

Univerzita Jana Evangelisty Purkyně
v Ústí nad Labem
Přírodovědecká fakulta



Porovnání kvality kódu generovaného pomocí
různých velkých jazykových modelů v Pythonu

BAKALÁŘSKÁ PRÁCE

Vypracoval: Martin Renner

Vedoucí práce: Ing. Mgr. Pavel Beránek, MBA

Studijní program: Aplikovaná informatika

Studijní obor: Aplikovaná informatika

ÚSTÍ NAD LABEM 2025

UNIVERZITA JANA EVANGELISTY PURKYNĚ V ÚSTÍ NAD LABEM
Přírodovědecká fakulta
Akademický rok: 2024/2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: Martin RENNER
Osobní číslo: F22149
Studijní program: B0613P140005 Aplikovaná informatika
Téma práce: Porovnání kvality kódu generovaného pomocí různých velkých jazykových modelů v Pythonu
Zadávající katedra: Katedra informatiky

Zásady pro vypracování

V současné době stále více vývojářů využívá generativní AI modely, jako jsou velké jazykové modely (LLM), pro zjednodušení a zefektivnění implementační části softwarového vývoje. S rostoucím využíváním těchto modelů se zvyšuje potřeba porozumět jejich schopnostem i omezením při generování kódu.

Cílem této bakalářské práce je komplexně zhodnotit a porovnat kvalitu kódu generovaného různými velkými jazykovými modely (LLM), konkrétně modely ChatGPT, Claude a Gemini, v programovacím jazyce Python. Práce se zaměří nejen na funkční správnost kódu, ale i na jeho mimofunkční vlastnosti, jako jsou efektivita a udržovatelnost. K hodnocení budou použity vhodné metriky a nástroje, například jednotkové testy pro ověření funkčnosti nebo lintery pro statickou analýzu kódu. Současně bude věnována pozornost různým technikám výzev (promptů) a jejich vlivu na kvalitu generovaného kódu. Klíčovou otázkou, kterou práce řeší, je, zda a jak se kvalita kódu mezi jednotlivými modely liší a který z těchto modelů je pro generování kvalitního kódu v jazyce Python nevhodnější.

Osnova:

Úvod

Přehled současného stavu problematiky

Teoretická část

- životní cyklus vývoje
- kvalita kódu
- velké jazykové modely (LLM)
- inženýrství výzev

Praktická část

- metodika výzkumu
- tvorba výzev pro generování kódu
- zkoumání kvality vygenerovaného kódu a vlivu použití různých technik výzev

Výsledky výzkumu a diskuse

Závěr

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. CHEN, Banghao; ZHANG, Zhaofeng; LANGRENÉ, Nicolas a ZHU, Shengxin. *Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review*. Online. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2310.14735>. [cit. 2024-10-25].
2. MIGUEL, Jose P.; MAURICIO, David a RODRIGUEZ, Glen. *A Review of Software Quality Models for the Evaluation of Software Products*. Online. 2014. Dostupné z: <https://doi.org/10.48550/arXiv.1412.2977>. [cit. 2024-10-25].
3. OpenAI. *Prompt Engineering For ChatGPT: A Quick Guide To Techniques, Tips, And Best Practices*. Online. 2023. Dostupné z: <https://www.techrxiv.org/doi/full/10.36227/techrxiv.22683919.v1>. [cit. 2024-10-25].
4. SOMMERVILLE, Ian. *Software Engineering*. 9th Edition. Computer Press, 2013. ISBN 978-80-251-3826-7.
5. VASWANI, Ashish; SHAZER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion et al. *Attention Is All You Need*. Online. 2017. Dostupné z: <https://doi.org/10.48550/arXiv.1706.03762>. [cit. 2024-10-25].
6. WEI, Chengwei; WANG, Yun-Cheng; WANG, Bin a KUO, C.-C. Jay. *An Overview on Language Models: Recent Developments and Outlook*. Online. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2303.05759>. [cit. 2024-10-25].

Vedoucí bakalářské práce: **Ing. Mgr. Pavel Beránek, MBA**
Katedra informatiky

Datum zadání bakalářské práce: **28. března 2025**
Termín odevzdání bakalářské práce: **7. května 2025**

L.S.

doc. RNDr. Michal Varady, Ph.D.
děkan

RNDr. Jiří Škvor, Ph.D.
vedoucí katedry

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a použil jen pramenů, které cituji a uvádím v přiloženém seznamu literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., ve znění zákona č. 81/2005 Sb., autorský zákon, zejména se skutečností, že Univerzita Jana Evangelisty Purkyně v Ústí nad Labem má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Jana Evangelisty Purkyně v Ústí nad Labem oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladu, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

V Ústí nad Labem dne 7. května 2025

Podpis:

Děkuji vedoucímu práce Ing. Mgr. Pavlovi Beránkovi, MBA
za cenné rady a odborné vedení při zpracování bakalářské práce.
Poděkování patří také mé rodině a přátelům za podporu během studia.

POROVNÁNÍ KVALITY KÓDU GENEROVANÉHO POMOCÍ RŮZNÝCH VELKÝCH JAZYKOVÝCH MODELŮ v PYTHONU

Abstrakt:

Tato bakalářská práce analyzuje kvalitu kódu generovaného vybranými velkými jazykovými modely v programovacím jazyce Python. Hlavním cílem je porovnat programovací schopnosti těchto modelů při použití identických výzev a zároveň zhodnotit vliv různých technik inženýrství výzev na generovaný kód. Porovnání se soustředí na klíčové aspekty kvality kódu, jako jsou funkční vhodnost, výkonová efektivita a udržovatelnost. Pro měření těchto charakteristik jsou využity kvantitativní metriky, jako jsou jednotkové testy, měření doby běhu programu nebo statická analýza kódu. Výsledky této práce přispějí k hlubšímu porozumění rozdílů mezi modely, vlivu využití různých technik výzev a mohou pomoci při výběru nejhodnějšího jazykového modelu pro generování kvalitního kódu v Pythonu.

Klíčová slova: velké jazykové modely, inženýrství výzev, kvalita kódu, generování kódu

COMPARISON OF CODE QUALITY GENERATED BY DIFFERENT LARGE LANGUAGE MODELS IN PYTHON

Abstract:

This bachelor's thesis analyzes the quality of code generated by several large language models in Python. The aim of this study is to systematically evaluate the programming capabilities of these models when using identical prompts, while also evaluating the impact of different prompt engineering techniques on the generated code. The comparison focuses on key aspects of code quality, such as functional suitability, performance efficiency, and maintainability, which are evaluated using quantitative metrics including unit testing, execution time measurement, and static code analysis. The findings of this thesis aim to contribute to a deeper understanding of the differences between models, the effects of various prompt engineering techniques, and may assist in selecting the most suitable language model for generating high-quality code in Python.

Keywords: large language models, prompt engineering, code quality, code generation

Obsah

1. Úvod	19
2. Přehled současného stavu problematiky	21
3. Teoretická část	23
3.1. Životní cyklus vývoje softwaru	23
3.2. Kvalita softwaru	25
3.3. Velké jazykové modely	29
3.4. Inženýrství výzev	35
4. Praktická část	41
4.1. Výzkumné otázky	41
4.2. Metodika výzkumu	41
4.3. Výběr velkých jazykových modelů	43
4.4. Výběr technik inženýrství výzev	43
4.5. Explorační analýza	44
4.6. Výběr testovaných úloh a tvorba výzev	47
4.7. Výběr testovaných metrik kvality kódu a tvorba testovacích skriptů	50
4.8. Generování kódu různými velkými jazykovými modely	54
4.9. Vyhodnocení kvality kódu	55
5. Výsledky výzkumu a diskuse	57
5.1. Kompilovatelnost	57
5.2. Funkční úplnost	61
5.3. Funkční správnost	65
5.4. Časová náročnost	72
5.5. Statická analýza kódu	78
5.6. Počet řádků	84
5.7. Celkové hodnocení	92
6. Závěr	95
A. Externí přílohy	101

Seznam obrázků

3.1. Vodopadový model a Iterativní model. Vlastní zpracování dle [7].	25
3.2. Schéma hluboké neuronové sítě. Vlastní zpracování dle [19].	31
3.3. Schéma architektury transformátoru. Převzato z [21].	32
3.4. Ilustrace tokenizace. Vlastní zpracování dle [24].	33
3.5. Ilustrace embeddingu. Převzato z [26].	34
5.1. Kompilovatelnost – Celková úspěšnost	60
5.2. Funkční úplnost – Konzolové vykreslování	62
5.3. Funkční úplnost – Celkový přehled	64
5.4. Funkční správnost – Kalkulačka	66
5.5. Funkční správnost a statická analýza kódu – Kalkulačka	66
5.6. Funkční správnost – Konzolové vykreslování	67
5.7. Funkční správnost a statická analýza kódu – Konzolové vykreslování	68
5.8. Funkční správnost – Úkolníček	70
5.9. Funkční správnost a statická analýza kódu – Úkolníček	70
5.10. Statická analýza kódu – Kalkulačka	79
5.11. Statická analýza kódu – Konzolové vykreslování	80
5.12. Statická analýza kódu – Úkolníček	82
5.13. Počet řádků – Kalkulačka	85
5.14. Počet řádků a statická analýza kódu – Kalkulačka	86
5.15. Počet řádků – Konzolové vykreslování	87
5.16. Počet řádků a statická analýza kódu – Konzolové vykreslování	88
5.17. Počet řádků – Úkolníček	90
5.18. Počet řádků a statická analýza kódu – Úkolníček	90
5.19. Úspěšnost jednotlivých typů výzev napříč úlohami	92
5.20. Úspěšnost Zero-shot a Few-shot výzev napříč úlohami	93
5.21. Úspěšnost jednotlivých modelů napříč úlohami	93

Seznam tabulek

5.1. Kompilovatelnost – Kalkulačka	58
5.2. Kompilovatelnost – Konzolové vykreslování	58
5.3. Kompilovatelnost – Úkolníček	59
5.4. Funkční úplnost – Kalkulačka	61
5.5. Funkční úplnost – Konzolové vykreslování	62
5.6. Funkční úplnost – Úkolníček	63
5.7. Funkční správnost – Kalkulačka	65
5.8. Funkční správnost – Konzolové vykreslování	67
5.9. Funkční správnost – Úkolníček	69
5.10. Časová náročnost – Kalkulačka	73
5.11. Časová náročnost – Konzolové vykreslování	74
5.12. Časová náročnost – Úkolníček	76
5.13. Statická analýza kódu – Kalkulačka	78
5.14. Statická analýza kódu – Konzolové vykreslování	80
5.15. Statická analýza kódu – Úkolníček	81
5.16. Počet řádků – Kalkulačka	84
5.17. Počet řádků – Konzolové vykreslování	87
5.18. Počet řádků – Úkolníček	89

Seznam ukázek kódu

4.1. Kompilovatelnost – Kalkulačka	51
4.2. Funkční správnost – Kalkulačka	52
4.3. Časová náročnost – Kalkulačka	53
4.4. Statická analýza kódu – Kalkulačka	54
4.5. Počet řádků – Kalkulačka	54

1. Úvod

V posledních letech dochází k dynamickému rozvoji velkých jazykových modelů, které nacházejí široké uplatnění v mnoha oblastech, včetně softwarového inženýrství. Tyto modely lze využít v různých fázích vývoje softwaru, od analýzy požadavků přes návrh architektury až po implementaci, testování a údržbu kódu. Tato práce se zaměřuje na implementační fázi, konkrétně na generování zdrojového kódu v jazyce Python.

Přestože velké jazykové modely dokáží produkovat kód splňující stanovené standardy, jeho kvalita není vždy konzistentní. Významnou roli v tomto procesu hrají nejen schopnosti jednotlivých modelů, ale i techniky inženýrství výzev, které mohou zásadně ovlivnit strukturu a kvalitu generovaného kódu. Tato variabilita představuje významnou výzvu, neboť kvalita kódu přímo ovlivňuje charakteristiky softwaru, jako jsou funkčnost, efektivita, bezpečnost, spolehlivost, udržovatelnost a další.

S ohledem na tuto problematiku si tato práce klade za cíl nejen zhodnotit, jak různé techniky inženýrství výzev ovlivňují kvalitu generovaného kódu, ale také porovnat jednotlivé modely při použití totožné výzvy. Pozornost je zaměřena na modely OpenAI o3-mini-high, Anthropic Claude 3.7 Sonnet a Google Gemini Pro 2.0 Experimental, které v současné době patří mezi nejvýkonnější dostupné modely na trhu. Hlavními kritérii hodnocení kvality kódu jsou funkční vhodnost, výkonová efektivita a udržovatelnost. Klíčovými otázkami, kterými se práce zabývá jsou:

- 1. Jak různé techniky inženýrství výzev ovlivňují kvalitu kódu generovaného velkými jazykovými modely?*
- 2. Jak se liší kvalita generovaného kódu mezi vybranými velkými jazykovými modely?*

Odpovědi na tyto otázky mohou přispět k hlubšímu porozumění vlivu výzev i specifik jednotlivých modelů na výslednou kvalitu kódu a naznačit, zda lze systematickým přístupem dosáhnout lepsích výsledků.

Kromě akademického přínosu má práce i praktickou ambici usnadnit programátorům a studentům softwarového inženýrství výběr vhodného modelu a techniky inženýrství výzev pro optimalizaci kvality generovaného kódu. Systematické srovnání modelů a analýza vlivu různých metod může přispět k efektivnějšímu využívání velkých jazykových modelů v oblasti programování.

Pro dosažení stanovených cílů je navržena následující metodika. Nejprve proběhne výběr velkých jazykových modelů a technik inženýrství výzev. Následně bude provedena explorační analýza, na jejímž základě budou vytvořeny výzvy pro testované úlohy. Dále budou definovány klíčové

charakteristiky kvality kódu a vytvořeny testovací skripty pro jejich objektivní měření. Poté budou jednotlivé modely použity ke generování kódu, jehož kvalita bude následně analyzována. Na závěr budou výsledky zhodnoceny a interpretovány s cílem identifikovat nejlepší přístupy ke generování kvalitního kódu pomocí velkých jazykových modelů.

Struktura práce je členěna do tří hlavních částí. První se věnuje teoretickému ukotvení tématu a vymezení klíčových pojmů, jako jsou životní cyklus vývoje softwaru, kvalita kódu, velké jazykové modely a inženýrství výzev. Druhá část se zaměřuje na praktickou realizaci výzkumu, zahrnující tvorbu promptů, návrh testovacích skriptů a měření kvality generovaného kódu. Třetí část obsahuje vyhodnocení výsledků a diskusi nad hlavními poznatky. Na závěr jsou shrnuty hlavní přínosy práce a možné směry budoucího výzkumu v této oblasti.

2. Přehled současného stavu problematiky

Evaluace výstupů velkých jazykových modelů představuje zásadní součást procesu posuzování jejich schopností a praktické využitelnosti. Hodnocení je realizováno prostřednictvím benchmarků, které umožňují objektivní srovnání výkonnosti napříč různými velkými jazykovými modely. Zatímco některé benchmarky mají univerzální charakter, jiné jsou cíleně zaměřeny na specifické domény či úzce vymezené oblasti použití. Mezi známé benchmarky patří MATH, GSM8K a MMLU. [1]

Pro vyhodnocení kvality výstupů zavádí každý z benchmarků vlastní metriky a přístupy, které se liší v závislosti na charakteru hodnocených úloh. Mezi základní přístupy patří hodnocení na základě kvantitativních metrik, lidské hodnocení (Human Evaluation) a hodnocení pomocí jiného velkého jazykového modelu (LLM-as-a-judge). [2]

Výsledky benchmarkových testů jsou standardně publikovány ve formě leaderboardů, které přehledně prezentují srovnání výkonnosti jednotlivých modelů. Tyto žebříčky představují důležitý nástroj pro orientaci v rychle se rozvíjejícím prostředí dostupných modelů a výrazně usnadňují rozhodování při jejich výběru. [1]

V kontextu této práce jsou důležité zejména přístupy k hodnocení kvality generovaného kódu. V této oblasti existuje řada specializovaných benchmarků, které posuzují schopnost modelů generovat kód v různých programovacích jazycích, úrovních obtížnosti nebo typech programovacích úloh. Mezi takové benchmarky patří APPS, HumanEval, MBPP a ClassEval [3].

Jeden z těchto benchmarků, HumanEval hodnotí funkční správnost generovaného kódu na základě sady 164 programovacích úloh. Každá úloha obsahuje specifikaci požadovaného chování a sadu jednotkových testů. Výkonnost modelu je měřena prostřednictvím úspěšnosti splnění těchto testů, vyjádřené pomocí standardizované metriky `pass@k`. Zadání úloh jsou koncipována tak, aby ověřovala schopnost aplikace programovacích znalostí, práce s algoritmickými strukturami a základních matematických dovedností. [4]

Tato práce si neklade za cíl návrh nového benchmarku ani tvorbu univerzálního hodnoticího skóre. Využívá však principy běžné v rámci stávajících benchmarkových evaluací a přistupuje k hodnocení kvality generovaného kódu systematickým způsobem. Zaměřuje se přitom nejen na jeho funkční správnost, ale také na mimofunkční charakteristiky, jako jsou efektivita a udržovatelnost. Tyto charakteristiky jsou hodnoceny na základě vlastních kvantitativních metrik, které vycházejí z obecně využívaných principů.

3. Teoretická část

Tato kapitola poskytuje teoretický základ nezbytný pro pochopení problematiky hodnocení kvality kódu generovaného velkými jazykovými modely. Nejprve se kapitola zaměřuje na životní cyklus vývoje softwaru a definici kvality softwaru. Poté jsou vysvětleny základní principy velkých jazykových modelů a závěrečná část kapitoly se věnuje problematice inženýrství výzev.

3.1. Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru představuje strukturovaný proces, který definuje jednotlivé fáze a činnosti nezbytné k vytvoření kvalitního softwarového produktu. Tento cyklus rozděluje vývoj do logicky navazujících etap a umožňuje systematické řízení i sledování pokroku v rámci celého projektu. Zahrnuje kroky od plánování po údržbu a aktualizaci softwaru. Jeho hlavním přínosem je podpora kvality, efektivity a celkové úspěšnosti vývoje softwarového projektu. [5]

Fáze životního cyklu vývoje softwaru

Typický životní cyklus vývoje softwaru je rozdělen do několika fází, z nichž každá má svůj specifický účel a výstup:

1. Sběr a analýza požadavků

Tato fáze zahrnuje shromažďování a podrobnou analýzu požadavků od zákazníka a dalších zainteresovaných stran. Výstupem je přesně definovaný a zdokumentovaný soubor požadavků, který slouží jako základ pro celý další vývoj. Klíčové je zajistit jejich jednoznačné vyjádření a dokumentaci, aby se předešlo nedorozuměním v pozdějších etapách. [5]

2. Návrh

Na základě specifikovaných požadavků se v této fázi vytváří architektura a technický návrh systému. Používají se diagramy, modely a specifikace, které popisují strukturu a chování softwaru. Cílem je navrhnout systém, jenž bude modulární. Tím se zajistí jednodušší pochopení, údržba, opakované použití a snazší testování. [6]

3. Implementace

V této fázi dochází k samotnému vývoji softwaru podle připravených návrhů. Vývojáři vytvářejí zdrojový kód v souladu se stanovenými standardy a využívají nástroje pro správu verzí s cílem podpořit týmovou spolupráci. Součástí implementace je také dokumentace kódu, která usnadňuje jeho budoucí údržbu. [6]

4. Testování

Cílem testování je ověřit, že software odpovídá požadavkům a neobsahuje chyby. Pro ověření se používají různé testovací metody, jako jsou jednotkové testy, integrační testy, systémové testy a akceptační testy. Tato fáze je důležitá pro odhalení a odstranění chyb ještě před nasazením softwaru do provozu. [6]

5. Nasazení

Po úspěšném dokončení testování je software nasazen do produkčního prostředí a zpřístupněn koncovým uživatelům. Cílem nasazení je nejen uvedení softwaru do provozu, ale i zajištění jeho správného používání uživateli. To zahrnuje pořádání školení nebo tvorbu uživatelské dokumentace. [5]

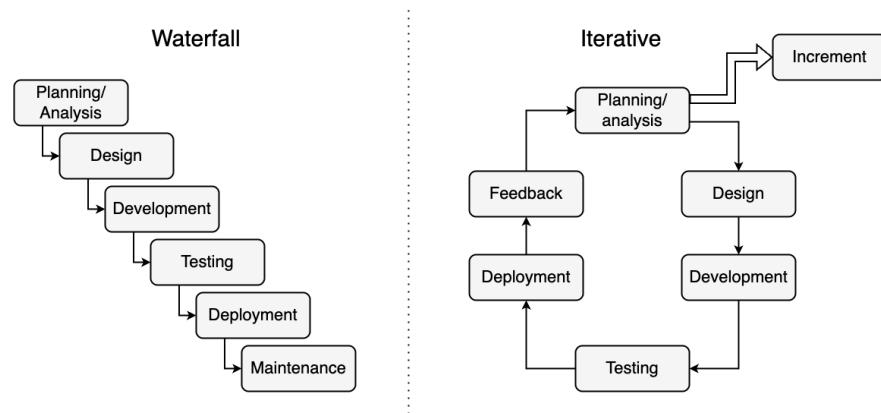
6. Údržba

Údržba zahrnuje opravy chyb, pravidelné aktualizace a rozšiřování funkcionality v reakci na měnící se požadavky uživatelů nebo technologického prostředí. Tato fáze je nezbytná pro zachování dlouhodobé relevance, bezpečnosti a funkčnosti softwaru. [5]

Cílem definovaného procesu je pokrýt všechny aspekty softwarového vývoje. Obsah i členění jednotlivých fází se může mírně lišit v závislosti na zvolené metodice a povaze konkrétního projektu. V praxi se pro uplatnění životního cyklu softwaru využívají modely, které určují způsob realizace vývoje. [5]

Modely vývoje softwaru

V rámci životního cyklu vývoje softwaru existuje několik vývojových modelů, které určují strukturu a způsob organizace jednotlivých fází. K nejzákladnějším patří vodopádový model, který je charakteristický sekvenčním průchodem jednotlivými fázemi, kdy každá etapa musí být dokončena před přechodem na následující. Tento model je jednoduchý a přehledný, avšak méně flexibilní vůči změnám v požadavcích. Naopak agilní přístupy vycházejí z principů Agilního manifestu, který klade důraz na spolupráci se zákazníkem, schopnost reagovat na změny nebo týmovou komunikaci. Agilní vývoj místo pevně daných plánů preferuje rychlou zpětnou vazbu a průběžné zlepšování, což podporuje pružnost a přizpůsobení se měnícím požadavkům. Iterativní model umožňuje vývoj ve více cyklech, kde každá iterace představuje částečně funkční produkt, jenž je postupně rozšiřován až do konečné podoby. V praxi existuje celá řada dalších modelů, které kombinují prvky různých přístupů a přizpůsobují se specifickým potřebám projektů. Bez ohledu na zvolený model vývoje má správné a důsledné uplatnění životního cyklu zásadní dopad na výslednou kvalitu softwarového produktu. [5]



Obrázek 3.1.: Vodopadový model a Iterativní model. Vlastní zpracování dle [7].

Vliv životního cyklu vývoje softwaru na kvalitu kódu

Systematické uplatnění životního cyklu vývoje softwaru významně přispívá ke zvýšení kvality kódu. Postupné členění vývoje do fází, jako je analýza požadavků, návrh architektury, implementace, testování a údržba, zajišťuje strukturovaný a přehledný přístup k jeho tvorbě. Jednotlivé fáze podporují tvorbu modulárního, dobře zdokumentovaného a snadno testovatelného kódu, což výrazně zvyšuje jeho čitelnost, udržovatelnost i rozšiřitelnost. Důraz na průběžné testování a kontrolu kvality zajišťuje, že výsledný kód je funkčně správný a plně odpovídá původním požadavkům. [5]

3.2. Kvalita softwaru

Definice kvality softwaru odpovídá „Míře, do jaké systém, komponenta nebo proces splňuje stanovené požadavky.“ [8] (překlad) a „Míře, do jaké systém, komponenta nebo proces splňuje potřeby nebo očekávání zákazníka nebo uživatele.“ [8] (překlad).

Kvalita softwaru jako celku hraje klíčovou roli, protože na něm dnes závisí chod podniků i celé společnosti. Nedostatečná kvalita může vést nejen k nespokojenosti uživatelů, ale také k vážným důsledkům, jako jsou finanční ztráty nebo ohrožení lidských životů. K objektivnímu hodnocení kvality softwaru byly navrženy modely kvality softwaru. [9]

Modely kvality softwaru

Modely kvality softwaru poskytují teoretický rámec pro hodnocení softwarových produktů. Jejich cílem je systematizovat pohled na kvalitu softwaru a usnadnit její hodnocení. Skládají se ze souboru charakteristik a jejich podrobnějších podcharakteristik, které popisují různé aspekty kvality a vzájemné vztahy mezi nimi. Jednotlivé podcharakteristiky jsou měřeny odpovídajícími metrikami. [10]

Vybrané modely kvality softwaru zahrnují:

- **McCallův model**

McCallův model byl jeden z prvních systematických přístupů k hodnocení softwarové kvality [9].

Rozděluje kvalitu softwaru do tří hlavních oblastí:

- **Činnost produktu (Product Operation):** Hodnotí, jak software plní své funkce při běžném používání [9].
- **Revize produktu (Product Revision):** Hodnotí schopnost softwaru být upravován a přizpůsobován [9].
- **Přenositelnost produktu (Product Transition):** Zabývá se přenositelností softwaru do jiných prostředí a technologií [9].

Každá z těchto oblastí je dále členěna na konkrétní faktory. Model celkem definuje jedenáct faktorů kvality, které zahrnují korektnost, spolehlivost, testovatelnost, znovupoužitelnost a další. Tento model se stal základem pro další modely kvality softwaru. [10]

- **Boehmův model**

Boehmův model kvality představuje podrobnější a hierarchicky uspořádaný přístup k hodnocení kvality softwaru než McCallův model. Model strukturuje charakteristiky do tří úrovní, které zahrnují vysokou úroveň, střední úroveň a primitivní charakteristiky. Každá úroveň přispívá k celkovému hodnocení kvality softwaru. [10, 11]

Hlavní charakteristiky na vysoké úrovni jsou:

- **Přenositelnost (Portability):** Schopnost softwaru fungovat v různých prostředích nebo na různých platformách [11].
- **Užitečnost (Utility):** Kritérium hodnotí do jaké míry lze používat software v aktuálním stavu [11].
- **Udržovatelnost (Maintainability):** Schopnost softwaru být pochopen, upraven, testován [11].

- **ISO/IEC 9126**

Mezinárodní standard ISO/IEC 9126 byl vytvořen za účelem sjednocení přístupů k hodnocení kvality softwaru. Tento model navazuje na předchozí přístupy definované v McCallově a Boehmově modelu a dále je rozvíjí. Definuje šest hlavních charakteristik kvality, z nichž každá zahrnuje konkrétní podcharakteristiky. [9, 10]

Hlavní charakteristiky:

- **Funkčnost (Functionality):** Schopnost softwaru poskytovat funkce odpovídající specifikovaným a předpokládaným požadavkům a potřebám uživatelů [11].

- **Spolehlivost (Reliability):** Schopnost softwaru odolávat chybám, selháním a neočekávaným situacím [11].
- **Použitelnost (Usability):** Schopnost softwaru být snadno pochopen a ovládán cílovou skupinou uživatelů [11].
- **Účinnost (Efficiency):** Schopnost softwaru poskytovat požadovaný výkon při optimálním využití systémových zdrojů [11].
- **Udržovatelnost (Maintainability):** Schopnost softwaru být analyzován, upravován a testován [11].
- **Přenositelnost (Portability):** Schopnost softwaru být přenesen mezi různými prostředími [11].

Model ISO/IEC 9126 dále rozlišuje tři úrovně hodnocení kvality, které odrážejí různé fáze vývoje:

- **Interní kvalita:** Hodnocení vnitřních charakteristik softwaru bez jeho spuštění [10].
- **Externí kvalita:** Hodnocení chování softwaru během jeho vykonávání [10].
- **Kvalita v užívání:** Hodnocení kvality z pohledu koncového uživatele v reálném provozním prostředí [10].

