Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
2025, vol. 30, no. 1, pp. 105–121
https://doi.org/10.2478/acss-2025-0013
https://content.sciendo.com

1862
RIGA TECHNICAL UNIVERSITY

Analysing Software Quality of AI-Translated Code: A Comparative Study of Large Language Models Using Static Analysis

Vikram Bhutani[1*], Farshad Ghassemi Toosi[2], Jim Buckley[3]
[1,2]*Computer Science Department, Munster Technological University, Cork, Ireland*
[3]*Computer Science Department, University of Limerick, Limerick, Ireland*

***Abstract* – Context: Source code translation enables cross-platform compatibility, code reusability, legacy system migration, and developer collaboration. Numerous state-of-the-art techniques have emerged to address demand for efficient and accurate translation methodologies.**

Objective: This study compares code translation capabilities of Large Language Models (LLMs), specifically DeepSeek R1 and ChatGPT 4.1, evaluating their proficiency in translating code between programming languages. We systematically assess model outputs through quantitative and qualitative measures, focusing on translation accuracy, execution efficiency, and coding standard conformity. By examining each model's strengths and limitations, this work provides insights into their applicability for various translation scenarios and contributes to discourse on LLM potential in software engineering.

Method: We evaluated translation quality from ChatGPT 4.1 and DeepSeek R1 using SonarQube Analyzer to identify strengths and weaknesses through comprehensive software metrics including translation accuracy, code quality, and clean code attributes. SonarQube's framework enables objective quantification of maintainability, reliability, technical debt, and code smells which are critical factors in software quality measurement. The protocol involved randomly sampling 500 code instances from 1695 Java programming problems. Java samples were translated to Python by both models, then analysed quantitatively using SonarQube metrics to evaluate adherence to software engineering best practices.

Results: This comparative analysis reveals capabilities and limitations of state-of-the-art LLM-based translation systems, providing developers, researchers, and practitioners actionable guidance for model selection. Identified gaps highlight future research directions in automated code translation. Results demonstrate that DeepSeek R1 consistently generates superior software quality compared to ChatGPT 4.1 across Sonar-Qube metrics.

***Keywords* – Artificial intelligence, ChatGPT4.1, C++, deep learning, DeepSeek R1, Java, machine learning, NMT, object oriented, programming languages translation, Python, source code translation, SMT.**

I. Introduction

The necessity to migrate legacy software systems to modern platforms has made source code translation a pivotal research topic in software engineering. Manual translation is resource-intensive and error-prone, highlighting the imperative for automated tools that support seamless, reliable code conversion between programming languages. Despite substantial technological advancements, persistent challenges remain in maintaining semantic equivalence and ensuring readability and structural integrity in translated code. Thus, rigorous evaluation of AI-generated translations is required, including the use of software metrics such as McCabe's cyclomatic complexity [1] to assess testability and maintainability.

Given the inherent complexity of software development, robust safeguards and comprehensive review protocols are essential, particularly for AI-driven code translation. Best practices within a high-quality development framework include testing, code review, benchmarking, code style enforcement, and verification for functional equivalence, performance, and efficiency. Systematic integration of these practices ensures that AI-assisted translation upholds both functionality and quality standards central to contemporary software engineering [2]–[4].

Source-to-source code translation (transpilation) has evolved markedly since the 1950s [5], [6]. Early rule-based systems had limited scalability and adaptability [7]. Modern advances in statistical and neural network approaches have improved translation fidelity, though safeguarding semantic equivalence and data privacy remains challenging. Metrics such as BLEU are often used, but they insufficiently capture nuanced semantic correctness. Manual codebase migration remains lengthy, fuelling demand for streamlined, automated solutions.

Advances in artificial intelligence, particularly in NLP and LLM code generation, have produced AI-powered tools capable of translating requirements into executable code across numerous languages [8], unlocking new possibilities for automating the software development lifecycle [9]. Among these, AI-driven code translation stands out for enabling more efficient legacy system migration, cross-platform compatibility,

* Corresponding author's e-mail: vikram.bhutani@intel.com
Article received 2025-07-19; accepted 2025-10-03

and improved interoperability [10]. Accurate, efficient translation between languages such as Java and Python has become increasingly vital due to Java's prominence in enterprise and Android development and Python's dominance in machine learning, data science, and automation [11]. AI tools are also redefining programming education [12], providing personalized feedback and learning experiences [13]. While AI adoption expands, ethical and practical concerns including the appropriateness of AI use in education and continued challenges in higher-level algorithmic reasoning persist [14], [15].

Rigorous quality evaluation of LLM driven translation is therefore crucial, ensuring reliability and revealing limitations that guide responsible integration into engineering practice. Deep learning, especially transformer architectures [16], has become central to modern source code translation. AI-based automation increases efficiency and scalability when handling large codebases [17], but output quality its syntactic correctness, semantic fidelity, runtime performance, and maintainability remains decisive. Systematic evaluation must verify compliance with standards and support continuous human maintainability.

Source-to-source code translation commonly referred to as transpilation has witnessed profound evolution since its inception in the 1950s [5], [6]. Initial efforts predominantly employed rule-based systems, which, despite their utility, were often hampered by limited scalability and adaptability [7]. In recent years, the emergence of statistical and neural network-based methodologies has dramatically enhanced translation fidelity. Nevertheless, persistent challenges remain, particularly in safeguarding semantic equivalence and managing proprietary data. While metrics such as BLEU (Bilingual Evaluation Understudy) and ROGUE (Recall-Oriented Understudy) scores are routinely utilised to quantify translation quality in NLP tasks, they frequently fall short in capturing nuanced semantic correctness [18]. Additionally, the manual migration of extensive codebases can span several years, underscoring the demand for more streamlined and automated solutions.

The evolution of machine translation from rule-based/statistical origins to deep neural transformers reflects broader advances in cognitive science, AI, and computational thinking [19], [20]. Proliferation of open-source code, cloud platforms, and SaaS has fuelled the rise of AI-driven developer tools, including chatbots [21], which now demand strict quality assurance well beyond functional correctness [22], [23].

Key software quality dimensions include readability for maintainability, modularity, and debugging; maintainability itself (modularity, complexity, documentation) [24]; efficiency (execution/resource use); and security. Adherence to conventions, formatting, and architecture aids collaboration and reduces risk.

State-of-the-art code generation automation techniques blend NLP (Natural Language Processing), LLMs (Large Language Models) and program analysis. LLMs like ChatGPT [25], DeepSeek, Grok, Claude, and Gemini are already leading modern code generation demand in software industry. Code-based LLMs are achieving, and even surpassing, human performance in translation [26], as they excel in syntactic and semantic tasks and become fundamental to automated language processing [27], [28]. Still, challenges remain, especially in generating nuanced, robust explanations or performing comprehensive correction at industrial scale [3], [29], [30]. A major benefit of LLMs is semantic code comprehension, surfacing logic errors often missed by static analysis methods like AST, flow, or taint analysis [3], [29], [31]. LLMs now aid code generation [32], [33], debugging [34], and code explanation [35], [36]. Their multilingual reach facilitates broad adoption and integration.

LLM translation performance varies widely reported accuracy ranges from 2.1 % to 47.3 % [37] and some models even outperform transpilers though improvement is ongoing [38]. Since standard LLM prompts lack deep project-specific context, new methods are needed for broader contextual access.

This work empirically compares two top LLMs, ChatGPT and DeepSeek in translation accuracy, maintainability, efficiency, language coverage, and flexibility across diverse, real-world code. We also introduce federated LLM approaches for secure, distributed code translation evaluation. Our findings offer holistic insights into modern LLM translation, establish rigorous benchmarks, highlight remaining challenges, and guide practitioners in selecting the right tools for specific needs.

***Research Question**:* Does a significant difference exist in the software quality of code translated by different state-of-the-art LLMs, specifically ChatGPT and DeepSeek?

II. Background

Several research endeavours have focused on source code translation and comprehension, utilising a wide array of techniques and methodologies. Notable contributions in this domain are discussed below.

Alon et al. introduced Code2Vec, an innovative approach for learning distributed representations of code through neural networks [39]. This method harnesses code snippets along with their contextual surroundings to capture rich semantic information, thereby advancing both code understanding and translation. In the domain of paraphrase detection, Allamanis et al. explored convolutional, recurrent, and hybrid models to identify semantically analogous code segments [40]. Their research underscores the importance of recognising code similarities, a factor that is highly pertinent to source code translation tasks.

Automated program repair has garnered significant scholarly interest, with Jiang and Su providing a comprehensive literature review of the field [41]. While the primary focus of program repair is bug resolution, many of the transformation techniques surveyed are relevant to code translation, especially when structural modifications are necessary. Yang et al. proposed Deep Code Search, a deep learning-based approach designed to retrieve code snippets in response to natural language queries [32]. This work demonstrates the efficacy of leveraging natural language processing algorithms to enhance both code understanding and translation.

Raychev et al. pioneered the application of statistical language models for code completion, employing machine learning techniques to predict and suggest contextually appropriate code fragments [42]. Although their primary focus was on completion, their methodology facilitated code translation by offering accurate, context-aware suggestions throughout the translation process. Robbes et al. conducted user evaluations of automatic software repair methods [43], providing important insights into the usability and efficacy of automated code transformation techniques, which are also salient for source code translation initiatives. Sachdeva et al. presented a robust program synthesis framework through efficient code transplantation [44], illuminating the promise of transplantation as a powerful means for automating code transformation and translation workflows.

Collectively, these studies present foundational insights and rigorous methodologies that underpin advances in source code translation, augmenting both code analysis and transformation capabilities.

Among legacy approaches, tools such as Rosetta have employed neural machine translation strategies to convert code snippets between programming languages [45]. By comparison, transpilers are engineered to translate source code from one language to another while meticulously preserving structural and semantic fidelity [5]. More recent trends in AI-driven software engineering have yielded tools for tasks such as code summarisation [46], code comment analysis [31], [36], and code translation [39], [47], [48]. LLMs tailored for code are progressing rapidly, with code editing now emerging as a pivotal capability. The application of LLMs within Generative AI has gained significant traction in software engineering, notably in autocode generation [49], [50]. A proliferation of new code-centric LLMs has also marked the field, including models such as Mixtral [51], DeepSeek-Coder-33b-base [52], OpenCodeInterpreter-DS-6.7B [53], and Phind-Code-Llama [54], among others.

The use of LLMs for programming tasks represents a significant advancement in computer science, enabling capabilities that were once considered unattainable [55]. This progress is driven by the growing complexity of software development and the increasing demand for tools that assist programmers in automating repetitive or technically challenging tasks. One such task is code translation, which involves converting source code from one programming language to another, a process that often demands a deep understanding of the syntax, semantics and idiomatic practices of both languages. Consequently, LLMs have emerged as promising tools for automating code translation, offering the potential to save time and reduce human error.

