INVESTIGATING THE EFFECTIVENESS OF LARGE LANGUAGE MODELS IN AUTOMATED SOFTWARE ENGINEERING

JIHO SHIN

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY OF ELECTRICAL ENGINEERING & COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

AUGUST 2025

Abstract

Artificial intelligence has reshaped automated software engineering (ASE), shifting practice from task-specific fine-tuning of deep learning (DL) models towards prompt-based interaction with large language models (LLMs). This work addresses the unresolved question of when or how to use traditional fine-tuned models versus LLM prompting for code and proposes guidance grounded in empirical evidence.

First, we investigate Machine Learning (ML) code generation with fine-tuned models. We observe that models perform better on ML tasks than on general programs. However, generated code improves developer performance primarily by hinting at correct APIs, suggesting generation decomposition and incorporating human feedback.

Secondly, we compare *GPT-4* prompting with fine-tuned baselines for ASE tasks, i.e. code summarization, generation, and translation. *GPT-4* excels with task-specific prompts in some tasks (e.g., comment generation), but fine-tuned models remain stronger for code generation and translation. A user study shows conversational, iterative prompting significantly boosts perceived usefulness and output quality.

Thirdly, we assess the neural test oracle generators, focusing on the correlation between textual similarity metrics (static) and test adequacy (dynamic). We find that textual similarity metrics correlate weakly with dynamic adequacy metrics like coverage and mutation score, so solely relying on static metrics is misleading. Thus, execution-based metrics should be primarily used for assessing test oracle effectiveness.

Lastly, we investigated the effectiveness of different sources for RAG-based LLM test generation. Specifically, we mined API documentation, GitHub issues, and StackOverflow Q&As to form the RAG database. We find that RAG significantly improves coverage and bug detectability. We also observe that GitHub issues are especially beneficial due to rich, context-specific usage examples and bug-inducing behaviours.

Overall, the dissertation demonstrates that combining domain adaptation (fine-tuning) and retrieval-based prompting can mitigate limitations of both paradigms. Future work will broaden the evaluation to non-functional requirements and extend to other tasks, e.g., issue resolving and feature implementation.

To the Author of all knowledge and wisdom.

Acknowledgements

First and foremost, I offer my deepest gratitude to my supervisors, Dr. Song Wang and Dr. Hadi Hemmati, whose insight, patience, and steady encouragement shaped every stage of this dissertation. I am equally thankful to my lab mates and colleagues, Dr. Nima Shiri Harzevili, Dr. Moshi Wei, Mohammad Mahdi Mohajer, Alireza DaghighFarsoodeh, Chung-yu Wang, Yinghang Ma, Hamed Taherkhani, Mohammad Abdollahi, and Melika Sepidband, for the lively discussions, debates, and the camaraderie that kept research both rigorous and fun. I am deeply thankful to my PhD supervisory committee: Dr. Zhen Ming (Jack) Jiang and Dr. Marios Fokaefs. Their constant guidance, critiques, and support have been vital to my work. To my family and friends, thank you for your unwavering love, for believing in me, for celebrating each small milestone, and for reminding me to keep perspective. All your support, kindness, and constant presence made this journey possible and meaningful. I am truly indebted to each of you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Artificial intelligence has advanced dramatically over the last few years, driving major breakthroughs in areas such as natural language processing [1, 2, 3], computer vision [4, 5, 6], and automated software engineering (ASE) [7, 8, 9]. Recent works highlight both the promise and the obstacles of AI-based techniques in software development, demonstrating significant gains while also revealing continuous challenges [10, 11, 12, 13]. A prime example of this trend is the automatic generation of source code and tests, which could greatly boost developer productivity and lessen manual effort.

Historically, most AI code generators have begun with large pre-trained models that are then fine-tuned on target datasets. Such fine-tuned systems excel when large labelled data are available, transferring broad coding knowledge into domain-specific contexts. Empirical studies confirm that these models often outperform more general approaches in specialized tasks, including automated program repair [14, 15, 16], test generation [17, 18, 19], and vulnerability detection [20, 21, 22], yet they demand extensive annotated data and tend to struggle when applied beyond their training domains.

By contrast, recent foundation models (FMs) and large language models (LLMs) such as *GPT-4o* [23] provide a flexible alternative, using massive pre-training on diverse corpora to handle diverse tasks from simple natural-language prompts without extra fine-tuning. While these FMs and LLMs can still make domain-specific errors, techniques like In-Context Learning [24], Retrieval-Augmented Generation [25], Chain-of-Thought prompting [26], and Self-Consistency sampling [27] have significantly improved their reliability. This move from fine-tuning toward engineered prompting opens new possibilities for automated coding and testing, with FMs and LLMs now

outperforming traditional methods in several software engineering scenarios [10, 28, 29]. Nonetheless, these models still face challenges, such as heavy computational requirements, a tendency to hallucinate plausible but incorrect outputs [30], and difficulty in tasks requiring deeply specialized knowledge. Thus, there remains a shortage of guidelines on which techniques work best in which contexts.

Problem Statement: Despite the advancements and challenges, there is limited guidance on when to use traditional fine-tuning versus prompt engineering of LLMs in automated software engineering.

Dissertation Statement: To mitigate the aforementioned problem statement, this dissertation examines the shift from traditional fine-tuned models to prompt-based approaches with FMs and LLMs in automated software engineering, analyzing their respective strengths and limitations, and offering guidance on optimal usage based on specific requirements.

1.2 Taxonomy of Automated Software Engineering

Before presenting the study, we introduce a taxonomy designed to organize the diverse AI-driven research in ASE. This classification highlights three key dimensions, i.e., input, models, and evaluation, to frame the various studies and indicate where our investigations focus.

1. **Input**: denotes the specific task and data domain addressed by each study. Examples of ASE tasks include code generation, test generation, oracle generation, code summarization, code translation, etc. Domains can include both general-purpose programming and specialized fields, such as machine/deep learning (ML/DL), library or API usage, open-source, real-world projects, etc.

2. **Models**: covers the AI techniques and architectures applied. At a high level, approaches can be distinguished by the learning paradigm, i.e., fine-tuning versus prompt engineering, and further by details such as pre-training architectures, learning objectives, prompt design strategies, and supplementary components like instruction tuning or reinforcement learning from human feedback (RLHF).

3. **Evaluation**: concerns the criteria and methodologies used to evaluate the target models. Studies may focus on different aspects for evaluation: 1) functional requirements (e.g., correctness of generated code) or non-functional requirements

(e.g., performance, security, and maintainability), 2) utilizing different sets of evaluation metrics, such as execution-based metrics (e.g., code/test execution) or static, text-similarity measures, 3) applying different methodologies such as quantitative benchmarking, qualitative use case studies or surveys, etc.

1.3 Dissertation Overview

This section provides an overview of the studies included in this dissertation. The complete road map is illustrated in Figure 1.1, which is organized into three phases and four chapters. Each phase examines three AI paradigms: sole focus on traditional fine-tuning, a comparison of both fine-tuning and prompt engineering, and a sole focus on prompt engineering. The four main chapters correspond to the four studies conducted for this dissertation. These chapters are designed to be self-contained, which may result in some overlap in information. Additionally, there may be variations in the abbreviations and terminology used across the papers.

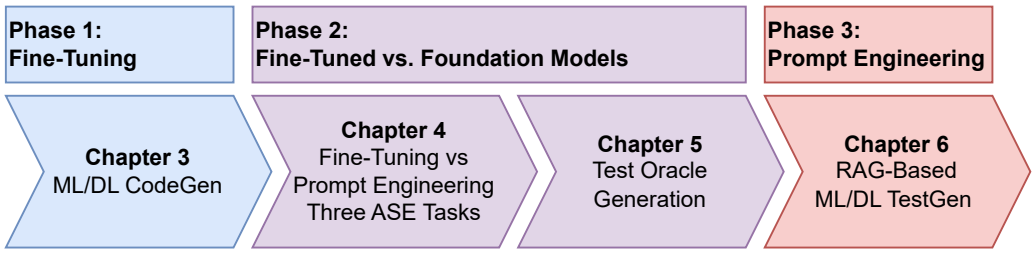


Figure 1.1: The road map of this dissertation.

1.3.1 Phase: Traditional Fine-Tuned Models

In this phase, we evaluate the state of ASE by examining traditional fine-tuned models before the introduction of LLMs and prompt engineering.

Chapter 3: Fine-Tuned Models on Machine Learning Tasks

In this chapter, we investigate the effectiveness of fine-tuned models at generating Machine Learning (ML) code, assessing their capabilities, limitations, and opportunities for improvement. According to the taxonomy introduced earlier, we focus on ML code generation (Input) via various fine-tuning strategies (Model) and evaluate

functional correctness using traditional textual similarity metrics, such as BLEU (Evaluation).

1.3.2 Phase 2: Jointly Assessing Fine-Tuned and LLMs

In this phase, we conduct a comparative analysis between traditional fine-tuned models and LLM prompting across ASE tasks, highlighting the shift in methodology and effectiveness.

Chapter 4: Prompt Engineering vs. Fine-Tuning in Three ASE Tasks

Next, a comparative analysis is carried out between traditional fine-tuned models and prompting LLMs for ASE tasks. We examine whether prompt engineering with *GPT-4* can outperform existing fine-tuned models on code generation, summarization, and translation tasks. Following our taxonomy, we address three ASE tasks, code summarization, generation, and translation in general programming (Input), using both prompt engineering and fine-tuning (Model). We quantitatively evaluate performance with textual similarity and execution-based metrics, and complement this with a qualitative user study on conversational prompting patterns (Evaluation). The goal is to determine which approach, prompting *GPT-4* or smaller fine-tuned models, yields better results and how each can be refined for ASE.

Chapter 5: Fine-Tuned Models vs. LLMs in Test Oracle Generation

Here, we compare fine-tuned Neural Oracle Generation (NOG) models with prompt-based LLMs (*GPT-3.5*) for test oracle generation. We evaluate both textual similarity and test adequacy, using coverage and mutation scores, to understand how well each approach captures intended test behaviour. According to our taxonomy, we focus on test oracle generation for general Java programs (Input) via fine-tuning and prompt engineering (Model) and assess functionality with both textual and execution-based metrics (Evaluation).

1.3.3 Phase 3: Improving LLM Prompting

This phase targets enhancements to modern LLMs by addressing limitations such as hallucination through advanced prompts and context engineering.

Chapter 6: LLMs on Retrieval-Augmented Test Generation

In this chapter, we explore Retrieval-Augmented Generation (RAG) for automated unit test creation. We assess whether RAG-based LLMs can produce more effective unit tests by leveraging three knowledge sources—API documentation, GitHub issues, and StackOverflow Q&A—across five popular Python ML/DL libraries: TensorFlow, PyTorch, Scikit-learn, Google JAX, and XGBoost. In line with our taxonomy, we examine test generation for ML/DL libraries (Input) using RAG-based prompting (Model) and evaluate results with execution-based metrics (Evaluation).

1.4 Dissertation Contributions

This dissertation makes the following contributions:

- We present the first empirical study that evaluates traditional fine-tuned models for ML code generation. In this study, we analyze the strengths and weaknesses of these models and propose guidelines for future research.
- We utilize a mixed-method approach, combining both quantitative and qualitative analyses, to compare traditional fine-tuned models with LLM prompting across three tasks in ASE: code generation, summarization, and translation.
- We assess state-of-the-art fine-tuned neural oracle generators against LLM prompting, examining the statistical correlations between textual similarity and test adequacy metrics. This examination serves as a caution against relying solely on textual similarity.
- We investigate the impact of various external knowledge sources on retrieval-augmented unit test generation for ML/DL APIs, and we provide recommendations for context engineering in complex ASE tasks.

1.5 List of Papers

The studies presented in this dissertation have been published (or are under review) in the following venues, listed in chronological order. My main contributions to these papers include ideation, planning, collecting and processing raw data, conducting experiments, analyzing results, and writing. Below is the list of papers:

Paper 1: The Good, the Bad, and the Missing: Neural Code Generation for Machine Learning Tasks

J. Shin, M. Wei, J. Wang, L. Shi, S. Wang
ACM Transactions on Software Engineering and Methodology **(TOSEM 2023)**. 24 pages.

Paper 2: Assessing Evaluation Metrics for Neural Test Oracle Generation
J. Shin, H. Hemmati, M. Wei, S. Wang
IEEE Transactions on Software Engineering **(TSE 2024)**. 13 pages.

Paper 3: Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code
J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, H. Hemmati
2025 IEEE/ACM 22nd International Conference on Mining Software Repositories **(MSR 2025)**. 13 pages. Acceptance Rate=29% (47/161).

Paper 4: Retrieval-Augmented Test Generation: How Far Are We?
J. Shin, N.S. Harzevili, R. Aleithan, H. Hemmati, S. Wang
Under Review **(ICSE 2026)**. 13 pages.

The following papers were written concurrently with the publications mentioned above during the program. Although these papers are not directly related to this dissertation, they explore methods for improving AI4SE (Artificial Intelligence for Software Engineering) and/or SE4AI (Software Engineering for Artificial Intelligence). The list includes both first-authored and co-authored papers.

A. API Recommendation for Machine Learning Libraries: How Far Are We?
M. Wei, Y. Huang, J. Wang, J. Shin, N.S. Harzevili, S. Wang
ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE 2022)**. 12 pages. Acceptance Rate=22% 99/449.

B. An Empirical Study on the Stability of Explainable Software Defect Prediction
J. Shin, R. Aleithan, J. Nam, J. Wang, N.S. Harzevili, S. Wang
IEEE 30th Asia-Pacific Software Engineering Conference **(APSEC 2023)**. 10 pages. Acceptance Rate=35% (43/128). Distinguished Paper Award.

C. Characterizing and Understanding Software Security Bugs in Machine Learning Libraries
N. S. Harzevili, J. Shin, J. Wang, S. Wang, N. Nagappan
2023 IEEE/ACM 20th International Conference on Mining Software Repositories **(MSR 2023)**. 12 pages. Acceptance Rate=37% (43/115).

D. Automatic Static Vulnerability Detection for Machine Learning Libraries: Are We There Yet?

N.S. Harzevili, J. Shin, J. Wang, S. Wang, N. Nagappan
2023 IEEE 34th International Symposium on Software Reliability Engineering **(ISSRE 2023)**. 12 pages. Acceptance Rate=25% (62/247).

E. Domain Adaptation for Code Model-Based Unit Test Case Generation

J. Shin, S. Hashtroudi, H. Hemmati, S. Wang
2024 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis **(ISSTA 2024)**. 12 pages. Acceptance Rate=21% (143/694).

F. Pre-trained Models for Bytecode Instructions

D. Kim, T. Kim, J. Shin, S. Wang, H. Choi, J. Nam
2025 18th IEEE International Conference on Software Testing, Verification and Validation **(ICST 2025)**. 5 pages. Acceptance Rate=42% (11/26).

G. Checker Bug Detection and Repair in Deep Learning Libraries

N.S. Harzevili, M.M. Mohajer, J. Shin, M. Wei, G. Uddin, J. Yang, J. Wang, S. Wang, Z. M. Jiang, N. Nagappan
Under Review **(TOSEM 2025)**. 12 pages.

H. SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection

M. B. U. Ahmed, N. S. Harzevili, J. Shin, H. V. Pham, S. Wang
Under Review **(NeurIPS-Data 2025)**. 17 pages.

I. StaAgent: An Agentic Framework for Testing Static Analyzers

E. Nnorom, M. B. U. Ahmed, J. Shin, H. V. Pham, S. Wang
Under Review **(ICSE 2026)**. 12 pages.

J. Toward Automated Validation of Language Model Synthesized Test Cases using Semantic Entropy

H. Taherkhani, J. Shin, M. A. Tahir, M. R. H. Misu, V. S. Gattani, H. Hemmati
Under Review **(TSE 2026)** 21 pages.

K. BloomAPR: A Bloom's Taxonomy-based Framework for Assessing the Capabilities of LLM-Powered APR Solutions

Y. Ma, D. S. Leuson, J. Shin, S. Wang, F. Khomh, S. H. Tan, Z. M. Jiang
Under Review **(TOSEM 2025)** 12 pages.

L. Surveying the Benchmarking Landscape of Large Language Models in Code Intelligence

M. Abdollahi, R.Zhang, N. S. Harzevili, J. Shin, S. Wang, H. Hemmati

Under Review **(TOSEM 2026)** 46 pages.

1.6 Dissertation Organization

The organization of the dissertation is as follows: Chapter 2 shows an extensive list of literature review following the aforementioned taxonomy. Chapter 3 examines the strengths and weaknesses of fine-tuned models in ML code generation. Chapter 5 compares the effectiveness of prompt engineering versus fine-tuning regarding the three most studied application-specific tasks in automated software engineering. Chapter 4 presents our research on neural test oracle generators and highlights the importance of evaluation metrics. Chapter 6 investigates the effectiveness of retrieval-augmented unit test generation for ML/DL APIs. Finally, Chapter 7 concludes the dissertation and explores potential directions for future research.

Chapter 2

Literature Review

This section delves into a comprehensive literature review on automated software engineering tasks, specifically code and test generation studies using fine-tuned and large language models. First, we divide the studies into code and test generation tasks. Then we sub-categorize based on the taxonomy mentioned in Chapter 1, i.e. Input, Model, and Evaluation. Since most papers deal with all aspects (Input, Models, and Evaluation), we categorize them based on the paper's emphasis. We also introduce other software engineering tasks that are commonly investigated in the field, i.e. code summarization, code translation, and automatic program repair (APR).

2.1 Code Generation

2.1.1 Input: Domain/Task-Specific Data

Prenner et al. [31] considered the efficacy of pre-trained *Transformers* on small software engineering datasets and found them competitive, especially for natural language (NL) tasks, while traditional methods often excel on very small source code datasets; it also explores techniques like active learning and data augmentation to enhance performance on limited data. Zan et al. [32] presented *CERT*, a novel library-oriented code generation model that uses continual pre-training on anonymized sketches to enhance code generation without needing labelled data; experiments show it improves baseline accuracy by over 15% on *PandasEval* and *NumpyEval* benchmarks for third-party library code generation. Zan et al. [33] proposed a framework to enable pre-trained LMs to generate code with private library APIs, introducing a novel two-module approach: *APIRetriever* to identify relevant APIs from documentation and *APICoder* to generate code using these APIs; evaluated on custom

benchmarks, the method shows marked improvements over baseline models in private library code generation. Yang et al. [34] proposed *ExploitGen*, a method for automating exploit code generation through a novel template-augmented approach leveraging *CodeBERT*, achieving significant improvements in generating contextually and semantically accurate exploit code. Yang et al. [35] introduced *TurduckenGen*, a novel syntax-guided multi-task learning model tailored for generating Turducken-style code, which efficiently combines declarative and imperative programming within the same framework. *TurduckenGen* addresses three core challenges: syntactic constraint representation, integration, and scalable syntax-first decoding, by employing an enhanced multi-task architecture with *CodeT5* and innovative techniques like the *SAT* algorithm and *SF-Beam Search*, achieving significant improvements over baselines in terms of syntactic accuracy and executable code quality. Yan et al. [36] presented the first large-scale empirical study evaluating *ChatGPT*'s ability to generate code for competition-level problems, revealing that, with optimized prompts, *ChatGPT* can achieve 25.4% test case accuracy, particularly leveraging feedback for partial solutions, while maintaining high-quality, clean code standards similar to professional benchmarks. Zhuge et al. [37] introduced the "Agent-as-a-Judge" to use agentic systems to evaluate other agentic systems, addressing limitations of traditional methods by providing rich intermediate feedback, and find that it aligns more closely with human evaluators than previous models, significantly reducing evaluation time and costs. Zhang et al. [38] assembled a benchmark for assessing LLM code generation in audio programming using visual dataflow languages, showing that meta-programming leads to more semantically correct outputs than direct JSON generation, especially when well-formed, and highlighting how richer meta-programming prompts create more complex code structures. Huang et al. [39] proposed *EFFI-CODE*. This dataset enhances code efficiency and correctness in LMs through a self-optimization process based on profiling execution overheads, significantly improving model performance by reducing execution time and increasing accuracy across benchmarks. Cherniavskii et al. [40] proposed the *STREAM* framework, which uses LLMs to generate code that enables spatiotemporal embodied reasoning in dynamic environments, achieving notable improvements in Embodied Question Answering by integrating vision-LMs, perception, and memory with code generation to answer environment-grounded questions effectively. Zhang et al. [41] investigated *GPT-3*'s AI capabilities in generating and parallelizing code for high-performance computing (HPC) software across C++, Fortran, Python, and Julia, finding that established languages and mature programming models like *OpenMP* and CUDA show high accuracy. At the same time, novel or complex tasks need prompt engineering for improvement. Liang et al. [42] proposed *Eurekaverse*, an innovative environment curriculum generation method using LLMs

to create progressively challenging tasks autonomously, enabling quadrupedal robots to master parkour skills efficiently, outperforming manually-designed courses in both simulations and real-world tests. Yin et al. [43] introduced the *PEAR* framework, a multi-agent system powered by LLMs to automate Ptychography parameter tuning, significantly improving workflow robustness and adaptability by combining custom knowledge bases, human-in-the-loop (HITL) controls, and flexible automation levels. Chen et al. [44] proposed *ScienceAgentBench*, a novel benchmark for rigorously assessing language agents' effectiveness in automating data-driven scientific discovery by evaluating their code generation for 102 real-world tasks from various disciplines, revealing current limitations as the best-performing model solved only 34.3% of tasks even with expert support. Black et al. [45] investigated commercial LLMs for secure code generation, finding that security-oriented prompts reduce vulnerabilities but can compromise code correctness, with newer models offering more secure outputs but at a higher computational cost. Kessel et al. [46] introduced *LASSO*, a platform that standardizes test-driven software experimentation by providing a streamlined scripting language (LSL) for creating scalable, reproducible studies to benchmark code generation reliability in LLMs. Cui et al. [47] benchmarked 16 LLMs on *WebApp1K* for web application code generation, finding that model differentiation mainly lies in mistake minimization, as most models possess similar knowledge but vary in error frequency; prompt engineering provides limited benefit beyond specific cases. Sarschar et al. [48] presented *PACGBI*, which integrates LLMs into *GitLab* CI to automate software development by generating code directly from backlog items, pushing changes, and creating merge requests, significantly reducing development time and costs. Zhong et al. [49] proposed *ROBUSTAPI*, a new benchmark dataset and evaluation framework to assess the reliability and robustness of LLM-generated code, revealing significant API misuse across models like *GPT-4* and *Llama-2* when generating Java API code. Yu et al. [50] introduced *CoderEval*, a new benchmark designed to evaluate code generation models on real-world, contextually-dependent code, highlighting the models' weaknesses in generating non-standalone functions. Toth et al. [51] evaluated security vulnerabilities in PHP code generated by *GPT-4*, finding significant risks, especially in insecure file upload and *SQL* injection, underscoring the need for rigorous testing when using AI-generated code in web applications.

2.1.2 Model: Novel Framework for Code Generation

Xu et al. [52] provided a systematic evaluation of large code models and introduced *PolyCoder*, a novel open-source model trained across 12 languages, finding that it outperforms *Codex* in the C language while emphasizing the need for publicly

accessible, multi-language code models. Lahiri et al. [53] proposed an interactive test-driven code generation framework, *TICODER*, which utilizes user feedback to formalize intent through test case generation, significantly improving code generation accuracy via a novel ranking and mutation approach. Doderlein et al. [54] investigated how variations in input parameters like prompt structure, temperature settings, and the number of generated solutions impact the quality of code generated by AI-based coding assistants like *Copilot* and *Codex*, revealing that optimal configurations vary significantly by problem, thus challenging the robustness and usability of these models. Shi et al. [55] investigated layer-wise encoding of code properties in pre-trained models during fine-tuning, revealing that lexical, syntactic, structural, and semantic properties reside in different model layers and proposing an efficient layer-freezing fine-tuning method ("Telly") that reduces computational cost while maintaining or improving task performance. Wang et al. [56] presented *CoCoGAN*, a novel *GAN*-based approach that improves both code generation and code search by treating these tasks as complementary within a generative adversarial framework; experiments show *CoCoGAN* increases *CodeBLEU* for code generation by up to 32% and mean reciprocal rank for code search by 12%. Zelikman et al. [57] presented the Self-Taught Optimizer (*STOP*), a framework where an LM recursively improves code by iteratively refining its scaffolding structure, achieving better code generation performance without altering the LM itself, and highlighting strategies such as genetic algorithms, beam search, and sandbox circumvention risks. Zhang et al. [58] proposed the Planning-Guided *Transformer* Decoding (*PG-TD*), a novel algorithm that integrates a planning approach with *Transformer*-based LMs to enhance code generation accuracy by incorporating test cases during generation, significantly outperforming traditional methods in quality and efficiency. Zhang et al. [59] introduced *Coder-Reviewer reranking*, a novel technique that combines code generation (Coder) with evaluation (Reviewer) to address the biases of standard reranking methods, achieving up to 17% improved accuracy by selecting programs based on mutual consensus between the two models. Zhang et al. [60] presented *Self-Edit*, a novel, fault-aware generate-and-edit approach to enhance code generation by leveraging a neural code editor that iteratively refines code based on execution feedback, achieving significant accuracy improvements (e.g., 89% on *APPS*-dev) over standard LM outputs for competitive programming tasks. Yen et al. [61] introduced a novel system, *CoLadder*, which aids programmers by decomposing programming tasks hierarchically to improve intent externalization, task structuring, and code generation across abstraction levels, ultimately enhancing controllability and reducing cognitive load during prompt-driven coding. Zheng et al. [62] proposed *ChainCoder*, a hierarchical, syntactically-guided, coarse-to-fine code generation model, which

enhances Python program synthesis by progressively constructing code in multiple passes, first outlining, then detailing, demonstrating improved performance and syntactic accuracy over single-pass models. Cloos et al. [63] introduced a scalable approach using LLMs to generate and validate code for agents and environments that simulate diverse personality profiles based on the *HEXACO* model, demonstrating the model's ability to capture personality-consistent behaviours and highlighting contextual prompt dependencies for accurate personality recovery. Salim et al. [64] presented a novel method of adversarial perturbations to reduce LLM effectiveness in solving programming assignments, finding that strategically modifying assignment prompts decreases LLM-generated solution accuracy by 77%, thus impeding LLM-assisted cheating in introductory programming courses. Ugare et al. [65] introduced a novel framework called *ITERGEN*, which enhances structured generation by enabling iterative, grammar-guided generation that can move forward and backward to refine outputs, significantly improving *SQL* query accuracy and privacy safeguards by reducing sensitive information leakage. Wu et al. [66] presented *AutoAPIEval*, an automated framework to evaluate LLMs for API-oriented code generation, focusing on API recommendation and code example generation with metrics for accuracy, and found that model performance varies, with factors like API popularity and model confidence strongly influencing code quality. Li et al. [67] proposed *RethinkMCTS*, a novel Monte Carlo Tree Search framework for code generation by integrating fine-grained verbal feedback from code execution to refine reasoning errors, leading to superior code quality and improved pass rates on benchmarks, notably enhancing *GPT-3.5* and *GPT-4o-mini* performance. Rasheed et al. [68] introduced *CodePori*, a novel multi-agent system for autonomously generating code for large projects, achieving 89% accuracy on *HumanEval* benchmarks and 85% accuracy in manual tests, demonstrating efficiency and reliability in complex software development tasks through collaborative, task-specific LLM-based agents. Xue et al. [69] presented a novel multi-programming language ensemble framework (*MPLE*) for code generation using LLMs, leveraging the strengths of multiple languages to mitigate language-specific biases, achieving up to 17.92% performance improvement and state-of-the-art results on benchmarks like *HumanEval*. Xie et al. [70] presented *COrAL*, a novel LM leveraging "Context-Wise Order-Agnostic Language Modelling" that improves LLM performance and efficiency through iterative, order-agnostic token refinement in sliding blocks, achieving notable accuracy gains and up to 3.9x speedups by enabling parallel forward and backward dependency modelling. Kabir et al. [71] introduced a novel zero-shot approach, *ZS4C*, for generating compilable code from incomplete snippets using LLMs by inferring import statements and iteratively fixing errors through

LLM-validator interactions, achieving a 95.1% compilation rate on the StatType-SO benchmark, a significant improvement over prior methods. Wang et al. [72] introduced *PLANSEARCH*, a novel algorithm that enhances LLMs' code generation by using diverse NL plans instead of directly sampling code, achieving higher success rates across benchmarks by increasing solution diversity compared to conventional search methods. Zhe et al. [73] presented "Language-Oriented Code Sketching", a novel method that provides real-time feedback during prompt crafting by generating intermediate code sketches from NL, improving user guidance and interaction with LLMs for code generation. Quoc et al. [74] proposed a novel self-correcting code generation framework called *CoT-SelfEvolve*, which iteratively refines data science code using chain-of-thought prompting and external knowledge from *StackOverflow*, resulting in significantly higher accuracy and efficiency in solving complex problems compared to existing models. Zheng et al. [75] introduced a "self-infilling" code generation framework that integrates infilling capabilities into autoregressive decoding, enabling non-linear generation with novel interruption and looping mechanisms that improve code consistency and regularity across benchmarks by dynamically deferring and refining specific code segments. Nunez et al. [76] proposed *AutoSafeCoder*, a multi-agent framework that enhances secure code generation by combining static analysis and fuzz testing through iterative feedback from three agents (coding, static analysis, and fuzzing), demonstrating a 13% reduction in vulnerabilities in generated code compared to baseline models, with minimal impact on functionality. Wang et al. [77] introduced LLM-Sym, an LLM-driven symbolic execution engine for Python that translates complex path constraints involving lists into *Z3 SMT* solver code, demonstrating substantial improvements in symbolic execution for dynamically-typed languages and achieving a 63% success rate on complex test cases compared to traditional symbolic methods. Zhu et al. [78] introduced Adaptive Temperature ($AdapT$) sampling, a dynamic approach for temperature adjustment in token sampling tailored for code generation with LLMs, which improves diversity for challenging tokens and stability for confident tokens, resulting in enhanced accuracy on code generation tasks. Zhang et al. [38] proposed *CODEAGENT*, an LLM-based framework with integrated external tools to enhance repository-level code generation, achieving significant improvements in accuracy and efficiency over simpler generation models and commercial tools like GitHub Copilot. Mathews et al. [79] introduced a novel application of Test-Driven Development (TDD) principles in LLM-based code generation, showing that providing tests to LLMs improves code accuracy and robustness, with experimental results demonstrating significant performance gains across various datasets when tests guide generation and correction processes.

2.1.3 Evaluation: Novel Evaluation Metrics for Code

Yang et al. [80] demonstrated that replacing human-generated comments in code datasets with *ChatGPT*-generated ones significantly enhances code intelligence models, like *CodeT5*, particularly in tasks such as code summarization, generation, and translation by ensuring higher semantic consistency between comments and code. Zhong et al. [81] presented *ROBUSTAPI*, a benchmark for assessing the reliability and robustness of LLMs in code generation, particularly highlighting API misuse issues, which are prevalent across models like *GPT-4* (62% misuse rate) and others, posing significant risks in real-world applications. Pudari et al. [82] explored the limitations of AI-supported code completion tools like *GitHub*'s *Copilot*, revealing that while they perform well in basic code syntax and structure, they struggle to follow language idioms and avoid code smells, and suggest a taxonomy that categorizes AI-assisted development from basic syntax to higher-level design abstraction, highlighting future challenges and research directions needed for AI tools to support complex software engineering tasks fully. Zhou et al. [83] presented *CodeBERTScore*, a novel evaluation metric for code generation that incorporates both generated code and NL context, showing higher alignment with human preference and functional accuracy across languages compared to prior metrics. Yeticstiren et al. [84] assessed the code quality of AI-assisted generation tools *GitHub Copilot*, Amazon *CodeWhisperer*, and *ChatGPT*, finding *ChatGPT* the most accurate (65.2% correctness), while Amazon *CodeWhisperer* had the lowest correctness but minimal technical debt; the study emphasizes varying tool strengths, particularly as prompts and function names impact output quality. Sun et al. [85] presented a novel framework leveraging LLMs to enhance software design and developer experience, focusing on CI/CD optimization, real-time feedback, and actionable insights through log analysis for improved productivity and workflow efficiency.

2.2 Test Generation

2.2.1 Input: Domain/Task-Specific Data

Tufano et al. [86] proposed a fine-tuned *BART Transformer* model that significantly improves the accuracy of generating assert statements for unit tests, achieving 62% top-1 accuracy, an 80% improvement over prior *RNN*-based methods. Zhang et al. [87] investigated *ChatGPT*'s effectiveness in generating security tests, finding that it outperforms other tools by leveraging prompt templates with rich vulnerability context, but it remains sensitive to prompt quality and lacks reliability without clear

exemplars. Lemieux et al. [88] presented *CODAMOSA*, a hybrid approach integrating LLMs like Codex with search-based software testing (SBST) to escape test generation coverage plateaus, achieving significantly higher test coverage by prompting LLMs at strategic stalls in SBST. Schafer et al. [89] presented an empirical study of *TESTPILOT*, a tool using LLMs for automated JavaScript unit test generation for npm, achieving higher coverage than traditional methods through prompt engineering with iterative adjustments. Zhong et al. [90] introduced *JSONTESTGEN*, a novel approach leveraging LLMs to automatically generate diverse and mutation-guided unit tests for the JSON library *Fastjson2*, successfully uncovering 34 previously unidentified bugs, including challenging non-crashing ones. Plein et al. [91] evaluated the feasibility of using LLMs, specifically *ChatGPT* and *CodeGPT*, for generating executable test cases from informal bug reports, finding that while both models show promise, *CodeGPT* performs better in single attempts, while *ChatGPT* excels with multiple attempts. Kirinuki et al. [92] compared black-box test cases generated by *ChatGPT* and human testers, finding that *ChatGPT* can create effective test cases with similar or slightly superior coverage, and highlights that human-*ChatGPT* collaboration enhances testing coverage more than human-only pairs. Mundler et al. [93] presented *SWT-BENCH*, a Python benchmark dataset for LLM-driven software test generation, showing Code Agents outperform traditional test generation methods by adapting code repair techniques for unit testing. Li et al. [94] introduced *DLLENS*, a novel differential testing approach that leverages LLMs to generate diverse, valid inputs and synthesize counterparts for APIs in different deep learning libraries, enabling the detection of functional bugs more effectively than prior methods. Karanjai et al. [95] proposed a novel approach using LLMs for automated unit test generation tailored for high-performance computing (HPC) applications, specifically parallel C++ programs with *OpenMP/MPI*, showing that LLMs can effectively create syntactically correct test cases with substantial coverage but some limitations in unique test design. Cui et al. [96] evaluated nine LLMs for generating text inputs to aid in automated GUI testing on Android applications, finding that specific LLMs like *GPT-4* produce high-quality, contextually relevant inputs that improve testing efficiency and can identify bugs Li et al. [97] compared the effectiveness of 11 LLMs in generating automated web-form tests and found that contextual prompt designs, particularly parser-processed HTML, significantly improve test accuracy, with models like *GPT-4*, *GLM-4*, and *Baichuan2* performing best in submission success rates (SSR) Xue et al. [98] introduced *LLM4Fin*, a fully automated framework for generating high-coverage test cases from unstructured business rules in FinTech software acceptance testing, leveraging fine-tuned LLMs combined with algorithmic methods to outperform state-of-the-art models and human testers in scenario and code coverage. Nabeel et al.

[99] proposed a two-stage generative model for telecom software testing, innovatively combining time-series data generation and LM-based code synthesis, demonstrating effectiveness in generating privacy-preserving, scenario-comprehensive test scripts. Arora et al.[100] introduced a novel approach, *RAGTAG*, which combines Retrieval-Augmented Generation (RAG) with LLMs to automate test scenario generation from bilingual NL requirements, achieving notable relevance and feasibility in industrial applications, though domain-specific nuances remain challenging.

2.2.2 Model: Novel Framework for Test Generation

Tufano et al. [101] presented *ATHENATEST*, a novel approach to automated unit test generation leveraging a sequence-to-sequence *Transformer* model trained on a unique, large parallel corpus (*METHODS2TEST*) of Java methods and developer-written tests. Key findings include *ATHENATEST*'s capability to produce syntactically correct, human-readable test cases with comparable coverage to *EvoSuite*, outperforming *GPT-3*, and showing developer preference due to its readability and effectiveness in testing. Chen et al. [102] introduced *CODET*, a novel method that uses pre-trained LMs to automatically generate test cases for code samples, allowing for effective solution selection through a dual execution agreement, significantly improving code generation accuracy on various benchmarks. Li et al. [103] evaluated the impact of prompt design on unit test generation by Code Interpreter for *Quixbugs* functions, finding that prompt detail minimally affects output quality, though certain prompts improve formatting reliability. Steenhoek et al. [104] introduced Reinforcement Learning from Static Quality Metrics (*RLSQM*) to improve LLMs for automated unit test generation, optimizing models to reduce test smells and enhance quality based on static quality metrics instead of human feedback, resulting in tests with up to 21.4% more assertions and up to 2.4% fewer test smells. Zhang et al. [105] proposed *ALGO*, a novel framework that synthesizes algorithmic programs using LLM-generated oracles as verifiers, showing improved accuracy in code generation by guiding candidate programs and refining their correctness through generated test cases. Chen et al. [106] proposed *ChatUniTest*, an automated unit test generation framework using LLMs, featuring an adaptive focal context mechanism and a generation-validation-repair cycle to enhance test accuracy, achieving superior code coverage compared to other tools like *TestSpark* and *EvoSuite*. Nashid et al. [107] introduced *CEDAR*, a novel retrieval-based prompt selection technique that improves few-shot learning in code-related tasks by retrieving relevant code demonstrations, achieving notable accuracy gains in test assertion generation and program repair over state-of-the-art models. Xiong et al. [108] proposed a novel approach to evaluating and enhancing

the testing capabilities of LLMs for code, revealing that self-generated test cases significantly improve program synthesis accuracy and code pass rates over traditional baselines. Rao et al. [109] introduced *CAT-LM*, a *GPT*-style LM uniquely trained to understand aligned code and test files, enhancing test generation by leveraging a novel pre-training approach with an extended context length (8,192 tokens), resulting in higher test coverage and quality, outperforming larger models like *CodeGen* and *StarCoder*. Lops et al. [110] presented *AGONETEST*, a novel automated system for generating and assessing complex Java unit test suites with LLMs, emphasizing class-level testing and incorporating comprehensive metrics for quality evaluation. Dakhel et al. [18] proposed *MuTAP*, a novel framework for test generation, that enhances the bug-detection capability of LLM-generated test cases by augmenting prompts with surviving mutants, achieving significantly higher mutation scores (up to 93.57%) compared to existing automated tools. Alagarsamy et al. [19] introduced *A3Test*, a novel DL-based test case generation approach, enhanced with assertion knowledge and test signature verification, achieving more accurate and readable test cases compared to *AthenaTest* and *ChatUniTest*. Ni et al. [111] presented *CasModaTest*, a novel, cascaded, model-agnostic unit test generation framework that enhances test generation by splitting the task into prefix and oracle generation phases, demonstrating improved accuracy and coverage compared to existing methods. Ryan et al. [112] introduced *SymPrompt*, a code-aware prompting approach for LLM-driven test generation that significantly improves test coverage and correctness by decomposing the process into prompts based on execution path constraints and contextual information. Pizzorno et al.[113] introduced *CoverUp*, a novel coverage-guided approach for LLM-driven test generation, iteratively refining prompts based on coverage metrics to improve line and branch coverage over previous methods significantly. Yang et al. [114] introduced *TELPA*, a novel program-analysis-enhanced prompting technique to improve LLM-based test generation for hard-to-cover branches by focusing prompts on relevant code segments, resulting in significant improvements in branch and line coverage over existing methods. Yuan et al. [115] evaluated *ChatGPT*'s ability to generate unit tests, highlighting its issues with correctness while proposing the novel ChatTester approach that enhances the accuracy of *ChatGPT*-generated tests through iterative refinement and intention-based prompts, achieving improved correctness and usability. He et al. [116] introduced *FuzzAug*, a novel data augmentation technique using fuzzing to enhance neural test generation, achieving increased assertion accuracy, compilation rate, and branch coverage by generating diverse and valid inputs. Wang et al. [117] proposed *HITS*, a novel approach for high-coverage unit test generation by using method slicing to allow LLMs to generate tests for complex Java methods slice by slice, achieving higher line and branch

coverage compared to other methods. Li et al. [118] evaluated the effectiveness of LLMs in generating high-quality test cases and introduced a novel *TestChain* framework that improves accuracy by separating input and output generation with a multi-agent system, achieving a notable performance increase with *GPT-4*. Jiri et al. [119] introduced *UTGen*, a novel approach combining search-based software testing (SBST) and prompt-engineered LLMs to enhance unit test understandability through refined test data, comments, and descriptive names, significantly aiding developer bug-fixing efficiency. Liu et al. [120] presented *AID*, a novel approach combining LLMs with differential testing to generate accurate defect-identifying test cases for tricky bugs in programs that pass existing tests, achieving superior recall, precision, and F1-score over prior methods. Etemadi et al. [121] proposed *Mokav*, an execution-driven differential testing tool that uses LLMs iteratively to generate difference-exposing tests (DETs) with feedback, significantly outperforming previous methods in functional difference detection between code versions. Karmarkar et al. [122] introduced *TestRefineGen*, an LLM-based technique for generating and refining test data in confidentiality-constrained settings, highlighting its counter-example guided refinement process and improvement in generating valid test data. Xiao et al. [123] presented an empirical investigation of LLMs integrated into search-based unit test generation (SBTG), finding that LLMs can enhance code coverage at different stages, especially during the initial population generation, when frequency and timing of intervention are carefully managed. Gu et al. [124] introduced *TestART*, a novel approach combining automated unit test generation and repair iterations for LLM-based unit test cases, achieving high pass and coverage rates by leveraging fixed repair templates and prompt injection to address model hallucination and repetition issues.

2.2.3 Evaluation: Novel Evaluation Metrics for Tests

Guilherme et al. [125] investigated *ChatGPT*'s ability to generate unit tests for Java code automatically, finding that while its generated tests achieve comparable code coverage to traditional tools, it falls short on mutation scores; the authors identify prompt-based adjustments to enhance test effectiveness. Siddiq et al. [126] empirically evaluated the performance of LLMs in generating JUnit tests, highlighting the challenges of compilation errors, low coverage, and prevalent test smells, with Codex and *StarCoder* outperforming *GPT-3.5-Turbo* due to code-specific fine-tuning. Vikram et al. [127] investigated using LLMs like *GPT-4* to generate property-based tests (PBTs) from API documentation with a focus on new metrics for validity, soundness, and a novel property coverage metric that measures the effectiveness of generated tests in detecting violations. Zhou et al. [128] introduced a novel Context

Consistency Criterion (*C3*) tool to measure and improve the readability of unit test inputs by aligning them with source code contexts, showing that their enhanced *EvoSuiteC3* significantly outperforms traditional tools in generating readable tests. Ouedraogo et al. [129] investigated the effectiveness of various LLMs for generating unit tests, highlighting that prompt engineering significantly impacts test quality, with the best prompts achieving high code readability and reducing test smells, but LLM-generated tests still lag behind traditional methods in test completeness. Bhatia et al. [130] compared *ChatGPT* and *Pynguin* for Python unit test generation, finding that *ChatGPT* can achieve comparable or better coverage but with a higher error rate in assertions, suggesting a combined approach might yield optimal results. Yang et al. [131] presented a comprehensive evaluation of open-source LLMs for unit test generation, highlighting the critical role of prompt design, challenges of adapting general methods like in-context learning, and underperformance in defect detection compared to traditional approaches. Tang et al. [132] presented a systematic comparison between *ChatGPT* and *EvoSuite* for unit test suite generation, revealing *ChatGPT*'s effectiveness in producing readable and contextually relevant test cases with fewer bugs but lower code coverage compared to *EvoSuite*'s optimization-driven approach. Huang et al. [133] examined the impact of incorrect source code on the effectiveness of test case generation by LLMs, showing that incorrect code significantly misguides LLMs, reducing test accuracy, coverage, and bug detection rates. Ouedraogo et al. [134] analyzed test smells in unit tests generated by LLMs, finding that these LLM-generated tests often contain flaws like Magic Number Test and Assertion Roulette, influenced by model size, context length, and prompt techniques, highlighting a need for refined test generation practices to reduce maintainability issues.