• ISO/IEC 25010

Standard ISO/IEC 25010 představuje aktuální, nejkomplexnější a mezinárodně uznávaný model pro hodnocení kvality softwarových produktů. Navazuje na starší normu ISO/IEC 9126, kterou rozšiřuje a modernizuje v souladu s aktuálními požadavky na vývoj a provoz softwaru tím, že přidává další charakteristiky. [10]

Norma ISO/IEC 25010 původně definovala osm charakteristik. V roce 2023 byla rozšířena o devátou charakteristiku, bezpečnost (safety). Současně došlo k přejmenování některých již existujících charakteristik. [12]

Hlavní charakteristiky kvality softwaru dle ISO/IEC 25010 jsou:

- **Funkční vhodnost (Functional Suitability):** Schopnost softwaru plnit požadované funkce v souladu s požadavky uživatele. Posuzuje se, zda systém poskytuje všechny potřebné funkce, zda jsou výsledky správné a přesné a zda jsou funkce vhodně navržené tak, aby efektivně podporovaly uživatele při plnění jeho úkolů. [13]
- **Výkonová efektivita (Performance Efficiency):** Schopnost softwaru rychle a efektivně vykonávat své funkce a hospodárně při tom nakládat s dostupnými prostředky. Sleduje zda systém dosahuje požadované rychlosti, efektivně využívá systémové prostředky, a zvládá různé úrovně zátěže. [13]
- **Kompatibilita (Compatibility):** Schopnost softwaru spolupracovat s jinými produkty, systémy nebo komponentami. Zahrnuje interoperabilitu, tedy schopnost komunikace a výměny dat s jinými systémy, a schopnost sdílet společné prostředky bez negativního ovlivnění ostatních systémů. [13]

- **Použitelnost (Interaction Capability):** Schopnost softwaru, která hodnotí, zda mohou uživatelé systém snadno ovládat a používat. Posuzuje se naučitelnost, ovladatelnost, ochrana před chybami, jednoduchost a celková přístupnost rozhraní pro různé skupiny uživatelů, včetně uživatelů se specifickými potřebami. [13]
- **Spolehlivost (Reliability):** Schopnost softwaru fungovat v spolehlivě za definovaných podmínek. Zahrnuje bezchybnost, tedy schopnost fungovat správně bez výskytu poruch, dostupnost systému v okamžiku potřeby, odolnost vůči chybám hardwaru či softwaru a schopnost obnovit data a systémový stav po selhání. [13]
- **Zabezpečení (Security):** Schopnost systému odolat neoprávněným přístupům, útokům či zneužití. Mezi dílčí charakteristiky patří důvěrnost, integrita, nepopiratelnost, odpovědnost nebo autenticita. [13]
- **Udržovatelnost (Maintainability):** Schopnost software být upravován, opravován či přizpůsoben změnám. Klíčovými schopnostmi jsou modularita, znovupoužitelnost, analyzovatelnost, modifikovatelnost a testovatelnost. Díky těmto charakteristikám je systém flexibilní, snadno spravovatelný a odolný vůči chybám při změnách. [13]
- **Flexibilita (Flexibility):** Schopnost systému přizpůsobit se změnám v požadavcích nebo způsobu použití. Zahrnuje adaptabilitu, škálovatelnost, instalovatelnost a nahraditelnost. Tato charakteristika umožňuje systém nasadit v různých scénářích a snadno ho upravit podle aktuálních potřeb. [13]
- **Bezpečnost (Safety):** Schopnost systému zajistit bezpečný provoz i za mimořádných nebo rizikových podmínek. Hodnotí se identifikace potenciálních rizik, schopnost přejít automaticky do bezpečného režimu v případě chyby, varování uživatele před nebezpečnými situacemi a zajištění bezpečné integrace s dalšími komponentami systému. Tyto charakteristiky zajišťují, že systém i v kritických situacích minimalizuje rizika a chrání uživatele i prostředí. [13]

Tento model slouží jako hlavní referenční rámec pro tuto práci. Metriky použité v rámci analýzy generovaného kódu jsou inspirovány právě tímto modelem kvality.

Kvalita kódu

Kvalita kódu představuje souhrn charakteristik zdrojového kódu, které ovlivňují jeho efektivitu, čitelnost, přehlednost, udržovatelnost, bezpečnost a rozšiřitelnost. Nejedná se pouze o funkční správnost nebo absenci chyb, ale také o míru srozumitelnosti kódu pro ostatní vývojáře a o to, jak snadno lze kód testovat, modifikovat či opětovně využít. [14]

Hodnocení kvality kódu probíhá pomocí kvantitativních i kvalitativních metrik, přičemž důraz je kladen nejen na technickou stránku, ale i srozumitelnost pro ostatní vývojáře. Pro dosažení vysoké kvality kódu je důležité dodržovat osvědčené postupy, jako je psaní přehledného a konsistentního kódu dle standardů, rozdělení logiky do modulů a tříd, používání smysluplných komentářů, vytváření automatizovaných testů a provádění pravidelných code reviews. [14]

Jedním z klíčových faktorů ovlivňujících kvalitu kódu je technický dluh, který označuje budoucí náklady způsobené používáním krátkodobých řešení. Takové kompromisy často vznikají pod tlakem na rychlé dodání funkčnosti, ale jejich důsledky se mohou projevit v podobě zvýšené chybovosti, horší údržby a složitějšího rozšiřování systému. Tvorba softwaru, který nevytváří technický dluh, vyžaduje rovnováhu mezi rychlým dodáním funkčnosti a dlouhodobou udržitelností kódu. [15]

Kvalitní kód má přímý a měřitelný dopad na úspěšnost softwarových projektů. Studie ukazuje, že kód s vyšší kvalitou vykazuje výrazně nižší chybovost, kratší a předvídatelnější dobu řešení problémů a snižuje technický dluh. Takový kód tvoří pevný základ pro udržitelný vývoj softwaru, zvyšuje produktivitu vývojářů a umožňuje rychlejší a spolehlivější implementaci nových funkcí. [16]

3.3. Velké jazykové modely

Velké jazykové modely představují zásadní pokrok v oblasti umělé inteligence. Jsou postaveny na principech hlubokého učení a využívají pokročilé architektury neuronových sítí, nejčastěji typu Generative Pre-trained Transformer (GPT). Tyto modely staví na transformátorové architektuře, která umožňuje efektivní paralelní zpracování sekvenčních dat a detailní práci s jazykovým kontextem. [17]

Jádrem těchto modelů jsou hluboké neuronové sítě obsahující stovky milionů až miliardy parametrů, které se během trénování optimalizují pomocí rozsáhlých korpusů textových dat. Díky tomu dokáží zachytit složité jazykové struktury, vzorce i významové souvislosti a na základě zadáного vstupu přesně predikovat další slova. Tréninková data obvykle pokrývají široké spektrum témat, což z těchto modelů činí nástroj s širokým potenciálem. [17]

V praxi se velké jazykové modely nejčastěji využívají v úlohách zpracování přirozeného jazyka. Uplatnění nacházejí při generování textu, strojovém překladu, extrakci informací, summarizaci rozsáhlých dokumentů, zodpovídání dotazů v přirozeném jazyce, generování programového kódu nebo při analýze dat. [17]

Zpracování přirozeného jazyka

Zpracování přirozeného jazyka je obor, který se zabývá interakcí mezi počítači a lidským jazykem. Přirozený jazyk, kterým lidé běžně komunikují, je pro počítače nestrukturovaný a obtížně uchopitelný. Aby s ním stroje mohly pracovat, musí být převeden do matematicky a algoritmicky zpracovatelné podoby. Cílem zpracování přirozeného jazyka je umožnit strojům porozumět textu, analyzovat jej a generovat přirozený jazyk. [18]

Dříve byly úlohy v oblasti zpracování přirozeného jazyka řešeny pomocí ručně definovaných pravidel a statistických metod. S nástupem hlubokého učení však došlo k zásadnímu posunu. Dnešní jazykové modely používají neuronové sítě, které se učí významové vztahy mezi slovy na základě obrovského množství trénovacích dat. [18]

Zpracování textu v oblasti zpracování přirozeného jazyka zahrnuje několik standardních kroků. Nejprve dochází k tokenizaci, tedy rozdělení textu na základní jednotky. Následně se text normalizuje, například převedením všech znaků na malá písmena, aby se minimalizovala variabilita způsobená rozdílnou kapitalizací. V dalším kroku se provádí odstranění gramatických slov, která nenesou významovou informaci a eliminace speciálních znaků či jiných nerelevantních symbolů. Výsledný, očištěný a standardizovaný text je poté připraven k dalšímu zpracování pomocí neuronových sítí nebo jiných algoritmů strojového učení. [18]

Technické základy

Pro hlubší pochopení fungování velkých jazykových modelů je důležité se seznámit s několika klíčovými koncepty a technikami, které tvoří jejich základ. Tyto koncepty zahrnují strojové učení, neuronové sítě a transformátory. Všechny tyto termíny se vzájemně prolínají a tvoří základ pro vývoj a tvorbu velkých jazykových modelů [17, 19].

Strojové učení

Strojové učení se zaměřuje na vývoj algoritmů a modelů, které se učí z dat a zlepšují své výkony na základě zkušeností, aniž by byly explicitně naprogramovány. Cílem je, aby algoritmy a modely dokázaly rozpoznávat vzory, klasifikovat data nebo predikovat výstupy na základě vstupů. [19]

Mezi základní přístupy strojového učení patří:

- **Učení s učitelem:** Model se na základě předpřipravených klasifikovaných dat pokouší předpovědět výstup, tak aby nastala shoda se skutečným označením výstupu [20].
- **Učení bez učitele:** Model se učí na základě neoznačených dat, kde se snaží najít skryté struktury nebo vzory v datech bez předem definovaných výstupů [20].
- **Učení s částečným učitelem:** Kombinace přístupů, kdy částečně se model učí na označených i neoznačených datech [20].
- **Zpětnovazební učení:** Přístup, kdy se model učí optimalizovat chování tím, že zkouší různé akce a na základě zpětné vazby upravuje své chování [20].

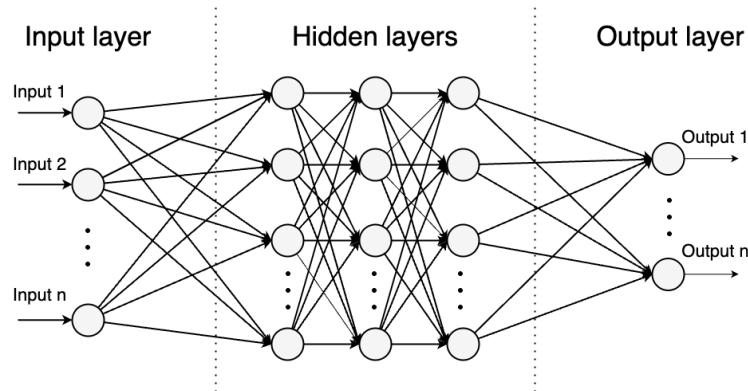
Proces učení v rámci strojového učení je iterativní a lze ho rozdělit do několika částí. První část se zaměřuje na volbu algoritmu, který bude provádět požadovaný úkol. Druhá část zahrnuje chybovou funkci, která měří, jak dobře model funguje. Nejjednodušší chybovou funkcí je porovnání předpovědi modelu s označenými daty. Třetí část zahrnuje optimalizaci vah modelu pro zlepšení jeho výkonu. [19]

Mezi nejčastější úlohy strojového učení patří:

- **Klasifikace:** Předpovídá kategorii nebo třídu, do které daný objekt patří, na základě jeho vlastností [20].
- **Regresce:** Odhaduje spojitou hodnotu na základě vstupních atributů [20].
- **Shlukování:** Rozděluje data do skupin na základě jejich podobnosti bez předem daných tříd [20].

Hluboké učení

Hluboké učení je podmnožinou strojového učení, která se zaměřuje na algoritmy založené na vícevrstvých neuronových sítích. Na rozdíl od tradičních algoritmů strojového učení, které spoléhají na zásah odborníků pro ruční výběr relevantních rysů, umožňuje hluboké učení modelům automaticky extrahat důležité rysy z nestrukturovaných dat bez lidské pomoci. Hluboké učení je schopno zpracovávat složité struktury dat, jako jsou obrázky, zvukové signály nebo texty, a vytvářet pokročilé reprezentace těchto dat. [19]



Obrázek 3.2.: Schéma hluboké neuronové sítě. Vlastní zpracování dle [19].

Neuronové sítě

Neuronové sítě jsou matematické modely inspirované biologickými neuronovými sítěmi v lidském mozku. Skládají se z uzlů (neuronů), propojení (synapsí) s příslušnými vahami a biasem. Neuronové sítě bývají uspořádány do několika vrstev, jako je vstupní vrstva, jedna nebo více skrytých vrstev a výstupní vrstva. Pokud síť obsahuje více skrytých vrstev, označuje se jako hluboká neuronová síť a její trénování spadá do oblasti hlubokého učení. [19]

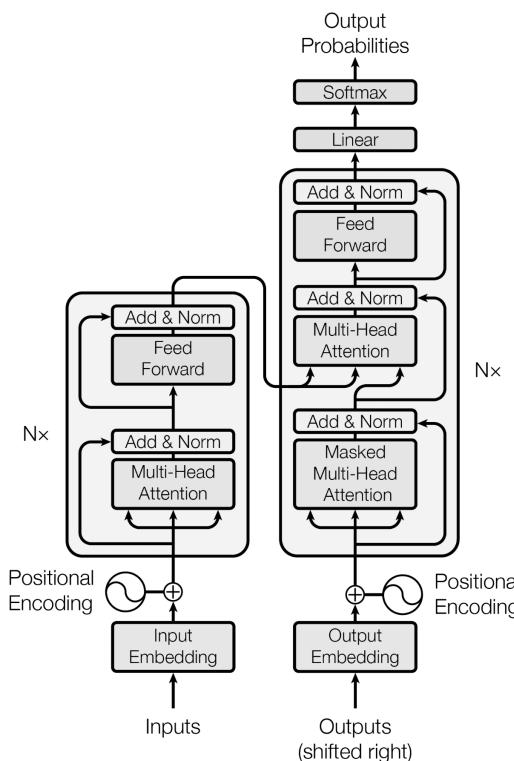
Každý neuron přijímá vstupní signály, provede jejich vážený součet, přičte bias, aplikuje aktivační funkci a výstup předá dál. Tento proces se vrstvou po vrstvě opakuje, dokud není získán výstup modelu. Jedním z nejdůležitějších pokroků v oblasti neuronových sítí je vznik transformátorové architektury, která zásadně zlepšila zpracování jazykových dat. [19]

Transformátor

Transformátory představují architekturu neuronových sítí, která se stala základem většiny moderních velkých jazykových modelů. Tato architektura je tvořena enkodérem, který slouží ke zpracování a interpretaci vstupní sekvence, a dekodérem, jenž na základě reprezentace vytvořené enkodérem generuje odpovídající výstupní sekvenci. [21, 22]

Jedním z důležitých prvků této architektury je mechanismus pozornosti, konkrétně tzv. self-attention, který umožňuje efektivní zpracování sekvenčních dat. Sekvenční data představují takové typy vstupů, u nichž je významně důležité pořadí jednotlivých prvků. Změna jejich pořadí může zásadním způsobem ovlivnit výsledný význam. Mechanismus pozornosti funguje nad třemi složkami přiřazenými každému prvku vstupní sekvence a to dotazem, klíčem a hodnotou. Model pro každý token ve vstupu vypočítává míru podobnosti mezi dotazem a klíči všech ostatních slov, čímž určuje, na které části sekvence by se měl při výpočtu reprezentace soustředit. Tyto podobnosti jsou následně normalizovány pomocí softmax funkce, která převádí skóre na pravděpodobnostní rozdělení. [21]

Pro zachycení komplexnějších vztahů mezi prvky sekvence využívají transformátory tzv. multi-head attention. Tento mechanismus provádí výpočet pozornosti paralelně ve více hlavách, přičemž každá pracuje s jinou projekcí dotazů, klíčů a hodnot. Díky tomu je model schopen zachytit různorodé typy závislostí a jemné vztahy mezi slovy. Tento přístup umožňuje modelu lépe rozpoznat významové souvislosti a soustředit se na relevantní části vstupního textu i v případě velmi dlouhých sekvencí, přičemž vzdálenost mezi slovy nehráje roli. [21]



Obrázek 3.3.: Schéma architektury transformátoru. Převzato z [21].

Další významnou výhodou transformátorů je jejich schopnost paralelního zpracování celé vstupní sekvence, na rozdíl od dřívějších architektur, jako jsou rekurentní neuronové sítě, které operovaly sekvenčně a postupně přenášely informace mezi jednotlivými kroky. Díky paralelizaci lze výrazně urychlit jak proces trénování modelu, tak generování textu. [22]

Principy velkých jazykových modelů

Pro úspěšné fungování velkých jazykových modelů je klíčové několik principů, které se vzájemně doplňují a tvoří základ pro jejich efektivní učení a generování textu. Mezi tyto principy patří tokenizace, embedding a trénování.

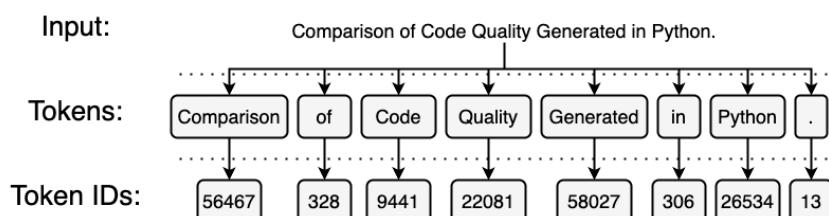
Tokenizace

Tokenizace je proces, při kterém se text rozděluje na menší části zvané tokeny. Token může odpovídat celému slovu, části slova nebo jednotlivému znaku. Tento krok je zásadní pro přípravu textu ke zpracování velkými jazykovými modely, protože umožňuje převést text do strukturované podoby, se kterou modely efektivně pracují. Namísto složité analýzy celého textu najednou se model zaměřuje na jednotlivé tokeny. [23]

Tokenizace se používá jak při trénování modelu, tak při generování textu. V obou případech je text převeden na číselnou reprezentaci, přičemž každému unikátnímu tokenu je přiřazena jednoznačná hodnota. Tyto číselné hodnoty se následně transformují do vektorové podoby pomocí procesu zvaného embedding. Výsledné vektory zachycují význam a kontext tokenů a slouží jako vstupní data pro model, který na jejich základě analyzuje vstup a vytváří výstup. [23]

Při generování textu model postupně předpovídá další tokeny na základě těch předchozích. V každém kroku vygeneruje několik možných variant a vybírá tu s nejvyšší pravděpodobností. Vybraný token se přidá k dosavadní sekvenci a celý proces se opakuje, dokud není výstup dokončen. [23]

Při práci s jazykovými modely je důležité zohlednit omezení vyplývající z tokenizace. Modely mají limitovaný počet tokenů, které jsou schopny zpracovat v jednom okamžiku, což přímo ovlivňuje maximální délku vstupních i výstupních textů. Zároveň bývá využití modelů zpoplatněno na základě počtu vstupních a výstupních tokenů, což má přímý dopad na náklady spojené s jejich praktickým nasazením. [23]

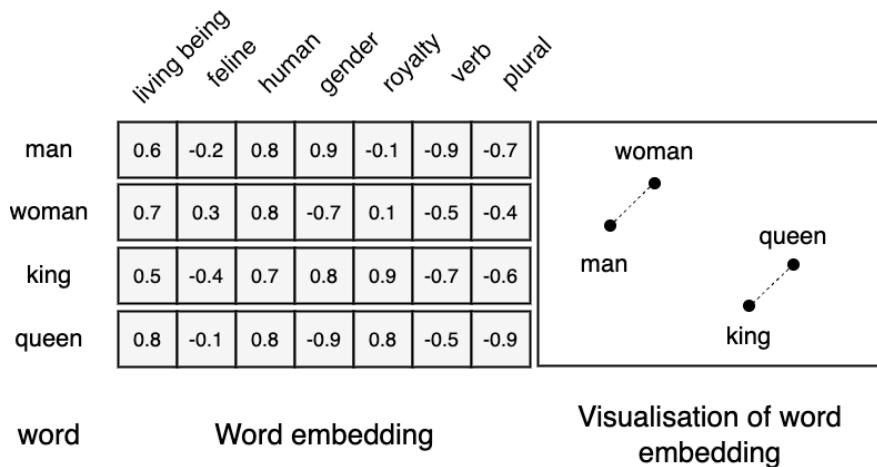


Obrázek 3.4.: Ilustrace tokenizace. Vlastní zpracování dle [24].

Embedding

Embedding představuje matematickou reprezentaci objektů, jako jsou slova, obrázků či jiné entity ve formě vektorů v mnohorozměrném vektorovém prostoru. Cílem této reprezentace je zachytit sémantické nebo strukturální vztahy mezi jednotlivými objekty tak, aby si podobné entity odpovídaly vektory s malou vzájemnou vzdáleností, zatímco rozdílné a nesouvisející objekty byly ve vektorovém prostoru od sebe výrazně vzdálené. Tímto způsobem lze například modelovat jazykové nebo vizuální podobnosti a kontextové souvislosti. [25]

Každý objekt je embeddingem převeden na numerický vektor, kde jednotlivé složky odpovídají různým vlastnostem objektu. Generování embeddingu se typicky realizuje prostřednictvím hlubokých neuronových sítí, které se trénují na rozsáhlých datech s cílem automaticky extrahovat skryté vzory mezi objekty. Tato metoda výrazně překonává tradiční přístupy založené na ručním zpracování, zejména díky schopnosti pracovat s vysoce dimenzionálními reprezentacemi, které mohou dosahovat i tisíců rozměrů v závislosti na složitosti modelovaného prostoru. [25]



Obrázek 3.5.: Ilustrace embeddingu. Převzato z [26].

Trénování

Velké jazykové modely se učí porozumět textu na základě rozsáhlých datových souborů. Jejich trénink probíhá ve dvou hlavních fázích předtrénování a doladění. Každá fáze má odlišný účel a přístup, ale navzájem se doplňují. [27]

Trénování modelu je iterační proces, při kterém se model opakovaně vystavuje textovým vstupům, na jejichž základě postupně upravuje své vnitřní parametry, aby minimalizoval chybu ve výstupech. Výsledkem je model, který dokáže porozumět přirozenému jazyku a generovat smysluplné a relevantní výstupy. [27]

1. **Předtrénování (Pre-training):** V této fázi se model učí z velkého množství neoznačených textových dat. Cílem je naučit ho jazykové vzory, strukturu a kontexty vět. Model se učí předpovědět následující slovo nebo token v textu. Tato fáze je velmi náročná na výpočetní výkon a může trvat týdny až měsíce. [27]

2. **Doladění (Fine-tuning):** Po předtrénování se model přizpůsobuje konkrétním úkolům nebo doménám. Učí se z menšího množství označených dat, která jsou relevantní pro daný problém. Tím dojde k zlepšení jeho výkonu v dané problematice. Tento proces je rychlejší než předtrénování, protože model již má obecné jazykové znalosti. Doladění je nezbytné pro dosažení kvalitních výstupů při práci s instrukcemi, protože umožňuje modelu lépe reagovat na konkrétní pokyny. [27]

Specifickou metodou pro optimalizaci výstupů generovaného velkými jazykovými modely je Retrieval Augmented Generation (RAG). Jedná se o přístup, který nezahrnuje další trénink modelu, ale zlepšuje výstupy pomocí externích znalostí. Model si během generování vyhledává relevantní informace například z dokumentů nebo databází, které následně využívá k tvorbě kvalitnějších odpovědí. Výsledkem je vyšší přesnost a relevance výstupů bez nutnosti měnit samotný model. [27]

Zlepšení kvality výstupů velkých jazykových modelů je možné také pomocí technik inženýrství výzev, které se zaměřují na optimalizaci pokynů a kontextu se kterým model pracuje. [28]

3.4. Inženýrství výzev

Inženýrství výzev je disciplína zaměřená na návrh a optimalizaci vstupů, které jsou předkládány velkým jazykovým modelům. Jeho hlavním cílem je vytvořit takové vstupy, které poskytnou modelu co nejpřesnější, nejúplnější a nejrelevantnější kontext, instrukce či příklady. [28]

Inženýrství výzev hraje klíčovou roli při použití generativní umělé inteligence v různých úkolech. To zahrnuje úlohy spojené se zpracováním přirozeného jazyka, jako je automatické generování textu, strojový překlad, summarizace, klasifikace textu nebo odpovědi na otázky. Využití se však neomezuje pouze na přímou práci s textem. Modely dnes dokáží generovat také kód nebo obrázky. V kontextu této práce je zvláště významné využití velkých jazykových modelů při generování a analýze zdrojového kódu. V rámci programování lze modely využít k návrhu kódu v různých programovacích jazycích, jeho doplnování, refactoring, ladění, stejně jako tvorbu dokumentace, komentářů nebo testů. [28]

Vhodně formulované výzvy zvyšují pravděpodobnost na generování relevantního výstupu jazykovým modelem, a to jak v případě jednoduchých dotazů, tak u komplexních úloh. Výsledkem jsou přesnější a konzistentnější odpovědi, při současném snížení rizika zkreslení nebo generování nevhodného obsahu. [28]

Výzva

Výzva je vstup, který je předložen velkému jazykovému modelu s cílem vyvolat specifickou reakci nebo výstup. Výzva může mít různé formáty od jednoduchých textových dotazů přes instrukce, příklady, až po ukázky kódu nebo jiné netextové formáty. [29]

Klíčovými faktory jsou její struktura, jazyk a rozsah, které významně ovlivňují, jakým způsobem model reaguje. Precizně formulovaná výzva dokáže výrazně zvýšit kvalitu, relevantnost a spolehlivost výstupů. Formulace výzvy však nezahrnuje pouze její obsah, ale i jazyk, ve kterém je formulována. [29]

Volba jazyku představuje významným faktorem ovlivňujícím kvalitu generovaného výstupu. Přestože jsou současné velké jazykové modely trénovány na vícejazyčných datech, angličtina zůstává primárním a nejlépe optimalizovaným jazykem těchto modelů. Formulace výzev v angličtině tak zpravidla vede k přesnějším a kvalitnějším výsledkům. [30]

Ukázka výzvy:

Write a long story about the men who first landed on the moon.

Výzva 3.1: Ukázka výzvy

Struktura výzvy

Každá výzva může obsahovat několik komponent, které dohromady určují jakým způsobem bude model reagovat na zadání. Každá výzva nemusí mít všechny tyto komponenty. [31]

Mezi hlavní součásti výzvy patří:

- **Instrukce:** Definice úkolu, který má model splnit [31].
- **Kontext:** Dodatečné informace, které modelu pomáhají lépe porozumět zadání [31].
- **Vstupní data:** Konkrétní hodnoty, příklady nebo jiná data, které slouží jako základ pro generování výstupu [31].
- **Výstupní formát:** Specifikace požadovaného formátu výstupu [31].

Strategie pro získání kvalitnějších výstupů

Pro dosažení maximální efektivity výzev je vhodné se řídit několika ověřenými strategiemi. Úspěšné vytváření výzev vyžaduje systematický přístup a často i iterativní experimentování. [28]

Mezi osvědčené strategie patří:

- **Formulace cílů a očekávání:** Výzva by měla přesně specifikovat požadovaný výstup, aby model věděl, co se od něj očekává [28].
- **Poskytnutí kontextu:** Přidáním faktických údajů, popisů nebo relevantních informací lze zvýšit přesnost odpovědi [28].
- **Jasná formulace:** Nejednoznačné nebo příliš obecné výzvy mohou vést k irelevantním výsledkům [28].
- **Použití příkladů a ukázků:** Uvedení několika vstupů a odpovídajících výstupů pomáhá modelu porozumět očekávanému chování [28].

- **Iterativní ladění:** Testování různých variant výzev, změna pořadí vět, použití synonym nebo upřesnění formulací často vede k výraznému zlepšení výstupů [28].

Tyto obecné principy tvoří základ pro tvorbu efektivních výzev a uplatňují se i při vytváření výzev pomocí technik inženýrství výzev.

Techniky výzev

Techniky návrhu výzev představují systematické přístupy ke strukturování a formulaci vstupů, jejichž cílem je maximalizovat kvalitu, přesnost a relevanci výstupů generovaných velkými jazykovými modely. Tyto metody vycházejí z hlubšího porozumění principům fungování modelů a znalosti dat, na nichž byly trénovány. [32]

Zero-shot Prompt

Zero-shot výzva je jednou z jednodušších forem zadání. Model obdrží pouze samotný úkol bez jakýchkoliv příkladů nebo dalších detailů. Tento přístup se hodí zejména pro jednodušší úlohy, protože neposkytuje širší kontext ani demonstraci řešení a spoléhá tak na schopnosti modelu. [32]

Ukázka výzvy:

Classify the sentiment of the following review as either positive, neutral, or negative.

