



Chain-of-Thought LLM-Based Code Translation
Using Chain-of-Thought to Improve LLM-Based Translation of C++ to Java

Blago Gunev¹

Supervisor(s): Soham Chakraborty¹, Przemyslaw Pawelczak¹
¹EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Blago Gunev

Final project course: CSE3000 Research Project

Thesis committee: Prof. Dr. Soham Chakraborty, Prof. Dr. Przemyslaw Pawelczak, Prof. Dr. Arie van Deursen

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

In recent times, large language models (LLMs) have gained a lot of traction for their ability to assist in day-to-day tasks, especially coding tasks. At the same time, cybersecurity threats are becoming increasingly complex, sometimes requiring thorough analysis by multiple experts in the area. Therefore, it would be valuable to assess the ability of LLMs to aid in malware inspection work.

This research focuses on the use of LLMs as a code translation tool for beginner malware researchers. Novices to the topic can use such a tool to translate malware source code from C++ to Java, which will help them understand the functionality by making the code readable. Two zero-shot prompting frameworks are presented and evaluated on their efficiency, achieving a rate of syntactically correct outputs between 62.2% and 63.9%, of which between 39.0% and 42.9% produce functionally preserved translations.

1 Introduction

Large Language Models (LLMs) have exploded in popularity in the last few years due to their success in natural language processing (NLP) and text generation. They have also shown great performance in code generation [23] and code understanding tasks [16], exhibiting reasoning and knowledge synthesis capabilities that could make them a useful tool for cybersecurity researchers. Therefore, it would be worthwhile to investigate their practicality as assistants in malware reverse engineering and binary code analysis. Much research has already been done about the use of LLMs for cybersecurity purposes, and a comprehensive literature review has been composed by Jelodar et al. [13].

This research explores the use of LLMs as a translation tool from C++ to Java. Such a tool can be of great help to beginners in malware analysis who might not be familiar with C/C++ syntax. The focus will be on the impact different prompting methods have on the quality of the output. The following research questions will be answered:

RQ1: Does Chain-of-Thought prompting improve the compilation success rate of C++ programs translated to Java when compared to simple prompting?

RQ2: Does Chain-of-Thought prompting result in a higher rate of behaviour-preserving programs than simple prompting when translating C++ source code to Java?

RQ3: What is the difference in code quality between Chain-of-Thought and simple prompting of LLMs when translating C++ code to Java?

Chain-of-Thought (CoT) prompting [24] is shown to improve the reasoning capabilities of LLMs by prompting them to reason on a task before completing it. Its main benefit compared to other methods is that it produces better outputs without requiring any form of pre-training, which is typically a performance-intensive process. Therefore, it can be used with any off-the-shelf model, and whenever a new one is presented, a user can take advantage of its improvements with the same strategy.

Extensive study on the bugs produced by LLMs in code translation tasks has been done by Pan et al. [19]. Their findings report varying success rates by LLMs in translation between different source and target programming languages, with up to 47.3% reported for GPT-4. Their work focused on multi-shot prompting, leaving zero-shot prompting unexplored. As their approach relied on one prompting method, no study was done on the effect different prompts have on the outputs of the code.

The main contributions of this paper are a CoT prompting approach to LLM-based code translation and an evaluation of the differences in produced code between the CoT approach and a simple approach. An experimental framework is established to translate C++ code to Java, which is then tested for syntax and semantic correctness.

2 Methodology for Evaluating LLM-Based Code Translation

Subject of research. The LLM chosen for the purposes of this research is **Qwen3.5-9B**[22]. At the time of writing this paper, it is considered state-of-the-art; therefore, it should represent the best attainable performance from an LLM. Its size is relatively small compared to other popular open-source models, including ones from the same Qwen family, which makes it usable on personal computer hardware.

The languages chosen are C++ and Java due to their widespread use, structural distinction, and expressivity. Those qualities allow for sufficient behavioural analysis in translation by LLMs, i.e., specific shortcomings or strong points could be identified. The C/C++ family is still predominantly used for malware development [6; 4], while Java is often used to teach object-oriented programming principles.

Test data. To verify the correctness of the translation, a dataset of Codeforces problems was chosen [20]. It contains human-written sources in different languages (C++ and Java included), whose behaviour is verified by test suites of input-output pairs. This dataset serves as a controlled environment to test for correctness of functional behaviour before extending the research to non-functional correctness. A total of 208 C++ correct solutions for different problems were selected.