2.3 Other Automated Software Engineering Tasks

In this subsection, we also introduce other software engineering tasks that are tackled using fine-tuned and large language models.

2.3.1 Code Summarization

The code summarization task aims to generate a summarization or documentation of the input source code. Sun et al. [135] was the first to systematically evaluate *ChatGPT*'s capabilities in automatic code summarization, comparing its performance against state-of-the-art models (*NCS*, *CodeBERT*, *CodeT5*) and identifying unique challenges and opportunities specific to *ChatGPT* in this domain. Sun et al. [136]

introduced *PromptCS*, a novel non-invasive prompt learning framework for source code summarization that trains a prompt agent to generate continuous prompts, significantly reducing the need for resource-intensive fine-tuning or manually crafted prompts, and achieving superior performance and training efficiency compared to existing approaches. Saberi et al. [137] introduced *AdvFusion*, a new Parameter Efficient Fine-Tuning (PEFT) architecture that enhances multilingual knowledge transfer for code summarization by enforcing the model to first learn from other programming languages before focusing on the target language, achieving superior performance with significantly fewer trainable parameters and reduced training time compared to full fine-tuning and existing adapter-based methods. Rukmono et al. [138] proposed a novel approach to automate high-level software component summarization by combining hierarchical chain-of-thought prompting with static code analysis, leveraging LLMs to provide bottom-up, inductively reasoned summaries from granular operations to cohesive software components. Wang et al. [139] introduced *ZEROVAR*, a novel approach leveraging code pre-trained models and zero-shot prompt learning to generate concise NL explanations for variables in code, addressing challenges in program comprehension and enhancing tasks like variable naming, abbreviation expansion, and spelling correction. Shi et al. [140] proposed *SoTaNa*, an open-source software development assistant that uses *ChatGPT* to generate high-quality, instruction-based data tailored for software engineering tasks and employs a PEFT approach to enhance the open-source *LLaMA* model, enabling it to perform effectively on developer-specific tasks like answering Stack Overflow questions, code summarization, and generation, all while being computationally accessible. Zhao et al. [141] proposed *SCCLLM*, a novel approach leveraging LLMs and in-context learning to generate high-quality comments for smart contract code by selecting customized demonstrations from historical corpora and outperforming state-of-the-art methods in both automatic and human evaluations. Wang et al. [142] presented *SparseCoder*, an identifier-aware sparse *Transformer* model designed for efficient file-level code summarization, leveraging structured attention mechanisms (local, global, and identifier-specific) to handle long code sequences effectively and demonstrating superior performance across multiple code-related tasks. Wang et al. [143] introduced *SlimCode*, a model-agnostic approach to simplify input code for pre-trained LLMs, significantly improving computational efficiency and reducing costs while maintaining or enhancing task performance in code search and summarization. Virk et al. [144] explored the calibration for code summarization by LLMs, introducing a method to provide well-calibrated confidence scores for LLM-generated summaries to reliably predict their similarity to human-produced summaries, leveraging *SentenceBERT* metrics and rescaling techniques like *Platt scaling*. Szalontai et al. [145] investigated

the novel capability of open-source LLMs for source code summarization and explanation, utilizing *HumanEvalExplain* as a benchmark while evaluating models like *CodeLlama* and *MagiCoder* for their performance on code-to-text tasks, which are less researched compared to code generation. Su et al. [146] proposed a novel approach to code summarization that incorporates context to explain why a Java method exists within the broader program, using a small, fine-tuned LM trained through knowledge distillation and guided by human-written exemplary summaries, achieving superior performance over *GPT-4* and other baselines in generating context-aware summaries. Su et al. [147] presented a semantic similarity metric as a novel loss function for neural source code summarization, which evaluates the predicted summary holistically and provides partial credit for synonymous predictions, improving upon the traditional categorical cross-entropy loss. Shi et al. [148] introduced NL outlines as a new form of AI-assisted code explanation that combines summarization and partitioning of code into logical sections using concise prose, enabling bidirectional synchronization between code and NL, and demonstrating its utility across a range of software development tasks, such as code maintenance, navigation, review, and malware detection, powered by advanced LLMs. Shen et al. [149] presented *Bash2Com*, a novel approach for Bash code comment generation. Its key innovations are the introduction of the NP-GD module for gradient-based adversarial data augmentation to address data scarcity and the MASA component to enhance semantic understanding by leveraging and fusing multi-layer representations in *CodeBERT* with *LSTM* and attention mechanisms, outperforming state-of-the-art methods in the domain. Poudel et al. [150] presented *DocuMint*, a novel supervised fine-tuning dataset of 100,000 samples and an evaluation framework specifically designed for assessing and enhancing the quality of Python docstrings generated by Small Language Models (SLMs), with a focus on metrics such as accuracy, conciseness, and clarity, making it the first comprehensive resource for fine-tuning and benchmarking SLMs in this domain. Pordanesh et al. [151] introduced a novel investigation into the efficacy of *GPT-4* in binary reverse engineering tasks, particularly focusing on decompiled code analysis and malware interpretation, presenting unique experimental methodologies to evaluate the model's capabilities in technical code explanations, renaming variables, and security analysis, highlighting both its potential and critical limitations for future advancements.

2.3.2 Code Translation

The task of code translation takes a source code of a certain programming language to generate functionality-wise equivalent code in a different programming language. Zhu et al. [152] proposed *CoST* (Code Snippet Translation), a novel multilingual dataset

offering snippet-level parallel data across seven programming languages, coupled with a unique multilingual snippet training approach (*MuST-PT*) that significantly enhances program translation performance, particularly for low-resource languages, achieving state-of-the-art results on the *CodeXGLUE* benchmark. Jiao et al. [153] introduced a novel taxonomy and benchmark, *G-TransEval*, for evaluating neural code translation models by categorizing translation tasks into four complexity levels (token, syntactic, library, and algorithm), addressing the limitations of existing benchmarks that overly favour simple tasks and providing a more rigorous, comprehensive evaluation framework. Wang et al. [154] proposed *TransMap*, the first automated approach to pinpointing semantic mistakes in neural code translation, achieving high accuracy and reducing debugging time significantly by leveraging a novel source map generation and dynamic trace comparison for Python-to-JavaScript translation. Tang et al. [155] did a systematic exploration of using self-generated NL explanations as an intermediate step in code-to-code translation tasks, demonstrating significant performance improvements, particularly in zero-shot scenarios, and providing a comprehensive evaluation across diverse programming languages and explanation types. Qi et al. [156] introduced Syntactic Unit Tests (*SUT*), a novel fine-grained evaluation framework for transcompiler models that identifies and addresses syntactic deficiencies by providing interpretable metrics and a targeted test suite, which is more aligned with human judgment than existing evaluation methods like *BLEU* and *CodeBLEU*. Huang et al. [157] proposed a novel approach to program translation called Code Distillation (*CoDist*), which leverages a distilled intermediate representation of code to serve as a translation pivot, enabling high-quality multilingual program translation without relying on parallel corpora and achieving state-of-the-art performance on key benchmarks. Yang et al. [158] presented *UniTrans*, a novel unified framework for automated code translation, leveraging LLMs like *GPT-3.5* and *LLaMA*. Its novelty lies in systematically integrating test case generation, translation augmentation, and iterative repair to significantly enhance translation accuracy across multiple programming language pairs, overcoming key limitations of existing learning-based transpilers and demonstrating generality across diverse LLMs. Xue et al. [159] introduced *kNN-ECD* and *kNN-ECSm*, novel error correction methods leveraging kNN search combined with datastores to significantly enhance code-to-code translation accuracy while providing interpretable decision-making. This approach avoids the need for resource-intensive retraining of *Transformer*-based models and achieves substantial improvements in accuracy, addressing key limitations of existing methods. Pan et al. [160] presented a comprehensive empirical study investigating the performance of LLMs in code translation, highlighting their limitations, categorizing translation bugs into a novel taxonomy, and proposing an iterative prompt-crafting approach to

mitigate these errors, marking a pioneering effort in the scale and depth of analyzing LLM-induced bugs in code translation. Li et al. [161] introduced *FSCTrans*, a novel PEFT framework for few-shot code translation, leveraging task-adapted prompt learning that bridges pre-trained *CodeT5* models with translation tasks using a new code-to-code generation objective, achieving substantial performance improvements over existing methods with minimal data and trainable parameters. Luo et al. [162] presented *TransGraph*, a novel method for LLM-based code translation that leverages call graphs and bridged debuggers to significantly reduce compilation and runtime errors, enabling more accurate and practical code translation for complex, multi-file software projects. Kumar et al. [163] introduced a novel automated validation framework for verifying the semantic equivalence of COBOL-to-Java code transformations, uniquely utilizing symbolic execution-based test generation and resource mocking to address challenges in handling enterprise-grade external dependencies, thereby ensuring reliable and scalable testing across diverse execution paths. Liu et al. [164] proposed a novel *LangChain*-based converter for translating concurrent Java programs to *ArkTS*, addressing the challenge of transitioning between Java's shared memory model and *ArkTS*'s actor-based model by employing a custom concurrency library, SharedArrayBuffer, and utilizing LLMs with Chain-of-Thought methodology to achieve efficient and accurate translation of complex concurrency constructs.

2.3.3 Automatic Program Repair

The task of Automatic Program Repair (APR)generates a fixed code (patch) when a piece of buggy code is given. Yuan et al. [165] introduced *CIRCLE*, a novel framework for APR that uniquely combines continual learning with multi-language support. It achieves state-of-the-art performance by using a *T5*-based model, a prompt-based input representation to leverage pre-trained knowledge, and innovative mechanisms like difficulty-based example replay and elastic weight consolidation to prevent catastrophic forgetting, making it the first system capable of handling bug fixes across multiple programming languages in a continual learning scenario. Zhang et al. [166] presented *Gmerge*, a novel tool leveraging k-shot learning with large pre-trained LMs (e.g., *GPT-3*) to resolve both textual and semantic merge conflicts in software development, showcasing state-of-the-art performance in semantic conflict resolution and emphasizing simplicity by avoiding domain-specific language fine-tuning. Zhang et al. [167] proposed *GAMMA*, a novel template-based APR tool that leverages pre-trained LMs, such as *UniXcoder*, to predict donor code directly through a mask prediction task instead of relying on local code retrieval, significantly enhancing repair performance and addressing the dataset overfitting

challenge common in APR. Xia et al. [168] presented *FitRepair*, a novel approach to APR that automates and enhances the "plastic surgery hypothesis" using LLMs. It integrates fine-tuning strategies and prompting mechanisms, significantly outperforming existing methods by leveraging project-specific code patterns for repairing bugs, showcasing state-of-the-art performance on benchmarks like *Defects4j*. Xia et al. [169] did a comprehensive evaluation of state-of-the-art large pre-trained LMs for APR, revealing that these models significantly outperform existing APR tools across multiple datasets and programming languages, and demonstrating the importance of suffix code context for generating effective fixes. Fan et al. [170] presented the novel concept of applying APR techniques to enhance the reliability of code generated by LLMs like *Codex*, revealing that leveraging APR tools and the *Codex* edit mode can fix common programming errors in auto-generated solutions, which align with mistakes typically made by human programmers. Wu et al. [171] introduced *ConDefects*, a novel dataset specifically curated to address data leakage issues in evaluating LLMs for fault localization and program repair by providing real-world faulty programs from the *AtCoder* platform with precise fault locations and corrected versions, along with features for time and difficulty-based task selection. Zhao et al. [172] presented *RePair*, a novel APR framework that employs process-based feedback, which integrates iterative repair steps supervised by reinforcement learning to achieve enhanced program robustness. The work is notable for its creation of the *CodeNet4Repair* dataset, a multi-step program repair dataset, and for demonstrating that small-scale LMs can rival large commercial models in performance through process-based methods rather than traditional outcome-based supervision. Zhang et al. [173] introduced *PyDex*, a system leveraging LLMs to repair both syntactic and semantic errors in students' introductory Python programming assignments by combining multimodal prompts, iterative querying, test-case-based few-shot learning, and program chunking, surpassing traditional APR methods in efficiency and accuracy. Yin et al. [174] proposed *ThinkRepair*, a self-directed APR framework that leverages LLMs with Chain-of-Thought reasoning and interaction feedback, introducing a novel two-phase approach, collection and fixing phases, to automatically generate and apply reasoning chains for bug fixes, significantly outperforming state-of-the-art methods on key datasets. Yang et al. [175] presented *CREF*, a novel semi-automatic conversational program repair framework leveraging LLMs like *GPT-4*, which uniquely integrates augmented information (e.g., tutor guidance, solution descriptions, and failing test cases) to assist programming tutors in debugging code. It achieves a significant performance boost in repair accuracy and reduces the computational overhead associated with lengthy prompts through conversational interaction, marking a key advancement in LLM-based program repair for educational settings. Xin et al.

[176] introduced *IBugFinder*, a novel enumeration-based approach for identifying and isolating indivisible multi-hunk bugs, creating an enhanced dataset (*CatenaD4J*), and providing comprehensive insights into multi-hunk patch relationships to guide the development of more effective APR techniques. Hossain et al. [177] presented a novel framework called *Toggle* for automated bug localization and repair, which utilizes token-granularity instead of the traditional line-granularity to enhance precision, incorporates a unique adjustment module to address tokenizer inconsistencies, and achieves state-of-the-art performance by optimizing prompt design and leveraging contextual information to separate bug localization and fixing tasks effectively. Li et al. [16] curated the APR-Instruction dataset and explored PEFT methods for APR, demonstrating that PEFT significantly reduces computational resource requirements while achieving up to 58% more fixed bugs compared to state-of-the-art techniques, particularly through the use of the novel IA^3 method for enhancing creativity and fixing capabilities of LLMs. Ouyang et al. [178] introduced a novel large-scale mutation-based benchmark (*MuBench*) and a multidimensional evaluation approach that includes new metrics, such as *SYE* and *TCE*, providing a more comprehensive analysis of APR techniques across real-world and artificial bugs.

Chapter 3

Fine-Tuned Models on Machine Learning Tasks

3.1 Introduction

Due to recent advancements in deep learning techniques [2, 3], many neural code generation models have been proposed and extensively studied [179, 180, 181, 182, 183]. Most of these models treat code generation as a neural machine translation task and are built in an end-to-end manner, i.e., a parallel corpus of source code and comment pairs is used to train the models, and given a natural language (NL) described programming task, the models will generate a source code snippet. Although many neural code generation models have achieved high accuracy in generating source code from NL-described programming tasks, most of which are mainly evaluated on general programming tasks, little is known about their effectiveness and usefulness for domain-specific tasks. In this study, we investigate Machine Learning (ML) programs, which have significantly different paradigms from traditional general programming tasks (relatively more deterministic and less statistically orientated) [184]. In addition, ML programming tasks often require many complex algorithms, mathematical operations, and data processing operations [185, 186].

To investigate the effectiveness of existing neural code generation models on ML programming tasks, we use six state-of-the-art neural code generation models as the baselines (details are in Section 3.3.2), and we select four widely used ML libraries,

i.e., Scikit-Learn[1], Keras[2], TensorFlow[3], and PyTorch[4] for ML programming task collection. Inspired by existing studies [186], we categorize the ML programming tasks according to the utilized APIs into three different categories based on their purposes in the ML pipeline, i.e., *data process*, *model building*, and *evaluation* (details are in Section 3.2.2).

For our analysis, we need a new dataset of ML programming tasks inclusively as most of the existing widely used code generation benchmark datasets, e.g., StaQC [187], CoNaLa [188], Django [189], and ReCa [190] mainly contain general-purpose programming tasks. Due to this constraint, we create a new ML task-related dataset by reusing the data from JuICe [191], i.e., a refined human-curated code generation dataset including 1.5 million examples mined from over 659K publicly available Jupyter notebooks from GitHub. We use the API information from the four studied ML libraries to extract ML-related programming tasks. Note that we also collected a non-ML task-related dataset from JuICe [191] as a comparison dataset. We then evaluate the six selected state-of-the-art neural code generation models on the two datasets (ML vs non-ML) regarding the accuracy, syntactic and semantic correctness, and usefulness of the generated code. Our empirical study reveals some good, bad, and missing aspects of state-of-the-art neural code generation models on ML tasks, with a few major ones listed below. (**Good**) Neural code generation models perform significantly better on ML tasks than on non-ML tasks, with an average difference of 10.6 points in BLEU-4 scores. (**Good**) The generated code can help developers write correct code by providing developers with clues for using the correct APIs found in user case studies from 12 graduate students. (**Bad**) More than 80% of the generated code is semantically incorrect. (**Bad**) Code generation models do not have significance in improving developers' completion time. (**Missing**) The observation from our user study reveals the missing aspects of code generation for ML tasks, e.g., decomposing code generation for divide-and-conquer into API sequence identification and API usage generation.

This paper makes the following contributions:

- The first empirical study for evaluating six state-of-the-art neural code generation models on ML programming tasks regarding the accuracy, syntactic and semantic correctness, and usefulness of the generated code.

[1] `https://scikit-learn.org/stable/`
[2] `https://keras.io/`
[3] `https://www.tensorflow.org/`
[4] `https://pytorch.org/`

- The missing viewpoints between existing automatic neural code generation models and practical ML programming tasks, as well as future research directions for improving code generation.
- A new dataset that is composed of 83K pairs of NL descriptions and the corresponding source code that are implemented by four ML libraries, facilitates the follow-up studies in this direction.
- The released source code of our tool and the dataset of our experiments to help other researchers replicate and extend our study[5].

The rest of this paper is organized as follows. Section 3.2 presents the background of this study. Section 3.3 describes the methodology and the study design of our work. Section 3.4 presents the evaluation results. Section 3.5 discusses open questions and the threats to the validity of this work. Section 3.6 presents the related studies. Section 3.7 concludes the study.

3.2 Background

This section introduces the background of this study, i.e., the neural source code generation model and ML programming tasks.

3.2.1 Neural Code Generation

Neural code generation techniques exploit deep neural generation models to learn the distribution or the density estimation of source code with different representations, i.e. the sequence of tokens or tree structures, e.g. abstract syntax tree (AST) [192, 193]. Due to the similarity of the data representation, source code generation models inherit many of the techniques from NL generation models. In NL generation, many tasks can be categorized, e.g. content determination, text structuring, sentence aggregation, lexicalization, referring expression generation, and linguistic realization [194]. There are also different tasks in the source code generation e.g. code completion [195], code search [196], program synthesis via rules or examples [197], domain-specific [198], or general-purpose code generation [199], code-to-code translation [200], API recommendation [201]. The generation model that this paper focuses on is the multi(bi)-modal modelling of source code and NL, which usually uses an end-to-end encoder-decoder architecture of the model that is widely used in neural machine

[5] `https://github.com/shinjh0849/codegen4ml_tasks.git`

Table 3.1: Statistics of existing benchmark datasets. **PL** denotes the program languages, **LOC** denotes the average lines of code, and **Length** denotes the average length of NL tasks in token.

Dataset	**Domain**	**PL**	**#Pairs**	**LOC**	**Length**
Django	general	Python	18.8K	1	14
CoNaLa	general	Python	2.9K	1.1	14
StaQC	general	Python/SQL	2.6k	10.1	10
ReCa	general	C/Java/Python	101K	37.2	185
JuICe	general	Python	1.5M	10.0	39.7

translation. The encoder takes in the sequence of an NL description of a code and maps it to a latent space, and then the decoder decodes it into a source code. The encoder-decoder layers learn to map similar data points of each modality to be closely mapped in the latent space so that they generate similar source codes. Code generation models can generate a code snippet for an NL description that includes usage patterns, correct arguments and values, and the code context of logic or API calls. Code generation is different from API recommendation as the latter fails to suggest the code context and correct usages of APIs with correct fields and arguments [202, 203, 204].

To evaluate code generation models, many different benchmark datasets have been proposed. Django [189] dataset was first mined to generate pseudo-code from source code for the Django framework (Python-to-English). The data contains 18K pairs of Python statements and their corresponding English pseudo-code. CoNaLa [188] is a dataset mined from Stack Overflow posts, which consists of 3K pairs of human-annotated Python code snippets and their NL intent. StaQC [187] is another source code and NL description pair that is mined from Stack Overflow. It consists of 1.4K Python and 1.2K SQL queries, which were mined using a neural network to incorporate textual similarities. ReCa [190] is a large-scale dataset of NL requirements and their programming language. ReCa consists of data for multiple general languages, i.e., C, C++, Java, and Python. It also has multiple programs for the same requirement. JuICe [191] is a new large-scale open-domain dataset composed of 1.5M pairs of Python code examples and the corresponding NL descriptions collected from 659K publicly available Jupyter notebooks on GitHub. Table 3.1 shows the detailed statistics of these existing datasets.

3.2.2 Machine Learning Tasks

Machine learning-based tasks often contain different steps, which are also known as the pipeline of machine learning [205].

Specifically, constructing an ML pipeline contains the following three steps. The first step is mainly regarding the processing of the dataset that is needed to train a model. It involves pre-processing tasks such as data loading, noise filtering, format converting, data imputation, scaling or normalization, and feature engineering (e.g., feature selection, elimination, and word embedding) [206]. After processing the data, one needs to further specify the learning algorithm to infer relations that capture the hidden patterns in the dataset. In the specified algorithm, there is information regarding the parameters that are learned and fit the input data, the loss function to be minimized, the optimizer and the learning rate to lead the model in better learning, and the architecture of the model that specifies the construction of the network's layers and nodes [1]. And finally, one needs to evaluate the trained model. This step includes choosing the evaluation metrics to assess the performance of trained models, e.g., accuracy, precision, and recall [207]. Evaluations are applied to an unseen dataset that is separate from the training data to assess the generalizability of the trained model. To understand the effectiveness of existing neural code generation models on different types of ML programming tasks, we categorize ML programming tasks into different steps in the ML pipeline [186].

- ***Data processing***: tasks that are related to data processing, e.g., data loading, format transformation, normalization, feature engineering, and data transformation.
- ***Model building***: tasks that are related to building the machine/deep learning models, which can be conventional ML models such as classification models, regression models, clustering models, or neural network models with a specific neural architecture regarding the type of layers and activation function used.
- ***Evaluation***: tasks that are related to evaluating the performance of the trained machine/deep learning models, e.g., getting the values of different evaluation metrics such as accuracy, f-measure, and AUC, or getting the probability of instances from the model.

Note that while we focus on three types of ML programming tasks in the ML pipeline (i.e., *data processing*, *model building*, and *evaluation*), there are other tasks in the ML pipeline, e.g., model deployment and performance monitoring [205]. As finishing these tasks mainly requires auxiliary functions or scripts rather than the APIs from

Table 3.2: The studied ML libraries in this work.

Library	**Description**	**#API**
Scikit-learn	A library for ML algorithms.	1.2k
Keras	An interface for artificial neural networks.	1.4k
TensorFlow	A library for DL developed by Google.	8.0k
PyTorch	A library for ML and DL tasks.	0.4k

Table 3.3: Statistics of the experimental datasets. **#Training** is the number of training instances, # Testing is the number of Testing instances, **LOC**, **Length**, and **#API** is the average LOC, length, and number of APIs involved, respectively.

ML task	**#Training**	**#Testing**	**LOC**	**Length**	**#API**
data	30.9K	5.5K	3.93	12.67	1.48
model	26.3K	5.5K	4.18	12.44	1.28
evaluation	13K	2.6K	3.28	11.59	1.13
non-ml	34K	5.5K	3.23	13.26	1.63

the ML libraries, we exclude these tasks in this study. To investigate the generation of ML programming tasks, this work assesses if the current neural code generation models can generate correct and useful code snippets regarding their contexts, values, arguments, and usage patterns for NL descriptions.

3.3 Study Design

In this section, we discuss the design of our empirical study on evaluating the performance of code generation models on ML programming tasks.

3.3.1 Data Collection

In this work, we prepare a new dataset that contains the three types of ML programming tasks (i.e., *data process*, *model building*, and *evaluation*) as most of the existing widely used code generation benchmark datasets, e.g., StaQC [187], CoNaLa [188], Django [189], and ReCa [190] mainly contain general-purpose programming tasks. Specifically, we create our dataset from JuICe [191], i.e., a refined human-curated code generation dataset including 1.5 million Python code examples mined from over 659K publicly available Jupyter notebooks from GitHub. The reason we chose JuICe is that mining paired data of NL description and source code is a challenging task. To mitigate the challenge, we exploit the structure of Jupyter Notebooks as it uses

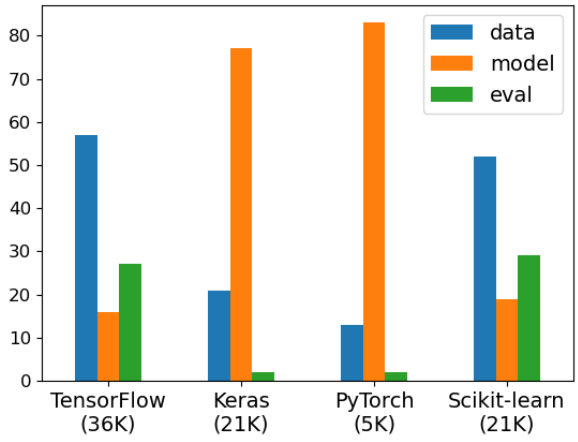


Figure 3.1: Distribution of experiment data in each ML library.

both natural language (markdown cells) and source code (code cells), which regular Python files don't always provide. Compared with other benchmark datasets (e.g., StaQC [187], CoNaLa [188], Django [189], and ReCa [190]), JuICe [191] is relatively large-scale, e.g., its training dataset has 1.5M examples of natural language descriptions and source code pairs. JuICe also has a high-quality dataset for validation and testing from exercise and assignment notebooks for education. Also, they are in Python as most of the ML programming is in the Python language.

ML API selection: The original JuICe dataset [191] contains open-domain programming tasks. For our study, we focus on ML programming tasks that are developed by four widely used ML libraries, i.e., Scikit-learn, Keras, TensorFlow, and PyTorch. The list of studied libraries is organized in Table 3.2. Specifically, three of the authors independently use the following two steps to manually identify all the APIs in each of the four ML libraries that are related to the three types of ML programming tasks (i.e., *data process*, *model building*, and *evaluation*) based on the documentation of these APIs: 1) We read the documentation of the API to understand its functionality. A category will be assigned to the API based on its major intent. 2) If a decision cannot be made based on the documentation of the API, as the description might not contain enough information, we further check the source code following the instructions in Step 1 to label the API. The value of Fleiss' kappa during this process is 0.90, which shows an almost perfect agreement. In total, 128, 244, and 51 APIs are identified for the three types of ML programming tasks, i.e., *data processing*, *model building*, and *evaluation*, respectively. Note that there are a large number of

APIs that cannot be included in the ML pipelines we categorized, e.g., utility APIs, APIs for testing/debugging, internal APIs used by library developers, and APIs for configuration of devices. Due to these reasons, only a small subset of APIs was included in this work.

ML-task data construction: After we identify the APIs in an ML library for each of the three ML programming tasks, we further use the corresponding APIs as the filter to collect ML programming tasks from the JuICe dataset [191], e.g., we scan the source code of each programming task in JuICe, a programming task will be labelled as a *data process* task if it only contains *data process* related APIs from each of the four ML libraries. We discard any instances that use multiple ML libraries to distinguish programming tasks that use different libraries.

Data cleaning: We also remove duplicated data instances, meaningless source code elements, e.g., '>>>', single and multi-line comments, lines that start with '!' and '%' for commands, code snippets with just imports, and non-ASCII code. We also apply additional filtering to further denoise the dataset and make the problem simpler by discarding instances with NL description lengths below 20 and over 150, and source code with more than 10 LOC, as suggested by existing work [208]. To assess the quality of the collected data, the first three authors performed a manual analysis on 100 random samples from each split, i.e. *data processing*, *evaluation*, and *model building*. The authors first check if the source code of the instance is relevant to the ML programming task. Then we check if the NL description is relevant to the source code. We found that 64%-81% of the samples have both source code and NL relevant to the ML programming task and its description. To mitigate the potential information leakage that can be caused by project-specific instances introduced to both the training and testing datasets, we carefully split the instances on a project level, i.e., instances from the same project go to the same split [209, 210, 211].

Non-ML related dataset: To explore the performance difference of neural code generation models on the ML programming tasks and the general programming or non-ML related tasks, we have created another dataset from the same context. For creating the non-ML programming task dataset, we randomly select samples from JuICe [191] with ML programming tasks removed. To make a fair comparison, we set the size of the non-ML task dataset to our biggest ML task dataset, i.e., the *data process* dataset, and we apply the same data cleaning process. To remove potential bias, we repeat the data creation for non-ML tasks 10 times and re-train the models accordingly. We use the average performance of each studied neural code generation model for comparison.

The new dataset of ML programming tasks is organized in Table 3.3. We also show the distribution of the different ML programming tasks in each ML library in

Figure 3.1. In Listing 3.1, we also provide three different examples from *data process*, *model building*, and *evaluation* datasets to show what our paired data of NL description and source code look like. Specifically, the "NL intent" describes the programming tasks in natural language, and the "code snippet" shows the corresponding source code for the programming task.

```
# An example of the Data processing dataset
# NL intent:
"now lets use opencv to read the image"
# code snippet:
img = cv2.imread(TRAIN_PATH+train.Image_id[0])

# An example of a model building dataset
# NL intent:
"train a logistic regression model"
# code snippet:
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)

# Example of the Evaluation dataset
# NL intent:
"evaluate model on test data"
# code snippet:
score = model.evaluate(X_test, Y_test, verbose=0)
```

Listing 3.1: Examples of our paired data of NL description and source code snippets.

3.3.2 Studied Neural Code Generation Models

In this work, we select six state-of-the-art neural code generation models proposed in the past three years as our experimental subjects. These models are selected mainly based on the following inclusion criteria:

- The model should not stick to a specific programming language, which makes it practical and generalizable for it to work on our dataset.
- The model's code of implementation should be publicly available and replicable to be trained from scratch, which can remove potential implementation bias and facilitate the evaluation process.
- The model should have been proposed in recent years and represent the state-of-the-art, i.e., it has a high performance in generation in BLEU score.

As a result, six neural code generation models are selected from 29 candidate papers published on ACL, ICLR, AAAI, TSE, EMNLP, etc. The reason why we

need their implementation code instead of their models is that we need to retrain the models from scratch with our newly curated ML programming task dataset, as the goal of this work is to assess these models' performance on ML programming tasks. The details of these models are as follows:

tranX [180]: proposed a transition-based neural semantic parser that maps NL utterances into formal meaning representations, e.g. executable programs. Specifically, it develops a transition system that decomposes the generation procedure of an AST into a sequence of tree-constructing actions.

External-Knowledge-Codegen (EK Codegen) [212]: proposed a model-agnostic approach based on data augmentation, retrieval, and data re-sampling, to incorporate external knowledge into code generation models. Specifically, the mined NL-code pairs from the online programming QA forum, StackOverflow and API documentation for pre-training. The basic architecture of this model uses **tranX** as its base.

CG-RL [213]: proposed an extended Seq2Tree model equipped with a context-based branch selector, which is capable of dynamically determining optimal branch expansion orders for multi-branch nodes. Specifically, it adopts reinforcement learning to train the whole model with an elaborate reward that measures the loss difference between different branch expansion orders.

Codegen-TAE [183]: exploited the use of a large amount of monolingual programming language corpus to fit a generic transformer-based Seq2Seq model with minimal code-generation-specific inductive bias.

TreeCodeGen [192]: proposed a Transformer-based structure-aware tree-to-tree model. They adopted a Tree Transformer, an attention-based tree-to-tree model with a hierarchical accumulation for code generation.

PyCodeGPT [32]: proposed a large pre-trained language model based on the GPT-Neo [214] with 110M parameters, which is comparable to the powerful GPT-3 model [215]. They mined 1.2M Python repositories in GitHub and filtered/pre-processed high-quality Python code, resulting in 96GB of training data.

In our experiment, to remove potential bias, we use the implementation of their publicly available code with the settings that have the best performance reported in their papers or used in their replication packages. Note that we did not include LLMs, e.g. CodeX and ChatGPT, mainly for two reasons. First, most of these models only open access to their APIs rather than their source code; thus, we cannot retrain and evaluate these models with ML-related programming tasks. Second, these models are trained with open-source repositories, which are highly vulnerable to information leaks, i.e., our test data may have already been included in the training datasets of these models.

3.3.3 Research Questions

We answer the following research questions to evaluate the performance of the studied neural code generation models on ML programming tasks.

- **RQ1 (Accuracy)**: How accurate are the code generation models on different ML programming tasks?
- **RQ2 (Syntactic Correctness)**: How often does the generated code for ML programming tasks pass syntactic checking?
- **RQ3 (Semantic Correctness)**: How often does the generated code for ML programming tasks pass semantic checking?
- **RQ4 (Usefulness)**: Is the generated code for ML programming tasks useful for developers?

In **RQ1**, we set out to investigate the performance of state-of-the-art neural code generation models on the three different types of ML programming tasks. In **RQ2** and **RQ3**, we assess the soundness of the generated code by checking its syntactic and semantic correctness, respectively. In **RQ4**, we explore the usefulness of the generated code to developers in practice via a user case study.

3.3.4 Experiment Setup

We retrain the selected six neural code generation models (details are in Section 3.3.2) from scratch with the newly constructed data mentioned in Section 3.3.1. Specifically, each of the six models is trained and fine-tuned on both ML programming tasks and non-ML programming tasks. Note that the selected six neural code generation models are required to apply specific data pre-processing steps, i.e., building AST from the source code used in **tranX** [180]. Thus, to rebuild these models, we first apply the data pre-processing scripts provided in their replication packages. We have trained two versions of models for the ML tasks, i.e., the universal model, which contains all three tasks mentioned in Section 3.3.1, and the task-only model, which we only use each of the three ML tasks for the training and testing. The reason for making two sets of models is to explore how these neural code generation models perform under different scenarios for ML programming tasks. We design four different scenarios to evaluate the universal models, i.e., *data-uni* (evaluation is conducted on *data process* related ML programming tasks), *model-uni* (evaluation is conducted on *model* related ML programming tasks), *eval-uni* (evaluation is conducted on *evaluation* related ML

programming tasks), and *universal* (evaluation is conducted on all three types of ML programming tasks combined). Note that the universal model and task-only model evaluate the same test set of the three ML tasks, and the universal model evaluates an additional combined ML task (universal). They also share the same settings when training the model for a fair comparison. To maximize the potential of the evaluated approaches, we perform hyperparameter tuning for each of the evaluated approaches. For **tranX**, we use 0.3 for the dropout, a hidden size of 256, an embedding size of 128, a learning rate of 0.001, and a batch size of 64. We train the model for a maximum of 20 epochs and use a beam size of 15 when testing. For **EK-codegen**, we use the same value of the hyperparameter used in **tranX**. For **CG-RL**, we used the same dropout, network size, and learning rate for the training. However, we use 10 epochs to pre-train the branch selector following the original paper of **CG-RL** [213]. For **Code-gen-TAE**, we use 1e-05 for the encoder learning rate and 7.5e-05 for the decoder learning rate, the hidden size of the encoder is 768 with 12 layers, the decoder size is 256 with 4 layers, and the batch size used is 16. We also trained them for 20 epochs, like the rest of the model and used a beam number of 10 when testing. For **TreeCodeGen**, 0.1 is used for the dropout, and the hidden and embedding size used is the same as **tranX**. The learning rate used is 1e-4. The max epoch used for training is 20, and the beam size used for testing is 30. For **PyCodeGPT**, the model is trained with 32K vocabularies, 768 for hidden size, 12 for the number of heads and layers, a window size of 256, and an initializer range of 0.02, which are the same settings reported from GPT-Neo and PyCodeGPT. The model was trained for 100K steps as suggested in the original paper.

All the aforementioned settings used for the training are from the papers of the six models and any details missing from the paper are set as the default settings provided by their replication packages. We used the GPUs in the Google Colab Pro[6], which dedicates either NVIDIA Tesla P100 or T4, for training models. For training PyCodeGPT, we used NVIDIA A100 40GB from the Google computing platform[7]. The time cost of model training of these models ranges from 3 hours (i.e., **tranX**) to 16 hours (i.e., **PyCodeGPT**).

3.3.5 Evaluation Metrics

To assess the performance of the models in generating high-quality source code, we use the following four evaluation measures, i.e., BLEU [216], METEOR [217], ROUGE [218], and CodeBLEU [219].

[6] `https://colab.research.google.com/`

[7] `https://cloud.google.com/`

BLEU score [216] is an evaluation metric commonly used in assessing the generation models in natural language processing. It measures the similarity of the model-generated text to the ground truth text by computing the overlapping n-grams of the two texts. A value of 0 indicates that the generated source code has no overlap of n-grams with the ground truth source code, while a value of 1 indicates that it is a perfect overlap with the ground truth source code. There are different ways to calculate the BLEU score. We can set a different number of n-grams we want to assess in overlapping, i.e. uni-gram, bi-gram, tri-gram, and 4-gram. This is also referred to as BLEU-1, BLEU-2, BLEU-3, and BLEU-4, respectively. METEOR metric [217] is also commonly used to assess machine language generation models. The metric is based on the harmonic mean of unigram precision and recall, but with a higher weight on recall. ROUGE [218] is a metric used for evaluating natural language generation models to compare the generated sequence to the reference. There are also multiple sets of assessing different ROUGE values. In this study, we report the ROUGE-L value of each model, which uses the Longest Common Sub-sequence (LCS) in assessing the similarity of the two texts. CodeBLEU [219] is a metric used to evaluate generated texts, specially designed to evaluate the quality of generated code. It uses a mixture of the original BLEU scores, a weighted BLEU score that gives more weight to keywords in the programming language, a syntactic match using AST, and a semantic match of data flow. We use the implementation in CodeXGLUE [220] to calculate this metric in this work.

3.4 Results and Analysis

This section presents the experimental results and answers the research questions discussed in Section 3.3.3.

3.4.1 RQ1: Accuracy

Approach: To answer this RQ, we follow our experiment setups (described in Section 3.3.4) to re-train each of the six neural code generation models with data from different ML programming tasks, i.e., *data process*, *model building*, and *evaluation*, for the task-only models. Note that we train the universal models on data including all three types of ML programming tasks. We also evaluate the performance of the six neural code generation models on the non-ML programming task dataset for comparison. To assess the model's capability of generating source code on the different ML libraries, we label each testing instance with the studied ML library using the ML API name.

Result: The performance of the studied six neural code generation models on different types of ML tasks and non-ML tasks is organized in Table 3.4. Overall, **PyCodeGPT** performs best, especially on ML-related tasks. For example, **PyCodeGPT** achieves the largest values regarding all the metrics on *model building* and achieves the best performance in ML tasks for 23 out of 28 (4 tasks * 7 metrics). However, on non-ML tasks, **EK Codegen** performs best with 6/7 of the metrics. A possible reason for these two models to have better performance is that they exploit external data for pre-training. Specifically, **PyCodeGPT** uses corpus mined from GitHub, and **EK Codegen** uses corpus mined from Python Standard Library[8] and StackOverflow[9]. Due to the abundance of ML projects on GitHub, **PyCodeGPT** benefits from generating ML-related code. Similarly, non-ML tasks are composed of short, standalone code snippets, which are very similar to the code snippets answered on StackOverflow. This could be the reason why **EK Codegen** has benefited in generating non-ML programming tasks. Regarding the performance on a specific type of ML task, we can see that the best-performing tasks for each neural code generation model vary. For example, **tranX** performed best on *model building* tasks over other tasks, while **CG-RL** and **Codegen-TAE** have better performance for *data processing* tasks than the performance on other tasks regarding BLEU scores. **EK Codegen** and **TreeCodeGen** perform better on *evaluation*, while **PyCodeGPT** performs better on *model building* related ML tasks than the other tasks.

In addition, we can also see that all six neural code generation models perform significantly better on ML tasks compared to non-ML tasks regarding all the evaluated metrics, i.e., BLEU-1/2/3/4, ROUGE-L, METEOR, and CodeBLUE. Possible reasons for this can be as follows. First, the non-ML tasks are much more diverse than ML tasks, as we limited the ML tasks to only four libraries, i.e., TensorFlow, Keras, PyTorch, and Scikit-learn. The variance of programs from ML tasks is less than the non-ML dataset. Second, the natural language descriptions of the non-ML-related tasks were also more varied and context-free since the domain does not have a constraint compared to the descriptions of ML tasks. This finding is interesting, as previous studies state that traditional programs are more deterministic and less statistically oriented [184]. However, from the generation point of view, it is the opposite. Even though the ML libraries are more dynamic and statistically oriented, programs built upon them are often not. Regarding the universal models and task-only models, we can see that the universal models generally perform worse than the models that are learned only on a specific task. This suggests that these neural code generation models fail to generalize to different tasks as the training space and

[8] `https://docs.python.org/3/library/index.html`

[9] `https://stackoverflow.com/`

Table 3.4: Performance of the studied six neural code generation models on different ML tasks and non-ML tasks. The best performance of a model on different tasks is shown in bold, while the overall best model on each ML task is shown with an asterisk (*). *data/model/eval-uni* shows the results of the universal model evaluated on each ML programming task, and *universal* model shows the results of the universal model evaluated on the universal ML programming tasks.