A. Overview of LLMs Used in Source Code Translation

The application of LLMs to programming tasks marks a profound advancement in computer science, enabling functionalities that were once unattainable [55]. This evolution is propelled by the escalating complexity of software development and the growing demand for intelligent tools that support programmers in automating both repetitive and technically demanding functions. Among these, code translation converting source code from one programming language to another stands as a particularly challenging task, necessitating a nuanced understanding of syntax, semantics, and idiomatic conventions unique to each language. LLMs have consequently emerged as highly promising instruments for automating code translation, holding the potential to significantly reduce time investment and mitigate human error.

General-purpose LLMs, such as ChatGPT, have demonstrated considerable efficacy in code generation, paving the way for the emergence of specialised models tailored to software engineering applications, commonly referred to as Code LLMs [56]. These models are pre-trained on vast corpora of source code and can be fine-tuned for specialised tasks, including code completion, bug detection and repair, and cross-language code translation [57].

With the rapid progress in Generative AI, a new generation of LLMs is achieving or even surpassing human-level accuracy across many code-related tasks, underscoring the necessity of rigorous comparative analyses of code translation performance across leading models such as DeepSeek and ChatGPT. It is imperative to systematically assess the strengths and limitations of each model, providing detailed insights into their effectiveness and applicability for code translation tasks. Benchmark datasets such as CodeXGLUE [58], CodeSearchNet [59], and XLCoST [60] serve as standard evaluation frameworks during inference and testing. These resources underpin the development and assessment of code-centric natural language processing tasks, including translation.

LLMs have consistently exhibited outstanding performance on a broad array of natural language processing challenges, such as text generation, machine translation, and question answering. Most of these models are architected using the Transformer framework, enabling the capture of long-range dependencies and complex contextual relationships within text data [61]. The landscape of language modelling has witnessed a transformative shift, largely attributed to the advent of pre-trained Transformer architectures [61].

DeepSeek [62] demonstrates remarkable prowess in code generation, particularly excelling at producing efficient and optimized code. It is adept at tackling complex programming tasks and consistently delivers outputs with high accuracy.

Grok [63] is an emerging, versatile LLM exhibiting robust capabilities in both natural language processing and code generation. While it may not possess the deep technical specialisation of some peer models, Grok offers a well-balanced performance across both language understanding and programming competence.

Claude [64] distinguishes itself as a reliable and consistent LLM, consistently yielding readable and maintainable code attributes highly valued in real-world software development projects. Named in honour of Claude Shannon, a luminary in information theory, Claude is engineered with a strong emphasis on safety, interpretability, and alignment with user intent. The model is underpinned by a transformer-based architecture and has been trained on an expansive and diverse dataset encompassing books, technical literature, programming

content, and publicly available web data. This extensive training corpus empowers Claude to perform a multitude of language tasks, including code generation, summarization, and translation.

A defining characteristic of Claude lies in its prioritization of AI safety and ethical value alignment. Employing strategies such as Reinforcement Learning from Human Feedback (RLHF), the model is designed to enhance output quality, reduce the risk of biased or harmful responses, and ensure close alignment with user expectations. These attributes render Claude especially suitable for contexts where adherence to ethical standards and safety is of paramount concern.

Gemini [65], developed by Google DeepMind, is a state-of-the-art multimodal large language model that represents a significant leap in artificial intelligence capabilities. Built upon an advanced transformer-based architecture, Gemini is engineered for both performance and scalability. A distinctive aspect of Gemini is its multimodal proficiency, enabling it to interpret and generate not only natural language text, but also visual and structured data inputs.

Gemini is widely recognised for its advanced language understanding, which translates directly to its effectiveness in code generation, debugging, and code explanation. Its ability to follow complex instructions and produce functionally accurate code makes it suitable for a wide range of development tasks. Gemini can generate, explain, and debug code in more than 20 programming languages, including Python, JavaScript, Prolog, Fortran, C++, Go, Java, and many others. With features like a large context window up to 2 million tokens and the ability to process up to 10 000 lines of code at once Gemini facilitates extensive code analysis and project-scale transformations.

Within development environments, Gemini Code Assist integrates Gemini's capabilities into popular IDEs (such as Visual Studio Code, JetBrains IDEs, and Android Studio), offering context-aware code completion, instant code review, automatic bug identification, and suggestions for code changes and optimisations. The system also supports multi-file edits, project-scale context awareness, human-in-the-loop supervision for critical tasks, and integration with ecosystem tools via the Model Context Protocol (MCP). Through its API, Gemini can iteratively generate and execute Python code, learn from execution results, and refine outputs a feature essential for code-based reasoning and complex solution synthesis.

In advanced use cases, Gemini underpins AlphaCode 2, which competes at a world-class level in programming competitions by solving advanced algorithmic tasks, demonstrating coding abilities that surpass most human participants.

Other application-specific LLMs, including LLaMA-3, Phi-3, Mistral, and Gemma, have also made notable advances in code summarization, offering sophisticated methods for generating concise, functionally accurate descriptions of source code [66]. These models contribute to the rapidly evolving landscape of LLMs for code, supporting broad applications from generation and translation to documentation and intelligent explanation.

Table I shows the detailed comparison of current widely used LLMs and supported programming languages along with trainable parameters.

The following LLMs have been extensively applied to code-related tasks such as code summarization, bug detection, code comprehension, and translation across different programming paradigms.

B. ChatGPT 4.1

ChatGPT-4.1, released by OpenAI in April 2025, is a state-of-the-art general-purpose LLM built upon the GPT-4 architecture. It is notable for its advanced reasoning abilities, improved instruction-following, and robust multimodal comprehension including both text and image inputs. GPT-4.1 represents the latest evolution in OpenAI's language model series.

TABLE I
COMPARISON OF LLMS USED FOR SOURCE CODE TRANSLATION

Model	Developer	Parameters	Supported Languages	Key Features
ChatGPT 4.1	OpenAI	1.76T	Python, Java, C++, JavaScript, TypeScript, SQL, HTML, etc.	General-purpose transformer model with strong reasoning and translation capabilities, trained with RLHF
DeepSeek R1	DeepSeek AI	Undisclosed (estimated 50B–100B)	Python, Java, C++, TypeScript, JavaScript, etc.	Lightweight reasoning model with strong performance in code generation and mathematical reasoning
Claude	Anthropic	Undisclosed (estimated 100B–200B)	Python, Java, C++, Go, JavaScript, etc.	Focuses on alignment, safety, and ethical reasoning with conversational and problem-solving proficiency
Gemini	Google DeepMind	Undisclosed (estimated >500B)	Python, Java, C++, JavaScript, HTML/CSS, SQL	Multimodal reasoning model capable of understanding text, images, charts; strong factual grounding
TransCoder	Meta AI	1B (estimate)	Python, Java, C++	Pretrained specifically for code translation tasks using supervised learning on parallel code datasets

ChatGPT-4.1 has demonstrated remarkable proficiency in Python code generation, delivering robust solutions for exploratory data analysis, data preprocessing, and categorical data analysis [67]. Its code generation capabilities establish it as a highly asset for data-driven development tasks [68]. More recently, its application in code translation has emerged as a promising research direction, with the model showing notable potential to automate code conversion between diverse programming languages [69].

Modern code-oriented LLMs, exemplified by ChatGPT, leverage billions of parameters and are fundamentally based on transformer-driven neural network architectures [48], [70]. These models are pre-trained on large-scale, application-specific corpora containing both natural language and source code [71]. The pre-training phase is typically supplemented by supervised fine-tuning and reinforcement learning from human feedback (RLHF). In the ChatGPT architecture, the predominant pre-training objective is language modelling, wherein the model learns to predict the next token in a sequence conditioned on the preceding context [72].

The key architectural and performance highlights include: ChatGPT-4.1 is constructed with 1.76 trillion parameters and a 120-layer transformer architecture and was trained on more than 25 000 GPUs over a period of nearly 100 days. It achieves outstanding performance in natural language understanding and code generation. Notably, ChatGPT-4.1 demonstrates significant improvements in multimodal benchmarks (e.g., MMMU, MathVista) and exhibits enhanced alignment with user intent through RLHF. Within the context of this study, GPT-4.1 is employed to translate Java source code into Python, utilising carefully engineered prompts to guide its behaviour, ensuring consistency and translation fidelity.

ChatGPT's widespread adoption, comprehensive support for a multitude of programming languages, and continual model refinement position it as an ideal baseline for comparison with specialised tools such as TransCoder and more recent entrants like DeepSeek.

C. DeepSeek R1

DeepSeek [62] is a state-of-the-art suite of LLMs that has rapidly evolved since its inception. Established in 2018, DeepSeek initially focused on accessible and efficient AI tools and has since developed into a leader in open-weight, high-parameter LLMs. Notable milestones include the release of DeepSeek Coder (2023), DeepSeek-V3 (2024), and DeepSeek-R1 (2025), each introducing major advancements in reasoning, efficiency, and cost-effectiveness [32], [52]. The flagship DeepSeek-V3 and R1 models comprise an immense 671 billion parameters, employing a Mixture-of-Experts (MoE) architecture with 37 billion parameters activated per token. DeepSeek-Coder-V2, specifically optimised for code-related tasks, features 236 billion parameters and supports a 128 000-token context window. Training DeepSeek-V3 required approximately 2000 NVIDIA H800 GPUs over 55 days, with each GPU equipped with at least 80GB of VRAM. The infrastructure leverages distributed computing, high-speed networking (InfiniBand), and frameworks such as PyTorch and DeepSpeed to optimise parallelism and efficiency [52]. The total training expenditure for DeepSeek Coder was estimated at $6 million, reflecting significant optimisation compared to competitors like ChatGPT and Gemini. DeepSeek utilises both data and model parallelism, incorporating advanced communication strategies and memory optimisations to efficiently scale across thousands of GPUs while minimising bottlenecks [52].

DeepSeek excels at code generation, debugging, refactoring, and optimisation across a variety of programming languages, including Python, JavaScript, C++, Java, and more. Like other leading LLMs, DeepSeek is trained to generate entire code blocks from natural language prompts, debug and correct code errors, and provide explanations and optimised solutions. It also possesses the capability to refactor disorganised or inefficient code, enhancing readability and performance. Currently, it supports project-level code completion and infilling with large context windows (up to 128 000 tokens), offers multi-language support, and can automatically generate documentation. With state-of-the-art performance on code benchmarks and open-source availability, DeepSeek has emerged as a preferred tool among developers, educators, and researchers seeking powerful, efficient AI-driven coding assistance. DeepSeek LLMs are meticulously designed to address complex AI challenges using advanced deep learning techniques. These models vary in architecture, training data, and performance metrics, underscoring the necessity of evaluating their suitability for specific code translation tasks [73].

D. CodeGen

TransCoder, developed by Facebook AI Research (FAIR), is a specialised neural trans-compiler engineered for translating source code between programming languages. Built upon the transformer architecture, it is trained using unsupervised learning techniques on expansive monolingual code corpora. The tool supports translation across several programming languages, including Java, Python, and C++, making it well-suited for code migration and interoperability tasks.