Text: The product is amazing. I love it.

Výzva 3.2: Zero-shot. Vlastní zpracování dle [33].

Model je požádán o klasifikaci sentimentu daného textu. Pokyny jsou explicitní, bez jakýchkoliv příkladů, což klade důraz na schopnost modelu zobecňovat na základě naučených vzorů.

Přidáním ukázkových příkladů lze výzvu transformovat na variantu One-shot nebo Few-shot. [33]

Few-shot Prompt

Tato technika je založena na poskytnutí několika reprezentativních příkladů, které slouží k nařízení chování modelu k požadovanému způsobu řešení a pochopení očekávaného formátu výstupu. Na základě počtu těchto příkladů se rozlišují varianty jako One-shot (jeden příklad), Two-shot (dva příklady) a obecnější označení Few-shot, které popisuje situace, kdy je modelu předloženo více příkladů. [32]

Ukázka výzvy:

Classify the sentiment of the following review as either positive, neutral, or negative.

Text: The product is amazing. I love it.

Examples:

Amazing product! Exceeded my expectations. - Positive

Terrible quality. Completely disappointed. - Negative

It works fine, but nothing special. - Neutral

Výzva 3.3: Few-shot. Vlastní zpracování dle [34].

Model je instruován klasifikovat sentiment na základě poskytnutých příkladů, které ilustrují požadovaný výstup. Tím se zvyšuje pravděpodobnost správné interpretace zadání.

Ani technika few-shot learning není ideální. Tento přístup může selhávat u složitějších úloh, které vyžadují hlubší logické uvažování. V takových případech je vhodné využít pokročilejší metody, jako je například Chain-of-Thought, která vede model k tomu, aby rozdělil výpočet na menší, lépe zvládnutelné kroky. [34]

Role Prompt

Role-based výzvy přidělují modelu konkrétní roli, čímž ovlivňují styl, rozsah i zaměření generované odpovědi. Tento přístup je velmi účinný, pokud je třeba zachovat odborný tón nebo specifický komunikační styl. [32]

Ukázka výzvy:

I want you to act as a restaurant guide in Prague. I will tell you my preferences, such as the type of cuisine I'm in the mood for, dietary restrictions, or the part of Prague I'm currently in. You will recommend 3 suitable restaurants based on that information. Each recommendation should include the name of the restaurant and a brief description.

My suggestion: I'm near Prague 1 and I'm looking for a vegetarian-friendly place with a cozy atmosphere.

Výzva 3.4: Role Prompt. Vlastní zpracování dle [32].

Model je instruován vystupovat jako průvodce doporučující vhodné restaurace na základě zadaných preferencí. Tím je zajištěno zaměření odpovědi na relevantní aspekty, jako jsou lokalita a specifické požadavky uživatele.

Chain-of-Thought Prompt

Technika Chain-of-Thought podporuje schopnost modelu provádět vícestupňové logické uvažování při řešení složitějších úloh. Místo okamžitého vygenerování finální odpovědi je model veden k tomu, aby své řešení strukturoval do jednotlivých myšlenkových kroků. Klíčovým prvkem této techniky je výzva k postupnému přemýšlení, často iniciovaná frází „Let's think step by step.“ [32]

Tato metoda zvyšuje transparentnost procesu rozhodování, což je přínosné zejména v případech, kdy je třeba ověřit logickou platnost výstupu. Na druhé straně však tento přístup vede ke zvýšení počtu generovaných tokenů, což zároveň prodlužuje dobu odezvy. [32]

Ukázka výzvy:

When I was 5 years old, my older sibling had 4 times more money than I did. Now , I'm 25 years old and I have \$1,000. How much money would my sibling have now if the same ratio stayed the same? Let's think step by step.

Výzva 3.5: Chain-of-Thought. Vlastní zpracování dle [32].

Model je veden k tomu, aby nejprve provedl analýzu a vyhodnocení jednotlivých kroků, než dospěje k závěrečnému výsledku.

Pokročilé techniky výzev

Kromě základních výše definovaných přístupů existují i sofistikovanější strategie pro návrh výzev, které mohou dále zvýšit kvalitu generovaných výstupů. Tyto techniky často využívají kombinaci více kroků nebo zavádějí mechanismy pro zvýšení robustnosti a konzistence odpovědí. [32]

- **Step-back prompting:** Model je instruován, aby nejprve reflektoval širší kontext, než přistoupí k samotnému řešení [32].
- **Self-consistency:** Model generuje více nezávislých řešení, přičemž finální odpověď je vybrána na základě nejčastěji se opakujících nebo nejvíce konzistentních výsledků [32].
- **ReAct (Reasoning and Acting):** Metoda, která kombinuje logické uvažování s vykonáváním akcí, jako je vyhledávání informací nebo spouštění a interpretace kódu [32].
- **Tree of Thoughts:** Model rozvíjí více paralelních myšlenkových větví pro řešení úkolu, přičemž následně vybírá tu nejvhodnější na základě definovaných kritérií [32].

Optimalizace výzev

Součástí moderních přístupů je i snaha o automatizaci návrhu výzev. Příkladem je metoda Automatic Prompt Engineering (APE), která generuje různé varianty výzev, experimentálně je vyhodnocuje a selektuje tu nejfektivnější. Tento iterativní přístup umožňuje postupnou optimalizaci výzvy za účelem dosažení maximálního výkonu modelu. [32]

4. Praktická část

Tato část práce se zaměřuje na praktickou realizaci výzkumu zaměřeného na hodnocení kvality kódu generovaného vybranými velkými jazykovými modely. Kapitola obsahuje definici zvolené metodiky a podrobný popis jednotlivých fází výzkumného procesu. Součástí jsou informace o výběru modelů, testovaných úlohách, nebo o nástrojích použitých pro analýzu. Cílem této části je poskytnout systematický přehled kroků podniknutých za účelem zjištění schopnosti jednotlivých velkých jazykových modelů generovat kvalitní zdrojový kód v jazyce Python.

4.1. Výzkumné otázky

Výzkumné šetření vychází z následujících výzkumných otázek:

1. *Jak různé techniky inženýrství výzev ovlivňují kvalitu kódu generovaného velkými jazykovými modely?*
2. *Jak se liší kvalita generovaného kódu mezi vybranými velkými jazykovými modely?*

Tyto otázky slouží jako výchozí rámec pro návrh výzkumné metodiky, sběr dat a interpretaci výsledků.

4.2. Metodika výzkumu

Na základě definovaných výzkumných otázek byla navržena metodika výzkumu vycházející z experimentálního přístupu. Ten spočívá v systematickém testování různých modelů a technik výzev za účelem pozorování dopadů na kvalitu generovaného kódu. Výzkumný proces byl rozdělen do několika navazujících kroků, které byly realizovány v rámci praktické části práce.

Tyto kroky zahrnují:

1. Výběr velkých jazykových modelů
2. Výběr technik inženýrství výzev
3. Explorační analýza
4. Výběr testovaných úloh a tvorba výzev
5. Výběr charakteristik kvality kódu a tvorba testovacích skriptů

6. Generování kódu různými velkými jazykovými modely
7. Analýza kvality kódu
8. Vyhodnocení kvality generovaného kódu

Kritéria hodnocení

Hodnocení kvality vygenerovaného kódu bylo realizováno na základě vybraných charakteristik kvality kódu, inspirovaných standardem ISO/IEC 25010. Tyto charakteristiky byly rozpracovány do konkrétních podcharakteristik, které umožňují detailnější analýzu prostřednictvím měřitelných metrik. Jednotlivé metriky byly hodnoceny odděleně, přičemž výsledky byly následně porovnávány napříč jazykovými modely a variantami výzev.

Vzhledem k rozdílnému charakteru jednotlivých metrik nebylo cílem vytvořit souhrnné skóre reprezentující celkovou kvalitu kódu. Každá metrika má totiž specifický interpretační rámec. Zatímco u některých platí, že vyšší hodnota značí lepší výsledek (např. úspěšnost testů či statistická analýza kódu), u jiných je naopak preferována nižší hodnota (např. časová náročnost).

Z těchto důvodů byly metriky analyzovány individuálně a až následně bylo provedeno hledání souvislostí mezi výsledky v rámci jednotlivých podcharakteristik kvality.

Testovací prostředí

Hodnocení kvality vygenerovaného kódu bylo realizováno na lokálním zařízení Apple MacBook Pro s čipem M1 Pro, vybaveném 16 GB operační paměti a 512GB SSD úložištěm. Výpočetní prostředí běželo pod operačním systémem macOS Sequoia ve verze 15.3.1.

Každá kombinace velkého jazykového modelu, typu výzvy a testované úlohy byla podrobena deseti nezávislým iteracím za účelem zachycení variability výstupů jednotlivých modelů. Tento přístup umožňuje ilustrovat rozdíly v kvalitě generovaného kódu, avšak neumožňuje formulovat statisticky podložené závěry.

Pro přesné vymezení testovacího prostředí je nezbytné specifikovat hodnoty parametrů, které ovlivňují způsob generování výstupů velkými jazykovými modely. Tyto parametry zásadně ovlivňují míru kreativity, rozmanitosti a opakovatelnosti výstupů. Mezi klíčové patří zejména temperature, top_p, frequency_penalty a repetition_penalty.

Parametr temperature, jenž určuje míru náhodnosti při výběru tokenů, byl nastaven na hodnotu 1.0. Tato úroveň podporuje mírně rozmanitější výstupy, kdy vyšší hodnoty obecně vedou ke kreativnějším a méně předvídatelným výstupům. Parametr top_p, který omezuje výběr tokenů pouze na určité procento nejpravděpodobnějších tokenů, byl rovněž nastaven na 1.0, což modelu ponechává maximální volnost při generování.

Parametr `frequency_penalty`, který snižuje pravděpodobnost výběru tokenů opakujících se s vysokou četností ve výstupu, byl nastaven na 0.0. V praxi to znamená, že model nebyl nijak omezován v opakování stejných slov, což je při generování kódu žádoucí vzhledem k častému výskytu klíčových výrazů. Parametr `repetition_penalty`, sloužící k potlačení opakování stejných frází, byl nastaven na hodnotu 1.0, což odpovídá neutrálnímu nastavení bez zvýšeného postihu za opakování.

Pro vyhodnocení vybraných podcharakteristik kvality kódu byly vytvořeny testovací skripty v programovacím jazyce Python. Evaluace těchto metrik byla provedena nad verzí Pythonu 3.12, přičemž testování probíhalo v izolovaném virtuálním prostředí, které zajišťovalo konzistentní podmínky a kontrolu nad verzemi použitých knihoven.

Jednotlivé testovací skripty byly spouštěny sekvenčně, což minimalizovalo riziko vzájemného ovlivnění běhů. Tento způsob provádění zároveň umožňoval přesné sledování časové náročnosti generovaného kódu v jednotlivých iteracích.

4.3. Výběr velkých jazykových modelů

Cílem tohoto výzkumu bylo porovnat nejpokročilejší aktuálně dostupné velké jazykové modely, u nichž je programování uváděno jako jedno z doporučených použití. Výběr se nezaměřoval na úzce specializované modely určené pouze pro generování programového kódu, ale spíše univerzální jazykové modely schopné generovat přirozený text, které zároveň disponují výrazným potenciálem v oblasti tvorby funkčního kódu v jazyce Python.

Zařazení modelů do tohoto výzkumu bylo podmíněno splněním několika praktických kritérií. Jedním z faktorů byla dostupnost modelu, kdy byly zkoumány pouze veřejně přístupné modely, jejichž využití nevyžaduje zvláštní oprávnění či exkluzivní přístup. Dále byl kladen důraz na to, aby bylo programování uvedeno jako podporovaný nebo doporučený scénář použití daného modelu, což zvyšuje pravděpodobnost kvalitních výstupů v dané oblasti. Specifickým požadavkem byla dostupnost aplikačního rozhraní (API), které umožňuje programovou interakci s modelem.

Na základě těchto kritérií byly do experimentální části výzkumu zařazeny následující modely: OpenAI o3-mini-high, Anthropic Claude 3.7 Sonnet, Google Gemini 2.0 Pro Experimental, Mistral Large 2 a xAI Grok 2.

4.4. Výběr technik inženýrství výzev

Pro účely tohoto výzkumu byly vybrány techniky Structured Prompt, Chain-of-Thought Prompt a Role Prompt. Každý z těchto přístupů formuluje výzvu k danému úkolu z odlišné perspektivy a využívá specifické strategie k jejímu zpřesnění a optimalizaci.

Structured Prompt se zaměřuje na vytvoření jasného a strukturovaného zadání, které modelu poskytuje jednoznačné instrukce. Chain-of-Thought Prompt podporuje krokové myšlení modelu tím, že ho vede k logickému uvažování a postupnému vyvozování řešení. Role Prompt pak navádí model k přijetí určité identity, čímž mu poskytuje odborný kontext pro aplikaci odborného přístupu a hlubší porozumění úloze.

Výběr těchto technik byl motivován snahou o relativně malou úpravu původní výzvy, přičemž se očekávalo, že i tyto malé změny mohou vést k výraznějšímu zlepšení kvality generovaného kódu. Také byla zohledněna jejich snadná implementace, která umožňuje jejich efektivní využití jak v rámci experimentálního výzkumu, tak v běžné praxi.

4.5. Explorační analýza

Explorační analýza byla realizována jako pilotní fáze výzkumu, jejímž cílem bylo ověřit, zda navržená metodika a přístupy umožňují identifikovat měřitelné rozdíly v kvalitě kódu generovaného jednotlivými jazykovými modely.

Analýza byla realizována iterativním způsobem, kdy docházelo k postupné úpravě výzev i ke zpřesňování měřených podcharakteristik kvality kódu. Každý cyklus přinesl nové poznatky, které vedly k úpravám výzev na základě konkrétních výsledků měření i praktických zkušeností s velkými jazykovými modely. Tento přístup umožnil systematicky sledovat dopad jednotlivých změn na generované výstupy a hodnotit jejich vliv na kvalitu kódu.

Do explorační analýzy byly zvoleny následující jazykové modely: OpenAI o3-mini-high, Anthropic Claude 3.7 Sonnet, Google Gemini 2.0 Pro Experimental, Mistral Large 2 a xAI Grok 2.

Explorace

Explorační fáze výzkumu byla realizována na základě implementace jednoduché konzolové aritmetické kalkulačky v programovacím jazyce Python a následného testování vybraných podcharakteristik kvality kódu.

1. iterace: základní výzva

První výzva byla formulována jako stručný odstavec v angličtině s obecnými požadavky na funkcionality kalkulačky. Tato výzva sloužila jako výchozí bod pro zkoumání vlivu různých změn výzvy na kvalitu generovaného kódu.

Výsledky:

Vybrané velké jazykové modely byly ve většině případů schopny vygenerovat syntakticky validní kód bez závažných chyb, který bylo možné přeložit. Výraznější rozdíly se začaly projevovat při ověřování funkční správnosti. Modely Mistral Large 2 a xAI Grok 2 dosahovaly nízké úspěšnosti,

kdy v opakovaných testech nepřesahovala míra správně fungujících výstupů 50 %. Ani model Google Gemini 2.0 Pro Experimental negeneroval obvykle plně funkční kód, ale oproti zmíněnými modelům vykazoval vyšší míru variability a pravidelně je překonával.

Testování dále ukázalo, že při opakovaném použití identické výzvy generují jazykové modely odlišné implementace. Tato skutečnost byla potvrzena nejen vizuální inspekcí výstupního kódu, která odhalila variabilitu přístupů jednotlivých modelů při řešení téhož zadání, ale také rozdílnými výsledky funkční správnosti.

Na základě těchto zjištění byly modely xAI Grok 2 a Mistral Large 2 vyřazeny z dalších iterací testování.

2. iterace: strukturovaná výzva rozdělená do seznamu požadavků

Ve druhé iteraci byla původní výzva přeformulována do strukturované podoby, v níž byly jednotlivé požadavky přehledně rozděleny do kategorií formou seznamu. Cílem této úpravy bylo zvýšení srozumitelnosti výzvy a minimalizace rizika její nesprávné interpretace.

Formulace výzvy:

Generate high-quality Python code for console-based arithmetic calculator that meets the requirements of ISO/IEC 25010.

Functionality Requirements:

[embedded functionality details]

Implementation Requirements:

[embedded implementation details (interfaces, functions, parameters, etc.)]

Code Quality Requirements:

[embedded code quality attributes]

The generated code must be clean, efficient, and easy to maintain.

Výzva 4.1: 2. iterace

Výsledky:

Strukturovaná podoba výzvy nevedla k výraznému zlepšení kvality výstupů napříč modely. Největší přínos byl zaznamenán u modelu OpenAI o3-mini-high, který dosáhl větší funkční správnosti. Výstupy ostatních modelů zůstaly na podobné úrovni jako v předchozí iteraci.

Statická analýza generovaného kódu ukázala, že i přes formální splnění zadání modely často nedodržovaly zásady psaní čitelného, modulárního a dokumentovaného kódu. Výsledky rovněž ukázaly na rozdíly mezi jednotlivými modely v míře dodržení těchto zásad.

Na základě zjištěného zlepšení u modelu o3-mini-high bylo rozhodnuto zachovat strukturovanou podobu výzvy i v následujících iteracích.

3. iterace: přidání ilustračních ukázek kvalitního kódu

Ve třetí iteraci byla výzva rozšířena o konkrétní příklady dobré strukturovaného a komentovaného kódu, které měly modelům sloužit jako referenční vzor pro generovaný výstup.

Formulace výzvy:

Generate high-quality Python code for console-based arithmetic calculator that meets the requirements of ISO/IEC 25010.

Functionality Requirements:

[embedded functionality details]

Implementation Requirements:

[embedded implementation details (interfaces, functions, parameters, etc.)]

Code Quality Requirements:

[embedded code quality attributes]

Use the following examples as references for code quality and structure:

[embedded quality code samples]

The generated code must be clean, efficient, and easy to maintain.

Výzva 4.2: 3. iterace

Výsledky:

Ukázkové příklady vedly v některých případech k mírnému zlepšení, nicméně efekt nebyl konzistentní napříč modely a opakovanými iteracemi stejné výzvy. Vzhledem k těmto zjištěním bylo rozhodnuto, že v hlavní části výzkumu budou testovány varianty výzev jak s ukázkovým kódem, tak bez něj, aby bylo možné lépe posoudit jejich vliv na kvalitu výstupů.

Výsledná výzva z této iterace zároveň posloužila jako šablona pro tvorbu dalších výzev v rámci hlavní části výzkumu.

Shrnutí poznatků z explorační analýzy

Na základě získaných výsledků byly pro hlavní fázi výzkumu zvoleny tři modely s nejvyšší funkční správností: OpenAI o3-mini-high, Anthropic Claude 3.7 a Google Gemini 2.0 Pro Experimental.

Zjištění z explorační analýzy ukázala, že:

1. Výstupy jazykových modelů nejsou deterministické, kdy opakované použití stejného výzvy může vést k odlišným výsledkům.
2. Většina modelů generuje kód, který lze přeložit.

3. Funkční správnost se výrazně liší mezi jednotlivými modely, tak i při opakovaném použití stejné výzvy.
4. Struktura a srozumitelnost výzvy ovlivňuje kvalitu výsledného kódu.
5. Modely často nedodržují osvědčené zásady pro tvorbu čitelného, modulárního a dokumentovaného kódu.

Všechna zjištění vyplývající z explorační analýzy jsou založena na individuálních pozorováních provedených během této fáze výzkumu, které nebyly podrobeny statistickému testování. Přesto jednoznačně naznačují, že mezi jednotlivými modely i výzvami existují významné rozdíly a má smysl provádět další výzkum v této oblasti.

4.6. Výběr testovaných úloh a tvorba výzev

V rámci této části výzkumu byly vybrány testované úlohy, jejichž cílem bylo pokrýt široké spektrum programovacích dovedností a umožnit tak komplexní vyhodnocení kvality kódu generovaného jazykovými modely. Pro každou z vybraných úloh bylo následně vytvořeno více variant výzev, které reprezentují různé přístupy k formulaci zadání.

Výběr testovaných úloh

Pro účely tohoto výzkumu byly zvoleny tři úlohy lišící se svou náročností i zaměřením. Tyto úlohy pokrývají základní aspekty programování v jazyce Python od aritmetických výpočtů, přes práci s textovým výstupem až po správu strukturovaných dat.

Zvolený soubor úloh zahrnuje:

- **Kalkulačka (Calculator):** Cílem úlohy je vytvořit konzolovou kalkulačku, která umožní provádět základní aritmetické operace nad zadáným výrazem v textové podobě. Tato úloha ověřuje schopnost modelu správně pracovat se syntakticky složitějším vstupem, dodržet pravidla precedencí operátorů a implementovat řešení v objektově orientovaném stylu bez použití vestavěné funkce eval().
- **Konzolové vykreslování (Ascii Art):** Druhá testovaná úloha se zaměřuje na schopnost jazykového modelu generovat kód pro kreslení jednoduchých dvourozměrných tvarů pomocí ASCII znaků. Cílem je otestovat práci s iteracemi, podmínkami, manipulaci se znaky a základní validaci vstupů. Všechny tvary musí být generovány pomocí metod objektově orientované třídy a vráceny jako vícerádkové textové řetězce.
- **Úkolníček (To Do list):** Nejkomplexnější úloha, která slouží k ověření schopnosti jazykového modelu generovat objektově orientovanou aplikaci pro správu úkolů. Aplikace musí umožňovat základní operace vytvoření, zobrazení, vyhledání a smazání úkolu. Cílem tohoto úkolu je otestovat práci s datovými strukturami, validaci vstupů a efektivní vyhledávání.

Tento výběr úloh pokrývá konzolové aplikace s odlišnou složitostí i doménou. Úlohy byly navrženy tak, aby byly dostatečně jednoduché pro praktické testování, ale zároveň dostatečně komplexní pro analýzu různých kvantitativních podcharakteristik generovaného kódu.

Tvorba výzev

Pro každou z testovaných úloh bylo navrženo celkem šest variant výzev, které reprezentují různé přístupy a techniky inženýrství výzev. Postup jejich tvorby vycházel z aplikace vybraných strategií na Structured Prompt, který tvořil výchozí bod pro všechny varianty. Všechny výzvy byly formulovány v anglickém jazyce.

1. Structured Prompt

Základní výzva obsahuje přesně formulované požadavky na implementaci úlohy, členěné do několika logických částí oddělených prázdnými řádky. Jednotlivé části zahrnují popis úlohy, seznam požadované funkcionality nebo výčet implementačních detailů. Výzva je navržena jako Zero-shot, tedy bez jakýchkoli příkladů.

Pojem Structured Prompt není v literatuře běžně používán, avšak v rámci této práce je zaveden jako označení pro výzvy, které jsou strukturovány do přehledných bloků s požadavky.

2. Chain-of-Thought Prompt

Varianta základní výzvy, doplněná o instrukci modelu k postupnému uvažování před samotným generováním kódu frází „Let's think step by step“. Tento přístup má za cíl podpořit logické uvažování modelu a vést k lepšímu porozumění úloze. Výzva je rovněž navržena jako Zero-shot.

Let's think step by step:

[Structured prompt]

Výzva 4.3: Chain of thought (Zero-shot)

3. Role Prompt

Rozšířená verze základní výzvy, ve které je model explicitně instruován, aby vystupoval jako zkušený softwarový inženýr specializující se na psaní kvalitního kódu. Tento přístup má podpořit model v dodržování osvědčených postupů a zásad pro psaní kvalitního a udržovatelného kódu. Tato varianta výzvy je také navržena bez ukázkových příkladů.

Role: You are a senior software developer with expertise in writing high-quality, maintainable Python applications. You adhere to the ISO/IEC 25010 standard while following best practices.

Task: [Structured prompt]

Výzva 4.4: Role prompt (Zero-shot)

K výše uvedeným Zero-shot variantám byly dále vytvořeny jejich Few-shot alternativy. Každá z těchto variant obsahuje tři příklady dobře strukturovaného a komentovaného kódu v jazyce Python.

Pro ilustraci struktury jednotlivých výzev je uvedena ukázka typu Structured Prompt (Zero-shot) na úloze kalkulačky. Ostatní testované úlohy využívají obdobnou strukturu výzvy, přičemž se liší pouze definicí úlohy a specifickými požadavky na funkcionalitu a implementaci. Plné znění výzev pro všechny testované úlohy je dostupné v příloze.

Generate high-quality Python code for console-based arithmetic calculator that meets the requirements of ISO/IEC 25010.

Functionality Requirements:

- Implement the following operations: addition (+), subtraction (-), multiplication (*), and division (/)
- Support parentheses ()
- Ensure correct operator precedence
- Accept both integers and floating-point numbers (including negative values)

Implementation Requirements:

- Implement the code using OOP, define a class Calculator and implement following interface within this class
- Implement interface calculate(expression: str) -> float: Evaluates the expression
- Implement validation using built-in error types (e.g., invalid input - unbalanced parentheses, invalid characters, division by zero)
- Do not use eval() or any equivalent methods for evaluating expressions
- Format the outputted code using Markdown code blocks (```python ```)

Code Quality Requirements:

- Correctness: The code must produce expected results for various expressions
- Performance: The code must use an efficient algorithm
- Modularity: Logically separate parts of the code
- Safety: Protection against invalid inputs
- Testability: The code must be easily testable
- Readability and Documentation: Use docstrings, clearly named variables, and comments

The generated code must be clean, efficient, and easy to maintain.

Výzva 4.5: Structured Prompt (Zero-shot) – Kalkulačka

4.7. Výběr testovaných metrik kvality kódu a tvorba testovacích skriptů

Pro hodnocení kvality kódu generovaného velkými jazykovými modely byly vybrány tři klíčové charakteristiky, inspirované normou ISO/IEC 25010. Jedná se o funkční vhodnost, výkonovou efektivitu a udržovatelnost. Každá charakteristika byla dále rozpracována do konkrétních podcharakteristik, které byly měřeny pomocí konkrétních metrik.

Vybrané charakteristiky kvality kódu

- **Funkční vhodnost (Functional Suitability)**
 - Funkční úplnost (Functional Completeness)
 - Funkční správnost (Functional Correctness)
- **Výkonová efektivita (Performance Efficiency)**
 - Časová náročnost (Time Behaviour)
- **Udržovatelnost (Maintainability)**
 - Analyzovatelnost (Analysability)

Tvorba testovacích skriptů

Pro systematické měření jednotlivých podcharakteristik byly navrženy a implementovány automatizované testovací skripty. Tyto skripty, vytvořené v programovacím jazyce Python, umožňují objektivní a reprodukovatelné hodnocení kvality generovaného kódu. Pro každou testovanou úlohu byly vytvořeny samostatné skripty.

Demonstrace testovacích metod je provedena na příkladu implementace kalkulačky. Ukázky testovacích skriptů jsou pro přehlednost zjednodušené (např. odstranění nadbytečných funkcí, odebrání komentářů, atd.), přičemž cílem je ilustrace jádra testovací logiky.

Kompilovatelnost

Test kompilovatelnosti slouží jako vstupní validace před prováděním dalších testů. Ověřuje, zda lze generovaný zdrojový kód úspěšně přeložit do bytekódu bez syntaktických chyb. V případě, že test selže, není možné kód dále analyzovat pomocí ostatních testů, neboť jeho import do testovacích skriptů by vyvolal výjimku.

Pro tento účel byl využit vestavěný modul `py_compile`, který umožňuje kontrolu syntaktické správnosti bez spuštění programu. Tento test detekuje pouze syntaktické chyby a nikoliv chyby vznikající za běhu aplikace.

```

import py_compile

def test_code_compilability(path) -> None:
    py_compile.compile(path, doraise=True)

```

Ukázka kódu 4.1: Kompilovatelnost – Kalkulačka

Funkční vhodnost

Funkční vhodnost je základním předpokladem hodnocení kvality softwaru. Bez jejího zajištění není možné relevantně analyzovat další kvalitativní aspekty.

V rámci této práce se testování funkčnosti zaměřuje na ověření, zda generovaný kód odpovídá specifikaci uvedené ve výzvě. Tato charakteristika je dále členěna do funkční úplnosti a funkční správnosti.