Models	Tasks	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	METEOR	CodeBLEU
tranX	*data process*	0.3606	0.2670	0.2237	0.1905	0.3380	0.3600	0.1950
	data-uni	0.3232	0.2280	0.1864	0.1559	0.2992	0.3307	0.2121
	model building	**0.3646**	**0.2717**	**0.2277**	**0.1940**	0.3167	0.3598	0.2040
	model-uni	0.3558	0.2521	0.2065	0.1725	0.2887	0.3397	0.1937
	evaluation	0.3621	0.2677	0.2233	0.1886	**0.3532**	**0.3712**	**0.2170**
	eval-uni	0.3351	0.2251	0.1797	0.1476	0.3142	0.3583	0.1902
	universal	0.3455	0.2439	0.1997	0.1672	0.2974	0.3421	0.2019
	non-ml	0.2855	0.1714	0.1236	0.0929	0.1636	0.2282	0.1412
EK Codegen	*data process*	0.3391	0.2541	0.2154	0.1853	0.3441	0.3592	0.1877
	data-uni	0.3335	0.2361	0.1944	0.1635	0.3076	0.3303	**0.2186**
	model building	0.3549	0.2690	**0.2289**	**0.1978**	0.3241	0.3633	0.2082
	model-uni	**0.3671**	0.2619	0.2155	0.1801	0.2987	0.3455	0.1971
	evaluation	0.3593	**0.2698***	0.2282	0.1953	**0.3616**	**0.3759***	0.2164
	eval-uni	0.3422	0.2305	0.1842	0.1511	0.3106	0.3637	0.1948
	universal	0.3557	0.2523	0.2076	0.1741	0.3043	0.3465	0.2069
	non-ml	0.2969*	0.1838*	0.1348*	0.1028*	0.1705	0.2365*	0.1454*
CG-RL	*data process*	**0.2503**	**0.1943**	**0.1678**	**0.1471**	0.3397	0.3333	0.1558
	data-uni	0.1535	0.1172	0.1000	0.0867	0.3071	0.2873	0.1626
	model building	0.1923	0.1471	0.1257	0.1089	0.3160	0.3057	0.1551
	model-uni	0.1410	0.1060	0.0896	0.0772	0.2932	0.2826	0.1373
	evaluation	0.2404	0.1840	0.1575	0.1366	**0.3620**	**0.3480**	**0.1752**
	eval-uni	0.1633	0.1148	0.0904	0.0796	0.3099	0.3220	0.1409
	universal	0.1518	0.1144	0.0971	0.0840	0.3022	0.2967	0.1488
	non-ml	0.1213	0.0776	0.0578	0.0450	0.1723	0.1916	0.1004
TreeCodeGen	*data process*	0.3465	0.2540	0.2125	0.1810	**0.3530***	0.3476	0.1800
	data-uni	0.2785	0.1847	0.1437	0.1142	0.2976	0.2695	0.1744
	model building	0.3483	0.2559	0.2134	0.1812	0.3122	0.3377	0.2044
	model-uni	0.3391	0.2280	0.1815	0.1477	0.2725	0.2947	0.1717
	evaluation	**0.3644***	**0.2681**	**0.2235**	**0.1883**	**0.3530**	**0.3619**	**0.2154**
	eval-uni	0.3282	0.2032	0.1533	0.1191	0.2957	0.3139	0.1716
	universal	0.3210	0.2137	0.1684	0.1358	0.2874	0.2920	0.1746
	non-ml	0.1997	0.1255	0.0932	0.0723	0.1929*	0.1979	0.1135
Codegen-TAE	*data process*	**0.3775***	**0.2444**	**0.1857**	**0.1441**	**0.3145**	0.3233	0.1367
	data-uni	0.3304	0.1938	0.1349	0.0937	0.2784	0.2844	0.1440
	model building	0.3279	0.1993	0.1438	0.1052	0.2339	0.2623	0.1321
	model-uni	0.3499	0.2056	0.1456	0.1047	0.2476	0.2913	0.1361
	evaluation	0.2919	0.1743	0.1260	0.0914	0.2357	0.2358	0.1261
	eval-uni	0.3396	0.2188	0.1716	0.1377	0.2990	**0.3239**	**0.1531**
	universal	0.3514	0.2102	0.1514	0.1109	0.2681	0.2972	0.1442
	non-ml	0.2620	0.1451	0.0985	0.0703	0.1217	0.1944	0.1126
PyCodeGPT	*data process*	0.3347	0.2562	0.2219	0.1958	0.3333	0.3462	0.2403
	data-uni	0.3567	0.2784*	0.2422*	0.2137*	0.3436	0.3626*	0.2477*
	model building	0.4036	0.3286	**0.2925***	0.2634	**0.3701***	**0.3893***	**0.2678***
	model-uni	**0.4139***	**0.3296***	0.2898	**0.2898***	0.3357	0.3773	0.2582
	evaluation	0.3397	0.2693	0.2361*	0.2094*	0.3667*	0.3654	0.2451*
	eval-uni	0.3089	0.2400	0.2108	0.1882	0.3230	0.3177	0.2284
	universal	0.3826*	0.3030*	0.2661*	0.2370*	0.3371*	0.3595*	0.2528*
	non-ml	0.0789	0.0352	0.0217	0.0141	0.0619	0.0938	0.1164

variance increase. However, we can see that the universal model of **PyCodeGPT** evaluated on *data processing* and *model building* benefits from the larger training data. Since **PyCodeGPT** is pre-trained on a large corpus of source code, it could have learned to generalize code from the knowledge learned from pre-training. We also find that, except for **TreeCodeGen**, all models on *data processing* tasks have higher CodeBLEU scores compared to the task-only models. One of the possible reasons could be that *data processing* provides relatively more data for training models, as shown in Table 3.3.

Good: Neural code generation models generate significantly better performance on ML tasks than non-ML tasks.
Bad: Neural code generation models show mixed results when generating different ML programming tasks.

3.4.2 RQ2: Syntactic Checking

Approach: To answer this RQ, we conduct syntactic checking on the generated programs for each instance in the testing dataset. Following existing work [190], we conduct a static syntactic checking on the generated programs with the state-of-the-practice tool Pylint[10], a Python static code analysis tool that looks for programming errors. Note that **Codegen-TAE** generates all code statements in one line for a given programming task, while Python programs have strict style requirements, e.g., Python uses indentation to indicate a block of code. To avoid any style errors, we first parse the generated source code and output it with proper indentation. During this step, we do not change any content to avoid potential bias. We then use Pylint to check the syntactic correctness of each instance in the testing dataset. We consider an instance syntactically correct if it passes the checking.

Result: Table 3.5 shows the results of the syntactic checking for programs generated by different neural code generation models in different ML tasks and non-ML tasks. We can observe that, in general, all the state-of-the-art models have a high accuracy in generating syntactically correct programs. However, we observed a significant decline in generating syntactically correct code using **PyCodeGPT** compared to other models. One of the possible reasons for this is that these five models are AST-based approaches. Given a programming task, they first generate the corresponding AST and then transfer it into a code snippet. On the other hand, **PyCodeGPT** uses a token-based seq2seq Transformer, giving the model the freedom to generate any token sequences, which can be easily syntactically incorrect. In

[10] `https://pylint.pycqa.org/`

addition, we can also see that all six neural code generation models perform better on most of the ML tasks compared to non-ML tasks, which is consistent with our findings in RQ1.

In Listing 3.2, we show two examples of common syntactical errors in code that are generated from the neural code generation models. The first example is a *model building* task generated by the **PyCodeGPT** model. The NL description asks to generate a snippet that creates a logistic regression model. We can see that the model successfully generated a code snippet that creates and fits a logistic regression with a cross-validation estimator (CV). However, in the first line, it generated a natural language description which causes a syntax parsing error. Since **PyCodeGPT** is a token-based generator, it does not restrict its generations within an AST form. The second example is an *evaluation* task generated by the **Codegen-TAE** model. The NL description asks to predict at point (0, 0). However, the model generated "0 .0, 0.0", where the white space before the .0 token causes a syntax error. Despite **Codegen-TAE** using AST encoding when training, the decoder does not force it to generate within an AST, leaving the results vulnerable to syntax errors. We have manually checked 1k samples, and the most common ones we found are generating non-Python code tokens, i.e., natural language comments, incorrect white space, and incomplete code (e.g., closing brackets/parentheses, right-hand side of assignment operation). These are deemed to be minor errors that could be easily handled with AST-based decoders, just as other AST-based decoder models have shown.

```
# Model building example generated by PyCodeGPT
# NL intent:
'create a logistic regression model'
# Ground truth snippet:
voicemodel = LogisticRegression()
# Generated snippet:
using CV to optimize value of C
lrCV = LogisticRegressionCV(Cs=np.logspace(-4, 4, 10), cv=5, n_jobs=-1,
                            solver='liblinear', random_state = 420)
lrCV.fit(X_train, y_train)

# Evaluation example generated by Codegen-TAE
# NL intent:
'make prediction on point ( 0, 0 ) works same way as sklearn'
# Ground truth snippet:
point = np.array([[0, 0]])
model.predict(point)
# Generated snippet:
0 .0,0.0
```

Listing 3.2: Syntax error examples generated by the models.

Table 3.5: Syntactical correctness (in percentage) of the generated code by different neural code generation models on different types of machine learning tasks.

Methods	data	model	evaluation	non-ml
tranX	98.40	98.68	99.36	97.06
EK codegen	96.39	97.92	97.68	95.16
CG-RL	97.37	98.07	97.97	94.70
TreeCodeGen	92.24	95.46	97.00	82.85
Codege-TAE	92.35	93.99	90.70	93.22
PyCodeGPT	71.36	70.53	72.88	67.16

Good: The neural code generation models generate highly accurate syntactically correct programs for ML tasks.
Good: Our results confirm the advantage of AST-based code generation used in current neural code generation models compared to the token-based model.

3.4.3 RQ3: Semantic Checking

Approach: To further analyze the soundness of the generated programs by these state-of-the-art neural code generation models, we also check the semantics of the generated programs manually, as most of the generated programs require specific inputs to be executed, which cannot be generated automatically. For this experiment, we examine the semantic correctness of the generated programs from the best three models, i.e., **tranX**, **EK codegen**, and **PyCodeGPT** on the TensorFlow library. Specifically, we first randomly select 50 examples from each type of ML programming task, resulting in a total of 450 programming tasks to be checked (50 samples $\times$ 3 types of ML programming tasks $\times$ 3 models). We consider a generated program of a given programming task to be semantically equivalent to the ground-truth program if the generated program shares the same logic as the ground-truth program and can solve the given programming task. To check the semantic correctness of each sample, all the authors work together and make decisions together.

To further illustrate the effectiveness of the generated programs for ML programming tasks, we also evaluate the correctness of a generated program regarding the APIs called in the program. Specifically, given an ML programming task, we first parse its corresponding ground truth source code and extract the APIs from these four ML libraries that are used to solve the task. Then we follow the same process to extract APIs in the generated program for this task by a neural code generation model. Note that, for the non-ML tasks, we also follow the same process to collect APIs used in the ground truth and generated programs. For measuring the performance

Table 3.6: Semantic correctness (in percentage) of the generated code by different neural code generation models on different types of machine learning tasks.

Methods	data	model	evaluation	non-ml
tranX	6%	14%	34%	0%
EK codegn	16%	12%	30%	10%
PyCodeGPT	34%	14%	30%	6%

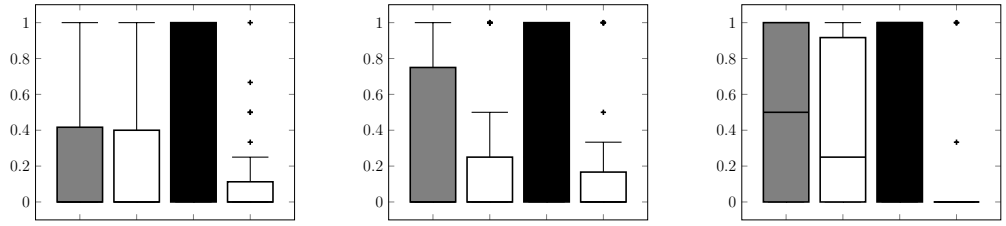
Table 3.7: Correctness of identified APIs on the generated code by different neural code generation models on different types of machine learning tasks.

Methods	Metrics	data	model	evaluation	non-ml
tranX	Precision	0.2129	0.2740	0.4348	0.1214
	Recall	0.2215	0.2729	0.5105	0.1364
EK codegen	Precision	0.3458	0.2104	0.4234	0.2278
	Recall	0.3400	0.2693	0.4452	0.2078
PyCodeGPT	Precision	0.6133	0.4085	0.5500	0.0867
	Recall	0.5811	0.3943	0.4649	0.0783

of neural code models regarding identifying the correct APIs, we use `Precision` and `Recall`, where `Precision` measures the percentage of correctly identified APIs among all the used APIs in the generated programs, and `Recall` is the percentage of correctly identified APIs among the ground-truth APIs.

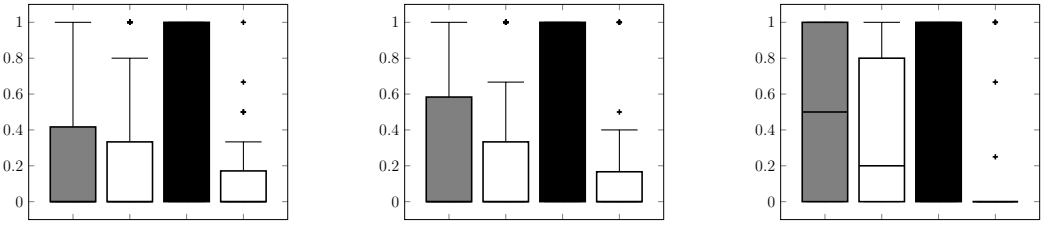
Result: Table 3.6 shows the semantic correctness of the generated programs of the three neural code generation models. We can observe that only a small number of samples of the generated source code are semantically correct, with an average of 17% out of the 450 samples. **PyCodeGPT** outperforms **EK Codegen** followed by **tranX** with 13%, 17%, and 21% of the average percentage, respectively, which is consistent with the results from RQ1. Similar to the trends revealed in RQ1 and RQ2, the examined models also have better performance on ML programming tasks than non-ML programming tasks.

Despite the low values in semantic correctness, we observe that these neural code generation models have better performance in suggesting the correct APIs for the given programming tasks compared to directly generating source code. Table 3.7 shows the performance of these models in identifying the correct APIs. As we can see, the precision and recall values for identifying correct APIs are higher across all different ML programming tasks, which suggests they can help identify an average of 39% of correct APIs for solving ML tasks. We further show the box plots of precision and recall values regarding identifying correct APIs of the three models in Figure 3.2 and Figure 3.3, respectively.



(a) tranX (b) EK Codegen (c) PyCodeGPT

Figure 3.2: Precision of identifying correct APIs of **tranX** and **EK Codegen PyCodeGPT**, on *data*, *model*, *eval*, *nonml*, respectively.



(a) tranX (b) EK codegen (c) PyCodeGPT

Figure 3.3: Recall of identifying correct APIs of **tranX** and **EK codegen PyCodeGPT**, on *data*, *model*, *eval*, *nonml*, respectively.

We further show common bad and good examples of ML tasks generated by existing neural code generation models in Listing 3.3. Specifically, the first one is a bad example, which shows that the generated code does not fulfill the query. However, some of the required APIs can be identified. The task is labelled as a *model building* task finished with the TensorFlow library, which is about building and running a session by loading a dictionary object. The ground truth code instantiates two placeholders with a *tf.int32* type and add them before running a session. When running the session, a dictionary object should be called. While the generated source code failed to generate the dictionary part, it only generates code that instantiates a *tf.int32* typed placeholder. The precision and recall regarding identifying the correct APIs are 100% and 57%, respectively. For a good example, the model successfully generates snippets with the correct APIs, with only the data paths differing from each other. Based on our observations of the common bad and good examples, we find that the models have difficulties in generating correct code that has multiple complex API call sequences or requires multiple operations. On the contrary, the

models successfully generate correct code for tasks that require a smaller set of API calls or use common API calls.

The models successfully generate the correct APIs, as the commonly used ML-related APIs are limited compared to APIs of general programming tasks. Limiting the number of APIs could also contribute to making the generation of machine learning programming tasks easier. As shown in Listings 3.2 and 3.3, the difference between the generated code to the ground truth in semantics is mostly due to different contexts, e.g., variable names, number of parameters used, and different numerical values. This is somewhat inevitable, as most NL descriptions do not provide specific values or names for the query (similar to how developers search queries on the web). Constructing a dataset that provides specific context for APIs will be a promising direction to improve these models.

```
# A bad example generated by the EK codegen model for a model building task
# NL intent:
'using a feed_dict, same graph, but without hardcoding inputs when building session'

# Ground truth snippet:
a = tf.placeholder(dtype=tf.int32, shape=(None,))
b = tf.placeholder(dtype=tf.int32, shape=(None,))
c = tf.add(a, b)
with tf.Session() as sess:
    result = sess.run(c, feed_dict={
        a: [3, 4, 5], b: [-1, 2, 3]})
    print(result)

# Generated snippet:
x = tf.placeholder(tf.float32)
with tf.Session() as sess:
    print(sess.run(x))

# A good example generated by the PyCodeGPT model for a data processing task
# NL intent:
'first let us import the train file and get some idea about the data'

# Ground truth snippet:
train_df = pd.read_csv("../input/train.csv")
train_df.shape

# Generated snippet:
train_df = pd.read_csv("../data/train_mod.csv")
train_df.shape
```

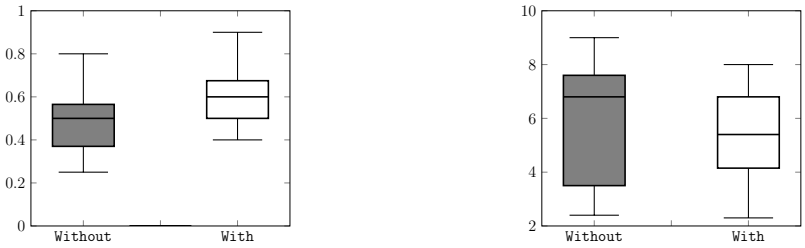
Listing 3.3: Examples of common good and bad generations from the neural code generation models for ML programming tasks.

Bad: All the examined neural code generation models are weak in generating semantically correct programs.
Good: The generated programs have the potential to be used for identifying correct APIs for ML programming tasks.

3.4.4 RQ4: Usefulness

Approach: To explore the practical value of neural code generation models, we further conduct a user case study to investigate whether programs generated by these neural code models can help developers finish given programming tasks efficiently and accurately. In this experiment, we examine the programs generated by the best two models, i.e., **PyCodeGPT** and **EK codegen**, on the TensorFlow library. We randomly selected 20 programming tasks from the test dataset of TensorFlow. We have sent an email to all the graduate students in our EECS department. Out of the 240 M.Sc. and Ph.D. students, 12 of them have responded (response rate=5%). We invited these 12 students (i.e., six Ph.D. students and six MS students) who are familiar with the TensorFlow framework to complete the 20 programming tasks. Their experience in developing ML tasks based on TensorFlow varied from two to four years, with an average of three years. We then divided the participants into two groups (Group 1 and Group 2) based on their experience. The 20 tasks were also randomly divided into two groups (Task1 and Task2). The experiment was conducted in two steps. In the first step, Group1 and Group2 were asked to work on Task1, while Group1 would be given the generated program from both **PyCodeGPT** and **EK codegen**. In the second step, Group1 and Group2 were asked to work on Task2, while Group2 would be given the generated program from both **PyCodeGPT** and **EK codegen**. Each participant was asked to record his/her screen during the experiment to get the time he/she spent on solving each programming task. Following existing studies [190, 221], we use correctness and completion time to measure the performance of the participants in solving ML tasks. Specifically, correctness evaluates whether a participant can write the correct code for a given task. Note that, as developers might solve the same task with different logic, we measure the proportion of correct APIs submitted by a participant among all APIs in the ground truth answer of the task. Completion time evaluates how quickly a participant can solve a given task. For each task, we recorded the correctness and completion time of each participant.

Result: Figure 3.4 shows the distribution of correctness and completion time of participants to finish the given 20 ML tasks. Overall, with the generated code from the two code generation models, participants completed the programming tasks more accurately and took slightly less time than participants without any clues.



(a) Correctness (b) Completion Time (mins)

Figure 3.4: Performance of participants with (denoted as '**With**' in the figure) and without (denoted as '**Without**' in the figure) the help of code generation models.

On average, the correctness and completion time of participants with and without the generated code are 0.62 and 5.2, versus 0.51 and 5.6, respectively. We further used the Wilcoxon signed-rank test [222] to verify the statistical significance of the differences. The p-value for correctness is smaller than 0.05, while the p-value for completion time is larger than 0.05. We also report the effect size of the two groups using Cohen's d [223]. The effect size for the correctness result is 0.7788 and 0.1826 for the completion time result. As the results show, the effect size of correctness is considerable (close to 0.8, being large). However, for completion time, the effect size is very small (lower than 0.2, which is considered small). The result suggests that the generated code does not have significance in reducing the time cost of ML programming tasks, while it showed significance in guiding developers to write more correct code. One possible reason for this is that most of the generated programs are semantically incorrect, and modifying the generated programs is not easy and requires a great amount of time. Our discussion with the participants confirmed that the examined code generation models can help find a part of the correct APIs for ML tasks, which provides developers with useful clues, thus further improving developers' correctness.

Bad: Code generation models do not have significance in improving developers' completion time, as most of the generated programs are semantically incorrect, which requires a non-trivial time for modifications.
Good: The generated code can help developers write more correct code by providing developers with clues for using correct APIs.

3.5 Discussion

3.5.1 What Is Missing in Code Generation for ML Tasks

The above results and analysis have revealed several good and bad aspects of state-of-the-art neural code generation models on ML programming tasks. This section further summarizes the missing aspect that can serve as the practical guidelines for improving code generation tasks for ML tasks.

Decompose code generation for divide-and-conquer. Even powered with state-of-the-art deep neural techniques, code generation for ML programming tasks is still a challenge. Directly generating code from a given NL-described programming task often produces incorrect and incompetent programs that cannot be used by developers. Nevertheless, the findings of our user case study reveal that incorrectly generated code could still be useful for developers if it contains essential APIs to solve tasks. Meanwhile, identifying the correct APIs is easier than generating the correct programs directly. Taken in this sense, future code generation techniques for ML tasks can be decomposed into two sequential tasks, i.e., API sequence identification and API usage generation. Crowd-sourced knowledge from Stack Overflow can be used to help generate the context of the identified API sequences, which has been proven to be useful in mining API usage scenarios [224, 225].

Human-machine collaborated code generation. The examined state-of-the-art neural code generation models treat code generation as a machine translation task and generate statements/blocks from a given NL-described programming task sequentially, during which any incorrect statement could mislead the generation of the sequential code snippets. Meanwhile, during our case study, similar to the findings of Murali et al. [226], we also find that, despite lacking the whole knowledge of certain programming tasks, developers usually know whether it is correct for some generated statements, or whether a specific API should be utilized based on their knowledge of an ML library. If human knowledge could be integrated into the process of automatic code generation, e.g., in the form of simple feedback from end-users to assess the correctness of the intermediate uncertain statements, the code generation could be significantly improved.

Such practice has another advantage, i.e. personalizing the generated code that fits the knowledge of different end-users, by integrating developers' knowledge into the code generation process.

Focus on domain-specific tasks. Compared to the general non-ML programming tasks, all six neural code generation models exert better performance regarding

syntactic correctness, semantic correctness, and identifying essential APIs, on domain-specific tasks, i.e., ML programming tasks. We assume this might be because the domain-specific task only involves a smaller set of APIs (indicating little knowledge needed to be learned), thus training an effective model is relatively easier compared to training a model on a dataset with diversified tasks. We also notice that previous practices on code generation were usually conducted on general tasks [187, 188, 189, 191]. Thus, future code generation should focus on training or fine-tuning models with data from specific domains to improve the performance of generating programs for domain-specific tasks, which have much more potential than general programming tasks. We also encourage the need for the design of domain-specific models by incorporating the characteristics of the domain, which might further improve the performance of specific tasks.

3.5.2 Threats to Validity

Internal Validity. The main threat to the internal validity of our study is the limited number of models that are compared. Due to this limitation, we can not generalize our findings from this paper to all source code generation models. However, we have used the most recently published state-of-the-art performing ones to conduct our experiments. We also described a detailed methodology, the setup of the study, and the data used, which will allow future researchers to replicate our study and further explore other source code generation models. Another threat to our study is that all 12 participants for RQ4 are students. Although these students have at least two years of experience in developing ML tasks, they cannot fully represent the professionals in the industry. In the future, we plan to recruit real-world ML developers to evaluate the usefulness of these neural code generation models. In this work, following the six studied baseline models, we also used the hold-out evaluation, i.e., 80% of the data is used for training and the left is used for testing. To show the impact of randomness in our experiments, we evaluated one of the baselines (i.e., PyCodeGPT) on one of the datasets (i.e., model-building) with five-fold cross-validation. As a result, we did not observe a significant difference in the metric scores among the five runs.
External Validity. The main threat to external validity is the labelling of ground-truth data explored in the study. Although we have put much effort into maintaining high-quality data by using real-world programs contributed by developers on GitHub, it is not enough to validate the implementation of the source code. Also, programming or implementing a programming task can lead to many different forms and logics of source code. Thus, these programs in our dataset might not represent all the ways to implement a given programming task.

Construct Validity. The main threat to the construct validity can be the evaluation metrics we used. BLEU, ROUGE-L, and METEOR scores can assess program generation by calculating how the generated programs of neural code generation models are similar to the ground-truth programs of given tasks in our dataset. However, as mentioned from the external validity, other implementations can be a reasonable answer rather than using similarity to the ground truth. In our future study, we plan to examine a different set of evaluation metrics to assess the generation of the program.

3.6 Related Work

3.6.1 Automatic Code Generation

Besides the six neural code generation models used in this paper, there are many other methods for automatic code generation from natural language descriptions proposed in the recent decade [227, 228, 229]. More specifically, deep neural networks in the natural language processing field have been increasingly adopted to encapsulate the complexity of generating general-purpose programming languages such as Java and Python. Dong and Lapata [198] proposed *Coarse2Fine*, which uses a structure-aware neural architecture that applies two decoding steps of semantic parsing. First, the approach decodes a rough sketch of the meaning representation at a high level, where it omits the details such as identifiers. Then they apply another mechanism that fills in the missing details by conditioning the natural language description together with the sketch of the general meaning representation. Hayati et al. [230] proposed *ReCode*, which exploits sub-tree retrieval for explicitly referencing code examples within a neural code generation model. This approach adopts a retrieval-based neural machine translation approach in the context of source code generation. This is the first approach that introduced the retrieval technique to code generation. Murali et al. [226] proposed an approach that combines neural networks and combinatorial search to generate source code from sketches, which is an abstract style of source code that masks user-defined constants and identifiers. Sun et al. [231] proposed a grammar-based structural convolutional neural network (CNN) that utilizes tree-based convolution, pre-order convolution, and attentive pooling layers for a more compact prediction than seq2seq models. The tree-based convolution applies a sliding window on the AST structures. Pre-order convolution that traverses the nodes of the partial AST. These two modules of CNN help capture the neighbouring information of the trees. The attentive pooling is designed to aggregate the CNN features and better predict the identifier names. Sun et al. [181] proposed *TreeGen*, which is a tree-based

neural architecture that uses the attention mechanism of Transformers to alleviate the long dependency issues and utilizes an AST encoder to incorporate grammar rules and AST structures into the network. They utilized a convolution sub-layer only on the first Transformer decoder blocks to prevent the Transformer from mixing information from other nodes, which may leak the original information. Lyu et al. [193] proposed a generation model that exploits the dependencies of the API method as an API dependency graph and utilizes the graph embedding into a seq2seq model. They incorporate an encoder-decoder module that captures both the global structural dependencies and textual program descriptions when generating the source code. Ahmad et al. [232] proposed *PLBART*, which is an auto-regressive transformer model that pre-trains on an extensive collection of unlabelled Java and Python functions that are paired with natural language texts through denoising auto-encoders. This approach helps reduce the effort of mining highly human-curated annotations when fine-tuning specific tasks. They have evaluated *PLBART* on several tasks, i.e. code summarization, code generation, code translation, program repair, clone detection, and vulnerability detection.

API recommendation can be seen as a kind of code generation, which targets to generate a specific API set for a given query from developers [233, 234, 235, 204, 236, 221, 201, 237]. Liu et al. [236] proposed a method called *RecRank* that applies a novel ranking-based discriminative approach that leverages API usage path features to improve their performance on the recommendation. Xie et al. [221] studied the relationship between verb phrase patterns that describe the functionality of APIs and the user queries for finding those APIs. Chen et al. [201] proposed a novel API recommendation method that combines API usage with the text information in the source code based on an API context graph network. Chen et al. [237] propose *COOK*, which is an API sequence recommendation that combines deep learning models and post-processing strategies. They have enhanced the beam search with code-specific heuristics that helped them improve the quality of the recommendation.

3.6.2 Analysis of Model Effectiveness

There have also been numerous empirical studies that investigated the effectiveness of source code generation models in different contexts. Liu et al. [190] investigated the effectiveness of source code generation models when natural language requirement texts of general programming are given. They have built a large-scale dataset, *ReCa*, to assess five source code generation models in handling longer sequences of requirements. Chirkova and Troshin [238] conducted an empirical study on approaches that apply Transformers [2] to source code in the software engineering domain. They investigated

how Transformers utilize the syntactic information of source code in modelling code completion, function naming, and bug-fixing tasks. Antonio et al. [239] empirically assessed the effect of T5 (Text-to-Text Transfer Transformer) [240] architecture when applied to source code. They found that their T5 model outperforms four existing baselines when bigger data is used in pre-training. Peng et al. [241] did a study to unify the tasks and benchmark datasets of API recommendations to fully facilitate future studies. They grouped the different approaches concerning the input they handle (i.e. query or code). They propose a general benchmark dataset, *APIBENCH*, to assess 11 existing approaches in both query and code-based API recommendations.

These empirical studies assess the effectiveness of source code generation models in generating general programming tasks regarding different levels of NL complexity [190] and different architectures [238, 239]. The difference in our work is that we investigate the effectiveness of generation models in generating machine/deep learning-related source code using ML libraries.

3.7 Conclusion

This paper conducts an empirical study to explore the performance of six state-of-the-art neural code generation models on ML programming tasks. A new benchmark dataset that contains 83K pairs of natural language-described programming tasks and the corresponding programs implemented by four ML libraries is introduced. Our empirical study reveals some good, bad, and missing aspects, with a few major ones listed below. (**Good**) Neural code generation models perform significantly better on ML tasks than on non-ML tasks. (**Bad**) Most of the generated code is semantically incorrect. (**Bad**) Code generation models do not have significance in improving developers' completion time. (**Good**) The generated code can help developers write more correct code by providing developers with clues for using correct APIs. (**Missing**) The observation from our user study reveals the missing aspects of code generation for ML tasks, e.g., decomposing code generation for divide-and-conquer into two tasks: API sequence identification and API usage generation.

Chapter 4

Prompt Engineering vs. Fine-Tuning in Three ASE Tasks

4.1 Introduction

Language models have gained significant interest, leading to numerous studies adapting their use for automated code-related tasks. Although originally developed for natural language, Large Language Models (LLMs) demonstrate potential in automated code-related tasks. Previous literature mainly focuses on pre-training on large corpora of Source Code (SC) and Natural Language (NL) and fine-tuning with task-specific datasets for various tasks. State-of-the-art LLMs, however, are trained using massive unsupervised corpora and use prompts for task querying. The large training corpus and numerous parameters enable LLMs to perform well on automated code-related tasks with simple prompts.

Prompting offers advantages over fine-tuning: it does not require labelled datasets, which are costly to acquire. Furthermore, running a prompt is less resource-intensive than fine-tuning an LLM. Despite its potential, prompt engineering is still in its early stages and lacks systematic studies on its performance compared to fine-tuned models in code tasks.

This paper presents a quantitative and qualitative investigation of *OpenAI's ChatGPT*, specifically the latest version (*GPT-4*), which has shown notable improvements over its predecessor, *GPT-3.5*. We focus on three automated code tasks: code summarization (SC-to-NL), code generation (NL-to-SC), and code translation (SC-to-SC). These tasks are chosen for their commonality among developers and prevalence in the literature. We employ three automated prompting techniques (basic, in-context learning, and task-specific prompts) and compare them with 17 fine-tuned language

models. Additionally, we surveyed 27 graduate students and 10 industry practitioners to gather qualitative insights through conversational prompts. We address the following research questions:

RQ1: How does *GPT-4* with automated prompting compare to fine-tuned models in performance? We quantitatively assess *GPT-4* using three prompting strategies: (a) basic prompt, (b) in-context learning, and (c) task-specific prompt, comparing results with fine-tuned language models.

RQ2: How do participants perceive the usefulness of *GPT-4* using a basic prompt? We ask participants to assess the basic prompting strategy qualitatively to understand its usefulness.

RQ3: How do participants refine their prompts when interacting with *GPT-4*? We investigate how participants refine prompts to achieve better results by analyzing their interaction logs and summarizing prompt evolution patterns.

RQ4: What is the impact of different prompt evolution patterns? We investigate the impact of prompt evolution patterns derived from participant interactions.

Our quantitative analysis suggests that prompt-engineered *GPT-4* does not consistently outperform fine-tuned LLMs in all tasks. For code summarization, *GPT-4* with task-specific prompting outperforms the top fine-tuned model by 8.33% points in the BLEU score. In code generation, *GPT-4* outperforms fine-tuned models by 8.59% points on the HumanEval dataset. However, it is outperformed by fine-tuned models by 28.3% points on the MBPP dataset. In code translation, *GPT-4* and fine-tuned models also show mixed results. Analysis of participant interactions reveals significant performance improvements when using conversational prompts compared to automated strategies, with improvements of 15.8% points, 18.3% points, and 16.1% points for code summarization, generation, and translation, respectively. Participants frequently requested improvements, added context, or provided specific instructions to enhance *GPT-4*'s output. Our results indicate that *GPT-4* with conversational prompting holds potential for automated code-related tasks, but fully automated prompt engineering requires further development.

This paper contributes the following:

- The first empirical study comparing automated prompting strategies on *GPT-4* with fine-tuned LLMs for three automated code-related tasks.
- A user study with 27 graduate students and ten industry practitioners exploring the evolution of conversational prompts in automated code-related tasks.
- Identification of gaps between conversational and automated prompts, and suggestions for leveraging LLMs in automated code-related tasks.

- Release of all study artifacts to facilitate replication and extension by other researchers[11].

4.2 Background and Related Work

4.2.1 Fine-Tuning Language Models for Code

The field of automating code-related tasks has increasingly adopted language models (LMs) [242, 243, 180, 244, 245]. These methods offer significant advantages over traditional approaches such as domain-specific models, probabilistic grammars, and simple neural LMs. In recent years, LMs have been applied to various ASE applications, including code completion [246], code search [196], code generation [8, 247], test case generation [17, 248], test oracle generation [249], code summarization [250], code translation [251], and automated program repair [252].

Pre-trained code models learn general-purpose code representations capturing lexical, syntactic, semantic, and structural information. Fine-tuning adapts these models to specific tasks using task-specific data, allowing them to outperform existing baselines. Numerous studies have leveraged pre-training on source code and natural language corpora, followed by fine-tuning for downstream tasks, e.g., code search, code summarization, code generation, etc. [253, 254, 255, 232, 256].

4.2.2 Prompt Engineering in Software Engineering

Prompt engineering is an alternative to fine-tuning that adapts pre-trained LMs without requiring a supervised dataset. Instead, it uses prompts to treat different tasks as generation problems. These models, termed Large Language Models (LLMs), have larger corpora and more parameters. The advent of LLMs and prompt engineering has significantly improved task performance [215, 257, 23, 258, 259, 260, 261]. Studies have explored LLMs and prompt engineering to tackle code tasks with various prompting strategies, such as basic prompting, in-context learning, task-specific prompting, chain-of-thought prompting, auto-prompting, and soft prompting [262, 263, 246, 264, 265, 266, 267, 268, 269, 26, 270, 271].

Gao et al. [263] investigated three key factors in in-context learning for code tasks: selection, order, and number of examples. They found that both similarity and diversity in example selection were crucial for performance and stability. Li et al. [272] studied *ChatGPT*'s ability to find failure-inducing test cases, showing that

[11] `https://github.com/shinjh0849/gpt4_ase_tasks`

performance improved drastically with correct guidance. Feng et al. [264] introduced *AdbGPT*, an LLM-based approach for reproducing bugs using few-shot learning and chain-of-thought reasoning. Kabir et al. [273] analyzed *ChatGPT*'s responses to Stack Overflow questions, finding over half of the answers incorrect and 77% verbose. Geng et al. [265] adopted in-context learning for code summarization, using customized strategies like selection and re-ranking to enhance performance.

Wang et al. [274] compared prompt-tuning and fine-tuning on three code tasks (defect prediction, code summarization, and code translation) using two pre-trained models, *CodeBERT* and *CodeT5*. In contrast, we explored the effectiveness of *GPT-4* using three prompting techniques (basic, in-context learning, and task-specific prompting) against 16 fine-tuned LLMs. The main difference between the studies lies in model size: they assessed models with 220M and 125M parameters, whereas we evaluated *GPT-4* with 1.76T parameters. We also included a qualitative analysis surveying academia and industry participants to explore the impact of different prompting strategies.

Despite growing interest in prompt engineering, comparisons between fine-tuned and prompt-engineered LLMs in ASE tasks remain limited. Fine-tuning modifies pre-trained LM parameters to optimize a task, while prompt engineering crafts natural language queries to obtain the desired output without parameter changes. Fine-tuning can achieve better performance but requires more data and resources [275, 274]. Prompt engineering leverages LLMs' versatility but may face issues like inconsistency and insufficient domain knowledge optimization [276, 275, 277]. To address this gap, we conducted the first quantitative and qualitative comparison of fine-tuning and prompt engineering methods.

4.3 Empirical Study Setup

4.3.1 Downstream Tasks on Code Automation

We select three typical automated code-related software engineering tasks for the experiments, i.e., **Code summarization:** [278, 279, 280]The model generates a short natural language summary from a source code snippet (SC-to-NL), **Code generation:** [281, 282, 283]The model generates the corresponding code snippet from a natural language description (NL-to-SC), and **Code translation:** [162, 160, 158] The model translates a source code snippet into another programming language (SC-to-SC).

These tasks assess *GPT-4*'s ability to generate various modalities (SC and NL). The programming languages differ per task: code summarization involves `Ruby`, `JavaScript`, `Go`, `Python`, `Java`, and `PHP`; code generation targets `Python`; and code

Table 4.1: Stats of dataset used in quantitative analysis

Tasks	Dataset	Language	# of test instance
CodSum	CSN [284]	Python	14,918
		PHP	14,014
		Go	8,122
		Java	10,955
		JavaScript	3,291
		Ruby	1,261
CodTran	CT [220]	C#	1,000
		Java	1,000
CodGen	HumanEval [285]	Python	164
	MBPP [286]	Python	500

translation involves `Java` and `C#`, covering both translation directions. The languages are chosen based on the benchmarks assessed, which are discussed in the following section.

4.3.2 Dataset and Data Process

We use well-known benchmarks for each examined task. The rationale for using these benchmarks is their commonality, enabling comparison with other studies without retraining models, and allowing assessment of generalization across multiple languages. For code summarization and translation, we use CodeXGLUE (CSN and CT) [220]. For code generation, we use HumanEval[285] and MBPP [286]. Dataset statistics are in Table 4.1. Since we do not train a new model, we show only the number of instances in the test sets used for evaluation.

Since *GPT-4*'s training data is not public, it is uncertain if these benchmarks are included in its training. To reduce potential information leakage, we collected sample test sets from GitHub repositories created after Sept. 2021 (*GPT-4*'s end of training) for qualitative analysis. The criteria for selecting repositories included: created after Oct. 2021, written in `Java`, actively maintained, not a fork, and with English comments. We selected three top-rated projects in different domains (library, algorithm, web) and picked two examples per task, totalling 18 sets of problems. The average token length is 14 for NL descriptions and 38 for SC snippets. The selected examples cover different domains, include multi-modalities and vary in difficulty.

We followed common pre- and post-processing for each task. For code summarization, we removed special characters (except commas and periods) and tokenized them. For code generation, we formatted the generated `Python` code to match the datasets. For code translation, we tokenized the generated code using *ctok* in `Python`,

Table 4.2: Baseline Fine-tuned models in three code tasks.

Tasks	Baseline	Pre-trained Model	Param	Fine-Tuned
Code Summarization	PolyglotCodeBERT [256]	CodeBERT [253]	125M	CodeSearchNet (CSN)
	CoTexT [287]	T5-base [240]	220M	
	ProphetNet-X [288]	ProphetNet-Code [288]	300M	
Code Generation	PanGu-Coder2 [289]	StarCoder [290]	15B	HumanEval (HE)
	WizardCoder [291]	StarCoder [290]	15B	
	XCoder [292]	LLaMA3-8B-Base [293]	8B	
	CODE-T-ITER [102]	cushman-001	12B	HE & MBPP
	CODE-T [102]	davinci-001&002 [285]	175B	MBPP
	StarCoder [290]	StarCoderBase [290]	15.5B	
	StarCoder2 [294]	StarCoder2-15B [294]	15B	
Code Translation	StructCoder [295]	CodeT5 based [296]	224M	CodeTrans (CT)
	PLBART [232]	PLBART [232]	140M	

as evaluation metrics are sensitive to tokenization. *GPT-4* sometimes generated non-code text, requiring post-processing to keep only code. To automate *GPT-4* prompting, we used the *OpenAI* API[12], specifying the *GPT-4* model. We set the temperature parameter to zero for better determinism [276]. Metrics were calculated using scripts from the benchmark dataset's repository.

4.3.3 Baselines

Table 4.2 lists the fine-tuned baseline models compared with *GPT-4*. These are the best-performing fine-tuned models reported in each benchmark's leaderboard. Note that some baselines reported on the leaderboard were not publicly available, so we only included publicly available models in the table. The leaderboard can be found from each dataset: CodeXGLUE (CSN & CT)[13], HumanEval[14], and MBPP[15].

4.3.4 Evaluation Metrics

We use the same metrics as the benchmarks to evaluate *GPT-4*. For code generation, we use *pass@k* [285], defined as the probability that one of the top k-generated samples passes the unit tests (with k set to 1). For code summarization, we use *BLEU* [216], which assesses similarity to the ground truth using n-gram precision. For

[12] `https://platform.openai.com/`

[13] `https://microsoft.github.io/CodeXGLUE/`

[14] `https://paperswithcode.com/sota/code-generation-on-humaneval`

[15] `https://paperswithcode.com/sota/code-generation-on-mbpp`

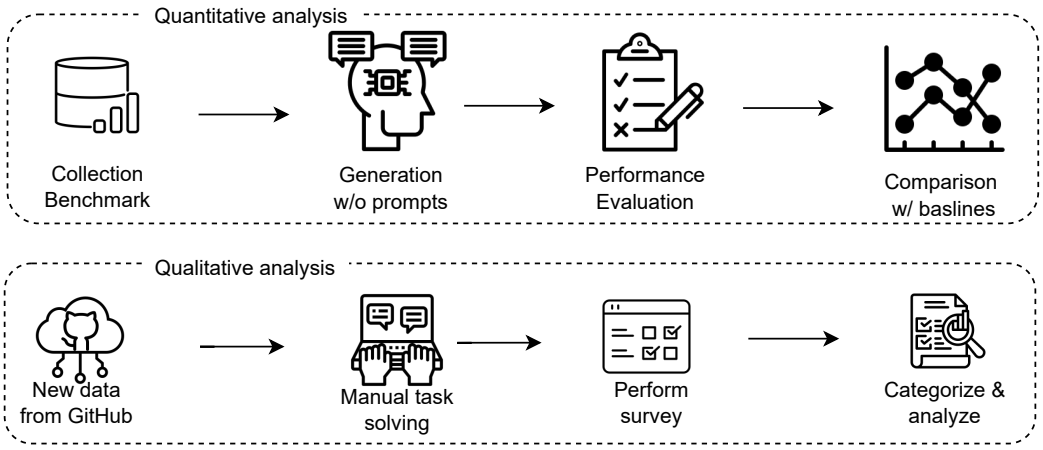


Figure 4.1: Overall workflow of our study.

code translation, we use *BLEU*, *ACC* [297], and *CodeBLEU* [219], which combines n-gram precision, keyword matching, AST matching, and dataflow matching.

4.4 Methodology and Protocols

In this section, we discuss the details of the methodology and the protocol designs for our quantitative and qualitative study. Figure 4.1 shows the overall workflow of this study.

