M1.L0_M_CR.war"],
19     ...
20 }

```

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 *error*y (kritické chyby, kvůli kterým nelze vygenerovaný test ani spustit). Protože tento výstup obsahuje spoustu redundantních informací, je v rámci programu zpracován a uložen na 2 samostatná místa. Z původního výstupu jsou vyjmuty 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 nešel spustit nebo při jeho běhu došlo k chybě nesouvisající se scénářem, bylo potřeno toto rozlišení implementovat na bázi přítomných *errorů* (jakožto značek) 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,

```

1  Report name: 20240411140043-GPT-4-Turbo run
2
3
4  Container name: defect-00_free.war
5  +-----+-----+-----+-----+-----+
6  | Name                               | Status | Passed | Failed | Errored |
7  +-----+-----+-----+-----+-----+
8  | openai-gpt45-turbo/spec-12-10.robot | FAIL   | 1      | 1      | 0      |
9  +-----+-----+-----+-----+-----+
10 | openai-gpt45-turbo/spec-1-8.robot   | FAIL   | 0      | 3      | 0      |
11 +-----+-----+-----+-----+-----+
12 | openai-gpt45-turbo/spec-11-7.robot  | PASS   | 1      | 0      | 0      |
13 +-----+-----+-----+-----+-----+
14 | openai-gpt45-turbo/spec-1-3.robot   | PASS   | 3      | 0      | 0      |
15 +-----+-----+-----+-----+-----+
16 | openai-gpt45-turbo/spec-18-4.robot  | PASS   | 1      | 0      | 0      |
17 +-----+-----+-----+-----+-----+
18 | openai-gpt45-turbo/spec-12-1.robot  | FAIL   | 1      | 1      | 0      |
19 +-----+-----+-----+-----+-----+
20 | openai-gpt45-turbo/spec-10-7.robot  | FAIL   | 0      | 2      | 0      |
21 +-----+-----+-----+-----+-----+
22 | openai-gpt45-turbo/spec-12-4.robot  | FAIL   | 0      | 2      | 0      |
23 +-----+-----+-----+-----+-----+

```

Obrázek 5.2: Ukázka textové tabulky výsledků

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 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 výsledků 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.

	id	time	name	container	test_name	result	123 success	123 fail	123 error
1	111	2024-04-16 20:35:50	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-12-10.robot	PASS	2	0	0
2	112	2024-04-16 20:36:02	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-1-8.robot	FAIL	0	4	0
3	113	2024-04-16 20:37:11	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-11-7.robot	PASS	1	0	0
4	114	2024-04-16 20:37:36	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-1-3.robot	FAIL	0	3	0
5	115	2024-04-16 20:37:58	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-18-4.robot	FAIL	0	1	0
6	116	2024-04-16 20:38:02	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-12-1.robot	FAIL	0	2	0
7	117	2024-04-16 20:38:11	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-10-7.robot	FAIL	0	2	0
8	118	2024-04-16 20:38:17	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-12-4.robot	FAIL	1	1	0
9	119	2024-04-16 20:38:23	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-9-7.robot	PASS	1	0	0
10	120	2024-04-16 20:38:49	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-6-7.robot	PASS	2	0	0
11	121	2024-04-16 20:38:58	20240416223456-Claude 3 Opus runs	defect-00_free.war	anthropic-claude-3-opus/spec-8-2.robot	FAIL	0	2	0

Obrázek 5.3: Ukázka databázové tabulky výsledků

Vyhodnocení výsledků

6

V rámci této kapitoly jsou *analyzovány* výsledky testů, které byly pro účely této práce *vygenerované a spuštěné* skrze metody popsané v předchozích kapitolách. Popsána je zde kompletní metodologie, která pro vyhodnocení výsledků byla využita a také jsou popsány tabulky výsledků. V rámci analýzy se budeme zaměřovat primárně na kvalitativní stránku vygenerovaného zdrojového kódu testů. V závěru kapitoly je také rychlé srovnání s lisky psanými testy, které jsou k testované aplikaci k dispozici a je diskutována i náročnost generování na zdroje (*finance, čas, energie*).

6.1 Parametry pro spuštění

Vygenerované testy z kroku 4 (konkrétně scénáře vypsány 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 popsána v sekci 5.1.2. Tyto vygenerované testy, využívané k následnému testování poruchových klonů, jsou součástí programového repozitáře projektu. Do konfiguračního souboru byly 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.

Výpis 6.1:

1	15	milan	staff	480	May	5	09:59	.
2	19	milan	staff	608	May	6	22:38	..

3	102	milan	staff	3264	Apr	30	15:07	anthropic-claude-3-opus
4	102	milan	staff	3264	Apr	16	07:40	codellama
5	102	milan	staff	3264	May	4	09:45	gemini-1.5-pro
6	102	milan	staff	3264	May	3	18:52	local-llama3
7	102	milan	staff	3264	Apr	30	15:07	mistral-large
8	102	milan	staff	3264	May	5	11:05	mistral-v0.2
9	3	milan	staff	96	May	5	09:59	openai-gpt35
10	102	milan	staff	3264	Apr	30	15:07	openai-gpt4
11	102	milan	staff	3264	May	3	18:52	openai-gpt4-turbo
12	102	milan	staff	3264	May	5	09:59	wizardcoder-python

6.2 Metodologie vyhodnocení

Následující text v krátkosti popisuje, jaké statické metody a postupy jsou k vyhodnocení výsledků využity a na jakých předpokladech pro výstupy automatizovaného testování je vytvořený statistický nástroj postaven, případně jaké parametry od uživatele vyžaduje. Společně s metodou je vysvětlena i vizualitace a přidán je zde také důkaz o předpokladu pro pravá a případně falešná negativa, která jsou v rámci metody detekována. Hodnotit se bude, zda zaznamenané výstupní stavy testů odpovídají předpokladu.

6.2.1 Vyhodnocovací program

Pro vyhodnocení výsledků byl do programové složky vytvořen program `stats.py`, který jako první argument vyžaduje cestu s `.sqlite` výstupnímu reportu hlavního programu v režimu běhu (viz sekce 5.1.2). Ukázkové volání programu je zobrazeno v ukázce 6.2. Tento report si uživatel může libovolně přesouvat v souborovém systému. Reporty vygenerované v rámci tohoto projektu a využité pro hodnocení modelů byly uloženy do podsložky `results` kořenové složky repozitáře. Program má i další argumenty, které jsou jeho klíčovou součástí:

- `-t [nadpis]` - Nastaví nadpis nad tabulkou (název modelu).
- `-e [navez_souboru]` - Vyexportuje soubor do PDF. Hodnotou je název výstupního PDF souboru (s příponou).

Pro každý test je vyhodnoceno, kolik z jeho vygenerovaných variant pro každou variantu testovacího programu skončilo s *očekávaným výsledkem*. Tím je zde myš-

leno, že test prošel úspěšně na variantách aplikace, kde se nevyskytuje chyba a je tak očekáván bezchybný průchod, a že test selhal v na takových variantách, kde je definováno specifikací (diskutováno v sekci 6.1), že by selhat měl. Chybový test je vždy považován za *neočekávaný výsledek*. Při takto naivním přístupu by však jako *očekávané* byly označovány i testy (resp. jejich vygenerované varianty) chybující i v případě nasazených variant aplikace bez konkrétní testované chyby (tedy falešná negativa). Z důvodu, aby takové testy nezkreslovali výsledky, bylo přistoupeno ke kroku jejich vyřazení a následnému ignorování při vyhodnocování výsledků.

Pro odhalení falešných negativ budeme využívat předpokladu výsledku na *neporuchové* variantě aplikace, že varianta testu skončí úspěšně. V případě, kdy dojde k jejímu selhání nebo skončí chybou, je taková testová varianta ignorována při vyhodnocení výsledků testů na všech poruchových kontejnerech, protože se předpokládá, že takový test není schopen správně odhadit chybu aplikace, protože by chybový stav hlásil vždy. Počet ignorovaných variant od každého testu je zobrazen ve spodní řádce tabulky výsledků (více v následující sekci 6.2.2).

Výpis 6.2: Volání programu pro vyhodnocení dat.

```
1 milan:dp/program$ python3 stats.py
   ../results/claude-3-opus-run.sqlite -t "Claude 3 Opus" -e
   claude-results.pdf
```

6.2.2 Legenda tabulek

Výstupem vyhodnocení výsledků je tabulka (jako příklad poslouží výstup v obr. 6.1), jejíž řádky reprezentují testované *varianty aplikace* (v tabulce označované jako *container*) a jednotlivé sloupce představují konkrétní test (dle specifikace), který byl spouštěn. Jednotlivé buňky poté obsahují informaci o počtu variant daného testu, které vyhovují předpokládanému výsledku pro danou variantu aplikace. Tyto hodnoty jsou po vyfiltrování, které se vždy děje dle první řádky od shora (v našem označení jde o variantu `defect-00_free_war`). Tento přístup byl popsán v předchozí sekci 6.2.1. Ve spodním řádku tabulky je také vyznačeno, kolik variant daného testu bylo *ignorováno*. Celá tabulka zároveň funguje jako *tepelná mapa* a je tedy nejen číselně, ale i barevně vyznačeno, jaký test má pro kterou variantu aplikace nejvyšší míru úspěšnosti (shody s předpokládaným výsledkem). Barevná škála a její odpovídající hodnoty se nachází po pravé straně tabulky.

6.2.3 Předpoklady

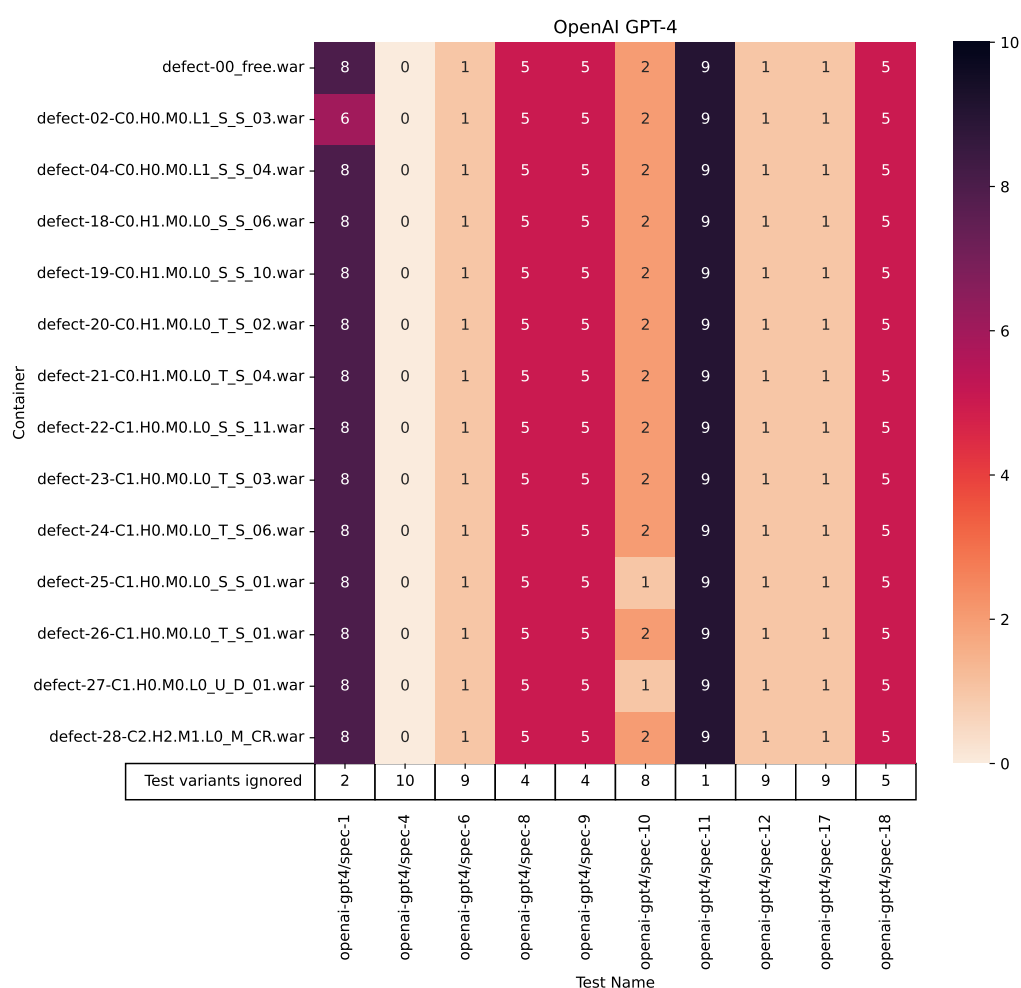
U variant testované aplikace, které jsou označeny v tabulkách 4.2 a 4.2 jako poruchové pro danou specifikaci je očekáváno *selhání* testu a pokud toto selhání při spuštění testu na takové variantě detekujeme, dostáváme korektně negativní výsledek. V ostatních případech variant aplikace považujeme za očekávaný výsledek testu jeho *úspěšný průchod* a takový výsledek je brán za korektně pozitivní. Ostatní výsledky jsou poté považovány za falešně negativní či pozitivní. Zatímco falešná pozitiva nelze odhalit při průchodu *neporuchovou* variantou aplikace, tak falešná negativa je takto možné odhalit, protože při této kombinaci je očekáván vždy průchod testu bez selhání. Mimo tuto klasifikaci se poté řadí testy, které jsou chybové a není je možné spustit nebo dokončit jejich chod a obvykle končí chybovým kódem. Selhání jednotlivých testových případů na konkrétních variantách aplikace, jak je definuje dokumentace projektu, je ověřeno *akceptačními testy* vzniklými v práci [50]. Mimo toho bylo v úvodu na testech vygenerovaných modelech *GPT-4* otestováno jejich spuštění na všech 29 variantách, které projekt TbUIS nabízí a jejich výsledky jsou dostupné ve složce výsledků pro každou ze specifikací samostatně jakožto soubory `test-<číslo specifikace>-all.sqlite`.

6.3 Výsledky pro jednotlivé modely

Tato sekce je rozdělena do kategorií dle původního vývojáře modelu, ve kterých jsou poté postupně popisovány výsledky jednotlivých modelů (tabulky) a je analyzována jejich struktura a kvalita výstupního kódu. V rámci kvality se zaměřujeme jak na validní tak chybné úsudky, které model predikoval.

6.3.1 OpenAI

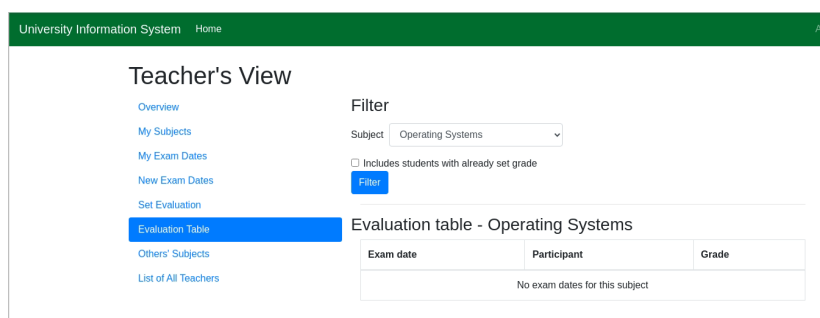
Model **GPT-4** od společnosti OpenAI dokázal pro většinu *testových případů* vygenerovat minimálně *jeden* test podávající pro všechny testované varianty aplikace očekávaný výsledek. Výsledky zobrazené v obr. 6.1. Jediný test, pro který model *GPT-4* nebyl schopen vygenerovat ani jednu variantu testu, která by podala očekávaný výsledek, pro specifikaci 4 (viz 4.2). Tento konkrétní testový případ popisuje odhlášení studenta ze zkoušky a následnou kontrolu učitelem, zda je seznam studentů, přihlášených na zkoušku, prázdný. Každá z vygenerovaných variant hlásila pro tento případ poruchu i na kloněch, které by ji pro tento případ neměli vykazovat. Primárním probléme ve vygenerovaném kódu zde byla nesprávná identifikace



Obrázek 6.1: Výsledky pro model GPT-4

prvků, které mají být stisknuty nebo měla být detekována jejich hodnota následně srovnána s očekávanou. V případě studenta častokrát nedochází k úspěšnému odhlášení ze zkoušky, což vede i k následnému selhání *test casu* ze strany učitele. Na této straně zároveň docházelo k nesprávnému výběru předmětu z rozbalovacího seznamu. Například v obr. 6.2, kde správně má být zvolený předmět „Database Systems“.

V případě testu dle specifikace 1 dochází u varianty aplikace 02 (dle tab. 4.2) k nedetekování poruchy, která po analýze 2 testů, u kterých se tento nedostatek vyskytl, je dána absenci detekce nadpisu a jeho obsahu, jehož kontrolu vyžaduje vstupní prompt, který byl využit (viz vstup spec-01.text v repozitáři projektu). Zároveň právě *nesprávný nadpis* je poruchou, které se v klonu 02 vyskytuje. Z tohoto



Obrázek 6.2: Nesprávně vybraný předmět z rozbalovacího seznamu.

důvodu vychází tato dvojice vygenerovaných testů s kladným výsledkem, i přesto že nasazená varianta aplikace obsahuje namísto hodnoty nadpisu „Student’s View“ hodnotu „Stu View“. U testu 10 poté jedna ze dvou validních variant neodpovídá předpokládanému výsledku. Tato konkrétní varianta končí v neúspěšném stavu u nasazené varinaty aplikace 25 (tabulka studentů prázdná), protože server vrací chybový kód 500 a nejedná se tedy o chybu testu. V rámci varinaty 27 naopak testovaný web aplikace vypisuje prázdnou tabulku učitelů, kdy test vyhledává pole v tabulce a porovnává jeho hodnotu. V tomto případě však nedokáže ono pole lokalizovat. Tento konkrétní test (konkrétně jde o soubor `openai-gpt4/spec-10-3.robot`) tedy není chybný, ale v jednom případě se jedná o odlišnou chybu testované aplikace a v druhém případě jde o nedostatečnou granularitu specifikace testu a pouze v závislosti na její reprezentaci jde varianta 27 aplikace také považovat za chybovou. Obě nevyfiltrované varianty testu 10 mohou být považovány za validní na všech verzích testovaného webu.

Naopak model **GPT-4 Turbo** dokázal v některých případech, kde model GPT-4 selhával, vygenerovat validní testy, avšak pro 3 testové specifikace nevygeneroval ani jeden test, který by prošel filtračním kritériem (viz obr. 6.3). Jde konkrétně o testy 6, 10 a 12. U testů dle specifikace 6 dochází k nesprávnému indexování sloupce tabulky, kdy druhý prvek `<td>` v rámci řádku `<tr>` získává za pomoci dotazovacího jazyka XPath indexem 1, kdežto druhý prvek má v rámci tohoto jazyka index 2. Tuto chybu lze vidět v datazu ukázky 6.3 na řádcích 23 a 24. V případě testu 10 dochází k nesprávné interpretaci specifikace (resp. *promptu*), která říká, že všechna tlačítka na dané stránce mimo tlačítka stisknutého v nahrávce musí být *deaktivovaná* (HTML argument `disabled` s hodnotou `"true"`). GPT-4 Turbo vyhledává tato tlačítka nekorektně jako např. dceřiné prvky tlačítka stisknutého v nahrávce (v ukázce 6.4), zatímco správný odhad provedl model GPT-4 (v příkladu 6.5), který usoudil, že všechna tlačítka budou mít stejný základ identifikátoru `id` a pouze tlačítko se shodným identifikátorem z nahrávky bude *aktivní*.

Zdrojový kód 6.3: Ukázka z vygenerovaného testu "openai-gpt4-turbo/spec-6-4.robot"(zbytek kódu vynechán).