Funkční úplnost

Cílem testu funkční úplnosti je ověřit, zda generovaný kód implementuje všechny požadované komponenty specifikované v zadání. Tyto komponenty zahrnují třídy, metody a funkce, včetně správného typu a počtu parametrů.

Pro tuto analýzu byla využita standardní knihovna inspect, která umožňuje zkoumání struktury objektů v Pythonu. Výstupem tohoto testu je podrobný seznam všech zjištěných nedostatků.

Funkční správnost

Záměrem hodnocení funkční správnost je ověření, zda aplikace vykazuje očekávané chování podle požadované specifikace. Test kontroluje, zda metody a funkce pro dané vstupy vrací správné výstupy a adekvátně reagují na různé chybové stavy.

Pro posouzení funkční správnosti byl použit framework pytest, který umožňuje automatizované testování kódu v Pythonu. Výstupy tohoto testu zahrnují přehled o počtu úspěšných a neúspěšných kontrol, přičemž každý neúspěšný případ je podrobně popsán spolu s příčinou selhání.

Pro každou úlohu byl vytvořen skript, který zahrnuje desítky scénářů. Tyto scénáře byly organizovány podle typu vstupních dat a jednotlivých funkcí. U úlohy kalkulačky bylo připraveno celkem 88 scénářů, pokrývajících různé kombinace aritmetických operací a vstupních hodnot.

Tyto scénáře zahrnují:

- Základní aritmetické operace (sčítání, odčítání, násobení, dělení, složené operace)
- Různé typy čísel (celá čísla a desetinná čísla, záporná a kladná čísla, malá a velká čísla)
- Reakce na neplatné vstupy (dělení nulou, chybějící operátor, neplatné znaky)

```

import pytest

@pytest.fixture
def calc():
    return Calculator()

def test_add_positive(calc):
    assert calc.calculate("1+2") == 3
    assert calc.calculate("1+2+3") == 6
    assert calc.calculate("1000000000+2000000000") == 3000000000
    assert calc.calculate("999999999999999+1") == 1000000000000000

def test_add_negative_parantheses(calc):
    assert calc.calculate("1+(-2)") == -1
    assert calc.calculate("(-1)+2") == 1
    assert calc.calculate("(-1)+(-2)") == -3
    assert calc.calculate("(-1000000000)+(2000000000)") == 1000000000
    assert calc.calculate("999999999999999+(-1)") == 999999999999998

def test_divide_neutral(calc):
    with pytest.raises(ZeroDivisionError):
        calc.calculate("0/0")

```

Ukázka kódu 4.2: Funkční správnost – Kalkulačka

Výkonová efektivita

Výkonová efektivita představuje další důležitou charakteristiku kvality kódu. Pokud jsou výpočetní operace příliš náročné, může dojít k neefektivnímu využití dostupných výpočetních prostředků, což negativně ovlivňuje celkový výkon aplikace.

V rámci této práce byla efektivita hodnocena prostřednictvím podcharakteristiky časové náročnosti.

Časová náročnost

Cílem testu měřícího časovou náročnost je stanovit průměrnou dobu potřebnou k vykonání jednotlivých operací. Tato metrika poskytuje přehled o rychlosti, s jakou je daná implementace schopna provádět požadované výpočty.

Měření probíhá pomocí modulu time, jenž umožňuje zaznamenat rozdíl mezi časem zahájení a ukončení výpočtu. Výsledná hodnota je získána opakováním vykonáváním testované operace

vydělené celkovým počtem provedených iterací. Výsledkem je průměrná doba výpočtu jedné operace vyjádřená v sekundách.

V případě úlohy kalkulačky bylo provedeno 100 000 opakování pro každou testovanou operaci s cílem minimalizovat vliv náhodných výkyvů v měření.

```
import time

def time_operation(instance, method, iterations, expression) -> None:
    try:
        start_time = time.time()
        for _ in range(iterations):
            getattr(instance, method)(expression)
        end_time = time.time()

        print(f"Average time for {method} ({expression}): {((end_time - start_time) / iterations):.20f} seconds")
    except Exception as _:
        print(f"Method {method} ({expression}) failed with error.")
```

Ukázka kódu 4.3: Časová náročnost – Kalkulačka

Udržovatelnost

Udržovatelnost je posledním z hodnocených charakteristik kvality kódu v rámci této práce. Tento faktor zásadně ovlivňuje dlouhodobou životaschopnost softwarových systémů, jejich schopnost adaptace, rozšiřování a efektivní údržby v průběhu času.

V rámci této práce je udržovatelnost vnímána jako klíčový ukazatel dlouhodobé kvality softwaru. Pokud kód není navržen s ohledem na budoucí správu a rozvoj, jeho složitost se postupem času zvyšuje, což vede k rostoucím nákladům na jeho údržbu. Udržovatelnost je hodnocena prostřednictvím podcharakteristiky analyzovatelnosti.

Analyzovatelnost

Analyzovatelnost popisuje, do jaké míry je kód srozumitelný, čitelný a odpovídá obvyklým konvencím a standardům. Hodnocení zahrnuje aspekty jako vhodnost a konzistence názvů proměnných, funkcí a tříd, přítomnost komentářů a dokumentace, logická struktura kódu či dodržování stylových pravidel.

Tato podcharakteristika byla vyhodnocena pomocí nástroje pro statickou analýzu kódu Pylint, který provádí kontrolu syntaxe, stylu i struktury. Výstupem analýzy je číselné skóre v rozsahu nula až deset, které poskytuje celkové zhodnocení kvality daného kódu. Součástí výstupu je také seznam konkrétních varování a chyb, které byly v kódu identifikovány.

```
import inspect
import pylint.lint

def test_code_analysability(module) -> None:
    file = inspect.getfile(module)
    pylint.lint.Run([file])
```

Ukázka kódu 4.4: Statická analýza kódu – Kalkulačka

Pomocné metriky

V rámci tohoto výzkumu byla dále sledována pomocná metrika, která přímo neodpovídá žádné z výše uvedených charakteristik kvality kódu, avšak poskytuje doplňující informace o vlastnostech generovaného kódu.

Počet řádků

Počet řádků představuje jednoduchý, avšak důležitý ukazatel rozsahu a potenciální složitosti kódu. V kontextu generování kódu pomocí jazykových modelů je tento ukazatel užitečný pro posouzení rozsahu generovaného kódu a to jak mezi jednotlivými modely, tak i mezi různými variantami výzev.

Testovací skript byl vyhodnocen pomocí iterace přes jednotlivé řádky zdrojového souboru, přičemž jsou zahrnuty i prázdné řádky a komentáře.

```
import inspect

def count_lines_in_module(module) -> int:
    source_file = inspect.getfile(module)
    with open(source_file, "r", encoding="utf-8") as f:
        return sum(1 for _ in f)
```

Ukázka kódu 4.5: Počet řádků – Kalkulačka

4.8. Generování kódu různými velkými jazykovými modely

Generování kódu bylo realizováno prostřednictvím API rozhraní, které poskytují všechny analyzované velké jazykové modely. Pro komunikaci s jednotlivými modely nebyl použitý přímý přístup, ale byla použita služba openrouter.ai, která nabízí jednotný přístup k široké škále velkých jazykových modelů prostřednictvím unifikovaného API rozhraní. Tento nástroj zjednodušuje práci s mnoha velkými jazykovými modely tím, že není nutná registrace u každého poskytovatele a zároveň odpadá nutnost správy samostatných přístupových údajů pro každou platformu.

Pro systematické generování kódu byl navržen a implementován automatizovaný skript, který zajišťuje opakované odesílání výzev jednotlivým modelům a následné zpracování odpovědí. Skript je konfigurovatelný a umožňuje snadnou úpravu parametrů generování, jako je počet iterací, použité modely nebo obsah výzev.

Celý proces generování kódu probíhá v následujících krocích:

1. Načtení předpřipravené výzvy ze vstupního souboru ve formátu txt.
2. Iterativní odesílání výzvy každému nakonfigurovanému modelu prostřednictvím API. Každý model obdrží výzvu opakovaně dle nastaveného počtu iterací s cílem získat větší vzorek výstupů.
3. Uložení celé odpovědi a extrahovaného kódu z odpovědi pomocí identifikace kódových bloků ve formátu Markdown.

4.9. Vyhodnocení kvality kódu

Pro účely komplexního vyhodnocení všech testovaných podcharakteristik kvality generovaného kódu byl navržen a implementován postup skládající se ze skriptu pro vyhodnocení jednotlivých testovaných podcharakteristik a nástroje pro agregaci výsledků do vhodného formátu.

Základem celého procesu je bash skript, který zajišťuje sekvenční spouštění jednotlivých testovacích skriptů pro každou kombinaci úlohy, iterace, typu výzvy a jazykového modelu. Výstupy jsou ukládány do samostatných souborů ve formátu txt, kdy každý výstupní soubor obsahuje výsledky právě jednoho konkrétního testu, modelu, úlohy a iterace.

Vzhledem k velkému množství takto generovaných výstupů byl dále vytvořen podpůrný Python skript, který slouží k agregaci a transformaci jednotlivých dat do podoby vhodné pro následnou analýzu. Tento nástroj načítá výstupní soubory z jednotlivých testů, extrahuje relevantní výsledky pomocí regulárních výrazů a ukládá je do souhrnného CSV souboru, který slouží jako podklad pro další analýzy a vizualizace.

Výsledné aggregované datasety tvoří klíčový podklad pro srovnání výstupů jednotlivých modelů a technik výzev v dalších částech této práce. Díky automatizaci celého postupu je možné snadno reprodukovat výsledky nebo rozšířit testovací sadu o nové úlohy či modely s minimálními úpravami.

5. Výsledky výzkumu a diskuse

Tato kapitola představuje výsledky provedeného výzkumu a zaměřuje se na interpretaci získaných dat. V úvodní části jsou analyzovány jednotlivé metriky kvality generovaného kódu ve vztahu ke konkrétním testovaným úlohám. Výsledky jsou prezentovány pomocí tabulek a grafických vizualizací, doplněných o komentář a interpretaci pozorovaných trendů. Závěrečná část kapitoly poskytuje souhrnné zhodnocení kvality generovaného kódu a formuluje ucelené odpovědi na stanovené výzkumné otázky.

5.1. Kompilovatelnost

V rámci tohoto testu byla hodnocena schopnost generovaného Python kódu úspěšně projít překladem do bytekódu. Za kompilovatelný kód je považován takový, který neobsahuje žádné syntaktické chyby vedoucí k výjimkám při překladu. Kód, jenž úspěšně neprojde zkompilováním, je klasifikován jako plně nefunkční a není dále zahrnut do vyhodnocení ostatních měřených metrik kvality kódu.

Výsledky jsou prezentovány jak v absolutních hodnotách, tak v procentuálním vyjádření. Vyšší počet úspěšně zkompilovaných výstupů odpovídá lepšímu výsledku. Například kompilovatelnost na úrovni 90 % znamená, že devět z deseti různých verzí kódu, vygenerovaných pod totožnou výzvou, bylo úspěšně přeloženo.

Kalkulačka

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Structured Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Chain-of-Thought (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Chain-of-Thought (Few-shot)	100 % (10/10)	100 % (10/10)	80 % (8/10)

Role Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Role Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	90 % (9/10)

Tabulka 5.1.: Kompilovatelnost – Kalkulačka

Z výsledků kalkulačky lze pozorovat, že většina testovaných modelů generovala kód, který byl plně kompilovatelný. Nejnižší úspěšnost zaznamenal model Google Gemini 2.0 Pro Experimental, který selhal ve třech z celkových 60 případů. Jednotlivé chyby se vyskytly ve výzvách typu Chain-of-Thought (Few-shot) a Role Prompt (Few-shot). Nejčastější příčiny selhání zahrnovaly syntaktické chyby, zejména v oblasti nesprávného odsazení, závorkování a použití čárek. Tyto chyby se vyskytovaly zpravidla v částech, kde model samovolně doplnil demonstraci testování funkčnosti, ačkoli takové chování nebylo výzvou přímo požadováno.

Konzolové vykreslování

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Structured Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	90 % (9/10)
Chain-of-Thought (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Chain-of-Thought (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Role Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Role Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)

Tabulka 5.2.: Kompilovatelnost – Konzolové vykreslování

Také v této úloze modely prokázaly vysokou úspěšnost překladu. Jediná zaznamenaná chyba opět pocházela od modelu Google Gemini 2.0 Pro Experimental, konkrétně při použití Structured Prompt (Few-shot). I v tomto případě se jednalo o chybu v odsazení, která vedla k nekomplilovatelnému výstupu.

Úkolníček

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Structured Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Chain-of-Thought (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Chain-of-Thought (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Role Prompt (Zero-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)
Role Prompt (Few-shot)	100 % (10/10)	100 % (10/10)	100 % (10/10)

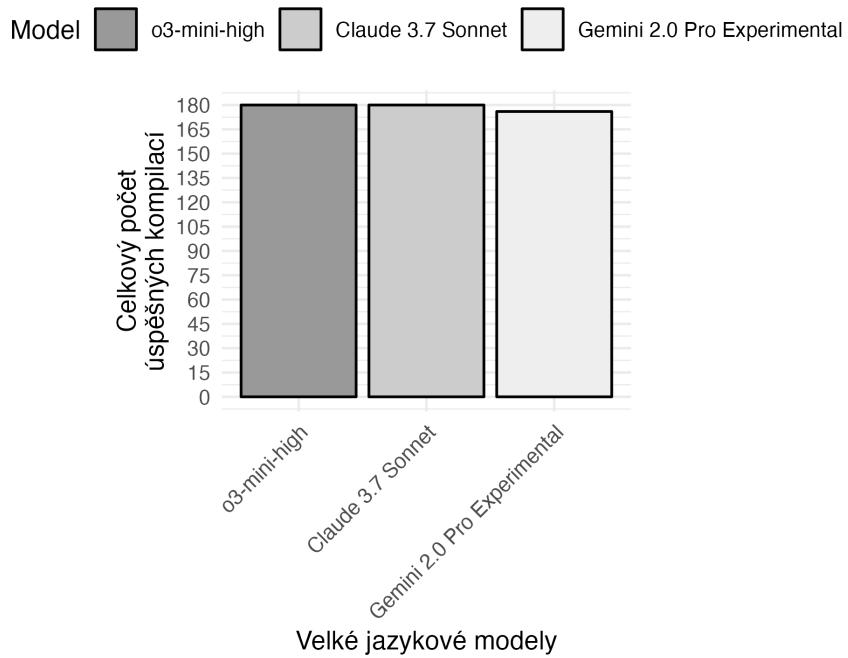
Tabulka 5.3.: Kompilovatelnost – Úkolníček

U poslední úlohy proběhl překlad vygenerovaného kódu úspěšně, bez výskytu jakýchkoli chyb ve všech případech.

Shrnutí

Cílem tohoto kritéria hodnocení bylo posoudit schopnost velkých jazykových modelů generovat syntakticky správný kód v programovacím jazyce Python. Všechny testované modely a výzvy dosáhly obecně vysoké úrovně kompilovatelnosti zdrojového kódu.

Analýza ukázala, že nejvyšší míry kompilovatelnosti bylo dosaženo při použití technik Structured Prompt (Zero-shot), Chain-of-Thought (Zero-shot) a Role Prompt (Zero-shot), kde se podařilo dosáhnout 100 % úspěšnosti ve všech testovaných úlohách. Naopak technika Few-shot vykazovala mírně nižší spolehlivost, kdy v některých případech došlo k selhání komplikace v důsledku syntaktických chyb. Tyto chyby se vyskytly výhradně u modelu Google Gemini 2.0 Pro Experimental. Tento fakt naznačuje jeho vyšší citlivost modelu na rozšířený kontext ve výzvách. Přítomnost demonstračních příkladů mohla vést k narušení struktury generovaného kódu místo očekávaného zlepšení.



Obrázek 5.1.: Kompilovatelnost – Celková úspěšnost

Z vizualizace celkové úspěšnosti modelů je patrné, že nejlepší výsledky vykázaly modely OpenAI o3-mini-high a Anthropic Claude 3.7 Sonnet, jejichž veškeré výstupy byly vždy kompilovatelné. Jediným modelem, u něhož se objevily chyby, byl Google Gemini 2.0 Pro Experimental. Ve všech případech šlo o drobné syntaktické nedostatky, které by bylo možné snadno opravit. Z celkového pohledu si však model Google Gemini 2.0 Pro Experimental navzdory dílčím selhání zachoval velmi vysokou míru kompilovatelnosti, přesahující 97 %.

5.2. Funkční úplnost

Test funkční úplnosti ověřuje, zda výstupy generované velkými jazykovými modely obsahují všechny požadované prvky rozhraní specifikované ve výzvě. Mezi tyto prvky patří zejména deklarace tříd, metod a funkcí, správný počet parametrů a odpovídající typové anotace. Za funkčně úplný je považován takový kód, který beze zbytku naplňuje všechny stanovené požadavky. Naopak výstup, který je funkční a syntakticky správný, avšak postrádá některý z předepsaných prvků, je hodnocen jako neúplný.

Výsledky jsou prezentovány jak v procentuální úspěšnosti, tak v absolutních počtech. Vyšší hodnota v obou případech značí lepší výsledek. Celkový počet hodnocených výstupů se mezi jednotlivými variantami může mírně lišit, neboť některé kódy byly předem vyřazeny kvůli nekomplikovanosti. Maximálně bylo hodnoceno deset výstupů pro každou variantu modelu a typu výzvy.

Kalkulačka

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Structured Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Chain-of-Thought (Zero-shot)	100% (10/10)	90% (9/10)	100% (10/10)
Chain-of-Thought (Few-shot)	100% (10/10)	100% (10/10)	100% (8/8)
Role Prompt (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Role Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (9/9)

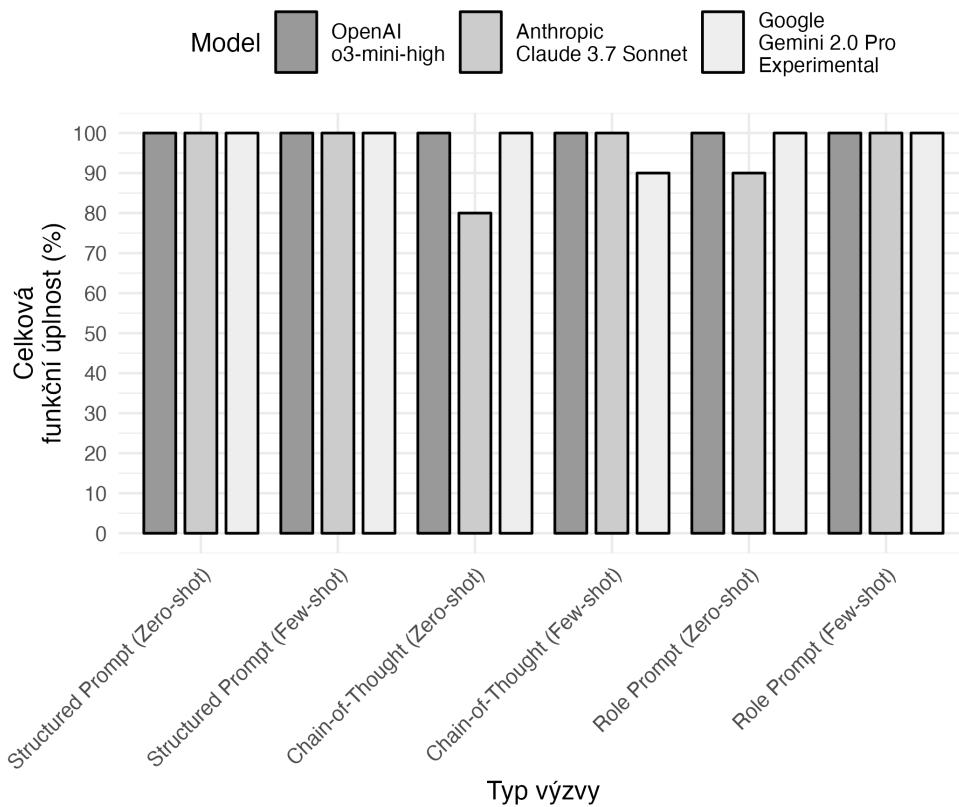
Tabulka 5.4.: Funkční úplnost – Kalkulačka

V rámci úlohy kalkulačky dosáhly všechny tři testované modely velmi vysoké úrovně funkční úplnosti. Modely OpenAI o3-mini-high a Google Gemini 2.0 Pro Experimental vykazovaly ve všech variantách výzev zcela kompletní implementaci požadovaného rozhraní v souladu se stanovenými požadavky. Model Anthropic Claude 3.7 Sonnet selhal pouze v jediném případě při použití techniky Chain-of-Thought (Zero-shot), kdy u jedné z funkcí chyběla typová anotace. Tento nedostatek by však neměl zásadně ovlivnit samotnou funkčnost generovaného kódu.

Konzolové vykreslování

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Structured Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (9/9)
Chain-of-Thought (Zero-shot)	100% (10/10)	80% (8/10)	100% (10/10)
Chain-of-Thought (Few-shot)	100% (10/10)	100% (10/10)	90% (9/10)
Role Prompt (Zero-shot)	100% (10/10)	90% (9/10)	100% (10/10)
Role Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (10/10)

Tabulka 5.5.: Funkční úplnost – Konzolové vykreslování



Obrázek 5.2.: Funkční úplnost – Konzolové vykreslování

U úlohy konzolového vykreslování byl oproti předchozí úloze zaznamenán mírný pokles úrovně funkční úplnosti. Nejlepších výsledků dosáhl model OpenAI o3-mini-high, který jako jediný dosáhl 100 % úspěšnosti ve všech variantách výzev. Ostatní model Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental zaznamenaly alespoň jeden případ funkční neúplnosti.

Model Anthropic Claude 3.7 Sonnet nejčastěji selhával při použití techniky Chain-of-Thought (Zero-shot), konkrétně ve dvou případech. Třetí případ neúplné implementace byl u tohoto modelu zaznamenán při použití Role Prompt (Zero-shot). Ve všech případech spočíval problém v opomenutí typových anotací parametrů, což sice nesnižuje funkčnost jako takovou, ale negativně ovlivňuje celkovou kvalitu výstupu.

Google Gemini 2.0 Pro Experimental zaznamenal chybu v jednom případě při využití techniky Chain-of-Thought (Few-shot), kdy došlo k vynechání povinného parametru ve funkci. Tento nedostatek je z hlediska funkční úplnosti závažnější než absence typové anotace.

Úkolníček

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Structured Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Chain-of-Thought (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Chain-of-Thought (Few-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Role Prompt (Zero-shot)	100% (10/10)	100% (10/10)	100% (10/10)
Role Prompt (Few-shot)	100% (10/10)	100% (10/10)	100% (10/10)

Tabulka 5.6.: Funkční úplnost – Úkolníček

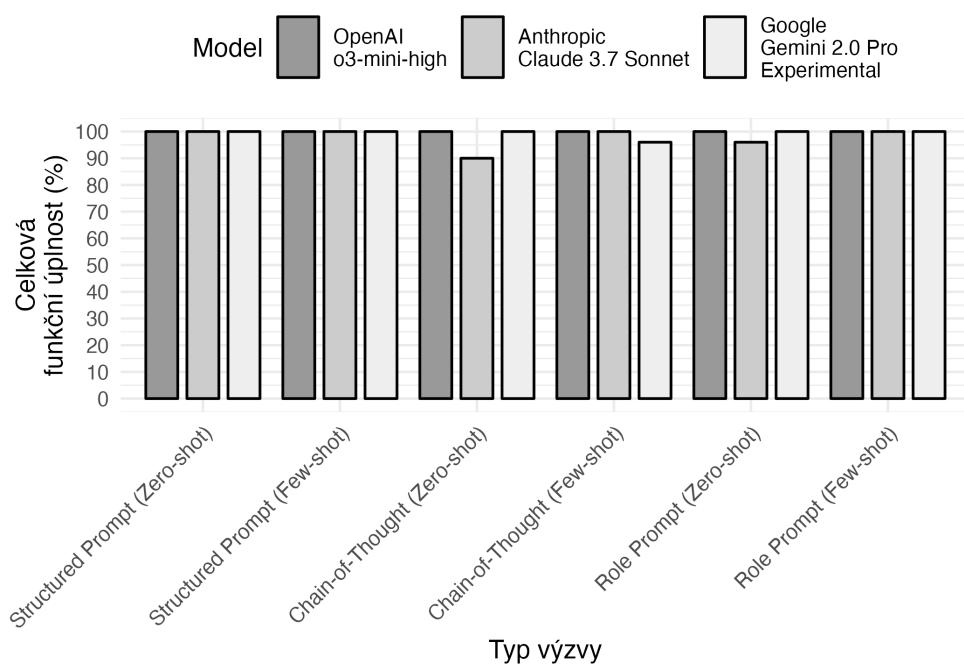
V úloze úkolníčku bylo dosaženo zcela konzistentních výsledků napříč všemi modely i variantami výzev. Všech 180 hodnocených výstupů splnilo kritéria funkční úplnosti bez jediné chyby.

Shrnutí

Testování funkční úplnosti prokázalo vysokou míru souladu generovaného kódu s požadavky stanovenými ve výzvách u všech testovaných modelů. Nejvyšší úspěšnosti dosáhl model OpenAI o3-mini-high, který nevykázal žádné selhání, zatímco ostatní modely vykazovaly drobné nedostatky ve funkční úplnosti.

Jedinou technikou, která vedla k úplné implementaci požadované funkcionality ve všech případech, byla Structured Prompt. Ostatní typy výzev vykazovaly chyby ať už ve variantě Zero-shot nebo Few-shot.

Zaznamenané nedostatky pravděpodobně více odrážejí rozdíly v návrhu a chování jednotlivých modelů než vliv použitých technik inženýrství výzev. U modelu Anthropic Claude 3.7 Sonnet se chyby častěji vyskytovaly při využití techniky Chain-of-Thought (Zero-shot), což může naznačovat, že tento typ výzev podporuje vyšší míru kreativity na úkor důsledného dodržování formálních náležitostí, jako jsou typové anotace. Ostatní chyby byly spíše výjimečné a nevykazovaly systematický vzorec napříč jednotlivými úlohami a použitými technikami.



Obrázek 5.3.: Funkční úplnost – Celkový přehled

Z hlediska celkové funkční úplnosti si nejlépe vedl model OpenAI o3-mini-high, který dosáhl nejvyšší úspěšnosti ve všech úlohách i variantách výzev. Model Google Gemini 2.0 Pro Experimental selhal pouze v jediném případě, přičemž chyba byla závažného charakteru. Naopak model Anthropic Claude 3.7 Sonnet vykázal více případů menších, avšak opakujících se nedostatků, typicky souvisejících s absencí typových anotací parametrů.

5.3. Funkční správnost

Cílem testování funkční správnosti je ověřit, zda generovaný kód splňuje požadavky definované ve výzvě. Hodnocení bylo provedeno na základě předem připravených testovacích scénářů, které pokrývají různé aspekty požadované funkcionality. Scénář je považován za úspěšný pouze v případě, že byly splněny všechny jeho případy. Neúspěch v jediném případě znamená označení celého scénáře za neúspěšný.

Tabulky zobrazují medián počtu úspěšně zvládnutých scénářů z celkového množství testovaných scénářů. Mediánové hodnoty vycházejí ze všech přeložitelných kódů vygenerovaných pomocí stejného typu výzvy, kterých bylo nejvíce deset. Vyšší hodnota metriky indikuje lepší výsledek.