Prompting methods. The chosen method to improve the quality of the LLM output is Chain-of-Thought prompting. It is shown to significantly improve the reasoning abilities of LLMs in certain tasks [24], which sparks the question of whether it can help in code translation. The main focus will be on zero-shot prompting to keep the translation mechanism similar to that of compilers. It is expected that the non-deterministic structure and non-perfect knowledge resource of the LLM will lead to failed samples. Previous research on LLM-based code translation has shown that conversing with the model leads to significant improvements [19]; therefore, it will not be investigated in this research.

3 LLM-Based Code Translation

Two frameworks for LLM-based code translation are presented – a simple one and a Chain-of-Thought-based one. The simple one consists of briefly conveying the task and

prompting the LLM to complete it. The CoT approach instructs the LLM to make a step-by-step plan, identify potential problem points and only then translate the code.

3.1 Simple Prompt Template

The simple prompt consists of three sentences which serve a different purpose – state the task, give context and specify constraints. The full text is laid out below:

Translate this C++ code to Java code.
The main class name should be {class_name}.
Output only the code of the resulting program.
<C++ source code>

The class name parameter is provided to avoid compilation errors due to Java’s strict class naming conventions. The test data set used provides C++ sources along with their problem identifier and contest ID. Problem identifiers are usually a single Latin letter, with some problems having a numerical suffix, while contest IDs are integers up to 4 digits in length. The class name is constructed as <problem.id>_<contest.id>.

3.2 Chain-of-Thought Prompt Template

The CoT prompt retains the structure of the simple prompt – task, context, and constraints – while elaborating more on the context. The constraints sentence is modified to enable the LLM to output its thinking process and gives clear instructions on the expected format of the final code. The full text is laid out below:

Translate this C++ code to Java code.
The main class name should be {class_name}.
First, make a plan for the translation by considering what are C++ specific features and how they could be translated. Also consider whether the work could be split up in small incremental tasks.
Then, output the code of the resulting program with token <CODE> at the beginning and at the end.
<C++ source code>

Similarly to the first prompt strategy, the class name parameter is provided to avoid compilation errors.

It is thus expected that the LLM will first output its plan (CoT), which is followed by token <CODE> at the beginning of the code block. The code block should be closed by a </CODE> token.

4 Experimental Setup and Results

The experiment flow is illustrated in Figure 1. The experiment is split in two parts according to resource needs – LLM part, consisting of prompting the LLM and getting a response, and benchmark part, consisting of compilation, behaviour tests and code quality measurements.

4.1 LLM Setup

The LLM prompting part of the research was carried out on the DelftBlue supercomputer [8]. It grants access to Nvidia A100 compute nodes with 10 GB of VRAM, which provide resources to run sufficiently large models, such as the chosen Qwen3.5-9B. CUDA version 12.9 was used.

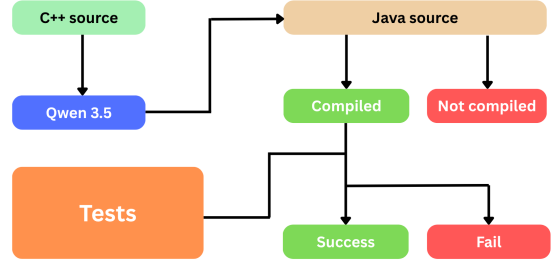


Figure 1: Illustration of the pipeline for translating and evaluating the code. The C++ sources are fed into the LLM with the prompts described in section 3. The produced Java sources are first tested for compilation correctness. The successfully compiled ones are tested for behaviour correctness.

The prompt pipeline was built in Python 3.12, based on the **transformers** library, version 5.9.0 [25]. **Bitsandbytes** [9] was used for 4-bit quantisation to reduce VRAM allocation and allow for a bigger context window. The choice of used libraries was largely influenced by hardware and LLM compatibility needs. The model parameters used are suggested by the creators of Qwen3.5 [22] – temperature = 0.6, top_p = 0.95, top_k = 20, min_p = 0.0, and repetition_penalty = 1.0.