```

1 ...
2 Student Enrollment and Verification
3   Open Browser      http://localhost:4680/tbuis/index.jsp
   chrome
4   Set Window Size   1501    1104
5   Click Element     xpath=//*[@id="header.link.login"]
6   Sleep             2s
7   Click Element     xpath=//*[@id="loginPage.userNameInput"]
8   Sleep             1s
9   Input Text        xpath=//*[@id="loginPage.userNameInput"]
   maroon
10  Press Keys        None     TAB
11  Sleep             1s
12  Input Text        xpath=//*[@id="loginPage.passwordInput"]
   pass
13  Press Keys        None     ENTER
14  Sleep             3s
15  Click Element     xpath=//*[@id="stu.menu.otherSubjects"]
16  Sleep             2s
17  Click Element     xpath=//*[@id="stu.otherSubjects.table.
enrollButton-10"]
18  Sleep             2s
19  Page Should Contain Element    xpath=//*[@id="stu.
otherSubjects.successAlert"]
20  Sleep             2s
21  Click Element     xpath=//*[@id="stu.menu.mySubjects"]
22  Sleep             2s
23  Element Should Contain    xpath=//tr[@id="stu.mySubjects.
enrolledTable.subjectRow-2"]/td[1]    Software Quality
Assurance
24  Element Should Contain    xpath=//tr[@id="stu.mySubjects.
enrolledTable.subjectRow-2"]/td[2]    Peter Strict
25  Close Browser
26 ...
27

```

Zdrojový kód 6.4: Klíčové slovo z testu "openai-gpt4-turbo/spec-10-10.robot".

```

1 ...
2 Validate My Subjects Page
3   ${disabled_buttons}=    Get Element Count    xpath=//*[@
id="tea.mySubjects.table.unregisterSubjectButton-0"] [
@disabled='disabled']
4   Should Be Equal As Integers    ${disabled_buttons}    1
5   Click Element    xpath=//*[@id="tea.mySubjects.table.
unregisterSubjectButton-0"]

```

```

6     Sleep      1s
7     Element Should Be Visible      xpath=//*[@id="tea.
    mySubjects.successAlert"]
8 ...
9

```

Zdrojový kód 6.5: Příklad správné implementace specifikace 10 dle modelu GPT-4.

```

1 Navigate To My Subjects And Verify
2     Click Element      xpath=//*[@id="tea.menu.mySubjects"]
3     Sleep      1s
4     ${remove_buttons}=    Get WebElements      xpath=//button[
    contains(@id, 'unregisterSubjectButton') and not(@id='tea.
    mySubjects.table.unregisterSubjectButton-0')]
5     FOR      ${button}    IN      @{remove_buttons}
6     Element Should Be Disabled      ${button}
7     END
8     Click Element      xpath=//*[@id="tea.mySubjects.table.
    unregisterSubjectButton-0"]
9     Sleep      1s
10    Element Should Be Visible      xpath=//*[@id="tea.
    mySubjects.successAlert"]
11

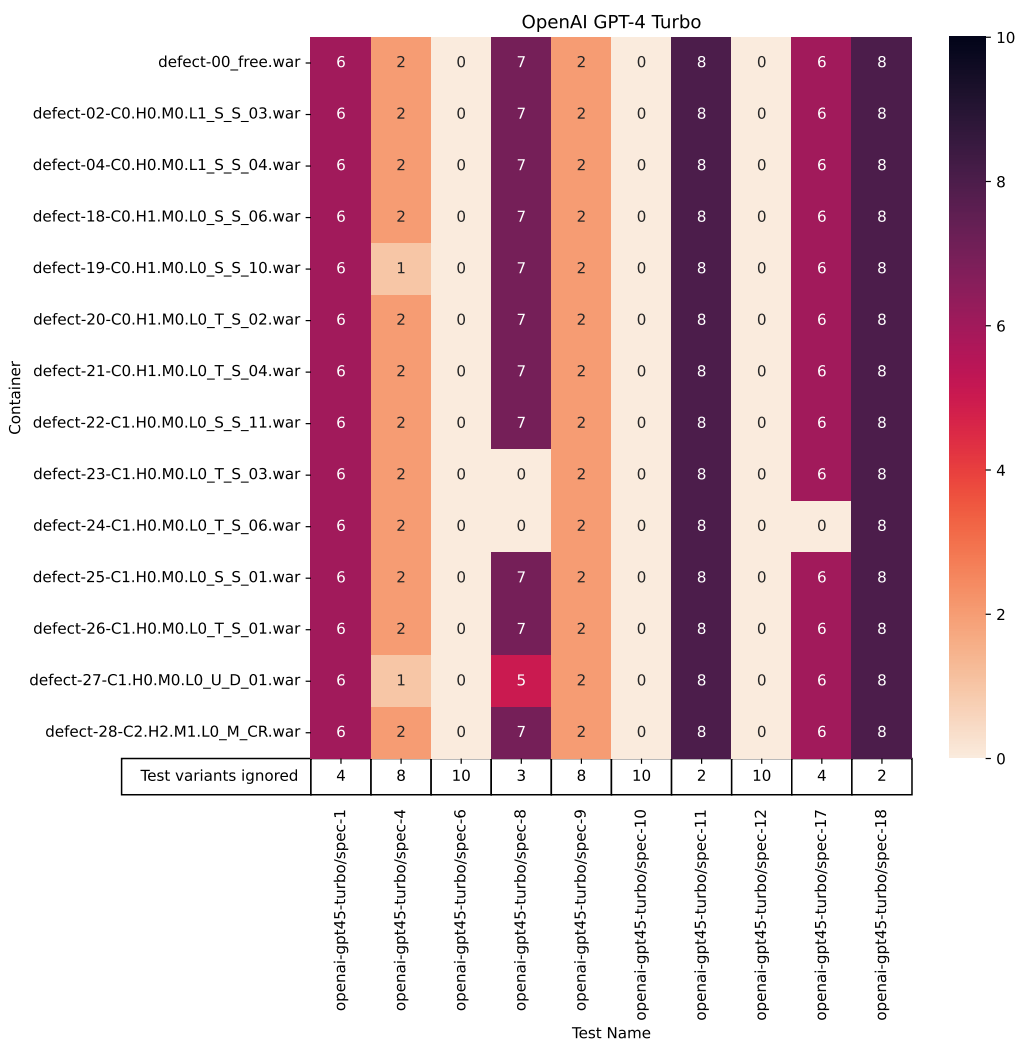
```

V testu 12 se objevuje požadavek odsouhlasit vyskakovací upozornění (*alert*). Ten výstup modelu GPT-4 Turbo nesprávně odsouhlasuje za pomoci klíčového slova `Accept Alert`, kdežto správné klíčové slovo je `Handle Alert`¹ jako jde vidět například v testu `openai-gpt4/spec-12-10`. Ze 7 validních varint testu 8 selhaly veškeré na poruchových klonech 23, 24 a na klonu 27 pouze 5 z nich skončilo očekávaným výsledkem. Zde je nutné připomenout, že ani na jednom ze zmíněných klonů (dle tab. 4.2) by neměl test skončit neúspěšně.

V případě specifikace 17 žádný z 6 testů určených jako validní, nedokázal ojevit chybu na kontejneru 24 a neselhal. Zde se učitel zapíše k výuce nového předmětu, což mu dá hlášku o úspěšném zapsání, avšak do databáze se tento fakt nezapíše a na ostatních stránkách se uživatel nenachází v seznamu učitelů pro daný předmět.

Model **GPT-3.5 Turbo** dokázal vygenerovat validní testy pouze pro testové scénáře 11 a 18, v celkově třech varintách (obr. 6.4). Tyto varinaty testů však dokázali odhadit veškeré zavedené chyby. Zbytek vygenerovaných variant pro veškeré testy obsahoval selhával na využitím klíčového slově `Set Viewport`, kterým test má dle

¹Dle dokumentace, dostupné na adrese: <https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html#Handle%20Alert>

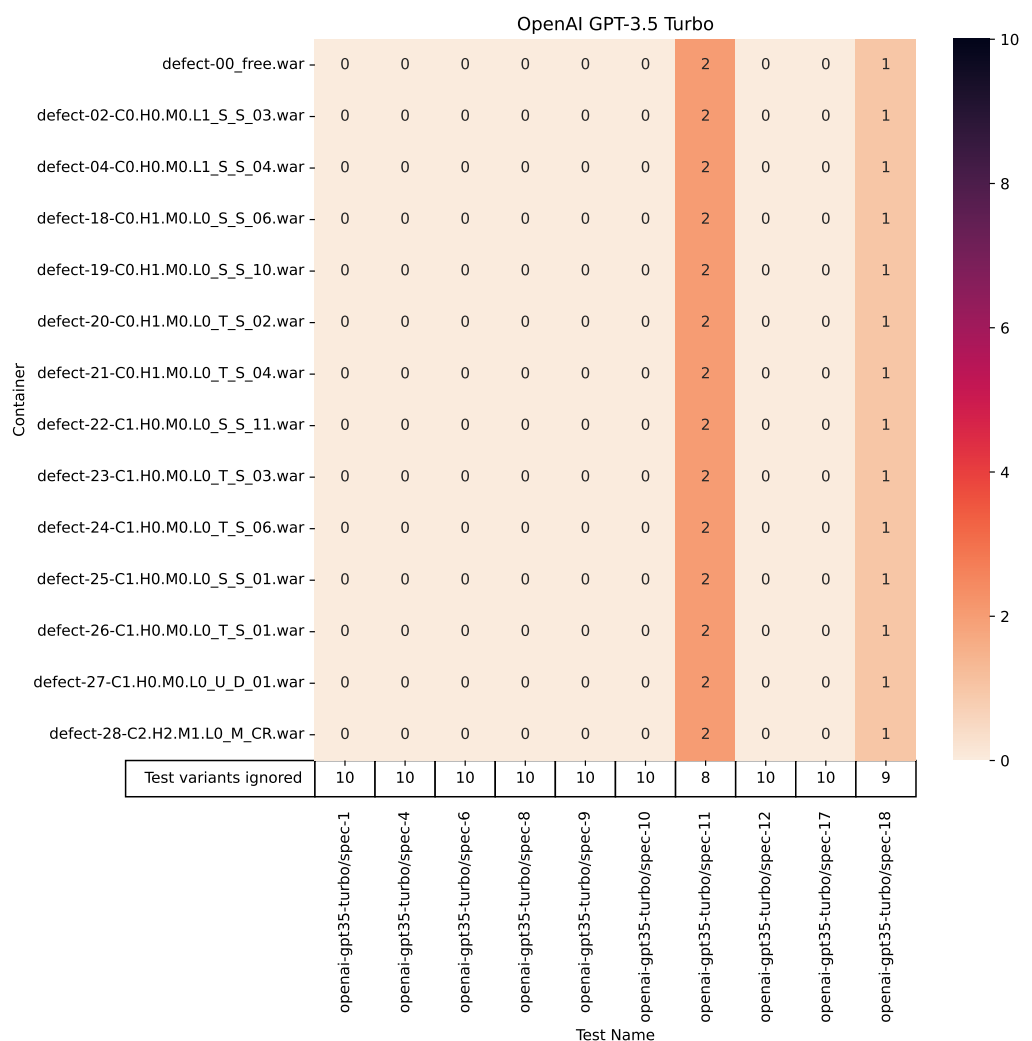


Obrázek 6.3: Výsledky pro model GPT-4 Turbo

modelu nastavovat velikost okna webového prohlížeče. Správným klíčovým slovem pro toto použití je Set Window Size.

6.3.2 Anthropic

Testovaný model **Claude 3 Opus** od společnosti Anthropic byl schopen pro každý test vygenerovat nejméně jednu variantu, která je schopna odhladit každou chybu vloženou do systému (obr. 6.5). Mezi nejméně úspěšné testy se zde řadí varianty dle specifikace 12 a 17. Časté chyby variant jsou například nesprávné adresování prvků za pomoci *XPath* dotazovacího jazyka, kde namísto počátku selektoru //, který vy-



Obrázek 6.4: Výsledky pro model GPT-3.5 Turbo

hledává daný prvek kdekoliv v dokumentu, resp. v poskytnutém kořenovém prvku, využívá nesprávný selektor /// (příklad 6.6). Správné použití lze vidět v ukázce 6.5. Pozn: Pro oddělení názvu argumentu a jeho hodnoty lze v rámci *RobotFramework* využít jak znak : tak =, které lze vidět v obou verzích využívané napříč ukázkami.

Zdrojový kód 6.6: Ukázka nesprávného adresování XPath.