4.4.1 Quantitative Study of *GPT-4* with Baselines (RQ1)

To quantitatively evaluate the performance of *GPT-4* on each ASE task, we utilize well-known benchmark datasets: CodeXGLUE, HumanEval, and MBPP (details in Section 4.3.2). For the baseline on fine-tuned models, we report the scores from the leaderboard of each benchmark and compare them with the evaluation metrics calculated from the results of *GPT-4*. To generate the results from *GPT-4*, we use three different kinds of prompting strategies:

1. **Basic prompting [215, 266]:** we directly query *GPT-4* with the input (code or description) and ask to generate solutions in the form of the desired output.

2. **In-context prompting [267, 265]:** together with the basic prompt, we give a set of input/output examples to *GPT-4*. This idea is very similar to few-shot learning in the context of fine-tuned language models.

3. **Task-specific engineered prompting [268, 269, 298]:** together with the basic prompt, we design additional prompts to guide *GPT-4* in generating better results for each task.

We have selected three prompting strategies covering a range of prompts from simple to more advanced. We use a small subset (between 10 and 50, depending on the task) to evaluate how a crafted prompt results from the actual data (using the BLEU metric). This phase is done to avoid obvious mistakes and not to optimize the prompt for each task. Note that finding the optimal prompting strategy is not the goal of the paper. Although the selections of prompts do not cover all possibilities, our goal is to provide an initial study on automated prompting techniques in three different levels of complexity. We select basic prompting as a baseline or the least complex, in-context learning as one of the best and most generic prompting approaches (at the time of conducting this study) and task-specific prompting as a specialized ad-hoc prompt that potentially outperforms in-context learning. We make the basic prompt as simple and generic as possible to compare how adding different strategies shows/improves different results. Similarly, for in-context learning, we want to keep the prompt as close to the basic prompt as possible and only add different contexts so we can compare how giving examples improves the basic prompts.

Basic Prompting

The basic prompts used for each task are shown in Listing 4.1. They are the simplest prompts that consist of the input code/description and the description of the expected output.

```
### code summarization task
f"Given a {lang} code snippet surrounded in ???, generate one line of semantic focused and
     abstract summarization ???{code}???"

### code generation task
f"Generate Python source code for a function, given
a natural language prompt (surrounded by ???)
describing the function's purpose. Output only the
Python source code on exactly one line. ???{nl}???"

### code translation task
f"Given that the code surrounded by ??? is written in C#, output only the corresponding
     Java code condensed on one line ???{code}???"
```

Listing 4.1: Basic prompt used to query.

In-Context Prompting

For in-context prompting, we add input/output examples on the basic prompt as additional contexts. In-context prompting is one of the advanced types of automated prompts that help LLM learn additional context. We select three input-output examples for the context. A very high number of examples will consume more tokens, which may not fit into the allowed context length, and also be more expensive. To select the context examples, we follow the existing studies [263, 299] that used BM25 similarity-based selection [300]. We calculate the similarity of each test instance's input to each training instance's input and select the top three similar instances for the context examples. The training set here refers to the benchmark datasets, i.e., the dataset in the Table. 4.1. Although we do not use them for training, we use them to select the context examples. It was shown to outperform the random selection or the fixed selection for in-context learning [263, 299].

Task-Specific Engineered Prompts

We created task-specific prompts using simple heuristics to address the challenges we faced with basic and in-context prompting strategies. We manually analyzed a subset of outputs to identify the root causes of issues. We discuss the challenges and how we addressed them with task-specific prompts in the following paragraphs.

For code summarization, we tested *GPT-4*'s initial responses using ten samples and found that the BLEU score decreased due to verbose explanations. To improve the score, we instructed *GPT-4* to limit the output to 15 tokens, the average token length of the training set's outputs. We also found cases where *GPT-4* tries to summarize code for each line of code at the syntax level. To mitigate this, we added "generate one line of semantic-focused and abstract summary of the code" to the prompt. Also,

due to the creativity and immense dictionary of *GPT-4*, it generates synonyms that are not used in the input. To mitigate this problem, we added instructions to guide the model in using naturalized identifiers to form the summary.

For code generation, we found that *GPT-4* had a hard time inferring the steps that were missing in the input (description of code). To mitigate this, we tried to guide *GPT-4* to consider the steps required to solve the described problem carefully. First, we selected five potential prompts. Two of them used in-context prompting, one used chain-of-thought [26], and the final two prompts combined in-context and chain-of-thought. For the prompts with in-context prompting, we randomly selected three samples from the sanitized train set for the MBPP dataset and the test set for the HumanEval dataset. For the samples, we selected 50 examples out of 120 from the sanitized train set for the MBPP dataset and 50 randomly selected examples out of 164 from the test set for the HumanEval dataset. Based on the Pass@1 scores, we chose the best-performing prompt. From the preliminary evaluation, we found that using only the chain-of-thought resulted in the best performance.

For code translation, we selected ten samples to check the initial results. We observed that *GPT-4* fails to generate syntactic keywords that were prevalent in the ground truth, e.g., `virtual` and `override`. However, missing syntactic keywords were easily mitigated by giving more context to the different languages, i.e., input-output examples. Therefore, we decided to modify the in-context prompt to overcome its weaknesses. Five code pair examples were chosen as opposed to three for the in-context prompt, with the rationale being that a wider variety of lengths and keywords would provide better exposure to the "quirks" of the dataset that the first in-context prompt missed (e.g., `C#` code referring to `Java` libraries such as `java.nio.ByteBuffer`). Listing 4.2 shows the resulting task-specific prompt.

```
### code summarization task
f"Generate one line of semantic focused and abstract summary of the code surrounded by
    ???. Compose the summarization by naturalizing the identifier of variables and
    function names in the code as keywords. The summarization should be very concise,
    with an approximate limitation of around 15 tokens in length."

### code generation task
f"Generate Python source code for a function, given a natural language prompt (surrounded
    by ???) describing the function's purpose. Carefully consider the steps required to
    fulfill the function's purpose stated in the natural language prompt. Output only the
    Python source code on exactly one line. ???{nl}???"

### code translation task
f"Given the five examples of translating Java code to C# code (both surrounded by ???),
    translate the 5 Java code samples (surrounded by ???) to C# code and output each C#
    code on exactly one line. Do not include any extra text such as 'C# code sample', and
    do not include surrounding ??? either. ???{5_i_o_examples}???"
```

Listing 4.2: Task-specific prompt used to query.

4.4.2 Qualitative Analysis of Participants' Perception and Interaction with GPT-4 (RQ2 and RQ3)

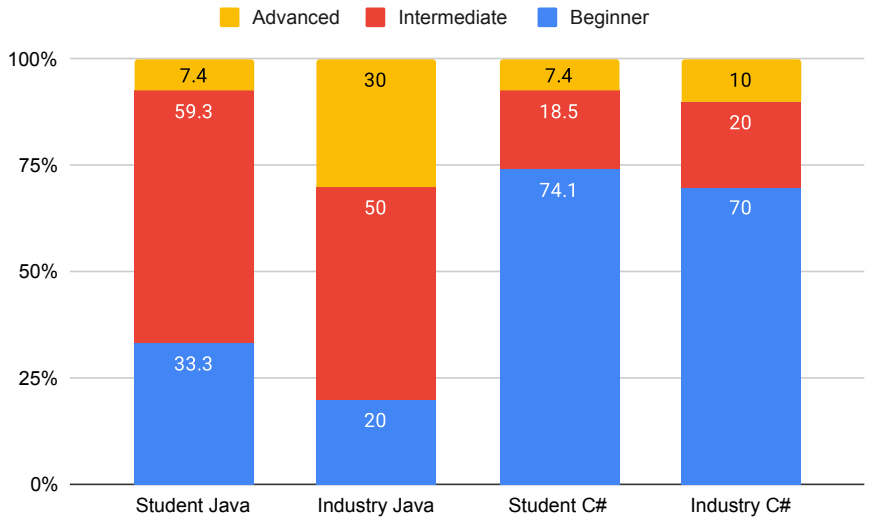
To answer RQ2 and RQ3, we performed a user study by surveying participants from both academia and industry. We first discuss the protocols for sampling and conducting this empirical evaluation.

Survey Participants: We conducted a user study with 27 graduate students (recruited via convenience sampling [301] from an EECS mailing list; 31 respondents, 27 selected for availability) and 10 industry professionals (recruited from 26 invited across Canadian, U.S., and Asian tech companies; 12 respondents, 10 selected).

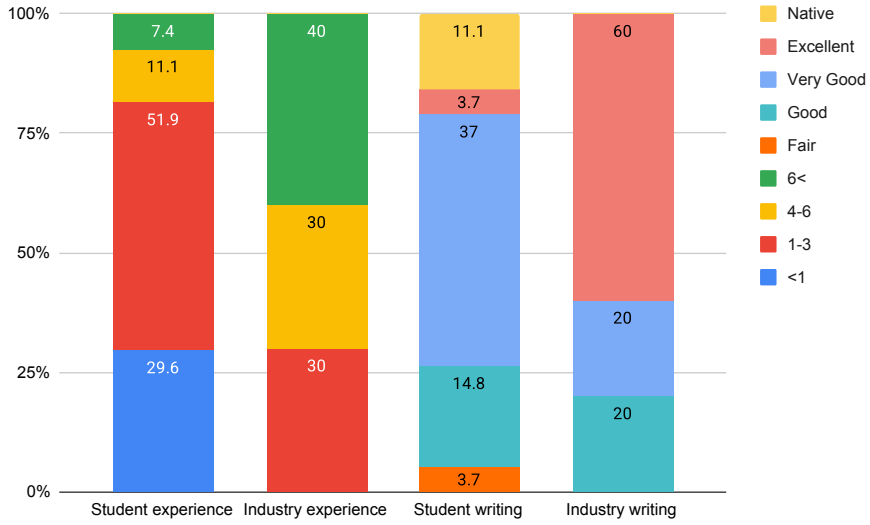
Participants' demographics are shown in Figure 4.2. From Academic participants, the distributions are 77.8% M.Sc., 18.5% Ph.D., 1 Postdoc. From Industry participants, the distributions are 4 senior software engineers, 3 software engineers, 1 associate ML developer, 1 lead engineer, and 1 MLOps lead. For LLM familiarity, 5.4% never used, 37.8% application-level users, 29.7% users with blog/research exposure, 8.1% implemented models, 16.2% active researchers, 2.7% no prior exposure.

Study Design: The survey consists of three primary phases, i.e., the background and demographic survey, user observation, and post-experimentation survey.

Background and Demographic: We ask for participants' demographic information, i.e., years of experience in programming outside of school, degree of familiarity with LLMs, current position, degree of familiarity with the target language (`Java` and `C#`), and the level of technical writing skills in English. We asked six closed-ended questions overall.



(a) Participant programming language proficiency.



(b) Programming experience and writing proficiency.

Figure 4.2: Demographics from the survey.

User Observation Study: We assign two out of three tasks to each participant, considering their familiarity with programming languages. Since participants should have some knowledge about `C#` for the code translation task, we first asked if they

knew `C#` and only gave them the code translation task. The rest of the tasks were assigned randomly. For each task, we gave one sample from each of the 3 projects. The participants worked on the task in a conversation with *ChatGPT-4*. *ChatGPT-4* is a chat-interfaced framework that runs on *GPT-4* [16]. We keep the settings of *ChatGPT-4* to default. We ensured a balanced allocation of task samples to participants randomly (with an equal number of participants per project and task). We ask participants to start with their basic prompt and then proceed with conversations to improve the result based on their first evaluation until they consider the solution satisfactory or have spent 5 minutes per task sample. In this setup, we have not provided any example prompts to avoid any bias and to get participants' fresh perspectives. Overall, we evaluated three tasks over six samples per task. Each sample has been assigned and solved with an average of 12 times by different participants [302]. The survey was designed to take no more than 60 minutes for each participant.

Post-Experimentation Survey: After the observation study, we posed four closed-ended questions to the participants. We inquire whether they received an incorrect response from the model, the degree to which they found *GPT-4* useful for the assigned task, their perception of response quality on the first and last attempt, and their overall assessment. Additionally, we included four open-ended questions to assess the utility of the *GPT-4* model. Strengths and weaknesses of the first and last responses for both tasks.

For RQ2, we report the responses from the four closed-ended questions explained in the post-experimentation survey. We show the percentage of the evaluation responses from all the participants. The evaluation of each survey question can be answered with one out of the five following selections: 1) poor, 2) fair, 3) good, 4) very good, and 5) excellent. We consider the evaluation to be satisfactory if they answer either good, very good, or excellent.

For RQ3, we investigate the chat logs of the participants and manually categorize the conversational prompts. We first extracted the conversational prompts as a list of abstract instructions for the qualitative analysis. We then used an Open Coding Approach, where two authors of this paper, who were not involved in the extraction process, performed sessions of card sorting for our qualitative data analysis [303]. This analysis design was chosen to make the categorization and extraction as independent from each other as possible. After that, a third author acted as a moderator to resolve the disagreements. Finally, all other authors are involved in manually reviewing the forms completed by the three authors and resolving any disagreements that occurred during the process to ensure consensus among reviewers. Note that we also provide the category of conversational prompts and the raw chat logs for each participant in

[16] `https://chat.openai.com/`

our replication package to provide transparent results of our work. After categorizing the types of conversational prompts, we count and rank the occurrences for each task to show the overall trends among the participants.

4.4.3 Analysis of Users' Conversational Prompts (RQ4)

For RQ4, we investigate each prompt created by the participants and the response by *GPT-4*. Since *GPT-4* generates verbose explanations for its response, we manually extract the core part of the answer for each task. For *code generation* and *translation tasks*, we excerpt the code snippets from the generated response. For *code summarization tasks*, we excerpt the texts that summarize the given code snippet. The responses are written in a specific code editor format, allowing us to extract them from the logs manually. After we extract the answers, we rank them based on the BLEU score. For each category of conversational prompt, we calculate the mean reciprocal rank (MRR) to find which type of prompt contributes the most to a higher rank. Lastly, we report the average BLEU score of the conversational prompts, the highest BLEU score achieved by a participant, and the generation from the two automated prompt strategies we used in our quantitative study, i.e., basic prompts and task-specific prompts, to compare the performance of conversational prompts. We did not include the in-context prompt as a comparison baseline for this experiment, as we do not have a separate training set to select similar context examples (as mentioned in Section 4.4.1).

All the survey questions, survey responses, the raw chat logs from the participants, and the analyzed prompt categories by the authors are available in our replication package.

4.5 Experimental Results

4.5.1 RQ1: Automated Prompts vs. Fine-Tuned Models

Table 4.3 shows the results of fine-tuned baselines and *GPT-4* on the code summarization task. The basic prompting improved the 1st-ranked baseline by 1.34% points on average. However, this pattern wasn't consistent across all programming languages, where for `PHP`, we see a decrease of 10.13% points. We observe the biggest improvement from `Go` with 8.09% points. The in-context prompting improved the 1st-ranked baseline by 0.13% points on average. However, the inconsistent pattern was similar to the basic prompt, where a decrease was shown in `PHP` with a 9.52% points and the highest improvement from `Go` with a 7.37% points improvement.

Table 4.3: Result of code summarization. Bold denotes the highest score. The asterisk (*) denotes that the source was not public. GPT-4 + B/IC/TS denotes basic, in-context, and task-specific prompts, respectively.

	BLEU						
Model	**Ruby**	**JS**	**Go**	**Py**	**Java**	**PHP**	**Avg**
StarCoder-LoRA*	17.21	18.15	21.61	23.13	22.61	28.76	21.91
DistillCodeT5*	15.75	16.42	20.21	20.59	20.51	26.58	20.01
PolyglotCodeBERT [256]	14.75	15.80	18.77	18.71	20.11	26.23	19.06
CoTexT [287]	14.02	14.96	18.86	19.73	19.06	24.68	18.55
ProphetNet-X [288]	14.37	16.60	18.43	17.87	19.39	24.57	18.54
GPT-4 + B	19.93	19.96	29.70	27.19	24.11	18.63	23.25
GPT-4 + IC	19.60	20.08	28.98	19.90	24.45	19.24	22.04
GPT-4 + TS	**25.48**	**27.23**	**35.34**	**29.61**	**33.08**	**30.67**	**30.24**

Table 4.4: Result of code generation task.

HumanEval		**MBPP**	
Model	**Pass@1**	**Model**	**Pass@1**
CODE-T (davinci-002) [102]	65.80	CODE-T (davinci-002)[102]	**67.70**
CODE-T-Iter (davinci-002) [102]	65.20	StarCoder2 [294]	66.20
PanGu-Coder2 [289]	61.64	CODE-T (davinci-001)[102]	61.90
WizardCoder[291]	57.30	CODE-T (cushman-001)[102]	55.40
XCoder [292]	57.30	StarCoder [290]	52.70
GPT-4 + B	69.51	GPT-4 + B	39.60
GPT-4 + IC	64.63	GPT-4 + IC	40.60
GPT-4 + TS	**74.39**	GPT-4 + TS	39.40

When comparing in-context prompting to basic prompting, it didn't improve the basic prompting, showing a 1.21% points decrease on average. The lowest decrease was found in `Python` with 7.29% points. The highest improvement was found in `PHP` with 0.61% points. The task-specific prompt had the most noticeable improvement compared to the 1st ranked baseline with an 8.33% points improvement on average. When comparing task-specific to the basic prompt, it improved by 6.99% points overall.

Table 4.4 shows the results for the code generation tasks. The basic prompting was able to improve the 1st-ranked baseline with an increase of 3.71% points for HumanEval. However, for MBPP, it decreased by 28.10% points. The in-context prompting also couldn't outperform the baselines (1.17% points decrease for HumanEval and 27.64% points decrease for MBPP). When comparing in-context prompting to basic prompting, there is a decrease of 4.88% points in HumanEval and a small increase of 1.00% points in MBPP. The task-specific prompting improved from the baseline, with an 8.59% points increase in HumanEval. However, for MBPP, it decreased by

Table 4.5: Result of code translation. Bold denotes the highest score. The asterisk (*) denotes that the source was not public. GPT-4 + B/IC/TS denotes basic, in-context, and task-specific prompts, respectively.

	Java to C#			C# to Java		
Model	**BLEU**	**ACC**	**CB**	**BLEU**	**ACC**	**CB**
StructCoder [295]	**85.02**	**66.60**	**88.42**	80.66	**67.70**	86.03
PLNMT-sys0*	83.37	64.60	87.38	80.91	66.80	85.87
PLBART [232]	83.02	64.60	87.92	78.35	65.00	85.27
CodePALM*	83.26	65.50	86.37	78.94	65.20	83.74
ContraBERT_G*	80.78	59.90	84.98	76.24	60.50	81.69
GPT-4 + B	55.33	5.50	68.34	63.08	20.20	73.57
GPT-4 + IC	80.72	34.10	86.69	**84.12**	50.20	**87.44**
GPT-4 + TS	80.55	34.30	86.87	81.29	49.20	85.33

28.30% points. When comparing task-specific to basic prompting, it didn't show a significant improvement, showing a 4.88% points improvement in HumanEval and a decrease of 0.20% points in MBPP.

Table 4.5 shows the results for code translation. The basic prompting couldn't outperform the 1st-ranked baseline with a decrease of 29.69% points in BLEU, 61.10% points in ACC, and 20.08% points in CodeBLEU for translating `Java-C#`. Similar patterns showed when translating `C#-Java` with a decrease of 17.83% points in BLEU, 47.50% points in ACC, and 12.46% points in CodeBLEU. The low ACC scores when translating `Java-C#` are mainly caused by the failure to predict the frequently used syntactic keywords (e.g., `virtual` and `override`). For example, the keyword `override` appeared in 216, and `virtual` appeared in 431 out of the 1,000 test samples (totalling 647, as no sample used both keywords). Conversely, *GPT-4* generated the keyword `override` in only 48 results and never generated `virtual`. We also observed that it failed to infer code elements that are missing from the input, e.g., parameters in API calls that are not in the source language but are in the target language. Since ACC requires exact matching, a single token could prevent an exact match, severely impacting the overall ACC score. We observed another type of failure, which was caused by the LLM hallucination [304, 305, 306, 307], where the model generates content that is irrelevant, made-up, or inconsistent with the input data. For example, the model would generate APIs that are not used in the source language or even APIs that don't exist. For `C#-Java`, we did not find any noticeable frequently omitted syntactic keyword, which could explain the relatively higher ACC score. A drastic improvement was achieved by in-context and task-specific prompting, especially for generating the missing syntactic keywords. Overall, *GPT-4* with in-context prompting

showed mixed performance, i.e., it couldn't outperform the baselines in translating `Java-C#` while there was an improvement in `C#-Java`. When comparing in-context prompting and basic prompting, it showed a drastic improvement in both `Java-C#` and `C#-Java`. We observed that task-specific prompting outperforms basic prompting. However, it could not outperform in-context learning, as we found in our initial subset when designing the prompt.

Overall, *GPT-4* with different automated prompting strategies could not outperform the fine-tuned models significantly. For code summarization, a reason for this can be due to the verbose and creative generation of *GPT-4*. Since the evaluation metrics heavily rely on textual similarity, diverse and verbose generation will significantly impact the metric scores. When we tried to mitigate this with our task-specific prompt, the result improved by a big margin, outperforming the fine-tuned baselines. For code generation, the code generated by the *GPT-4* seems plausible and executable. However, they fail to pass the tests, e.g., off-by-one error, using imprecise logical operators (choosing between *and* and *or* operator), or failing to generate intermediate steps that are not mentioned in the NL description. A possible reason is that since it is not fine-tuned, it is harder to generate code that requires project-/data-specific knowledge, which could be leveraged to generate code with correct functionality. For code translation, the model failed to generate the same code (low ACC) for both target languages. The possible reasons can be: 1) a much larger and diverse corpus is used for training, resulting in a more creative generation, and 2) generating something irrelevant to the input, i.e., LLM hallucination. Failing to generate similar (low BLEU) `C#` code compared to `Java` code can be due to the difference in data size used for training *GPT-4*, since `Java` is more commonly used than `C#` in GitHub[17].

Answer to RQ1: Automated prompting with *GPT-4* did not significantly outperform fine-tuned baselines, except for code summarization, indicating the need for more sophisticated prompting strategies to fully leverage *GPT-4*'s capabilities.

4.5.2 RQ2: Participants' Perception of *GPT-4*

Figure 4.3 shows how the participants qualitatively assessed the responses of *GPT-4*. The first bar shows participants' prior perception of *GPT-4*'s general usefulness. More than 83.8% of the participants were satisfied (>= Good) with *GPT-4*. Some of the participants (5.4%) have never used *GPT-4* before. The second bar shows if they encountered any incorrect responses on any steps during the study. We could see that almost half (48.6%) of the participants did not encounter any incorrect responses

[17] `https://gitnux.org/github-languages-statistics/`

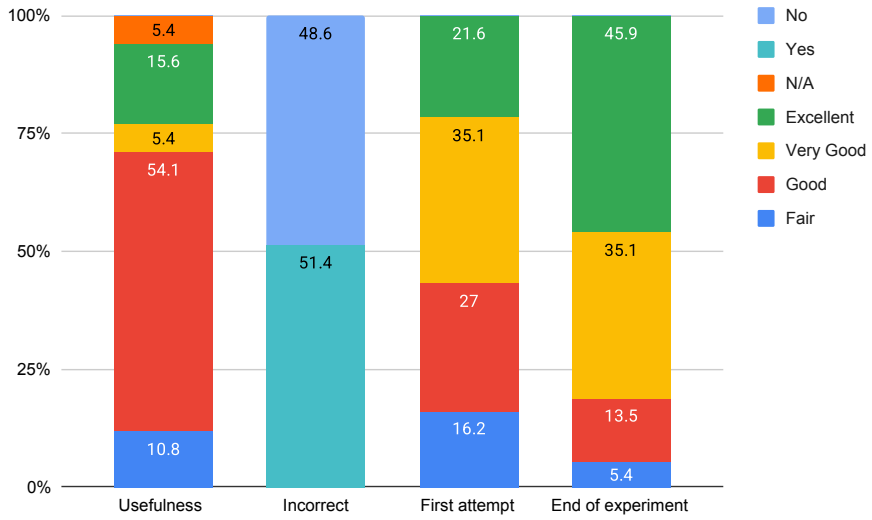


Figure 4.3: RQ2: Qualitative results.

during the study. The third bar shows their evaluation of the responses from the first attempt of the prompts, and the last bar shows their evaluation at the end of the experiment. The evaluations for initial responses were already considerably good with more than 83.8% satisfactory score (>= Good). After applying the conversational prompt, the responses received higher evaluations with a 94.6% satisfactory score (>= Good), an increase of 10.8% points.

We can see that their evaluations change positively from the initial responses to the end of the experiment (excellent +24.3%), indicating that their conversational prompt helped *GPT-4* in generating better responses. This result is also consistent with previous studies that multiple conversational prompting improves the results of LLMs [308, 309, 310].

Answer to RQ2: Our qualitative analysis shows that participants were generally satisfied with *GPT-4*'s responses. Satisfaction increased from 83.8% to 94.6% after using conversational prompts, a 10.8% point improvement, indicating the benefit of conversational prompts.

4.5.3 RQ3: Trend in Forming Conversational Prompts

From analyzing the chat logs, we identified nine types of conversational prompts: (1) Request improvements with certain keywords, (2) Provide more context, (3) Add specific instructions, (4) Point mistakes then request fixes, (5) Ask questions to guide the correct way, (6) Request verification, (7) Request more examples, (8) Request more detailed description, and (9) Request another or a different version of generation.

Table 4.6 shows the categorized conversational prompts and their numbers used for each task. We can see that the trends tend to differ per task. For code summarization, we can see that requesting certain improvements with different keywords was the most common. The keywords for requesting improvement are, e.g., "more natural", "abstract", "semantic-focused", "human-readable", etc. We suspect they requested these improvements due to *GPT-4*'s verbose generations, which generate line-by-line explanations of the code's syntax. Code summarization focuses on automating the technical documentation process, as it is a nontrivial task [311]. Explaining a method's syntax line-by-line would defeat the whole purpose of the task, as developers would not read all the syntax explanations. To get the implicit summary of the method's functionality, we observe that most of the participants request improvements in abstracting and focusing on the semantics. Previous literature also points to the verbosity of LLMs' explanations of code [312, 273]. In addition, some participants requested more descriptions for core logic, where syntax-level explanations are lacking, and added specific instructions to move the line-level explanations to the top of the method to resemble technical documentation, e.g., JavaDoc. Since the output of the code summarization task is a natural language description of code, participants also request the model to generate descriptions with a certain style or format.

Adding specific instructions was found to be the most common for code generation tasks. The instructions have a variety of ranges, e.g., "remove the main method", "remove the import statements", "use a certain type for the variables/parameters", "remove external packages", etc. Close runner-ups were "requesting improvements" and "asking questions to guide the model". The keywords used for requesting improvement in code generation are, e.g., "good format", "readable", "better quality", "most succinct working", "clear", etc. An example for asking questions to the model would be, e.g., "Is the following code snippet syntactically correct?", "Is the generated code executable?", "Is this the most efficient version of code?", etc. These questions aren't the type to ask to perform a specific instruction, but to question the reasoning path of the model to rethink their decision and then make changes if they were incorrect. We suspect these conversational prompts are made as *GPT-4* fails to infer some missing specifications that aren't explained in the abstract text description of the code. To fill these gaps, participants would 1) add specific instructions, 2) ask for

improvement of code quality, or 3) ask questions for verification or to guide the model into a different reasoning path. Generally, our findings are aligned with previous literature [313] in that developers tend to add specific instructions for refactoring (i.e., add/remove certain code elements) or request improvement in code quality from LLMs. "Asking questions for verification to guide the model to correct reasoning paths" is a new pattern we found in this study. We also observed that participants requested to remove irrelevant generations (e.g., important statements, main method, external packages) due to LLM hallucination.

For the code translation task, the most common conversational prompt was to give certain instructions to perform. These instructions include, e.g., "using a certain type", "using a certain logic", "modifying the code to support certain data types", "removing import statements", "handling certain parameters", etc. The runner-up asked certain questions to verify or double-check if the generated code met the specific syntax of the target language. Generally, the results were similar to code generation as their output is both source code. Other commonly used conversational prompts include "adding more context information", "requesting more details or descriptions", "pointing out mistakes and then requesting fixes", etc. A previous study found that the most prevalent translation bug was data-related, i.e., data types, parsing input data, and output formatting issues [160]. We can observe that the participants try to address these problems by adding specific instructions like "use a certain type", "modify to support a certain data type", or "handle certain parameters". We also observe that, similar to the code generation task, they tried to remove irrelevant generations due to LLM hallucination. Previous studies have also reported that more than one-third of translation bugs are due to syntactic and semantic differences between languages. This finding is also in line with our previous findings in Section 4.4.1 that they fail to generate certain keywords specific to the target language. We observe that participants tried to mitigate this by adding specific instructions or asking questions for target language-specific syntax.

Answer to RQ3: Different trends were observed in conversational prompts per task. "Requesting improvements with general keywords" was most common for code summarization, while "adding specific instructions" was most frequent for code generation and translation. Participants used conversational prompts to address known LLM limitations, such as verbosity, missing information, and hallucination.

Table 4.6: Trend: the number of different prompt categories participants used for each task.

Prompts	**ComGen**	**Codgen**	**CodTrans**
Request improvements	12	7	5
Request more description	7	-	-
Add specific instructions	7	8	9
Ask questions to find correct way	6	7	7
Add more context	5	6	3
Request examples	3	1	1
Request verification	2	1	1
Point mistake then request fix	2	5	4
Request another generation	1	-	2

4.5.4 RQ4: Efficacy of Conversational Prompts

RQ4 aims to investigate which category of conversation prompts contributes to better results. Table 4.7 shows the top five ranks of conversational prompts that contribute the most to better results. "Requesting improvement" was shown to be the most effective conversational prompt category for code summarization. "Adding more context information" was shown to improve code generation the most. For code translation, "asking questions to find the correct reasoning path" was shown to have the best performance. We can observe that the ranking of trends and the ranks that best improve the responses are very similar, but not the same. For example, the first rank in trend and improvement for code summarization was requesting certain improvements. For code generation, adding specific instructions was first ranked on the trend ranking but appeared second in the improvement ranking. Adding more context was found to be the first for improvement, but was found to be the third most common in the trends. For code translation, asking questions to find the correct path was the first in improvement, but appears to be the second in the trend ranking. Adding specific instructions was the most common prompt, but it was found to be the third most effective in improvement. From the observation, we can state that the conversational prompt that requested certain improvements was most effective in mitigating verbose and syntax-focused code summarization. For example, "please make summaries more natural and abstract", "improve this by generating semantic-focused and abstract summaries", etc. The prompt resulted in *GPT-4* generating responses that are closer to the ground truth, which are short phrases that summarize the functionality of the method. For code generation, adding more context was most effective to mitigate the limitation of *GPT-4* in inferring the missing steps that the abstract text description encapsulates. For example, they would add the

class information, e.g., class signature, description of what the class does, and other public method signatures within the class. The model was able to infer the missing steps by the project- or class-specific knowledge provided by the additional context. For code translation, asking questions to find the correct way was the most effective in generating the correct syntax of different languages and data types. For example, "what about the @Parameter annotation?", "The JSON.parseArray function needs to know the generic type T to know the format of the return value. How should I modify the code?", etc. As the examples show, they would ask questions to guide the model in resolving the data issue.

The similarity of the two ranks can be interpreted as participants having a good knowledge of how to guide the models for better results, even though the ground truth wasn't provided. It is interesting to find that "requesting improvement" actually helps generate better responses, even though these prompts do not necessarily provide specific knowledge on how to improve.

We also report the average BLEU score from the responses of conversational prompts in Table 4.8. We can see that the average BLEU score of all conversational prompts is comparable to the two automated prompts, i.e., basic and task-specific prompts. However, only the code translation task outperforms the better-automated prompt, i.e., task-specific prompt, by a mere 0.27% points. The reason is due to the large variance caused by the different types of conversational prompts. The conversational prompt that produces the highest BLEU score outperforms the automated prompts by a big margin, 15.8% points for code summarization, 18.3% points for code generation, and 16.1% points for code translation tasks.

Answer to RQ4: Overall, conversational prompts can improve code tasks. "Requesting improvement" worked best for code summarization, "adding context" for code generation, and "asking questions" for code translation. The rank of trends and improvements showed similar patterns, indicating participants effectively guided the models. The best conversational prompts significantly outperformed automated prompts, improving by 15.8% points, 18.3% points, and 16.1% points for code summarization, generation, and translation, respectively.

4.5.5 Discussion

In this section, we discuss the main findings of this study in terms of prompt engineering and summarize them as recommendations for future work. Then, we discuss the trade-offs between prompt engineering and fine-tuning.

Table 4.7: Improvement: Top-5 prompt categories that effectively improve the basic prompts.

Rank	Code summarization	Code generation	Code translation
1	Request improvements.	Add more context.	Ask questions.
2	Add more context.	Add instructions.	Point mistake, then fix.
3	Ask questions.	Request improvements.	Add instructions.
4	Add instructions.	Ask questions.	Request improvements.
5	Request verification.	Point mistake, then fix.	Add more context.

Table 4.8: Result of conversational prompts (conv.) vs automated prompts. Bold indicates the highest score.

	BLEU			
Task	**Average conv.**	**Highest conv.**	**Basic**	**Task-specific**
CodSum	36.22	**55.75**	35.76	39.95
CodGen	53.15	**69.16**	57.33	50.85
CodTran	67.43	**83.26**	52.10	67.16

As the quantitative results indicate, even though *GPT-4* is a larger model than its baselines, it does not always yield better results. Even typical prompt engineering strategies, such as in-context learning, did not show significant improvement. On the other hand, the qualitative user study revealed that the key potential of *GPT-4* is on conversation-based prompting, which includes human feedback in the loop.

There are two takeaway messages from our results: (a) To leverage LLMs to their best, a sequence of back-and-forth queries with the model may be needed. Iterative processes such as search-based optimization or reinforcement learning-based agents might be useful to "engineer" prompts for ASE tasks with full automation. (b) Human feedback still plays a crucial role in optimizing the prompts, as shown by conversational prompting. Thus, to remove humans from the loop completely, future studies are needed to analyze developer-written prompts more carefully and extract patterns that can be implemented as rules, fitness functions, rewards, and policies.

To summarize our findings, we conclude that, like most engineering problems, the choice between prompt engineering and fine-tuning LLMs is a trade-off. The following are some of the main factors that play a role in the trade-off:

- **Performance:** The most explicit aspect is the performance of the models. From the results, neither approach always dominates the other. However, conversational prompt engineering has been shown to have great potential.

- **Cost:** Although fine-tuning is usually considered more expensive than prompt engineering, depending on the subscription models of the commercial LLMs, inference fees can quickly become a huge burden. For example, running *GPT-4* for this study costs around 1,500 USD. Therefore, developers who have a large number of instances for inference should consider using a cheaper model or even fine-tuning a smaller in-house model.

- **Ease of Use:** Prompt engineering has a significant benefit from this aspect, especially when using natural language-based interfaces. It becomes more significant as we see participants with little to no background in LLMs can perform ASE tasks with *ChatGPT-4*. In contrast, they would not be able to perform with fine-tuned models without certain knowledge of deployment, configuration, and developing fine-tuned models.

- **Control:** Finally, another aspect of the trade-off is the level of control on the internal configurations and outputs. For example, with *GPT-4*, although you can set the temperature to zero, you still get non-deterministic answers, which makes the verification of the models more difficult and thus less interesting to some users.

4.6 Threats to Validity

Internal Validity: A key threat is whether the designed prompts truly reflect the intended strategy or if better approaches could exist. For example, poor results of *GPT-4* with in-context learning might be due to confounding factors, such as suboptimal sample choices or vocabulary. To address this, we reported the exact prompts used and conducted a qualitative survey where participants created their own prompts, providing a broader view of prompt engineering effectiveness. Another potential threat to validity arises from not explicitly examining biases such as task order and participant learning effects, though we mitigated order-related risks by randomizing task sequences across participants. The implications of participant segregation, while a compelling avenue for future analysis, were beyond the scope of this work due to brevity and will be explored in future research.

Construct Validity: We used common metrics, such as BLEU scores, as in the original benchmarks. However, these metrics may not fully capture what they aim to measure. In the qualitative study, participants' evaluations of *GPT-4*'s responses might be subjective. To mitigate this, we also performed a manual analysis by three

co-authors. Investigating better human-aligned metrics remains an open area in the field.

Conclusion Validity: The non-deterministic behaviour of LLMs [276] means results may not be reproducible. We set the temperature parameter to zero to increase determinism, but acknowledge this does not eliminate the issue. We provide generated responses from the quantitative study and raw chat logs from the survey to address this. A further limitation lies in the absence of statistical tests for quantitative comparisons, as fine-tuned model baseline distributions were unavailable (benchmark scores were sourced directly from leaderboards), and the qualitative analysis's limited sample size (6 samples per task) precluded meaningful statistical testing. While these constraints temper the robustness of our conclusions, we intend to address them in future work through expanded datasets and formal statistical evaluations.

External Validity: We used recent state-of-the-art fine-tuned models for each task, but new baselines could alter our observations. Although we used widely accepted benchmark datasets, they do not represent all domains. For the survey, we used 18 samples to limit participant time, acknowledging that this limits generalizability. We chose a 1-hour survey duration to recruit more participants, balancing practical constraints like participant availability and budget. To mitigate this threat, we considered diverse domains and difficulty levels in the examples. We also included three different project types (library, algorithm, and web) to cover different domains. While in-context learning was one of the best automated prompting strategies at the start of our study, newer methods like Retrieval Augmented Generation (RAG) [314] and Self-Reflection [315] could affect our findings, warranting further study.

4.7 Conclusion

In this paper, we evaluated *GPT-4* using basic, in-context learning, and task-specific prompts and compared them with fine-tuned language models. We also conducted a user study with 27 academic and 10 industry participants to assess the quality of *GPT-4*'s responses and examine how participants designed conversational prompts. Our quantitative results show that *GPT-4* does not significantly outperform the baselines. Qualitative results indicate that participants often request improvements, add context, or give specific instructions, which helps *GPT-4* generate better responses. Our study suggests that *GPT-4* has great potential for code tasks but requires careful human verification and interpretation.

Chapter 5

Fine-Tuned Models vs. LLMs in Test Oracle Generation

5.1 Introduction

Unit testing is considered a standard procedure when developing a software product. The primary goal of unit testing is to verify whether a unit behaves as expected. However, writing an effective unit test requires non-trivial effort for developers. To mitigate this challenge, there have been numerous recent studies to automate the process by exploiting deep neural generation models [316, 317, 318, 17]. These models do not consider the specific state (behaviour) they want to test or verify (oracle), and mainly focus on catching higher-level exceptions or crashes of the software. Recently, researchers have further proposed test oracle generators based on deep learning and large language models (LLMs), which aim to generate meaningful assertions or exception-handling logic for a method under test by providing the state they want to verify [319, 320, 321, 86, 322]. We call these approaches neural oracle generation (NOG).

Textual similarity metrics such as BLEU [216] and exact match accuracy [323] have been widely adopted to evaluate NOG models' performance. These metrics are calculated automatically and statically based on the textual similarity of the generated oracle and the ground truth oracle. Due to the ease of collecting benchmarks (no test case execution is needed) and the cost efficiency in the calculation (string similarity calculation), most studies adopt these metrics to evaluate the performance of generated test oracles. Although these generic metrics are useful to evaluate the quality in readability and maintainability of tests (similarity with human-written),

their benefit in special domains, such as test adequacy of unit test cases and test oracle generation, remains unclear.

Although the current field of NOG only considers textual similarity metrics, test adequacy metrics, i.e. code coverage and mutation scores, were widely used to evaluate generated test cases and oracles [324, 325]. These metrics evaluate the effectiveness of unit tests and oracles in their completeness of test coverage and fault detectability. Test adequacy metrics are dynamic, meaning they are only obtainable by executing the software under test and the test suite, and thus cannot be evaluated using static/automated textual similarity metrics. These are the currently well-known metrics that can evaluate the actual capability of tests in the literature [326, 327, 328].

Problem Statement: Despite the wide use of textual similarity metrics in evaluating NOGs, no study confirms their correlation with the generated oracle's testing capabilities. Without such a study, there is no guarantee that a generated oracle with higher BLEU (but not 100%) is "better" (i.e., covering the same functionality and potentially detecting the same bug as the ground truth oracle) than one with lower BLEU. The oracle might be textually similar to ground truth, but functionality-wise, it verifies a completely different assertion. In other words, if the textual similarity (i.e., BLEU-type metrics) and the test adequacy metrics have a low correlation, it would be a problem to only consider the textual similarity metric as the primary evaluation criterion for NOGs. This paper aims to fill this gap and comprehensively analyze the relationship between dynamic test adequacy metrics and static textual similarity metrics for NOGs.

```
// Test prefix
testFileManagerNoFile () {
    FileManager fileManager = FileManager.create();
    fileManager.addLocatorFile();
    try {
        InputStream in = fileManager.open(filenameNonExistent);
        closeInputStream(in);
        "<AssertPlaceHolder>";
    } catch(NotFoundException ex) {
    }
}
// Focal method
private void closeInputStream(InputStream in ) {
    try {
        if ( in != null ) in.close ( ) ;
    } catch ( Exception ex ) {
    }
}

// Ground truth oracle
assertNull("Found non-existent file: " + filenameNonExistent, in)

// Generated oracle by GPT-3.5
assertEquals(null, in)
```

Listing 5.1: An example of oracle generation.

Listing 5.1 shows an example of when these two metrics do not agree with each other. The listing shows an input and output example of an oracle generated by *gpt-3.5*, i.e., an LLM developed by *OpenAI* which is a widely used generative model in various domains including software engineering [329, 17]. The task of a NOG is to generate an appropriate oracle that will be substituted with "`<AssertPlaceHolder>`" when an aggregation of the test prefix and the focal method is given as input. As we can observe, the generated oracle and the ground truth oracle function have the same functionality (i.e., check whether '`in`' is null) but differ in the type of assert statement used. We can consider that the model generated a correct oracle with the same functionality and testing capability, leading to the same code coverage and mutation score. However, the textual similarity metrics cannot capture their capability, resulting in a low BLEU score of 0.21. Such examples call for a more thorough investigation of using textual similarity metrics and their correlation to test adequacy metrics in evaluating oracle generation tasks.

Before delving into the correlation between the two metrics, we first investigate current NOG literature to find their performance in various textual similarity metrics, i.e. BLEU, CodeBLEU, ROUGE-L, METEOR, Accuracy, Edit Similarity, and ChrF. Then we check the testing effectiveness of the oracles by measuring the test adequacy metrics, i.e., line coverage and mutation score. After, we calculate the correlations between pairs of evaluation metrics (seven textual similarity metrics and two dynamic

test adequacy metrics). Finally, we perform an ablation study on some of the results to further analyze why we observe such discrepancies in the two types of metrics.

The results show that the correlation between these two evaluation metric categories is insignificant. Our manual analysis found cases where the static metrics show an oracle is textually similar to a developer-written oracle but are quite different semantically (their line coverage and mutation scores are very different). Or vice versa, the static metrics suggest the two oracles are syntactically very different, but semantically very similar (e.g., same line coverage and mutation score). These disagreements were the root cause of the low correlation between the two metrics. Thus, we claim that researchers should not only consider textual similarity metrics, with the assumption of a high correlation between the textual similarity metrics and the test adequacy metrics.

In summary, our main contributions are as follows:

- We revisit the performance of four state-of-the-art neural oracle generation models on two categories of metrics, i.e., static textual similarity metrics and dynamic test adequacy metrics.
- As far as we know, this paper is the first to show that there is no significant correlation between the two categories of metrics, which shapes future research in this domain.
- We provide a manual analysis of the findings and provide justifications on why there is a mismatch between static and dynamic metrics in this domain, based on our observations.
- We provide a new benchmark for evaluating and studying NOG models, with almost 30K executable methods under test and their corresponding metrics from 13 open-source projects. We also release our experiments' dataset and source code to help other researchers replicate and extend our study[18].