Output was constrained to 30 000 new tokens to prevent endless reasoning loops, whose impact is described in the latter sections. The LLM part was run for 100 hours to produce a pair of responses for all 208 samples.

4.2 Benchmark Setup

The benchmarking step of the pipeline was carried out on Apple M1 with 16GB of system RAM, running macOS 26.5. The process is orchestrated in Python 3.12 with subprocesses. The compilation step invokes the Java compiler (Java 25) for each file separately. The successfully compiled files are tested on the matching test pairs in the open-r1/codeforces dataset. Input is passed through standard input; standard output is recorded in a file for later benchmarking and investigation of errors. To limit execution time and avoid infinite loops, a time limit of 5 seconds per test case is enforced, while the usual time limits given for competitive programming problems are 2 seconds. The objective of this research is not to investigate execution speed, so this limit is a reasonable assumption.

4.3 Prompt Extraction Results

The first step of getting results is extracting the Java code from the LLM response. Prompt extraction results are presented in Figure 2 and Figure 3. Out of 208 samples, for the simple prompt strategy, 196 “Java-esque” responses were recorded. Manual investigation of the data reveals the following failures:

1. Model gets stuck in a reasoning loop. This results in an endless stream of text up to the limit of the context window.
2. Not translating the input to Java. Some of the inputs were output as C++ code.

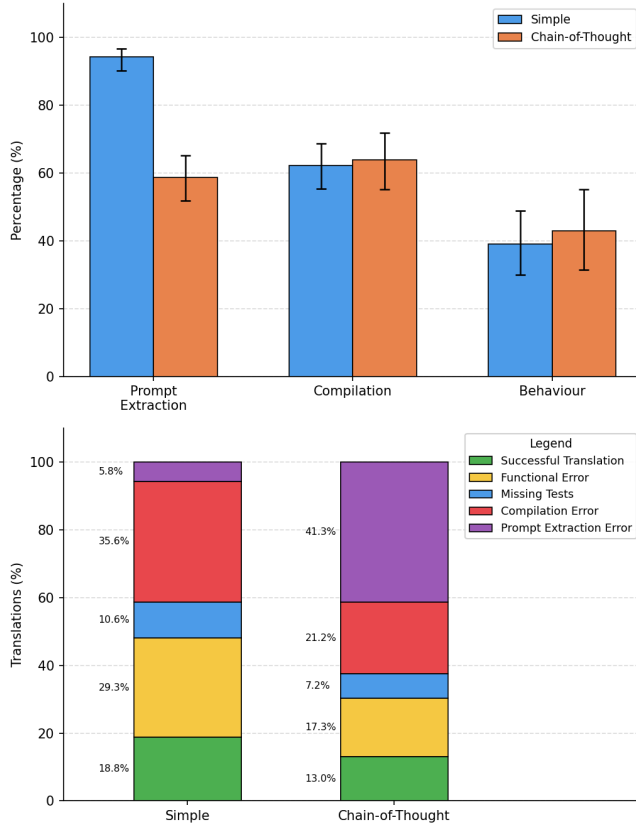


Figure 2: Results summary. The top chart presents the success rates at prompt extraction, compilation and behaviour verification steps. Data presented with 95% Wilson confidence intervals. The bottom chart presents the percentage-wise distribution of samples across prompt extraction, compilation and functional errors, missing tests and functionally correct translations.

Out of the 208 responses from the CoT strategy, 122 had the correct format listed in subsection 3.2. Due to the higher amount of failures of this strategy, no in-depth manual analysis was made, but some samples were manually investigated. The following failures were discovered:

1. Incorrect format, i.e., the token `<CODE>` was modified or not used by the LLM.
2. Endless code stream up to the end of the context window. It must be noted that even if the `<CODE>` token is put in the beginning, it is missing at the end due to truncated output.

4.4 Compilation Results

The samples which were successfully extracted in the previous step are evaluated on syntax correctness. Results are presented in Figure 2 and Figure 3. Of the simple prompt samples, 122 could be compiled, and of the CoT samples, 78 could be compiled. This results in 62.2% success in compilation for the simple strategy and 63.9% for the CoT one. CoT prompting yields slightly better results, but both measurements are within the 95% Wilson confidence intervals; thus, it is hard to draw conclusions about **RQ1**.