CodeGen, previously known as TransCoder, is a dedicated tool developed by Meta™ for converting code among languages such as C++, Java, and Python. Trained on a substantial corpus of code examples, it utilises a multilingual approach to accommodate both source and target languages. While CodeXGLUE encompasses a broader range of code-related NLP tasks, TransCoder is specialised for code translation and is intended as a practical resource for developers and researchers operating across programming environments.

Training Architecture: CodeGen utilises a Transformer-based architecture with 6 layers, 8 attention heads, and a model dimensionality of 1024. A single encoder-decoder pair handles all supported programming languages. After pre-training, the model undergoes alternating training phases using Denoising Auto-Encoding and Back-Translation objectives. The Adam optimizer is employed for model optimisation.

Dataset Used in TransCoder: For training, the authors leverage the publicly available GitHub dataset from Google BigQuery, filtering for repositories that allow redistribution,

and selecting Java, Python, and C++ projects. Training and evaluation occur at the function level, as modest-sized code snippets are manageable within a single batch and facilitate clear assessment. For validation and testing, multi-language solutions to problems from the GeeksforGeeks platform are utilised. TransCoder integrates static analysis with machine learning techniques to facilitate cross-lingual code retrieval and translation [48].

E. Transpiler

A transpiler tool [5] developed by researchers at the University of California, San Diego, is capable of translating source code between diverse programming languages, leveraging a combination of machine learning and advanced program analysis techniques. Transpiler is a sophisticated AI-driven system specifically designed to automate code translation [74], [75], code generation, and optimisation across multiple programming languages. Its development aligns with the rising demand for technologies that bridge language barriers in software engineering, thus serving as an invaluable asset for developers engaged in modernising legacy codebases or migrating projects across technology stacks.

Transpiler models typically span from hundreds of millions to several billion parameters, depending on the specific version and target application. State-of-the-art variations employ transformer architectures analogous to those powering large language models, enabling nuanced code understanding and generation [5], [76]. Training durations are contingent on model scale and dataset complexity, with large models often trained over several weeks on distributed clusters of NVIDIA A100 or H100 GPUs with some iterations utilising between 512 and 2048 GPUs. Each GPU is generally furnished with 40–80 GB of VRAM to efficiently manage the expansive context windows required for comprehensive code translation tasks. The cost of training a high-performance Transpiler model can escalate into several million dollars, accounting for computational infrastructure, storage, and engineering efforts. To optimise both efficiency and expenditure, techniques such as mixed-precision training and parallelism are implemented, alongside high-speed interconnects to facilitate distributed training across substantial GPU clusters.

Designed specifically for code-centric applications, Transpiler excels at automating complex software engineering workflows [5], [6], [48] demanding deep semantic comprehension and accurate language mapping. Among its principal capabilities are:

- translating code between major programming languages (e.g., Python to Java, C++ to Rust) while faithfully preserving logical structure;
- generating idiomatic code in the target language that adheres to language-specific conventions and library usage;
- refactoring and modernising legacy codebases to enhance readability and replace obsolete patterns;
- providing contextual code completion, infilling, and automated documentation generation for multilingual, large-scale software projects;
- supporting project-wide migration through large context windows, thereby ensuring cross-file and cross-module consistency [77].

Transpiler's robust performance on code translation and migration benchmarks [76], [78], coupled with its adaptability to emerging languages and frameworks, establishes it as a preferred solution for organisations engaged in large-scale code modernisation and cross-platform software development initiatives.

III. Methodology

The principal aim of this study is to achieve a holistic understanding of the quality of source code produced by two state-of-the-art (SOTA) LLMs, namely ChatGPT and DeepSeek, coupled with a comparison of software quality metrics of translated source code from each LLM. This investigation is designed to elucidate the intrinsic strengths and potential drawbacks of these advanced techniques, thereby equipping practitioners with actionable insights for selecting the most appropriate solution tailored to their unique project demands. Figure 1 illustrates the methodological framework adopted to assess the efficacy of various LLMs within the domain of source code analysis.

A. Prompts Used for ChatGPT and DeepSeek LLMs

Prompt engineering is a critical determinant in fully leveraging the capabilities of LLMs for code generation and translation. In this study, prompts are crafted with meticulous attention to detail and systematically optimised to elicit high-quality outputs from each model.

Each prompt delivers clear and succinct instructions, delineating the target programming language, coding task, and any specific constraints or requirements. A variety of prompting strategies are assessed, including detailed step-by-step directives, examples reflecting preferred coding conventions, and precise definitions of input/output formats.

For ChatGPT, prompt templates are strategically utilised to enhance code translation accuracy, facilitate code refactoring, and support requirements elicitation [79]. These prompt formulations undergo iterative tuning generated results are actively evaluated, and prompts are refined in accordance with performance metrics.

This cyclical process systematically adjusts prompt parameters, with continual measurement of their impact on code quality indicators such as syntactic correctness, execution efficiency, and readability. The prompt designs are adaptable across different programming languages and coding scenarios, thus ensuring an equitable and consistent assessment of the LLMs under varied experimental conditions. The overarching objective is to craft prompts that minimise ambiguity and optimise the models' ability to produce accurate, idiomatic code translations.

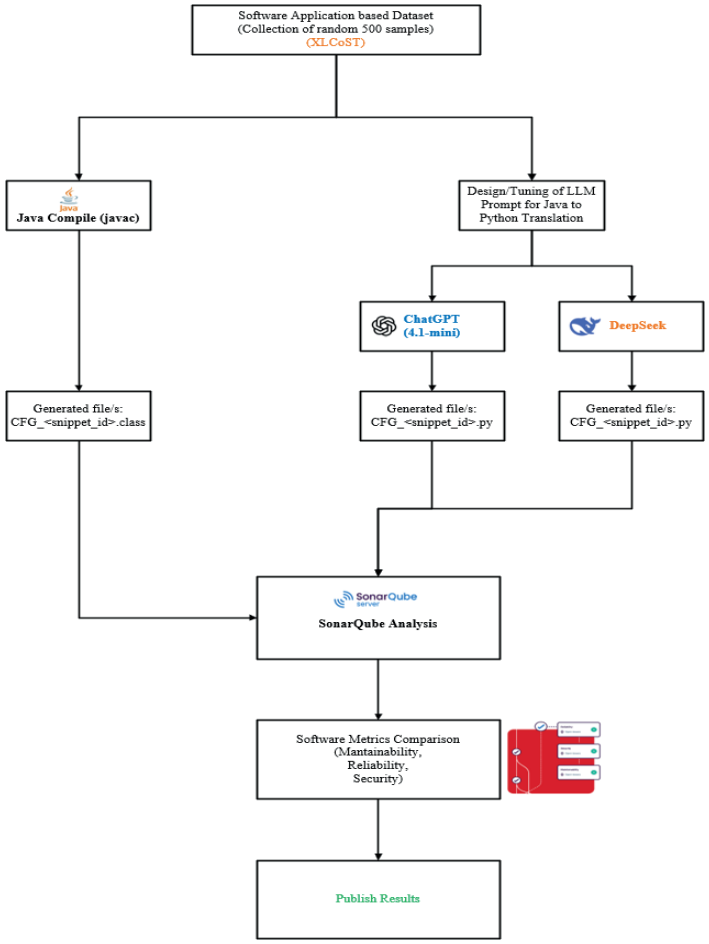


Fig. 1. Methodology for source code analysis using LLMs.

Prompt patterns act as structured directives that guide LLMs in applying rules, adhering to workflows, and achieving predetermined quality attributes in generated code [79]. However, empirical studies have revealed that common prompting approaches, including simple, persona-based, and conversational prompting, do not consistently surpass the default performance of models such as ChatGPT-3.5 [80].

These results highlight the necessity of refining prompt engineering practices, including the integration of elements from multiple prompting techniques, to further enhance LLM performance [81].

Prompts Used in ChatGPT and DeepSeek Source Code Translation: The same prompt was employed for both LLMs following multiple iterations and validations to ensure output fidelity. These prompts were then further optimised to eliminate unnecessary additions during the translation process.

The final prompt applied is as follows:

" Translate the following Java code to equivalent Python code without adding complexity. Keep the same logic and structure. Only provide the Python code without any additional explanation. Please do not add any code block markers in the translated code."

B. SonarQube Analysis

To rigorously assess the quality and security of the translated code, SonarQube, a widely adopted static analysis platform, is integrated into the evaluation workflow [82]. SonarQube facilitates the detection of code smells, potential bugs, and security vulnerabilities within the generated outputs [83].

Setup and Configuration: A dedicated SonarQube server will be established, equipped with language-specific plug-ins for both Python and Java to enable a robust static code analysis. The configuration will include tailored measurement of software quality metrics across three principal categories: maintainability, reliability, and security [82]. Quality gates will be defined to enforce coding conventions and ensure alignment with industry-standard security practices.

Code Analysis: Each code artefact produced by the LLMs will be subjected to a comprehensive static analysis via SonarQube [84]. The analysis will encompass an extensive evaluation of multiple quality dimensions, such as cyclomatic complexity, maintainability, and potential security vulnerabilities [85].

Metric Collection: Key software metrics, including the count of code smells, identified bugs, and detected vulnerabilities, will be systematically recorded for every translated code instance. These quantitative indicators will provide objective measures of the code quality produced by each LLM.

Comparative Assessment: The results from SonarQube analyses will be comparatively assessed across the evaluated LLMs to identify which models generate code that is more

secure and maintainable. Statistical methods will be employed to determine the significance of observed disparities in the collected metric values.

Integration of Low-Level Software Metrics: The principal SonarQube metrics pertaining to reliability, maintainability, and overall code quality will be integrated with additional low-level sub-metrics such as execution success rate, code readability, and code comprehensibility. This holistic approach will yield a comprehensive and multidimensional evaluation of each LLM's code translation performance.

By incorporating SonarQube analysis within the methodological framework, this study delivers a robust and nuanced examination of both the functional correctness and the key quality characteristics of code produced by large language models.

IV. Java-Python Dataset Selection

For evaluation, we selected two of the most prevalent programming languages: Java and Python. Both are prominent representatives of the object-oriented paradigm, consistently ranking among the most widely utilised languages as reported by Statistica [86]. This choice not only reflects industry relevance but also allows for a more controlled comparison of software metrics, given their shared paradigm yet distinct syntactic and semantic features. The selected languages encapsulate diverse yet widely adopted software ecosystems, supporting a robust and generalizable evaluation.

For each language, we curated representative datasets comprised of high-quality code samples. These samples served as input for both ChatGPT and DeepSeek LLMs, with the translations performed into Python code. The use of high-calibre datasets is essential for training and evaluating code translation systems. HumanEval, for instance, is a widely adopted benchmark that evaluates code generation from doc-strings; it contains 164 programming problems, each accompanied by unit tests to rigorously verify functional correctness. Another important dataset, CoNaLa, offers pairs of natural language and code, which has been leveraged extensively in code generation research for fine-tuning and evaluation tasks [87].