```
1 Teacher Scenario
2   Open Browser      http://localhost:4680/tbuis/index.jsp
   Chrome
3   Set Window Size   1501      1104
4   Click Element     xpath:///*[@id="header.link.login"]
5   Wait Until Page Contains Element  xpath:///*[@id="
```

```

loginPage.userNameInput"]
6   Input Text      xpath:///*[@id="loginPage.userNameInput"]
   scatterbrained
7   Input Password  xpath:///*[@id="loginPage.passwordInput
   "]    pass
8   Click Button    xpath:///*[@id="loginPage.loginFormSubmit
   "]
9   Wait Until Page Contains Element  xpath:///*[@id="tea.
   menu.myExamDates"]
10  Click Element   xpath:///*[@id="tea.menu.myExamDates"]
11  Wait Until Page Contains Element  xpath:///*[@id="tea.
   myExamDates.table.cancelButton-0-0"]
12

```

Dalším problémem také bylo vybírání prvku z rozbalovací nabídky za pomoci hodnoty `|value|` prvku `option`, které se však při každé inicializaci DB mění a tedy nelze se na tyto hodnoty spolehnout. Z tohoto důvodu bylo do instrukcí pro prompt specificky přidáno, aby model vybíral položku v těchto seznamech za pomoci obsahu prvku `option` (lze vidět v ukázce 6.8 na řádce 6 a 7). Chybová zpráva o nenalezeném prvku je zobrazena v ukázce 6.7. V některých testech se také objevil defekt, kdy model namísto názvu knihovny `SeleniumLibrary` použil název `Selenium2Library`, což vede k *erroru* testu. Podobně jako u modelu GPT-4 i zde se objevuje nesprávná indexace prvků v rámci XPath. Některé z testů také obsahovali neplatná klíčová slova jako `Page Should Be` nebo `Element Text Should Match`.

Výpis 6.7: Ukázka selhání testu po nenalezeném prvku s danou hodnotou.

```

1 milan:dp/program$ python3 -m robot
  ../generated/anthropic-claude-3-opus/spec-12-2.robot
2 =====
3 Spec-12-2
4 =====
5 Teacher Scenario | FAIL |
6 NoSuchElementException: Message: Cannot locate option with
  value: 1202; For documentation on this error, please
  visit:
  https://www.selenium.dev/documentation/webdriver/...
7
8 Student Scenario | PASS |
9
10 Spec-12-2 | FAIL |
11 2 tests, 1 passed, 1 failed
12 =====
13 Output: /Users/milan/Škola/DP/program/tbuis/output.xml
14 Log: /Users/milan/Škola/DP/program/tbuis/log.html
15 Report: /Users/milan/Škola/DP/program/tbuis/report.html

```

16

Zdrojový kód 6.8: Ukázka promptu dle testového scénáře 10.

```

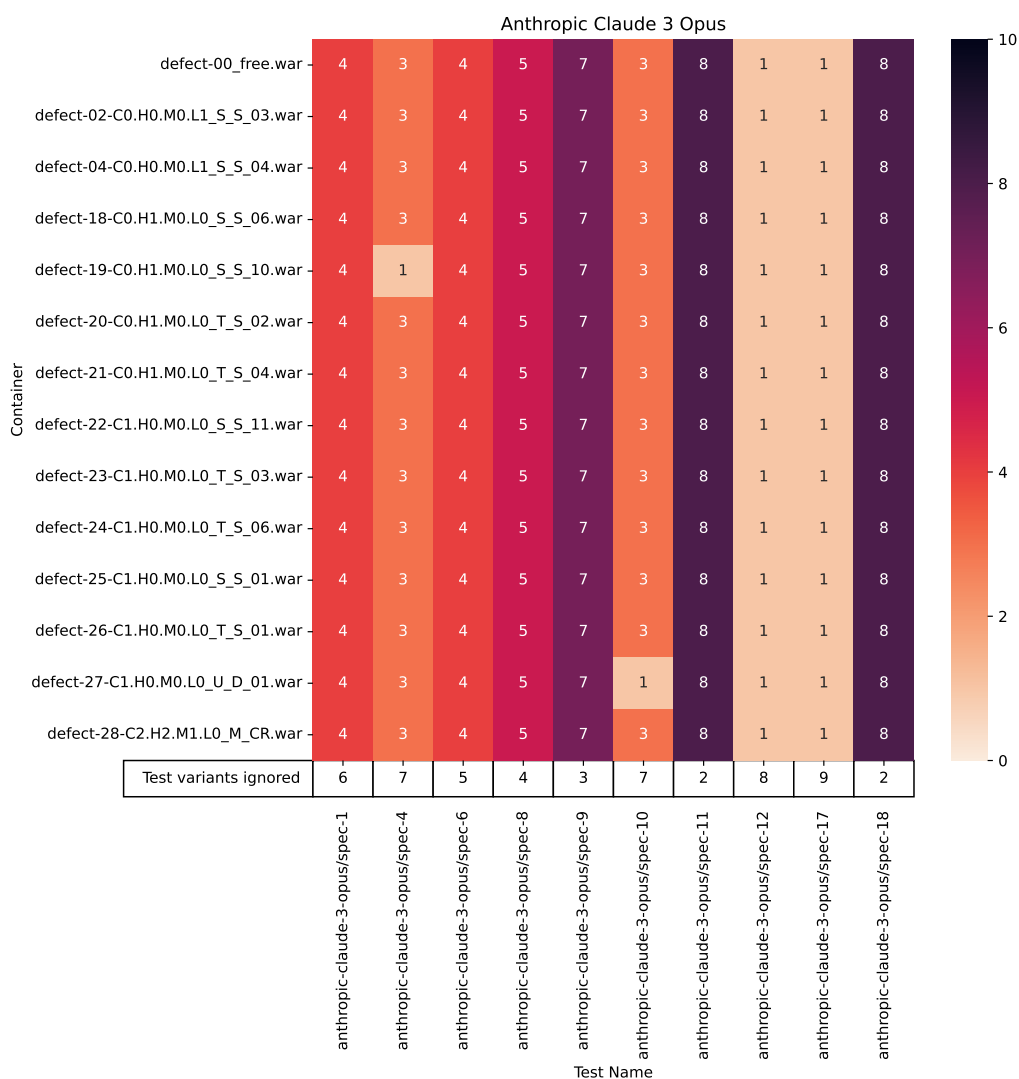
1 Write Robot Framework scenario. Open page like in this JSON
  recording and then when you execute all the steps in the
  recording, do this:

3 — On page "My Subjects" before clicking the "Remove" button
  check if element all the other button with value "Remove"
  are disabled
4 — On page "My Subjects" after clicking the "Remove" button
  check if element with id "tea.mySubjects.successAlert" has
  appeared
5 — On page "My Exam Dates" check if element <th> with text "
  Operating Systems" is not present
6 — On page "New Exam Dates" check if element <option> with
  text "Operating Systems" is not present
7 — On page "Set Evaluation" check if element <option> with
  text containing string "Operating Systems" is not present
8 — On page "Evaluation Table" check if element <option> with
  text "Operating Systems" is not present
9 — On page "Other's Subjects" check if element <td> with text
  "Operating Systems" is present
10 — On page "List of All Teachers" check if inside element <tr>
  with id "tea.listOfAllTeachers.table.teacherRow-5" there
  is not <td> element containig string "Operating Systems"

12 {% include 'recording-spec-10-teacher.json' %}
13 ...

```

Pro specifikaci 4 jsou indikovány 2 z validních testů jako varianty, nesplňující předpoklad. Chyba tohoto nasazeného klonu 19, na kterém se však tato nedokonalost projevuje, spočívá v náhodně mizejícím tlačítku při odepsání předmětu, avšak toto tlačítko je při každém načtení stránky jiné a tedy nemusí se jednat přesně o tlačítko vyžadované pro odhlášení konkrétního předmětu, který vyžaduje přiložená nahrávka promptu. V tomto případě se tedy jedná o správný výsledek, protože jak úspěšný tak neúspěšný výstup testu zde mohou být validní a záleží jen na tom, jak test selže. Stejně jako v případě modelu GPT-4 i zde validní testy dle specifikace 10 selhávají na kontejneru 27, protože celá tabulka není vykreslena a selektory využívající její přítomnosti pro nalezení hledaného řetězce selhávají.



Obrázek 6.5: Výsledky pro model Claude 3 Opus

6.3.3 Meta

Codellama Instruct 34B byl jedním z modelů testovaných *lokálně*, ale bohužel pro polovinu testových případů nebyl schopen vygenerovat žádný validní test a pro druhou polovinu pouze jeden či dva použitelné testy, častokrát neshopné odhalit vloženou chybu (obr. 6.6). Pro první test model v několika z nevalidních variant vyhalucinoval test připomínající ověření uživatelského rozhraní před samotným přihlášením (viz test `codellama/spec-1-2.robot`). Zároveň ve dvou testech, označených za validní, nezkontrolovat hodnotu nadpisu podobně jako modely v sekci

6.3.1. Oba tyto testy také nejsou kompletní, protože neobsahují všechny 4 scénáře, které by měli obsahovat (různé přihlašovací údaje), ale pouze jeden nebo dva. Při manuálním srovnání se tedy ani jedna z vygenerovaných variant testu 1 nedá považovat za správnou. V některých z variant testů se také model rozhodl pro identifikaci prvku využít jeho "aria" atribut, který v rámci knihovny *SeleniumLibrary* není podporovaný jako selektor, resp. byla by potřeba využít CSS selektor. V případě varianty testu 4 naopak nebyla dodržena pomalejší rychlost testování (resp. vkládání pauz), které je promptem vyžadováno a tedy ověření přítomnosti prvku selhává, protože ještě není načtení a ani nebyla zvolena žádná alternativní metoda, která by na jeho načtení vyčkala. Pro generování varianty testu 6 docházelo ke stejnému problému jako v případě modelu *GPT-4 Turbo* a to nesprávné indexaci v rámci *XPath*. V neplatných variantách testu 8, 9 a dalších se objevuje stejný problém jako v případě testu 1, tedy chybějící časování kroků. V jednom z testů (*codellama/spec-8-9.robot*) také chyběl import knihovny. Stejně jako v případě modelu *GPT-3.5 Turbo* docházelo (alespoň při testu 9) k nesprávnému nastavení velikosti okna neexistujícím klíčovým slovem.

V případě testu 10 jediná varianta, značená jako „validní“ ve skutečnosti obsahuje pouze příkaz pro otevření *přihlašovací URL* a zbytek souboru jsou pouze komentáře (ukázka ve výpisu 6.9). I ostatní varianty dle této specifikace obsahují prázdné či nesmyslné testy. To stejné se vztahuje i na test 11, kde častokrát chybí klíčové části kódu (např. *login*). Výstupy pro specifikaci 12 obsahují stejné chyby jako předchozí testy. Vygenerované varianty testu 17 obsahují dvě varianty procházející filrem, avšak jen jedna z nich je po prozkoumání platný test. Druhá varianta obsahuje pouze otevření dané URL a tedy není schopna odhalit problémy, jak také napovídají výsledky na poruchových klonech. Platný test však z neznámých důvodů na poruchové variantě 18 neselhává. U posledního testu 18 vyšla 1 platná varianta, ale podobně jako v předchozích případech ani ta není kompletní a obsahuje pouze zobrazení tabulky, nikoli už její kontrolu.

Zdrojový kód 6.9: Ukázka chybně vygenerovaného testu "codellama/spec-10-7.robot".

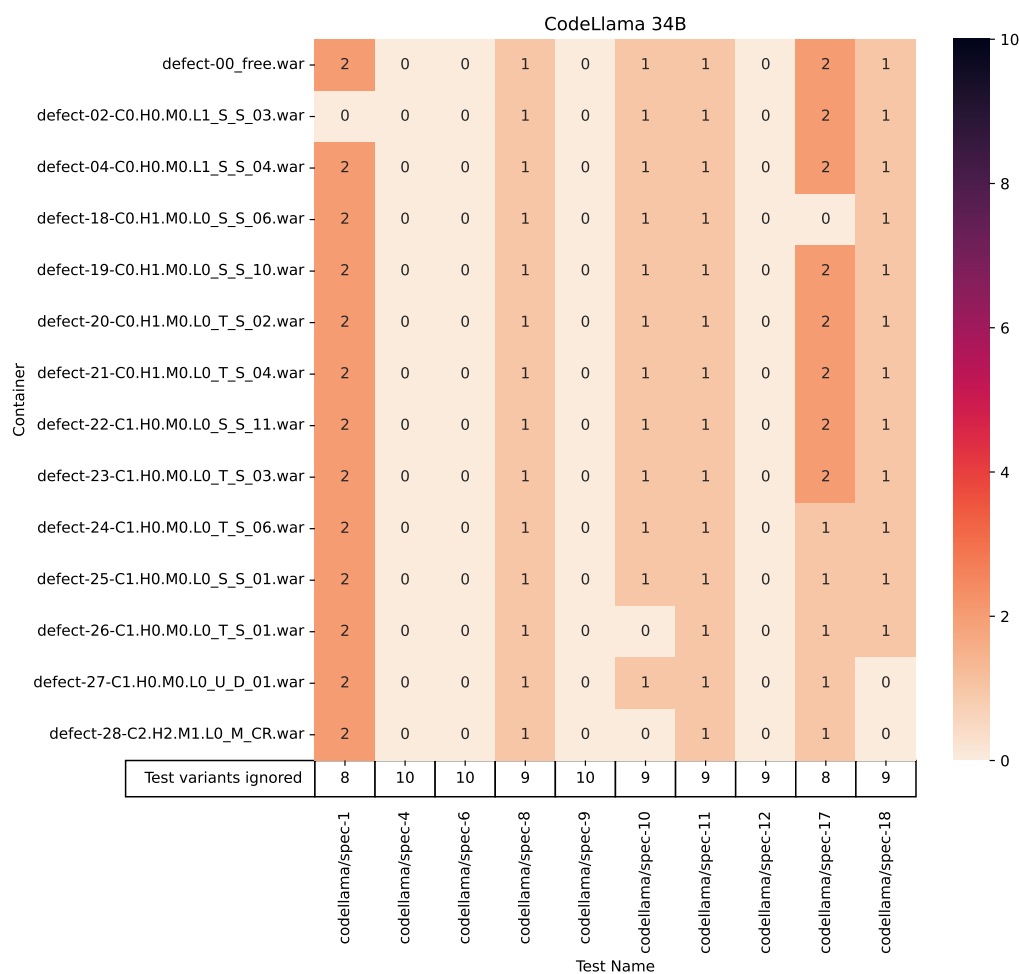
```

1 *** Settings ***
2 Library                               SeleniumLibrary

4 *** Variables ***
5 ${URL}                               http://localhost:4680/tbuis/index.jsp