4.5 Behaviour Results

The successfully compiled samples are evaluated for functional correctness. Results are presented in Figure 2 and Figure 3. Programs that exhibit the correct behaviour amount to 39.0% for the simple prompt strategy and 42.9% for the CoT one. These results are within the confidence intervals of each other; therefore, no definitive conclusions about **RQ2** can be drawn.

It can be seen that the number of tested samples is smaller than the number of compiled samples for both proposed strategies. Deeper investigation showed that this is a result of the test dataset missing test cases for particular problems.

4.6 Code Quality Metrics

All successfully extracted source files were evaluated for the code quality metrics. The results are presented in Figure 4. Empty lines of code do not count towards any of the metrics, and lines that do not have functional behaviour are not included in the lines of code metric. Lines that have both functional behaviour and a comment in them count towards the score for both metrics. Average lines of code for the simple and CoT prompting strategies are 80.38 and 87.15, respectively. The average lines of comments for both strategies are 56.10 and 17.59, respectively. Addressing **RQ3**, CoT prompting results in a higher amount of effective lines of code, while the simple prompting produces code with more lines of comments.

5 Discussion

The results presented in section 4 indicate a slight increase of 1-2% in performance for syntax correctness and about 4% for semantic correctness, although it should be noted the number of data samples is too small to draw definitive conclusions. This measured improvement has a small chance of impacting the real-world use of an LLM-powered tool. The main benefit of LLMs is that a person can converse with them and present, for example, the compile error message or a wrong test case, which will allow the LLM to improve the output further. Previous research on the topic has shown that about 48% of problems are fixed after the first message and a further 17% after the second [19].

The code quality measurements show that the average lines of code per program stay mostly equal between prompting methods, which suggests that both strategies produce code of roughly the same functional complexity. The results also show a vastly higher average amount of comments for the simple prompt strategy. Investigation exposed the same problem found in subsection 4.3 – some of the extracted samples represent a valid prefix of a Java program and thus were accepted in the prompt extraction step.

5.1 Comparison to Other Research

Pan et al. [19] investigate the performance of LLMs in translating between many sources and target programming languages, which include C++ and Java. Their work investigates several models – CodeGen, CodeGeeX, StarCoder and more, none of which are from the Qwen3.5 family – and does not consider prompt failures due to their experimental design;

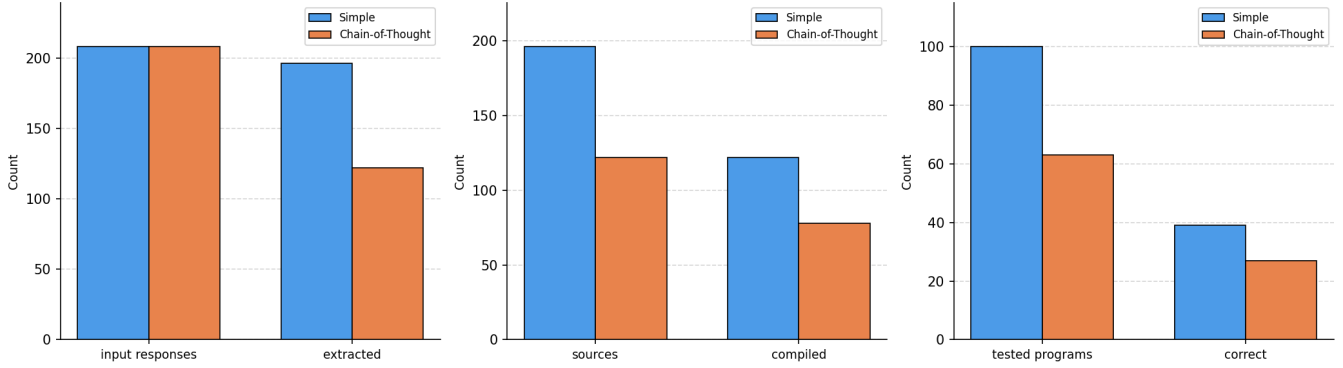


Figure 3: Results at each step of the testing pipeline. From left to right – source code extraction metrics, compilation success metrics, and functional correctness metrics.