Numerous datasets have been curated for source code machine learning tasks, such as CODEXGLUE, CodeSearchNet, Tree-to-Tree, and XLCoST. Further details and dataset comparisons are presented in Section IV. Among these, XLCoST stands out as a critical benchmark for cross-lingual code intelligence tasks, including code translation [60].

Implementation and Setup: To ensure a comprehensive and fair comparison between ChatGPT 4.1 and DeepSeek, a dedicated evaluation framework was established as described in Section V. Implementations were methodically set up, with all dependencies addressed and parameters finely tuned.

These datasets are typically sourced from real-world repositories, comprising aligned code snippets and their counterparts across various languages [88]. Recent research has prioritized the construction of large-scale multilingual datasets that cover a spectrum of programming languages and paradigms [89]. These datasets are often mined from prominent open-source hosting sites, online coding forums, and publicly available repositories, providing a rich substrate for data-driven model development [90].

Data augmentation techniques such as code mutation and back-translation are frequently employed to expand dataset size and diversity, enhancing model robustness and generalization. Moreover, rigorous data cleaning and preprocessing (including noise filtering, error correction, and standardization of code formatting) are crucial to maintain dataset quality and consistency for downstream tasks.

Other notable datasets include Concode, which encapsulates natural language problem descriptions via JavaDoc comments within Java class contexts [91], and the IRCoder dataset, which utilises publicly available code files for extensive pre-training [92]. Additionally, synthetic datasets are often constructed to bolster pre-training and fine-tuning stages [93].

While labelled data is plentiful for certain tasks (e.g., code completion), complex endeavours, including code generation, translation, summarization, and repair, depend on meticulously curated, high-quality datasets [94]. Previous research has demonstrated that careful cleaning and structural refinement of code substantially boosts the effectiveness of code generation models [95].

A summary of the key datasets used for machine learning-based source code translation is given below.

TABLE II
Source Code Datasets for Machine Learning

Dataset	Languages Supported	Size	Source code Translation
Google Code Jam [96]	20	2.4 M	Yes
Code Search Net [97]	20	908 K	Yes
XLCoST [98]	7	132 K	Yes
Project CodeNet [59]	55	13.9 M	No
Tree-to-Tree Dataset [99]	2	20 K	No
CodeXGLUE-CodeTrans Dataset [58]	2	13 K	Yes (Java to C#)
GraphCodeBert [100]	2	2.4 M	No
AVATAR [101]	2	9 K	No

After a careful evaluation of the available code translation datasets, summarised in Table II, we selected the XLCoST dataset for our empirical study. XLCoST stands out for its comprehensive scope and fine-grained snippet-level alignment, covering 1625 parallel programming problems across seven major programming languages, notably including Java and Python. Unlike method-level alignments found in datasets such as CodeXGLUE, the snippet-level annotation of XLCoST provides substantially finer granularity for direct comparison, making it particularly suitable for assessing the quality and fidelity of translated code in a multilingual context.

Introduced by Zhu et al., XLCoST is a benchmark specifically designed for cross-lingual code intelligence. The dataset features fine-grained parallel data spanning seven widely used programming languages (C++, Java, Python, C, JavaScript, PHP, and C), as well as natural language (English), supporting ten cross-lingual code-related tasks. These tasks

include code translation, code search, and code classification, among others, making XLCoST a comprehensive resource for multilingual code intelligence research.

A key distinguishing feature of XLCoST is its snippet-level alignment: for each code snippet in a given source language, the dataset contains equivalent translations in all other supported languages such as Python. Programs are segmented into aligned code snippets, each paired with a language-consistent comment that provides a uniform natural language description across all language variants. This enables precise comparisons during evaluation and facilitates supervised training of translation and code understanding models. The consistent annotation across languages also enhances the dataset's value for semantic alignment tasks, such as code summarization and comment generation.

With the largest parallel coverage in terms of both size and number of supported languages, XLCoST is a pivotal resource for developing and validating cross-lingual code intelligence methods and benchmarking state-of-the-art code translation models.

A. Sample Size Determination for Source Code Analysis

Sample size is a fundamental statistical concept, representing the number of observations or replicates required in a study to draw reliable and generalizable inferences about a population [102]. In empirical software engineering, determining an appropriate sample size is essential to ensure that research findings achieve statistical significance and can be extrapolated beyond the studied sample.

The XLCoST dataset offers approximately 132 k unique Java samples collected from open-source GitHub repositories. For this study, random selection of 500 samples from the training set of XLCoST dataset is used for evaluation. To substantiate this choice, we reference the standard statistical significance [102] calculation formula (1) for calculating the required sample size (n), based on predetermined confidence levels and margins of error.

For a significance level of $p_{value} \leq 0.05$*, z-score = 1.96, confidence interval = 95 %, and margin of error = 5 %.*

$$n = e^2 \times Z^2 \times p(1-p), \tag{1}$$

where:

Z = 1.96 (95 % confidence level);

p = expected accuracy proportion (use 0.5 for conservative estimates);

e = margin of error (±5 % typical).

In this empirical study, ChatGPT 4.1 and DeepSeek R1 are employed for Java-to-Python translation tasks, with the objective of evaluating code quality, translation fidelity, and static analysis performance using the SonarQube Static Analysis Tool (SAT).

V. Evaluation

This section provides an overview of the results and the empirical comparison between the two models. The translated code generated by ChatGPT (v4.1) and DeepSeek R1 was subjected to analysis using SonarQube.

A randomly selected batch of Java code snippets was extracted from the XLCoST dataset for this analysis. These Java snippets were formatted according to standard conventions using Google Java Format to prevent any formatting discrepancies during the subsequent translation process by the LLMs. The batch was then submitted to both ChatGPT and DeepSeek using an identical prompt. Representative outputs from each model, along with a visual depiction of a key software metric such as Lines of Code (LOC) is presented in Fig. 2.

A. Evaluation Flow for Analysis of LLM Translated Code

Figure 3 presents the evaluation pipeline utilised to assess the performance of ChatGPT and DeepSeek in the source code translation task. The outputs generated by both models are subsequently subjected to comprehensive static analysis using SonarQube.

Source Code Translation Evaluation Framework: A fully automated evaluation framework was developed, based on the methodology outlined in Section III. This framework facilitated a series of systematic experiments designed to thoroughly evaluate the selected AI models. Each experiment targeted distinct evaluation dimensions, including translation accuracy, performance profiling, language coverage, reliability, and flexibility analysis. All assessments were conducted within a controlled environment to minimise external variability and ensure the validity of results.

Evaluation Metrics: A comprehensive suite of software evaluation metrics, as provided by SonarQube SAT, was employed to measure the effectiveness and performance of code translated by ChatGPT and DeepSeek. Metrics include not only standard dimensions of reliability, maintainability, and security but also translation accuracy, execution time, language support, and structural flexibility. Language coverage quantified the breadth of programming languages supported by each model, while flexibility was gauged based on each model's aptitude for managing complex code structures and edge cases. Previous study by Weisz et al. [103] conducted an empirical study to assess the effectiveness of use of AI in programming language translation. The study explored the capabilities of AI techniques in translating code between different programming languages, evaluating factors such as accuracy, efficiency, and performance.

1) Weighted Methods per Class (WMC).
2) Depth of Inheritance Tree (DIT).
3) Number of Children (NOC).
4) Coupling Between Objects (CBO).
5) Response for a Class (RFC).
6) Lack of Cohesion in Methods (LCM).

Original Java Code

```java
class CFG_19 {
  static long modexp(long x, long n, long m) {
    if (n == 0) {
      return 1;
    } else if (n % 2 == 0) {
      return modexp((x * x) % m, n / 2, m);
    } else {
      return (x * modexp((x * x) % m, (n - 1) / 2, m) % m);
    }
  }

  static long modInverse(long x, long m) {
    return modexp(x, m - 2, m);
  }

  static void countNumbers(String N) {
    long m = 1000000007;
    long factorial[] = new long[100001];
    factorial[0] = 1;
    for (int i = 1; i < 100001; i++) {
      factorial[i] = (factorial[i - 1] * i) % m;
    }
    long count[] = new long[10];
    for (int i = 0; i < 10; i++) {
      count[i] = 0;
    }
    long length = N.length();
    for (int i = 0; i < length; i++) count[N.charAt(i) - '0']++;
    long result = factorial[(int) length];
    for (int i = 0; i < 10; i++) {
      result = (result * modInverse(factorial[(int) count[i]], m)) % m;
    }
    System.out.println(result);
  }

  public static void main(String args[]) {
    String N = "0223";
    countNumbers(N);
  }
}
```

Python (ChatGPT)

```python
class CFG_19:
    @staticmethod
    def modexp(x, n, m):
        if n == 0:
            return 1
        elif n % 2 == 0:
            return CFG_19.modexp((x * x) % m, n // 2, m)
        else:
            return (x * CFG_19.modexp((x * x) % m, (n - 1) // 2, m)) % m

    @staticmethod
    def modInverse(x, m):
        return CFG_19.modexp(x, m - 2, m)

    @staticmethod
    def countNumbers(N):
        m = 1000000007
        factorial = [0] * 100001
        factorial[0] = 1
        for i in range(1, 100001):
            factorial[i] = (factorial[i - 1] * i) % m
        count = [0] * 10
        length = len(N)
        for i in range(length):
            count[ord(N[i]) - ord('0')] += 1
        result = factorial[length]
        for i in range(10):
            result = (result * CFG_19.modInverse(factorial[count[i]], m)) % m
        print(result)

if __name__ == "__main__":
    N = "0223"
    CFG_19.countNumbers(N)
```

Python (DeepSeek)

```python
def modexp(x, n, m):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return modexp((x * x) % m, n // 2, m)
    else:
        return (x * modexp((x * x) % m, (n - 1) // 2, m)) % m

def modInverse(x, m):
    return modexp(x, m - 2, m)

def countNumbers(N):
    m = 1000000007
    factorial = [1] * 100001
    for i in range(1, 100001):
        factorial[i] = (factorial[i - 1] * i) % m

    count = [0] * 10
    length = len(N)
    for c in N:
        count[int(c)] += 1

    result = factorial[length]
    for i in range(10):
        result = (result * modInverse(factorial[count[i]], m)) % m
    print(result)

N = "0223"
countNumbers(N)
```

Fig. 2. Comparison of translated Python code of ChatGPT and DeepSeek LLMs.

The metrics in source code translation encompass functionality equivalence, object-oriented metrics like CKMS, code similarity metrics like CrystalBLEU, comparative metrics for different high-level languages, and code plagiarism detection metrics. These metrics provide valuable insights into the correctness, design, efficiency, and originality of the translated code, facilitating the evaluation and improvement of source code translation techniques. The tools for evaluation of AI translated code such as Codegen [48] introduced heterogeneous solution and software metric CA (Computational Accuracy).

Data Collection and Analysis: Batches of randomly selected Java snippets from the XLCoST dataset were translated using both ChatGPT and DeepSeek, employing the same prompts for consistency. The translated outputs were recorded and aggregated into batches, then analysed in SonarQube for quality and correctness. The evaluation encompassed translation accuracy scoring, execution time measurement, and qualitative observations concerning language coverage and model flexibility. Statistical analysis techniques were applied to the resulting dataset to extract meaningful insights and elucidate comparative trends.