The rest of this paper is organized as follows. Section 5.2 presents the background. Section 5.3 shows the experimental setup. Section 5.4 presents the evaluation results. Section 5.5 discusses the threats to the validity of our study. Section 5.6 presents the related studies. Section 5.7 concludes this paper.

[18] `https://github.com/shinjh0849/assessing_ntog`

Table 5.1: Performance of NOGs in the original papers.

NOGs	BLEU	ACC
ATLAS	61.85	17.66
IR Old	**78.86**	46.54
IR New	60.92	42.20
TOGA	-	**69.00**

5.2 Background

5.2.1 Current Evaluations of NOGs

The current literature on NOG mainly focuses on the JUnit frameworks, a `Java` unit testing framework. Here we list the most closely relevant studies in NOG that generate assertion oracles when the test prefix and method under test are given. *ATLAS* [319] was the first study to exploit a deep neural generative model for generating assertion oracles. They have evaluated their models in two different settings: 1) the abstract data, where they normalize all identifiers, e.g. IDENT_0/1/2 and METHOD_0/1/2, and 2) the raw data, where they keep all the raw identifiers. They have used exact match accuracy and the BLEU-4 score to evaluate the performance. They also investigated what types of assertions were generated by the models, e.g. `asserEquals`, `assertNull`, `assertThat`, etc. Yu and Lou et al. [320] extended the former by integrating simple information retrieval techniques (abbr. *IR*), i.e. Jaccard coefficient [330], Overlap [331], and Dice coefficient [332]. They have used the same evaluation metrics from the former, i.e. accuracy and BLEU-4. *TOGA* [321] exploits a unified transformer-based neural model to generate exception handling and assertions for unit test case oracles. It first generates oracle candidates using type constraints for assertion types and forms vocabularies by observing the variables accessible from the test for the parameters. Then it ranks the best candidate using CodeBERT [253], choosing the top-1 as the generated oracle. These proposed methods have shown great potential in NOGs using textual similarity metrics, ranging from 17.66% [319] to as high as 96% *TOGA* in accuracy according to their reports. In Table 5.1, we report the static textual similarity metrics used in the relevant studies, i.e., BLEU and accuracy score.

5.2.2 Test Adequacy Metrics

In software testing [324, 325], the effectiveness of a unit test or an oracle is evaluated using test adequacy metrics, e.g., code coverage and mutation score. Code coverage

metrics evaluate how much the software under test is exercised [333, 334] by the test case. When software test suites have low coverage, some parts of the software remain untested, leading to incomplete testing. However, high coverage (e.g., covering all lines of code) does not guarantee that the test cases detect all bugs in that code. Different code coverage metrics exist, e.g., function, line, statement, condition, branch, path, data-flow coverage, etc. For example, if a test case exercises 5 lines of software under test with a total number of 10 lines, the line coverage of the test case will be 50%.

Another category of test adequacy metrics is fault-based coverage criteria. Unlike coverage-based metrics, fault-based metrics focus on detecting faults (defects). A test is effective if it finds all defects in the code. This is the ultimate metric for any testing approach, but given that the defects are typically not known while testing, in practice, fault-based testing must simulate defects. There are many approaches for this, for example, seeding old bugs into code or manually adding likely bugs, but the most systematic way is mutation testing [335, 336]. In mutation testing, bugs are seeded into code as small changes, syntactic changes, or mutations. This can be done automatically by defining mutation operators that systematically insert bugs in the code wherever applicable. A mutation score is the corresponding adequacy metric that calculates the test cases' ability to identify (kill) those seeded faults (mutants). For example, if 10 mutations were applied to the software under test, and the test cases reveal (kill) 5 mutants, the mutation score is 50%.

Although covering all the code or killing all mutants still does not guarantee bug-free software, these two metrics, so far, are the best evaluation methods in software testing, which target evaluating the completeness and effectiveness of the generated test cases/oracles. These properties of tests can only be assessed with dynamic execution and thus cannot be evaluated from generic static/textual similarity metrics, which assess only the syntactic similarity (and not semantic or functionality). One simple example of syntactic vs. semantic similarity is when two oracles can be syntactically very similar (e.g., only one character difference; one testing a method with 0 and one with 1, as one of the method's parameter values), but semantically very different (e.g., the zero case covers a very different path in the control flow graph that results to a failure). Thus, a purely syntactic metric will have a hard time distinguishing between the two oracles, and if used as the only metric for NOG, will result in creating sub-optimal oracles with high BLEU values. Therefore, at best, most NOG literature that only evaluates textual similarity metrics implicitly assumes a strong positive correlation between dynamic test adequacy metrics and textual similarity metrics. However, there is no study to support this assumption.

5.3 Experimental Design

In this section, we explain the neural oracle generation models, our dataset, the evaluation metrics, the correlation analysis used in the paper, our research questions, and their corresponding procedures.

5.3.1 Neural Oracle Generators (NOGs)

We select three state-of-the-art (SOTA) NOG models based on the following criteria:

- Should be a neural oracle generation model proposing a novel approach for oracle generation.
- Should be published within 3 years to represent SOTA.
- The replication package should be publicly available.
- The study should report evaluation results of their generated oracles' quality (using any comparable metric).

As a result, three NOG models were selected from 13 candidate papers published on main software engineering and testing venues (ICSE, FSE, ASE, ISSTA, ICST, AST, TSE, TOSEM, EMSE, IST, and arXiv). Out of 13 candidates, three baselines were selected as 1) two papers did not have a public replication package [337, 86], 2) two used oracle generation as a downstream task on a pre-training model (not a novel approach for oracle generation) [239, 338], 3) four focused on a different task, i.e., test completion [339], bug reproduction [340], enhancing SBST with LLM [88], and whole unit test case generation [322], and 4) two evaluated with a different perspective, i.e. oracle ranking [341] and predicting passing/failing oracles [342]. We included *gpt-3.5* as a baseline for SOTA LLM, which has not been applied to NOG at the time of conducting this study, but is deemed relevant. The four selected NOGs in this study are the following:

- **ATLAS** [319]: is the first study to apply a deep neural generation model on oracle generation. It exploits a sequence-to-sequence encoder-decoder (RNN) model to learn and automatically generate an oracle when the test prefix and the focal method are given as input.
- **IR** [320]: extends the previous study *ATLAS* by leveraging information retrieval techniques and integrating deep learning to enhance performance. They also expanded the evaluation by adding unknown vocabulary to make the oracle generation problem more challenging.

- **TOGA** [321]: proposed a unified transformer-based neural model for exceptional and assertion oracles. They first identify if the test method should raise an exception or generate an assertion. If it decides to raise an exception, it generates a try-catch clause. If not, it generates candidate assertions with different parameters. Then, an assertion oracle ranker decides the best candidate for generating the assertion.

- **gpt-3.5** [343]: is a generative large language model (LLM) developed by OpenAI. *gpt-3.5* is one of the most widely used LLMs. We wanted to investigate the impact of LLM as it has shown great potential in generating source code for different software engineering tasks [322, 329, 17].

5.3.2 Dataset

As all three NOG models evaluate their performance with the ATLAS dataset [319], i.e., a set of Java projects with test cases, we reuse the same dataset to facilitate a direct comparison between their results and ours. We do not investigate additional datasets as we deem this dataset comprehensive, i.e. covers over 9K projects in various domains with more than 320K instances. The dataset is a parallel corpus of test prefixes plus the focal method paired with the oracle. However, to calculate the test adequacy metrics, we have to execute each test method and its corresponding focal method. To execute them, we need the package information of the projects from which they were originally mined. In the replication package of the *ATLAS* paper, they provide the resulting raw dataset of test-oracle pairs and the project lists they used for mining. Since the script that constructs the original dataset was not publicly available and each instance was randomly shuffled, retrieving the package information was not possible. So, we replicate and re-implement the *ATLAS* dataset by following the steps reported in their papers.

The steps to mine the *ATLAS* dataset are as follows: First, we clone the project list that is publicly available in the original *ATLAS* paper. We extract all methods with the @*Test* annotation, which is inherently used by the JUnit framework for unit test cases. For retrieving the oracle, we parse the test methods and look for the specific invocation of assertion APIs, e.g., assertEquals, assertTrue, assertNull, etc. To parse the corresponding focal method, we first extract all methods declared within the project. Then we apply a heuristic to iterate over all the invoked methods within the test method. The invoked methods are queried from the list of declared methods in the project. Then we get the last matched method before the oracle statement, assuming it's the focal method. Note that if there is a method invoked within the assertion parameter, we take the method as a focal method instead. Test methods

Table 5.2: Experiment data

	Training	Validation	Test	Total
#of instances	261.7K	32.4K	29.7K	323.9K
Avg. input tokens	78.96	78.39	85.42	79.99
Avg. output tokens	13	12.87	13.03	12.99
Unique input tokens	706.5K	110.7K	96.4K	851K
Unique output tokens	172.7K	29.2K	22.9K	204.6K

with more than one assert line are filtered as *ATLAS* only focuses on single-line assertion generation, as well as duplicate instances.

After retrieving the test-oracle pairs, we split the train, validation, and test sets at a project level so that instances in the same project are not in different splits to remove potential information leakage within the projects. We randomly shuffle the projects and assign them to the train-valid-test set with a ratio of 8:1:1, which is the same ratio as the *ATLAS* dataset. For evaluating projects on an execution level, we use a subset of the test set comprised of 13 projects, as not all the projects were deployable. We considered the raw version of *ATLAS*, where we keep identifier names since it is a more challenging problem and *IR* and *TOGA* only consider them as well. We also adapt the ideas of *IR*'s new dataset that considers unknown vocabularies to better reflect the data distribution of the real world [320]. We didn't consider *Methods2Test* used in *TOGA* as we only focus on assertion oracles rather than exceptional oracles or whole unit test case generations. As shown in Table 5.2, after re-implementing the data construction, we resulted in a total of 323,994 test-oracle pairs, 261,794 for the train set, 32,476 for validation, and 29,724 for the test set.

5.3.3 Evaluation Metrics

Textual Similarity Metrics

Existing oracle generation methods used textual similarity metrics, which were originally adopted from the natural language processing (NLP) field to assess the performance of their models. The three NOGs used BLEU and Accuracy to assess their models. However, researchers pointed out that BLEU and Accuracy are not suitable to assess code generation models due to the distinct difference between natural language and source code [219, 344]. For a more comprehensive experiment, we have also added other metrics, i.e., CodeBLEU, METEOR, ROUGE-L, Edit Similarity, and ChrF, which are widely used to assess code generation models. Note

that all these metrics are textual similarity metrics that compute a similarity score between the generated oracle and the ground-truth reference. All the studied metrics range between 0 to 1, with 0 being a complete mismatch and 1 being a perfect match. However, we report them as rates with percentage points (0 to 100%).

1. **BLEU-4 (abbr. BLEU)** [216] is a widely used evaluation metric that evaluates the text quality generated by the model. It is calculated by an n-gram precision, which is the number of matching n-grams from the generated and the ground truth text. We use 4-gram, i.e., BLEU-4, as it is the most widely used and also used by the baseline approaches in this work.

2. **Accuracy (abbr. ACC)** [323] is the number of true predictions over the total number of instances. The generated text is a true positive only if it is an equivalent text, i.e. exact match, to the ground truth.

3. **CodeBLEU (abbr. CB)** [219] is an evaluation metric specifically designed for code generation models. It is calculated by the combination of a weighted n-gram precision (token similarity), AST matching (syntax similarity), and data flow matching (semantic similarity).

4. **ROUGE-L (abbr. RL)** [218] is another metric used in similarity-based metric. Out of the different ROUGE values, we report the ROUGE-L value, which exploits the Longest Common Sub-sequence (LCS) in evaluating the similarity of the texts.

5. **METEOR (abbr. MT)** [217] is also commonly used to evaluate textual similarity. The metric is calculated by the harmonic mean of unigram precision and recall, but with a higher weight on recall.

6. **Edit Similarity (abbr. ES)** [345] is a metric that implements Levenshtein distance to measure the distance between two sequences by calculating the smallest number of edit operations (i.e., insertion, deletion, and replacement) required to transform one string into another.

7. **Character n-gram F-score (abbr. ChrF)** [346] focuses on character-level n-grams, i.e., a sub-sequence of characters. It is calculated by the harmonic mean of how many n-grams are correct (Precision) and how many reference n-grams are covered (Recall).

We used a Python script to calculate all the textual similarity metrics. We calculated BLEU and METEOR using the `nltk` package. We used (0.25, 0.25,

0.25, 0.25) as the weights for BLEU and a basic smoothing function to match previous studies. For ACC and ChrF, we created a simple script without any packages. For CodeBLEU, we used the script provided by CodeXGLUE [220]. We used the `rouge_scorer` package to calculate ROUGE-L. For Edit similarity, we used `SequenceMatcher` in the `difflib` package. All the non-mentioned parameters were set to default.

Test Adequacy Metrics

Following existing work, we use line coverage and mutation score in this work to measure the test adequacy of generated test oracles [347, 348, 186]. Note that these metrics are calculated by executing the project (thus dynamic) to evaluate the adequacy of the generated tests. All range between 0 (not adequate at all) and 1 (perfect adequacy). We also report them as percentage points (0 to 100%).

1. **Line Coverage**: is a common coverage metric to measure the number of lines covered when a test case is executed. It shows the test case's strength regarding its completeness in exercising the source code line by line.

2. **Mutation Score**: is a fault-based metric to measure the quality of the generated test case by measuring how it can effectively detect synthetic faults (code mutants). The ability to detect a mutant means that the tests are likely to be effective at catching real faults.

5.3.4 Correlation Analysis

To measure the correlation between each pair of metrics (i.e. one textual similarity and one test adequacy metric), we have chosen two correlation analysis methods, namely Spearman's and Kendall's rank correlation analysis. We chose these two methods because we need a statistical test that can be applied to non-parametric distributions. Spearman's and Kendall's rank correlation measures the correlation between two ranked variables. They are non-parametric measures of statistical dependence between two variables and work on non-normal distributions. The reason for using non-parametric measures is that we cannot guarantee a normal distribution of the variables, where parametric tests assume them to be normally distributed. If parametric tests are applied to non-normally distributed variables, it can lead to inaccurate results. Spearman's correlation is calculated based on deviations from the mean ranks, while Kendall's correlation is calculated based on concordant and discordant pairs of ranks. Previous studies have commonly used these methods

for correlation analysis on evaluation metrics; thus, we follow their methodology [349, 350, 351, 352, 353].

5.3.5 Research Questions

We design experiments to answer the following research questions:

RQ1: How do current NOGs perform based on textual similarity evaluation metrics? In the literature, NOGs are evaluated with two metrics, i.e. Accuracy and BLEU score. We added five other majorly used textual similarity metrics, i.e. CodeBLEU, ROUGE-L, METEOR, Edit Similarity, and ChrF, for a more comprehensive empirical study. Thus, this RQ will show a thorough comparison among SOTA NOG models regarding seven different textual similarity metrics.

RQ2: How do current NOGs perform based on test adequacy evaluation metrics? RQ1 compares different baselines based on the quality of the generated oracles measured by the textual similarity metrics. RQ2 repeats this experiment with dynamic test adequacy metrics. Since the goal of a NOG is to create an oracle as close as possible to a developer-written oracle, we measure how close the line coverage and mutation score of the generated oracles are to the developer-written oracles. Thus, the main motivation behind RQ2 is whether we see the same trends when comparing baselines as RQ1.

RQ3: What is the correlation between textual similarity and test adequacy metrics? The main motivation of RQ3 is to come up with a recommendation for the research community on which static textual similarity metrics (if any) can safely be used as a surrogate measure for expensive adequacy metrics when evaluating their NOGs. A high/low correlation between a textual similarity metric and any test adequacy metric will be used as evidence of whether the textual similarity metric should be used in future research in this domain.

RQ4: What are the reasons behind the correlation in RQ3? To further analyze the root cause of correlations in RQ3, we manually analyze samples with mismatched correlation coefficients. Looking at these examples, we want to find out why some of the generated oracles have high scores in one metric category but low in the other. The goal is to provide justifications, based on the observed patterns, for the recommendations given in RQ3.

5.3.6 Experiment Procedure

Experiment Setting for RQ1: We assess the textual similarity evaluation metrics, i.e. BLEU, Accuracy, CodeBLEU, ROUGE-L, METEOR, Edit Similarity, and ChrF, on the studied NOGs. To generate oracles from the existing NOGs, i.e., *ATLAS*, *IR*, and *TOGA*, we download the publicly available replication package shared in the paper and follow their instruction to train and run these tools on our dataset. We also report the BLEU and Accuracy scores reported in the original paper for comparison.

For *gpt-3.5*, following existing work [322], we feed a single basic query (zero-shot) to suggest a single line oracle be replaced in the oracle placeholder given the text prefix and its focal method. We used *gpt-3.5* APIs for the model version as it was the latest model accessible to us during the time of our experiment.

We expect there would be a noticeable gap in the textual similarity metrics between our evaluation and the originally reported evaluation, as we have divided the train-valid-test splits so that instances from the same project would fall into the same split. This is to mitigate any possible project-specific information leak between the train and the evaluation sets [17].

Table 5.3: The number of injected oracles generated by each NOG. Ms denotes the number of test methods, Cs denotes the number of affected classes, and Ss denotes the number of affected sub-modules.

	ATLAS			***IR***			***TOGA***			***gpt-3.5***		
project name	**Ms**	**Cs**	**Ss**	**Ms**	**Cs**	**Ss**	**Ms**	**Cs**	**Ss**	**Ms**	**Cs**	**Ss**
activemq-artemis	379	177	21	489	211	21	66	43	13	472	210	21
cayenne	417	204	13	460	224	13	94	65	4	463	217	21
cloudstack	410	198	41	466	223	43	190	80	31	466	218	43
cxf	709	291	60	1091	329	62	66	52	22	690	323	62
drill	295	144	39	386	165	40	73	49	17	349	164	40
hadoop	537	356	41	667	426	41	91	67	20	172	92	2
ignite	573	377	15	214	120	8	158	112	9	655	409	15
itext7	520	242	10	606	266	10	61	39	9	586	257	10
jackrabbit-oak	907	504	29	1107	581	29	398	246	21	1111	566	29
james-project	252	121	45	325	138	48	4	4	1	282	129	46
jena	815	273	24	920	293	24	132	49	12	930	293	24
nifi	468	289	118	541	338	128	159	115	64	546	328	128
openmrs-core	507	157	3	540	170	3	215	82	3	391	132	1
total	6789	3333	459	7812	3484	470	1707	1003	226	7113	3338	442

Experiment Setting for RQ2: To calculate the test adequacy metrics, we build and execute each project and run the test cases with the generated oracles injected

into the project. Since the *ATLAS* dataset only considers single-line oracles, there are numerous test cases in the repository that are not our target. So we construct a sub-project for each project that only has the target test cases (those with single-line oracles). Specifically, we first remove all the test cases that are not in our test set for each project. After we get the sub-projects, we replace the generated oracles with the original oracles written by the developers. Since the generated oracles are not guaranteed to be executable, after the replacement, we check if the modified test case with the generated oracles is still executable on the project (note that the original developer-written tests are all executable before replacing their oracles). We exclude test cases if they become non-executable after the oracle injection, as we need all test suites to pass to calculate the test adequacy metrics. The resulting numbers of injected oracles are organized in Table 5.3. We use PIT [354] to calculate the test adequacy metrics, i.e. line coverage and mutation score. We also calculate and report (as the original score plus/minus the difference after the replacement) the test adequacy metrics per sub-project within the dataset.

Experiment Setting for RQ3: We conduct a correlation analysis between the textual similarity metrics and the test adequacy metrics. We get both metrics at the project level and perform correlation analysis using Spearman's and Kendall's rank correlation coefficient, as they are the most common analyses done by different previous metric analysis studies.

The two variables being measured for rankings are one textual similarity metric versus one test adequacy metric. Note that for the adequacy metrics, similar to RQ2, we use the differences (deltas) between the score of the developer-written oracle and the generated oracle. We calculate the correlations between each textual similarity metric and the normalized inverse of |delta| of line coverage and mutation scores (min-max normalization). We use delta and not the actual metric score to capture the difference in test adequacy between the ground truth (human-written) and the model-generated oracle. This is to match what the similarity metrics are also assessing, capturing the similarity of the ground truth and the model-generated. We take the normalized inverse to make the delta (difference/distance) into similarity to match textual similarity metrics.

Since we have seven textual similarity metrics and two test adequacy metrics, we have a total of 14 correlation test runs per project and baseline. However, the number of samples (projects) per baseline is limited (only 13 projects), which harms the statistical tests' p-values. We report this individual baseline analysis in the supplementary material in the replication package. But in the paper, we aggregate the four baseline data and look at a pool of 52 (13 X 4) samples per distribution (i.e., each distribution consists of 52 sample values for a given metric). Then we report

Table 5.4: Performance of NOGs with our new dataset. Each column denotes the score of the seven textual similarity metrics. Bold denotes the best score from a NOG.

NOGs	BLEU	ACC	CB	RL	MT	ES	ChrF
ATLAS	32.88	2.15	12.83	22.21	41.98	46.93	22.77
IR	34.35	5.28	21.29	21.99	40.11	49.17	24.48
TOGA	12.78	9.73	12.79	12.3	12.73	12.93	11.29
gpt-3.5	**49.65**	**11.35**	**26.63**	**44.13**	**51.26**	**61.78**	**30.85**

Table 5.5: Automatic similarity metric scores on *ATLAS*.

Project Name	BLEU	ACC	CB	RL	MT	ES	ChrF
activemq-artemis	34.32	3.66	13.85	20.94	42.41	45.02	17.92
cayenne	**40.66**	0.72	17.76	31.24	**50.42**	**54.06**	28.80
cloudstack	30.42	0.24	11.43	26.22	46.51	50.10	25.88
cxf	34.20	0.14	10.19	25.05	47.17	48.64	24.07
drill	32.95	0.00	10.92	20.20	39.73	43.98	23.20
hadoop	31.85	0.74	12.26	22.39	41.51	47.95	23.40
ignite	36.81	1.05	18.29	28.25	46.91	51.88	25.75
itext7	33.52	**8.46**	13.34	22.90	41.65	47.03	25.70
jackrabbit-oak	30.40	0.33	13.25	17.06	41.84	45.61	19.93
james-project	26.29	0.00	**20.61**	9.35	27.53	36.53	12.59
jena	35.46	0.86	7.61	**32.67**	47.28	50.89	**29.45**
nifi	32.87	2.35	14.72	28.20	49.86	50.15	27.24
openmrs-core	30.11	0.99	8.66	14.20	40.36	42.68	17.46
average	33.07	1.50	13.30	22.97	43.32	47.27	23.18
stdev	3.54	2.33	3.81	6.72	5.96	4.60	4.93

the correlation coefficients (ρ or τ) and p-values per correlation test when comparing two metrics (each with a distribution of 52 samples).

From the results, we want to see how many metrics have a significant correlation between them, i.e. p-values of less than 0.05, and ρ or τ values of at least 0.5. We expect these two metrics to have a negative correlation because high similarity metrics mean the two sequences are similar, whereas similar code will result in lower |delta| (difference). We expect a delta score of zero for a perfectly generated oracle. A higher value of |delta| means the two oracles are semantically different.

Experiment Setting for RQ4: In this RQ, we select 104 random samples of generated oracles to understand what causes the correlations. We are specifically interested in cases that have a high disagreement between the textual similarity and adequacy metrics. We pick random 2 samples from each project generated by each

Table 5.6: Automatic similarity metric scores on *IR*.

Project Name	**BLEU**	**ACC**	**CB**	**RL**	**MT**	**ES**	**ChrF**
activemq-artemis	28.40	3.89	13.16	18.34	35.86	47.14	22.42
cayenne	35.89	0.00	11.72	34.17	41.24	52.54	26.61
cloudstack	44.83	18.24	27.18	34.17	48.51	60.40	39.70
cxf	32.71	1.19	14.12	19.66	38.44	49.96	25.87
drill	25.17	0.00	11.08	14.89	29.26	44.15	20.04
hadoop	**53.08**	**29.54**	**33.97**	**44.37**	**55.77**	**65.92**	**47.05**
ignite	26.61	0.80	8.68	13.52	33.15	46.34	21.90
itext7	31.79	8.42	18.47	19.73	34.87	49.99	26.79
jackrabbit-oak	32.64	4.52	18.11	18.90	40.11	50.06	24.32
james-project	24.83	0.31	14.17	9.17	29.15	40.25	17.20
jena	34.00	6.65	16.28	38.48	50.59	61.19	45.28
nifi	34.00	6.65	16.28	22.10	41.20	50.69	27.65
openmrs-core	28.64	0.74	12.08	10.05	35.94	44.55	19.48
average	33.28	6.23	16.56	22.89	39.55	51.01	28.02
stdev	7.98	8.65	6.96	11.26	8.04	7.42	9.75

NOG, where the models are given the same input, i.e., the same generation problem (2 X 13 X 4 = 104).

While doing the manual analysis, we look for any patterns of syntax and semantics of the test code or the oracle itself that may have caused the disagreement.

5.4 Experimental Results

5.4.1 RQ1: Textual Similarity Metrics

The results of the four studied NOGs are organized in Table 5.4. We show the scores of all seven textual similarity metrics discussed in Section 5.3.3. We also organize the results reported in the original papers for comparison, in Table 5.1. In Table 5.1, the results of *ATLAS* are from evaluating the raw dataset (without normalizing identifier names), which is the version used by the other NOGs, i.e. *IR* and *TOGA*. For the results of *IR*, we report the results from the old and new datasets. *IR Old* is the result of using the same dataset as *ATLAS*, without unknown tokens. *IR New* is the new dataset that keeps instances that were dropped from the old dataset due to the unknown tokens. For *TOGA*, we report the results that used the *ATLAS* dataset for their assertion oracle generation. We can see that *TOGA* does not report the BLEU score as the methodology treats this problem as a ranking problem. However, considering the first-ranked candidate assertion as a generation

Table 5.7: Automatic similarity metric scores on *TOGA*.

Project Name	**BLEU**	**ACC**	**CB**	**RL**	**MT**	**ES**	**ChrF**
activemq-artemis	9.16	7.36	9.18	8.88	9.14	9.23	7.99
cayenne	11.70	5.70	8.27	10.63	11.51	11.99	11.29
cloudstack	**25.81**	**23.97**	19.34	**25.42**	**25.74**	**25.95**	**24.35**
cxf	5.78	5.38	5.79	5.74	5.78	5.80	5.52
drill	6.38	5.45	4.71	6.27	6.34	6.40	6.34
hadoop	8.04	6.92	8.02	7.79	7.98	8.10	7.66
ignite	16.92	11.57	17.13	16.06	16.91	17.21	14.55
itext7	5.17	4.33	5.19	5.03	5.15	5.22	4.66
jackrabbit-oak	20.72	16.01	**20.78**	20.12	20.65	20.88	17.61
james-project	0.27	0.27	0.20	0.27	0.26	0.27	0.27
jena	8.55	7.08	8.54	8.35	8.54	8.63	7.70
nifi	17.04	12.73	12.56	16.26	16.26	17.24	15.50
openmrs-core	17.11	8.19	17.50	15.60	17.00	17.52	10.24
average	11.74	8.84	10.55	11.26	11.64	11.88	10.28
stdev	7.26	6.05	6.37	7.00	7.19	7.34	6.37

Table 5.8: Automatic similarity metric scores on *gpt-3.5*. Bold denotes the best score. The asterisk denotes the best average metric score across all NOGs.

Project Name	**BLEU**	**ACC**	**CB**	**RL**	**MT**	**ES**	**ChrF**
activemq-artemis	45.11	9.32	28.66	40.48	48.78	60.44	25.41
cayenne	60.31	18.79	**54.67**	56.54	62.39	72.77	45.22
cloudstack	55.30	18.45	33.36	49.03	57.09	68.75	38.28
cxf	53.28	11.16	31.85	51.39	59.66	67.51	35.82
drill	47.36	6.30	35.58	39.67	45.30	60.21	29.79
hadoop	59.23	26.24	40.36	48.23	55.84	66.89	38.14
ignite	43.28	7.79	32.21	39.08	46.96	59.37	27.46
itext7	49.78	17.58	25.80	46.24	51.49	63.03	33.54
jackrabbit-oak	44.78	7.29	26.04	37.64	49.13	60.96	26.19
james-project	43.68	3.19	26.04	31.78	34.60	53.78	20.29
jena	**69.96**	**47.85**	42.54	**68.07**	**70.76**	**78.65**	**63.26**
nifi	54.03	19.05	33.93	49.83	58.72	67.79	37.48
openmrs-core	58.91	10.79	29.44	48.00	56.61	68.88	35.60
average	**52.69***	**15.68***	**33.88***	**46.61***	**53.64***	**65.31***	**35.11***
stdev	8.03	11.69	8.14	9.34	9.05	6.54	10.81

output, we can calculate the BLEU score. So, for comparison, we have included them in our results.

Using the original *ATLAS* dataset, the best accuracy reported in the original papers is *TOGA*. Since TOGA has a very high score on accuracy, it is very likely that it also achieves the best BLEU score because accuracy is a much stricter evaluation

metric to achieve. However, since they do not report the exact BLEU score, it is uncertain. So the NOG that has the best reported BLEU score is generated from *IR Old*.

Comparing what is reported in the original papers in Table 5.1 and the ones we evaluate in Table 5.4, we can see that the BLEU and accuracy scores reported from the original papers have a big gap from the results we got from the newly processed data. As mentioned in Section 5.3.5, one possible reason for this is that the splitting of train-valid-test splits at a project level is impacted by removing information leaked into the new test set [17]. By keeping the instances from the same project in each split, similar code structures and project-dependent information are removed from the training set, making it harder for the NOG models to generate project-specific oracles.

Overall, in terms of the average scores across the projects, *gpt-3.5* exhibits the best performance among all textual similarity metrics (See average values with an asterisk in Table 5.8). In Tables 5.5-5.8, we also report the textual similarity metrics at a project level to investigate their effectiveness in each project. Each table has a bold score for each metric that achieves the highest with its model. For instance, *ATLAS* achieves the highest BLEU, METEOR, and Edit Similarity scores when evaluating *cayenne* project, the highest accuracy on *itext7*, the highest CodeBLEU on *james-project*, the highest ROUGE-L and ChrF score on *jena*. For *IR*, the best score was achieved when evaluating the *hadoop* project. For *TOGA*, it achieves the highest on six metrics when evaluating *cloudstack*, except for CodeBLEU, which is from *jackrabbit-oak*. And lastly, *gpt-3.5* achieves the best on *jena* for six metrics, except for CodeBLEU, which achieves on the *cayenne* project. Looking at all these project-level data, *gpt-3.5* has the highest average metric values (marked with an asterisk in Table 5.8).

Another clear observation is that the results of the baseline NOGs can vary depending on the project, and the variation is not consistent across the techniques. To summarize these variations, we also report the standard deviation per baseline metric over the 13 projects. The results show that *gpt-3.5* had the highest standard deviation in five out of seven metrics (only the *IR*'s RL and ES had a higher standard deviation than *gpt-3.5*). Despite having the best performance on the textual similarity metrics, it was shown that it didn't have the best reliability in generating a consistent performance across different projects.

Table 5.9: The results of test adequacy metrics for each NOG. LC denotes the line coverage metric, and MS denotes the mutation score. Column ±% denotes the absolute difference in percentage points. The green colour shows an increase after the injection, and the red shows a decrease after the injection.

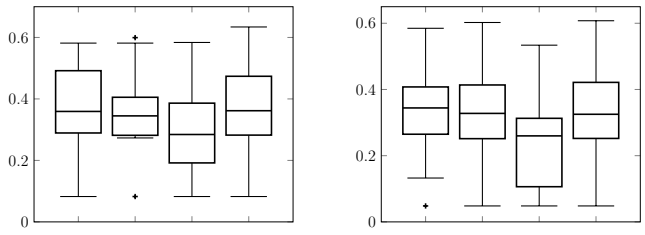
	ATLAS				**IR**				**TOGA**				**gpt-3.5**			
project name	LC	±%	MS	±%	LC	±%	MS	±%	LC	±%	MS	±%	LS	±%	MS	±%
activemq-artemis	28.31	-0.02	27.09	0.79	27.97	-0.23	29.69	-0.26	28.58	**0.24**	28.77	**1.65**	22.71	-0.19	21.68	-0.20
cayenne	46.69	0.00	38.47	0.11	44.32	0.00	40.53	-0.49	56.42	0.00	31.59	0.56	45.71	0.00	41.73	0.32
cloudstack	35.76	0.00	34.21	-0.15	35.19	0.00	32.93	-0.20	38.63	0.00	39.18	-0.40	34.39	1.73	33.50	1.88
cxf	29.56	0.00	25.08	0.00	28.60	0.00	24.75	0.00	22.30	0.00	17.72	0.24	33.44	**2.77**	32.25	0.89
drill	49.04	**0.41**	36.68	**1.06**	35.59	0.00	25.17	**0.36**	14.64	0.00	5.49	0.00	43.18	2.63	26.64	2.30
hadoop	21.02	0.00	13.33	-0.08	27.31	0.00	23.53	0.00	16.07	0.00	6.05	0.00	21.02	-0.66	17.23	-3.06
ignite	48.93	0.00	42.95	0.00	33.78	0.00	37.27	0.00	22.86	0.00	25.90	0.00	48.93	0.00	42.95	-0.72
itext7	53.51	0.00	44.96	-0.41	54.01	0.00	48.99	-0.15	38.85	0.00	15.20	0.00	51.96	0.00	46.13	-0.61
jackrabbit-oak	36.10	0.00	33.93	0.85	36.76	0.00	42.67	0.00	28.03	0.00	26.00	0.06	36.36	-0.68	32.70	-0.83
james-project	31.22	0.00	32.77	-2.87	33.94	-2.74	34.54	-1.73	35.60	0.00	28.30	0.00	33.94	0.00	33.07	-1.39
jena	52.75	-0.05	52.27	0.76	**59.94**	0.00	**60.23**	0.00	38.62	0.00	35.43	0.00	59.87	0.52	57.53	**3.23**
nifi	**58.16**	0.00	**58.48**	0.00	58.16	0.00	58.48	0.00	**58.37**	0.00	**53.38**	0.00	**63.41**	0.00	**59.74**	-0.46
openmrs-core	8.25	0.00	4.82	0.00	8.25	0.00	4.82	0.00	8.25	0.00	4.82	0.00	8.25	0.00	4.85	-0.03
median	36.10	0.00	34.21	0.00	35.19	0.00	34.54	0.00	28.58	0.00	26.00	0.00	36.36	0.00	33.07	-0.20
average	38.41	0.03	34.23	0.00	37.22	-0.23	35.66	-0.19	31.32	0.02	24.45	0.16	38.71	0.47	34.62	0.10
stdev	14.64	0.12	14.63	0.98	14.18	0.76	15.14	0.50	15.16	0.07	14.37	0.49	15.75	1.15	15.39	1.66

From what is originally reported, *TOGA* has the best overall score for textual similarity metrics. From evaluating the newly curated dataset, *gpt-3.5* has the best scores for all seven textual similarity metrics. There is a big drop in textual similarity metrics when we consider project-level splitting. However, we also found that *gpt-3.5* had the highest standard deviation, showing that it has a less reliable performance in generating oracles in different projects.

5.4.2 RQ2: Test Adequacy Metrics

The overall results of test adequacy metrics are shown in Table 5.9. We also report the box plot of test adequacy metrics to compare the different NOGs evaluated in this study in Fig. 5.1. As discussed in Section 5.3.6, we execute the sub-projects before and after replacing the oracles generated by each studied NOG. The test adequacy metrics scores reported in Table 5.9 are calculated before the injection, and the column denoted with ±% on the right side of each metric shows the absolute percentage point difference by calculating the metrics after the generated oracles are injected.

As explained in Section 5.3.6, the changes in the test adequacy metrics are not as significant as we expected, since the generated oracle should closely resemble the one written by the developer. For the results of *ATLAS*, we can see that the highest



(a) Line coverage (b) Mutation score

Figure 5.1: Line coverage and mutation scores of *ATLAS*, *IR*, *TOGA*, and *gpt-3.5*, respectively.

increase in line coverage and mutation score is from evaluating the *drill* project. For *IR*, *drill* had the highest increase in mutation score, but no projects were increased for line coverage. For *TOGA*, *activemq-artemis* had the highest increase in both line coverage and mutation score. For *gpt-3.5*, *cxf* had the highest increase in line coverage and *jena* in mutation score.

To compare which NOG has the best performance overall, we can observe that *gpt-3.5* has the highest increase in line coverage when evaluating *cxf* and mutation score when evaluating *jena*. If we consider the average increase of line coverage, *gpt-3.5* still has the best performance, while *TOGA* has the best performance in the average increase of mutation score. Overall, *gpt-3.5* has the highest performance, which is in line with the results from RQ1. However, it is also very interesting to find that *gpt-3.5* also has the largest number of projects that have a decrease in the test adequacy metrics, with 11 out of 26 fields, which accounts for around 42%. This could also tell us that, despite having a very high score in textual similarity metrics, it could harm the quality or the strength of the generated oracle. This calls to our attention that we must be careful in using textual similarity metrics as the standard to evaluate NOGs.

Also, unlike what we have observed from textual similarity metrics in RQ1, there was a similar trend in the ranges of scores in test adequacy metrics for projects throughout the NOGs. For instance, the metric scores for *activemq-artemis* ranged in the 20s, *cloudstack* ranged in the 30s, *james-project* ranged in the 30s, etc. Some projects had higher variance than others, but the patterns and ranges are much clearer than the ones we observed in the results for textual similarity metrics. One reason that we suspect is that the textual similarity metric has a high fluctuation in the score if the model generates even one token that is different from the ground truth text. Since the task is to generate a one-liner assertion oracle with shorter tokens, the metric will have a very big decrease. However, for test adequacy, even though the

Table 5.10: The result of correlation analysis on the two different types of metrics. Bold denotes the p-value less than 0.05, meaning the correlations are statistically significant. The red cell shows "Very Weak Agreement" and the orange cell shows "Weak Agreement". LC and MS are the inverse values of their deltas (details in Section 5.3.6).

	Spearman's		**Kendall's**	
Metrics	ρ	**P-value**	τ	**P-value**
LC vs BLEU	-0.3461	**0.0070**	-0.2760	**0.0057**
LC vs ACC	-0.0394	0.3944	-0.0342	0.3773
LC vs CB	-0.2806	**0.0250**	-0.2225	**0.0223**
LC vs RL	-0.3096	**0.0148**	-0.2499	**0.0110**
LC vs MT	-0.3347	**0.0090**	-0.2600	**0.0086**
LC vs ES	-0.2821	**0.0243**	-0.2169	**0.0234**
LC vs ChrF	-0.2234	0.0612	-0.1737	0.0556
MS vs BLEU	-0.3982	**0.0021**	-0.2894	**0.0019**
MS vs ACC	-0.0927	0.2637	-0.0695	0.2444
MS vs CB	-0.3198	**0.0121**	-0.2320	**0.0114**
MS vs RL	-0.3286	**0.0102**	-0.2307	**0.0106**
MS vs MT	-0.3178	**0.0126**	-0.2249	**0.0124**
MS vs ES	-0.2926	**0.0202**	-0.2021	**0.0218**
MS vs ChrF	-0.2567	**0.0371**	-0.1874	**0.0307**

generation is different token-wise, if it correctly generates the same type and values in a similar range for an argument, the executed line of code will be similar. Additionally, *gpt-3.5* showed the highest standard deviation, showing unreliable performance for test adequacy metrics.

Overall, *gpt-3.5* showed the best performance in the test adequacy metrics concerning the maximum score it reached and the average score. However, we also found that *gpt-3.5* had the largest number of projects that had decreased in the test adequacy metrics and the highest standard deviation, showing us that it can be unreliable on different projects. We also found that test adequacy metrics are more stable and have a much clearer trend than textual similarity metrics based on the variation of scores across different projects.

5.4.3 RQ3: Correlation Analysis

We report the correlation coefficient (ρ and τ) and the p-value of Spearman's and Kendall's rank correlation in Table 5.10. The bold number in the table shows p-values less than 0.05, which shows the statistical significance of the results. The different colours in the cell show the level of agreement that each coefficient has. Red colour

shows a very weak agreement, orange colour shows a weak agreement, and green colour shows a moderate agreement. As expected, the two metrics show a negative correlation, since we compare the inverse delta of test adequacy metrics to the textual similarity metrics. This means the more textually similar oracles (higher score), the lower their semantic differences (lower delta). For ρ, 0.01 to 0.10 is considered a very weak agreement, 0.10 to 0.39 is considered a weak agreement, 0.4 to 0.69 is considered a moderate correlation, 0.70 to 0.89 is considered a strong correlation, and 0.90 to 1.00 is considered a very strong correlation [355]. For τ, 0.01 to 0.24 is considered a very weak agreement, 0.25 to 0.34 is considered a weak agreement, 0.35 to 0.39 is considered a moderate agreement, and 0.40 to 1.0 is considered a strong agreement [356].

As we can see from the results, except for LC and MS vs Accuracy and LC vs ChrF, the tests from all other pairs result in p-values less than 0.05, which shows their correlation results are statistically significant. However, using either test, all correlations are low. The highest correlation is for MS vs BLEU with ρ = -0.39, which is still considered a weak agreement. Considering these results, we can say that although there is a direct correlation between syntactic and semantic similarity, the strength of the correlations is weak or very weak. Therefore, we do NOT recommend using textual similarity metrics (such as the BLEU category) to evaluate the generated oracle. In other words, unless there is an exact match between the generated oracle and the ground truth, higher BLEU does not necessarily indicate a more effective generated oracle.

Using Spearman's and Kendall's rank correlation analysis, we found that the most closely correlated textual similarity metric to the test adequacy metrics was the BLEU score with $\rho(BLEUvsMS)$=-0.39, which is considered a weak correlation. Based on this finding, we recommend NOG researchers avoid only using textual similarity metrics when evaluating NOGs and make sure to include at least one test adequacy metric as their main metric.

5.4.4 RQ4: Manual Analysis

To further investigate why these two metrics are not significantly correlated, we manually inspect the randomly selected 104 examples of the generated oracles from all projects. We have found three major types of patterns in the generated oracles. 1) Type-1: identical, 2) Type-2: textually different but same/similar functions, 3) Type-3: textually similar but different in semantics. Type-1 exhibits a high textual similarity metric and a close to zero increase/decrease in the test adequacy metric.

That means both metrics agree that the generated oracle and the developer-written one are similar. Type-2 and 3 are the interesting ones. Type-2 exhibits a low score for textual similarity metrics; however, it achieves close to zero increase/decrease in test adequacy. This means the two oracles are textually different but semantically (from a testing perspective) similar. Type-3 also shows a disagreement between the metrics by exhibiting high textual similarity metrics (textually similar) but different in the testing semantics.