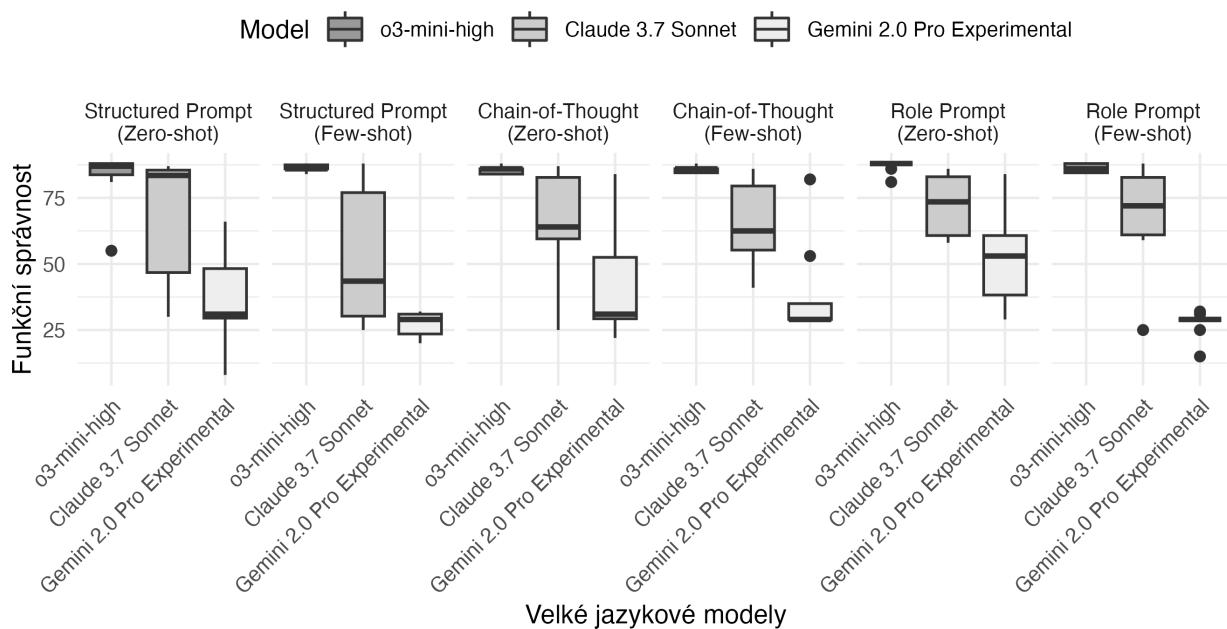
Kalkulačka

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	87/88	83/88	31/88
Structured Prompt (Few-shot)	86/88	43/88	29/88
Chain-of-Thought (Zero-shot)	86/88	64/88	31/88
Chain-of-Thought (Few-shot)	86/88	62/88	29/88
Role Prompt (Zero-shot)	88/88	73/88	53/88
Role Prompt (Few-shot)	86/88	72/88	29/88

Tabulka 5.7.: Funkční správnost – Kalkulačka

Na základě dosažených mediánových hodnot funkční správnosti u úlohy kalkulačky lze pozorovat, že nejlepších výsledků dosáhl model OpenAI o3-mini-high. Tento model vykazoval nejvyšší úroveň funkční správnosti napříč všemi variantami výzev. Také model Anthropic Claude 3.7 Sonnet získal ve většině případů vysoké hodnocení, avšak jeho úspěšnost výrazně poklesla u výzvy typu Structured Prompt (Few-shot). Naopak model Google Gemini 2.0 Pro Experimental zaznamenal mediánově nejnižší výsledky ve všech testovaných variantách výzev.

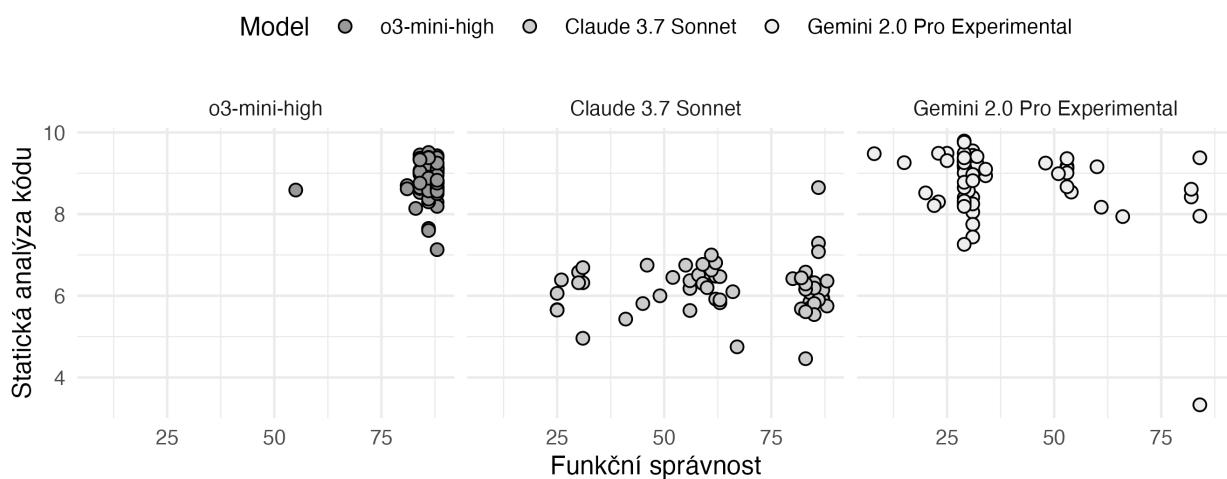
Zajímavým zjištěním je, že žádná z variant využívajících Few-shot přístup nepřekonala mediánové výsledky dosažené pomocí Zero-shot výzev. Tento výsledek naznačuje, že zařazení demonstračních příkladů nemusí vždy vést ke zvýšení výkonu modelu, což je v rozporu s běžně očekávaným přínosem Few-shot přístupu.



Obrázek 5.4.: Funkční správnost – Kalkulačka

Při detailnější analýze výsledků lze pozorovat, že model OpenAI o3-mini-high řešil testovací scénáře s nejvyšší konzistencí. Opakovaně dosahoval nejvyššího hodnocení napříč různými variantami výzev a současně vykazoval nejnižší rozpětí naměřených hodnot. Naproti tomu modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental vykazovaly vyšší variabilitu výsledků, kdy v některých případech generovaly téměř bezchybný výstup, zatímco v jiných selhávaly již u základních požadavků.

Z hlediska efektivity typů výzev se jako nejúčinnější ukázala varianta Role Prompt (Zero-shot), která poskytla optimální rovnováhu mezi vysokou úspěšností a stabilitou výstupů napříč modely.



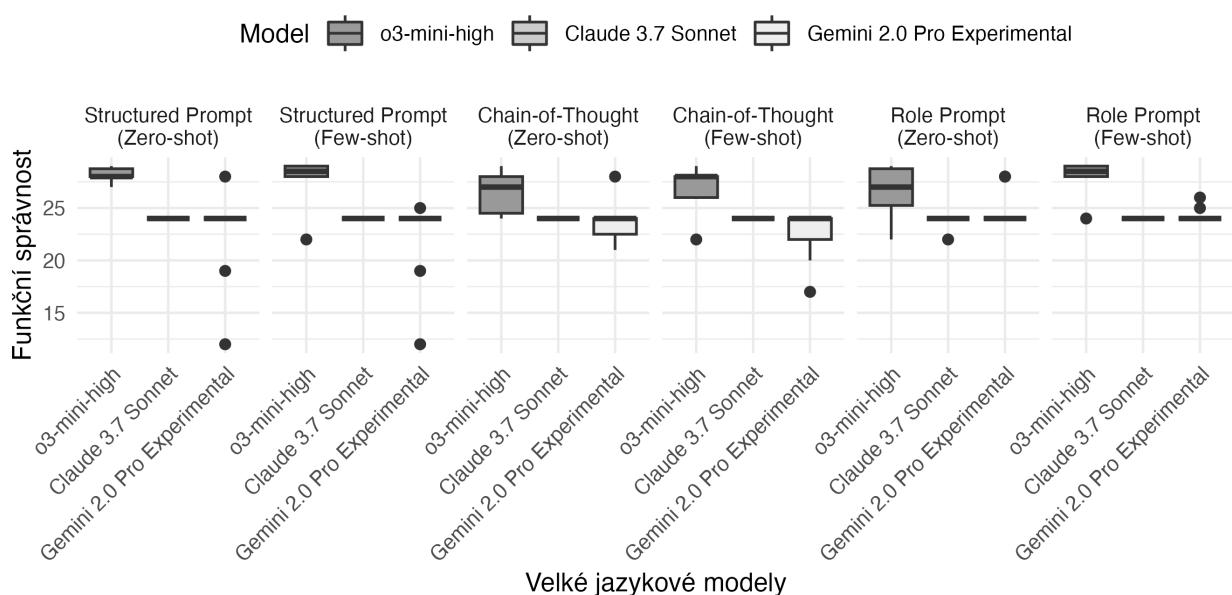
Obrázek 5.5.: Funkční správnost a statická analýza kódu – Kalkulačka

Porovnání výsledků funkční správnosti a skóre statické analýzy pomocí bodového grafu ukazuje, že si model OpenAI o3-mini-high udržel vysoké skóre v obou oblastech. Zatímco ostatní modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental vykazovaly nesoulad mezi těmito dvěma metrikami. Tyto modely měly tendenci udržovat typickou úroveň skóre statické analýzy bez ohledu na to, jak dobře zvládly splnit funkční požadavky.

Konzolové vykreslování

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	28/35	24/35	24/35
Structured Prompt (Few-shot)	28/35	24/35	24/35
Chain-of-Thought (Zero-shot)	27/35	24/35	24/35
Chain-of-Thought (Few-shot)	28/35	24/35	24/35
Role Prompt (Zero-shot)	27/35	24/35	24/35
Role Prompt (Few-shot)	28/35	24/35	24/35

Tabulka 5.8.: Funkční správnost – Konzolové vykreslování



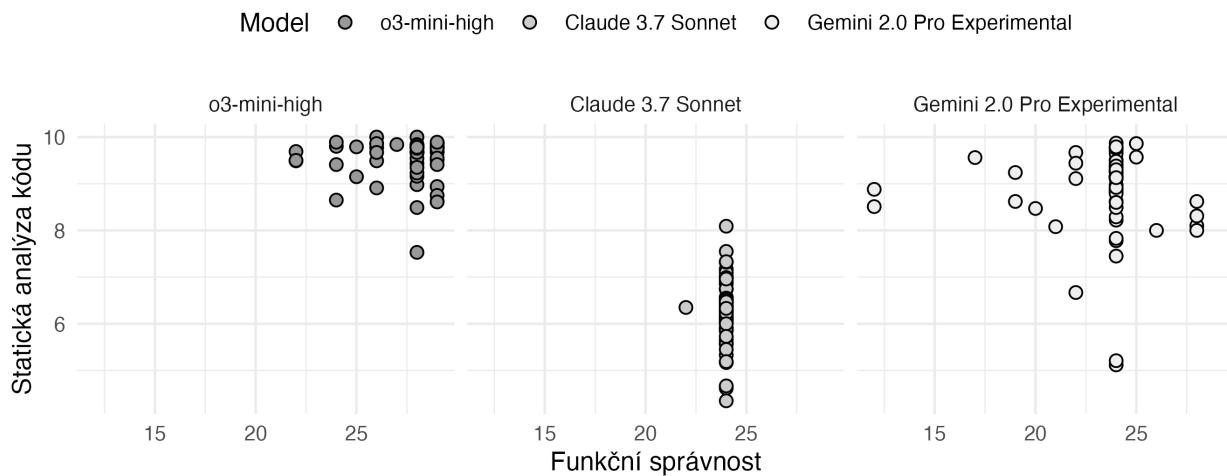
Obrázek 5.6.: Funkční správnost – Konzolové vykreslování

Podrobná analýza jednotlivých pozorování ukázala zvýšené mezikvartilové rozpětí výsledků modelu OpenAI o3-mini-high při řešení zadané úlohy. Po vyloučení odlehlých hodnot si tento model udržel jeden z nejširších rozsahů dosažených hodnot napříč většinou sledovaných typů výzev. Přestože jeho výstupy vykazovaly vyšší variabilitu, ve většině případů překonávaly výsledky ostatních modelů. OpenAI o3-mini-high tak dosahoval nejvyšší úspěšnosti v testovacích scénářích zaměřených na ověření funkční správnosti a současně vykázal nejvyšší mediánovou hodnotu funkční správnosti ze všech zkoumaných modelů a variant výzev.

Nejnižší mezikvartilové rozpětí bylo naopak zaznamenáno u modelu Anthropic Claude 3.7 Sonnet, jehož výstupy byly mimořádně konzistentní a téměř bez odlehlých hodnot. Výsledky však ukazují, že i přes stabilní úroveň funkční správnosti dosáhl nižší mediánové hodnoty než model OpenAI o3-mini-high.

Srovnatelnou úspěšnost vykazoval také model Google Gemini 2.0 Pro Experimental, jehož výstupy se vyznačovaly vyšší variabilitou. Ta vedla k širšímu rozpětí výsledků, kdy v některých případech model překonal Anthropic Claude 3.7 Sonnet, nicméně v jiných dosahoval nižší funkční správnosti. Mediánová úspěšnost modelu Google Gemini 2.0 Pro Experimental přitom ve všech analyzovaných typech výzev zůstávala na stejné úrovni jako u modelu Anthropic Claude 3.7 Sonnet.

Z hlediska efektivity jednotlivých typů výzev dosahovaly všechny varianty srovnatelného mediánu funkční správnosti. Rozdíly se však projevily v hodnotách průměru a směrodatné odchylky, kdy nejlepších výsledků dosáhly výzvy typu Role Prompt (Zero-shot) a Role Prompt (Few-shot). Tyto výzvy byly charakteristické vysokou průměrnou úspěšností a nízkou variabilitou bez ohledu na použitý velký jazykový model.



Obrázek 5.7.: Funkční správnost a statická analýza kódu – Konzolové vykreslování

Při porovnání funkční správnosti a statické analýzy kódu se ukazuje, že model OpenAI o3-mini-high i zde vykazoval vysokou míru konzistence mezi oběma metrikami. Naproti tomu modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental dosahovaly nižší shody. U modelu Anthropic Claude 3.7 Sonnet je patrná tendence generovat výstupy s konzistentním

počtem úspěšně splněných testů, avšak s výrazným rozpětím ve výsledcích statické analýzy. Model Gemini 2.0 Pro Experimental navíc vykazoval zvýšenou nestabilitu výstupů v rámci funkční správnosti.

Úkolníček

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	32/47	31/47	32/47
Structured Prompt (Few-shot)	32/47	32/47	26/47
Chain-of-Thought (Zero-shot)	34/47	30/47	32/47
Chain-of-Thought (Few-shot)	35/47	34/47	27/47
Role Prompt (Zero-shot)	32/47	33/47	25/47
Role Prompt (Few-shot)	33/47	33/47	26/47

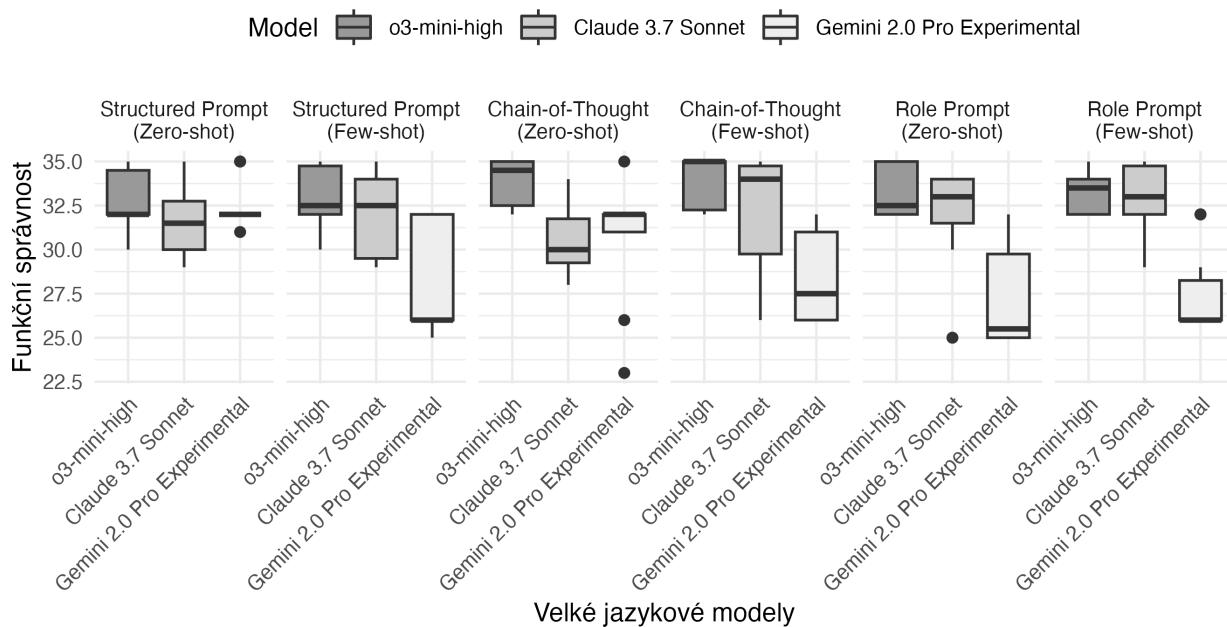
Tabulka 5.9.: Funkční správnost – Úkolníček

Výsledky analýzy úlohy úkolníčku ukazují, že model OpenAI o3-mini-high dosáhl nejvyšší mediánové úspěšnosti ve funkční správnosti. Ve srovnání s předchozími úlohami byl však rozdíl oproti modelu Anthropic Claude 3.7 Sonnet méně výrazný.

Model Anthropic Claude 3.7 Sonnet dokonce mediánově překonal model OpenAI o3-mini-high. Tato situace nastala v případě výzvy typu Role Prompt (Zero-shot), ale i v ostatních případech tento model dosahoval srovnatelných výsledků. Jedná se o jediný případ mezi hodnocenými úlohami, kdy došlo ke změně pořadí těchto dvou modelů v mediánové úspěšnosti funkční správnosti.

Tato změna je pozoruhodná, protože v předchozích úlohách vykazoval model OpenAI o3-mini-high výrazně vyšší mediánovou úspěšnost než model Anthropic Claude 3.7 Sonnet. Tato změna může být důsledkem specifických charakteristik úlohy úkolníčku nebo rozdílné schopnosti modelů při řešení různých typů úloh.

Model Google Gemini 2.0 Pro Experimental si udržel nejnižší úspěšnost napříč všemi technikami. Nejvýrazněji se jeho snížená úspěšnost projevovala při použití strategie Few-shot, kde model mediánově dosahoval podprůměrných výsledků.

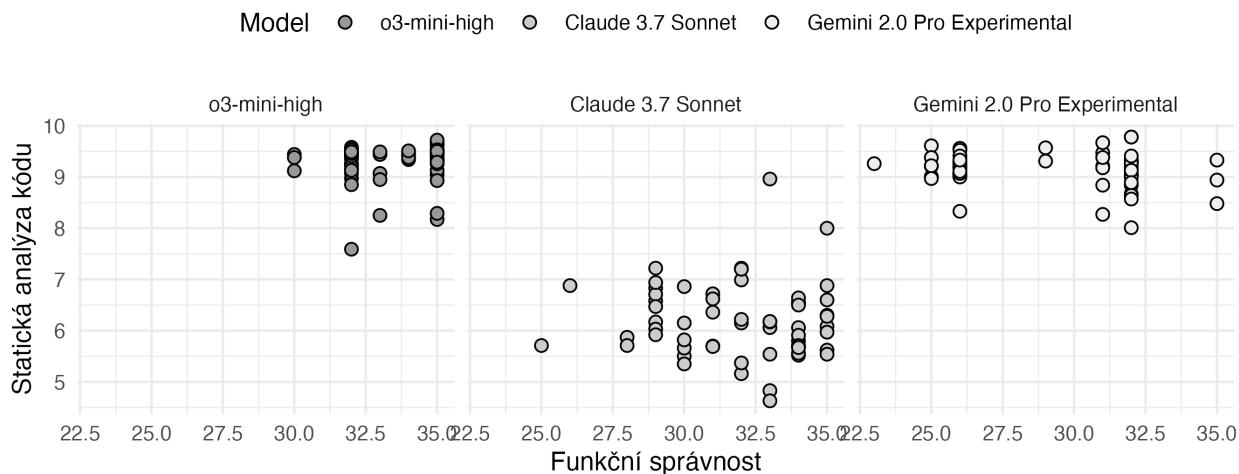


Obrázek 5.8.: Funkční správnost – Úkolníček

Detailnější analýza naznačuje, že model OpenAI o3-mini-high nejenže dosahoval vysoké úspěšnosti, ale zároveň vykazoval nejnižší variační rozpětí naměřených výsledků. Jeho úspěšnost byla téměř nezávislá na použité technice výzvy, což svědčí o vysoké stabilitě napříč různými přístupy.

Naopak model Anthropic Claude 3.7 Sonnet vykazoval vyšší citlivost na změnu typu výzvy. Zatímco v některých případech generoval srovnatelné výsledky s modelem OpenAI o3-mini-high, v jiných došlo k poklesu úspěšnosti.

Podrobnější analýza vlivu jednotlivých technik výzvy ukázala shodnou mediánovou úspěšnost napříč variantami. Rozdíly se projevily až při hodnocení průměrných hodnot a směrodatných odchylek, přičemž nejvyšší úspěšnosti bylo dosaženo pomocí strategie Structured Prompt (Zero-shot).



Obrázek 5.9.: Funkční správnost a statická analýza kódu – Úkolníček

Analýza bodového grafu, který porovnává funkční správnost se skóre statické analýzy kódu, odhaluje výraznou shodu těchto metrik u modelu OpenAI o3-mini-high. Tento model typicky generoval výstupy, které jsou nejen funkčně správné, ale i kvalitní z hlediska statické analýzy.

Modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental naopak vykazovaly nižší míru konzistence mezi oběma sledovanými kritérii. Oba modely generovaly kód s výrazným rozpětím hodnot funkční správnosti i skóre statické analýzy. Přesto si Google Gemini 2.0 Pro Experimental udržoval vyšší typické skóre statické analýzy než Anthropic Claude 3.7 Sonnet, což poukazuje na vyšší kvalitu kódu generovaného tímto modelem.

Shrnutí

Testování funkční správnosti generovaného kódu odhalilo výrazné rozdíly mezi jednotlivými velkými jazykovými modely i použitými technikami inženýrství výzv. Model OpenAI o3-mini-high dosahoval nejvyšší a nejstabilnější úspěšnosti, vykazoval konzistentně spolehlivé výsledky napříč všemi úlohami i typy výzv. Naopak model Google Gemini 2.0 Pro Experimental dosahoval trvale nejnižší úspěšnost a žádná ze zvolených technik jeho výkon významně nezlepšila.

Model OpenAI o3-mini-high vynikal zejména schopností přesné interpretace zadání a jako jediný dosáhl 100 % mediánové úspěšnosti alespoň v jedné kombinaci úlohy a výzvy, což podtrhuje jeho nadprůměrné schopnosti. Anthropic Claude 3.7 Sonnet byl v některých případech srovnatelný, avšak jeho výsledky vykazovaly vyšší variabilitu, což naznačuje nižší robustnost a potenciální nutnost dodatečného testování či ladění v praktickém nasazení.

Výsledky rovněž ukázaly, že neexistuje univerzální vliv konkrétní techniky výzvy na funkční správnost napříč všemi modely a úlohami. Například strategie Role Prompt byla nejfektivnější u prvních dvou úloh, avšak u poslední úlohy vykazovala vyšší směrodatnou i mediánovou absolutní odchylku oproti jiným technikám. Ukázalo se také, že optimální volba techniky výzvy závisí jak na specifikách modelu, tak na povaze řešeného úkolu. Zatímco u některých modelů byla technika Few-shot přínosná, u jiných tento efekt pozorován nebyl.

5.4. Časová náročnost

Testování časové náročnosti se zaměřuje na vyhodnocení výpočetního času potřebného k provedení jednotlivých operací ve vygenerovaném kódu. Pro každou analyzovanou úlohu jsou identifikovány specifické funkce, na kterých proběhlo systematické měření. Naměřené časy představují průměrné hodnoty získané z opakovaného provádění daných operací. Tento průběh měření byl zvolen s cílem minimalizovat vliv náhodných odchylek.

Prezentované tabulky uvádějí mediány časové náročnosti pro jednotlivé funkce. Mediánová hodnota je založena na všech úspěšně komplikovaných kódech vzniklých ze stejné výzvy. Nižší hodnota značí kratší dobu běhu a tím vyšší efektivitu generovaného kódu z hlediska výpočetního výkonu.

Kalkulačka

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	add 2.54×10^{-6} s	add 2.62×10^{-6} s	add 2.93×10^{-6} s
	subtract 2.80×10^{-6} s	subtract 2.60×10^{-6} s	subtract 2.89×10^{-6} s
	multiply 2.45×10^{-6} s	multiply 2.57×10^{-6} s	multiply 3.11×10^{-6} s
	divide 2.53×10^{-6} s	divide 2.68×10^{-6} s	divide 3.11×10^{-6} s
	composi. 6.72×10^{-6} s	composi. 5.56×10^{-6} s	composi. 7.02×10^{-6} s
Structured Prompt (Few-shot)	add 3.25×10^{-6} s	add 3.56×10^{-6} s	add 3.25×10^{-6} s
	subtract 3.25×10^{-6} s	subtract 3.92×10^{-6} s	subtract 3.25×10^{-6} s
	multiply 3.22×10^{-6} s	multiply 3.53×10^{-6} s	multiply 3.29×10^{-6} s
	divide 3.25×10^{-6} s	divide 3.56×10^{-6} s	divide 3.35×10^{-6} s
	composi. 7.23×10^{-6} s	composi. 8.17×10^{-6} s	composi. 7.55×10^{-6} s
Chain-of-Thought (Zero-shot)	add 2.04×10^{-6} s	add 3.00×10^{-6} s	add 2.79×10^{-6} s
	subtract 2.09×10^{-6} s	subtract 3.07×10^{-6} s	subtract 2.69×10^{-6} s
	multiply 2.09×10^{-6} s	multiply 3.06×10^{-6} s	multiply 2.82×10^{-6} s
	divide 2.13×10^{-6} s	divide 3.01×10^{-6} s	divide 2.85×10^{-6} s
	composi. 5.28×10^{-6} s	composi. 7.80×10^{-6} s	composi. 6.92×10^{-6} s
Chain-of-Thought (Few-shot)	add 3.12×10^{-6} s	add 3.51×10^{-6} s	add 2.93×10^{-6} s
	subtract 3.28×10^{-6} s	subtract 3.49×10^{-6} s	subtract 2.98×10^{-6} s
	multiply 3.12×10^{-6} s	multiply 3.42×10^{-6} s	multiply 2.96×10^{-6} s
	divide 3.21×10^{-6} s	divide 3.58×10^{-6} s	divide 3.09×10^{-6} s
	composi. 6.93×10^{-6} s	composi. 8.76×10^{-6} s	composi. 6.59×10^{-6} s

Role Prompt (Zero-shot)	add	2.25×10^{-6} s	add	4.00×10^{-6} s	add	3.77×10^{-6} s
	subtract	2.31×10^{-6} s	subtract	4.01×10^{-6} s	subtract	3.70×10^{-6} s
	multiply	2.28×10^{-6} s	multiply	3.69×10^{-6} s	multiply	3.61×10^{-6} s
	divide	2.29×10^{-6} s	divide	3.66×10^{-6} s	divide	3.44×10^{-6} s
	composi.	5.39×10^{-6} s	composi.	8.03×10^{-6} s	composi.	7.49×10^{-6} s
Role Prompt (Few-shot)	add	3.61×10^{-6} s	add	3.43×10^{-6} s	add	3.18×10^{-6} s
	subtract	3.90×10^{-6} s	subtract	3.44×10^{-6} s	subtract	3.20×10^{-6} s
	multiply	3.71×10^{-6} s	multiply	3.51×10^{-6} s	multiply	3.16×10^{-6} s
	divide	3.67×10^{-6} s	divide	3.42×10^{-6} s	divide	3.16×10^{-6} s
	composi.	8.60×10^{-6} s	composi.	7.91×10^{-6} s	composi.	7.35×10^{-6} s

Tabulka 5.10.: Časová náročnost – Kalkulačka

Z mediánových hodnot časové náročnosti implementací základních aritmetických operací, jako je sčítání, odčítání, násobení, dělení a složené operace vyplývá, že nejfektivnější kód generoval model OpenAI o3-mini-high. Tento model dosahoval nejnižší časovou náročnost napříč většinou typů výzev, konkrétně u Structured Prompt (Few-shot), Chain-of-Thought (Zero-shot) a Role Prompt (Zero-shot). Výjimkou byla výzva Role Prompt (Few-shot), kde naopak vykazoval nejvyšší mediánovou časovou náročnost ze všech testovaných modelů.

Model Anthropic Claude 3.7 Sonnet naopak vykazoval konzistentně nejvyšší mediánovou časovou náročnost ve většině typů výzev. Nejvyšší hodnoty byly zaznamenány zejména u Structured Prompt (Few-shot), Chain-of-Thought (Zero-shot), Chain-of-Thought (Few-shot) a Role Prompt (Zero-shot). Jediným typem výzvy, ve které model dosáhl nejlepšího mediánového času při implementaci operace odčítání a složené operace, byla výzva typu Structured Prompt (Zero-shot).