Comparative Analysis: A comparative assessment was carried out to evaluate the software quality of translations produced by ChatGPT and DeepSeek. Evaluation outcomes were compared across key dimensions, including software quality, translation effectiveness, language support, performance, and the capacity to handle diverse and complex code structures.

The findings are organised across three primary software quality domains, as established by SonarQube analysis:

Maintainability: Evaluated through the presence of code smells, duplicated code blocks, and cyclomatic complexity.

Reliability: Assessed by identifying bugs and potential logical errors in the translated output.

Security: Measured by detecting common vulnerabilities and risky coding patterns.

This structured methodology enables a robust comparison of code quality and correctness delivered by each large language model and aligns with current best practices in the evaluation of LLM-driven code translation.

In addition to the high-level metrics comparison shown in Fig. 4a, we conducted an in-depth analysis of low-level metrics, as depicted in Fig. 4b. These fine-grained metrics, utilised by SonarQube as rule-based tags, enable more nuanced filtering and evaluation of specific software quality attributes within the analysed source code.

The comparison of software metrics, as depicted in Fig. 4, indicates that DeepSeek consistently outperforms ChatGPT across all assessed domains. Furthermore, we observed that DeepSeek's translations are generally more efficient, frequently yielding fewer lines of code compared to those produced by ChatGPT. A comprehensive summary of the software quality measurements obtained through SonarQube analysis is provided in Table III.

The analysis of software quality metrics from SonarQube, as presented in Table III, reveals that the original Java code achieves the highest maintainability score (3432), which encompasses measurements of code smells, technical debt, technical debt on new code, technical debt ratio, and code complexity. In contrast, the translated outputs from ChatGPT (score: 1500) and DeepSeek (score: 832) demonstrate substantially lower values across maintainability, reliability, and security metrics. Java code exhibits the greatest volume of issues at all severity levels, especially within the medium (747) and low (3400) severity categories.

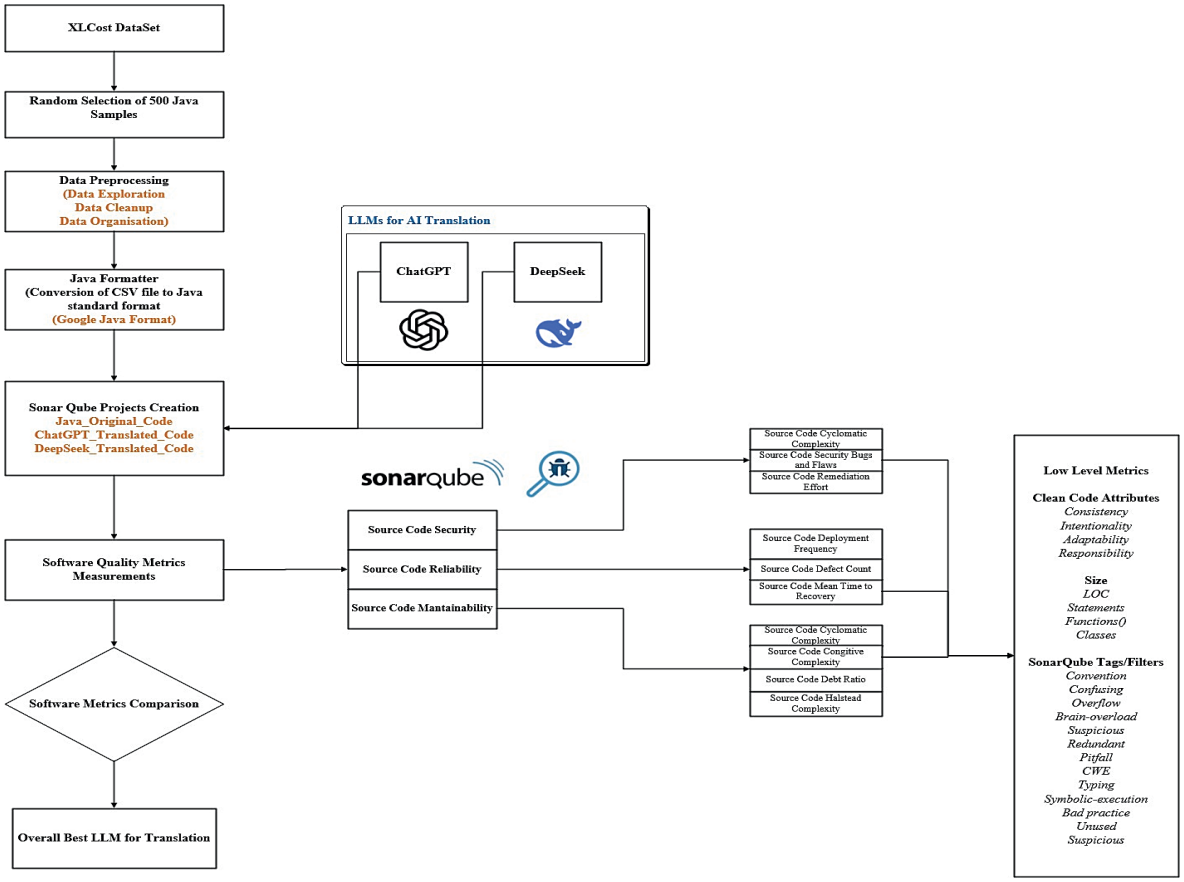
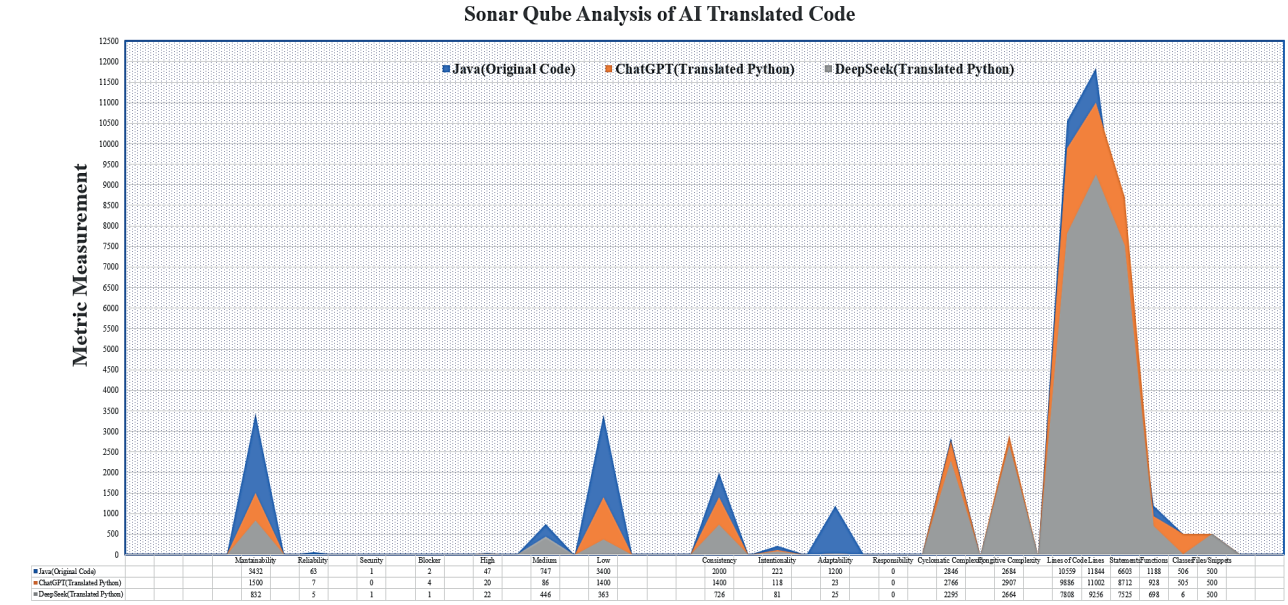


Fig. 3. Source code translation evaluation flow.

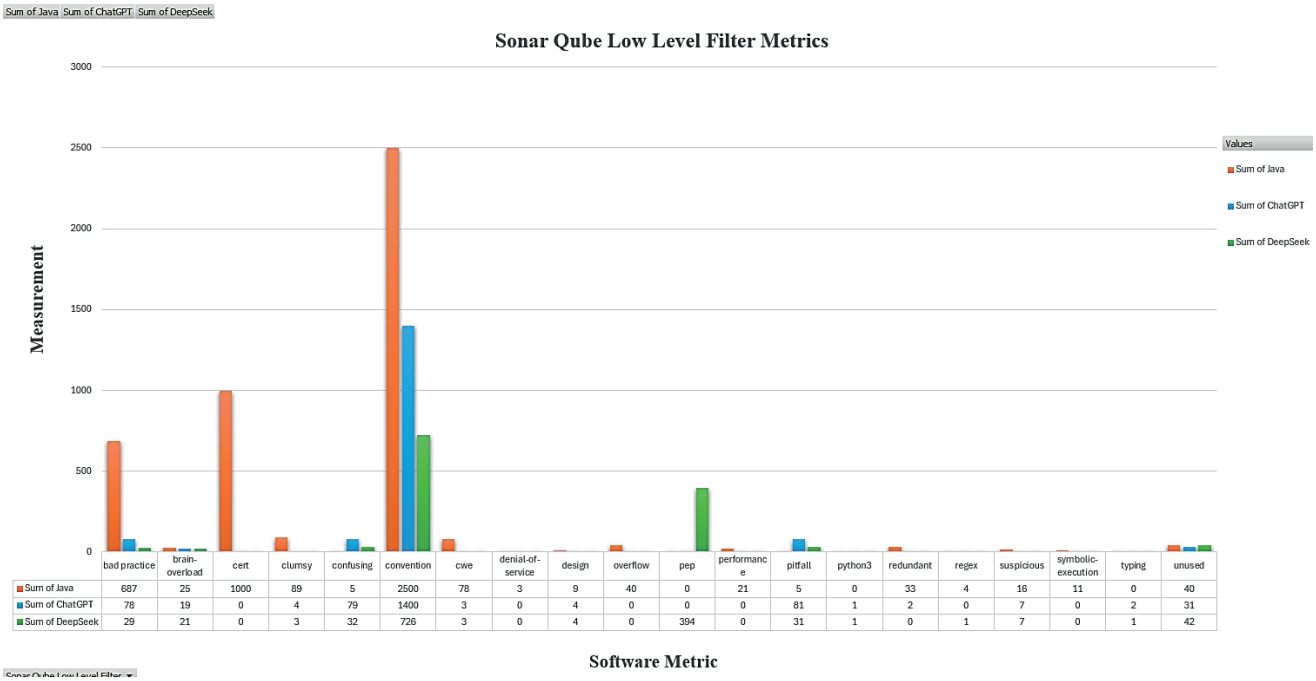
TABLE III

SOFTWARE METRICS MEASUREMENTS TRANSLATED CODE FROM CHATGPT AND DEEPSEEK LLMS

Metrics Category	Java (Original)	ChatGPT (Python)	DeepSeek (Python)
Maintainability	3432	1500	832
Reliability	63	7	5
Security	1	0	0
Severity Blocker	2	4	1
Severity High	47	20	22
Severity Medium	747	86	446
Severity Low	3400	1400	363
Clean Code: Consistency	2000	1400	726
Clean Code: Intentionality	222	118	81
Clean Code: Adaptability	1200	23	25
Clean Code: Responsibility	0	0	0
Cyclomatic Complexity	2846	2766	2295
Cognitive Complexity	2684	2907	2664
Size: LOC	10559	9886	7808
Size: Lines	11844	11002	9256
Size: Statements	6603	8712	7525
Size: Functions	1188	928	698
Size: Classes	506	505	6
Size: Programs/Files	500	500	500
Total Remediation Effort	6 h 45 m	1 h 30 m	1 h
Total Maintainability Debt	42 days	13 days	11 days