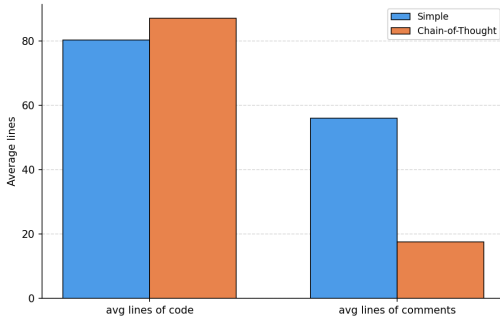


Figure 4: Code quality metrics. Average lines of code computed over all lines with functional code. Average lines of comments computed over lines that contain a single comment or are part of a comment block.

therefore, it is difficult to draw direct comparisons to their results. The data is also aggregated by source language; thus, it is impossible to know the result for C++-to-Java translation specifically.

Scaling the data from this research paper to exclude prompt extraction errors and missing tests, the simple prompt achieves about 22.5% successful translation, while the CoT prompt achieves 25.2%. These results are most similar to the results of CodeGen, which achieves 14% correct translations from C++, but it should be noted that this comparison also relies on the assumption that success rates between target languages are similar.

5.2 Threats to Validity

LLM train data. Some of the plans extracted from the CoT prompts were manually evaluated to gain insights into the work of the LLM, and it was found that they often contained references to competitive programming concepts, such as input time optimisation. This evidence strongly implies that Qwen was trained on competitive programming datasets, which means it might have encountered the data of this experiment in its training process and might have memorised solutions in both C++ and Java. Further investigation must be done to confirm that the translation is replicating the be-

haviour of the source rather than providing an alternative solution.

Model choice. The experiment was conducted on a single LLM – Qwen3.5-9B. Despite being relatively recent, it cannot be relied upon to make conclusions about the general usability of LLMs for code translation. This is due to differences in architecture, training data and behaviour alignment measures that might have been taken when releasing a model for open use.

Malware behaviour. This research presented a use case for translating malware for learning purposes, but the evaluation did not use malware samples. The choice to use the dataset described in section 2 was motivated due to its completeness and user-generated nature, therefore serving as a foundation to future research but not being conclusive.

Prompt templates. The CoT prompt resulted in high failure rates for extracting the code. Those failures could be a result of a poor prompt or a lack of pre-training for the task. The conditions for response validity could also be relaxed if other strategies do not alleviate the problem.

6 Related Work

LLM code translation. Despite the non-deterministic nature of LLMs making them impractical for compiler use, research has been done to try and achieve an LLM compiler nonetheless. Cummins et al. proposed LLM Compiler [7], an optimising compiler working on LLVM-IR and assembly code. Their evaluation showed that it achieves “77% of the optimising potential”. Other frameworks for LLM translation are INTERTRANS [14] and UniTrans [26], which use intermediate translation languages and automated test generation, respectively.

Non-LLM code translation. Tools for code translation have been developed on top of program-analysis techniques. Such tools include C2Rust [12] and C2Go [11], which translate C to Rust and Go, respectively, and Cappuccino [5] is a C++ to Java transpiler. Similar to LLM approaches, they are not able to translate any code perfectly but are an alternative and predictable approach.

7 Responsible Research

7.1 Ethical Reflection

Even though this research focuses on the application of LLMs as malware analysis assistants, concerns can be voiced about the possibility of LLMs being used for malicious tasks. Research has been done on generating malware with LLMs [1], and while it can be used by adversaries, the insights from the workings of such models are valuable for red team experts, also known as "attack teams".

Another cause for concern is the possibility of incorrect outputs. LLMs are not perfect in their knowledge and are prone to errors; therefore, users should be warned of the potential problems before use. Translations made by an LLM should not be trusted blindly.

7.2 Reproducibility

The open-r1/codeforces [20] dataset is published under the Open Data Commons Attribution License (ODC-By) 4.0 [17], which allows use granted the authors of the dataset and the ODC licence are credited. The Qwen3.5-9B [22] model is distributed under the Apache 2.0 license [3], which allows free use for research purposes.

To make the presented research reproducible, the full code is published online¹. Included are the chosen C++ code samples and test data. Due to the non-deterministic nature of LLMs, it is highly likely that a new run of the experiment will produce different results from the ones presented here. To achieve a stable outcome of the experiment, efforts were made to obtain as large a data sample as possible within the time constraints of the projects. The responses are also published in the repository to make the results verifiable.