Model Google Gemini 2.0 Pro Experimental se mediánově pohyboval mezi předchozími dvěma modely. Nejlepších výsledků dosahoval ve výzvách Chain-of-Thought (Few-shot) a Role Prompt (Few-shot), kde generoval nejfektivnější kód napříč všemi operacemi. Naopak nejméně efektivní byl u Structured Prompt (Zero-shot).

Z pohledu typů výzev byla nejfektivnější Chain-of-Thought (Zero-shot), v jejímž rámci modely OpenAI o3-mini-high a Google Gemini 2.0 Pro Experimental dosahovaly nejnižších mediánových časů. Tento typ výzev rovněž vykazoval nejlepší kombinaci průměru, mediánu i směrodatné odchylky bez ohledu na použitý velký jazykový model.

Z mediánových výsledků vyplývá, že jednoduché aritmetické operace byly obecně časově méně náročné než složené výrazy, což odpovídá očekávanému chování, kdy kombinace více operací bude časově náročnější.

Významným zjištěním je také fakt, že výzvy bez explicitních ukázek kódu (Zero-shot) vedly zpravidla k efektivnějším implementacím, s výjimkou Role Prompt (Few-shot), kde modely Anthropic Claude Sonnet a Google Gemini 2.0 Pro Experimental dosáhly lepších mediánových výsledků než v jejich Zero-shot variantách.

Konzolové vykreslování

	OpenAI o3-mini-high		Anthropic Claude 3.7 Sonnet		Google Gemini 2.0 Pro Experimental	
Structured Prompt (Zero-shot)	square	4.48×10^{-6} s	square	1.57×10^{-6} s	square	4.44×10^{-6} s
	rectangle	2.52×10^{-6} s	rectangle	0.91×10^{-6} s	rectangle	2.37×10^{-6} s
	parallelo.	5.27×10^{-6} s	parallelo.	5.03×10^{-6} s	parallelo.	5.21×10^{-6} s
	triangle	7.67×10^{-6} s	triangle	8.44×10^{-6} s	triangle	3.17×10^{-6} s
	pyramid	6.53×10^{-6} s	pyramid	6.76×10^{-6} s	pyramid	6.71×10^{-6} s
Structured Prompt (Few-shot)	square	4.54×10^{-6} s	square	1.96×10^{-6} s	square	4.54×10^{-6} s
	rectangle	2.57×10^{-6} s	rectangle	1.08×10^{-6} s	rectangle	2.34×10^{-6} s
	parallelo.	5.49×10^{-6} s	parallelo.	5.38×10^{-6} s	parallelo.	5.27×10^{-6} s
	triangle	7.83×10^{-6} s	triangle	6.72×10^{-6} s	triangle	4.04×10^{-6} s
	pyramid	6.77×10^{-6} s	pyramid	6.60×10^{-6} s	pyramid	8.03×10^{-6} s
Chain-of-Thought (Zero-shot)	square	4.48×10^{-6} s	square	1.96×10^{-6} s	square	1.25×10^{-6} s
	rectangle	2.48×10^{-6} s	rectangle	1.09×10^{-6} s	rectangle	0.70×10^{-6} s
	parallelo.	5.45×10^{-6} s	parallelo.	5.16×10^{-6} s	parallelo.	5.16×10^{-6} s
	triangle	7.01×10^{-6} s	triangle	8.92×10^{-6} s	triangle	3.14×10^{-6} s
	pyramid	6.56×10^{-6} s	pyramid	6.60×10^{-6} s	pyramid	6.56×10^{-6} s
Chain-of-Thought (Few-shot)	square	4.49×10^{-6} s	square	1.59×10^{-6} s	square	4.40×10^{-6} s
	rectangle	2.52×10^{-6} s	rectangle	0.92×10^{-6} s	rectangle	2.33×10^{-6} s
	parallelo.	5.53×10^{-6} s	parallelo.	5.22×10^{-6} s	parallelo.	5.13×10^{-6} s
	triangle	5.69×10^{-6} s	triangle	6.64×10^{-6} s	triangle	5.73×10^{-6} s
	pyramid	6.48×10^{-6} s	pyramid	6.57×10^{-6} s	pyramid	6.82×10^{-6} s
Role Prompt (Zero-shot)	square	4.46×10^{-6} s	square	1.61×10^{-6} s	square	4.44×10^{-6} s
	rectangle	2.54×10^{-6} s	rectangle	0.91×10^{-6} s	rectangle	2.35×10^{-6} s
	parallelo.	5.51×10^{-6} s	parallelo.	5.15×10^{-6} s	parallelo.	5.17×10^{-6} s
	triangle	6.58×10^{-6} s	triangle	8.98×10^{-6} s	triangle	5.89×10^{-6} s
	pyramid	6.70×10^{-6} s	pyramid	6.57×10^{-6} s	pyramid	6.50×10^{-6} s
Role Prompt (Few-shot)	square	4.46×10^{-6} s	square	2.04×10^{-6} s	square	4.57×10^{-6} s
	rectangle	2.54×10^{-6} s	rectangle	1.72×10^{-6} s	rectangle	2.48×10^{-6} s
	parallelo.	5.49×10^{-6} s	parallelo.	5.15×10^{-6} s	parallelo.	5.14×10^{-6} s
	triangle	8.65×10^{-6} s	triangle	9.62×10^{-6} s	triangle	6.01×10^{-6} s
	pyramid	6.56×10^{-6} s	pyramid	6.49×10^{-6} s	pyramid	6.48×10^{-6} s

Tabulka 5.11.: Časová náročnost – Konzolové vykreslování

Výsledky časové náročnosti v úloze konzolového vykreslování ukazují odlišné chování modelů ve srovnání s předchozí analyzovanou úlohou kalkulačky. Model OpenAI o3-mini-high zde vykazoval nejnižší časovou efektivitu. Ve většině výzev a typů tvarů generoval implementace s vyšší mediánovou časovou náročností a v žádné výzvě nedosáhl nejfektivnější implementace napříč všemi operacemi současně. Výjimku představovalo vykreslování pyramidy, kde model dosáhl

nejlepších mediánových výsledků ve výzvách Structured Prompt (Zero-shot), Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot), ovšem rozdíly oproti modelům Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental byly minimální.

Nejvyšší časovou efektivitu v této úloze prokázaly modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental. Model Anthropic Claude 3.7 Sonnet exceloval zejména ve výzvách Structured Prompt (Zero-shot) a Role Prompt (Zero-shot) při vykreslování tvarů, jako je čtverec, obdélník a rovnoběžník. Naopak model Google Gemini 2.0 Pro Experimental generoval mediánově nejfektivnější kód při vykreslování všech tvarů v rámci výzvy Chain-of-Thought (Zero-shot).

Nejfektivnější technikou výzev se ukázala být Chain-of-Thought (Zero-shot), vedoucí k nejnižší mediánové časové náročnosti napříč modely i vykreslovanými tvary. Dopad využití ukázk kódu (Few-shot) nebyl v této úloze konzistentní v závislosti na modelu i typu vykreslovaného tvaru. V některých případech došlo k mediánovému zlepšení výkonu pouze u vybraných tvarů, což naznačuje existenci dalšího neznámého faktoru, který ovlivňuje časovou náročnost.

Úkolníček

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	add 7.25×10^{-7} s	add 8.16×10^{-7} s	add 7.07×10^{-7} s
	sea_tit 10.09×10^{-4} s	sea_tit 9.94×10^{-4} s	sea_tit 14.52×10^{-4} s
	sea_des 9.42×10^{-4} s	sea_des 9.49×10^{-4} s	sea_des 14.46×10^{-4} s
	get_all 0.44×10^{-4} s	get_all 14.67×10^{-4} s	get_all 0.43×10^{-4} s
	finish 4.80×10^{-7} s	finish 6.49×10^{-7} s	finish 4.80×10^{-7} s
	remove 4.91×10^{-7} s	remove 5.14×10^{-7} s	remove 4.91×10^{-7} s
Structured Prompt (Few-shot)	add 7.18×10^{-7} s	add 8.13×10^{-7} s	add 7.00×10^{-7} s
	sea_tit 9.87×10^{-4} s	sea_tit 10.04×10^{-4} s	sea_tit 14.09×10^{-4} s
	sea_des 9.23×10^{-4} s	sea_des 9.47×10^{-4} s	sea_des 13.91×10^{-4} s
	get_all 0.45×10^{-4} s	get_all 14.69×10^{-4} s	get_all 0.43×10^{-4} s
	finish 4.84×10^{-7} s	finish 5.13×10^{-7} s	finish 4.77×10^{-7} s
	remove 4.92×10^{-7} s	remove 5.05×10^{-7} s	remove 4.91×10^{-7} s
Chain-of-Thought (Zero-shot)	add 7.24×10^{-7} s	add 7.36×10^{-7} s	add 6.80×10^{-7} s
	sea_tit 9.92×10^{-4} s	sea_tit 9.89×10^{-4} s	sea_tit 14.57×10^{-4} s
	sea_des 9.29×10^{-4} s	sea_des 9.37×10^{-4} s	sea_des 14.41×10^{-4} s
	get_all 2.34×10^{-4} s	get_all 7.02×10^{-4} s	get_all 0.44×10^{-4} s
	finish 4.81×10^{-7} s	finish 5.04×10^{-7} s	finish 4.72×10^{-7} s
	remove 4.93×10^{-7} s	remove 4.97×10^{-7} s	remove 4.94×10^{-7} s

Chain-of-Thought (Few-shot)	add	7.26×10^{-7} s	add	7.57×10^{-7} s	add	6.79×10^{-7} s
	sea_tit	9.93×10^{-4} s	sea_tit	9.91×10^{-4} s	sea_tit	14.54×10^{-4} s
	sea_des	9.23×10^{-4} s	sea_des	9.31×10^{-4} s	sea_des	14.35×10^{-4} s
	get_all	1.39×10^{-4} s	get_all	9.68×10^{-4} s	get_all	6.32×10^{-4} s
	finish	4.76×10^{-7} s	finish	5.77×10^{-7} s	finish	4.78×10^{-7} s
	remove	4.93×10^{-7} s	remove	5.03×10^{-7} s	remove	4.92×10^{-7} s
Role Prompt (Zero-shot)	add	7.26×10^{-7} s	add	7.26×10^{-7} s	add	6.83×10^{-7} s
	sea_tit	10.04×10^{-4} s	sea_tit	9.89×10^{-4} s	sea_tit	14.32×10^{-4} s
	sea_des	9.36×10^{-4} s	sea_des	9.25×10^{-4} s	sea_des	14.08×10^{-4} s
	get_all	3.21×10^{-4} s	get_all	6.41×10^{-4} s	get_all	0.43×10^{-4} s
	finish	4.75×10^{-7} s	finish	4.99×10^{-7} s	finish	4.75×10^{-7} s
	remove	4.93×10^{-7} s	remove	4.97×10^{-7} s	remove	4.93×10^{-7} s
Role Prompt (Few-shot)	add	7.00×10^{-7} s	add	8.96×10^{-7} s	add	6.82×10^{-7} s
	sea_tit	10.14×10^{-4} s	sea_tit	9.94×10^{-4} s	sea_tit	14.39×10^{-4} s
	sea_des	9.25×10^{-4} s	sea_des	9.39×10^{-4} s	sea_des	13.96×10^{-4} s
	get_all	1.42×10^{-4} s	get_all	14.86×10^{-4} s	get_all	0.43×10^{-4} s
	finish	4.83×10^{-7} s	finish	5.16×10^{-7} s	finish	4.75×10^{-7} s
	remove	4.93×10^{-7} s	remove	5.08×10^{-7} s	remove	4.89×10^{-7} s

Tabulka 5.12.: Časová náročnost – Úkolníček

Z analýzy mediánové časové náročnosti jednotlivých operací úkolníčku, konkrétně vytvoření úkolu, vyhledání úkolu podle názvu či popisu, získání všech úkolů, označení úkolu jako dokončeného a jeho smazání lze pozorovat, že nejfektivnější implementace generoval model Google Gemini 2.0 Pro Experimental. Tento model vykazoval nejnižší mediánovou časovou náročnost u většiny operací.

Dobrých výsledků dosáhl také model OpenAI o3-mini-high, který vynikal zejména při použití technik Structured Prompt (Zero-shot) a Chain-of-Thought (Few-shot), kde mediánově generoval implementace s nejnižší časovou náročností. Naproti tomu model Google Gemini 2.0 Pro Experimental dominoval ve zbývajících typech výzev. U tohoto modelu se však opakovaně vyskytoval problém s výrazným zhoršením efektivity při operacích sea_tit (search_title) a sea_des (search_description). Tyto operace byly ve srovnání s ostatními modely výrazně pomalejší. Pravděpodobně v důsledku neefektivní iterace nebo nevhodné metody filtrování datové struktury uchovávající úkoly.

Nejhorskou efektivitu v této úloze prokázal model Anthropic Claude 3.7 Sonnet, který ve většině případů generoval implementace s vyšší časovou náročností. Ačkoli v některých případech dosáhl přijatelných výsledků, jeho celková výkonnost zůstala výrazně pod úrovní ostatních modelů. Výrazné nedostatky se projevily zejména u operace get_all sloužící k vrácení všech úkolů, kde model Anthropic Claude 3.7 Sonnet zvolil neefektivní konstrukci výstupní datové struktury. Pravděpodobně ji vytvářel až při volání operace, namísto její přípravy v průběhu zpracování.

Z hlediska typů výzev se jako nejfektivnější ukázaly techniky Role Prompt (Zero-shot) a Role Prompt (Few-shot), které vedly k nejnižším průměrným i mediánovým časům napříč modely, a to při zachování nízké variability výsledků. Z mediánových hodnot je navíc patrné, že použití ukázek kódu (Few-shot) mělo pozitivní vliv na časovou efektivitu u modelů OpenAI o3-mini-high a Google Gemini 2.0 Pro Experimental. Přidání ukázek vedlo u těchto modelů ve většině případů ke snížení mediánové časové náročnosti oproti výzvám bez ukázek (Zero-shot), případně ke generování kódu se srovnatelnou efektivitou.

Shrnutí

Analýza časové náročnosti implementací generovaných velkými jazykovými modely odhalila rozdíly v efektivitě vygenerovaného kódu. Tyto rozdíly se projevily jak mezi jednotlivými modely, tak v závislosti na použitých technikách inženýrství výzev a charakteru konkrétních testovaných úloh. Žádný z hodnocených modelů ani žádná z testovaných technik se neukázaly být konzistentně nejfektivnější napříč všemi scénáři.

Z hlediska inženýrství výzev se ukázalo, že různé přístupy mohou ovlivnit časovou efektivitu výsledného kódu. Přesto se žádná z testovaných technik, včetně metod Chain-of-Thought a Role Prompt, neprojevila jako univerzálně nejvýhodnější. V některých případech tyto přístupy vedly k mírně efektivnějším implementacím, zatímco v jiných byla dosažená efektivita obdobná nebo dokonce nižší než u jednodušších typů výzev.

Použití ukázek kódu (Few-shot) nemělo konzistentně pozitivní vliv, a výraznější zlepšení bylo pozorováno pouze u úlohy úkolníčku, kde modely OpenAI o3-mini-high a Google Gemini 2.0 Pro Experimental dosahovaly nižší časové náročnosti oproti Zero-shot variantám.

Také mezi jednotlivými modely byly zjištěny rozdíly v efektivitě, avšak žádný model nevykazoval systematicky vyšší výkonnost napříč všemi úlohami. Nejvýraznější rozdíly se objevily u implementací konkrétních operací, kde jednotlivé modely volily odlišné přístupy. Příkladem je operace `get_all` v úloze úkolníčku, kde model Anthropic Claude 3.7 Sonnet zvolil nevhodný způsob implementace, což vedlo k podstatně horším výsledkům ve srovnání s ostatními modely.

Vzhledem k tomu, že analyzované operace byly převážně triviálního charakteru, nejsou zjištěné rozdíly natolik zásadní, aby měly významný dopad na praktické využití těchto programů. Přesto výsledky naznačují, že volba vhodného modelu a techniky inženýrství výzev může v některých případech přinést nezanedbatelné časové úspory a tím zvýšit efektivitu generovaných řešení.

5.5. Statická analýza kódu

Statická analýza kódu posuzuje výstupy generované jednotlivými modely z hlediska souladu se stylistickými a strukturálními konvencemi programovacího jazyka Python. Výsledné skóre se pohybuje v rozmezí od nuly do deseti, přičemž hodnota deset označuje kód postrádající varování či nedostatky. Nižší skóre naopak indikuje přítomnost stylistických nebo strukturálních problémů.

Prezentované výsledky vycházejí z mediánových hodnot dosaženého skóre pro jednotlivé techniky výzev a vybrané velké jazykové modely. Mediánové hodnoty byly získány z maximálně deseti úspěšně komplikovaných kódů vygenerovaných jedním typem výzvy. Vyšší hodnota skóre odpovídá vyšší kvalitě vygenerovaného kódu.

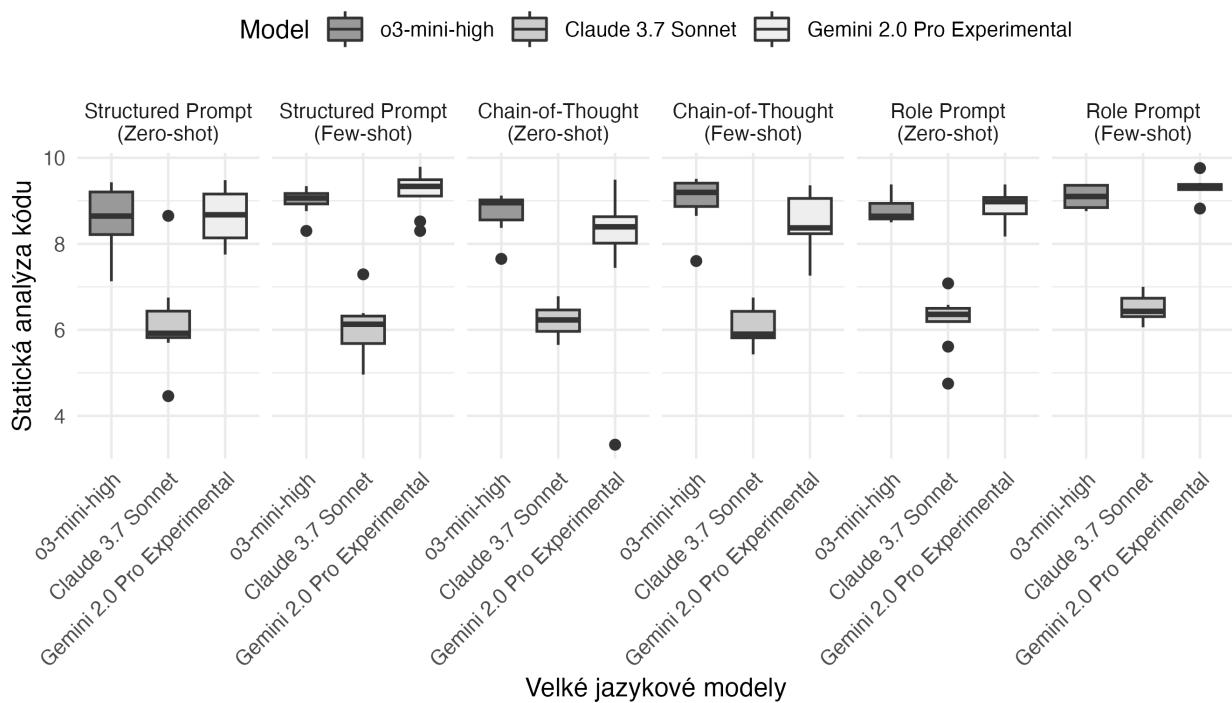
Kalkulačka

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	8.64/10	5.92/10	8.67/10
Structured Prompt (Few-shot)	9.06/10	6.13/10	9.33/10
Chain-of-Thought (Zero-shot)	8.95/10	6.23/10	8.39/10
Chain-of-Thought (Few-shot)	9.19/10	5.90/10	8.37/10
Role Prompt (Zero-shot)	8.64/10	6.36/10	8.98/10
Role Prompt (Few-shot)	9.10/10	6.43/10	9.31/10

Tabulka 5.13.: Statická analýza kódu – Kalkulačka

Výsledky statické analýzy kódu pro úlohu kalkulačky ukazují, že nejvyšší mediánové skóre dosáhly modely OpenAI o3-mini-high a Google Gemini 2.0 Pro Experimental, které byly nejúspěšnější v závislosti na typu použité techniky výzvy. Naproti tomu model Anthropic Claude 3.7 Sonnet vykazoval ve všech testovaných přístupech konzistentně nejnižší úroveň kvality, kdy jeho mediánové skóre bylo v průměru o tři body nižší než u ostatních modelů.

Model OpenAI o3-mini-high dosahoval nejlepších výsledků zejména při použití technik Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot). Podobně Google Gemini 2.0 Pro Experimental exceloval u výzev typu Structured Prompt (Few-shot), Role Prompt (Zero-shot) a Role Prompt (Few-shot).



Obrázek 5.10.: Statická analýza kódu – Kalkulačka

Při celkovém srovnání jednotlivých technik se jako nejúspěšnější ukázala metoda Role Prompt (Few-shot), která nejenže vykázala vysoké mediánové skóre, ale také nejnižší směrodatnou odchylku, což potvrzuje vysokou konzistenci výsledků. Přesto je vhodné upozornit, že technika Structured Prompt (Few-shot) dosáhla ještě vyššího mediánu, avšak v ostatních sledovaných metrikách zaostávala.

Z vizualizace je patrné, že přidání reprezentativních ukázků do výzvy (Few-shot) ve většině případů přispívá k mírnému zlepšení kvality kódu a ke snížení rozpětí hodnot získaných statickou analýzou. Tento trend je podpořen i skutečností, že ve výstupech u Few-shot přístupů se nevyskytují extrémně odlehlé hodnoty.

Konzolové vykreslování

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	9.67/10	6.49/10	9.04/10
Structured Prompt (Few-shot)	9.69/10	6.22/10	9.12/10
Chain-of-Thought (Zero-shot)	9.82/10	6.53/10	8.88/10

Chain-of-Thought (Few-shot)	9.73/10	6.45/10	9.18/10
Role Prompt (Zero-shot)	9.67/10	6.14/10	8.84/10
Role Prompt (Few-shot)	9.65/10	5.64/10	9.02/10

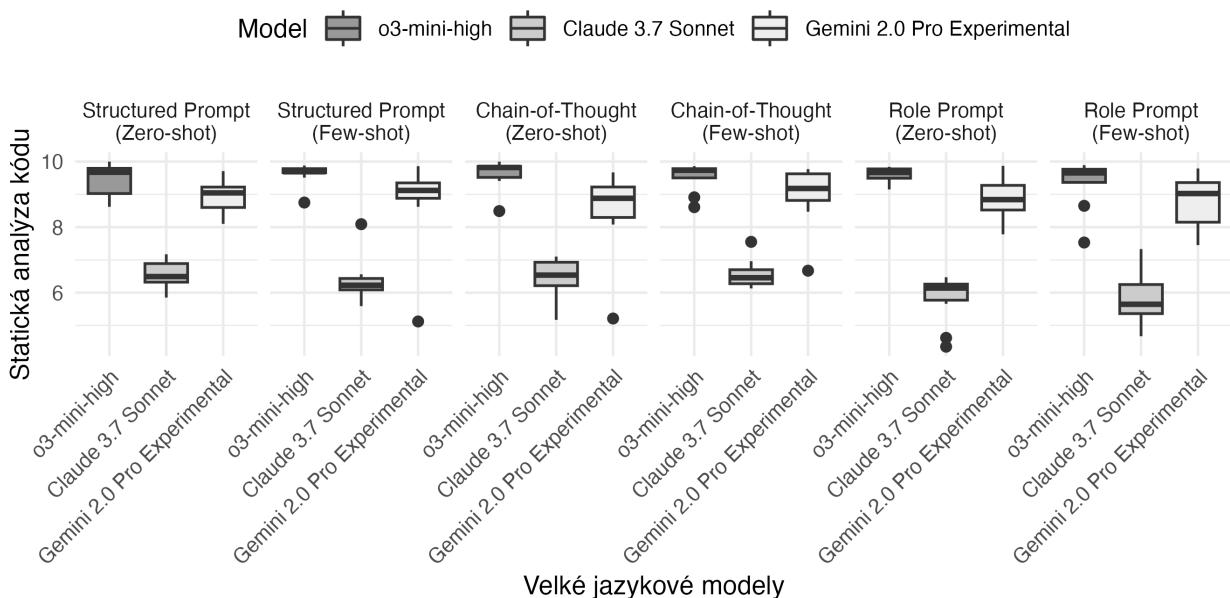
Tabulka 5.14.: Statická analýza kódu – Konzolové vykreslování

Výsledky statické analýzy kódu pro úlohu konzolového vykreslování vykazují podobné trendy jako v případě kalkulačky. Nejvyšší mediánové skóre napříč všemi technikami výzev dosáhl model OpenAI o3-mini-high.

Model Google Gemini 2.0 Pro Experimental se rovněž vyznačoval velmi vysokým skóre statické analýzy, přičemž jeho mediánové skóre bylo v průměru přibližně o půl bodu nižší než u nejlepšího modelu. Zajímavé je, že u tohoto modelu přidání ukázkových příkladů (Few-shot) vedlo k mediánovému zlepšení výsledků oproti variantě bez příkladů (Zero-shot), což nebylo u ostatních modelů tak výrazné.

Naopak Anthropic Claude 3.7 Sonnet vykazoval nejnižší mediánové hodnoty, přičemž rozdíl oproti ostatním modelům se pohyboval kolem tří bodů. Takto nízké výsledky tohoto modelu naznačují, že generovaný kód často neodpovídá stylistickým a syntaktickým normám jazyka Python.

Zajímavým zjištěním je i fakt, že OpenAI o3-mini-high v této úloze mírně překonal své vlastní výsledky z úlohy kalkulačky a to konkrétně o půl bodu. Tento posun může reflektovat nižší komplexitu úlohy. Naproti tomu mediánové skóre modelů Google Gemini 2.0 Pro Experimental a Anthropic Claude 3.7 Sonnet zůstalo stabilní a bez výraznějších změn.



Obrázek 5.11.: Statická analýza kódu – Konzolové vykreslování

Z podrobnější analýzy jednotlivých technik výzev vyplývá, že nejlepších výsledků bylo dosaženo pomocí Structured Prompt (Zero-shot) a Structured Prompt (Few-shot). Tyto přístupy vykazovaly nejvyšší mediánové skóre, vysoký průměr a zároveň nízkou směrodatnou odchylku, což ukazuje na jejich spolehlivost a stabilitu.

Za pozornost stojí, že technika Role Prompt, která v úloze kalkulačky patřila k nejúspěšnějším, zde naopak dosáhla výrazně horších výsledků. V úloze konzolového vykreslování vykazoval Role Prompt jednu z nejnižších průměrných i mediánových hodnot, přičemž se objevila i vyšší variabilita v kvalitě výstupů. Tento kontrast naznačuje, že účinnost jednotlivých výzev je úzce závislá na typu řešené úlohy.