(a) Comparison of high-level software metrics

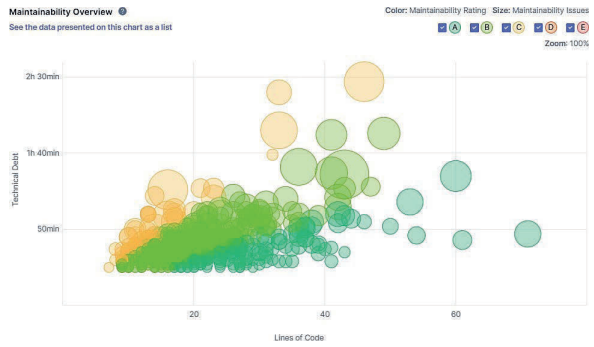


(b) Comparison of low-level software metrics

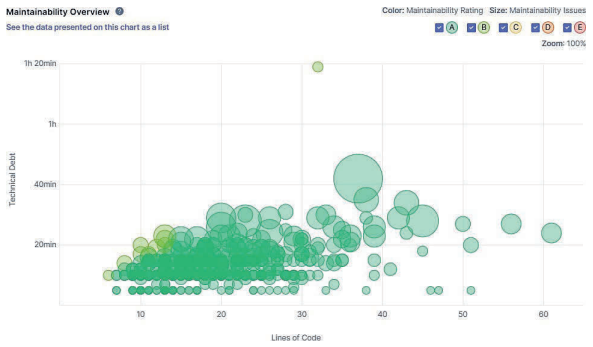
Fig. 4. Comparison of software metrics.

Among the translated outputs, DeepSeek demonstrates superior maintainability, evidenced by a notably lower technical debt of 11 days compared to ChatGPT's 42 days. In SonarQube, technical debt is calculated as the total estimated remediation effort required to resolve all maintainability issues (code smells), typically expressed in working days. The technical debt ratio quantifies this as the proportion of remediation cost to total development cost, highlighting the extent of maintainability challenges in each codebase.

Regarding low-level code quality attributes, the original Java code attains the highest scores in clean code characteristics such as consistency (2000), intentionality (222), and adaptability (1200), while both ChatGPT and DeepSeek fall short in these dimensions. Notably, all three code sets Java, ChatGPT, and DeepSeek report zero for responsibility metrics. Cyclomatic complexity is comparable between Java (2846) and ChatGPT (2766), whereas DeepSeek achieves greater simplicity with a lower value (2295).



(a) Java Code Maintainability Distribution



(b) ChatGPT Python Code (Translated) Maintainability Distribution

(c) DeepSeek Python Code (Translated) Maintainability Distribution

Fig. 5. Maintainability distribution of original Java, ChatGPT (Python) and DeepSeek (Python).

For size-related metrics, the Java codebase is the largest, comprising 10 559 lines of code and 1188 functions, followed by ChatGPT (9886 LOC and 928 functions) and DeepSeek (7808 LOC and 698 functions), confirming DeepSeek's ability to generate more concise and efficient code. Collectively, these results demonstrate that DeepSeek outperforms ChatGPT in code translation tasks, particularly with respect to maintainability, code size, and significant reductions in technical debt.

The overall distribution of the entire data analysed through SonarQube is depicted in Fig. 5. The figure clearly demonstrates that DeepSeek exhibits minimal maintainability issues and technical debt across the 500 randomly selected samples used for translation.

VI. Conclusion

By conducting empirical analysis of ChatGPT and DeepSeek, this study systematically evaluated and compared software metrics of LLM-translated code. Research assessed effectiveness, performance, and applicability of each model in source code translation contexts. Software quality indicators including execution time, language coverage, and flexibility were examined, offering comprehensive perspective on distinctive strengths and limitations of both approaches.

While AI can effortlessly generate code, it does not inherently ensure high-quality, maintainable output. Recent research in this area commonly employs static code analysis tools, such as SonarQube, PMD, and FindBugs, for Python and Java to assess code quality. However, these studies primarily focus on a broad codebase analysis rather than the specific challenges posed by code translation workflows. Additionally, their evaluations largely concern syntactic and semantic correctness of AI-translated code, without systematically quantifying improvements in software quality dimensions like reliability, maintainability, and overall software metrics.

This study insights suggest that supplying rich contextual information during code translation can enhance LLM output quality. However, prompt strategies effective for one LLM may not yield comparable improvements with another, due to variations in model architecture, training data, and fine-tuning methodologies. Employing static analysis tools can help uncover quality and security issues in AI-generated outputs. Integrating real-time static analysis platforms, such as SonarQube, into evaluation pipelines is essential for comprehensive translated code assessment and quality software development.

Despite being released three months before ChatGPT-4.1, DeepSeek demonstrates superior performance across all assessed software quality metrics for code translation. This outcome highlights the pivotal role of targeted pre-training and fine-tuning strategies in developing effective LLMs. Results from Java-to-Python translation experiments reveal notable quality gaps between DeepSeek and ChatGPT, reinforcing the importance of additional evaluation layers for AI-generated code.

This paper provides researchers, practitioners, and developers actionable insights for selecting appropriate source code translation solutions for specific use cases. This work

advances the field by delivering thorough evaluation of current SOTA techniques and promoting adoption of more robust and reliable translation methods.

VII. Future Work

As LLMs continue evolving with new content and architectural innovations, maintaining pace with these advancements ensures an ongoing, rigorous code quality analysis.

Introduction of New Code-Based LLMs: With AI expanding into agentic architectures and tool-using autonomous agents, LLM workflows are evolving at pace. Next generation LLM models are being developed with more efficient training paradigms, expanded context windows, and superior reasoning capabilities, moving closer to achieving nuanced, human-level comprehension and decision-making. Future research iterations will incorporate emerging LLMs, including Tesla Grok, Perplexity AI, Google Gemini, and Anthropic Claude. These models draw considerable attention due to multi-modal functionality, grounded reasoning, and real-time scenario applicability.

Enhanced Accuracy Metrics Assessment: Evaluating source code translation accuracy across programming languages remains inherently complex due to fundamental syntactic and semantic disparities. While automated tools provide substantial support, human expertise is often indispensable for interpreting edge cases, semantic ambiguities, and idiomatic constructs inadequately addressed by current automated systems.

Future research proposes integrating AI-driven code analysis platforms such as DeepCode, employing advanced machine learning algorithms for semantic analysis extending beyond conventional static analysis techniques. DeepCode represents new generation intelligent code review tools, leveraging large-scale code repositories and machine learning models to detect bugs, security vulnerabilities, and anti-patterns that might elude traditional rule-based systems.

System/Application-Level Dataset Expansion: While this study primarily evaluates software quality metrics on procedure-level code snippets in object-oriented languages, future research aims at expanding an analysis to encompass larger, system- or application-level datasets. Such extension would facilitate LLM evaluation on complex, multifaceted codebases, yielding a deeper understanding of translation capabilities within realistic software project environments.

System-level datasets would allow integration of advanced object-oriented metrics including Depth of Inheritance Tree (DIT), Number of Methods (NOM) per class, and Coupling Between Objects (CBO). These metrics are crucial for assessing maintainability, modularity, and overall complexity in object-oriented paradigms.

VIII. AI Ethics

Another important consideration is the ethical implications of using AI in source code translation. As with any technology, there are potential risks associated with the use of AI, such as the potential for biased or unfair translations. It will be important for researchers to carefully consider these issues and develop appropriate safeguards and guidelines for the use of AI in source code translation.

A recent study by PwC [104] suggests that 84 % of CEO agree that AI-based decisions need to be explainable to be trusted. There is a clear need to review the AI practice in source code translation and ask a series of key questions related to trustworthy AI [105], take steps to tackle the variety of risks from AI-based solutions. There is also a need to examine how the interactions between people and AI models influenced the accuracy, reliability, and fairness of their decisions. Although AI-supported decisions were more accurate, people struggled to determine the accuracy of their own decisions as well as the AI recommendations. Systems with artificial intelligence or machine learning components raise new challenges and require careful engineering.

References

[1] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976. https://doi.org/10.1109/TSE.1976.233837

[2] L. Coyle, M. Hinchey, B. Nuseibeh, and J. L. Fiadeiro, "Guest editors' introduction: Evolving critical systems," *Computer*, vol. 43, no. 05, pp. 28–33, May 2010. https://doi.org/10.1109/MC.2010.139

[3] *IEEE standard glossary of software engineering terminology*. Institute of Electrical and Electronics Engineers, IEEE Std 610.12-1990, 1990.