From the 104 samples we inspected, we found 36 Type-1 examples, 37 Type-2 examples, and 1 Type-3 example. *TOGA* had the most Type-1 with 22, *gpt-3.5* had the most Type-2 with 17 and the only Type-3 as well. We give examples of the 3 types for better demonstration in Listing 5.2. For Type-1, most of the oracles exactly match the ground truth, leading to high similarity and identical semantics. As introduced also in Listing 5.1, the examples demonstrate Type-2 with a high textual difference but similar functions. Oracles with this type tend to have different assertion types to check the status or call different methods in their parameters with similar or identical semantics to the ground truth oracle. An example of Type-3 is also shown, where it is textually similar but not semantically. Type-3 has a close textual similarity where the parameters are all included in the ground truth oracle with a BLEU score of 0.54. However, the type of assertion is different, and the generated oracle doesn't cover the lines for $IsoMatcher.isomorphic()$. This would affect the test adequacy metrics. The ground truth oracles of this type had multiple or complex method invocations within the parameter, making it hard for the model to generate. These examples show how the generated oracles can cause the two metrics to disagree.

From our manual analysis, we have found three major types of patterns in the generated oracles, and two of them contribute to the nonalignment of textual similarity and test adequacy metrics. Type-1 is an oracle generated with a close agreement to ground truth (high similarity, high test adequacy metric), Type-2 is an oracle with disagreement by low similarity but high test adequacy metric, and Type-3 with disagreement with high similarity but low test adequacy metric. We found that Type-2 is the most common. They tend to have different methods for achieving a similar function. Type-3 wasn't found much; however, it was due to the model's failure to generate the whole sequence of method chain that hindered the test adequacy metric. The first two types were the cause of major disagreement between the two different metric sets.

5.4.5 Practical Implications of the Results

From the results of RQ3, we have observed that the textual similarity metrics and test adequacy metrics exhibit "Very Weak Agreement" to "Weak Agreement", meaning they do not have a strong correlation. This empirical evidence implies that using only textual similarity is not sufficient for evaluating NOG models. However, textual similarity to human-written tests/oracles is still a good measure of understandability and maintainability [101, 357]. Therefore, our recommendation is to use both types of metrics (textual similarity to human-written oracles and test adequacy metrics).

From RQ4, we have observed that the current state-of-the-art models still have challenges in correctly generating oracles with complex parameters (e.g., a long sequence of method chains). To improve the current NOG models, future researchers should: a) guide the models to correctly infer the target library and its APIs and b) use external knowledge to improve oracle generation. These directions could be challenging as library and API information is not provided in the inputs. State-of-the-art LLMs and Retrieval Augmented Generation (RAG) [314] could be a possible solution.

```
// Type-1 example
// Ground truth oracle
assertTrue(answer.getResult())
// Generated oracle from TOGA
assertTrue(answer.getResult())

// Type-2 example
// Ground truth oracle
assertFalse (newMember.isActive(dateToTest))
// Generated oracle
assertEquals (false, newMember.isActive())

// Type-3 example
// Ground truth oracle
assertTrue(IsoMatcher.isomorphic(dsgData, dataset.asDatasetGraph()))
// Generated oracle from gpt-3.5
assertEquals(dsgData, dataset.asDatasetGraph())
```

Listing 5.2: Example of Type-1/2/3.

5.5 Threats to Validity

Internal Validity. To avoid all confounding factors, we use metrics used in the literature for textual similarity and test adequacy metrics. We use the same implementation of the models and the metrics from the original papers to the best of our ability.

Construct Validity. The main threat to the construct validity can be the evaluation metrics we used. The test adequacy metrics used in this study are line coverage and mutation score. Although these metrics are widely used to evaluate the effectiveness of test cases, they may not be highly correlated with finding actual bugs (the ultimate goal of testing). We plan to examine correlations to real bug detection in our future study.

Conclusion Validity. We have conducted two statistical tests and carefully analyzed the statistical significance when reporting correlations, to avoid conclusion validity threats.

External Validity. The main threats to external validity in this study are the limitations to (a) baseline models, (b) test adequacy metrics, (c) the dataset size, and (d) and programming language. Regarding baselines, we have used the most recently published state-of-the-art performing models of neural oracle generation models that were publicly available in which propose a novel way of generating assertion statement lines when the test prefix and focal method are given. Including other baselines might change the observations from this study and will be worth exploring when they are proposed in the future. Regarding test adequacy metrics, we use the line coverage and mutation score. It is possible that our findings do not apply to other levels of code coverage, i.e., branch coverage, statement coverage, or method coverage. In the future, we plan to include more types of code coverage. Regarding the dataset, although we have put much effort into maintaining high-quality data by using real-world test methods contributed by developers on GitHub and using multiple varieties in the domain of projects, it may not be enough to generalize to all data points. This was inevitable as we had limited resources to evaluate projects that needed to build, execute, and run the entire project to get test adequacy metrics, which takes a lot of time and computational complexity. However, we deem we have devised a good number of projects from different domains and a good number of instances in test methods, to show significant observations from the used test set. Regarding the scope of programming language, we only investigate `Java` as all of the most relevant NOGs were based on *ATLAS*, which is a `Java` dataset. Our findings are limited to `Java` and whether they could be generalized to other programming languages is left for future studies.

5.6 Related Work

5.6.1 Traditional Test Oracle Generation

There have been numerous traditional test oracle generation techniques before neural generation models. In this subsection, we very briefly mention them. Peter and Parnas [358] proposed a test oracle generator tool that uses relational program specifications or documents to generate expected outputs of tests as tabular expressions. Bousqpent et al. [359] proposed *Lutess*, a framework that automatically constructs test harnesses from various formal descriptions, i.e. software environment constraints, functional and safety-oriented properties, software operation profiles, and software behaviour patterns. Shahamiri et al. [360] proposed an automated test oracle framework using I/O relational analysis to generate the output domain, a multi-network oracle for input-to-output domain mapping, and a comparator to adjust the precision of the generated oracle by defining the comparison tolerance. Liu and Shin [361] proposed a new method, *V-method*, for automatic test case and test oracle generation from model-based formal specifications. They exploit functional scenarios defined in the formal specification, test generation criteria, algorithms, and mechanisms for deriving test oracles.

5.6.2 Re-evaluating Evaluation Metrics

Since our study is about re-evaluating NOG evaluation metrics, in this subsection, we also cover most related work that re-evaluates evaluation metrics but in domains other than NOG. Recently, there have been numerous studies about revisiting evaluation metrics in the natural language processing field. Mathur et al. [362] did a re-evaluation study on automatic machine translation evaluation metrics and how they correlate with human judgments. They argue that the current methods for evaluating metrics are unreliable because they depend on the choice and quality of the translations. They also propose a new method for comparing different systems based on how well they agree with human judgments, and how to measure the errors of accepting or rejecting systems that are better or worse than others. Roy et al. [350] empirically investigated how well automatic metrics, such as BLEU, METEOR, and ROUGE, can measure the quality of code summaries generated by data-driven methods. They find that small differences in metric scores (less than 2 points) are not reliable indicators of better summaries and that some metrics (METEOR and ChrF) are more consistent with human evaluations than others (corpus BLEU). Liu et al. [341] pointed out three inappropriate settings in existing evaluation methods of TOGA

and comprehensively investigated their impacts on evaluating and understanding the bug-finding performance of TOGA.

Some studies revisited textual similarity metrics in the code generation domain. Takaichi et al. [363] evaluated various NLP metrics for their suitability in assessing code generated by automated techniques, particularly when the code is syntactically incorrect. They conclude that METEOR is the most effective metric for evaluating the ease of modifying generated code that aligns with the natural language-based inputs. Evtikhiev et al. [364] evaluated six metrics, i.e., BLEU, ROUGE-L, METEOR, ChrF, CodeBLEU, and RUBY for code generation models and found that none can emulate human judgment with high certainty for the CoNaLa dataset. They suggest that ChrF is a better fit for evaluating code generation models, but emphasized the need for a new metric that closely agrees with human evaluation. Liguori et al. [365] evaluated various textual similarity metrics for assessing AI-based offensive code generators, comparing them to human judgment to determine their effectiveness in different contexts. They provided insights into the strengths and limitations of these metrics, highlighting the need for more accurate evaluation methods for code correctness. Sikand et al. [366] revisited the metrics used to evaluate generative AI models for code development. They surveyed satisfaction, well-being, performance, activity, communication, and collaboration. Textual similarity metrics fall into the performance aspect. They concluded that there needs to be new metrics concerning the maintainability and the security of generated code. Unlike the above studies, in this work, we revisit the performance of three recent neural oracle generation models and *gpt-3.5* on oracle generation with static textual similarity and dynamic test adequacy metrics.

5.7 Conclusion

This paper conducted an empirical study of existing neural oracle generation models. We first investigated the models' performance on different textual similarity evaluation metrics. We assessed the generated oracles' performance in their test adequacy metrics, i.e. line coverage and mutation score. We performed a quantitative and qualitative analysis of the textual similarity metrics and the test adequacy to find the correlation of these models and find the gaps between what is currently being used and what we should aim to achieve from the study of this field. We found that the correlation between textual similarity metrics and the test adequacy metrics was not significant, meaning the currently assessed textual similarity metrics, i.e. BLEU, accuracy, Rouge-L, METEOR, and CodeBLEU, have no significant relationship with test adequacy metrics, which we use to evaluate the effectiveness of test cases. This shows that

textual similarity metrics are not optimal for showing the quality of the generated oracles and that test adequacy metrics should be considered the main evaluation metrics in this field.

Chapter 6

LLMs on Retrieval-Augmented Test Generation

6.1 Introduction

Retrieval Augmented Generation (RAG) has seen increasing use in various automated software engineering tasks, including code generation [367, 368], code search [369], commit message generation [370], code suggestions [371], code completion [372, 373], assertion generation [107, 374], and automated program repair [375, 107], showing significant improvements. RAG is an approach that enhances LLMs by fetching relevant external documents and using them to ground and improve their responses [25]. A key factor behind RAG's effectiveness is that although LLMs are trained on massive corpora, they often cannot fully leverage the knowledge when tackling complex tasks. Augmenting them with targeted data surfaces and grounds insights to a more correct response [376]. Despite its notable advancements, the potential of RAG in unit test generation remains underexplored. To bridge this gap, we investigate the efficacy of RAG in test generation. As RAG can be built on different knowledge bases, we explore the impact of different document sources on test generation to provide insights into its practical benefits and limitations.

Specifically, we consider the effect of three types of domain knowledge for the RAG used in this work: 1) API documentation, 2) GitHub issues, and 3) StackOverflow Q&As. We include these sources as they could provide examples of API usage for creating tests. Each source offers knowledge from different perspectives: API documentation with official usage guidelines, GitHub issues offer resolution from the library developers, and StackOverflow Q&As present community-driven solutions and best practices.

In our experiments, we focus on five well-known Python-based Machine and Deep learning (ML/DL) libraries: TensorFlow, PyTorch, Scikit-learn, Google JAX, and XGBoost. The domains were chosen for two key reasons. First, they play a critical role in modern software development by providing essential tools, frameworks, and abstractions for building, training, and deploying complex neural networks efficiently [377, 378]. Second, unlike traditional software, ML/DL systems are inherently statistical and non-deterministic. Their APIs reveal expansive, often infinite parameter spaces and exhibit complex object behaviours, making them fundamentally different from conventional, deterministic code. These differences make unit testing more challenging, yet many ML libraries remain under-tested. This study demonstrates how RAG-based LLMs can systematically generate robust, targeted tests to improve the reliability of such frameworks [186, 247, 379, 380].

We evaluate the effectiveness of four state-of-the-art LLMs, i.e., *GPT-3.5-Turbo, GPT-4o, Mistral MoE, and Llama 3.1.* In total, we inspect the unit tests generated for 694 APIs across the five ML/DL libraries, with 19K GitHub issues and 22K StackOverflow Q&As.

We start by quantitatively evaluating the generated tests based on their Abstract Syntax Tree (AST) parse rates, execution rates, and pass rates. Next, we calculate the line coverage on the API under test. Finally, we conduct a manual analysis on a subset of the generated tests to evaluate the bug detection rate and the impact of different RAG sources on the software under test in greater depth.

Our findings indicate that LLMs with basic instruction prompting generate syntactically correct test cases with an average parse rate of 93.89%, an execution rate of 71.35%, and a pass rate of 61.11%. When evaluating RAG-based unit test case generation, we observe promising results compared to basic instruction prompting. Specifically, RAG unit test generation improves line coverage by an average of 6.5%, with the highest improvement reaching an average of 8.1% when using API-level GitHub issue RAG.

We also analyzed the bug detectability from the generated unit tests and found that it detected 28 bugs, 24 unique bugs were reported to developers, so far, ten were confirmed. Finally, we find that RAG helps cover unique lines by providing unique states of programs that cannot be generated directly by LLMs.

This work pioneers RAG-based LLMs for unit test generation in ML/DL libraries and systematically evaluates how three domain-specific knowledge sources, i.e., official documentation and community-based forums, can advance code coverage and bug detectability. We also propose a simple yet effective RAG approach, API-level RAG, that improves both test adequacy.

This paper makes the following contributions:

- We are the first to explore the potential of RAG-based LLMs for unit test generation tasks on ML/DL libraries.
- We investigate the effectiveness of RAGs with different sources of domain knowledge for unit test generation, i.e., API documentation, GitHub issues, and StackOverflow Q&As.
- We perform a comprehensive manual analysis of the generated tests that detect bugs and examine the advantages and limitations of various domain knowledge sources in RAG-based unit test generation.
- We created a benchmark dataset for RAG-based test generation. All code and data used in the experiment are shared as a replication package for future researchers[19].

We organized the rest of this paper as follows. Section 6.2 introduces the background of our study. Section 6.3 details the data collection and implementation of the RAG-based test generation. Section 6.4 shows the results of our experiments. Section 6.5 discusses the possible threats in the study. Section 6.6 concludes this paper.

6.2 Background and Related Works

6.2.1 RAG Incorporation in ASE

Retrieval-augmented generation is a prompting strategy that enhances the responses of LLMs by incorporating additional knowledge sources beyond the pre-trained knowledge base [381]. The query intended for the LLM is used to search and retrieve relevant documents from a database [381, 314]. This process leverages external knowledge, such as documents from various sources, to produce more contextually relevant and accurate responses compared to traditional generation-only models [314].

One advantage of RAG is that it eliminates the need for retraining, as the LLM can search the latest information to generate more reliable output through retrieval-based generation [382]. These techniques have been successful in numerous software-related tasks, including code generation [367], code search [369], commit message generation [370], code suggestions [371], assertion generation [107], and automated program repair [375, 107]. However, the effectiveness of RAG relies heavily on the quality of

[19] `https://zenodo.org/records/16136969`

the retrieved knowledge [383]. Therefore, it is essential to assess the impact of various knowledge resources on the performance of RAG-based techniques.

6.2.2 Unit Test Generation using LLMs

Recently, numerous attempts have been made to generate tests using LLMs automatically [249, 384, 17, 115, 106]. Yuan et al. [115] found that while *ChatGPT* generates unit tests with good coverage and readability, it struggles with correctness. To address this, they propose *ChatTester*, which enhances test correctness through iterative refinement. Chen et al. [106] introduced *ChatUniTest*, an LLM-based unit test generation framework with an adaptive focal context mechanism and a generation-validation-repair workflow, enhancing accuracy and outperforming existing tools. Yang et al. [385] empirically evaluated open-source LLMs for unit test generation, comparing them to *GPT-4* and *Evosuite*. They emphasize prompt design, hallucination issues, and in-context learning, concluding that while promising, LLMs are still outperformed by traditional methods in coverage and defect detection. Zhang et al. [386] introduced Property-Based Retrieval Augmentation, implemented in *APT*, to enhance LLM-based unit test generation using task-specific code context and the Given-When-Then (GWT) structure. This approach improves correctness, completeness, and maintainability through iterative refinement and structured property retrieval. Others studied test generation to increase their code coverage using search-based algorithms, symbolic execution, and genetic algorithms [186, 387, 324, 388]. In contrast to these studies, our study focuses on the impact of different knowledge sources for RAG-based unit test generation with a specific emphasis on ML/DL libraries.

Apart from previous studies that focus on Java projects, studies on Python projects, specifically in the ML/DL domain. Yang et al. [389] introduced $\nabla Fuzz$, the first general-purpose API-level fuzzer for testing automatic differentiation in deep learning libraries. It detects inconsistencies in higher-order gradients using differential testing across execution modes, outperforming prior fuzzing techniques in bug detection and code coverage. Zou et al. [390] proposed *Ramos*, a novel hierarchical heuristic test generation approach for ML projects. Their study employed a hierarchical structure of machine learning models to generate more models by mutation techniques. Narayanan et al. [391] proposed *MUTester*, which mines API usage patterns from StackOverflow to enhance search-based unit test generation for ML projects.

Note that previous work on LLM-driven unit test generation has largely focused on iterative prompting strategies via execution feedback [112, 106] or limited RAG

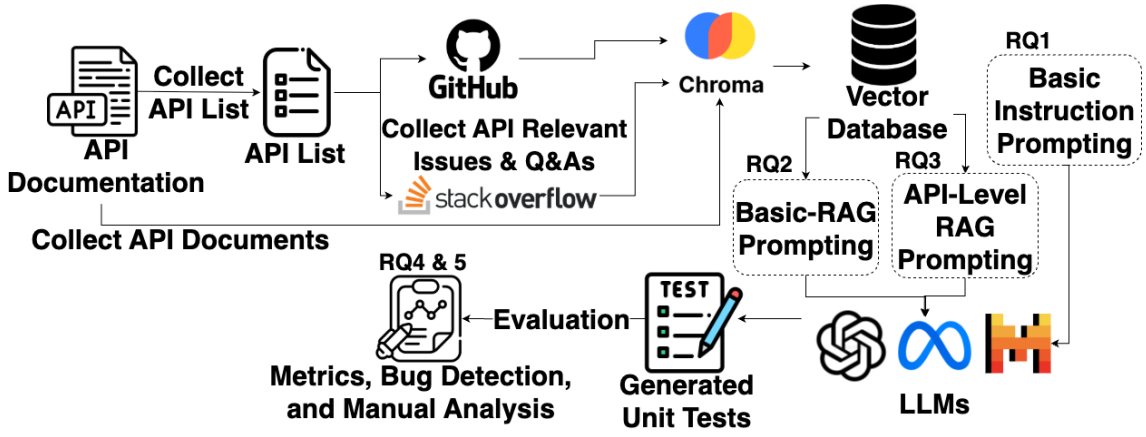


Figure 6.1: Overview of the proposed method.

without systematically exploring how different domain-specific context or community knowledge can improve coverage and bug detection [391, 385]. In contrast, our study is the first to systematically investigate and compare the effectiveness of three distinct external knowledge sources as RAG databases for unit test generation, bringing new insights into their comparative performance.

6.3 Methodology

This section details the methodology of our study, i.e., the research questions, the data collection, the retrieval-augmented test generation framework, baseline LLMs, and the evaluation metrics. Figure 6.1 depicts the overall flow of our study.

6.3.1 Research Questions

To investigate the effectiveness of RAG-based unit test generation for ML/DL libraries, we devised the following research questions:

RQ1: How effective are LLMs in generating unit test cases?
Motivation. This RQ investigates the ability of state-of-the-art LLMs using basic instruction prompting to generate unit test cases without document augmentation. This RQ serves as a baseline for subsequent research questions.

RQ2: How effective is the Basic RAG approach in enhancing LLMs' unit test generation using domain knowledge from API documentation, GitHub issues, and Stack Overflow Q&As?

Motivation. This RQ examines the impact of the Basic RAG approach using documents from three external sources. We investigate improvements in test generation quality, focusing on syntactical and execution correctness and test adequacy (line coverage).

RQ3: How effective is the API-level RAG approach compared to the Basic RAG in the LLM-based unit test case generation?
Motivation. In this RQ, we designed an API-level RAG database to improve upon the Basic RAG approach. While Basic RAG may retrieve irrelevant documents related to other APIs, our API-level RAG aims to ensure that only documentation relevant to the target API under test is retrieved, thereby improving precision and relevance.

RQ4: How effective are the unit test cases generated by RAG-based LLMs in bug detection?
Motivation. In this RQ, we manually investigate how many true bugs can be detected when executing the generated unit tests to evaluate their practicality.

RQ5: What are the underlying reasons when a particular source of knowledge improves the baseline LLMs?
Motivation. In this RQ, we conduct a manual analysis of a subset of APIs to further investigate the root causes of how and why these RAG sources and approaches influence unit test generation.

6.3.2 Data Collection

Studied ML/DL libraries

For our experiments, we focus on five major Python-based ML/DL libraries, i.e., **TensorFlow**[20] (abbr. **tf**), **PyTorch**[21] (abbr **torch**), **Scikit-learn**[22] (abbr. **sklearn**), **Google JAX**[23] (abbr. **jax**), and **XGBoost**[24] (abbr. **xgb**). They were selected due to their widespread popularity and adoption in both academia and industry, their large and active communities with extensive documentation, and their significant roles in the field of machine and deep learning. These factors make them ideal candidates for a study aimed at improving test case generation using RAG.

[20] `https://www.tensorflow.org/`
[21] `https://pytorch.org/`
[22] `https://scikit-learn.org/stable/index.html`
[23] `https://github.com/google/jax`
[24] `https://github.com/dmlc/xgboost`

Table 6.1: Statistics of collected data.

Project	# of API Docs	# of Issues	#of Q&As
tf	638	5,317	8,254
torch	327	7,146	5,036
sklearn	409	4,274	7,946
jax	257	1,708	326
xgboost	237	796	464

Experiment Data Collection.

The collected data include 1) the list of API names, 2) API documentation, 3) GitHub issues and the responses, and 4) the StackOverflow Q&As. Initially, we gathered all API lists from their respective official API documentation website. For API documentation, we extract 1) the signature information with parameter details, 2) the natural language description detailing the functionality of the API and its parameters, and 3) example code demonstrating correct usage patterns, if available.

For the second RAG source, we mine closed issues from their GitHub repositories, specifically extracting the title, description content, and associated comments. For the final RAG source, we collect relevant StackOverflow questions using the tag feature and the associated accepted answers by the poster. Each API document, GitHub issue thread, and StackOverflow Q&A pair is treated as a separate document chunk for the RAG database. To control dataset size and inference costs within budget and ensure compatibility with various LLM context windows, we truncated all documents to a maximum of 5K tokens. This prevents a few exceptionally long texts (which often come from lengthy issue threads) from exceeding context limits or incurring unnecessary computational overhead. We found that the latter parts of these threads (e.g., replies or solutions) are generally less relevant for test generation, though they may be useful for tasks like program repair.

Then, we apply a simple string-matching heuristic to filter out documents irrelevant to API usage. Specifically, we only include GitHub issues and StackOverflow Q&A documents that mention any of the APIs for the RAG document (either in the title or the content). We link the mentioned APIs to each document and use this mapping information for the API-level RAG approach (Details in 6.3.3). Any APIs not mentioned in either source (issues and Q&As) are excluded from data collection. This results in a many-to-many relationship between issues and Q&A documents with APIs, while API documents maintain a one-to-one relationship.

The version of the API documentation collected is tf: 2.16.1, torch: 2.6.0, sklearn: 1.5.0, jax: 0.4.28, and xgb: 2.1.0, which was the stable version at the time of the

conducted study. The time frame of the GitHub and StackOverflow document collection is from the beginning to Jun. 2024. The distribution of the number of APIs after filtering and the number of associated documents is presented in Table 6.1. To ensure a fair and comprehensive evaluation of RAG across different external knowledge sources, we only considered those APIs for which both GitHub issues and StackOverflow Q&A each contain at least ten documents (to ensure sufficient context for retrieval), yielding a final set of 694 APIs in total.

6.3.3 Retrieval-Augmented Test Generation

The RAG approach searches relevant documents and augments the query to assist LLMs in generating more effective and correct unit test cases for ML/DL APIs. We consider two settings for the RAG approach, i.e., the Basic RAG and the API-level RAG. The key difference between Basic RAG and API-level RAG lies in the structure and scope of their retrieval databases. Basic RAG retrieves the top-k globally relevant documents from a shared corpus, regardless of whether they mention the target API. In contrast, API-level RAG constructs a separate database for each API using keyword search, then applies a dense retriever within that API-specific set to ensure more focused and relevant retrieval. This hybrid sparse and dense strategy was shown to outperform either approach alone in previous studies [392, 393].

RAG Database Creation:

The Basic RAG approach stores all documents in a single database, reflecting the most common cases of RAG-based LLMs [394, 371, 107]. Basic RAG retrieves the top-k documents from a shared database that aggregates documents related to all APIs across the five ML/DL libraries for each source (API documentation, GitHub Issues, StackOverflow Q&As, and their combination). To investigate the impact of different RAG sources, we construct four types of Basic RAG databases: 1) all API documents combined, 2) all GitHub issues combined, 3) all StackOverflow Q&A combined, and 4) an aggregation of the three. As mentioned previously, we discard all documents irrelevant to API usage. When filtering these documents, we additionally associate documents with APIs using simple string-matching heuristics. We use this mapping information to construct the RAG database for each API, which strongly constrains the search to relevant documents. This method provides a solid constraint for retrieving only pertinent documents. Consequently, we create a database for each API to facilitate searching within when generating unit test cases for a particular API, thereby defining the API-level RAG approach.

```
basic_instruction_prompt = [
    {"role": "system",
     "content": f"You are a unit test suite generator for {ML_LIB} library."},
    {"role": "user",
     "content": f"Generate a python unit test case to test the functionality of {API_NAME}
     API in {ML_LIB} library with maximum coverage. Only create new tests if they cover
     new lines. Generate test suite using unittest library so it can be directly runnable
     (with the necessary imports and a main function)."
]
```

Figure 6.2: The template used for basic instruction prompting.

```
final_prompt = f"{instruction_prompt} Use the following documents (surrounded by
 @@@) to make the test case more compilable and passable, and cover more lines.
 @@@ Doc_1: {doc1) @@@\n @@@ Doc_2: {doc2} @@@\n @@@ Doc_3: {doc3} @@@"
```

Figure 6.3: Additional prompt used for augmented generation.

We construct the vector databases for the RAG using the *chromadb*[25] library, an open-source vector database designed for storing vector embeddings. It employs the nearest neighbour embedding search for efficient semantic search, utilizing Sentence Transformers [395] to embed the documents, which is considered a dense retrieval method. We employ the dense retrieval method as prior work has generally shown it to be more accurate than sparse retrieval methods [396, 397].

Prompting LLMs:

We task an LLM to generate unit test cases for each API to ensure maximum coverage of its functionality. Additionally, we require generating new tests only if they cover new lines, allowing the model to determine the number of test cases needed autonomously. The generated test suite must utilize the *unittest* library and include all necessary imports and main functions to ensure the code is directly executable.

We followed a bottom-up, task-specific approach to design our prompts. First, we defined a system prompt framing the model as a test suite generator for a given ML/DL library, encouraging developer-style output. The instruction prompt was crafted to produce standalone/executable test suites, including all necessary imports and a main function. We also instructed the model to generate tests only if they cover untested lines. The prompt was refined through small pilot runs. We observed whether the outputs were executable, diverse, and free of redundancy, and iteratively tuned the phrasing until the results consistently met our criteria. Figure 6.2 illustrates the prompt used for basic instruction prompting. When augmenting the prompt

[25] `https://www.trychroma.com/`

with documents retrieved from the RAG database, we instruct the model to utilize the provided documents to enhance the test case's effectiveness and exercise more lines. The additional prompt we use for augmented generation is listed in Figure 6.3. A more detailed demonstration of the engineered prompts can be found in our replication package.

For Basic RAG prompting, we set the number of retrieved documents to three, as previous research [371] indicated that metric improvements plateaued beyond this number. Specifically, we retrieve the top three API documents for Basic API-documentation RAG, the top three issue documents for Basic GitHub-issue RAG, and the top three Q&A documents for basic StackOverflow Q&A RAG. For the Basic combined RAG, note that the top three documents retrieved are not necessarily from each of the three sources, as the retrieval criterion is general relevance.

To retrieve the documents, we use a query to search for relevant documents and specify the number of documents to retrieve. The query used for retrieval is the instruction prompt: "Generate a Python unit test case to test the functionality of *API_NAME* in *ML/DL_LIB* library with maximum coverage."

For API-level RAG prompting, we set different numbers of documents for retrieval: 1) one for the API document, as there is only one document per API, 2) three for GitHub issues and StackOverflow Q&A, similar to Basic RAG prompting, and 3) one document each (total of three) for the combined API-level RAG approach. We add the query listed in 6.3 for each augmented document. We use the top response generated by the LLMs, with the temperature set to zero to ensure a more deterministic and reliable response [276].

LLM Baselines

The following LLM baselines are chosen due to their recency, state-of-the-art, and various architectures: *GPT-3.5-Turbo*[26], *GPT-4o*[27], *Mistral 8x22B Instruct*[28], and *Llama 3.1 405B Instruct* [293]. These models represent a range of capabilities and architecture designs, from the highly versatile and widely adopted *GPT* series to the specialized mixture of experts (MoE) approach in Mistral and the efficient Llama model. We also diversified the open and closed-source models, where the *GPTs* are the two closed models, and Mistral and Llama are the open-sourced models.

[26] `https://platform.openai.com/docs/models/gpt-3-5-turbo`
[27] `https://openai.com/index/hello-gpt-4o/`
[28] `https://mistral.ai/news/mixtral-8x22b/`

Table 6.2: RQ1: Result of basic instruction prompting. The first row denotes the five libraries. The second row denotes the metrics. PR% denotes parse rate, EX% denotes execution rate, PS% denotes pass rate, and CV denotes code coverage. Bold denotes the highest CV among the models.

	tf				torch				sklearn				xgb				jax			
FM	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**
4o	100.00	89.17	81.29	**39.08**	100.00	91.09	84.49	**25.66**	100.00	92.51	80.22	**68.19**	100.00	45.70	33.33	**56.93**	100.00	89.38	83.95	**34.45**
3.5T	99.04	71.84	60.58	36.72	93.20	88.01	76.40	25.43	96.45	91.41	74.48	59.76	99.28	64.08	49.51	48.12	100.00	81.75	77.78	33.14
MS	95.19	42.73	32.04	37.05	95.15	64.19	50.68	18.29	92.39	69.67	54.38	58.49	95.65	38.46	25.34	51.52	97.92	75.98	61.57	33.01
LM	80.77	50.28	43.50	32.84	85.44	79.09	71.52	17.43	64.97	80.00	70.00	22.78	94.93	55.09	55.56	16.72	87.50	66.67	55.56	30.95

6.3.4 Evaluation Metrics

We evaluate the generated test cases both statically and dynamically, utilizing the following commonly used metrics. We follow and adopt the metrics used in previous studies to quantitatively assess the performance of the generated unit tests [17, 19, 89].

1. **Parse Rate** (abbr. PR%): The AST parsability to check the syntactical correctness of the generated tests. The parse rate is calculated at the test suite level since we prompt the models to generate directly runnable, standalone tests.

2. **Execution Rate** (abbr. EX%): For parsable test suites, we execute them to assess correctness, computing the pass rate as the proportion of tests that run without errors.

3. **Pass Rate** (abbr. PS%): Since not all the executable tests pass, we also report the pass rate of the test cases, calculated as the number of passing tests divided by the total number of executable tests.

4. **Line Coverage** (abbr. CV): We calculate line coverage to evaluate the test adequacy of the generated tests. Specifically, after executing a test suite for the API, we calculate the target classes' coverage.

5. **Bug Detectability** (abbr. BD): This metric evaluates a unit test's ability to identify faults. We count the number of true bugs revealed by the generated unit test cases.

Note that these metrics are intentionally designed with different granularities to assess the test cases comprehensively. The parse rate is evaluated at the level of the test suite code generated per API. Given that multiple unit test cases are defined within the test suite, we calculate the execution and pass rates at the method level. We

assess class-level line coverage, as the code within the class and the API method are highly cohesive. Lastly, BD is calculated at the unit test (method) level.

6.4 Experiment Design and Results

6.4.1 RQ1: Performance of LLMs with Basic Instruction Prompting

Design. In this RQ, we prompt the models to generate unit test cases for each API without augmenting them with external sources. We extract only the Python code part from the response, as LLMs tend to generate verbose natural language explanations.

We parse the generated test suite using the *ast* built-in module and calculate the parse rate. Then, we execute the ones that are parsable. We then parse the log to identify unit test cases that result in errors or failures, thereby calculating the execution and pass rates. For line coverage, we use coverage.py[29] to obtain the coverage report. We calculate the coverage score of the class that defines the API to assess the coverage of the affected class.

It is important to note that these coverage reports do not explore the library's back-end implementation, such as the C++ implementation, which many of these libraries rely on for faster computation. Investigations into the back-end implementation must incorporate each library's specialized testing framework, which requires different expertise, which we leave for future work.

Result. Table 6.2 presents the results of LLM-generated unit tests across five projects. Most SOTA LLMs can generate syntactically correct unit test cases, with parse rates ranging from 65% to 100%. Additionally, most LLMs could generate a considerable number of executable unit test cases, with some exceptions.

This was mainly due to the complex unit tests it generated, which required references to unimported packages, causing them to fail. Similarly, *Mistral* failed to generate executable test cases in certain instances, with a minimum execution rate of 38%. This failure is attributed to the model's inability to generate directly runnable test suites from the prompt. *Mistral* tends to produce a sequence of unstructured code fragments and natural language, including unnecessary elements such as the code base of the actual API and its explanation. These individual code pieces lack the necessary structure and dependencies, such as the main class, unit test functions, and imports, leading to low executability.

[29] `https://coverage.readthedocs.io/en/latest/index.html`

GPT-4o consistently achieved the best overall results, ranking first in parse rate and coverage. It also achieved the highest execution rate, except for xgb (3rd). It also ranked best in pass rate, except for xgb (2nd). *GPT-3.5-turbo* and *Mistral* generally ranked in the middle; *GPT-3.5-turbo* being second three times and *Mistral* twice for code coverage. Both models demonstrated good parse rates and relatively high execution and pass rates.

Llama, on the other hand, had the lowest coverage scores, placing fourth across all projects. This was due to its outputs' low parsability, often mixing code snippets with natural-language descriptions (which prevent automated post-processing from working correctly), and low executability (since it failed to produce standalone, runnable test code). However, the unit test cases generated by Llama appeared valid and appropriate for the target APIs. With a prompt carefully constraining its freedom or a sophisticated parser, Llama shows strong potential to generate effective unit tests.

We also observed that syntactical and execution correctness do not always correlate with higher coverage. Although more tests can aid in testing different program states, additional cases may result in testing similar or identical parts of the code. For instance, *Mistral* had lower scores in parse, execution, and pass rates for xgb compared to *GPT-3.5-turbo*. Despite executing fewer test cases (102 vs. 106), *Mistral* achieved higher code coverage (52% vs. 48%).

While it is important to measure the syntactical and execution correctness of generated unit tests, from a software testing perspective, the code coverage of the test suites is ultimately more important. Therefore, in subsequent RQs, we will further investigate the effectiveness of RAG on code coverage.

Answer to RQ1: The state-of-the-art LLMs have demonstrated promising results in generating syntactic and executable unit test cases. *GPT-4o* showed the most promising results in most aspects. From the initial study, we also observe that syntactical and execution correctness do not always lead to higher code coverage, suggesting further investigation into code coverage.

6.4.2 RQ2: Impact of External Sources in Basic RAG

Design. This RQ aims to investigate how different external sources, namely API documentation, GitHub issues, and StackOverflow Q&As, can enhance the Basic RAG in generating effective and practical unit tests. To determine which documents are most effective in complementing test case generation, we construct four vector databases: (1) API documentation only, (2) GitHub issues only, (3) StackOverflow Q&As only, and (4) a combination of all three.

Table 6.3: RQ2: Result of LLMs for using basic RAG prompting. BL denotes the baselines: basic instruction (BI), combined (CB), GitHub issues (GH), and StackOverflow Q&As (SO). In the second row, PR% denotes parse rate, EX% denotes execution rate, PS% denotes pass rate, and CV denotes coverage. Bold denotes the highest CV among the prompting strategies.

		tf				torch				sklearn				xgb				jax			
LLM	**BL**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**
4o	**BI**	100.00	89.17	81.29	39.08	100.00	91.09	84.49	25.66	100.00	92.51	80.22	68.19	100.00	45.70	33.33	56.93	100.00	89.38	83.95	34.45
	CB	100.00	85.49	75.29	39.12	100.00	89.28	81.00	31.21	100.00	88.68	74.55	66.52	100.00	60.27	45.21	56.06	100.00	85.38	76.32	34.59
	AD	100.00	85.30	76.33	39.25	100.00	88.45	83.17	18.77	100.00	85.65	74.01	**69.52**	100.00	59.81	47.91	**57.17**	100.00	86.89	80.46	**35.03**
	GH	100.00	87.58	78.40	39.28	100.00	90.50	83.25	**31.31**	98.98	89.37	75.30	67.15	100.00	72.48	54.13	55.42	100.00	85.38	79.53	34.66
	SO	100.00	75.83	69.61	**39.35**	100.00	92.15	86.88	25.83	100.00	92.83	80.98	64.94	100.00	66.19	52.16	55.93	97.92	94.12	88.24	34.79
3.5T	**BI**	99.04	71.84	60.58	36.72	93.20	88.01	76.40	25.43	96.45	91.41	74.48	59.76	99.28	64.08	49.51	48.12	100.00	81.75	77.78	33.14
	CB	95.67	68.71	57.78	**37.95**	94.17	82.97	71.38	**30.95**	97.46	78.97	60.82	56.35	100.00	52.53	43.04	45.61	95.83	76.97	70.39	32.80
	AD	96.63	75.52	65.17	36.89	93.20	79.39	68.79	18.09	94.42	82.44	66.81	55.89	97.10	56.71	40.24	**49.65**	95.83	69.44	63.89	**34.02**
	GH	95.67	69.76	56.80	37.22	97.09	84.46	71.28	30.92	92.68	74.19	62.37	44.51	100.00	55.56	48.77	44.80	100.00	75.46	63.19	33.66
	SO	91.35	35.63	26.57	35.96	95.15	79.93	67.03	25.62	95.43	74.15	60.84	53.68	99.28	64.60	50.31	41.56	70.83	62.79	50.00	33.50
MS	**BI**	95.19	42.73	32.04	37.05	95.15	64.19	50.68	18.29	92.39	69.67	54.38	**58.49**	95.65	38.46	25.34	**51.52**	97.92	75.98	61.57	33.01
	CB	74.04	44.66	35.28	**37.68**	64.08	51.80	35.97	**31.21**	72.59	63.77	46.15	49.32	82.61	45.86	24.81	41.35	85.42	72.16	55.11	33.53
	AD	55.77	41.98	27.30	37.03	37.86	59.17	54.17	17.49	52.79	65.57	53.01	42.72	51.45	39.81	26.85	42.74	43.75	67.74	50.00	31.79
	GH	82.21	41.37	30.58	37.51	63.11	51.77	35.46	30.82	76.14	58.33	37.35	49.94	86.23	48.20	29.50	44.61	89.58	76.27	57.63	**35.04**
	SO	75.48	31.40	21.04	35.64	86.41	61.00	35.00	25.47	59.90	60.49	44.96	47.64	57.97	39.60	23.49	50.10	75.00	52.50	40.00	32.88
LM	**BI**	80.77	50.28	43.50	32.84	85.44	79.09	71.52	17.43	64.97	80.00	70.00	22.78	94.93	55.09	55.56	16.72	87.50	66.67	55.56	30.95
	CB	86.54	69.66	58.91	38.22	79.61	81.57	72.63	25.40	69.54	79.00	66.77	50.03	86.23	46.13	31.00	**54.27**	93.75	77.73	70.74	34.24
	AD	86.54	72.41	61.58	**38.94**	80.58	75.88	64.64	18.24	75.63	78.37	64.35	54.77	78.26	45.21	32.98	44.52	79.17	75.12	60.20	33.70
	GH	87.50	67.31	57.14	38.66	78.64	77.60	69.13	17.46	52.79	75.42	64.08	47.39	92.75	49.22	35.16	53.70	93.75	84.62	78.73	34.38
	SO	89.90	59.43	51.15	38.82	88.35	83.29	72.62	26.03	85.28	84.92	70.87	**61.33**	94.93	52.92	38.52	53.34	89.58	70.90	59.79	**35.77**

Since Basic RAG does not differentiate between document types, all documents from the five projects are aggregated into each of the four databases. Similar to the previous RQ, we evaluate the parse rate, execution rate, pass rate, and line coverage.

To further investigate the impact of Basic RAG on code coverage performance, we undertake the following steps: (1) count the wins of each RAG approach versus basic instruction prompting, (2) count the number of wins for each Basic RAG approach against each other, and (3) perform a pairwise ranking of the Basic RAG approaches with the four databases and the basic instruction prompting using the Friedman tests [398] among basic instruction prompting, Basic RAG combined, Basic RAG API documents, Basic RAG GitHub issues, and Basic RAG StackOverflow Q&As.

Result. Table 6.3 presents the results of LLMs in generating unit tests using the Basic RAG. For comparison, we also report the results of basic instruction prompting. Overall, we observe that employing Basic RAG can enhance code coverage compared to basic instruction prompting. However, the augmentation of documents appears to negatively impact the parse rate for certain models, such as *GPT-3.5-turbo* and *Mistral*. This is attributed to the fact that these relatively smaller models are disrupted by the lengthy augmented documents, causing them to deviate from the initial prompt to "generate directly runnable test suites" and produce unparsable code [399].

We report the win count on code coverage for each approach, as depicted in Figure 6.4a and 6.4c. In a total of 20 cases (5 projects × 4 LLMs), the combined Basic RAG outperformed basic instruction prompting 13 times, the API document Basic RAG 12 times, the GitHub issues Basic RAG 14 times, and the StackOverflow Q&As Basic RAG 11 times. Contrary to our assumption that combined RAGs would outperform individual RAGs due to the inclusion of all documents, the results did not consistently support this hypothesis. When comparing the combined Basic RAG to the individual RAGs (Figure 6.4c), the combined Basic RAG outperformed the API documents 11 times, the GitHub issues 10 times, and the StackOverflow Q&As 12 times. From the Friedman ranking test, we obtained the following average rankings: (1) Basic combined: 2.7, 2) Basic RAG GitHub issues: 2.7, 3) Basic RAG API documents: 3.05, 4) Basic RAG StackOverflow Q&As: 3.05, 5) Basic instruction prompting: 3.5. The p-value from the Friedman test was 0.4809, indicating no statistical significance of the rankings (p-value ¿ 0.05).

The results indicate that GitHub issues were equally effective as the combined Basic RAG in generating unit test cases, both having the same average ranking of 2.7. A possible reason for this could be that GitHub issues provide detailed code examples for teams to replicate their bugs, and through multiple comments, they offer detailed mitigation strategies. Further analysis through manual assessment will be discussed in RQ5.

Answer to RQ2: The application of Basic RAG improves the generation of unit tests, particularly in code coverage. However, the statistical significance of ordering was not significant using Basic RAGs.

6.4.3 RQ3: Impact of External Sources in API-Level RAG

Design. Similar to RQ2, this research question investigates the effects of different external sources within an API-level RAG setting. Utilizing the available mapping information that links documents to each API, we construct a dedicated vector database for each API. As with the previous RQs, we calculate the four metrics, count the wins for each RAG approach, and perform the Friedman tests on the code coverage for the four API RAGs versus basic instruction prompting. Additionally, we conduct a Friedman test comparing the basic and API RAG approaches, resulting in nine rank comparisons.