7 *** Test Cases ***
8 Teacher Test
9     Open Browser    ${URL}    Chrome
10    # Teacher steps go here

```



Obrázek 6.6: Výsledky pro lokální model Codellama Instruct 34B

```

11  # Teacher's steps from the provided JSON have been
12  # replaced with comments due to the character limit
13  # Assert elements are present/not present
14  # Assert element with id "tea.mySubjects.successAlert"
15  # has appeared
16  # Assert elements are disabled
17  # Close Browser

```

Model **Llama 3** ve verzi se 70 miliardy parametrů vygeneroval funkční nebo splňující filtry pouze 2 varianty testu a to pro specifikaci 10. Oba testy (a to konkrétně llama3/spec-10-3 a llama3/spec-10-6) se i po manuálním přezkoumání zdají být validní, avšak nesplňují přesně požadavky promptu na časování mezi jed-

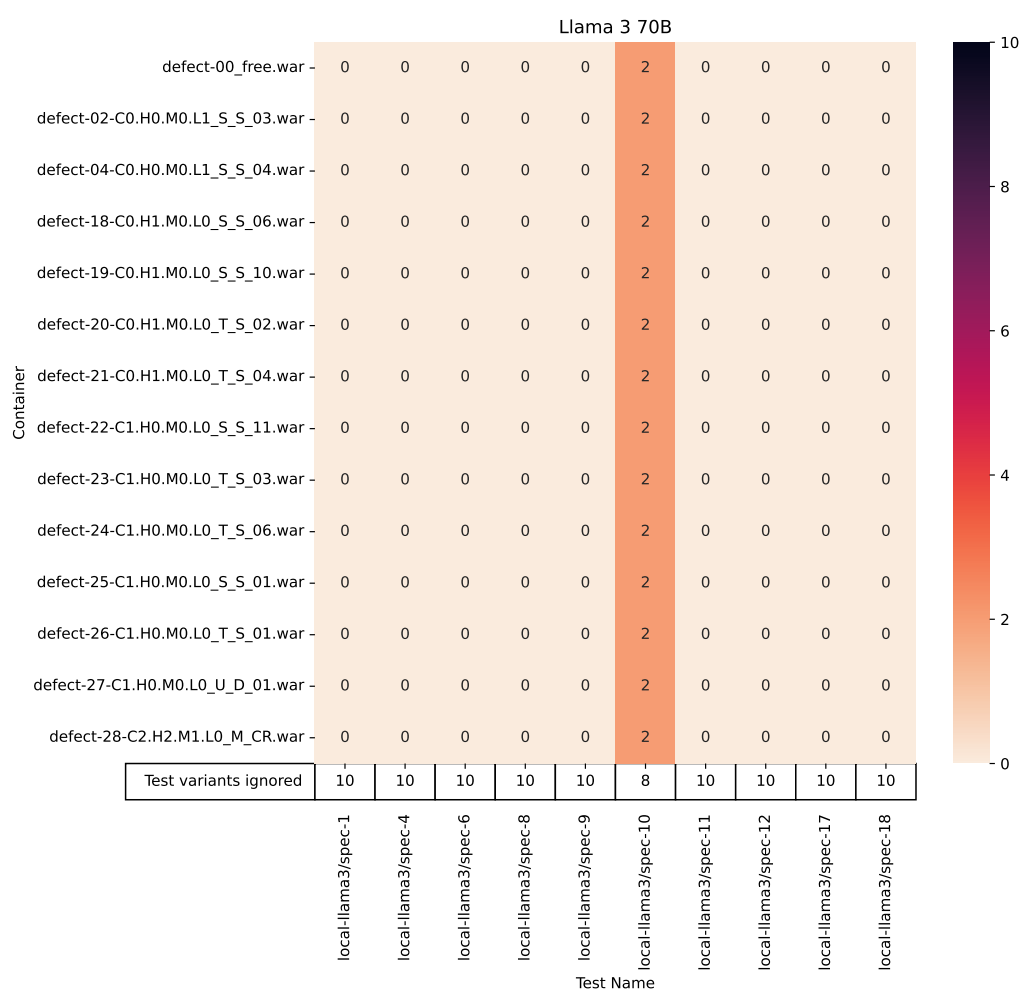
notlivými kroky. Mezi časté chyby tohoto modelu se řadí například *špatné pořadí argumentů pro otevření webového prohlížeče*, kdy pro klíčové slovo `Open Browser` uvedl jako první argument URL a jako druhý název prohlížeče, kdežto správné pořadí argumentů je opačně. Dále byla existence prvků ověřována klíčovým slovem `Element Should Exists`, které však neexistuje a správně by měla být kontrolována použitím klíčového slova `Page Should Contain Element`. V některých případech také nedošlo k načtení *přihlašovací URL* mezi jednotlivými scénáři či k uzavření prohlížeče bez jeho následného otevření v následujícím scénáři. Frekvencí se ve výsledcích objevuje i nesprávné použití selektoru, kdy model využije selektor včetně *třídy* pouze jako *ID* (příklad v ukázce 6.10). Podobně jako v předchozích modelech i zde byl pro nastavení velikosti okna čteně použit nesprávný příkaz `Set Viewport`. U některých vygenerovaných varint lze spozorovat využití příkazu `:FOR` pro smyčku, který však již v rámci *RobotFramework* není podporovaný a byl nahrazen novějším zápisem. Ve zbytku varint neplatných testů se poté znovu objevují chyby jako *neexistující klíčová slova* nebo *nesprávné otevření prohlížeče*. Některé z chyb popsané u předchozích modelů byly opakovány.

Zdrojový kód 6.10: Nesprávné použití selektoru.

```

1      ...
2      Scenario 2: Teacher Login
3      ...
4      Element Should Be Visible      xpath://*[@id="
header.teacher-view-nav"]
5      ...

```



Obrázek 6.7: Výsledky pro lokální model Llama 3

Více vyladěným modelem vycházejí z modelu Codellama je **WizardCoder Python 34B**, který je konkrétně optimalizovaný pro jazyk Python a jeho generování. Zvolený byl, protože RobotFramework nad Pythonem pracuje, může volat jeho skripty a je tedy pravděpodobnost, že mezi sadou ladících dat budou právě i testy využívající RobotFramework. Podobně jako původní Codellama model zde není vygenerovaná žádná validní varianta pro 11 ze všech 12 testů. Pro jediný test 18 dokázal model vygenerovanou jednu platnou variantu, odpovídající všem požadavkům. Specificky se v neplatných variantách nacházeli následující chyby:

- Opomenuté vložení *přihlašovacích údajů* - Tento krok byl přeskočen a testy selhávají na neuskutečněném zobrazení (případně zmizení) prvků (viz ukázka

6.11). Případně model pouze nevygeneroval příkaz pro vstup na přihlašovací stránku ze úvodní obrazovky.

- Přetěžování *klíčových slov* - Pro vlastní definice klíčových slov zvolil model stejný název jako pro klíčová slova z knihovny a tedy dochází ke kolizi. V krajním případě může dojít i k rekuzi, kterou RobotFramework nedokáže řešit a test selže na počtu definovaných klíčových slov.
- Využití aria atributů pro lokaci prvku (diskutováno výše pro model Co-dellama).
- Použití nedefinované proměnné.
- Zastaralý zápis FOR smyčky.
- Zobrazení špatné URL adresy. Namísto cesty /tbui s použita pouze kořenová cesta.
- Plně chybějící nebo neplatný obsah testu.

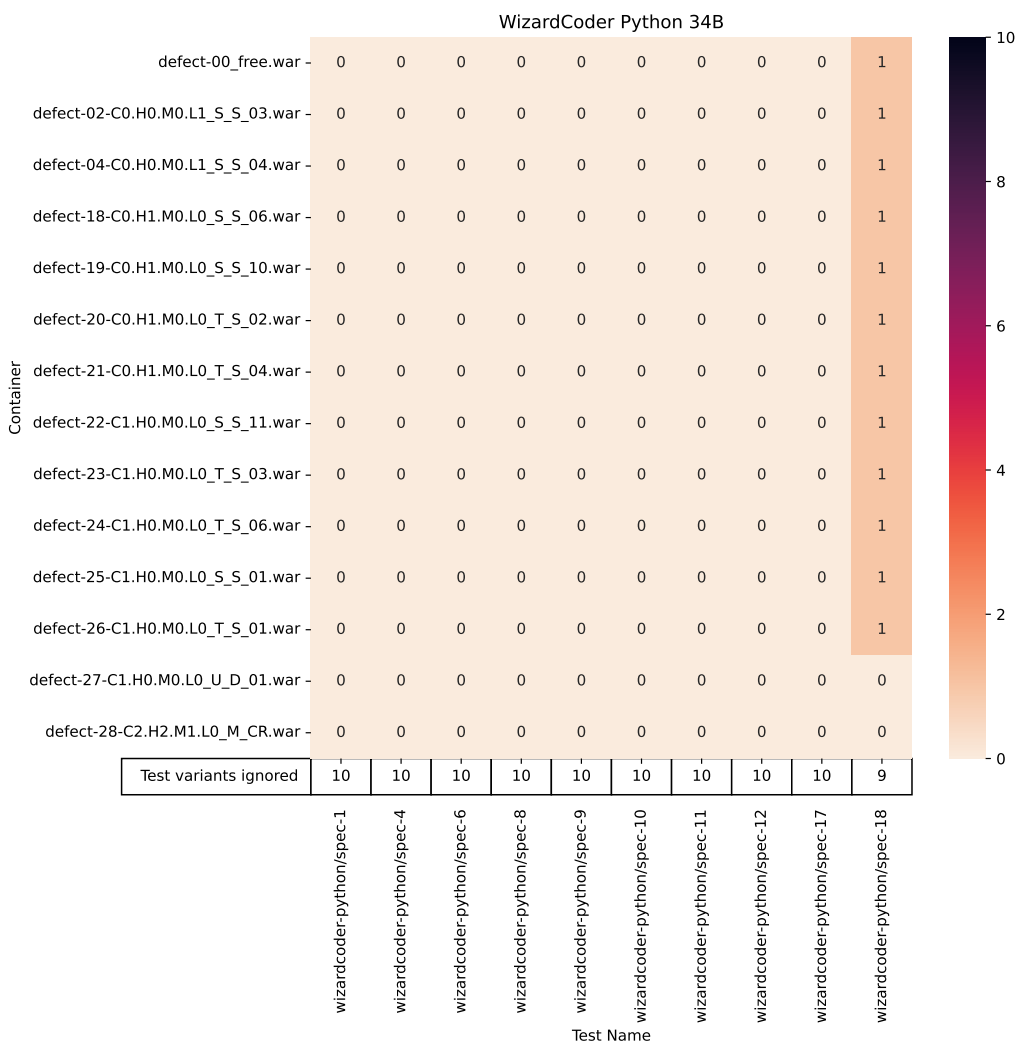
Jediný platný test však byl schopen projít pouze na neporuchových variantách aplikace a nebyl schopen odhladit chyby zanesené klony 27 a 28.

Zdrojový kód 6.11: Ukázka opomenutého vložení přihlašovacích údajů.

```

1 *** Test Cases ***
2 Scenario 1
3     Open Browser      ${URL}      ${BROWSER}
4     Wait Until Element Is Not Visible      id:header.link.login
5     Wait Until Element Contains      id:header.title.userHome
        Noah Brown
6     Wait Until Element Is Visible      id:header.link.logout
7     Wait Until Element Exists      xpath://*[@id='header.
student-view-nav']
8     Wait Until Element Contains      id:stu.view.title
        Student's View
9     Wait Until Element Exists      id:overview.personalInfoForm
10    Close Browser
11

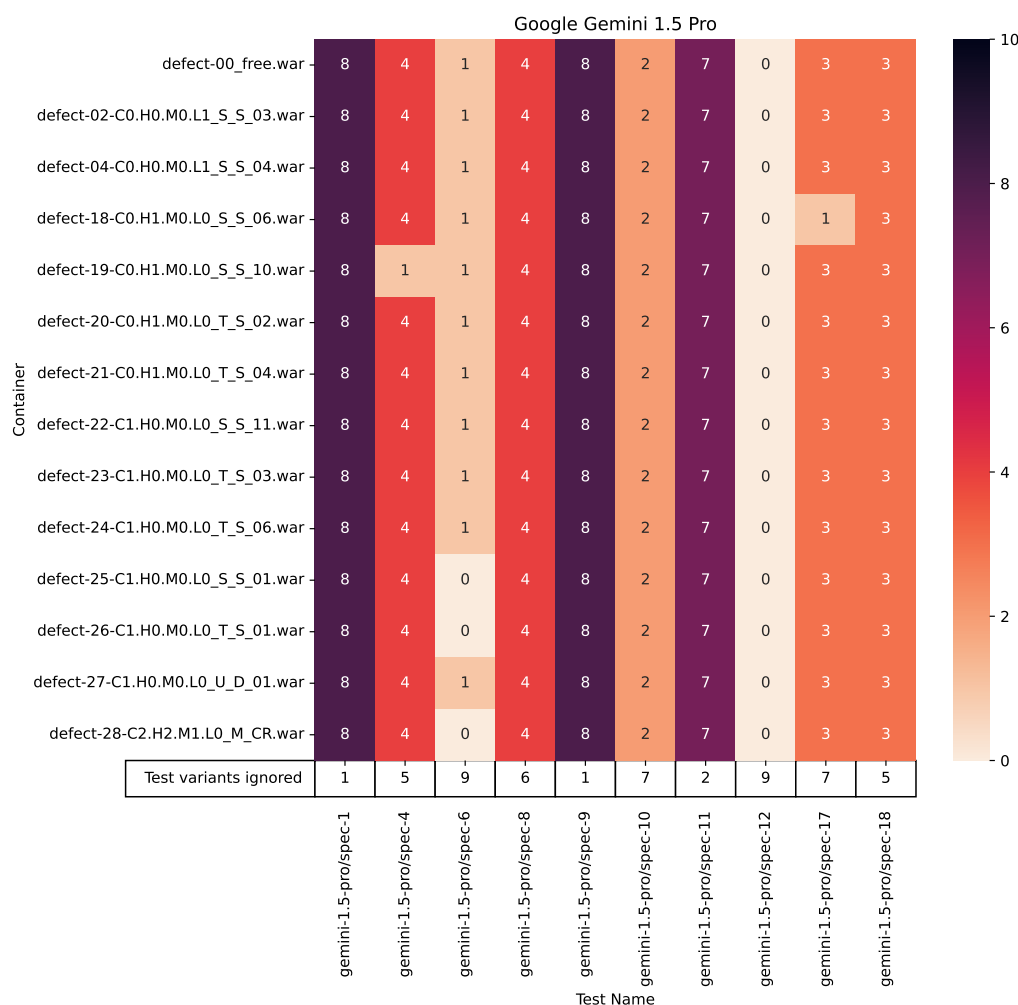
```



Obrázek 6.8: Výsledky pro lokální model WizardCoder Python 34B

6.3.4 Google

Uspokojivé výsledky také poskytl model *Gemini 1.5 Pro* od společnosti Google. Ten byl schopen pro každý test (vyjma specifikace 12 a 6) vygenerovat minimálně jednu variantu testu, schopnou odhalit veškeré chyby aplikace, jak lze vyčíst z obr. 6.9. Jedním z problémů objevujících se např ve 2 z nevalidních variant testu 1 je nesprávné escapování znaků, kdy namísto jednoho escape indikátoru "byli použity 2 (např. v ukázce 6.12 na řádce 16). V druhé chybné varintě model definoval více vlastních klíčových slov se stejným názvem. V některých případech také došlo k nesprávnému umístění asserce nebo nesprávnému pojmenování proměnné nevyhovující podmínkám názvu v rámci *RobotFramework* (např. v ukázce 6.13 na



Obrázek 6.9: Výsledky pro model Google Gemini 1.5 Pro

řádce 5) či k jejímu úplnému nedefinování. Dalším problémem také je vytváření přiřazení, která nejsou v rámci promptu definovaná. Stejně tak je u některých volání klíčových slov opomenut argument. Stejně jako v případě modelu *Llama 3* zde v některých selhávajících testech na bezchybné variantě aplikace dochází k chybám jako *chybějící znovuotevření webového prohlížeče* nebo *využití již nepodporovaného zápisu FOR cyklu* nebo jeho neukončení. Z chyb, které také šlo pozorovat u jiných modelů se zde opakuje *nesprávné přiřazení hodnoty cesty XPath v argumentu* (viz 6.3.2), *posunutá indexace prvku v rámci cesty*, *nesprávné určení cesty k prvku*, *výběr z nabídky dle ID namísto pořadí* či *využití nevhodného klíčového slova pro odsouhlasení upozornění*.

V případě testu 4 selhávají 2 varianty (resp. procházejí, protože selhání je očekáváno) ze stejného důvodu jako v případě předchozích modelů, protože chyba je

založena čistě na pseudonáhodě a nemusí se projevit vždy, takže veškeré z variant lze považovat za obecně platné. Stejně jako v případě modelu *Codellama 2* z platných testů pro specifikaci 17 na poruchovém klonu 18 neselhávají. Naopak varianta 1 testu 10 selhává na neplatném importu knihovny, ale samotné testové scénáře jsou platné a prochází.

Zdrojový kód 6.12: Ukázka nesprávného escapování znaků.