Úkolníček

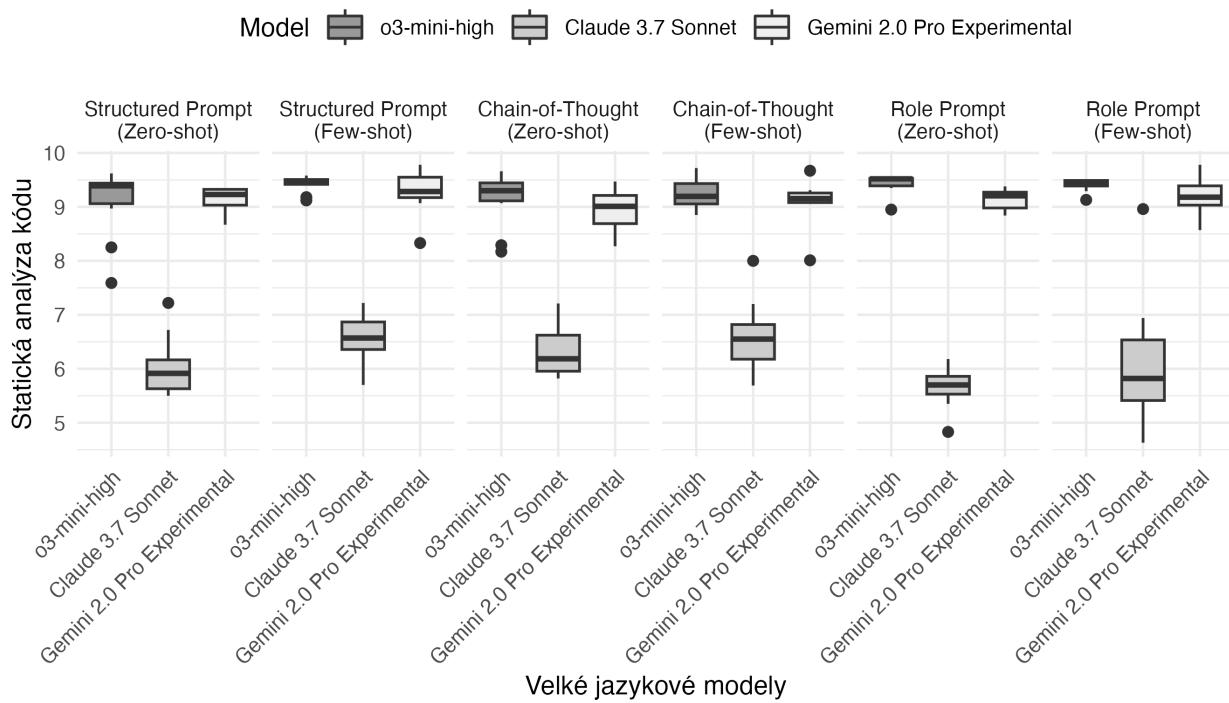
	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	9.39/10	5.91/10	9.23/10
Structured Prompt (Few-shot)	9.45/10	6.57/10	9.28/10
Chain-of-Thought (Zero-shot)	9.30/10	6.18/10	9.01/10
Chain-of-Thought (Few-shot)	9.19/10	6.55/10	9.15/10
Role Prompt (Zero-shot)	9.52/10	5.70/10	9.20/10
Role Prompt (Few-shot)	9.44/10	5.82/10	9.18/10

Tabulka 5.15.: Statická analýza kódu – Úkolníček

V závěrečné úloze zaměřené na generování kódu pro úkolníček dosáhl nejvyšších mediánových hodnot kvality napříč všemi variantami výzev model OpenAI o3-mini-high. Tento výsledek potvrzuje jeho schopnost konzistentně generovat kód s vysokou stylistickou a strukturální správností.

Na opačném konci spektra stál model Anthropic Claude 3.7 Sonnet, jenž vykazoval nejnižší úroveň formální správnosti. Model Google Gemini 2.0 Pro Experimental si udržel stabilní hodnocení s mediánovým skóre přibližně o čtvrt bodu nižším oproti modelu OpenAI o3-mini-high.

Z výsledků je patrné, že přidání ukázkových příkladů správně strukturovaného kódu (Few-shot) mělo nejvýraznější pozitivní dopad na modely Google Gemini 2.0 Pro Experimental a Anthropic Claude 3.7 Sonnet. U modelu OpenAI o3-mini-high vedlo přidání ukázek pouze k nepatrnému poklesu kvality, případně zachovalo obdobnou úroveň jako u variant bez příkladů (Zero-shot).



Obrázek 5.12.: Statická analýza kódu – Úkolníček

Podrobná analýza všech naměřených hodnot jednotlivých modelů ukazuje, že model OpenAI o3-mini-high dosahuje nejlepších výsledků ve většině typů výzev. U výzev typu Structured Prompt (Zero-shot) a Chain-of-Thought (Zero-shot) však vykazuje mírně vyšší variabilitu výsledků. Model Anthropic Claude 3.7 Sonnet opakovaně vykazoval nejnižší kvalitu výstupů napříč všemi typy výzev. Nejvýraznější pokles hodnocení byl zaznamenán při použití výzvy Role Prompt, kde tento model dosáhl nejnižšího skóre mezi všemi testovanými modely a variantami výzev. Zároveň tento model u výzvy Role Prompt (Few-shot) dosáhl neobvykle vysokého skóre, konkrétně devět bodů z deseti, což je pro tento model atypické.

Nejfektivnější technikou výzvy v této úloze se ukázal být Structured Prompt (Few-shot), který napříč všemi modely dosáhl nejvyššího průměrného i mediánového skóre.

Shrnutí

Testování kvality generovaného kódu pomocí statické analýzy odhalilo významné rozdíly v dodržování obecných standardů a stylistických konvencí programovacího jazyka Python jednotlivými jazykovými modely, kdy žádný z modelů nedokázal soustavně generovat kód, který by splňoval všechny požadavky na statickou analýzu.

Výsledky ukázaly, že neexistuje jediná univerzálně nejfektivnější technika výzvy. Úspěšnost přístupů se totiž výrazně liší v závislosti na kontextu a na konkrétním typu úlohy. Například technika Role Prompt vykázala výborné výsledky při generování kódu pro úlohu kalkulačky, zatímco pro jiné úlohy se ukázaly jako efektivnější jiné typy výzev.

Technika Few-shot měla ve většině případů mírně pozitivní dopad na kvalitu generovaného kódu, přičemž její účinnost nebyla rovnoměrná. U některých modelů a typů úloh byl její vliv výraznější, u jiných se téměř neprojevil.

Z pohledu statické analýzy vykazoval model OpenAI o3-mini-high konzistentně nejvyšší úroveň souladu s konvencemi a doporučenými standardy jazyka Python. Naopak model Anthropic Claude 3.7 Sonnet dosahoval nejnižší kvalitu generovaného kódu ve všech testovaných úlohách a variantách výzev, a to s rozdílem dvou až tří bodů ve srovnání s ostatními modely. Model Google Gemini 2.0 Pro Experimental vykazoval mediánová skóre podobná modelu OpenAI o3-mini-high, přičemž jejich výsledky byly ve většině případů velmi blízké.

5.6. Počet řádků

V rámci tohoto testu je hodnocen celkový počet řádků kódu. Ten lze použít jako zjednodušený ukazatel pro přibližné posouzení složitosti a potenciální udržovatelnosti kódu generovaného velkými jazykovými modely. Přestože jde pouze o hrubou metriku, její využití umožňuje identifikovat základní rozdíly v rozsahu generovaného kódu.

Výsledky v tabulkách jsou shrnutý pomocí mediánového počtu řádků kódu. Mediánové hodnoty vycházejí ze všech kompilovalních kódů vygenerovaných pomocí stejného typu výzvy, kterých bylo nejvíce deset. Tato metrika sama o sobě neumožňuje jednoznačně určit, zda delší či kratší kód představuje kvalitnější výstup. Slouží spíše jako orientační ukazatel rozdílů mezi modely a její interpretaci je vždy nutné vnímat v kontextu dalších podcharakteristik generovaného kódu.

Kalkulačka

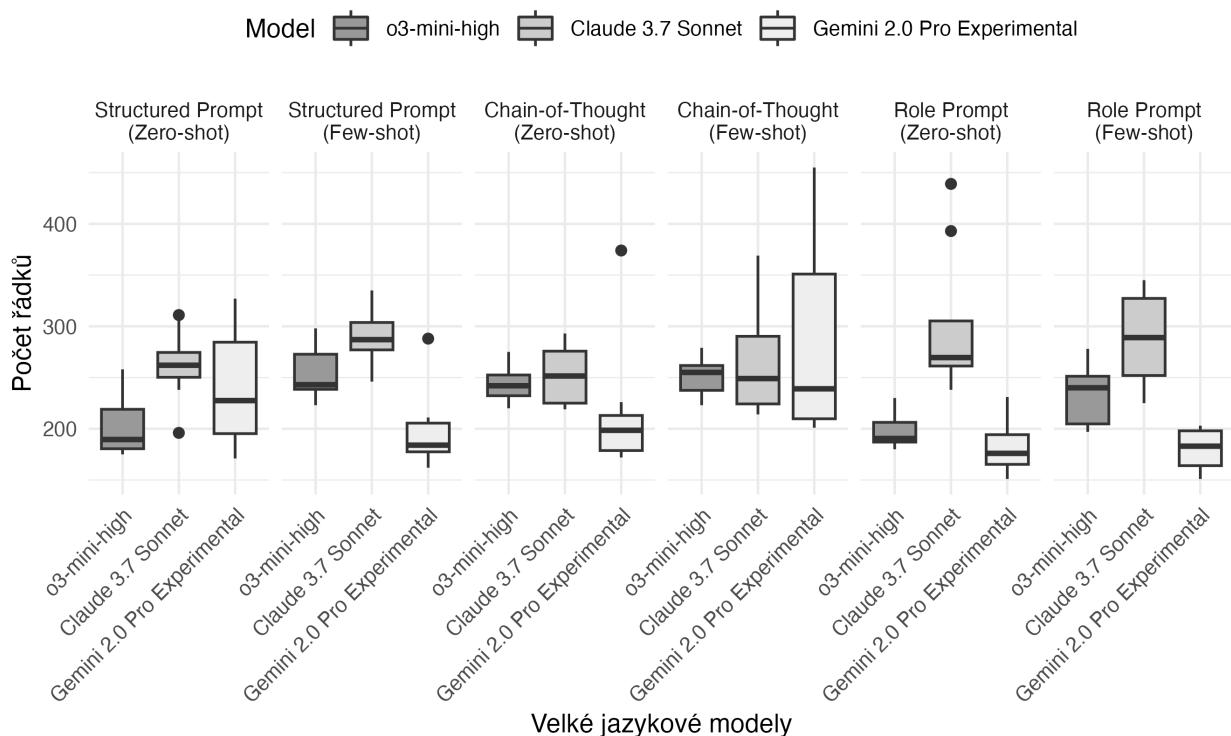
	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	190	262	228
Structured Prompt (Few-shot)	243	287	184
Chain-of-Thought (Zero-shot)	242	252	199
Chain-of-Thought (Few-shot)	255	249	239
Role Prompt (Zero-shot)	191	270	176
Role Prompt (Few-shot)	240	289	183

Tabulka 5.16.: Počet řádků – Kalkulačka

Z tabulkových výsledků implementace kalkulačky lze pozorovat výrazné rozdíly v rozsahu generovaného kódu. Nejkompaktnější výstupy produkoval model Google Gemini 2.0 Pro Experimental, zatímco Anthropic Claude 3.7 Sonnet generoval výstupy s nejvyšší mediánovou délkou.

Google Gemini 2.0 Pro Experimental se od ostatních modelů odlišoval nejen svou tendencí produkovat mediánově nejkratší výstupy, ale rovněž tím, že při použití komplexnějších typů výzv, jako jsou Chain-of-Thought (Zero-shot) či Role Prompt (Zero-shot), docházelo k redukci počtu generovaných řádků. Tento jev může naznačovat snahu modelu o zjednodušení nebo optimalizaci logiky řešení.

Při použití technik typu Few-shot je z mediánových hodnot patrná tendence k prodlužování výstupu, případně k zachování obdobné délky. Tento trend pravděpodobně souvisí s přítomností demonstračního kódu ve výzvě, který modely využívají jako šablonu pro generovaný výstup. Výjimku představoval model Google Gemini 2.0 Pro Experimental, jenž v případě techniky Structured Prompt (Few-shot) mediánově vygeneroval kratší kód než při použití Zero-shot alternativy.



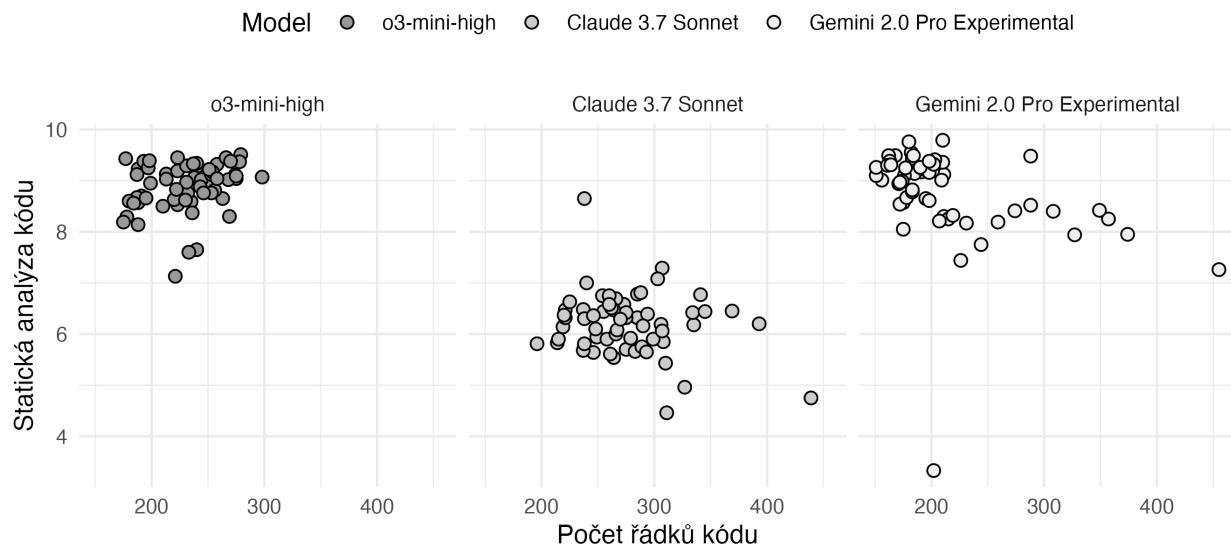
Obrázek 5.13.: Počet řádků – Kalkulačka

Detailní analýza pozorovaných délek generovaného kódu ukazuje výraznou variabilitu mezi jednotlivými velkými jazykovými modely i typy výzev. Nejnižší rozpětí hodnot vykazoval model OpenAI o3-mini-high, jehož maximální délka kódu nikdy nepřekročila přibližně 300 řádků kódu, což svědčí o vysoké konzistenci výstupů napříč různými výzvami. Z jednotlivých pozorování tohoto modelu je zároveň patrná tendence k postupnému prodlužování výstupů při použití technik typu Few-shot.

Naopak modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental vykazovaly podstatně širší rozpětí délky kódu, přičemž v extrémních případech dosahovala délka vygenerovaného kódu až k hranici 500 řádků. Tyto situace byly zpravidla spojeny s použitím technik Chain-of-Thought (Zero-shot), Chain-of-Thought (Few-shot) a Role Prompt (Few-shot), které pravděpodobně podporují podrobnější implementaci, rozsáhlejší komentování nebo implementaci základního testování.

Pozoruhodným zjištěním je také chování modelu Google Gemini 2.0 Pro Experimental, který i přes tendenci k tvorbě kompaktních výstupů vykazoval extrémní variabilitu, zejména při využití Chain-of-Thought (Few-shot). V rámci této výzvy model dokonce vytvořil jak nejkratší, tak i nejdelší výstupy napříč testovanými modely.

Z hlediska jednotlivých výzev vedla mediánově ke generování nejstručnějších výstupů technika Role Prompt (Zero-shot), zatímco nejdelší kód byl typicky vygenerován při použití varianty výzvy Chain-of-Thought (Few-shot).



Obrázek 5.14.: Počet řádků a statická analýza kódu – Kalkulačka

Bodový graf znázorňující vztah mezi délkou kódu a výsledky statické analýzy ukazuje, že model OpenAI o3-mini-high dosahoval nejlepšího poměru mezi stručností a kvalitou generovaného kódu. Podobný trend vykazuje i model Google Gemini 2.0 Pro Experimental, který však vedle krátkých a kvalitních výstupů produkoval dlouhé a méně kvalitní kódy. U tohoto modelu je patrná tendence ke snižování skóre statické analýzy s rostoucí délkou výstupu, což naznačuje, že delší kód není vždy přínosný. Stejný trend je patrný i u modelu Anthropic Claude 3.7 Sonnet, jehož skóre je obecně nižší a s narůstající délkou kódu dále klesá.

Konzolové vykreslování

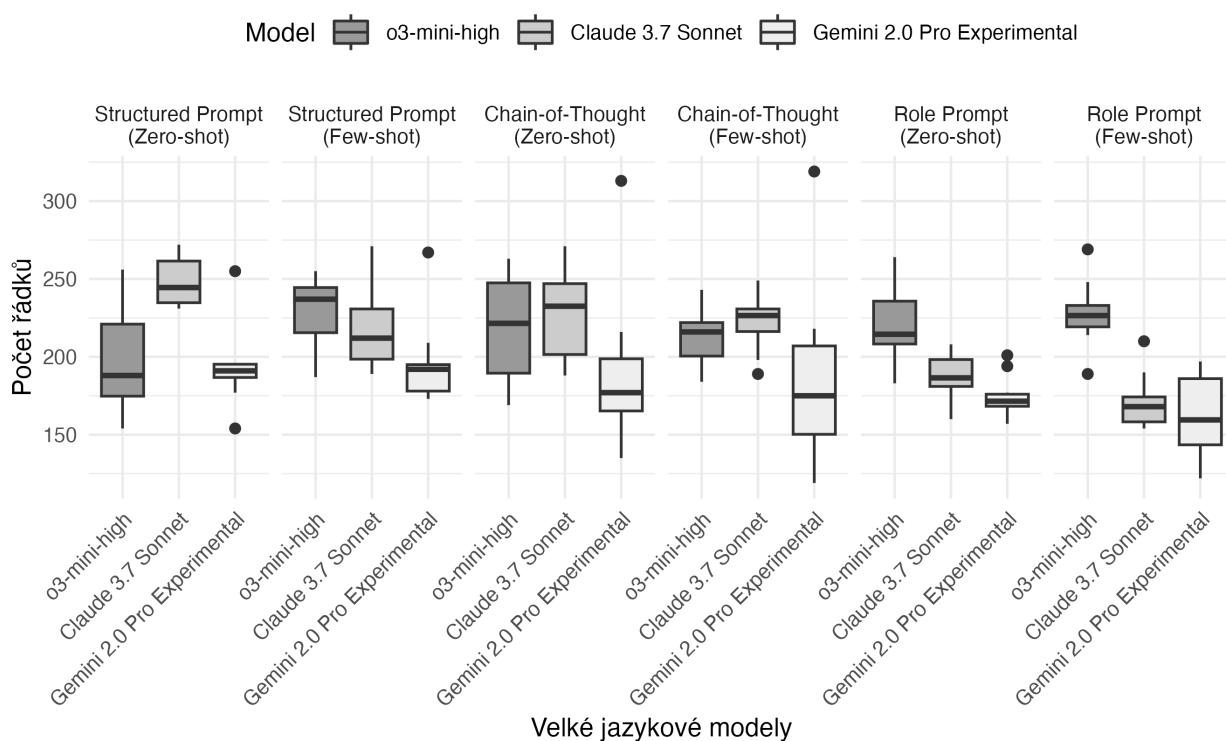
	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	188	245	191
Structured Prompt (Few-shot)	237	212	192
Chain-of-Thought (Zero-shot)	222	233	177
Chain-of-Thought (Few-shot)	216	227	175

Role Prompt (Zero-shot)	215	187	172
Role Prompt (Few-shot)	227	168	160

Tabulka 5.17.: Počet řádků – Konzolové vykreslování

V úloze konzolového vykreslování přetrvávají mezi modely rozdíly v rozsahu generovaného kódu. Nejdelší mediánové výstupy byly opakovaně generovány modely OpenAI o3-mini-high a Anthropic Claude 3.7 Sonnet, zatímco model Google Gemini 2.0 Pro Experimental produkoval mediánově stručnější kód.

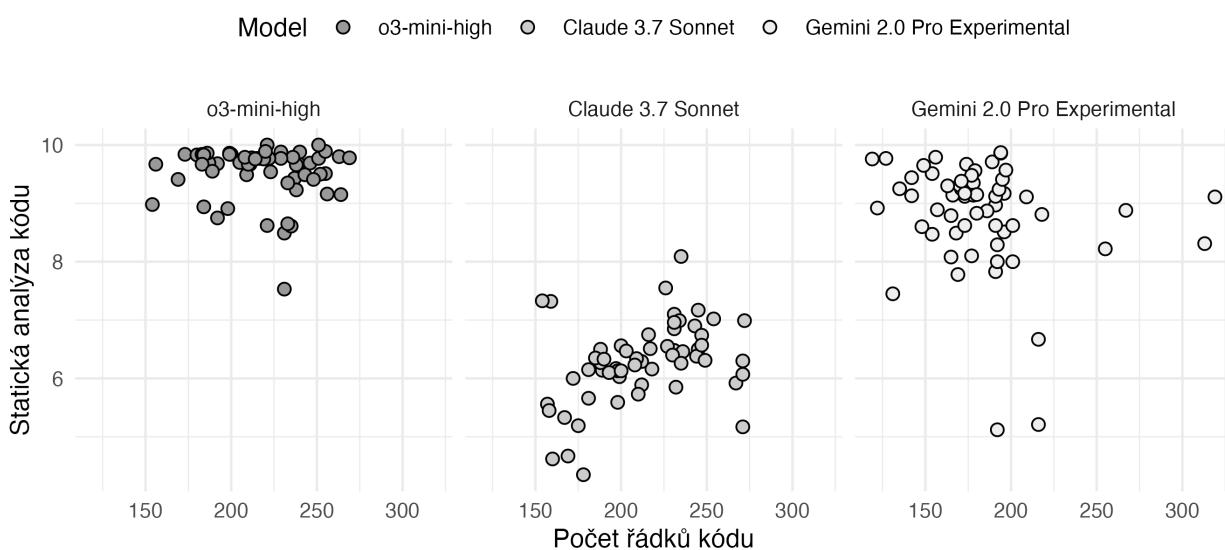
Model OpenAI o3-mini-high vykazuje stabilní mediánové výstupy napříč všemi variantami výzev. Naproti tomu modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro projevovaly výraznější tendenci ke zkracování výstupů při využití složitějších technik, jako jsou Chain-of-Thought nebo Role Prompt. Tento trend se pak u těchto modelů prohlubuje při použití výzvy s ukázkami (Few-shot), kdy dochází k dalšímu mediánovému zkrácení výstupů.



Obrázek 5.15.: Počet řádků – Konzolové vykreslování

Podrobnější analýza naměřených hodnot zvýrazňuje rozdíl mezi jednotlivými modely při použití různých technik inženýrství výzev. V rámci této testované úlohy se naměřené hodnoty pohybovaly v rozmezí 150 až 260 řádků, přičemž zvláštní pozornost si zaslouží model Google Gemini 2.0 Pro Experimental, který vykazoval značnou variabilitu v generovaných výstupech. Tato variabilita byla patrná zejména při použití technik Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot), kdy model generoval jak nejkratší, tak i nejdelší výstupy v celé úloze.

Z hlediska vlivu konkrétních výzev vedla ke generování nejkompaktnějších výstupů technika Role Prompt (Few-shot), zatímco nejdelší kód byl generován při využití techniky Chain-of-Thought (Few-shot).



Obrázek 5.16.: Počet řádků a statická analýza kódu – Konzolové vykreslování

Bodová vizualizace vztahu mezi délkou kódu a kvalitou výstupů v podobě skóre statické analýzy ukazuje, že model OpenAI o3-mini-high i v této úloze dosahoval nejlepšího poměru mezi rozsahem a kvalitou. Nejhorší kombinace těchto metrik u modelu OpenAI o3-mini-high odpovídala nejlepší kombinaci u modelu Anthropic Claude 3.7 Sonnet.

Model Google Gemini 2.0 Pro Experimental se vyznačuje výraznou variabilitou naměřených hodnot, kdy v některých případech vytvořil stručný a kvalitní kód, jindy naopak obsáhlé a nekvalitní výstupy. Výsledky modelu Anthropic Claude 3.7 Sonnet naznačují jeho nižší úroveň z hlediska kvality generovaného kódu ve srovnání s ostatními testovanými modely. Ačkoli v několika případech dochází ke zlepšení skóre statické analýzy s rostoucí délkou výstupu, tento trend je nutno interpretovat s rezervou vzhledem k omezenému počtu pozorování.

Úkolníček

	OpenAI o3-mini-high	Anthropic Claude 3.7 Sonnet	Google Gemini 2.0 Pro Experimental
Structured Prompt (Zero-shot)	255	233	237
Structured Prompt (Few-shot)	247	249	221
Chain-of-Thought (Zero-shot)	252	228	293
Chain-of-Thought (Few-shot)	239	257	226
Role Prompt (Zero-shot)	249	192	207
Role Prompt (Few-shot)	239	192	139

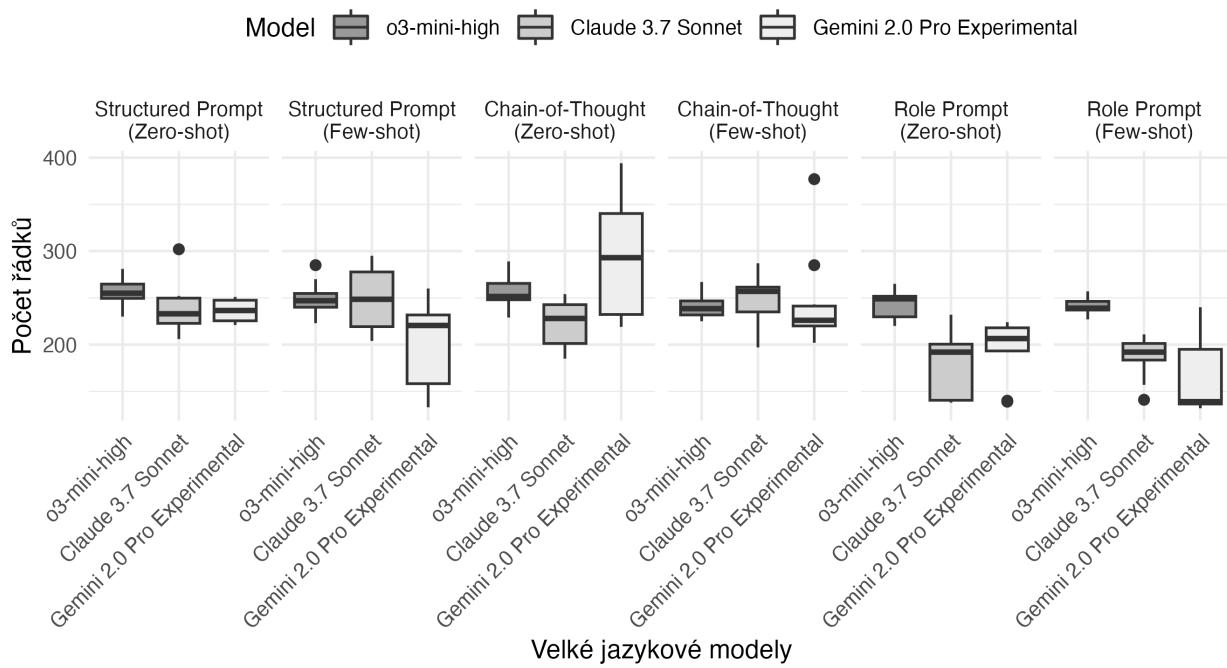
Tabulka 5.18.: Počet řádků – Úkolníček

Mediánové hodnoty počtu řádků kódu generovaného jednotlivými velkými jazykovými modely v úloze implementace úkolníčku vykazují odlišné tendenze než v předchozích úlohách. Přestože model OpenAI o3-mini-high nejčastěji produkuje nejdelší mediánové výstupy, v závislosti na typu výzvy je v této pozici doplňován ostatními modely. Zejména při použití technik Structured Prompt (Few-shot), Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot) dosahují mediánově nejdelších výstupů modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental.

Model OpenAI o3-mini-high vykazuje konzistentní mediánovou délku napříč všemi variantami výzev, přičemž lze identifikovat mírnou tendenci ke zkracování kódu při aplikaci Few-shot přístupů. Podobný trend je patrný také u modelu Google Gemini 2.0 Pro Experimental. Naproti tomu Anthropic Claude 3.7 Sonnet tento vzorec nenásleduje. U tohoto modelu se délka výstupů při použití Few-shot technik zpravidla nemění nebo roste ve srovnání s výstupy generovanými ve výzvě typu Zero-shot.

Za pozornost stojí vysoká variabilita mediánových délek kódu vygenerovaných modelem Google Gemini 2.0 Pro Experimental. Tento model vykazuje široké rozpětí mediánových hodnot v závislosti na použité technice. Výzva Chain-of-Thought (Zero-shot) vedla ke generování mediánově nejdelších výstupů v celé úloze, zatímco technika Role Prompt (Few-shot) produkovala nejkratší výstupy. Mediánový rozdíl mezi těmito technikami překračuje dvojnásobek, což ukazuje na výraznou citlivost modelu na typ výzvy.