Result. Table 6.4 presents the results of LLMs in generating unit test cases using the API-level RAG. For comparative purposes, basic instruction prompting results

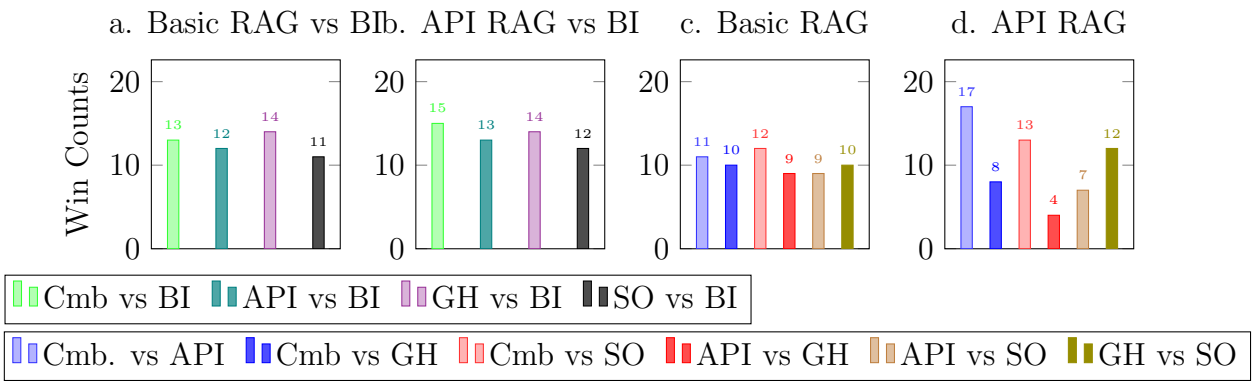


Figure 6.4: The left two sub-figures show the win counts of the RAG approaches vs the basic instruction prompting (BI). Cmb denotes combined RAG, API denotes API documents, GH denotes GitHub issues, and SO denotes StackOverflow Q&As. The right two sub-figures show the win counts within the RAG approaches.

are also included. Consistent with the findings in RQ2, it is evident that the API-level enhances performance in terms of code coverage compared to basic instruction prompting. A similar decline in parse rate was observed for *GPT-3.5-turbo* and *Mistral*, attributable to the same underlying factors.

Figures 6.4b and 6.4d illustrate the win counts of API-level RAG versus basic instruction prompting within each API-level RAG. Out of 20 cases, the combined API-level RAG outperformed basic instruction prompting in 15 instances, the API document API-level RAG in 13 instances, the GitHub issues API-level RAG in 14 instances, and the StackOverflow Q&As API-level RAG in 12 instances. The win count for API-level RAG versus basic instruction prompting was higher compared to Basic RAG versus basic instruction prompting, with two additional wins for the combined approach, one additional win for the API document approach, an equal number of wins for the GitHub issues approach, and one additional win for the StackOverflow Q&As approach.

Furthermore, the number of wins for the combined API-level RAG versus the individual RAGs was also analyzed. Similar to the Basic RAG results, the combined API-level RAG outperformed the API documents 17 times, GitHub issues 8 times, and StackOverflow Q&As 13 times. Compared to Basic RAG, the win counts increased by six for API documents and one for StackOverflow Q&As, while decreasing by two for GitHub issues. These results suggest a trend similar to Basic RAG, with the combined RAG outperforming the other RAGs except for the GitHub issues RAG.

Table 6.4: RQ3: Result of LLMs using API RAG prompting.

		tf				torch				sklearn				xgb				jax			
LLM	**BL**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**	**PR%**	**EX%**	**PS%**	**CV**
4o	**BI**	100.00	89.17	81.29	39.08	100.00	91.09	84.49	25.66	100.00	92.51	80.22	68.19	100.00	45.70	33.33	56.93	100.00	89.38	83.95	34.45
	CB	100.00	82.70	72.99	**40.09**	100.00	90.08	83.24	26.38	100.00	84.27	69.17	**69.54**	100.00	72.77	57.18	**58.71**	95.83	83.05	75.71	35.88
	AD	100.00	84.96	75.57	39.37	100.00	94.34	94.34	19.18	100.00	82.57	70.31	69.82	100.00	66.60	49.90	57.65	100.00	80.92	76.33	34.92
	GH	100.00	80.76	70.06	40.28	100.00	91.62	82.81	**31.89**	100.00	85.14	72.83	49.59	100.00	59.07	45.57	57.73	100.00	89.68	80.52	35.33
	SO	100.00	77.37	69.76	38.83	100.00	94.07	89.17	25.76	100.00	91.67	80.28	66.15	100.00	65.77	51.34	55.19	100.00	89.18	81.97	**35.98**
3.5T	**BI**	99.04	71.84	60.58	36.72	93.20	88.01	76.40	25.43	96.45	91.41	74.48	59.76	99.28	64.08	49.51	**48.12**	100.00	81.75	77.78	33.14
	CB	96.15	58.77	49.13	**38.91**	93.20	82.67	67.17	31.27	96.45	71.06	50.92	**22.28**	93.48	64.21	51.58	36.23	91.67	62.14	55.71	**34.52**
	AD	96.63	72.38	63.35	37.47	94.17	89.10	78.19	25.50	96.45	73.91	62.88	21.67	99.28	40.41	30.05	44.11	97.92	66.08	62.57	33.41
	GH	96.15	54.43	42.20	38.53	98.06	80.25	64.20	**33.21**	95.94	61.29	38.71	22.11	100.00	51.11	44.44	43.59	95.83	59.85	44.70	34.59
	SO	91.35	47.11	38.45	37.22	97.09	81.49	61.21	25.90	96.45	96.45	96.45	96.45	97.83	26.23	20.49	32.88	77.08	77.08	41.49	**36.61**
MS	**BI**	95.19	42.73	32.04	37.05	95.15	64.19	50.68	18.29	92.39	69.67	54.38	**58.49**	95.65	38.46	25.34	**51.52**	97.92	75.98	61.57	33.01
	CB	82.21	45.65	28.23	37.16	78.64	70.20	48.48	30.79	73.60	67.42	50.47	55.88	89.86	44.53	33.59	47.57	79.17	70.51	58.97	**34.30**
	AD	83.65	42.97	31.18	36.97	95.15	67.65	56.30	25.78	81.73	71.08	57.70	55.57	86.96	38.05	32.74	44.81	81.25	59.79	47.42	33.54
	GH	81.73	41.18	27.78	**37.78**	83.50	59.50	43.50	**32.35**	74.62	55.09	37.89	51.79	88.41	35.19	16.05	47.09	79.17	65.28	41.67	35.12
	SO	75.00	37.98	26.87	36.50	73.79	69.27	48.78	25.50	64.47	69.54	53.00	57.41	75.36	26.24	18.55	45.74	66.67	66.10	44.07	**36.19**
LM	**BI**	80.77	50.28	43.50	32.84	85.44	79.09	71.52	17.43	64.97	80.00	70.00	**22.78**	94.93	55.09	55.56	16.72	87.50	66.67	55.56	30.95
	CB	85.58	59.50	61.21	**39.31**	93.20	89.27	81.57	32.16	71.57	79.55	66.56	18.73	93.48	64.21	51.58	36.23	87.50	73.83	65.89	34.21
	AD	69.71	72.21	61.18	38.10	90.29	89.33	81.17	18.96	72.08	69.08	60.40	19.85	72.46	20.59	17.65	29.97	66.67	70.69	66.09	33.29
	GH	79.81	65.13	55.00	38.94	88.35	84.81	71.03	**32.90**	61.93	76.67	58.89	19.32	84.78	44.83	32.76	**41.65**	89.58	80.79	70.94	34.84
	SO	90.87	59.03	53.08	38.93	95.15	89.58	81.66	26.01	85.79	67.30	52.13	20.82	86.23	81.82	65.15	35.66	85.42	77.51	65.09	**37.26**

Additionally, we conducted the Friedman ranking test on the line coverage from API-level RAG and basic instruction prompting and the results yielded the following rankings: 1) API RAG GitHub issues: 2.3, 2) API RAG combined: 2.35, 3) API RAG StackOverflow Q&As: 3.0, 4) API RAG API documentation: 3.65, 5) Basic instruction prompting: 3.7. This indicates that for API-level RAG, GitHub issues were more effective than the combined version, a trend also observed in the previous research question. The p-value from the Friedman test was 0.0056, indicating the significance of the rankings (p-value ¡ 0.05). To demonstrate the magnitude of the difference between the methods, we have augmented the Friedman test with Kendall's W [400]. The effect size for this ranking was 0.38, indicating a moderate effect size.

Finally, we conducted the Friedman ranking test for all combined approaches, which yielded the following rankings: 1) API RAG GitHub issues: 3.4, 2) API RAG combined: 3.7, 3) Basic RAG combined: 4.9, 4)Basic RAG GitHub issues: 4.9, 5) API RAG StackOverflow Q&As: 4.9, 6) Basic RAG StackOverflow Q&As: 5.55, 7) Basic RAG API Documentation: 5.75, 8) API RAG API Documentation: 5.8, 9) basic instruction prompting: 6.2. These rankings indicate that API RAG on GitHub issues was the most effective. Additionally, API RAG generally achieved better rankings than Basic RAG. The p-value from the Friedman test was 0.0127, confirming the significance of the rankings (p-value ¡ 0.05). The effect size using Kendall's W for this ranking was 0.23, indicating a small effect size.

Answer to RQ3: API-level RAG has been shown to improve unit test generation, particularly in enhancing code coverage, outperforming Basic RAG. Notably, API-level RAG incorporating GitHub issues achieves the best performance, with statistically significant improvements.

6.4.4 RQ4: Bug Detection

Design. This research question investigates the effectiveness of the generated tests in detecting real-world bugs. Since no established bug benchmark exists for ML/DL APIs, we manually evaluate bug detectability. For this experiment, we focus on test cases generated by *GPT-4o*, the best-performing model, on the same sets of APIs as the previous RQs. Given that GitHub issues and StackOverflow Q&As often highlight common bugs that developers encounter, there's a potential risk of information leakage. Thus, we compare the API documentation RAG approach at the API level against basic instruction prompting. While the previous RQs prioritized test effectiveness in terms of coverage, here we explicitly direct the models to generate test cases aimed at revealing bugs by including the following instruction: "Your ultimate goal is to generate test cases that reveal bugs in this API." For the evaluation, we only consider generated test cases that exhibit failure for bug detection. We then analyze the failing test cases and their execution logs to determine whether the test assumptions align with the API documentation. If so, the failure is likely caused by an issue in the API's source implementation. Two authors manually label the detected failures, resolving any disagreements through discussion [401, 402]. For the confirmed true positives, we search the issue repository for similar bugs previously and only report the new ones. The links to the reported issues are included in the replication package.

Table 6.5: Results of bug detection. TC denotes test case (method-level). EX denotes Execution Rate, PS denotes Pass Rate, and BD denotes Bug Detectability. Basic refers to basic instruction prompting. A-RAG refers to API-level API documentation RAG.

Library	**tf**		**torch**		**sklearn**		**jax**		**xgb**	
Prompting	**Basic**	**A-RAG**	**Basic**	**A-RAG**	**Basic**	**A-RAG**	**Basic**	**A-RAG**	**Basic**	**A-RAG**
Total APIs	208		103		197		48		138	
Total TCs	1,355	1,239	603	530	1,213	1,028	255	240	324	402
Failed TCs	167	168	77	50	205	192	15	21	95	95
EX (%)	83.99	83.21	87.40	87.17	89.78	88.91	87.84	87.50	78.40	72.39
PS (%)	71.66	69.65	74.63	77.74	72.88	70.23	81.96	78.75	49.07	48.76
BD	1	10	4	3	4	5	0	0	0	1

```
# Generated unit test for tf.boolean_mask
...
def test_mask_with_different_dtype(self):
    # Test with a mask of different dtype (int)
    tensor = tf.constant([1, 2, 3, 4, 5])
    mask = tf.constant([1, 0, 1, 0, 1], dtype=tf.int32)
    with self.assertRaises(TypeError):
        tf.boolean_mask(tensor, mask)
...
# Error Log
======================================================================
FAIL: test_mask_with_different_dtype (__main__.TestBooleanMask)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/tf.boolean_mask.py", line 28, in test_mask_with_different_dtype
    tf.boolean_mask(tensor, mask)
AssertionError: TypeError not raised
```

Figure 6.5: An example bug detected by a generated test.

Result. Table 6.5 summarizes the bug-detection results for our generated unit tests. We first report the total number of APIs (test class-level) and the total number of unit test cases generated (test method-level). We then present the number of failing tests, which are the ones of our interest. Next are the execution and pass rates, using the same definitions as the previous RQs. And finally, the bug detectability, where the failure reveals a true positive of a real bug. From the results, we observe that most unit tests still pass (average 69.5%) despite our prompts to generate bug-revealing tests. Also, out of the failed tests, only a few reveal real bugs. This suggests that the libraries robustly handle common exceptions and that it is very challenging for the LLMs to generate unit tests that reveal true bugs.

Using the API Documentation RAG approach, we identified a total of 19 true bugs from the five libraries. In contrast, basic instruction prompting uncovered 9 true bugs. In total, the tests detected 28 bugs. Out of them, three were redundant tests (either revealing the same bug or similar tests generated from both basic prompts and API doc RAG), and one of them had already been reported by someone else. So in total, 24 new bugs were reported to their respective GitHub issue repository. At the time of submission, the developers confirmed ten as bugs, four as not a bug, and the rest (ten) are awaiting confirmation. The progress of this can be followed by the links that we provide in the replication package.

Figure 6.5 illustrates a unit test generated by *GPT-4o* using the API Documentation RAG approach, which reveals a bug. The API documentation specifies that only boolean types should be used as mask values; however, the current implementation doesn't enforce or validate this, leading to an assertion error.

Most detected errors stemmed from inconsistencies between the API implementation and its documentation or from inadequate validation mechanisms that failed to raise the expected errors, which aligns with previous findings [403, 404]. Since the prompt explicitly instructed the models to generate test cases for bug detection, many of the tests pushed the API into invalid states and asserted specific error conditions, functioning similarly to fuzz testing. API documentation provides detailed specifications regarding expected input types and values, which helped the API Documentation RAG approach generate more comprehensive test cases to check invalid inputs compared to basic instruction prompting. As a result, the API Documentation RAG was more effective in uncovering bugs, as observed in the example. We can observe that not all failed tests can detect bugs. This issue arises from the model's hallucinations, which lead to an incorrect understanding of the API's intended behaviour, generating incorrect assertions, i.e., test cases that do not align with the actual API specification [405].

Answer to RQ4: LLM-based unit test case generation can detect real-world bugs. Using API Documentation RAG prompting, we found it detected more bugs (19 vs. 9) by generating test cases more aligned with the API documentation. These bugs often stemmed from implementation inconsistencies or insufficient validation mechanisms. We reported 24 new bugs in their respective issue repositories.

6.4.5 RQ5: Qualitative Analysis of RAG

Design. This RQ aims to investigate how different knowledge sources support the generation of more comprehensive and effective tests. To maintain statistical significance while keeping manual analysis feasible, we randomly sampled 250 APIs out of the total 694 (the minimum threshold for 95% confidence with a 5% margin of error is 248). We sampled equal distribution from each library, selecting 75 from tf, 37 from torch, 70 from sklearn, 18 from jax, and 50 from xgb. All evaluations focus on tests generated by *GPT-4o*, with the four API-level RAGs and the basic instruction as our baseline approaches.

Before the manual evaluation, we first identify unique lines covered by each method but not by others. We then analyze the types of information contained in the retrieved documents and how they influenced or guided the model in generating tests that exercised these unique lines.

Next, we categorize the retrieved documents according to the types of information they provide to the LLM, such as usage examples, parameter constraints, or error-handling patterns, and analyze their specific roles and impact in guiding the model's

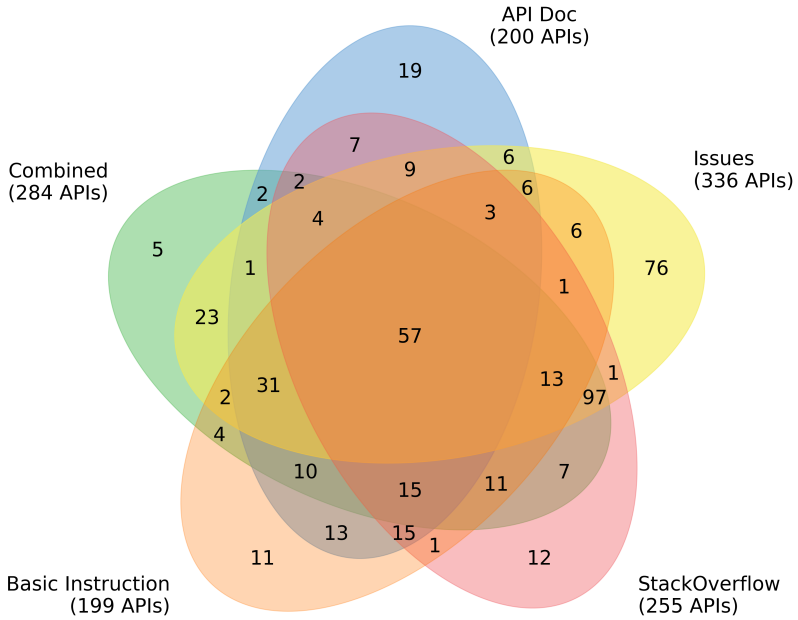


Figure 6.6: The number of APIs with unique coverage.

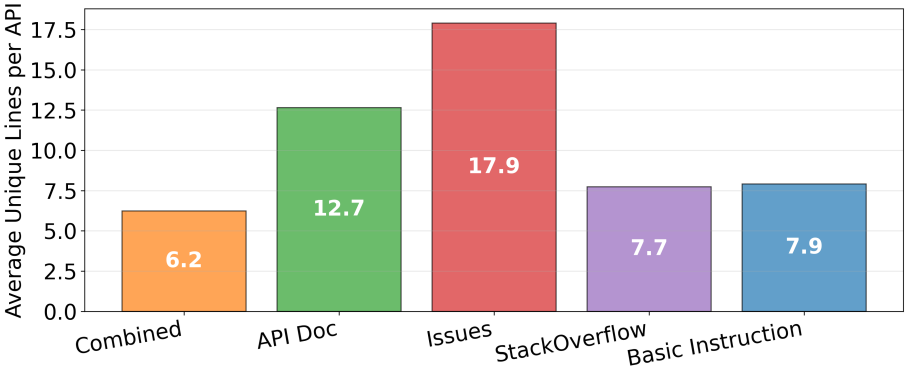


Figure 6.7: The average number of unique lines covered per API by each approach.

test generation. First, we started with initial categories based on our knowledge of the three types of documents: bug-inducing (issues), standard use cases (documentation), and unique/edge use cases (Q&As). Then we performed open-coding thematic analysis, clustered codes through a systematic taxonomy development method, and built a concise code-book [406]. Two participants independently applied it across all documents, and two more participants were involved to resolve disagreements through

consensus. The following is the final list of seven information categories contributed by the retrieved documents:

1. Bug-inducing [407, 408]: Documents under this category often reveal actual defects or incorrect software behaviour, helping the model learn faulty usage patterns or edge cases.

2. Standard use cases [409, 410]: Documents under this category often provide clear and concise examples that demonstrate typical API interactions, helping guide the model toward generating valid and representative test cases.

3. API usage sequences [411, 412]: Documents under this category often illustrate multi-step workflows and sequences of dependent API calls, helping the model understand typical API invocation patterns in realistic contexts.

4. Unique/complex input/parameters [413, 414]: Documents under this category often showcase advanced or edge-case input configurations and parameter settings, enabling the model to generate tests that explore uncommon or corner-case behaviours.

5. Exception handling patterns [415, 416]: Documents under this category often explain proper error handling strategies and the management of expected failure conditions, guiding the model to generate more robust and fault-tolerant tests.

6. Integration patterns [417, 418]: These documents describe how APIs interoperate within larger systems or interact with external libraries, helping the model construct tests that reflect real-world integration scenarios.

7. Performance considerations [419, 420]: Documents in this category focus on efficiency, scalability, and best practices for resource usage, enabling the generation of tests that capture performance-critical behaviours.

Note that the number of categories doesn't always match the number of documents per instance, as a document could have multiple (or even no) parts that influence the test generation. Also, that basic instruction was only used for set deduction to calculate unique coverage, as they do not use any external documents for test generation. All the artifacts that were produced in this process can be found in our replication package.

Result. We first show how each RAG source affects the unique line coverage. Figure 6.6 presents the number of APIs for which each approach achieves unique line coverage, reflecting breadth-wise API coverage. Figure 6.7 shows the depth-wise line

Table 6.6: RQ5: The number of documents in each information category. Bold denotes the highest count for a category compared to other RAGs.

Categories	API Docs	GH Issues	SO Q&As	Combined	Total
API sequence	35 (19%)	86 (21%)	**99** (33%)	79 (23%)	299
Bug-inducing	-	**72** (18%)	6 (2%)	25 (7%)	103
Exception	37 (20)	**49** (12%)	27 (9%)	40 (12%)	153
Integration	1 (1%)	26 (6%)	**33** (11%)	20 (6%)	80
Performance	1 (1%)	**12** (3%)	6 (2%)	2 (1%)	21
Standard cases	27 (15%)	78 (19%)	78 (26%)	**96** (28%)	279
Unique input	80 (44%)	**82** (20%)	55 (18%)	81 (24%)	298
Total	181	405	304	343	1233

coverage, measured as the average number of uniquely covered lines per approach. From these figures, we can observe that GitHub issues have the most APIs with unique lines and the average number of lines per API, aligning with the previous RQs. We can also observe that most APIs are covered (97) when using StackOverflow Q&As, GitHub Issues, and the combined approach. However, if we look at average unique lines covered, API documentation was in second place, not StackOverflow. With the numerous demonstrations of API sequences, StackOverflow gave wide coverage in terms of API numbers. However, with the depth of the information regarding all the parameters provided by the API documentation, the absolute number of lines covered was higher. Basic instruction showed unique coverage, showing 11 APIs that only basic instruction had unique coverage. It also had relatively high average line coverage (7.9), having a higher count than combined and StackOverflow, indicating that higher coverage can be achieved when we utilize complementarily.

Table 6.6 shows the overall results from our manual analysis, the number of each category identified per RAG approach. We can observe that GitHub issues had the most identified documents that influenced tests with unique coverage, with the largest number of categories for four out of seven. Combined RAG was the next RAG approach with the most identified document categories and with the most standard use cases. Although it didn't cover the next most unique lines, it covered the next most APIs in breadth. API Documentation had the least amount of categorized documents and the least variety, due to its fixed style of documentation. The example code and tutorials contributed to standard use cases and API usage sequences, and the detailed parameter information contributed to unique/edge use cases (unique input/parameters). StackOverflow Q&As exhibited the most varied sequences of API usage and integration patterns, as both questioners and answerers contribute custom code snippets that also use other third-party libraries for integration. In contrast, the API documentation featured a high number of unique and complex input and parameter categories, providing comprehensive details about all parameters.

This information guided LLMs in generating unique parameters that are often set as default in standard use cases. In theory, the combined RAG could incorporate advantages from all three sources. However, it shows that applying either source benefits in unique coverage by focusing on its strengths further.

Answer to RQ5: From our manual analysis, we identify seven types of information in documents that aid test generation. We also find GitHub issues cover more lines by providing diverse and unique testing knowledge.

6.4.6 Discussions

Key Insights

Our experimental results show that RAG significantly enhances LLM-based test generation, especially for complex and domain-specific tasks. Although pre-training data may contain relevant documentation, LLMs often struggle to recall specific details or stay up-to-date with API changes. This limitation arises from its static and incomplete parametric memory [421, 422]. RAG addresses this by providing up-to-date, API-specific snippets at generation time, highlighting edge cases from community-reported issues the model may have forgotten. Rather than supplying entirely new data, RAG grounds generation in a precise, relevant context. This reduces ambiguity, limits hallucinations, and leads to measurable gains in coverage and bug detection. Our study provides empirical evidence: RAG achieved a 6.5% point increase in code coverage and identified 10 additional real-world bugs, which is 111% more than without RAG.

Implications for Research and Practice

For researchers, the strong performance of API-level RAG, especially when sourcing from GitHub issues, highlights the value of fine-grained, domain-specific retrieval strategies. Future work should explore automated document selection, hybrid sparse-dense indexing, and extending evaluation to mutation or property-based coverage. In practice, teams aiming to improve code coverage and bug detection should consider integrating an API-level RAG pipeline. For best results, build a lightweight vector store of GitHub issue texts and query it alongside official docs. Budget permitting, GPT-4o is the preferred generator; for cost-sensitive settings, *GPT-3.5-Turbo* may suffice, but with modest performance trade-offs.

Cost, Challenges, and Limitations

Building and maintaining per-API vector databases incurs storage overhead and query latency, and must be refreshed as APIs evolve. Smaller LLMs (e.g., *Mistral, Llama*) faced parse rate declines when handling lengthy retrieved documents, indicating a trade-off between context richness and model capacity. Our requirement of at least 10 documents per API excludes less-popular functions, limiting coverage of newly introduced APIs.

6.5 Threats to Validity

Internal Validity. The primary threat to internal validity arises from the nondeterministic nature of LLMs. To mitigate this, we set the temperature parameter to zero, thereby constraining randomness and ensuring the most deterministic response possible [276]. Additionally, we conducted a preliminary study by running a subset of the experiment five times (using only the TensorFlow and PyTorch libraries) and found that the average difference in metrics was less than 2.5%. There may also be limitations in the prompts used for the LLMs. To address these, we conducted multiple series of prompt optimizations before applying them to the entire dataset, ensuring the prompts were as effective as possible.

Construct Validity. The primary threat to construct validity lies in the evaluation metrics employed. The test adequacy metric used in this study is line coverage. Although line coverage is widely utilized to assess the effectiveness of test cases, it may not strongly correlate with the detection of actual bugs, which is the ultimate goal of testing. The mutation score is another widely used metric in the domain, designed to detect mutations (potential defects) in the software under test. However, in our case, mutating and building the source projects is too time-consuming, rendering our study infeasible. Therefore, we decided to add a manual analysis of the bug detectability of the unit test cases to mitigate this. Further investigation on using fault-based testing in this matter necessitates additional future studies. Another threat is that we did not explicitly evaluate the relevance of the retrieved documents for the different RAGs. In the interest of maintaining scope, we evaluated the end-to-end test coverage and bug-finding performance, rather than examining detailed retrieval accuracy metrics. Consequently, we will address this aspect in future work. Lastly, our benchmarks rely on Python-level coverage tools and thus do not exercise core logic implemented in C/C++ extensions. Instrumenting these lower-level components would require specialized tooling and deep integration with their build systems, which is beyond

our current scope. Consequently, our coverage metrics reflect only the front-end API surface; comprehensive C/C++ level testing remains important for future work.

Conclusion Validity. We conducted both a quantitative and qualitative study to investigate the effect of different RAG approaches and the detection of bugs in the generated unit test cases. However, we examined a subset of libraries and APIs, which could pose a potential threat to the validity of our claims. A further threat arises from the reliance on closed-source LLM APIs (e.g., *GPT-3.5-Turbo, GPT-4o*) that regularly update without public visibility or transparency. Although we cannot control these models, the model version used for our experiment is preserved to replicate as much as possible.

External Validity. The main threats to external validity in this study include limitations related to (a) LLMs, (b) the projects, (c) the dataset, and (d) the programming language investigated. Regarding LLMs, we utilized the most widely used state-of-the-art models in various sizes, considering our budget constraints. Including other LLM baselines might alter the observations from this study and will be worth exploring when they become available in the future. We selected five projects for this study, but these do not encompass all projects available. Concerning the dataset, we tried to mine all relevant information from the API documentation, GitHub, and StackOverflow. Nevertheless, there may be missing information that we failed to mine. We also limited the documents to those relevant to the APIs using string-matching heuristics. To make the experiment manageable in terms of budget and execution time, we used APIs that have at least 10 documents for GitHub issues and StackOverflow Q&As, as sufficient documents are needed for RAG to function. This approach excluded a significant amount of data from the experiment; however, with more resources and data, this could be further improved in future studies. Despite these limitations, we have devised sufficient projects, instances, and documents to demonstrate significant observations from the study.

6.6 Conclusion

This paper evaluates different RAG approaches for generating unit tests in ML/DL libraries, comparing Basic and API-level RAG using API docs, GitHub issues, and Stack Overflow. Our results indicate that GitHub issues are particularly effective, often outperforming other sources and basic instruction prompting in terms of code coverage. Combining RAG sources further improves results. Our manual analysis shows LLM-generated tests can find real bugs. While LLMs handle general cases without extra docs, adding unique external examples helps cover untested code lines. Future work includes improving retrieval algorithms for corner cases and exploring

broader knowledge sources like specification documents and technical blogs to enhance automated testing.

Chapter 7

Conclusions and Future Work

Although the current FMs and LLMs have significantly improved the effectiveness of automated software engineering, our studies have shown that with the correct tuning, traditional fine-tuning can be more effective. This dissertation takes a deep dive into different ASE tasks in terms of different inputs, models, and evaluations. Through comprehensive empirical investigations, our findings provide practical guidelines for both researchers and industry practitioners on how to tackle the two commonly utilized paradigms, prompt engineering and fine-tuning.

The structure of this chapter is as follows: first, we summarize our findings and contributions from this dissertation (Section 7.1). Next, we discuss potential future works (Section 7.2). Finally, we provide our closing remarks (Section 7.3).

7.1 Dissertation Findings and Contributions

- **Fine-Tuned Models on Machine Learning Tasks**: We conducted the first empirical evaluation of six state-of-the-art fine-tuned neural code generation models on ML programming tasks by curating an 83K example dataset of natural-language–code pairs from Scikit-Learn, Keras, TensorFlow, and PyTorch, and assessed their performance in terms of accuracy, syntactic correctness, semantic correctness, and developer usefulness. Our study yielded four key findings: (1) models achieve on average a 10.6-point higher BLEU-4 score on ML tasks versus non-ML tasks; (2) while syntactic correctness exceeds 90%, over 80% of generated programs are semantically incorrect; (3) generated snippets significantly aid API identification and improve developer correctness but do not reduce completion time; and (4) current models lack mechanisms to decompose generation into API-sequence identification and usage generation. Based on

these insights, we release our dataset and tooling, and propose actionable future directions: decomposed generation pipelines, human-in-the-loop collaboration, and domain-specific fine-tuning, to bridge the gap between neural generation and practical ML programming

- **Prompt Engineering vs. Fine-Tuning in Three ASE Tasks**: We present the first empirical comparison of automated prompting on *GPT-4* against 17 fine-tuned language models across code summarization, generation, and translation. Through quantitative evaluation on CodeXGLUE, HumanEval, and MBPP, we find that automated prompting alone does not consistently outperform fine-tuned models: GPT-4 trails by 28.3% points on MBPP, while a user study with 27 graduate students and ten industry practitioners shows that conversational prompting with human feedback can narrow this gap by up to 18.3% points. We further identify key gaps between automated and conversational prompts and release all study artifacts, including prompts, logs, and analysis scripts, to support replication and guide future research.

- **Fine-Tuned Models vs. LLMs in Test Oracle Generation**: We revisit the performance of four state-of-the-art neural oracle generation models on two categories of metrics, static textual similarity and dynamic test adequacy, to empirically assess their correlation. Surprisingly, we found no significant correlation between these metric categories, revealing that high textual similarity does not guarantee stronger testing effectiveness. Our manual analysis of 104 randomly selected oracle examples uncovered patterns behind metric mismatches, such as complex chained invocations and varied assertion types. Based on these insights, we offer guidelines for more comprehensive evaluation and release a benchmark dataset of nearly 30K executable methods with their corresponding metrics, along with all replication artifacts to support replication and extension of our study.

- **LLMs on Retrieval-Augmented Test Generation**: We introduce the first work to explore RAG-based LLMs for unit test generation on five widely-used Python ML/DL libraries (TensorFlow, PyTorch, Scikit-learn, JAX, XGBoost), evaluating four state-of-the-art models (GPT-3.5-Turbo, GPT-4o, Mistral MoE, Llama 3.1) across basic prompting, Basic RAG, and API-level RAG on 694 APIs with 19K GitHub issues and 22K StackOverflow Q&As. Quantitatively, RAG boosts line coverage by 6.5% points on average (with GitHub issues yielding the largest gains) and generates tests that uncover real bugs; 28 were detected, 24 unique ones reported (ten confirmed by developers). Through a

comprehensive manual analysis, we dissect how different knowledge sources guide test adequacy and bug detectability. Our contributions include the first investigation on RAG-driven unit test generation in ML/DL, systematically comparing three domain-specific sources, conducting an in-depth bug-detection study, and releasing a benchmark dataset with a replication package to support future research.

7.2 Future Work

- **Direction 1: Evaluation of Code Generation for Non-Functional Requirements:** While there are numerous metrics and benchmarks available to assess the functionality of code generation, there is a distinct lack of studies focused on evaluating non-functional requirements, which are equally important. In this direction, our goal is to define and implement rigorous benchmarks for several non-functional requirements: 1) Performance: This includes measuring runtime efficiency, execution time, memory consumption, and composite performance metrics such as Differential Performance Evaluation (DPE) [423, 424]. 2) Security: We aim to integrate automated vulnerability scanners and LLM-based detectors to evaluate the security compliance of generated code. This builds on recent work in LLM-driven vulnerability management [425]. 3) Maintainability: We seek to establish dynamic maintainability metrics that assess factors such as modularity, readability, and code health over time, similar to the framework demonstrated by MaintainCoder's dynamic assessment approach [426].

- **Direction 2: Generating Diverse Testing Types:** Most test generation scenarios primarily focus on unit test generation or fuzzing. However, there is a need for greater emphasis on enabling LLMs to generate various types of tests, such as: 1) Utilizing LLMs to create performance tests that replicate high-load and parallel scenarios, extending methods for high-performance computing unit tests to stress-test workflows [95]. 2) Developing automated techniques for acceptance and integration test generation, which translate user stories into executable scripts (e.g., from Gherkin to Cypress) to validate end-to-end system behaviour [427]. 3) Facilitating the automatic production of regression tests aimed at code changes to ensure that new commits do not introduce any regressions [428]. 4) Benchmarking project-level test generation pipelines to assess how LLMs can enhance and correct their own test outputs at a repository level [429].

- **Direction 3: Iterative Refinement of Test Generation for Code Agents**: While recent work has focused on generating and refining code with agents, there is a critical gap in automatically assessing the validity of the generated tests when the code under test is either unavailable or its correctness unknown. Empirical studies reveal that state-of-the-art models such as *GPT-4o* produce valid tests at rates as low as 71% on benchmarks like MBPP, meaning that nearly one-third of generated assertions may be hallucinated or semantically incorrect. Incorporating such invalid test cases into continuous integration pipelines risks masking real defects and undermining confidence in automated testing tools, underscoring the urgent need for dedicated validation frameworks to filter out or even correct invalid LLM-generated tests before evaluating their adequacy [405].
- **Direction 4: Enhancing automatic program repair (APR) Benchmarks through Rigorous Test Augmentation:** Although benchmarks for APR like SWE-bench [430] and RepairBench [431] received great attention, their reliance on a narrow set of manually mined tests leaves numerous plausible yet incorrect patches undetected, inflating repair success rates and undermining confidence in APR tools [432, 433]. By introducing an automated test-augmentation framework that systematically generates and validates new assertions grounded in both golden patches and code change contexts, we can fortify post-patch verification, expose hidden failures, and deliver more discriminative, trustworthy benchmarks. This direction is crucial for accurately gauging the repair effectiveness, driving the development of more robust APR techniques, and ensuring that future research addresses the real challenges of software reliability.

7.3 Closing Remarks

This dissertation has thoroughly examined both traditional fine-tuning methods and modern prompt-based strategies in key areas of ASE, including machine learning code generation to retrieval-augmented test generation. The findings indicate that neither approach alone is sufficient. Instead, a hybrid strategy that breaks down the generation process, incorporates human-in-the-loop feedback, and customizes retrieval contexts can leverage the strengths of both methods. These insights offer practitioners clear guidelines for selecting and fine-tuning ASE tools based on task demands and evaluation metrics. Looking ahead, expanding this study to include non-functional requirements, such as performance, security, and maintainability, and to generate diverse test types through iterative, chain-of-thought refinement holds

the potential to further close the gap between AI-driven automation and the complex realities of software engineering. t

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[8] J. Shin and J. Nam, "A survey of automatic code generation from natural language," *Journal of Information Processing Systems*, vol. 17, no. 3, pp. 537–555, 2021.

[9] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in neural information processing systems*, vol. 31, 2018.

[10] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[11] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From llms to llm-based agents for software engineering: A survey of current, challenges and future," *arXiv preprint arXiv:2408.02479*, 2024.

[12] S. Yang, O. Nachum, Y. Du, J. Wei, P. Abbeel, and D. Schuurmans, "Foundation models for decision making: Problems, methods, and opportunities," *arXiv preprint arXiv:2303.04129*, 2023.

[13] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

[14] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1162–1174.

[15] S. Hao, X. Shi, H. Liu, and Y. Shu, "Enhancing code language models for program repair by curricular fine-tuning framework," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 136–146.

[16] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng, "Exploring parameter-efficient fine-tuning of large language model on automated program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 719–731.

[17] J. Shin, S. Hashtroudi, H. Hemmati, and S. Wang, "Domain adaptation for code model-based unit test case generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1211–1222.

[18] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024.

[19] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, "A3test: Assertion-augmented automated test case generation," *Information and Software Technology*, vol. 176, p. 107565, 2024.

[20] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, H. Liu, X. Shi, and H. Jin, "Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning," in *ACL (Findings)*, 2024.

[21] A. Z. Yang, H. Tian, H. Ye, R. Martins, and C. L. Goues, "Security vulnerability detection with multitask self-instructed fine-tuning of large language models," *arXiv preprint arXiv:2406.05892*, 2024.

[22] A. Chan, A. Kharkar, R. Z. Moghaddam, Y. Mohylevskyy, A. Helyar, E. Kamal, M. Elkamhawy, and N. Sundaresan, "Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?" *arXiv preprint arXiv:2306.01754*, 2023.

[23] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[24] K. Ahn, X. Cheng, H. Daneshmand, and S. Sra, "Transformers learn to implement preconditioned gradient descent for in-context learning," *Advances in Neural Information Processing Systems*, vol. 36, pp. 45 614–45 650, 2023.

[25] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[26] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[27] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language

models," in *The Eleventh International Conference on Learning Representations*, 2023.

[28] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[29] F. Lin, D. J. Kim *et al.*, "When llm-based code generation meets the software development process," *arXiv preprint arXiv:2403.15852*, 2024.

[30] N. M. Guerreiro, E. Voita, and A. F. Martins, "Looking for a needle in a haystack: A comprehensive study of hallucinations in neural machine translation," in *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2023, pp. 1059–1075.

[31] J. A. Prenner and R. Robbes, "Making the most of small software engineering datasets with modern machine learning," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5050–5067, 2021.

[32] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J.-G. Lou, "Cert: Continual pre-training on sketches for library-oriented code generation," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, L. D. Raedt, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2022, pp. 2369–2375, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2022/329

[33] D. Zan, B. Chen, Z. Lin, B. Guan, W. Yongji, and J.-G. Lou, "When language model meets private library," in *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022, pp. 277–288.

[34] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[35] G. Yang, Y. Zhou, X. Chen, X. Zhang, Y. Xu, T. Han, and T. Chen, "A syntax-guided multi-task learning approach for turducken-style code generation," *Empirical Software Engineering*, vol. 28, no. 6, p. 141, 2023.

[36] D. Yan, Z. Gao, and Z. Liu, "A closer look at different difficulty levels code generation abilities of chatgpt," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1887–1898.

[37] M. Zhuge, C. Zhao, D. Ashley, W. Wang, D. Khizbullin, Y. Xiong, Z. Liu, E. Chang, R. Krishnamoorthi, Y. Tian *et al.*, "Agent-as-a-judge: Evaluate agents with agents," *arXiv preprint arXiv:2410.10934*, 2024.

[38] W. Zhang, M. Leon, R. Xu, A. Cardenas, A. Wissink, H. Martin, M. Srikanth, K. Dorogi, C. Valadez, P. Perez *et al.*, "Benchmarking llm code generation for audio programming with visual dataflow languages," *arXiv preprint arXiv:2409.00856*, 2024.

[39] D. Huang, G. Zeng, J. Dai, M. Luo, H. Weng, Y. Qing, H. Cui, Z. Guo, and J. M. Zhang, "Effi-code: Unleashing code efficiency in language models," *arXiv preprint arXiv:2410.10209*, 2024.

[40] D. Cherniavskii, P. Lippe, A. Zadaianchuk, and E. Gavves, "Stream: Embodied reasoning through code generation," in *Multi-modal Foundation Model meets Embodied AI Workshop@ ICML2024*, 2024.

[41] S. Zhang, J. Lu, and X. Tang, "Molecular subgraph representation learning based on spatial structure transformer," *Complex & Intelligent Systems*, vol. 10, no. 6, pp. 8197–8212, 2024.

[42] W. Liang, S. Wang, H.-J. Wang, O. Bastani, D. Jayaraman, and Y. J. Ma, "Environment curriculum generation via large language models," in *8th Annual Conference on Robot Learning*, 2024.

[43] X. Yin, C. Shi, Y. Han, and Y. Jiang, "Pear: A robust and flexible automation framework for ptychography enabled by multiple large language model agents," *arXiv preprint arXiv:2410.09034*, 2024.

[44] Z. Chen, S. Chen, Y. Ning, Q. Zhang, B. Wang, B. Yu, Y. Li, Z. Liao, C. Wei, Z. Lu *et al.*, "Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery," *arXiv preprint arXiv:2410.05080*, 2024.

[45] G. S. Black, B. P. Rimal, and V. M. Vaidyan, "Balancing security and correctness in code generation: An empirical study on commercial large language models," *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2024.

[46] M. Kessel, "Test-driven software experimentation with lasso: an llm benchmarking example," *arXiv preprint arXiv:2410.08911*, 2024.

[47] Y. Cui, "Insights from benchmarking frontier language models on web app code generation," *arXiv preprint arXiv:2409.05177*, 2024.

[48] M. Sarschar, G. Zhang, and A. Nowak, "Pacgbi: A pipeline for automated code generation from backlog items," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2338–2341.

[49] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[50] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[51] R. Tóth, T. Bisztray, and L. Erdődi, "Llms in web development: Evaluating llm-generated php code unveiling vulnerabilities and limitations," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2024, pp. 425–437.

[52] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[53] S. K. Lahiri, S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, P. Choudhury, C. von Veh, J. P. Inala, C. Wang *et al.*, "Interactive code generation via test-driven user-intent formalization," *arXiv preprint arXiv:2208.05950*, 2022.

[54] J.-B. D¨oderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting copilot and codex: Hot temperature, cold prompts, or black magic?" *arXiv preprint arXiv:2210.14699*, 2022.

[55] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 39–51.

[56] S. Wang, B. Lin, Z. Sun, M. Wen, Y. Liu, Y. Lei, and X. Mao, "Two birds with one stone: Boosting code generation and code search via a generative adversarial network," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 486–515, 2023.

[57] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, "Self-taught optimizer (stop): Recursively self-improving code generation," *arXiv preprint arXiv:2310.02304*, 2023.

[58] S. Zhang, Z. Chen, Y. Shen, M. Ding, J. B. Tenenbaum, and C. Gan, "Planning with large language models for code generation," in *The Eleventh International Conference on Learning Representations*, 2023.

[59] T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang, "Coder reviewer reranking for code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 41 832–41 846.

[60] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: Fault-aware code editor for code generation," in *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.