```

1 ...
2 *** Test Cases ***
3 Student View Login
4     Open Browser      ${INDEX URL}      chrome
5     Maximize Browser Window
6     Sleep      2s
7     Click Element    xpath://*[@id="header.link.login"]
8     Sleep      2s
9     Login      brown      pass
10    Element Should Not Be Visible    xpath://*[@id="header.
link.login"]
11    Sleep      1s
12    Element Text Should Be    xpath://*[@id="header.title.
userHome"]    Noah Brown
13    Sleep      1s
14    Element Should Be Visible    xpath://*[@id="header.link.
logout"]
15    Sleep      1s
16    Page Should Contain Element    css=#header\\.student-view
-nav
17    Sleep      1s
18    Element Text Should Be    xpath://*[@id="stu.view.title"]
Student's View
19    Sleep      1s
20    Page Should Contain Element    xpath://*[@id="overview.
personalInfoForm"]
21    Sleep      1s
22    Close Browser
23 ...

```

Zdrojový kód 6.13: Ukázka nesprávného pojmenování proměnné. Zbytek kódu vynechán.

```

1 *** Variables ***
2 ${URL}      http://localhost:4680/tbuis/index.jsp
3 ${LOGIN PAGE}    http://localhost:4680/tbuis/login
4 ${OVERVIEW PAGE} http://localhost:4680/tbuis/student-view/
overview
5 ${MY EXAMS PAGE} http://localhost:4680/tbuis/student-view/
myExamDates

```

6.3.5 Mistral

Jedním z modelů, který společnost *Mistral* nabízí je poměrně malý model **Mistral 7B**, nyní ve verzi 0.2. Tento model však nebyl schopen vygenerovat ani jeden validní test dle výsledků v obr. 6.10. Primárními problémy zde byly:

- *Neplatná deklarace proměnných* - Byl použit neplatný zápis pro deklaraci i následné použití proměnných (např. v ukázce 6.14).
- Využití klíčového slova `Set Viewport` jako předchozí modely.
- Snaha o přímou interpretaci *JSON* nahrávky.
- Používání neexistujících klíčových slov a nastavení.
- Nesprávné načtení knihovny.

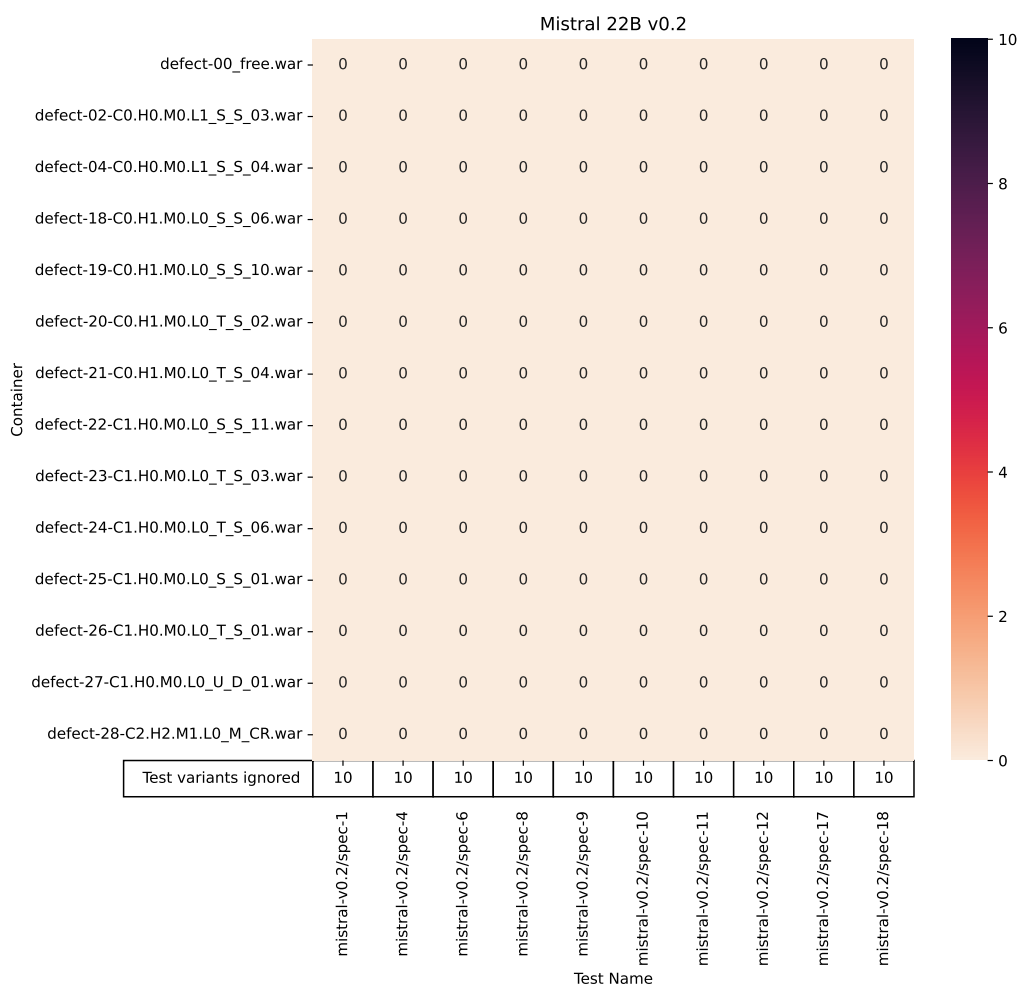
Zdrojový kód 6.14: Ukázka neplatné deklarace proměnných.

```

1 ...
2 *** Variables ***
3 username1 = "brown"
4 password1 = "pass"
5 username2 = "bla"
6 password2 = "pass"
7 username3 = "lazy"
8 password3 = "bla"
9 ...
10
```

Tato společnost nabízí i větší a proprietární model **Mistral Large**, který již byl schopen alespoň pro polovinu testových případů vygenerovat platný test. Konkrétně pro testy 4, 6, 10, 12, 17 a 18 nebyla vygenerována žádná z těchto variant (viz obr. 6.11) a ve všech chybových testech se objevovali následující chyby:

- *Neplatné přiřazení hodnoty XPath* - Stejný problém jako se objevoval např. v případě modelu Llama 3 (6.3.3) nebo Claude 3 Opus (6.3.2). Mezi argumentem a cestou chybí znak `=` nebo `:` a namísto nich se u cesty objevuje trojitě `/`.
- Neprovedení vstupu na *přihlašovací obrazovku*. Tlačítko pro *login* nebylo stisknuto a tedy prvky pro vyplnění přihlašovacího jména a hesla nebyly nalezeny.



Obrázek 6.10: Výsledky pro model Mistral 7B v0.2

- Využití neplatného *XPath* zápisu (viz ukázka 6.15). Zápis "xpath://*/[attr]" není v rámci XPath validní zápis a pro nalezení takového prvku by měl být využit zápis "xpath://*[attr]", tedy bez lomítka po *.
- Nevyužití zástupného znaku pro klávesu *Tab*. Tato klávesa může být využita pro přepnutí mezi textovým vstupem pro uživatelské jméno a heslo (není však nutná, protože RobotFramework vkládá hodnotu přímo do vstupního prvku). Protože model ke stisku klávesy využívá dnes zastaralé klíčové slovo `Press Key` je v rámci něj potřeba využít pro tuto klávesu zástupný znak, jinak dojde k vypsání textové hodnoty `TAB`, což se děje v některých vygenerovaných testech. Pokud by bylo využito klíčové slovo `Press Keys`, k této chybě by nedošlo.

- Starý zápis pro FOR smyčku (viz 6.3.3).
- Nesprávné *klíčové slovo* pro kontrolu obsahu prvku. V ukázce 6.16 lze vidět použití klíčového slova `Page Should Contain Element`, které je platné, avšak podporuje jen první z argumentů a to cestu prvku, jehož existenci v rámci DOM stromu ověřuje. Model se však skrže něj snaží kontrolovat i textový obsah prvku. Takovouto funkci skutečně RobotFramework (resp. SeleniumLibrary) nabízí, ale pod klíčovým slovem `Element Should Contain`.
- Vynechaná hlavička, obsahující import knihoven a základní nastavení testu. Viz výstup `mistral-large/spec-17-5.robot`.

Zdrojový kód 6.15: Ukázka neplatného XPath zápisu.

```

1 Input Text      xpath://*/[@id="loginPage.userNameInput"]
  maroon
2 Input Password  xpath://*/[@id="loginPage.passwordInput"]
  pass
3 Click Button    xpath://*/[@id="loginPage.loginFormSubmit"]
4

```

Zdrojový kód 6.16: Ukázka využití nesprávného klíčového slova.

```

1 Page Should Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-0"]/td[3]    Numerical
  Methods
2 Page Should Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-1"]/td[3]    Database
  Systems, Fundamentals of Computer Networks, Introduction
  to Algorithms, Mobile Applications, Web Programming
3 Page Should Not Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-2"]/td[3]
4 Page Should Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-3"]/td[3]    Computer
  System Engineering, Database Systems, Operating Systems,
  Programming Techniques
5 Page Should Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-4"]/td[3]
  Computation Structures
6 Page Should Contain Element  xpath://*[@id="tea.
  listOfAllTeachers.table.teacherRow-5"]/td[3]    Operating
  Systems, Programming in Java, Software Engineering,
  Software Quality Assurance
7

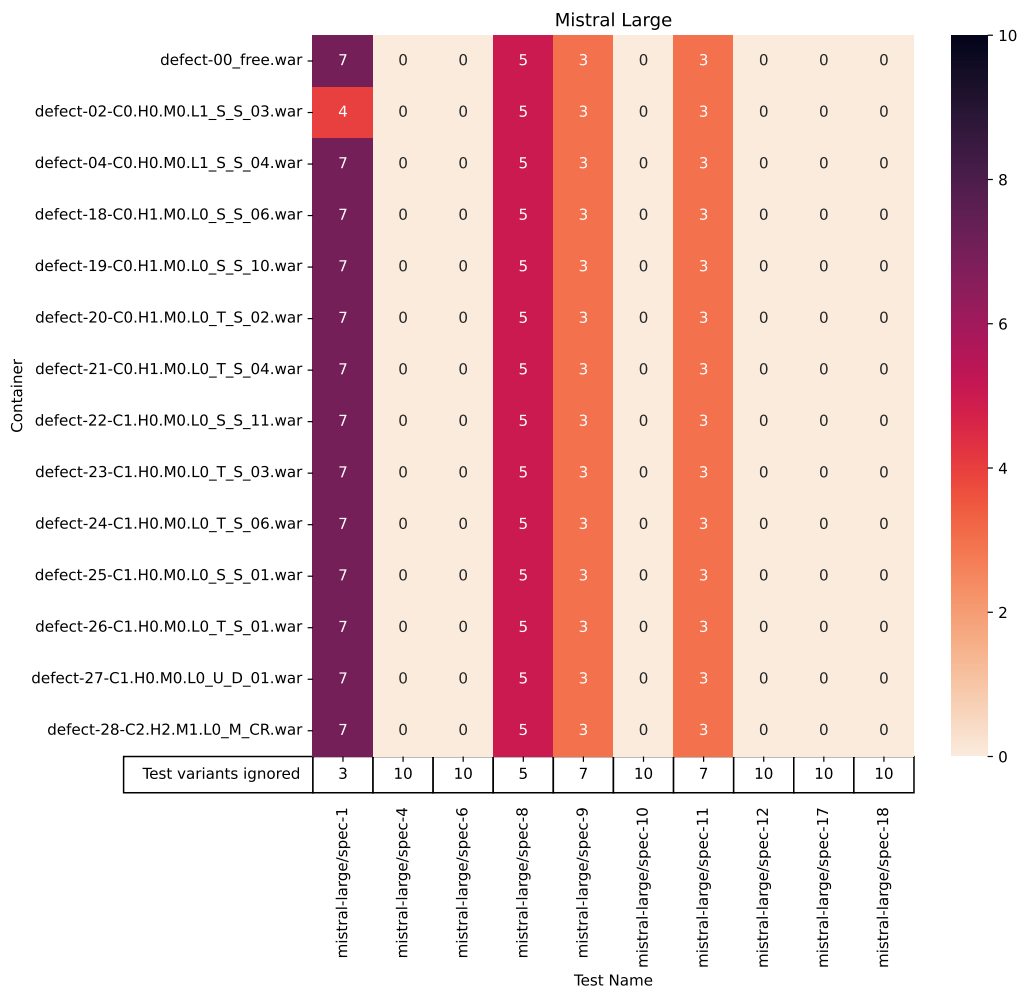
```

Z testů, které prošli kritérii pro ověření validity nedokázaly odhalit vložené chyby 3 z nich a to konkrétně pro test 1. Primárním problémem u variant testu 1, které

nebyli schopné odhalit chybu, bylo nesprávné pořadí argumentu a zároveň použití nesprávného klíčového slova pro tuto kontrolu (viz ukázka 6.17 na řádce 5), kde stejně jako bylo popsáno v předchozím odstavci by mělo být použito klíčové slovo `Element Should Contain` a pořadí argumentů by mělo být obrácené.

Zdrojový kód 6.17: Ukázka nesprávného pořadí argumentů.