[4] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, Jun. 1992. https://doi.org/10.1145/130844.130856

[5] Transpiler Contributors, "Transpiler," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Transpiler

[6] J. M. Boyle and M. N. Muralidharan, "Program reusability through program transformation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 574–588, Sep. 1984. https://doi.org/10.1109/TSE.1984.5010281

[7] R. C. Waters, "Program translation via abstraction and reimplementation," *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1207–1228, Aug. 1988. https://doi.org/10.1109/32.7629

[8] M. Mirchev, A. Costea, A. K. Singh, and A. Roychoudhury, "Assured automatic programming via large language models," *arXiv preprint arXiv:2410.18494*, Oct. 2024. https://doi.org/10.48550/arXiv.2410.18494

[9] Q. Zhang, C. Fang, Y. Shang, T. Zhang, S. Yu, and Z. Chen, "No man is an island: Towards fully automatic programming by code search, code generation and program repair," *arXiv preprint arXiv:2409.03267*, Sep. 2024. https://doi.org/10.48550/arXiv.2409.03267

[10] D. Kc and C. T. Morrison, "Neural machine translation for code generation," *arXiv preprint arXiv:2305.13504*, Jan. 2023. https://doi.org/10.48550/arXiv.2305.13504

[11] M. Atemkeng, S. Hamlomo, B. Welman, N. Oyentunji, P. Ataei, and J. L. E. K. Fendji, "Ethics of software programming with generative AI: Is programming without generative AI always radical?" *arXiv preprint arXiv:2408.10554*, Aug. 2024. https://doi.org/10.48550/arXiv.2408.10554

[12] Z. Bahroun, C. Anane, V. Ahmed, and A. Zacca, "Transforming education: A comprehensive review of generative artificial intelligence in educational settings through bibliometric and content analysis," *Sustainability*, vol. 15, no. 17, Aug. 2023, Art. no. 12983. https://doi.org/10.3390/su151712983

[13] R. Zviel-Girshin, "The good and bad of AI tools in novice programming education," *Education Sciences*, vol. 14, no. 10, Art. no. 1089, Oct. 2024. https://doi.org/10.3390/educsci14101089

[14] B. A. Becker, P. Denny, J. Finnie-Ansley, A. Luxton-Reilly, J. Prather, and E. A. Santos, "Programming is hard – or at least it used to be: Educational opportunities and challenges of AI code generation," *arXiv preprint arXiv:2212.01020*, Dec. 2022. https://doi.org/10.48550/arXiv.2212.01020

[15] E. Ortega-Ochoa, J. Sabaté, M. Arguedas, J. Conesa, T Daradoumis, and S. Caballé, "Exploring the utilization and deficiencies of generative artificial intelligence in students' cognitive and emotional needs: a systematic mini-review," *Frontiers in Artificial Intelligence*, vol. 7, Nov. 2024, Art. no. 1493566. https://doi.org/10.3389/frai.2024.1493566
[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 2017.
[17] A. Ni, P. Yin, Y. Zhao, M. Riddell, T. Feng, R. Shen, S. Yin, L. Ye, S. Yavuz, C. Xiong, S. Joty, Y. Zhou, D. Radev, and A. Cohan, "L2ceval: Evaluating language-to-code generation capabilities of large language models," *arXiv preprint arXiv:2309.17446*, Jan. 2023. https://doi.org/10.48550/arXiv.2309.17446
[18] R. Takaichi, Y. Higo, S. Matsumoto, S. Kusumoto, T. Kurabayashi, H. Kirinuki, and H. Tanno, "Are NLP metrics suitable for evaluating generated code?" in *Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022,* Jyväskylä, Finland, Nov. 2022, pp. 531–537. https://doi.org/10.1007/978-3-031-21388-5_38
[19] Y. Shao, "Human-computer interaction environment monitoring and collaborative translation mode exploration using artificial intelligence technology," *Journal of Environmental and Public Health*, vol. 2022, Jan. 2022. https://doi.org/10.1155/2022/4702003
[20] R. Zou, W. Chang, S. Gao, and S. Qin, "Strategies for improving the quality of computer-assisted translation based on internet," *International Journal of Advanced Academic Studies*, vol. 4, no. 4, pp. 156–158, Oct. 2022. https://doi.org/10.33545/27068919.2022.v4.i4c.890
[21] N. Radziwill and M. C. Benton, "Evaluating quality of chatbots and intelligent conversational agents," *arXiv preprint arXiv:1704.04579*, Jan. 2017. https://doi.org/10.48550/arXiv.1704.04579
[22] C. Merow, J. M. Serra-Diaz, B. J. Enquist, and A. M. Wilson, "AI chatbots can boost scientific coding," *Nature Ecology Evolution*, vol. 7, pp. 960–962, Apr. 2023. https://doi.org/10.1038/s41559-023-02063-3
[23] R. Michel-Villarreal, E. L. Vilalta-Perdomo, D. E. Salinas-Navarro, R. Thierry-Aguilera, and F. S. Gerardou, "Challenges and opportunities of generative AI for higher education as explained by ChatGPT," *Education Sciences*, vol. 13, no. 9, Aug. 2023, Art. no. 856. https://doi.org/10.3390/educsci13090856
[24] J. Zhou, H. Müller, A. Holzinger, and F. Chen, "Ethical ChatGPT: Concerns, challenges, and commandments," *Electronics*, vol. 13, no. 17, Aug. 2024, Art. no. 3417. https://doi.org/10.3390/electronics13173417
[25] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A brief overview of ChatGPT: The history, status quo and potential future development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, May 2023. https://doi.org/10.1109/JAS.2023.123618
[26] R. Karanjai, L. Xu, and W. Shi, "Teaching machines to code: Smart contract translation with LLMs," *arXiv preprint arXiv:2403.09740*, Mar. 2024. https://doi.org/10.48550/arXiv.2403.09740
[27] M. A. K. Raiaan, M. S. H. Mukta, K. Fatema, N. M. Fahad, S. Sakib, M. M. J. Mim, J. Ahmad, M. E. Ali, and S. Azam, "A review on large language models: Architectures, applications, taxonomies, open issues and challenges," *TechRxiv*, Sep. 2023. https://doi.org/10.36227/techrxiv.24171183
[28] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, Jul. 2023. https://doi.org/10.48550/arXiv.2307.06435
[29] "C static code analysis clean code programming language," Jan. 2025. [Online]. Available: https://www.sonarsource.com/knowledge/languages/csharp
[30] "Code quality standards," Jan. 2025. [Online]. Available: https://www.sonarsource.com/solutions/quality
[31] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, Aug. 2018. https://doi.org/10.48550/arXiv.1808.01400
[32] B. Yang, X. Luo, J.-G. Li, X. Xie, H. Li, and J. Gao, "Deep code search," *arXiv preprint arXiv:1702.08347*, 2017. https://arxiv.org/pdf/2305.05959
[33] 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, Jan. 2019, Art. no. 40. https://doi.org/10.1145/3290353
[34] C. Xiao, "Comparison of differences between artificial intelligence translation and artificial translation," *Journal of Physics Conference Series*, vol. 1992, Aug. 2021, Art. no. 22079. https://doi.org/10.1088/1742-6596/1992/2/022079
[35] T. Marjanov, I. Pashchenko, and F. Massacci, "Machine learning for source code vulnerability detection: What works and what isn't there yet," *IEEE Security & Privacy*, vol. 20, no. 5, pp. 60–76, Aug. 2022. https://doi.org/10.1109/MSEC.2022.3176058
[36] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, 2018, pp. 200–210. https://doi.org/10.1145/3196321.3196334
[37] 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 *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, Apr. 2024, Art. no. 82. https://doi.org/10.1145/3597503.3639226
[38] Z. Yang, F. Liu, Z. Yu, J. 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," in *Proceedings of the ACM on Software Engineering.*, vol. 1, no. FSE, pp. 1585–1608, Jul. 2024. https://doi.org/10.1145/3660778
[39] U. Alon, M. Zilberstein, and O. Levy, "Code2vec: Learning distributed representations of code," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, pp. 2733–2739. https://doi.org/10.1145/3290353
[40] M. Allamanis, M. Brockschmidt, M. Khademi, and C. Sutton, "Convolutional, recurrent, and hybrid models for paraphrase detection," *arXiv preprint arXiv:1601.06744*, 2016.
[41] Y. Jiang and Z. Su, "Automated program repair: a literature review," *Foundations of Secure Computation*, vol. 89, no. 5, pp. 517–590, 2012.
[42] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428. https://doi.org/10.1145/2594291.2594321
[43] R. Robbes, C. Petitpierre, and M. Monperrus, "User evaluations of automatic software repair," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1–5, 2012.
[44] S. Sachdeva, O. Polozov, and S. Gulwani, "Effective program synthesis through efficient code transplantation," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 722–736.
[45] Y. Qi, D. Sachan, M. Felix, S. Padmanabhan, and G. Neubig, "Rosetta: Large scale cross-lingual resource for natural language processing," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018, pp. 306–316.
[46] Z. Feng *et al.*, "Code BERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, Feb. 2020. https://doi.org/10.48550/arXiv.2002.08155
[47] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, Nov. 2017. https://doi.org/10.48550/arXiv.1711.00740
[48] M. Allamanis and C. Sutton, "Transcoder: Analyzing code similarities for porting legacy code," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 527–546, 2017.
[49] P. Tembhekar, M. Devan, and J. Jeyaraman, "Role of GenAI in automated code generation within DevOps practices: Explore how generative AI" *Journal of Knowledge Learning and Science Technology*, vol. 2, no. 2, pp. 500–512, Oct. 2023. https://doi.org/10.60087/jklst.vol2.n2.p512
[50] J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz, "Investigating explainability of generative ai for code through scenario-based design," in *Proceedings of the 27th International Conference on Intelligent User Interfaces*, 2022, pp. 212–228. https://doi.org/10.1145/3490099.3511119
[51] A. Q. Jiang *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, Jan. 2024. https://doi.org/10.48550/arXiv.2401.04088
[52] D. Guo *et al.*, "DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, Jan. 2024. https://doi.org/10.48550/arXiv.2401.14196
[53] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, "OpenCodeInterpreter: Integrating code generation with execution and refinement," *arXiv preprint arXiv:2402.14658*, 2024.
[54] A. Deroy and S. Maity, "Code generation and algorithmic problem solving using Llama 3.1 405b," *arXiv preprint arXiv:2409.19027*, Sep. 2024. https://doi.org/10.48550/arXiv.2409.19027