[61] R. Yen, J. Zhu, S. Suh, H. Xia, and J. Zhao, "Coladder: Supporting programmers with hierarchical code generation in multi-level abstraction," *arXiv preprint arXiv:2310.08699*, 2023.

[62] W. Zheng, S. Sharan, A. K. Jaiswal, K. Wang, Y. Xi, D. Xu, and Z. Wang, "Outline, then details: Syntactically guided coarse-to-fine code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 42 403–42 419.

[63] N. Cloos, M. G. Kumar, A. Manoogian, C. J. Cueva, and S. A. Rhoads, "Automating and validating agent and environment code generation with large language models," in *NeurIPS 2024 Workshop on Behavioral Machine Learning*, 2024.

[64] S. I. Salim, R. Y. Yang, A. Cooper, S. Ray, S. Debray, and S. Rahaman, "Impeding llm-assisted cheating in introductory programming assignments via adversarial perturbations," *arXiv preprint arXiv:2410.09318*, 2024.

[65] S. Ugare, R. Gumaste, T. Suresh, G. Singh, and S. Misailovic, "Itergen: Iterative structured llm generation," *arXiv preprint arXiv:2410.07295*, 2024.

[66] Y. Wu, P. He, Z. Wang, S. Wang, Y. Tian *et al.*, "Autoapieval: A framework for automated evaluation of llms in api-oriented code generation," *arXiv preprint arXiv:2409.15228*, 2024.

[67] Q. Li, W. Xia, K. Du, X. Dai, R. Tang, Y. Wang, Y. Yu, and W. Zhang, "Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation," *arXiv preprint arXiv:2409.09584*, 2024.

[68] Z. Rasheed, M. Waseem, M. Saari, K. Syst¨a, and P. Abrahamsson, "Codepori: Large scale model for autonomous software development by using multi-agents," *arXiv preprint arXiv:2402.01411*, 2024.

[69] T. Xue, X. Li, T. Azim, R. Smirnov, J. Yu, A. Sadrieh, and B. Pahlavan, "Multi-programming language ensemble for code generation in large language model," *arXiv preprint arXiv:2409.04114*, 2024.

[70] Y. Xie, A. Goyal, X. Wu, X. Yin, X. Xu, M.-Y. Kan, L. Pan, and W. Y. Wang, "Coral: Order-agnostic language modeling for efficient iterative refinement," *arXiv preprint arXiv:2410.09675*, 2024.

[71] A. Kabir, S. Wang, Y. Tian, M. Asaduzzaman, W. Zhang *et al.*, "Zs4c: Zero-shot synthesis of compilable code for incomplete code snippets using chatgpt," *arXiv preprint arXiv:2401.14279*, 2024.

[72] E. Wang, F. Cassano, C. Wu, Y. Bai, W. Song, V. Nath, Z. Han, S. Hendryx, S. Yue, and H. Zhang, "Planning in natural language improves llm search for code generation," *arXiv preprint arXiv:2409.03733*, 2024.

[73] C. Zhu-Tian, Z. Xiong, X. Yao, and E. Glassman, "Sketch then generate: Providing incremental user feedback and guiding llm code generation through language-oriented code sketches," *arXiv preprint arXiv:2405.03998*, 2024.

[74] T. T. Quoc, D. H. Minh, T. Q. Thanh, and A. Nguyen-Duc, "An empirical study on self-correcting large language models for data science code generation," *arXiv preprint arXiv:2408.15658*, 2024.

[75] L. Zheng, J. Yuan, Z. Zhang, H. Yang, and L. Kong, "Self-infilling code generation," in *Forty-first International Conference on Machine Learning*, 2024.

[76] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, "Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing," *arXiv preprint arXiv:2409.10737*, 2024.

[77] W. Wang, K. Liu, A. R. Chen, G. Li, Z. Jin, G. Huang, and L. Ma, "Python symbolic execution with llm-powered code generation," *arXiv preprint arXiv:2409.09271*, 2024.

[78] Y. Zhu, J. Li, G. Li, Y. Zhao, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 1, 2024, pp. 437–445.

[79] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1583–1594.

[80] K. Yang, X. Mao, S. Wang, T. Zhang, B. Lin, Y. Wang, Y. Qin, Z. Zhang, and X. Mao, "Enhancing code intelligence tasks with chatgpt," *arXiv preprint arXiv:2312.15202*, 2023.

[81] L. Zhong and Z. Wang, "Can chatgpt replace stackoverflow? a study on robustness and reliability of large language model code generation," *arXiv preprint arXiv:2308.10335*, 2023.

[82] R. Pudari and N. A. Ernst, "From copilot to pilot: Towards ai supported software development," *arXiv preprint arXiv:2303.04142*, 2023.

[83] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13 921–13 937.

[84] B. Yetiştiren, I. ¨Ozsoy, M. Ayerdem, and E. T¨uz¨un, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023.

[85] S. Sun, "Enhancing software design and developer experience via llms," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2498–2501.

[86] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 54–64.

[87] Y. Zhang, W. Song, Z. Ji, N. Meng *et al.*, "How well does llm generate security tests?" *arXiv preprint arXiv:2310.00710*, 2023.

[88] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[89] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2023.

[90] Z. Zhong, S. Wang, H. Wang, S. Wen, H. Guan, Y. Tao, and Y. Liu, "Advancing bug detection in fastjson2 with large language models driven unit test generation," *arXiv preprint arXiv:2410.09414*, 2024.

[91] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, "Automatic generation of test cases based on bug reports: a feasibility study with large language models," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 360–361.

[92] H. Kirinuki and H. Tanno, "Chatgpt and human synergy in black-box testing: A comparative analysis," *arXiv preprint arXiv:2401.13924*, 2024.

[93] N. Mündler, M. N. Müller, J. He, and M. Vechev, "Code agents are state of the art software testers," in *ICML 2024 Workshop on LLMs and Cognition*, 2024.

[94] M. Li, D. Li, J. Liu, J. Cao, Y. Tian, and S.-C. Cheung, "Dllens: Testing deep learning libraries via llm-aided synthesis," *arXiv preprint arXiv:2406.07944*, 2024.

[95] R. Karanjai, A. Hussain, M. R. I. Rabin, L. Xu, W. Shi, and M. A. Alipour, "Harnessing the power of llms: Automating unit test generation for high-performance computing," *arXiv preprint arXiv:2407.05202*, 2024.

[96] C. Cui, T. Li, J. Wang, C. Chen, D. Towey, and R. Huang, "Large language models for mobile gui text input generation: An empirical study," *arXiv preprint arXiv:2404.08948*, 2024.

[97] T. Li, C. Cui, R. Huang, D. Towey, and L. Ma, "Large language models for automated web-form-test generation: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, 2025. [Online]. Available: https://doi.org/10.1145/3735553

[98] Z. Xue, L. Li, S. Tian, X. Chen, P. Li, L. Chen, T. Jiang, and M. Zhang, "Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1643–1655.

[99] M. Nabeel, D. D. Nimara, and T. Zanouda, "Test code generation for telecom software systems using two-stage generative model," in *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2024, pp. 1231–1236.

[100] C. Arora, T. Herda, and V. Homm, "Generating test scenarios from nl requirements using retrieval-augmented llms: An industrial study," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024, pp. 240–251.

[101] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," *arXiv preprint arXiv:2009.05617*, 2020.

[102] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," in *The Eleventh International Conference on Learning Representations*, 2023.

[103] V. Li and N. Doiron, "Prompting code interpreter to write better unit tests on quixbugs functions," *arXiv preprint arXiv:2310.00483*, 2023.

[104] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, "Reinforcement learning from automatic feedback for high-quality unit test generation," in *2025 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*. IEEE, 2025, pp. 37–44.

[105] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, "Algo: Synthesizing algorithmic programs with generated oracle verifiers," *Advances in Neural Information Processing Systems*, vol. 36, pp. 54 769–54 784, 2023.

[106] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[107] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.

[108] W. Xiong, Y. Guo, and H. Chen, "The program testing ability of large language models for code," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 2024, pp. 23–34.

[109] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, "Cat-lm training language models on aligned code and tests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 409–420.

[110] A. Lops, F. Narducci, A. Ragone, M. Trizio, and C. Bartolini, "A system for automated unit test generation using large language models and assessment of generated test suites," in *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2025, pp. 29–36.

[111] C. Ni, X. Wang, L. Chen, D. Zhao, Z. Cai, S. Wang, and X. Yang, "Casmodatest: A cascaded and model-agnostic self-directed framework for unit test generation," *arXiv preprint arXiv:2406.15743*, 2024.

[112] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[113] J. Altmayer Pizzorno and E. D. Berger, "Coverup: Effective high coverage test generation for python," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2897–2919, 2025.

[114] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing llm-based test generation for hard-to-cover branches via program analysis," *arXiv preprint arXiv:2404.04966*, 2024.

[115] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[116] Y. He, J. Wang, Y. Rong, and H. Chen, "Exploring fuzzing as data augmentation for neural test generation," *arXiv preprint arXiv:2406.08665*, 2024.

[117] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1258–1268.

[118] K. Li and Y. Yuan, "Large language models as test case generators: Performance evaluation and enhancement," *arXiv preprint arXiv:2404.13340*, 2024.

[119] M. Jiri, B. Emese, and P. Medlen, "Leveraging large language models for python unit test," in *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2024, pp. 95–100.

[120] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, "Llm-powered test case generation for detecting tricky bugs," *arXiv preprint arXiv:2404.10304*, 2024.

[121] K. Etemadi, B. Mohammadi, Z. Su, and M. Monperrus, "Mokav: Execution-driven differential testing with llms," *Journal of Systems and Software*, p. 112571, 2025.

[122] H. Karmarkar, S. Agrawal, A. Chauhan, and P. Shete, "Navigating confidentiality in test automation: A case study in llm driven test data generation," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 337–348.

[123] D. Xiao, Y. Guo, Y. Li, and L. Chen, "Optimizing search-based unit test generation with large language models: An empirical study," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 2024, pp. 71–80.

[124] S. Gu, C. Fang, Q. Zhang, F. Tian, and Z. Chen, "Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration," *arXiv preprint arXiv:2408.03095*, 2024.

[125] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24.

[126] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, "Using large language models to generate junit tests: An empirical study," in *Proceedings of the 28th international conference on evaluation and assessment in software engineering*, 2024, pp. 313–322.

[127] V. Vikram, C. Lemieux, and R. Padhye, "Can large language models write good property-based tests?" *arXiv preprint arXiv:2307.04346*, 2023.

[128] Z. Zhou, Y. Tang, Y. Lin, and J. He, "An llm-based readability measurement for unit tests' context-aware inputs," *arXiv preprint arXiv:2407.21369*, 2024.

[129] W. C. Ouédraogo, K. Kaboré, H. Tian, Y. Song, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Large-scale, independent and comprehensive study of the power of llms for test case generation," *arXiv preprint arXiv:2407.00225*, 2024.

[130] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai: A comparative performance analysis of autogeneration tools," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 54–61.

[131] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529

[132] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "Chatgpt vs sbst: A comparative assessment of unit test suite generation," *IEEE Transactions on Software Engineering*, 2024.

[133] D. Huang, J. M. Zhang, M. Du, M. Harman, and H. Cui, "Rethinking the influence of source code on test case generation," *arXiv preprint arXiv:2409.09464*, 2024.

[134] W. C. Ouédraogo, Y. Li, K. Kaboré, X. Tang, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Test smells in llm-generated unit tests," *arXiv preprint arXiv:2410.10628*, 2024.

[135] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[136] W. Sun, C. Fang, Y. You, Y. Chen, Y. Liu, C. Wang, J. Zhang, Q. Zhang, H. Qian, W. Zhao *et al.*, "A prompt learning framework for source code summarization," *arXiv preprint arXiv:2312.16066*, 2023.

[137] I. Saberi, A. Esmaeili, F. Fard, and F. Chen, "Advfusion: Adapter-based knowledge transfer for code summarization on code language models," in *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2025, pp. 563–574.

[138] S. A. Rukmono, L. Ochoa, and M. R. Chaudron, "Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis," in *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2023, pp. 7–12.

[139] C. Wang, Y. Lou, J. Liu, and X. Peng, "Generating variable explanations via zero-shot prompt learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 748–760.

[140] E. Shi, F. Zhang, Y. Wang, B. Chen, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Sotana: The open-source software development assistant," *arXiv preprint arXiv:2308.13416*, 2023.

[141] J. Zhao, X. Chen, G. Yang, and Y. Shen, "Automatic smart contract comment generation via large language models and in-context learning," *Information and Software Technology*, vol. 168, p. 107405, 2024.

[142] Y. Wang, Y. Huang, D. Guo, H. Zhang, and Z. Zheng, "Sparsecoder: Identifier-aware sparse transformer for file-level code summarization," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 614–625.

[143] Y. Wang, X. Li, T. N. Nguyen, S. Wang, C. Ni, and L. Ding, "Natural is the best: Model-agnostic code simplification for pre-trained large language models," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 586–608, 2024.

[144] Y. Virk, P. Devanbu, and T. Ahmed, "Calibration of large language models on code summarization," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2944–2964, 2025.

[145] B. Szalontai, G. Szalay, T. Márton, A. Sike, B. Pintér, and T. Gregorics, "Large language models for code summarization," 2024. [Online]. Available: https://arxiv.org/abs/2405.19032

[146] C.-Y. Su, A. Bansal, Y. Huang, T. J.-J. Li, and C. McMillan, "Context-aware code summary generation," *arXiv preprint arXiv:2408.09006*, 2024.

[147] C.-Y. Su and C. McMillan, "Semantic similarity loss for neural source code summarization," *Journal of Software: Evolution and Process*, p. e2706, 2024.

[148] K. Shi, D. Altınbüken, S. Anand, M. Christodorescu, K. Grünwedel, A. Koenings, S. Naidu, A. Pathak, M. Rasi, F. Ribeiro *et al.*, "Natural language outlines for code: Literate programming in the llm era," *arXiv preprint arXiv:2408.04820*, 2024.

[149] Y. Shen, X. Ju, X. Chen, and G. Yang, "Bash comment generation via data augmentation and semantic-aware codebert," *Automated Software Engineering*, vol. 31, no. 1, p. 30, 2024.

[150] B. Poudel, A. Cook, S. Traore, and S. Ameli, "Documint: Docstring generation for python using small language models," *arXiv preprint arXiv:2405.10243*, 2024.

[151] S. Pordanesh and B. Tan, "Exploring the efficacy of large language models (gpt-4) in binary reverse engineering," *arXiv preprint arXiv:2406.06637*, 2024.

[152] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, no. 10, 2022, pp. 11 783–11 790.

[153] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, "On the evaluation of neural code translation: Taxonomy and benchmark," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1529–1541.

[154] B. Wang, R. Li, M. Li, and P. Saxena, "Transmap: Pinpointing mistakes in neural code translation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 999–1011.

[155] Z. Tang, M. Agarwal, A. G. Shypula, B. Wang, D. Wijaya, J. Chen, and Y. Kim, "Explain-then-translate: an analysis on improving program translation with self-generated explanations," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[156] M. Qi, Y. Huang, M. Wang, Y. Yao, Z. Liu, B. Gu, C. Clement, and N. Sundaresan, "Sut: Active defects probing for transcompiler models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 14 024–14 034.

[157] Y. Huang, M. Qi, Y. Yao, M. Wang, B. Gu, C. Clement, and N. Sundaresan, "Program translation via code distillation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 10 903–10 914.

[158] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[159] M. Xue, A. Andrzejak, and M. Leuther, "An interpretable error correction method for enhancing code-to-code translation," in *The Twelfth International Conference on Learning Representations*, 2024.

[160] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[161] X. Li, S. Yuan, X. Gu, Y. Chen, and B. Shen, "Few-shot code translation via task-adapted prompt learning," *Journal of Systems and Software*, vol. 212, p. 112002, 2024.

[162] Y. Luo, R. Yu, F. Zhang, L. Liang, and Y. Xiong, "Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2448–2449.

[163] A. Kumar, D. Saha, T. Yasue, K. Ono, S. Krishnan, S. Hans, F. Satoh, G. Mitchell, and S. Kumar, "Automated validation of cobol to java transformation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2415–2418.

[164] R. Liu, Y. Lin, Y. Hu, Z. Zhang, and X. Gao, "Llm-based java concurrent program to arkts converter," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2403–2406.

[165] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 678–690.

[166] J. Zhang, T. Mytkowicz, M. Kaufman, R. Piskac, and S. K. Lahiri, "Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 77–88.

[167] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 535–547.

[168] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 522–534.

[169] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[170] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[171] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 642–646.

[172] Y. Zhao, Z. Huang, Y. Ma, R. Li, K. Zhang, H. Jiang, Q. Liu, L. Zhu, and Y. Su, "Repair: Automated program repair with process-based feedback," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 16 415–16 429.

[173] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Pydex: Repairing bugs in introductory python assignments using llms," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 1100–1124, 2024.

[174] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT*

International Symposium on Software Testing and Analysis, 2024, pp. 1274–1286.

[175] B. Yang, H. Tian, W. Pian, H. Yu, H. Wang, J. Klein, T. F. Bissyandé, and S. Jin, "Cref: An llm-based conversational software repair framework for programming tutors," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 882–894.

[176] Q. Xin, H. Wu, J. Tang, X. Liu, S. P. Reiss, and J. Xuan, "Detecting, creating, repairing, and understanding indivisible multi-hunk bugs," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2747–2770, 2024.

[177] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1471–1493, 2024.

[178] Y. Ouyang, J. Yang, and L. Zhang, "Benchmarking automated program repair: An extensive study on both real-world and artificial bugs," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 440–452.

[179] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 33–43.

[180] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 7–12.

[181] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

[182] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2021.

[183] S. Norouzi, K. Tang, and Y. Cao, "Code generation from natural language with less prior knowledge and more monolingual data," in *Proceedings of the*

59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), 2021, pp. 776–785.

[184] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2022.

[185] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, A. Lopez Garcia, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[186] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1548–1560.

[187] Z. Yao, D. S. Weld, W.-P. Chen, and H. Sun, "Staqc: A systematically mined question-code dataset from stack overflow," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1693–1703.

[188] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 2018, pp. 476–486.

[189] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 574–584.

[190] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1268–1289, 2022.

[191] R. Agashe, S. Iyer, and L. Zettlemoyer, "Juice: A large scale distantly supervised dataset for open domain context-based code generation," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 5436–5446.

[192] S. Dahal, A. Maharana, and M. Bansal, "Analysis of tree-structured architectures for code generation," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 4382–4391.

[193] C. Lyu, R. Wang, H. Zhang, H. Zhang, and S. Hu, "Embedding api dependency graph for neural code generation," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–51, 2021.

[194] A. Gatt and E. Krahmer, "Survey of the state of the art in natural language generation: Core tasks, applications and evaluation," *Journal of Artificial Intelligence Research*, vol. 61, pp. 65–170, 2018.

[195] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2017. [Online]. Available: https://openreview.net/forum?id=rJbPBt9lg

[196] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[197] A. E. Cozzie and S. King, "Macho: Writing programs with natural language and examples," *UIUC-TechReport*, 2012. [Online]. Available: http://hdl.handle.net/2142/33791

[198] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 731–742.

[199] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[200] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," in *ACL (Findings)*, 2023.

[201] C. Chen, X. Peng, Z. Xing, J. Sun, X. Wang, Y. Zhao, and W. Zhao, "Holistic combination of structural and textual code information for context based api recommendation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, p. 2987–3009, aug 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3074309

[202] M. Wei, Y. Huang, J. Wang, J. Shin, N. S. Harzevili, and S. Wang, "Api recommendation for machine learning libraries: how far are we?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 370–381.

[203] M. S. Nawaz, S. U. R. Khan, S. Hussain, and J. Iqbal, "A study on application programming interface recommendation: state-of-the-art techniques, challenges and future directions," *Library Hi Tech*, vol. 41, no. 2, pp. 355–385, 2023.

[204] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[205] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[206] S. Khalid, T. Khalil, and S. Nasreen, "A survey of feature selection and feature extraction techniques in machine learning," in *2014 science and information conference*. IEEE, 2014, pp. 372–378.

[207] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[208] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[209] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of NAACL-HLT*, 2019, pp. 3931–3937.

[210] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th working conference on mining software repositories (MSR)*. IEEE, 2013, pp. 207–216.

[211] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[212] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[213] H. Jiang, C. Zhou, F. Meng, B. Zhang, J. Zhou, D. Huang, Q. Wu, and J. Su, "Exploring dynamic selection of branch expansion orders for code generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 5076–5085.

[214] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," *Software*, Mar. 2021, If you use this software, please cite it using these metadata. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[215] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[216] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[217] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[218] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[219] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[220] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark

dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[221] W. Xie, X. Peng, M. Liu, C. Treude, Z. Xing, X. Zhang, and W. Zhao, "Api method recommendation via explicit matching of functionality verb phrases," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1015–1026.

[222] F. Wilcoxon, "Individual comparisons by ranking methods. biom. bull., 1, 80–83," 1945.

[223] J. Cohen, "Statistical power analysis for the behavioral 698 sciences," *Statistical Power Analysis for the Behavioral Sciences*, vol. 2, p. 495, 1988.

[224] G. Uddin, F. Khomh, and C. K. Roy, "Mining api usage scenarios from stack overflow," *Information and Software Technology*, vol. 122, p. 106277, 2020.

[225] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[226] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," in *International Conference on Learning Representations*, 2018.

[227] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 440–450.

[228] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 599–609.

[229] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1139–1149.

[230] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

[231] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[232] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655–2668.

[233] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[234] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[235] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 357–367.

[236] X. Liu, L. Huang, and V. Ng, "Effective api recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 282–292.

[237] C. Chen, X. Peng, B. Chen, J. Sun, Z. Xing, X. Wang, and W. Zhao, ""more than deep learning": post-processing for api sequence recommendation," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–32, 2022.

[238] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 703–715.

[239] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[240] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.

[241] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. R. Lyu, "Revisiting, benchmarking and exploring api recommendation: How far are we?" *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1876–1897, 2023.

[242] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 297–308.

[243] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[244] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2019.

[245] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[246] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[247] J. Shin, M. Wei, J. Wang, L. Shi, and S. Wang, "The good, the bad, and the missing: Neural code generation for machine learning tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–24, 2023.

[248] J. Shin, R. Aleithan, H. Hemmati, and S. Wang, "Retrieval-augmented test generation: How far are we?" *arXiv preprint arXiv:2409.12682*, 2024.

[249] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing evaluation metrics for neural test oracle generation," *IEEE Transactions on Software Engineering*, 2024.

[250] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[251] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, "Code search based on context-aware code translation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 388–400.

[252] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[253] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[254] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International conference on machine learning*. PMLR, 2020, pp. 5110–5121.

[255] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[256] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1443–1455.

[257] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 27 730–27 744.

[Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[258] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[259] H. T. et al., "Llama 2: Open foundation and fine-tuned chat models," 2023.

[260] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[261] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[262] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6.

[263] S. Gao, X.-C. Wen, C. Gao, W. Wang, and M. R. Lyu, "Constructing effective in-context demonstration for code intelligence tasks: An empirical study," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[264] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2023.

[265] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[266] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[267] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for gpt-3?" in *Deep Learning Inside Out: 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, DeeLIO 2022*. Association for Computational Linguistics (ACL), 2022, pp. 100–114.

[268] Y. He, S. Zheng, Y. Tay, J. Gupta, Y. Du, V. Aribandi, Z. Zhao, Y. Li, Z. Chen, D. Metzler *et al.*, "Hyperprompt: Prompt-based task-conditioning of transformers," in *International Conference on Machine Learning*. PMLR, 2022, pp. 8678–8690.

[269] S. Carta, A. Giuliani, L. Piano, A. S. Podda, L. Pompianu, and S. G. Tiddia, "Iterative zero-shot llm prompting for knowledge graph construction," *arXiv preprint arXiv:2307.01128*, 2023.

[270] T. Shin, Y. Razeghi, R. L. Logan IV, E. Wallace, and S. Singh, "Autoprompt: Eliciting knowledge from language models with automatically generated prompts," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, pp. 4222–4235.

[271] K. Hambardzumyan, H. Khachatrian, and J. May, "Warp: Word-level adversarial reprogramming," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 4921–4933.

[272] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, "Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[273] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–17.

[274] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[275] F. Trad and A. Chehab, "Prompt engineering or fine-tuning? a case study on phishing detection with large language models," *Machine Learning and Knowledge Extraction*, vol. 6, no. 1, pp. 367–384, 2024.

[276] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–28, 2025.

[277] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–2111.

[278] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[279] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[280] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "On the evaluation of neural code summarization," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1597–1608.

[281] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2332–2354, 2024.

[282] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[283] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[284] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[285] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[286] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[287] L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian, and Y. Ye, "Cotext: Multi-task learning with code-text transformer," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 40–47.

[288] W. Qi, Y. Gong, Y. Yan, C. Xu, B. Yao, B. Zhou, B. Cheng, D. Jiang, J. Chen, R. Zhang, H. Li, and N. Duan, "ProphetNet-X: Large-scale pre-training models for English, Chinese, multi-lingual, dialog, and code generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*, Online, Aug. 2021, pp. 232–239. [Online]. Available: https://aclanthology.org/2021.acl-demo.28

[289] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao *et al.*, "Pangu-coder2: Boosting large language models for code with ranking feedback," *arXiv preprint arXiv:2307.14936*, 2023.

[290] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, vol. 2023, 2023.

[291] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*, 2024.

[292] Y. Wang, K. He, D. Fu, Z. Gongque, H. Xu, Y. Chen, Z. Wang, Y. Fu, G. Dong, M. Diao *et al.*, "How do your code llms perform? empowering code instruction tuning with really good data," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 14 027–14 043.

[293] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[294] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[295] S. Tipirneni, M. Zhu, and C. K. Reddy, "Structcoder: Structure-aware transformer for code generation," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 3, pp. 1–20, 2024.

[296] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[297] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote sensing of Environment*, vol. 62, no. 1, pp. 77–89, 1997.

[298] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan, "Improving chatgpt prompt for code generation," *arXiv preprint arXiv:2305.08360*, 2023.

[299] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[300] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[301] B. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 17–20, 2002.

[302] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough? an experiment with data saturation and variability," *Field methods*, vol. 18, no. 1, pp. 59–82, 2006.

[303] M. B. Miles and A. M. Huberman, *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[304] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd*

ACM International Conference on the Foundations of Software Engineering, 2024, pp. 185–196.

[305] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, "Harnessing the power of llms in practice: A survey on chatgpt and beyond," *ACM Transactions on Knowledge Discovery from Data*, 2023.

[306] Y. Chen, Q. Fu, Y. Yuan, Z. Wen, G. Fan, D. Liu, D. Zhang, Z. Li, and Y. Xiao, "Hallucination detection: Robustly discerning reliable answers in large language models," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 2023, pp. 245–255.

[307] N. M. Guerreiro, D. M. Alves, J. Waldendorf, B. Haddow, A. Birch, P. Colombo, and A. F. Martins, "Hallucinations in large multilingual translation models," *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 1500–1517, 2023.

[308] T. Wu, E. Jiang, A. Donsbach, J. Gray, A. Molina, M. Terry, and C. J. Cai, "Promptchainer: Chaining large language model prompts through visual programming," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–10.

[309] T. Wu, M. Terry, and C. J. Cai, "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts," in *Proceedings of the 2022 CHI conference on human factors in computing systems*, 2022, pp. 1–22.

[310] D. Trautmann, "Large language model prompt chaining for long legal document classification," *SwissText'23: The 8th edition of the Swiss Text Analytics Conference – Generative AI & LLM, June 12–14, 2023, Neuchâtel, Switzerland*, 2023.

[311] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, pp. 2179–2217, 2020.

[312] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 931–937.

[313] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, and A. Ouni, "How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 202–206.

[314] J. Chen, H. Lin, X. Han, and L. Sun, "Benchmarking large language models in retrieval-augmented generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, 2024, pp. 17 754–17 762.

[315] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[316] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 643–653.

[317] L. Saes, "Unit test generation using machine learning," *Universiteit van Amsterdamg*, 2018.

[318] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2test: A dataset of focal methods mapped to test cases," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 299–303.

[319] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1398–1409.

[320] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated assertion generation via information retrieval and its integration with deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 163–174.

[321] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "Toga: A neural method for test oracle generation," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[322] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660783

[323] I. Accuracy, "of measurement methods and results—part 1: General principles and definitions," *International Organization for Standardization, Geneva, Switzerland*, 1994.

[324] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[325] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[326] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[327] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2012.

[328] M. Barani, Y. Labiche, and A. Rollet, "On factors that impact the relationship between code coverage and test suite effectiveness: a survey," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 381–388.

[329] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine-tuning: An empirical assessment of llms for code," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 490–502.

[330] T. Tanimoto, *An Elementary Mathematical Theory of Classification and Prediction*. International Business Machines Corporation, 1958. [Online]. Available: https://books.google.ca/books?id=yp34HAAACAAJ

[331] M. Vijaymeena and K. Kavitha, "A survey on similarity measures in text mining," *Machine Learning and Applications: An International Journal*, vol. 3, no. 2, pp. 19–28, 2016.

[332] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[333] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493–510.

[334] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

[335] R. A. Silva, S. do Rocio Senger de Souza, and P. S. Lopes de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, no. 0950-5849, pp. 19–35, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916300167

[336] A. B. Sánchez, J. A. Parejo, S. Segura, A. Durán, and M. Papadakis, "Mutation testing in practice: Insights from open-source software developers," *IEEE Transactions on Software Engineering*, 2024.

[337] R. White and J. Krinke, "Reassert: Deep learning for assert generation," *arXiv preprint arXiv:2011.09784*, 2020.

[338] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1580–1598, 2022.

[339] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2111–2123.

[340] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring llm-based general bug reproduction," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2312–2323.

[341] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 589–600.

[342] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 70–81.

[343] OpenAI. (2023) Chatgpt. [Online]. Available: https://openai.com/chatgpt

[344] N. Tran, H. Tran, S. Nguyen, H. Nguyen, and T. Nguyen, "Does bleu score work for code migration?" in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 165–176.

[345] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[346] M. Popović, "chrf deconstructed: beta parameters and n-gram weights," in *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, 2016, pp. 499–504.

[347] M. M. Tikir and J. K. Hollingsworth, "Efficient instrumentation for code coverage testing," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 86–96, 2002.

[348] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005*. Springer, 2005, pp. 504–527.

[349] J. P. Lim and H. Lauw, "Large-scale correlation analysis of automated metrics for topic models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 13 874–13 898.

[350] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1105–1116.

[351] A. Shimorina, "Human vs automatic metrics: on the importance of correlation design," *CoRR*, vol. abs/1805.11474, 2018. [Online]. Available: http://arxiv.org/abs/1805.11474

[352] Y. Graham, T. Baldwin, and N. Mathur, "Accurate evaluation of segment-level machine translation metrics," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2015, pp. 1183–1191.

[353] Y. Liu, Y. Zhou, S. Wen, and C. Tang, "A strategy on selecting performance metrics for classifier evaluation," *International Journal of Mobile Computing and Multimedia Communications (IJMCMC)*, vol. 6, no. 4, pp. 20–35, 2014.

[354] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[355] P. Schober, C. Boer, and L. A. Schwarte, "Correlation coefficients: appropriate use and interpretation," *Anesthesia & analgesia*, vol. 126, no. 5, pp. 1763–1768, 2018.

[356] S. S. Smith, "Scope and methods of political science," *Teaching Political Science*, vol. 9, no. 1, pp. 60–62, 1981.

[357] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 287–298.

[358] D. Peters and D. L. Parnas, "Generating a test oracle from program documentation: work in progress," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, 1994, pp. 58–65.

[359] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, "Lutess: a specification-driven testing environment for synchronous software," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 267–276.

[360] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test oracle," *Information and Software Technology*, vol. 53, no. 7, pp. 774–788, 2011.

[361] S. Liu and S. Nakajima, "Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 691–712, 2020.

[362] N. Mathur, T. Baldwin, and T. Cohn, "Tangled up in bleu: Reevaluating the evaluation of automatic machine translation evaluation metrics," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 4984–4997.

[363] R. Takaichi, Y. Higo, S. Matsumoto, S. Kusumoto, T. Kurabayashi, H. Kirinuki, and H. Tanno, "Are nlp metrics suitable for evaluating generated code?" in *International Conference on Product-Focused Software Process Improvement*. Springer, 2022, pp. 531–537.

[364] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *Journal of Systems and Software*, vol. 203, p. 111741, 2023.

[365] P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo, "Who evaluates the evaluators? on automatic metrics for assessing ai-based offensive code generators," *Expert Systems with Applications*, vol. 225, p. 120073, 2023.

[366] S. Sikand, K. K. Phokela, V. S. Sharma, K. Singi, V. Kaulgud, T. Tung, P. Sharma, and A. P. Burden, "How much space do metrics have in genai assisted software development?" in *Proceedings of the 17th Innovations in Software Engineering Conference*, 2024, pp. 1–5.

[367] H. Su, S. Jiang, Y. Lai, H. Wu, B. Shi, C. Liu, Q. Liu, and T. Yu, "Arks: Active retrieval in knowledge soup for code generation," *arXiv preprint arXiv:2402.12317*, 2024.

[368] S. J. Rani, S. Deepika, D. Devdharshini, and H. Ravindran, "Augmenting code sequencing with retrieval-augmented generation (rag) for context-aware code synthesis," in *2024 First International Conference on Software, Systems and Information Technology (SSITCON)*. IEEE, 2024, pp. 1–7.

[369] A. D. Gotmare, J. Li, S. Joty, and S. C. Hoi, "Efficient text-to-code retrieval with cascaded fast and slow transformer models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 388–400.

[370] L. Zhang, H. Zhang, C. Wang, and P. Liang, "Rag-enhanced commit message generation," *arXiv preprint arXiv:2406.05514*, 2024.

[371] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[372] H. Tan, Q. Luo, L. Jiang, Z. Zhan, J. Li, H. Zhang, and Y. Zhang, "Prompt-based code completion via multi-retrieval augmented generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[373] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, and Q. Wang, "Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model," *arXiv preprint arXiv:2406.07003*, 2024.

[374] H. A. Quddus, M. S. Hossain, Z. Cevahir, A. Jesser, and M. N. Amin, "Enhanced vlsi assertion generation: Conforming to high-level specifications and reducing llm hallucinations with rag," in *DVCon Europe 2024; Design and Verification Conference and Exhibition Europe*. VDE, 2024, pp. 57–62.

[375] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 146–158.

[376] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, 2023.

[377] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[378] A. Daghighfarsoodeh, C.-Y. Wang, H. Taherkhani, M. Sepidband, M. Abdollahi, H. Hemmati, and H. V. Pham, "Deep-bench: Deep learning benchmark dataset for code generation," *arXiv preprint arXiv:2502.18726*, 2025.

[379] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Keeper: Automated testing and fixing of machine learning software," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–33, 2024.

[380] C. S. Xia, S. Dutta, S. Misailovic, D. Marinov, and L. Zhang, "Balancing effectiveness and flakiness of non-deterministic machine learning tests," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1801–1813.

[381] N. Baumann, J. S. Diaz, J. Michael, L. Netz, H. Nqiri, J. Reimer, and B. Rumpe, "Combining retrieval-augmented generation and few-shot learning for model synthesis of uncommon dsls," in *Modellierung 2024 Satellite Events*. Gesellschaft für Informatik eV, 2024, pp. 10–18 420.

[382] L. Zhang, K. Jijo, S. Setty, E. Chung, F. Javid, N. Vidra, and T. Clifford, "Enhancing large language model performance to answer questions and extract information more accurately," 2024.

[383] Y.-C. Lin, A. Kumar, N. Chang, W. Zhang, M. Zakir, R. Apte, H. He, C. Wang, and J.-S. R. Jang, "Novel preprocessing technique for data embedding in engineering code generation using large language model," in *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 2024, pp. 1–5.

[384] P. Derakhshanfar, X. Devroey, and A. Zaidman, "Basic block coverage for search-based unit testing and crash reproduction," *Empirical Software Engineering*, vol. 27, no. 7, p. 192, 2022.

[385] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang *et al.*, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1607–1619.

[386] Z. Zhang, X. Liu, Y. Lin, X. Gao, H. Sun, and Y. Yuan, "Llm-based unit test generation via property retrieval," *arXiv preprint arXiv:2410.13542*, 2024.

[387] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva, "Contract driven development= test driven development-writing test cases," in *ESEC/FSE*, 2007, pp. 425–434.

[388] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[389] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1174–1186.

[390] Y. Zou, H. Sun, C. Fang, J. Liu, and Z. Zhang, "Deep learning framework testing via hierarchical and heuristic model generation," *Journal of Systems and Software*, vol. 201, p. 111681, 2023.

[391] A. Narayanan, N. S. harzevili, J. Wang, L. Shi, M. Wei, and S. Wang, "Automatic unit test generation for deep learning frameworks based on api knowledge," 2023.

[392] M. G. Arivazhagan, L. Liu, P. Qi, X. Chen, W. Y. Wang, and Z. Huang, "Hybrid hierarchical retrieval for open-domain question answering," in *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, pp. 10 680–10 689.

[393] X. Wang *et al.*, "Searching for best practices in retrieval-augmented generation," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024, pp. 17 716–17 736. [Online]. Available: https://aclanthology.org/2024.emnlp-main.981/

[394] Z. Z. Wang, A. Asai, F. F. Xu, Y. Xie, G. Neubig, D. Fried *et al.*, "Coderag-bench: Can retrieval augment code generation?" in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 3199–3214.

[395] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3982–3992.

[396] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, 2021, pp. 2719–2734.

[397] K. Sawarkar, A. Mangal, and S. R. Solanki, "Blended rag: Improving rag (retriever-augmented generation) accuracy with semantic search and hybrid query-based retrievers," in *2024 IEEE 7th International Conference on Multimedia Information Processing and Retrieval (MIPR)*. IEEE, 2024, pp. 155–161.

[398] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine learning research*, vol. 7, pp. 1–30, 2006.

[399] P. Hosseini, I. Castro, I. Ghinassi, and M. Purver, "Efficient solutions for an intriguing failure of llms: Long context window does not mean llms can analyze long sequences flawlessly," in *Proceedings of the 31st International Conference on Computational Linguistics*, 2025, pp. 1880–1891.

[400] M. G. Kendall and B. B. Smith, "The problem of m rankings," *The annals of mathematical statistics*, vol. 10, no. 3, pp. 275–287, 1939.

[401] B. Shen, M. A. Gulzar, F. He, and N. Meng, "A characterization study of merge conflicts in java projects," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–28, 2023.

[402] Z. Li, A. R. Chen, X. Hu, X. Xia, T.-H. Chen, and W. Shang, "Are they all good? studying practitioners' expectations on the readability of log messages," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 129–140.

[403] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Automatic static vulnerability detection for machine learning libraries: Are we there yet?" in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 795–806.

[404] N. S. Harzevili, M. M. Mohajer, J. Shin, M. Wei, G. Uddin, J. Yang, J. Wang, S. Wang, Z. Ming, N. Nagappan *et al.*, "Checker bug detection and repair in deep learning libraries," *arXiv preprint arXiv:2410.06440*, 2024.

[405] H. Taherkhani and H. Hemmati, "Valtest: Automated validation of language model generated test cases," *arXiv preprint arXiv:2411.08254*, 2024.

[406] G. Guest, K. M. MacQueen, and E. E. Namey, *Applied thematic analysis*. sage publications, 2011.

[407] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.

[408] S. Hong and M. Kim, "Effective pattern-driven concurrency bug detection for operating systems," *Journal of Systems and Software*, vol. 86, no. 2, pp. 377–388, 2013.

[409] M. Felderer and I. Schieferdecker, "A taxonomy of risk-based testing," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 5, pp. 559–568, 2014.

[410] C. Wang, F. Pastore, A. Goknil, and L. C. Briand, "Automatic generation of acceptance test cases from use case specifications: an nlp-based approach," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 585–616, 2020.

[411] Y. Zhang, R. Zhu, Y. Xiong, and T. Xie, "Efficient synthesis of method call sequences for test generation and bounded verification," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[412] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[413] X. Guo, H. Okamura, and T. Dohi, "Optimal test case generation for boundary value analysis," *Software Quality Journal*, vol. 32, no. 2, pp. 543–566, 2024.

[414] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT software engineering notes*, vol. 30, no. 5, pp. 263–272, 2005.

[415] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 595–605.

[416] G. B. De Padua and W. Shang, "Studying the prevalence of exception handling anti-patterns," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 328–331.

[417] G.-D. Schwarz, A. Bauer, D. Riehle, and N. Harutyunyan, "A taxonomy of microservice integration techniques," *Information and Software Technology*, vol. 183, p. 107723, 2025.

[418] M. Saadatmand, "Integrationdistiller: Automating integration analysis and testing of object-oriented applications," in *2019 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2019, pp. 1385–1392.

[419] V. Costa, G. Girardon, M. Bernardino, R. Machado, G. Legramante, A. Neto, F. P. Basso, and E. de Macedo Rodrigues, "Taxonomy of performance testing tools: A systematic literature review," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 1997–2004.

[420] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, 2011, pp. 19–30.

[421] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar, "On mitigating code llm hallucinations with api documentation," *arXiv preprint arXiv:2407.09726*, 2024.

[422] J. Chen, S. Chen, J. Cao, J. Shen, and S.-C. Cheung, "When llms meet api documentation: Can retrieval augmentation aid code generation just as it helps developers?" *arXiv preprint arXiv:2503.15231*, 2025.

[423] D. G. Paul, H. Zhu, and I. Bayley, "Benchmarks and metrics for evaluations of code generation: A critical review," in *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2024, pp. 87–94.

[424] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. ZHANG, "Evaluating language models for efficient code generation," in *First Conference on Language Modeling*, 2024.

[425] S. Kaniewski, D. Holstein, F. Schmidt, and T. Heer, "Vulnerability handling of ai-generated code-existing solutions and open challenges," in *2024 Conference on AI, Science, Engineering, and Technology (AIxSET)*. IEEE, 2024, pp. 145–148.

[426] Z. Wang, R. Ling, C. Wang, Y. Yu, S. Wang, Z. Li, F. Xiong, and W. Zhang, "Maintaincoder: Maintainable code generation under dynamic requirements," *arXiv preprint arXiv:2503.24260*, 2025.

[427] M. Ferreira, L. Viegas, J. P. Faria, and B. Lima, "Acceptance test generation with large language models: An industrial case study," *arXiv preprint arXiv:2504.07244*, 2025.

[428] J. Liu, S. Lee, E. Losiouk, and M. Böhme, "Can llm generate regression tests for software commits?" *arXiv preprint arXiv:2501.11086*, 2025.

[429] Y. Wang, C. Xia, W. Zhao, J. Du, C. Miao, Z. Deng, P. S. Yu, and C. Xing, "Projecttest: A project-level llm unit test generation benchmark and impact of error fixing mechanisms," *arXiv preprint arXiv:2502.06556*, 2025.

[430] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[431] A. Silva and M. Monperrus, "Repairbench: Leaderboard of frontier models for program repair," in *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 9–16.

[432] Y. Wang, M. Pradel, and Z. Liu, "Are" solved issues" in swe-bench really solved correctly? an empirical study," *arXiv preprint arXiv:2503.15223*, 2025.

[433] T. Ahmed, J. Ganhotra, R. Pan, A. Shinnar, S. Sinha, and M. Hirzel, "Otter: Generating tests from issues to validate swe patches," in *Forty-second International Conference on Machine Learning*, 2025.