```
1 Page Should Not Contain Element      id=header.link.login
2 Page Should Contain Element          id=header.title.userHome
   Noah Brown
3 Page Should Contain Element          id=header.link.logout
4 Page Should Contain Element          css=#header.student-view-nav
5 Page Should Contain      Student's View      xpath://*[@id="stu.
   view.title"]
6 Page Should Contain Element          id=overview.personalInfoForm
7
```



Obrázek 6.11: Výsledky pro model Mistral Large.

6.4 Kvalita výsledků

Protože pro GUI testy nelze využít běžné metriky využívané v softwarovém testování [54], bylo pro zhodnocení jejich kvality využita statistika úspěšnosti vygenerovaných tetů dle předpokladu pro jednotlivé modely a následně také došlo ke srovnání s testy psané člověkem, které jsou dostupné k projektu TbUIS v rámci samostatného repozitáře ².

²<https://gitlab.kiv.zcu.cz/herout/tbuis-robotframework>

6.4.1 Úspěšnost LLM modelů v generování testů

Pro určení úspěšnosti a kvality jednotlivých modelů pro účely generování GUI testů v Robot Framework zápisu jsme definovali následující metriky:

- **Úspěšnost pro scénáře** - Pro kolik scénářů vygeneroval model alespoň jednu variantu testu, schopnou určit správné chování aplikace.
- **Celková úspěšnost případů** - Kolik ze všech 1400 spuštěných testů skončilo s očekávaným výsledkem pro nasazenou variantu aplikace při daném chodu (chybový stav se považuje za selhání).
- **Validita** - Kolik ze 100 vygenerovaných testových variant prošlo filtračním kritériem, popsaným v sekci 6.2.1.
- **Úspěšnost z validních** - Kolik z validních testů bylo skutečně schopno odhalit vloženou chybu do systému. Poměr vůči předchozí metrice.
- **Celková úspěšnost testů** - Kolik ze 100 vygenerovaných testů bylo schopno odhalit jak běžné chování nasazené varinaty softwaru tak její defekt. Jedná se o poměr předchozí metriky vůči celkovému počtu vygenerovaných testů (100).

Model	Úspěšnost pro scénáře	Celková úspěšnost případů	Validita	Úspěšnost z validních	Celková úspěšnost testů
Claude 3 Opus	100,00%	43,71%	47,00%	95,74%	45,00%
Gemini 1.5 Pro	90,00%	39,43%	48,00%	87,50%	42,00%
GPT-4	90,00%	36,71%	39,00%	92,31%	36,00%
GPT-4-Turbo	70,00%	37,29%	39,00%	61,54%	24,00%
Mistral Large	40,00%	17,79%	18,00%	83,33%	15,00%
GPT-3.5 Turbo	20,00%	3,00%	3,00%	100,00%	3,00%
CodeLlama	50,00%	7,07%	9,00%	22,22%	2,00%
Llama 3	10,00%	2,00%	2,00%	100,00%	2,00%
WizzardCoder	10,00%	0,86%	1,00%	0,00%	0,00%
Mistral 7B	0,00%	0,00%	0,00%	0,00%	0,00%

Tabulka 6.1: Výsledky metrik pro zhodnocení schopností modelů.

Z výsledků (v tabulce 6.1) lze usuzovat, že modelem s nejpřesnějšími výsledky je *Claude 3 Opus*, který pro každý *testový případ* byl jako jediný schopen vygene-

rovat alespoň jednu *validní* testovou variantu. Ve všech otestovaných kombinacích *testovací případ - varianta aplikace* vyšel v necelých 44% případů s očekávaným výsledkem. Přibližně polovina vygenerovaných testů šla považovat za validní. Z nich zde pouze 2 nebyly schopny odhalit vloženou chybu, což při ignorování modelů, které vygenerovali jen jednotky validních testů, dává nejvyšší přesnost ve vygenerovaných testových skriptech pro tento model. Model *Gemini 1.5 Pro* ve validitě o jeden test předběh model *Claude 3 Opus*. Výsledná úspěšnost z validních testů však již byla nižší, ale ve výsledku stále dokázal vygenerovat 42 ze 100 testů, které plně splňovali veškeré požadavky a podávali očekávané výsledky. Druhou největší úspěšnost validních vygenerovaných testů poté vykazuje model *GPT-4*, který však podobně jako *Gemini 1.5 Pro* nebyl schopen pro 1 testový případ vygenerovat ani jeden platný test. Zároveň přibližně pouze třetina ze všech 100 vygenerovaných testů uspojovala předpoklad. Stejných 39 validních testů dokázala vygenerovat i jeho varianta *GPT-4 Turbo*. Ta však již měla vysokou míru selhání jednotlivých variant testů, což se propsalo do výsledku, ve kterém jen 24 z nich šlo považovat za korektní z pohledu předpokladu. Vyšší úspěšnost jednotlivých variant zaznamenává model *Mistral Large*, který však korektně určil pouze 18 testů, z nichž 15 bylo schopno detekovat problém.

Ostatní modely jako *GPT 3.5 Turbo* nebo zejména lokální modely jako *CodeLlama* či *Mistral 7B* byly schopny vyprodukovat pouze jednotky úspěšných testů, případně ani jeden. Jediný *CodeLlama* nabídl výsledky obsahující polovinu testových případů, avšak objevuje se zde vysoká míra neúspěšného detekování poruchy. Z testovaných lokálních modelů ho však lze s jeho 2 úspěšnými testy považovat za schopné lokální LLM. Nízká úspěšnost těchto modelů může být způsobena například nízkým počtem jejich parametrů nebo nízkou mírou zastoupení právě dat o testech a příkladech Robot Framework skriptů v trénovací sadě těchto modelů.

6.4.2 Srovnání s lidsky psanými testy

V příloze A se nacházejí zdrojové kódy, které obsahují ukázky jak testů vygenerovaných LLM modely tak zároveň lidsky psané *akceptační* testy, které jsou součástí testovaného projektu. Při srovnání např. testů pro testový případ 10 (viz. 4.2, zrušení předmětu) lze přímo spozorovat, že zatímto strojově vytvořený test (ukázka ??) volá jednotlivá *klíčová slova* a využívá pevně definované vstupní hodnoty, tak lidsky psaný test (viz zdrojový kód A.2) využívá tzv. „data-driven“ přístup³, využívající šablony (templates). V rámci tohoto přístupu běžně umožňuje Robot Framework

³<https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#data-driven-style>

vytvořit jedno uživatelsky definované klíčové slovo vyššího řádu, jehož samotná logika je skritá a umožňuje volání s odlišnou sadou vstupních a výstupních dat. Dokumentace frameworku poskytuje příklad, ve kterém namíto opakování klíčových slov (viz příklad 6.18) stačí přidat nastavení testového případu pro šablonu (viz řádek 3 ukázky 6.19) a dále psát pouze argumenty pro klíčová slova bez volání jejich názvem. Tento test je navíc parametrizován skrze data v *YAML* (v ukázce A.3), kde pro různé scénáře jsou uloženy přihlašovací údaje uživatele a studenta společně s názvem předmětu, který má být odepsán. Jednotlivá vlastní klíčová slova jsou poté importovány z externích souborů, díky čemuž mohou být znovu používány v jiných testech.

Zdrojový kód 6.18: Ukázka logiky Robot Framework testu bez šablony.

```

1 *** Settings ***
2 Test Template      Login with invalid credentials should fail

4 *** Test Cases ***
5 Invalid User Name      USERNAME      PASSWORD
                        invalid         ${VALID
                        PASSWORD}
6 Invalid Password      ${VALID USER}  invalid
7 Invalid User Name and Password  invalid      invalid
8 Empty User Name      ${EMPTY}        ${VALID
                        PASSWORD}
9 Empty Password      ${VALID USER}    ${EMPTY}
10 Empty User Name and Password  ${EMPTY}    ${EMPTY}

```

Zdrojový kód 6.19: Ukázka logiky Robot Framework testu využívajícího šablonu.

```

1 *** Test Cases ***
2 Invalid Password
3     [Template]      Login with invalid credentials should fail
4     invalid         ${VALID PASSWORD}
5     ${VALID USER}  invalid
6     invalid         whatever
7     ${EMPTY}        ${VALID PASSWORD}
8     ${VALID USER}  ${EMPTY}
9     ${EMPTY}        ${EMPTY}

```

Zatímco u lidsky psaných testů je kladen důraz i na nefunkční požadavky jako *znovupoužitelnost kódu*, *modularita*, *škálovatelnost*, *parametrizace* či jednoduchá srozumitelnost *struktury* a případné *úpravy* dat, tak stroj (přesněji LLM) vygeneruje test pouze ve *strukturované* nemodulární podobě, funkční jako jeden soubor, ve kterém jsou všechny hodnoty *napevno* uloženy. Neznamená to však, že by ho nebylo možné parametrizovat podobně jako lidsky psané testy. Zde se však parametry nacházejí již v *promptu* (viz 4.4.1) a je potřeba je upravit tam, případně přepsat tyto hodnoty ve

vygenerovaném testu, pokud to jeho forma umožní. Při generování je však problematický jeho *nedeterministický* aspekt. Zatímco u lidsky psaných testů jednu funkční komponentu lze využít na více místech, zde bude vždy přegenerována a tedy není zajištěno, zda pokaždé bude korektní a spustitelná. Většina nedostatků vůči lidsky psaným testům je však očekávána a svou funkci z pohledu „black-box“ tyto testy splňují.

6.5 Cena a časová náročnost generování

Pro případný výběr modelu k účelu generování softwarových testů lze také zvážit jejich *cenu*, *časovou* a *hardwarovou* náročnost. Pro *proprietární* LLM modely hostované na serverech jejich poskytovatele je primární metrikou *cena*⁴. V případě těchto služeb není na místě hodnotit jejich rychlost, která je běžně vysoká a zároveň silně závisí na současném zatížení serverů a dalších faktorech. Cena je běžně odvozena od počtu *vstupních* (v rámci promptu) a *výstupních* tokenů (vygenerovaná reakce modelu). Jejich ceníky⁵ udávají cenu pro obě kategorie tokenů zvlášť a to v jednotkách za milion nebo tisíc tokenů. Z použitých modelů jediný *Google Gemini* ve službě *Vertex AI* na platformě *Google Cloud*⁶ využívá cenu za znak namísto za token. Tokeny vstupních a výstupních dat byli v rámci této sekce určeny za pomoci pomocných skriptů `token_counter_<model>.py`, které lze nalézt v adresáři *auxiliary* v rámci programové složky. Pro účely výpočtu tokenů a případně odhadované ceny je u nich potřeba upravit tyto konkrétní parametry na požadovanou hodnotu.

Ceny a celková náročnost za použité proprietární modely v rámci tohoto projektu k vygenerování celkových 100 testových variant jsou uvedeny v tabulce 6.2. Nejdražším z modelů je ze *GPT-4* (varianta s kontextovým oknem 32 tisíc tokenů). Oproti němu model *Claude 3 Opus*, podávající v některých případech lepší výsledky (viz 6.4), vychází na poloviční cenu. Stále se však jedná o nákladný model na využití. *GPT-4 Turbo* zde vychází na přibližně čtvrtinovou cenu originální *GPT-4* modelu a s přihlédnutím k jeho výsledkům (v sekci 6.3.1) se jedná o výhodný model. Jako modely s nízkými náklady na využití se zde ukazují *Mistral Large*, *GPT-3.5 Turbo* a také *Google Gemini*, které se vzhledem k jeho výsledkům (viz 6.3.4) jeví jako cenově nejvýhodnější z použitých modelů. V případě modelu *Mistral Large* je jeho cena odpovídající výkonu (dle 6.3.5) a pro určité účely se může jednat o vhodný a cenově přívětivý model.

⁴Ceny uvedené v tomto dokumentu jsou platné k 4/2024.

⁵Např. <https://openai.com/api/pricing/>

⁶Ceník: <https://cloud.google.com/vertex-ai/generative-ai/pricing>

Model	Vstupních tokenů	Výstupních tokenů	Cena [USD]
GPT-4	239 500	58 238	\$21.36
Claude 3 Opus	317 583	80 233	\$10.78
GPT-4 Turbo	239 500	61 546	\$4.24
Mistral Large	342 602	62 520	\$2.20
Gemini 1.5 Pro	118 951	62 239	\$0.81
GPT-3.5 Turbo	239 500	45 798	\$0.19

Tabulka 6.2: Cena a využití za proprietární modely pro generování sady testů.

U *lokálních* modelů je stěžejní *doba*, kterou model generuje sadu našich požadovaných testů. Z tohoto důvodu lze u každého z nich změřit rychlost v jednotce vygenerovaných tokenů za sekundu. Mimo samotných hardwarových nároků modelu, které mohou být dány i nastavením prostředí, má při volbě lokálního modelu k použití vliv i jeho *energetická náročnost*, tedy současná spotřeba elektrické energie (výkon) počítače, který model provozuje. Tato náročnost může být dána samotnou architekturou modelu, jeho využitím CPU a GPU nebo také pamětí RAM. [55] Celková elektrická spotřeba stroje⁷ (práce) je poté dána *časem*, po který je provozován. Tento údaj umožňuje zhodnotit cenové nároky na provoz lokálního LLM mimo samostatných nároků na pořízení hardwaru.

Model	Rychlost generování [tok/s]	Vygenerováno tokenů	Časová náročnost	Energetická spotřeba [W]
CodeLlama 34B	2.66	56 759	5h 56m	247
WizardCoder	2.65	61 126	6h 24m	251
Llama 3	1.33	53 313	11h 8m	269
Mistral 7B	10.86	134 695	3h 27m	218

Tabulka 6.3: Náročnost lokálních LLM modelů.

Naměřené nároky a čas potřebný k vygenerování sady testů lokálními LLM je zobrazen v tabulce 6.3. Zde je potřeba upozornit na fakt, že různým počtem výstupních tokenů mezi modely je nelze napřímo srovnávat pomocí této metriky a slouží spíše pro přehled. Více relevantní je *rychlost generování* jednotlivých modelů na daném hardwaru, kde nejrychlejším a zároveň nejméně náročným modelem je *Mistral 7B*, který díky své malé velikosti bylo možné provozovat čistě na VRAM grafické karty. Modely *CodeLlama* a *WizardCoder* byly provozovány z poloviny vrstev na CPU a

⁷Měřeno nástrojem: <https://github.com/milanfon/shelly-power-client>

z poloviny na GPU, což způsobuje i větší energetické nároky a také pomalejší generování (resp. interferenci). Nejvyšší nároky zde má model *LLama 3*, který v jeho variantě se 70 miliardy parametrů lze jen z přibližně třetiny provozovat na GPU a tedy většina interference je provozována na CPU, což vede k velmi nízké rychlosti.

Budoucí vylepšení

7

V předchozím zhodnocení šlo pozorovat některé nevýhody zvolené metody generování jednotkových testů skrze LLM modely a jejich spouštění. V rámci této sekce jsou navrženy některé úpravy a vylepšení, která by mohli navazující práce využít.

7.1 Vlastní formát nahrávky

Jedním z primárních nedostatků, které šlo pozorovat je velká délka *vstupního promptu*, kvůli které může u modelu docházet ke ztrátě kontextu (kontextové okno již nebude původní prompt obsahovat) nebo také způsobuje vysokou cenu v případě proprietárních modelů (viz sekce 6.5). Značnou část délky vstupního promptu tvoří vložená nahrávka ve formátu *JSON* (viz sekce 4.3). Tento formát je již své své podstaty nekompatní. V tomto případě navíc obsahuje spoustu redundantních informací navíc jako například čtveřici identifikátorů pro každý interagovaný prvek, ze kterých pouze jeden (konkrétně *XPath* cesta) je využit na sestavení výsledného testu. To lze vidět v ukázce zdrojového kódu 7.1.

Zdrojový kód 7.1: Ukázka interakce s prvkem v rámci JSON nahrávky.