[55] A. Sadik, A. Ceravola, F. Joublin, and J. Patra, "Analysis of ChatGPT on source code," *arXiv preprint arXiv:2306.00597*, Jun. 2023. https://doi.org/10.48550/arXiv.2306.00597
[56] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, Nov. 2023. https://doi.org/10.48550/arXiv.2311.10372
[57] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2023, pp. 2339–2356. https://doi.org/10.1109/SP46215.2023.10179324
[58] K. Huang, X. Wang, J. Fu, B. Xie, Z. Wang, Y. Yao, H. Guo, M. Tu, X. Sun, and H. Wang, "Codex GLUE: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, Feb. 2021. https://doi.org/10.48550/arXiv.2102.04664
[59] R. Puri *et al.*, "CodeNet: A large-scale ai for code dataset for learning a diversity of coding tasks," *IBM Watson AI*, 2021.
[60] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, "XLCoST: A benchmark dataset for cross-lingual code intelligence," *arXiv preprint arXiv:2206.08474*, Jun. 2022. https://doi.org/10.48550/arXiv.2206.08474
[61] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of NLP and software engineering: A survey on language models for code," *arXiv preprint arXiv:2311.07989v3*, Dec. 2023. https://arxiv.org/html/2311.07989v3
[62] H. Lu *et al.*, "DeepSeek-VL: towards real-world vision-language understanding," *arXiv preprint arXiv:2403.05525*, Mar. 2024. https://doi.org/10.48550/arXiv.2403.05525
[63] W. Kim, "Multimodal foundation models: a taxonomy and reflections," *International Journal of Web and Grid Services*, vol. 20, no. 4, pp. 505–531, Dec. 2024. https://doi.org/10.1504/IJWGS.2024.143177
[64] A. J. Adetayo, M. O. Aborisade, and B. A. Sanni, "Microsoft Copilot and Anthropic Claude AI in education and library service," *Library Hi Tech News*, Jan. 2024. https://doi.org/10.1108/LHTN-01-2024-0002
[65] M. Imran and N. Almusharraf, "Google Gemini as a next generation AI educational tool: a review of emerging educational technology," *Smart Learning Environments*, vol. 11, no. 1, May 2024, Art. no. 22. https://doi.org/10.1186/s40561-024-00310-z
[66] M. A. Akib, M. M. Mazumder, and S. Ahsan, "Analysis on LLMs performance for code summarization," *arXiv preprint arXiv:2412.17094*, Dec. 2024. https://doi.org/10.48550/arXiv.2412.17094
[67] D. Noever and F. McKee, "Numeracy from literacy: Data science as an emergent skill from large language models," *arXiv preprint arXiv.2301.13382*, Jan. 2023. https://doi.org/10.48550/arXiv.2301.13382
[68] 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 (MAPS 2022)*, Jun. 2022, pp. 1–10. https://doi.org/10.1145/3520312.3534862
[69] V. Corso, L. Mariani, D. Micucci, and O. Riganelli, "Assessing AI-based code assistants in method generation tasks," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion'24)*, May 2024, pp. 380–381. https://doi.org/10.1145/3639478.3643122
[70] A. Cohan, S. Feldman, I. Beltagy, D. Downey, and D. S. Weld, "Specter: Document-level representation learning using citation-informed transformers," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Jul. 2020, pp. 2270–2282. https://doi.org/10.18653/v1/2020.acl-main.207
[71] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk, "Evaluating and explaining large language models for code using syntactic structures," *arXiv preprint arXiv.2308.03873*, Aug. 2023. https://doi.org/10.48550/arXiv.2308.03873
[72] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, Jan. 2023. https://doi.org/10.1016/j.iotcps.2023.04.003
[73] L. Murr, M. Grainger, and D. Y. Gao, "Testing LLMs on code generation with varying levels of prompt specificity," *arXiv preprint arXiv.2311.07599*, Nov. 2023. https://doi.org/10.48550/arXiv.2311.07599
[74] "Java 2 CSharp Translator for Eclipse," *SourceForge*. [Online]. Available: https://sourceforge.net/projects/j2cstranslator
[75] S. Bhatia, S. Kohli, S. A. Seshia, and A. Cheung, "Building code transpilers for domain-specific languages using program synthesis (experience paper)," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023, pp. 1–30. https://d-nb.info/1367153115/34
[76] H. Lunnikivi, K. Jylkkä, and T. Hämäläinen, "Transpiling Python to rust for optimized performance," in *International Conference on Embedded Computer Systems*, Oct. 2020, pp. 127–138. https://doi.org/10.1007/978-3-030-60939-9_9
[77] F. A. Bastidas and M. Pérez, "A systematic review on transpiler usage for transaction-oriented applications," in *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)*, Cuenca, Ecuador, Oct. 2018, pp. 1–6. https://doi.org/10.1109/ETCM.2018.8580312
[78] M. Marcelino and A. M. Leitão, "Transpiling Python to Julia using PyJL," in *Proceedings of the 15th European Lisp Symposium (ELS'22)*, 2022, pp. 40–47.
[79] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," in *Generative AI for Effective Software Development*, A. Nguyen-Duc, P. Abrahamsson, and F. Khomh, Eds. Springer, Cham, Jan. 2024, pp. 71–108. https://doi.org/10.1007/978-3-031-55642-5_4
[80] Y. Chen, C. Wong, H. Yang, J. Aguenza, S. Bhujangari, B. C. Vu, X. Lei, A. Prasad, M. Fluss, E. Phuong, M. Liu, and J. Davis, "Assessing the impact of prompting, persona, and chain of thought methods on ChatGPT's arithmetic capabilities," *arXiv preprint arXiv.2312.15006*, Dec. 2023. https://doi.org/10.48550/arXiv.2312.15006
[81] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, "Unleashing the potential of prompt engineering in large language models," *Patterns*, vol. 6, no. 6, Jun. 2025, Art. no. 101260. https://doi.org/10.1016/j.patter.2025.101260
[82] B. Dornauer, M. Felderer, J. Weinzerl, M.-C. Racasan, and M. Heß, "SoHist: A tool for managing technical debt through retro perspective code analysis," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE '23)*, Jun. 2023, pp. 184–187. https://doi.org/10.1145/3593434.3593460
[83] A. Puspaningrum, M. A. A. Hilmi, Darsih, M. Z. Mustamiin, and M. I. Ginanjar, "Vulnerable source code detection using sonarcloud code analysis," in *Proceedings of the 5th International Conference on Applied Science and Technology on Engineering Science (iCAST-ES 2022)*, vol. 1, Bandung, Indonesia, Jan. 2022, pp. 683–687. https://doi.org/10.5220/0011862600003575
[84] V. Bhutani, F. G. Toosi, and J. Buckley, "Analysing the analysers: An investigation of source code analysis tools," *Applied Computer Systems*, vol. 29, no. 1, pp. 98–111, Jun. 2024. https://doi.org/10.2478/acss-2024-0013
[85] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, Lake Buena Vista, FL, USA, Oct. 2009, pp. 367–377. https://doi.org/10.1109/ESEM.2009.5314233
[86] "Most used languages among software developers globally 2021." [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used languages/
[87] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *arXiv preprint arXiv.2308.10462*, Aug. 2023. https://doi.org/10.48550/arXiv.2308.10462
[88] A. Malyala, K. Zhou, B. Ray, and S. Chakraborty, "On ML-based program translation: Perils and promises," *arXiv preprint arXiv.2302.10812*, Feb. 2023. https://doi.org/10.48550/arXiv.2302.10812
[89] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, "CodeTransOcean: A comprehensive multilingual benchmark for code translation," *arXiv preprint arXiv.2310.04951*, Oct. 2023. https://doi.org/10.48550/arXiv.2310.04951
[90] L. Chen, "Research on code generation technology based on LLM pre-training," *Frontiers in Computing and Intelligent Systems*, vol. 10, no. 1, pp. 69–75, Oct. 2024. https://doi.org/10.54097/scrwpt34
[91] C. Tony, N. E. D. Ferreyra, M. Mutas, S. Dhiff, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *arXiv preprint arXiv.2407.07064*, Jul. 2024. https://doi.org/10.48550/arXiv.2407.07064

[92] I. Paul, J. Luo, G. Glavaš, and I. Gurevych, "IRCoder: Intermediate representations make language models robust multilingual code generators," *arXiv preprint arXiv:2403.03894v1*, Mar. 2024. https://arxiv.org/html/2403.03894v1
[93] N. Raihan, C. D. Newman, and M. Zampieri, "Code LLMs: A taxonomy-based survey," *arXiv preprint arXiv.2412.08291*, Dec. 2024. https://doi.org/10.48550/arXiv.2412.08291
[94] P. Devanbu, "New initiative: The naturalness of software," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, Florence, Italy, May 2015, pp. 543–546. https://doi.org/10.1109/ICSE.2015.190
[95] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica, "LLM-assisted code cleaning for training accurate code generators," *arXiv preprint arXiv.2311.14904*, Nov. 2023. https://doi.org/10.48550/arXiv.2311.14904
[96] A. Back and E. Westman, "Comparing programming languages in Google Code Jam," Master's thesis, Chalmers University of Technology and University of Gothenburg, 2017. https://publications.lib.chalmers.se/records/fulltext/250672/250672.pdf
[97] 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. https://arxiv.org/pdf/1909.09436
[98] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, pp. 11783–11790, Jun. 2022. https://doi.org/10.1609/aaai.v36i10.21434
[99] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in Neural Information Processing Systems*, vol. 31, 2018.
[100] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, Sep. 2020. https://doi.org/10.48550/arXiv.2009.08366
[101] W. U. Ahmad, G. R. Tushar, S. Chakraborty, and K.-W. Chang, "AVATAR: A parallel corpus for java-python program translation," *arXiv preprint arXiv.2108.11590*, Aug. 2021. https://doi.org/10.48550/arXiv.2108.11590
[102] "Statistical Significance: Definition, Calculation, Importance." [Online]. Available: https://surveysparrow.com/blog/statistical-significance
[103] J. D. Weisz, M. Muller, S. I. Ross, F. Martinez, S. Houde, M. Agarwal, K. Talamadupula, and J. T. Richards, "Better together? An evaluation of AI-supported code translation," in *27th International Conference on Intelligent User Interfaces*, Mar. 2022, pp. 369–391. https://doi.org/10.1145/3490099.3511157
[104] PwC, "25th Annual Global CEO Survey." [Online]. Available: https://www.pwc.com/gx/en/ceo-agenda/ceosurvey/2022.html
[105] European commission, "Ethics guidelines for trustworthy AI," *Shaping Europe's digital future - European Commission*, Apr. 2019. [Online]. Available: https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai

Vikram Bhutani is a Principal Engineer (Artificial Intelligence) at Intel Corporation™ and PhD scholar at MTU, Cork, Ireland. Vikram holds an M.Sc. degree in Artificial Intelligence (AI) from the University of Limerick, Ireland.
Vikram embarked on his journey with Intel in 2002, amassing an impressive 22-year tenure as a professional in Pre-Si RTL validation/verification. Throughout his extensive career, Vik has been an integral force behind numerous Intel x86 CPU client projects within Intel, showcasing his expertise and leadership in key validation areas, most recently on AI in PC projects. As an IEEE Senior Member and a Fellow of IETE (India) and IEI (India), Vik has authored multiple publications in the areas of machine learning, hardware accelerators (NPU, GPU and CPU), x86 IA P-core validation, power and performance analytics, and micro-architectural features validation.
E-mail: vikram.bhutani@intel.com
ORCID iD: https://orcid.org/0000-0002-3565-2581

Farshad Ghassemi Toosi earned his PhD in Computer Science from the University of Limerick in Ireland in 2017. His doctoral research focused on data visualization, particularly in the domain of Graph Drawing, and the application of Genetic Algorithms in this field. Following the completion of his PhD, he served as a Post Doctorate for approximately two years, concentrating on Software Engineering with a specific emphasis on Source Code Manipulation and Feature Location. Subsequently, he assumed the role of a full-time Lecturer at the Department of Computer Science of Munster Technological University in Cork starting from 2019. Farshad Ghassemi Toosi is an esteemed academic member of Lero, the SFI Research Centre for Software in Ireland.
E-mail: farshad.toosi@mtu.ie
ORCID iD: https://orcid.org/0000-0002-1105-4819

Jim Buckley received his PhD in Computer Science from the University of Limerick, in 2002. He is a Professor with the Computer Science and Information Systems Department, University of Limerick, Ireland and is a Co-principal Investigator with Lero, the Irish Research Centre for Software. He was awarded the (Lero) Director's prize for Research Excellence in 2020 and his main research interests focus on supporting software developers who are tasked with maintaining and evolving software systems. Thus, specific areas of interests include feature location, software comprehension, and architectural analysis of such systems.
Email: Jim.Buckley@ul.ie
ORCID iD: https://orcid.org/0000-0001-6928-6746