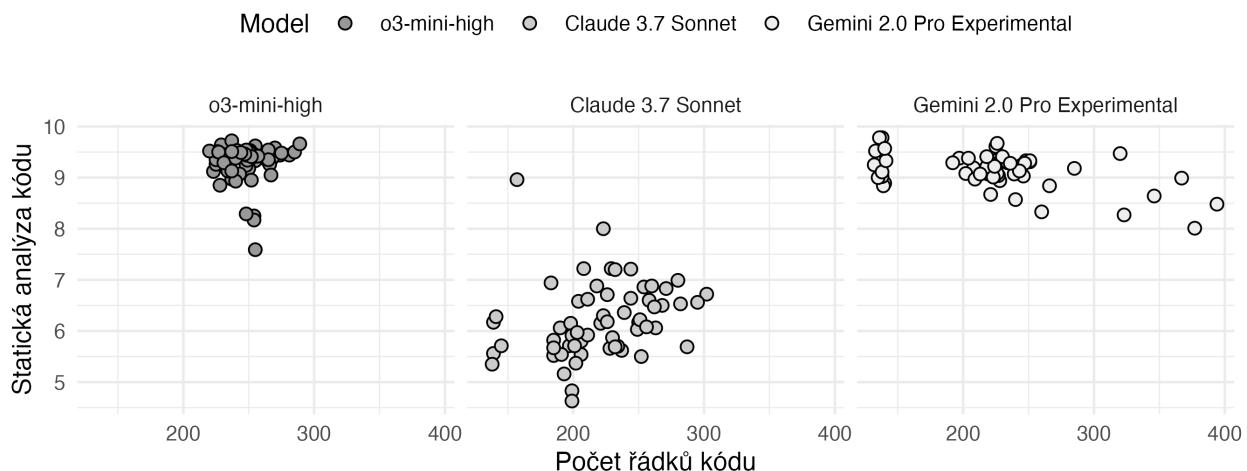
Z hlediska vlivu konkrétních výzev výsledky ukazují výsledky podobné trendy jako u úlohy konzolového vykreslování. Mediánově nejkompaktnější výstupy generovala technika Role Prompt (Few-shot), zatímco nejdelší kódy vznikaly při využití techniky Chain-of-Thought (Few-shot).



Obrázek 5.17.: Počet řádků – Úkolníček

Detailnější pohled na všechny naměřené délky generovaného kódu rozšiřuje interpretaci vyplývající z mediánových hodnot. Model OpenAI o3-mini-high se napříč všemi testovanými scénáři vyznačuje nejvyšší konzistencí a stabilitou. Variabilita jeho výstupů zůstává nízká i při změnách typu výzvy, což naznačuje nižší senzitivitu vůči způsobu formulace výzvy.

Naopak modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental generují kód s vyšší variabilitou délek. Nejvýraznější rozdíly byly pozorovány u modelu Google Gemini 2.0 Pro Experimental, jehož výstupy se pohybují v širokém rozpětí v závislosti na typu výzvy. Techniky Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot) u tohoto modelu vedly k výstupům dosahujícím až přibližně 400 řádků, což představuje více než dvojnásobek oproti jiným výzvám.



Obrázek 5.18.: Počet řádků a statická analýza kódu – Úkolníček

Analýza vztahu mezi délkou generovaného kódu a jeho kvalitou, měřenou pomocí skóre statické analýzy, ukazuje, že model OpenAI o3-mini-high znovu dosahuje nejlepšího poměru mezi těmito dvěma parametry. V porovnání s předchozími úlohami však lze pozorovat mírné zhoršení výsledků, kdy v několika případech byly identifikovány výraznější nedostatky v hodnocení kódu. Přesto si model udržuje vysokou konzistenci mezi rozsahem generovaného kódu a výsledky statické analýzy.

Model Google Gemini 2.0 Pro Experimental následuje podobný vývoj, avšak s podstatně vyšší variabilitou délky vygenerovaného kódu. Kromě toho se u tohoto modelu zřetelněji projevuje negativní vztah mezi délkou kódu a jeho kvalitou. U tohoto modelu dochází s rostoucím počtem řádků dochází ke zhoršení výsledků statické analýzy.

Model Anthropic Claude 3.7 Sonnet dosahuje nejnižšího skóre statické analýzy napříč všemi testovanými modely. Přestože je délka jeho výstupů srovnatelná s ostatními. Model dosahuje požadované kvality pouze ve výjimečných případech.

Shrnutí

Podrobné zkoumání délky generovaného kódu odhaluje zajímavé vzorce napříč různými modely a technikami výzev. Samostatná interpretace počtu řádků kódu generovaného jednotlivými modely musí být brána s opatrností, neboť délka kódu nemusí vždy odrážet přímo jeho kvalitu. Delší kód může znamenat buď větší detailnost, nebo naopak nadbytečnost, zatímco kratší kód může být výsledkem efektivity nebo absencí potřebných částí, jako jsou komentáře či validace vstupů.

Z provedené analýzy je patrné, že zvolený typ výzvy může výrazně ovlivnit rozsah generovaného kódu, přičemž konkrétní dopad se liší mezi jednotlivými modely. Z naměřených hodnot lze usuzovat, že technika Chain-of-Thought obecně vedla k delším výstupům než ostatní techniky, ačkoli tento trend byl výrazně ovlivněn specifiky jednotlivých modelů. Nejkratší výstupy naopak produkovala technika Role Prompt.

Modely Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental vykazují vyšší citlivost na typ výzvy, zejména u metod Chain-of-Thought (Zero-shot) a Chain-of-Thought (Few-shot), kde generovaly výrazně rozsáhlejší a variabilnější kód, přičemž v některých případech došlo k téměř dvojnásobnému nárůstu délky kódu oproti jiným výzvám. Naopak model OpenAI o3-mini-high si udržel vysokou konzistenci bez ohledu na použitou techniku. Tyto zjištění naznačují, že tento model je méně citlivý na různé formulace výzvy, což může být výhodou v praktických aplikacích, kde je důležitá stabilita a předvídatelnost výstupů.

5.7. Celkové hodnocení

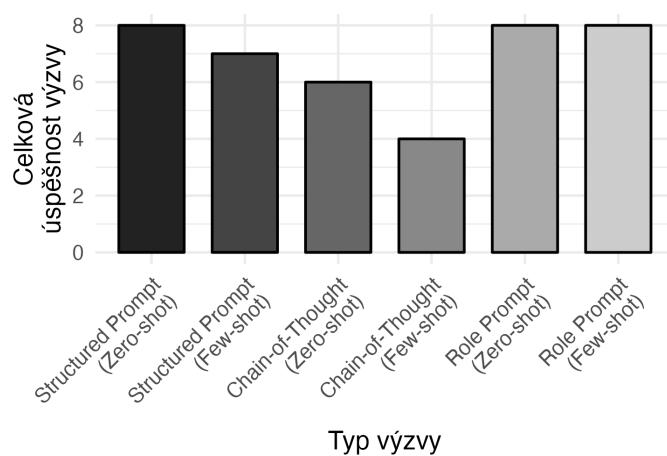
Tato část shrnuje výsledky hodnocení kvality generovaného kódu napříč velkými jazykovými modely a technikami inženýrství výzev. Celkové vyhodnocení vychází ze dříve analyzovaných metrik a jejich dílčích výsledků. Do hodnocení byly zahrnuty pouze ty podcharakteristiky, které bylo možné vyhodnotit nezávisle. Z tohoto důvodu nebyl do závěrečného hodnocení zařazen test délky generovaného kódu, neboť bez dodatečného kontextu jej nelze jednoznačně interpretovat.

S ohledem na různorodost metrik, kdy u některých vyšší hodnota značí lepší výsledek, zatímco u jiných jsou žádoucí nižší hodnoty, nebylo vhodné výsledky přímo agregovat do jednoho číselného skóre. Místo toho bylo použito bodové hodnocení. Pokud model nebo výzva v konkrétní úloze a podle dané metriky dosáhl nejlepšího výsledku, obdržel jeden bod úspěšnosti. Tento přístup umožnil porovnání napříč testy s různým charakterem hodnocení.

Celkové výsledky jsou prezentovány prostřednictvím grafických vizualizací. Pro metriky kompilovatelnosti a funkční úplnosti byly vybrány modely a techniky výzev vykazující nejvyšší absolutní úspěšnost. Ostatní podcharakteristiky kvality byly hodnoceny na základě kombinace průměrných hodnot, mediánu, směrodatné odchylky a mediánové absolutní odchylky. V případech, kdy více modelů či technik dosáhlo shodných výsledků, byly jako nejúspěšnější označeny všechny varianty.

Vyhodnocení kvality kódu podle typu výzvy

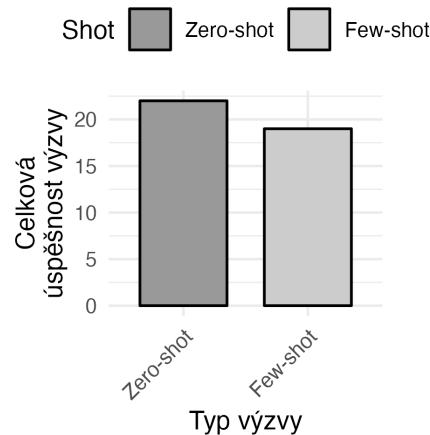
Porovnání úspěšnosti různých typů výzev ukazuje, že efektivita jednotlivých technik není univerzální, ale výrazně závisí na kombinaci velkého jazykového modelu, konkrétní úlohy a sledované metriky. Ve většině případů se žádná z technik neprojevila jako konzistentně nejúspěšnější napříč všemi testovanými úlohami.



Obrázek 5.19.: Úspěšnost jednotlivých typů výzev napříč úlohami

Nejvyšší četnosti úspěšnosti nejčastěji dosahovaly techniky Structured Prompt (Zero-shot), Role Prompt (Zero-shot) a Role Prompt (Few-shot). Naopak technika Chain-of-Thought vykazovala nižší efektivitu a dosáhla nejvyšší úspěšnosti pouze u omezeného počtu sledovaných metrik.

Grafická vizualizace dále ukazuje na rozdíly mezi výzvami typu Zero-shot (bez ukázek kódu) a Few-shot (s ukázkami kódu). Výsledky naznačují, že Zero-shot výzvy dosahovaly vyšší úspěšnosti častěji než Few-shot přístupy.

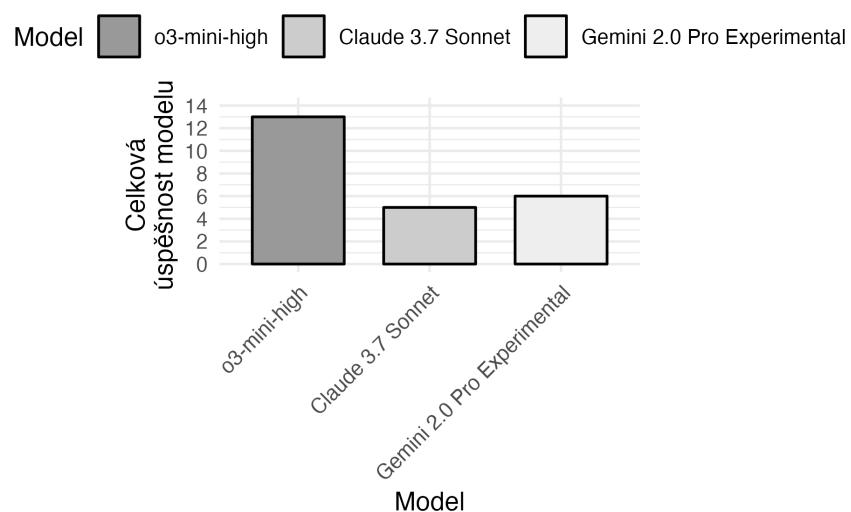


Obrázek 5.20.: Úspěšnost Zero-shot a Few-shot výzev napříč úlohami

Tento trend naznačuje, že velké jazykové modely byly schopné generovat kvalitní kód i bez dodatečných příkladů. Jednou z možných příčin by mohlo být, že výzvy již obsahovaly explicitní zadání generovat kvalitní kód dle předem definovaných standardů, aniž by potřebovaly dodatečné příklady. Ukázky kódu v rámci Few-shot výzev mohly naopak být pro modely rušivé, protože se zaměřovaly na napodobení příkladu místo optimalizace výstupu podle požadovaných kritérií kvality.

Vyhodnocení kvality kódu podle velkého jazykového modelu

Analýza ukázala, že i mezi špičkovými velkými jazykovými modely existují výrazné rozdíly v kvalitě generovaného kódu. Přestože některé modely dosahovaly stabilně lepších výsledků, žádný z nich nebyl absolutně nejlepší ve všech sledovaných úlohách a testovaných metrikách.



Obrázek 5.21.: Úspěšnost jednotlivých modelů napříč úlohami

Na základě četnosti dosažení nejlepšího umístění vyplývá, že nejvyšší úspěšnosti napříč sledovanými metrikami dosáhl model OpenAI o3-mini-high. Druhé místo obsadil model Google Gemini 2.0 Pro Experimental, nicméně s poměrně výrazným odstupem. Nejnižší úspěšnost pak zaznamenal model Anthropic Claude 3.7 Sonnet, který se umístil na posledním místě, přičemž rozdíl mezi ním a modelem Google Gemini 2.0 Pro Experimental byl minimální. Jednalo se o jeden test s úspěšným výsledkem navíc.

Odpovědi na výzkumné otázky

1. *Jak různé techniky inženýrství výzev ovlivňují kvalitu kódu generovaného velkými jazykovými modely?*

Techniky inženýrství výzev měly vliv na kvalitu generovaného kódu, avšak jejich účinnost byla silně závislá na konkrétním velkém jazykovém modelu, konkrétní úloze a sledované metrice. Žádná technika nebyla univerzálně nejlepší. Nejčastěji úspěšnými technikami byly Structured Prompt (Zero-shot), Role Prompt (Zero-shot) a Role Prompt (Few-shot).

2. *Jak se liší kvalita generovaného kódu mezi vybranými velkými jazykovými modely?*

Mezi zkoumanými velkými jazykovými modely existují významné rozdíly v kvalitě generovaného kódu. Nejčastěji nejlépe hodnoceným modelem byl OpenAI o3-mini-high, který získal nejvyšší úspěšnost napříč většinou testovaných úloh a sledovaných metrik. Ostatní modely vykazovaly přibližně poloviční úspěšnost.

Celkové shrnutí

Z výsledků analýz vyplývá, že výběr vhodného velkého jazykového modelu měl větší vliv na kvalitu generovaného kódu než použitá technika inženýrství výzev. Výkonnější modely byly schopny produkovat kvalitní výstupy i za použití základních výzev, zatímco méně výkonné modely vykazovaly nedostatečné výsledky i při aplikování pokročilých technik výzev.

Je však třeba zdůraznit, že žádný z modelů nedokázal generovat plně bezchybný kód, což potvrzuje, že i přes významný pokrok v oblasti velkých jazykových modelů je stále nezbytný lidský dohled a revize generovaných výstupů.

6. Závěr

Cílem této práce bylo zhodnotit kvalitu kódu generovaného vybranými velkými jazykovými modely. Výzkum se zaměřil na porovnání výstupů při použití různých technik inženýrství výzev a na rozdíly mezi jednotlivými modely při použití totožné výzvy. Hodnocení vycházelo z několika vybraných charakteristik kvality kódu, konkrétně z funkční vhodnosti, výkonové efektivity a udržovatelnosti. Analyzovány byly výstupy modelů OpenAI o3-mini-high, Anthropic Claude 3.7 Sonnet a Google Gemini 2.0 Pro Experimental.

V rámci práce byla navržena a implementována metodika pro porovnání kvality kódu generovaného různými velkými jazykovými modely. Nejprve byl proveden výběr testovaných velkých jazykových modelů a technik výzev, na který navazovala iterativní explorační analýza sloužící jako podklad pro návrh výzev. Následně byly definovány konkrétní úlohy s cílem pokrýt různorodé aspekty programování, které by měly být jazykovými modely zvládnutelné, a pro každou úlohu byly vytvořeny odpovídající výzvy.

Poté byly stanoveny klíčové charakteristiky a podcharakteristiky kvality kódu a vytvořeny nástroje pro jejich objektivní měření. Automatizační skripty umožnily efektivní generování kódu přes API rozhraní. Jiné automatizované skripty zajišťovaly konzistentní vyhodnocení bez vlivu subjektivity. Součástí návrhu byly i rozšiřující metriky jako měření využití výpočetních prostředků (CPU a RAM) a analýza modularity na základě struktury tříd, metod a funkcí, které však nebyly finálně použity do hodnocení.

Závěrečná fáze se soustředila na analýzu výsledků a jejich interpretaci s cílem identifikovat faktory ovlivňující kvalitu výstupního kódu. Na základě provedeného výzkumu se ukázalo, že největší vliv měla volba vhodného velkého jazykového modelu. Výsledky ukazují, že model OpenAI o3-mini-high nejčastěji dosahoval nejlepších hodnot napříč sledovanými podcharakteristikami. Ostatní modely vykazovaly menší konzistenci i nižší úroveň kvality. Vliv použitých technik výzev byl rovněž zřejmý, avšak méně výrazný. Z výsledků se podařilo jednoznačně určit techniky, které vedly k lepším výsledkům, a to Structured Prompt (Zero-shot), Role Prompt (Zero-shot) a Role Prompt (Few-shot).

Na základě provedeného výzkumu je patrné, že volba vhodné výzvy je značně závislá na konkrétním modelu. Proto by bylo vhodné v budoucím výzkumu postupovat odlišně. Nejprve se zaměřit na výběr optimálního velkého jazykového modelu a teprve následně pro něj hledat vhodnou formulaci výzvy. Tento přístup by umožnil přesněji oddělit vliv jednotlivých faktorů a lépe odhalit souvislosti mezi vlastnostmi modelu a efektivitou použitých technik inženýrství výzev.

Mezi hlavní omezení této práce patří malý datový vzorek, omezený počet úloh a doménové zaměření, stejně jako charakter výzkumu, který zcela nereflektuje běžnou praxi. V reálném prostředí bývá generování kódu iterativní, s důrazem na postupné úpravy a ladění výstupů, nikoli na spoléhání na výsledek první iterace. Rovněž je třeba zohlednit rychlý vývoj v oblasti velkých jazykových modelů, v jehož důsledku mohou být použité modely již zastaralé. Tato omezení snižují možnost zobecnění závěrů a ovlivňují celkovou validitu výsledků.

Do budoucna by bylo možné navázat rozšířením rozsahu experimentu, zapojením širší škály úloh, dat i programovacích jazyků. Zajímavým směrem by mohlo být i zkoumání dalších charakteristik kvality kódu nebo se případně zaměřit i na jiné fáze životního cyklu softwaru, například údržbu či refactoring. Tyto směry by mohly napomoci k hlubšímu pochopení možností a limitací velkých jazykových modelů napříč různými kontexty softwarového vývoje, a to nejen v prostředí konzolových aplikací v jazyce Python.

Seznam použitých zdrojů

1. CABALLAR, Rina Diane; STRYKER, Cole. *What are LLM benchmarks?* [online]. 2024. [cit. 2025-04-25]. Dostupné z: <https://www.ibm.com/think/topics/llm-benchmarks>.
2. JIANG, Juyong; WANG, Fan; SHEN, Jiasi; KIM, Sungju; KIM, Sunghun. *A Survey on Large Language Models for Code Generation*. 2024. Dostupné z arXiv: 2406.00515 [cs.CL].
3. PAUL, Debalina Ghosh; ZHU, Hong; BAYLEY, Ian. *Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review*. 2024. Dostupné z arXiv: 2406.12655 [cs.AI].
4. CHEN, Mark; TWOREK, Jerry; JUN, Heewoo; YUAN, Qiming; OLIVEIRA PINTO, Henrique Ponde de; KAPLAN, Jared; EDWARDS, Harri; BURDA, Yuri; JOSEPH, Nicholas; BROCKMAN, Greg; RAY, Alex; PURI, Raul; KRUEGER, Gretchen; PETROV, Michael; KHLAAF, Heidy; SASTRY, Girish; MISHKIN, Pamela; CHAN, Brooke; GRAY, Scott; RYDER, Nick; PAVLOV, Mikhail; POWER, Alethea; KAISER, Lukasz; BAVARIAN, Mohammad; WINTER, Clemens; TILLET, Philippe; SUCH, Felipe Petroski; CUMMINGS, Dave; PLAPPERT, Matthias; CHANTZIS, Fotios; BARNES, Elizabeth; HERBERT-VOSS, Ariel; GUSS, William Hebgen; NICHOL, Alex; PAINO, Alex; TEZAK, Nikolas; TANG, Jie; BABUSCHKIN, Igor; BALAJI, Suchir; JAIN, Shantanu; SAUNDERS, William; HESSE, Christopher; CARR, Andrew N.; LEIKE, Jan; ACHIAM, Josh; MISRA, Vedant; MORIKAWA, Evan; RADFORD, Alec; KNIGHT, Matthew; BRUNDAGE, Miles; MURATI, Mira; MAYER, Katie; WELINDER, Peter; MCGREW, Bob; AMODEI, Dario; MCCANDLISH, Sam; SUTSKEVER, Ilya; ZAREMBA, Wojciech. *Evaluating Large Language Models Trained on Code*. 2021. Dostupné z arXiv: 2107.03374 [cs.LG].
5. JACKSON, Gita. *What is the software development life cycle (SDLC)?* [online]. 2024. [cit. 2025-04-30]. Dostupné z: <https://www.ibm.com/think/topics/sdlc>.
6. GITHUB. *What is the SDLC?* [online]. 2025. [cit. 2025-04-30]. Dostupné z: <https://github.com/resources/articles/software-development/what-is-sdlc>.
7. GEEKSFORGEEKS. *Waterfall vs Iterative SDLC Model* [online]. 2023. [cit. 2025-04-30]. Dostupné z: <https://www.geeksforgeeks.org/waterfall-vs-iterative-sdlc-model/>.
8. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*. 1990, 1–84. Dostupné z DOI: [10.1109/IEEESTD.1990.101064](https://doi.org/10.1109/IEEESTD.1990.101064).
9. JAMWAL, Deepshikha. Analysis of Software Quality Models for Organizations. *International Journal of Latest Trends in Computing*. 2010, 1(2), 19–23. ISSN 2045-5364.

10. MIGUEL, José P.; MAURICIO, David; RODRÍGUEZ, Glen. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*. 2014, 5(6), 31–53. ISSN 0975-9018. Dostupné z DOI: 10.5121/ijsea.2014.5603.
11. BERANDER, Patrik; DAMM, Lars-Ola; ERIKSSON, Jeanette; GORSCHEK, Tony; HENNINGS-SON, Kennet; JÖNSSON, Per; KÅGSTRÖM, Simon; MILICIC, Drazen; MÅRTENSSON, Frans; RÖNKKÖ, Kari; TOMASZEWSKI, Piotr; LUNDBERG, Lars. Software quality attributes and trade-offs. 2005. Dostupné také z: https://www.researchgate.net/publication/238700270_Software_quality_attributes_and_trade-offs_Authors.
12. STARKE, Gernot. *Update on ISO 25010, version 2023* [online]. 2023. [cit. 2025-04-30]. Dostupné z: <https://quality.arc42.org/articles/iso-25010-update-2023>.
13. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 25010* [<https://iso25000.com/en/iso-25000-standards/iso-25010>]. 2011.
14. AMAZON WEB SERVICES. *What is Code Quality?* n.d. Dostupné také z: <https://aws.amazon.com/what-is/code-quality/>.
15. MUCCI, Tim. *What is Technical Debt?* [online]. 2025. [cit. 2025-05-01]. Dostupné z: <https://www.ibm.com/think/topics/technical-debt>.
16. TORNHILL, Adam; BORG, Markus. *Code Red: The Business Impact of Code Quality – A Quantitative Study of 39 Proprietary Production Codebases*. 2022. Dostupné z arXiv: 2203.04374 [cs.SE].
17. IBM. *What are large language models (LLMs)?* [online]. 2023. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/large-language-models>.
18. IBM. *What is NLP (natural language processing)?* [online]. 2024. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/natural-language-processing>.
19. IBM. *What is machine learning?* [online]. 2021. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/machine-learning>.
20. CHRYSTAL R. CHINA. *Five machine learning types to know* [online]. IBM, 2023. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/machine-learning-types>.
21. VASWANI, Ashish; SHAZEE, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. *Attention Is All You Need*. 2023. Dostupné z arXiv: 1706.03762 [cs.CL].
22. IBM. *What is a transformer model?* [online]. 2025. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/transformer-model>.
23. HAYWOOD, Sloan; WARREN, Genevieve; WOLF, Alex. *Understand tokens* [online]. 2024. [cit. 2025-05-03]. Dostupné z: <https://learn.microsoft.com/en-us/dotnet/ai/conceptual/understanding-tokens>.

24. OPENAI. *Tokenizer* [online]. 2025. [cit. 2025-05-03]. Dostupné z: <https://platform.openai.com/tokenizer>.
25. JOEL BARNARD. *What is machine learning?* [online]. IBM, 2023. [cit. 2025-05-03]. Dostupné z: <https://www.ibm.com/think/topics/embedding>.
26. CASTILLO, Francisco. *Embeddings: Meaning, Examples and How To Compute* [online]. 2023. [cit. 2025-05-03]. Dostupné z: <https://arize.com/blog-course/embeddings-meaning-examples-and-how-to-compute/>.
27. PARTHASARATHY, Venkatesh Balavadhani; ZAFAR, Ahtsham; KHAN, Aafaq; SHAHID, Arsalan. *The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities*. 2024. Dostupné z arXiv: 2408.13296 [cs.LG].
28. GOOGLE CLOUD. *Prompt engineering: overview and guide* [online]. 2025. [cit. 2025-05-02]. Dostupné z: <https://cloud.google.com/discover/what-is-prompt-engineering>.
29. IBM. *What is prompt engineering?* [online]. 2025. [cit. 2025-05-02]. Dostupné z: <https://www.ibm.com/think/topics/prompt-engineering>.
30. LI, Zihao; SHI, Yucheng; LIU, Zirui; YANG, Fan; PAYANI, Ali; LIU, Ninghao; DU, Mengnan. *Language Ranker: A Metric for Quantifying LLM Performance Across High and Low-Resource Languages*. 2024. Dostupné z arXiv: 2404.11553 [cs.CL].
31. DAIR.AI. *Elements of a Prompt* [online]. 2024. [cit. 2025-05-02]. Dostupné z: <https://www.promptingguide.ai/introduction/elements>.
32. BOONSTRA, Lee. *Prompt Engineering* [online]. 2025. [cit. 2025-05-02]. Tech. zpr. Google. Dostupné z: <https://www.kaggle.com/whitepaper-prompt-engineering>.
33. DAIR.AI. *Zero-Shot Prompting* [online]. 2024. [cit. 2025-05-02]. Dostupné z: <https://www.promptingguide.ai/techniques/zeroshot>.
34. DAIR.AI. *Few-Shot Prompting* [online]. 2024. [cit. 2025-05-02]. Dostupné z: <https://www.promptingguide.ai/techniques/fewshot>.

A. Externí přílohy

Externí přílohy této bakalářské práce jsou umístěny na adrese:

https://github.com/martinrenner/BP-Comparison_of_Code_Quality_Generated_by_Different_Large_Language_Models_in_Python.

Struktura repozitáře:

code/	Adresář s vlastními skripty využitými v praktické části
convertor_to_csv/	Adresář s konvertorem výsledků testů do CSV formátu
scraper/	Adresář se skriptem pro automatizované získávání výstupů
tests/	Adresáře s testovacími skripty pro jednotlivé úlohy
ascii_art/, calculator/, todo_list/	
generated/	Adresář s výstupy z velkých jazykových modelů
code/	Adresáře s extrahovaným Python kódem z výstupů
ascii_art/, calculator/, todo_list/	
responses/	Adresáře s kompletními výstupy
ascii_art/, calculator/, todo_list/	
prompts/	Adresáře s výzvami pro analyzované aplikace
ascii_art/, calculator/, todo_list/	
results/	Adresáře s výsledky měření kvality vygenerovaného kódu
ascii_art/, calculator/, todo_list/	
thesis.pdf	Text bakalářské práce v PDF