```
1 ...
2 {
3   "type": "click",
4   "target": "main",
5   "selectors": [
6     [
7       "aria/Login"
8     ],
9     [
10      "#header\\.link\\.login"
11    ],
12    [
```



```
13         "xpath//*[@id=\"header.link.login\"]"
14     ],
15     [
16         "pierce/#header\\.link\\.login"
17     ],
18     [
19         "text/Login"
20     ]
21 ],
22 "offsetY": 32,
23 "offsetX": 39.671875,
24 "assertedEvents": [
25     {
26         "type": "navigation",
27         "url": "http://localhost:4680/tbuis/login",
28         "title": "Login Page"
29     }
30 ]
31 }
32 ...
```

Návrhem v tomto případě je vytvořit vlastní formát pro nahrávku, který by obsahoval pouze relevantní informace pro tvorbu testu a zároveň je prezentoval ve formě přirozeného jazyka jakožto kroky namísto kódového zápisu. Takový návrh je ukázaný v ukázce 7.2. Takový formát by mohl vzniknout jak převodem z výstupu stávajícího nástroje pro nahrávání interakce s webovým rozhraním tak vytvořením vlastního nástroje, který by právě tyto informace přesně zaznamenával a ukládal do požadovaného formátu.

Zdrojový kód 7.2: Ukázkna navrhovaného formátu pro nahrávku.

```
1 — Open url "http://localhost:4680/tbuis/index.jsp"
2 — Click element xpath=//*[@id=\"header.link.login\"]
3 — Click element xpath=//*[@id=\"loginPage.userNameInput\"]
4 — Change value of the element xpath=//*[@id=\"loginPage.
   userNameInput\"] to "pedant"
5 — Click element xpath=//*[@id=\"loginPage.passwordInput\"]
6 — Change value of the element xpath=//*[@id=\"loginPage.
   passwordInput\"] to "pass"
7 — Click element xpath=//*[@id=\"loginPage.loginFormSubmit\"]
8 — Click element xpath=//*[@id=\"tea.menu.otherSubjects\"]
9 — Click element xpath=//*[@id=\"tea.otherSubjects.table.
   participateButton-3\"]
10 — Check if we are on the page "Others' Subjects" and if
    element with id "tea.otherSubjects.successAlert" is
    visible
11 — ...
```

7.2 Komprese promptu

Další možností, jak snížit náklady na cenu u *proprietárních* modelů je využít techniku tzv. *komprese promptu*. Příkladem takového kompresoru může být například LLMLingua od společnosti Microsoft. [56, 57] *Kompresor* zde funguje jako klasifikátor, který pro konkrétní modely určuje relevanci jednotlivých tokenů v promptu a v míře definované uživatelem může méně relevantní tokeny vyřazovat. Například kompresor *LLMLingua-2* byl vytvořen pro modely GPT-4 a Claude 3. Pro experimentální účely byla možnost komprese přidána i do orchestračního programu (sekce 5.1.2), kde se využívá ve fázi *generování testu* (sekce 4) a pro její použití je potřeba přidat argument `--compress`. Doporučená a použitá míra komprese je 40%.

7.3 Úpravy promptu dle modelu

Mimo velikosti promptu by také bylo vhodné adresovat kvalitativní nedostatky vygenerovaných testů jednotlivými modely. V sekci 6.3 jsme mohli pozorovat, že každý z modelů opakuje ve svých vygenerovaných varintách testu podobné chyby. Z tohoto důvodu by bylo možné tyto chyby adresovat přímo v promptu, aby se snížilo riziko, že je model bude opakovat. Pro každý model by tak vznikla odlišná varianta promptu (různá např. od ukázky 4.4 pro stejný testový případ). Takovýto přístup nebyl v rámci této práce využit a bylo jako řešení přistoupeno k požití obecného promptu, společného pro všechny modely, který adresuje pouze základní nedostatky, které modely mají. Příkladem takového rozšíření promptu by mohlo být předložení správného formátu adresování XPath cesty, dále by také mohlo být prospěšné nabídnout modelu správná klíčová slova, která může pro určité akce využít. Přínosné by také mohlo být určit v případě některých modelů již neplatná klíčová slova či zápisy, aby nedocházelo k jejich využívání modelem ve výstupu.

7.4 Fine-tuning

Při představení jazykových modelů v úvodu (1.2) došlo také ke zmínce o „fine-tuningu“, tedy možnosti již *předtrénovaný* modely doučit o expertízu v daném sektoru konkrétní datovou sadou. Tato možnost by šla aplikovat i v našem případě, kdy vyžadujeme, aby model schopný generovat zdrojový kód, byl obohacen o detailnější znalosti Robot Framework scénářů, případně dokumentace, atd. Tento krok by však vyžadoval vytvoření rozsáhlé datové sady obsahující tyto informace a příklady, které

by se využily k jeho dotrénování. Tento proces je však sám o sobě velmi náročný a komplexní do míry, kdy by se musela řešit například kvalita a relevance jednotlivých dat, apod. Fine-tuningem otevřených modelů se zabývají například práce [58, 59].

V rámci této práce byly prostřednictvím LLM modelů generovány softwarové testy jako Robot Framework skripty, které posléze byly otestovány na benchmarku (testovaném programu TbUIS, viz sekce 2.2), díky kterému došlo k ověření správnosti jednotlivých variant vygenerovaných testů. Tyto experimenty byly umožněny více variantami testovacího programu, mezi kterými se nacházejí defekty, které zaručují selhání v některých definovaných testových případech (popsány v sekci 4.3).

Jednotlivé testy, které připravený software s využitím LLM modelů vygeneruje, definuje uživatel skrze nahrávku scénáře ve webovém prohlížeči a následné popsání assercí pro jednotlivé prvky v testu (vysvětleno v 4.4.1). Pro *vytvoření a generování testů* byl vytvořen software, který tvoří výsledné prompty a interaguje s modely. Prozkoumány byly jak možnosti nasazení *lokálních* LLM modelů (popsáno v části 4.5.1.1) tak pronájem *proprietárních*. Vygenerované testy jsou poté zpracovány a uloženy jako *Robot Framework* scénáře, které má uživatel možnost organizovat, nahlížet do nich a případně je upravovat. Vytvořený software dále umožňuje tuto sadu testů *spouštět* a to skrze orchestraci nástroje *Docker*, ve kterém jsou nasazovány varinaty aplikací, a *Robot Framework*, který samotné testy spouští. Účelem je spustit zvolenou sadu testů na vybraném okruhu variant testovaného programu. Testovací nástroj postupně ukládá výsledky, které jsou následně zpracovány a uloženy jako report popsany v sekci 5.2.

Pro testy vygenerované v rámci tohoto projektu pro zvolené testové případy (tabulky 4.2 a 4.2) vyházela u některých modelů *celková úspěšnost* vygenerovaných testů v detekci jak správné tak i chybné funkce softwaru v rozmezí přibližně **35 až 50%** (dle tab. 6.1). Nejpresnější výsledky zde podávali LLM modely *Claude 3 Opus*, *Gemini 1.5 Pro* a *GPT-4*. Z celkového pohledu se z testovaných modelů jako nejkvalitnější jeví *Claude 3 Opus*, jehož výstup podal očekávaný výsledek v necelých **44%** testovaných případech a byl schopen pro každý ze scénářů (testových případů) vytvořit alespoň jeden platný test. Jeho cena za použití je však vyšší (dle tab. 6.2), ale pro pří-

padné nazazení by bylo možné implementovat některé z vylepšení popsané v části 7. Z proprietárních modelů se naopak cenově nejvýhodnějším jeví model *Gemini 1.5 Pro*, který byl schopen dosáhnout předpokládaného výsledku v přibližně **39.5%** a zároveň vykazoval nejvyšší počet validních testů. V rámci otestovaných lokálních LLM modelů dosahuje nelepššího výsledku *CodeLlama* s úspěšností v přibližně **7%** testovaných případů. Ve srovnání s proprietárními modely se však jedná řádově horší výsledek a při nasazení lokálních LLM modelů je tedy na místě uvažovat o modelech s vyšším počtem *parametrů*, kde by mohla být vyšší šance na přítomnost informací potřebných k jejich vygenerování.

Ve výsledku práce částečně uspokojuje motivaci (1.3), kdy nezkušený tester by měl být schopen za pomoci vytvořené *pipeline* a *specifikace* definovat a vytvořit softwarový test pro aplikaci s GUI. Tento proces se zde ověřil jako validní a je skrze něj možné vygenerovat platné testy. Samotná validita testů však zůstává nadále problémem, protože i v nejlepším případě dochází pouze k **50%** validitě vygenerovaných testových skriptů a v případných navazujících pracích bude potřeba tento nedostatek adresovat. I přesto, že LLM modely jsou schopné generovat platné Robot Framework testy, ne vždy jsou schopny odhalit případný problém a to má vliv na spolehlivost případného automatizovaného řešení.

Příložené zdrojové kódy



Zdrojový kód A.1: Korektně vygenrovaný test dle specifikace 10 (soubor „openai-gpt4/spec-10-3.robot“).

```
1 *** Settings ***
2 Library             SeleniumLibrary
3 Suite Teardown      Close Browser

5 *** Test Cases ***
6 Teacher My Subjects And Exam Date Checks
7     Open Browser      http://localhost:4680/tbuis/index.jsp
8     Chrome
9     Set Window Size   1501    1104
10    # Login Steps
11    Click Element      xpath=//*[@id="header.link.login"]
12    Sleep              1s
13    Input Text         xpath=//*[@id="loginPage.userNameInput"]
14    strict
15    Sleep              1s
16    Input Text         xpath=//*[@id="loginPage.passwordInput"]
17    pass
18    Sleep              1s
19    Click Element      xpath=//*[@id="loginPage.loginFormSubmit
20    "]
21    Sleep              2s
22    # My Subjects
23    Click Element      xpath=//*[@id="tea.menu.mySubjects"]
24    Sleep              2s
25    Wait Until Element Is Not Visible    xpath=//button[@id
26    !="tea.mySubjects.table.unregisterSubjectButton-0" and
27    @value="Remove"]    10s
28    Click Element      xpath=//*[@id="tea.mySubjects.table.
29    unregisterSubjectButton-0"]
30    Sleep              2s
```

```

24     Page Should Contain Element      id=tea.mySubjects.
    successAlert
25     # My Exam Dates
26     Click Element      xpath=//*[@id="tea.menu.myExamDates"]
27     Sleep      2s
28     Element Should Not Be Visible      xpath=//th[contains(., "
Operating Systems")]
29     # New Exam Dates
30     Click Element      xpath=//*[@id="tea.menu.newExamDates"]
31     Sleep      2s
32     Element Should Not Be Visible      xpath=//option[contains
(., "Operating Systems")]
33     # Set Evaluation
34     Click Element      xpath=//*[@id="tea.menu.setEvaluation"]
35     Sleep      2s
36     Element Should Not Be Visible      xpath=//option[contains
(., "Operating Systems")]
37     # Evaluation Table
38     Click Element      xpath=//*[@id="tea.menu.evaluationTable
"]
39     Sleep      2s
40     Element Should Not Be Visible      xpath=//option[contains
(., "Operating Systems")]
41     # Other's Subjects
42     Click Element      xpath=//*[@id="tea.menu.otherSubjects"]
43     Sleep      2s
44     Page Should Contain Element      xpath=//td[contains(., "
Operating Systems")]
45     # List of All Teachers
46     Click Element      xpath=//*[@id="tea.menu.
listOfAllTeachers"]
47     Sleep      2s
48     Element Should Not Contain      xpath=//*[@id="tea.
listOfAllTeachers.table.teacherRow-5"]      "Operating
Systems"
49     Close Browser

51 Student Other Subjects Check
52     Open Browser      http://localhost:4680/tbuis/index.jsp
    Chrome
53     Set Window Size      1501      1104
54     # Login Steps
55     Click Element      xpath=//*[@id="header.link.login"]
56     Sleep      1s
57     Input Text      xpath=//*[@id="loginPage.userNameInput"]
    orange
58     Sleep      1s

```

```
59   Input Text      xpath=//*[@id="loginPage.passwordInput"]
    pass
60   Sleep          1s
61   Click Element   xpath=//*[@id="loginPage.loginFormSubmit
    "]
62   Sleep          2s
63   # Other Subjects
64   Click Element   xpath=//*[@id="stu.menu.otherSubjects"]
65   Sleep          2s
66   ${is_present}=  Run Keyword And Return Status
    Element Should Not Contain    xpath=//tr[contains(., "
    Operating Systems")]    "Peter Strict"
67   Should Be True    ${is_present}
68   Close Browser
```

Zdrojový kód A.2: Ručně psaný test pro specifikaci 10.

```
1 *** Settings ***

3 Resource          ../../baseKeywords.robot
4 Resource          ../../keywords/scenarios/subjects/cancel/
    teaching.robot

6 Variables          ../data/teacher/UC10_CancellationSubject.
    yaml

8 Suite Setup        Prepare Suite without DB restore
9 Test Setup         Prepare Test with DB restore
10 Suite Teardown     End Test

12 *** Test Cases ***

14 TC.10.01 Teacher Cancel Teaching Of Subject
15   [Template]        Cancel Subject Teaching Success
16   [Tags]            UC.10 HappyDay MAJOR
17   [Documentation]    In this test case teacher cancel
    teaching of subject when all preconditions are fulfilled.
18   ...
19   FOR      ${params}    IN      @{{SuccessCancellations}}
20     @{{params}}
21   END

23 TC.10.02 Teacher Cancel Teaching Of Subject Fails — Some
    Students Enrolled
24   [Template]        Cancel Subject Teaching Failiture
25   [Tags]            UC.10 MINOR
26   [Documentation]    In this test case teacher tries
    cancel subject where some students are enrolled.
```



```
27     ...
28     FOR      ${params}      IN      @{{SomeStudentsEnrolled}}
29         @{{params}}
30     END

32 TC.10.03 Teacher Cancel Teaching Of Subject Fails – Teacher
    does not teach this subject
33     [Template]          Cancel Subject Teaching Not Found
34     [Tags]              UC.10  MINOR
35     [Documentation]     In this test case teacher try leave
    subject which is not taught by.
36     ...
37     FOR      ${params}      IN      @{{WrongSubject}}
38         @{{params}}
39     END
```

Zdrojový kód A.3: Vstupní data pro lisky psaný test.

```
1 SuccessCancellations:
2   —
3   — pedant
4   — Operating Systems
5   —
6   — strict
7   — Operating Systems

9 SomeStudentsEnrolled:
10  —
11  — keen
12  — Database Systems
13  —
14  — strict
15  — Programming in Java

17 WrongSubject:
18  —
19  — lazy
20  — Database Systems
```

Bibliografie

1. RAMESH, Aditya; DHARIWAL, Prafulla; NICHOL, Alex; CHU, Casey; CHEN, Mark. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. Dostupné z arXiv: 2204.06125 [cs.CV].
2. CHATTERJEE, Sayan; LIU, Ching Louis; ROWLAND, Gareth; HOGARTH, Tim. *The Impact of AI Tool on Engineering at ANZ Bank An Empirical Study on GitHub Copilot within Corporate Environment*. 2024. Dostupné z arXiv: 2402.05636 [cs.SE].
3. GEEKSFORGEEKS. *Types of Software Testing*. [B.r.]. Dostupné také z: <https://www.geeksforgeeks.org/types-software-testing/>. Přistoupeno: 10.5.2024.
4. BRADA, P. *Vývoj požadavků ve fázi projektování* [Přednáška předmětu KIV/ASWI]. Západočeská Univerzita v Plzni, 2023.
5. P. HEROUT, P. Brada a. *Základní přehled testování - a Crash Course* [Přednáška předmětu KIV/ASWI]. Západočeská Univerzita v Plzni, 2015.
6. ALÉGROTH, Emil; FELDT, Robert; KOLSTRÖM, Pirjo. *Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing*. 2016. Dostupné z arXiv: 1602.01226 [cs.SE].
7. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Dostupné z arXiv: 1810.04805 [cs.CL].
8. GOOGLE DEVELOPERS. *Introduction to Large Language Models*. 2023. Dostupné také z: <https://developers.google.com/machine-learning/resources/intro-llms>. Přistoupeno: 10.5.2024.
9. CLOUDFLARE. *What is a Large Language Model?* 2023. Dostupné také z: <https://www.cloudflare.com/en-gb/learning/ai/what-is-large-language-model/>. Přistoupeno: 10.5.2024.
10. PRAŽÁK, Ondřej; KONOPÍK, Miloslav. *Word Embeddings* [Přednáška předmětu KIV/ANLP]. 2023. Lecture 4.

11. PRAŽÁK, Ondřej; KONOPÍK, Miloslav. *Seq2Seq, Machine Translation, Attention* [Přednáška předmětu KIV/ANLP]. 2023. Lecture 9.
12. PAŠEK, Jan; PRAŽÁK, Ondřej. *Generative Models* [Přednáška předmětu KIV/ANLP]. 2023. Lecture 12.
13. FILIPPOVA, Katja. *Controlled Hallucinations: Learning to Generate Faithfully from Noisy Data*. 2020. Dostupné z arXiv: 2010.05873 [cs.CL].
14. MICROSOFT. *Grounding LLMs* [Microsoft Tech Community Blog]. 2023. Dostupné také z: <https://techcommunity.microsoft.com/t5/fasttrack-for-azure/grounding-llms/ba-p/3843857>.
15. VANMALI, Meenakshi; LAST, Mark; KANDEL, Abraham. Using a neural network in the software testing process. *International Journal of Intelligent Systems*. 2002, roč. 17.
16. SUTTON, Michael; GREENE, Adam; AMINI, Pedram. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN 0321446119.
17. PARÍZEK, Pavel. *Symbolic Execution* [Online]. [B.r.]. Dostupné také z: https://d3s.mff.cuni.cz/legacy/teaching/program_analysis_verification/files/lecture03.pdf. Lecture notes for Program Analysis and Verification course.
18. MICROSOFT. *Dynamic symbolic execution - Visual Studio (Windows)*. 2023-03. Dostupné také z: <https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/input-generation?view=vs-2022>. Čteno: 2023-10-10.
19. MICROSOFT. *Test generation - Visual Studio (Windows)* [<https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/test-generation?view=vs-2022>]. 2023-03. [cit. 2023-10-10]. Dostupné z: <https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/test-generation?view=vs-2022>. Čteno: 2023-10-10.
20. NLAB AUTHORS. *nondeterministic computation* [<https://ncatlab.org/nlab/show/nondeterministic+computation>]. 2024. Revision 13.
21. ALDRICH, Jonathan. *Concolic Testing*. 2019. Dostupné také z: <https://www.cs.cmu.edu/~aldrich/courses/17-355-19sp/notes/notes15-concolic-testing.pdf>. Přistoupeno: 28.4.2024.
22. KANNAVARA, Raghudeep et al. Challenges and opportunities with concolic testing. In: 2015, s. 374–378. Dostupné z DOI: 10.1109/NAECON.2015.7443099.

23. ENGLER, Dawson; CHEN, David Yu. EXE: Automatically Generating Inputs of Death. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA: ACM, 2006, s. 322–335. Dostupné z DOI: 10.1145/1180405.1180445.
24. SEN, Koushik; MARINOV, Darko; AGHA, Gul. CUTE: A Concolic Unit Testing Engine for C. In: *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2005. Dostupné také z: <https://web.archive.org/web/20100629114645/http://srl.cs.berkeley.edu/~ksen/papers/C159-sen.pdf>. Available at the University of Illinois at Urbana-Champaign.
25. ZHOU, Feng et al. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Seattle, WA: USENIX Association, 2006, s. 45–60. Dostupné také z: https://web.archive.org/web/20080829223442/http://cm.bell-labs.com/who/god/public_psfles/pldi2005.pdf. Available at Bell Labs.
26. PACHECO, Carlos; LAHIRI, Shuvendu K.; ERNST, Michael D.; BALL, Thomas. Feedback-Directed Random Test Generation. In: *ICSE '07*. Washington, DC, USA: IEEE Computer Society, 2007. Dostupné také z: <https://homes.cs.washington.edu/~mernst/pubs/feedback-testgen-icse2007.pdf>.
27. CAMPOS, José et al. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*. 2018, roč. 104, s. 207–235. ISSN 0950-5849. Dostupné z DOI: <https://doi.org/10.1016/j.infsof.2018.08.010>.
28. LUKASCZYK, Stephan; KROISS, Florian; FRASER, Gordon. An Empirical Study of Automated Unit Test Generation for Python. *CoRR*. 2021, roč. abs/2111.05003. Dostupné z arXiv: 2111.05003.
29. SAES, Laurence. *Unit test Generation Using Machine Learning*. 2018. Dipl. pr. Universiteit van Amsterdam. <https://research.infosupport.com/wp-content/uploads/Unit-test-generation-using-machine-Master-Thesis-Laurence-Saes.pdf>.
30. KUGLER MAAG CIE. *System Requirements Analysis*. [B.r.]. Dostupné také z: <https://www.kuglermaag.com/automotive-spice/sys2-system-requirements-analysis/>. Čteno: 30.4.2024.

31. SCHÄFER, Max; NADI, Sarah; EGHBALI, Aryaz; TIP, Frank. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *arXiv preprint arXiv:2302.06527*. 2023. Dostupné také z: <https://arxiv.org/pdf/2302.06527.pdf>.
32. LI, Raymond; ALLAL, Loubna Ben; ZI, Yangtian; MUENNIGHOFF, Niklas; KOCETKOV, Denis et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*. 2023.
33. SIDDIQ, Mohammed Latif et al. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418*. 2023.
34. *Test Smells* [<https://testsmells.org/pages/testsmells.html>]. 2021. Dostupné také z: <https://testsmells.org/pages/testsmells.html>. Čteno: 23.10.2023.
35. KATRIN KAHUR, Jennifer Su. *Java Unit Testing with AI: An AI-Driven Prototype for Unit Test Generation*. 2023. KTH Royal Institute of Technology.
36. PROJECT, BigCode. *CodeML OpenRAIL-M 0.1 License*. 2023. Dostupné také z: <https://www.bigcode-project.org/docs/pages/model-license/>. Čteno: 23.10.2023.
37. ROZIÈRE, Baptiste et al. *Code Llama: Open Foundation Models for Code*. 2023. Dostupné z arXiv: 2308.12950 [cs.CL].
38. META. *Meta LLaMA-3 Introduction*. 2024. Dostupné také z: <https://ai.meta.com/blog/meta-llama-3/>. Přistoupeno: 10.5.2024.
39. OPENAI et al. *GPT-4 Technical Report*. 2023. Dostupné z arXiv: 2303.08774 [cs.CL].
40. KEVIAN, Darioush et al. *Capabilities of Large Language Models in Control Engineering: A Benchmark Study on GPT-4, Claude 3 Opus, and Gemini 1.0 Ultra*. 2024. Dostupné z arXiv: 2404.03647 [math.OE].
41. ANTHROPIC. *Models Overview - Claude*. 2024. Dostupné také z: <https://docs.anthropic.com/claude/docs/models-overview>. Přistoupeno: 10.5.2024.
42. COHERE. *Cohere Models* [Cohere Documentation]. 2024. Dostupné také z: <https://docs.cohere.com/docs/models>.
43. RUDER, Sebastian. *Command R* [NLP News]. 2024. Dostupné také z: <https://newsletter.ruder.io/p/command-r>.
44. DEVELOPERS, Google. *Google Gemma Introduction*. 2024. Dostupné také z: <https://blog.google/technology/developers/gemma-open-models/>.
45. JIANG, Albert Q. et al. *Mistral 7B*. 2023. Dostupné z arXiv: 2310.06825 [cs.CL].

46. JIANG, Albert Q. et al. *Mixtral of Experts*. 2024. Dostupné z arXiv: 2401.04088 [cs.LG].
47. MATYÁŠ, Jiří. *Aplikace s možností injekce chyb pro ověřování kvality testů*. 2018. Dostupné také z: <https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Matyas.pdf>. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.
48. ŠMAUS, Jakub. *Rozhraní pro administraci aplikace s možností injekce chyb*. 2019. Dostupné také z: <https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Smaus.pdf>. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.
49. POUBOVÁ, Jitka. *Generování automatizovaných funkcionálních testů*. 2019. Dostupné také z: <https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/BP-Poubova.pdf>. University of West Bohemia. Accessed: 2024-04-18.
50. VAIS, Radek. *Akceptační testování v projektu TbUIS*. 2020. Dostupné také z: <https://projects.kiv.zcu.cz/tbuis/web/files/uis/kp/DP-Vais.pdf>. Dipl. pr. University of West Bohemia. Accessed: 2024-04-18.
51. GERGANOV, Georgi. *GGUF* [online]. 2024-04. [cit. 2024-04-23]. Dostupné z: <https://github.com/ggerganov/ggml/blob/master/docs/gguf.md>.
52. FACE, Hugging. *GGUF* [online]. 2024-04. [cit. 2024-04-23]. Dostupné z: <https://huggingface.co/docs/hub/en/gguf>.
53. FACE, Hugging. *Quantization* [online]. 2024-04. [cit. 2024-04-23]. Dostupné z: https://huggingface.co/docs/optimum/concept_guides/quantization.
54. COPPOLA, Riccardo; ALÉGROTH, Emil. A taxonomy of metrics for GUI-based testing research: A systematic literature review. *Information and Software Technology*. 2022, roč. 152, s. 107062. ISSN 0950-5849. Dostupné z DOI: <https://doi.org/10.1016/j.infsof.2022.107062>.
55. ZHANG, Hengrui; NING, August; PRABHAKAR, Rohan; WENTZLAFF, David. *A Hardware Evaluation Framework for Large Language Model Inference*. 2023. Dostupné z arXiv: 2312.03134 [cs.AR].
56. JIANG, Huiqiang; WU, Qianhui; LIN, Chin-Yew; YANG, Yuqing; QIU, Lili. *LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models*. 2023. Dostupné z arXiv: 2310.05736 [cs.CL].
57. PAN, Zhuoshi et al. *LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression*. 2024. Dostupné z arXiv: 2403.12968 [cs.CL].

58. WEYSSOW, Martin; ZHOU, Xin; KIM, Kisub; LO, David; SAHRAOUI, Houari. *Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models*. 2024. Dostupné z arXiv: 2308.10462 [cs.SE].
59. SHI, Andrew C.; GOVANDE, Soham; KANG, Taeuk. *Deep Learning Approaches for Natural Language Processing*. 2024. Dostupné také z: <https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1244/final-projects/AndrewCShiSohamGovandeTaeukKang.pdf>. CS224N Final Project.

Seznam obrázků

1.1	Ukázkové funkce kalkulačky vhodné pro jednotkové otestování. . . .	6
1.2	Ukázkové jednotkové testy pro funkce kalkulačky.	7
2.1	Prostředí systému TbUIS z pohledu studenta.	23
3.1	Návrh pipeline projektu.	27
4.1	Nahrávání scénáře za pomoci nástroje v Google Chrome	32
4.2	LM Studio	37
4.3	Zjednodušené schema algoritmu pro dotazování jazykového modelu při generování testu.	41
5.1	Znázornění orchestrace spouštění testů	49
5.2	Ukázka textové tabulky výsledků	51
5.3	Ukázka databázové tabulky výsledků	52
6.1	Výsledky pro model GPT-4	57
6.2	Nesprávně vybraný předmět z rozbalovacího seznamu.	58
6.3	Výsledky pro model GPT-4 Turbo	61
6.4	Výsledky pro model GPT-3.5 Turbo	62

6.5	Výsledky pro model Claude 3 Opus	65
6.6	Výsledky pro lokální model Codellama Instruct 34B	67
6.7	Výsledky pro lokální model Llama 3	69
6.8	Výsledky pro lokální model WizardCoder Python 34B	71
6.9	Výsledky pro model Google Gemini 1.5 Pro	72
6.10	Výsledky pro model Mistral 7B v0.2	75
6.11	Výsledky pro model Mistral Large.	78

Seznam tabulek

1.1	Zjednodušené testové požadavky pro ukáukový příklad kalkulačky. . .	6
2.1	Přehled a srovnání studií	20
2.2	Přehled a srovnání modelů generujících kód	21
4.1	Specifikace pro generované testy - část 1	30
4.2	Specifikace pro generované testy - část 2	31
4.3	Seznam poruchových klonů využitých pro testování.	31
4.4	Použité LLM modely	36
4.5	Seznam komponent testovací PC sestavy	38
6.1	Výsledky metrik pro zhodnocení schopností modelů.	79
6.2	Cena a využití za proprietární modely pro generování sady testů. . . .	83
6.3	Náročnost lokálních LLM modelů.	83

Seznam výpisů

3.1	Příklad struktury RobotFramework testu.	25
4.1	Ukázková struktura input složky	33
4.2	Hlavní režimy programu	33
4.3	Vytvoření nového testu (šablony)	34
4.4	Vzor pro vyplnění šablony testu	34
4.5	Vyplněná šablona testu pro specifikaci I8	34
4.6	Příklad systémového promptu	38
4.7	Ukáзка volání generace testu dle šablony	40
4.8	Použitý systémový prompt	40
4.9	Ukáзка „env“ souboru	42
4.10	Výstup modelu s nadbytečným textem.	42
5.1	Dockerfile pro sestavení obrazu varianty aplikačního serveru. . .	45
5.2	Docker Compose soubor pro sestavení kompozice	46
5.3	Vytvoření Docker kompozice z připravené konfigurace	47
5.4	Smazání Docker kompozice společně s vytvořenými obrazy . . .	47
5.5	Spuštění orchestračního programu v režimu spouštění testů . . .	48
5.6	Alternativní volání režimu spuštění testů	48

5.7	Zjednodušená ukázka konfiguračního souboru	50
53lstlisting.6.1		
6.2	Volání programu pro vyhodnocení dat.	55
6.3	Ukázka z vygenerovaného testu „openai-gpt4-turbo/spec-6-4.robot“ (zbytek kódu vynechán).	59
6.4	Klíčové slovo z testu „openai-gpt4-turbo/spec-10-10.robot“.	59
6.5	Příklad správné implementace specifikace 10 dle modelu GPT-4.	60
6.6	Ukázka nesprávného adresování XPath.	62
6.7	Ukázka selhání testu po nenalezeném prvku s danou hodnotou.	63
6.8	Ukázka promptu dle testového scénáře 10.	64
6.9	Ukázka chybně vygenerovaného testu „codellama/spec-10-7.robot“.	66
6.10	Nesprávné použití selektoru.	68
6.11	Ukázka opomenutého vložení přihlašovacích údajů.	70
6.12	Ukázka nesprávného escapování znaků.	73
6.13	Ukázka nesprávného pojmenování proměnné. Zbytek kódu vyne- chán.	73
6.14	Ukázka neplatné deklarace proměnných.	74
6.15	Ukázka neplatného XPath zápisu.	76
6.16	Ukázka využití nesprávného klíčového slova.	76
6.17	Ukázka nesprávného pořadí argumentů.	77
6.18	Ukázka logiky Robot Framework testu bez šablony.	81
6.19	Ukázka logiky Robot Framework testu využívajícího šablonu.	81
7.1	Ukázka interakce s prvkem v rámci JSON nahrávky.	85
7.2	Ukázka navrhovaného formátu pro nahrávku.	86

A.1	Korektně vygenrovaný test dle specifikace 10 (soubor „openai-gpt4/spec-10-3.robot“).	91
A.2	Ručně psaný test pro specifikaci 10.	93
A.3	Vstupní data pro lisky psaný test.	94

1101001
101011000011100010 1100001
101011010101 10



11010011101101001
01100001 101
11000101011 101