8 Conclusions and Future Work

Large language models have found much use as a tool for learning and have demonstrated great performance in coding tasks. The focus point of this paper was to investigate their ability to translate C++ code to Java to help beginners in learning the syntax of malware sources and propose a method to improve said ability. The empirical evaluation of a zero-shot simple prompt approach and a Chain-of-Thought (CoT) approach showed that both methods have similar performance. There is some evidence to show CoT is an improvement, but further investigation should be done. Chain-of-Thought prompting also produced, on average, longer code, which suggests that it is suited for more complex code. Neither method would be practical in real use, as the rate of successful translations is about one in five.

Furthermore, all tests were performed on benign code, which is not representative of a real-world use of the code translation tool for cybersecurity beginners proposed in this paper. The experimental methodology can be extended with non-functional behaviour tests and verification by formal proof methods. It would also be interesting to see the performance when translating from or to a less popular language, which could happen if a malware source is obfuscated. The

¹GitHub Repository – <https://github.com/BlagoGunev/TUDelft-CSE-research-project>

framework for such an experiment could be modified to include a specification of this language, or the model can be extended with web search functionality.

9 AI Usage Disclosure

9.1 Implementation Part

The LLM part of the research was implemented by hand. Claude AI [2] was used for syntax reference and to help overcome environment setup problems on the DelftBlue supercomputer [8]. All output from the model was manually inspected and verified not to have malicious intent.

The first iteration of the benchmark part was implemented by hand. GitHub Copilot [15] was used as an in-editor code agent. The model was prompted with a detailed overview of necessary functionality and the first iteration as a template. The final version of the benchmark was implemented by Copilot with GPT-5.4-mini as the backend and was manually verified to only collect metrics as described in section 4.

Claude AI was used to implement the code for visualisation generation. Multiple revisions of the graphics were created, and inspiration from previous research papers [19] was drawn. Generated code was manually inspected and verified. Small changes to formatting were implemented by hand.

9.2 Writing Part

All text in this research paper was written by hand. ChatGPT [18] was used for inspiration on the structure format, and its web search functionality was utilised to look up and summarise relevant work on the topic of LLM malware analysis. Elicit [10] was used as a search engine for related work and references. Grammar and fluency were checked with Quillbot [21].

References

- [1] ACOSTA-BERMEJO, R., TERRAZAS-CHAVEZ, J. A., AND AGUIRRE-ANAYA, E. Automated Malware Source Code Generation via Uncensored LLMs and Adversarial Evasion of Censored Model. *Applied Sciences (Switzerland)* 15, 17 (2025). <https://doi.org/10.3390/app15179252>.
- [2] ANTHROPIC. Claude AI. <https://claude.ai>.
- [3] APACHE SOFTWARE FOUNDATION. Apache License, Version 2.0. <https://www.apache.org/licenses/LICENSE-2.0>.
- [4] APOSTOLOPOULOS, T., KOUTSOKOSTAS, V., TOTOSIS, N., PATSAKIS, C., AND SMARAGDAKIS, G. Coding Malware in Fancy Programming Languages for Fun and Profit. In *CODASPY 2025 - Proceedings of the 15th ACM Conference on Data and Application Security and Privacy* (2025). <https://doi.org/10.1145/3714393.3726506>.
- [5] BUDDRUS, F., AND SCHÖDEL, J. Cappuccino-A C++ to Java translator. In *Proceedings of the ACM Symposium on Applied Computing* (1998), vol. 02-February-1998. <https://doi.org/10.1145/330560.331015>.

- [6] CALLEJA, A., TAPIADOR, J., AND CABALLERO, J. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security* 14, 12 (2019). <https://doi.org/10.1109/TIFS.2018.2885512>.
- [7] CUMMINS, C., SEEKER, V., GRUBISIC, D., ROZIERE, B., GEHRING, J., SYNNAEVE, G., AND LEATHER, H. Meta large language model compiler: Foundation models of compiler optimization, 2024. <https://doi.org/10.48550/arXiv.2407.02524>.
- [8] DELFT HIGH PERFORMANCE COMPUTING CENTRE (DHPC). DelftBlue Supercomputer (Phase 2). <https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2>, 2024.
- [9] DETTMERS, T., LEWIS, M., SHLEIFER, S., AND ZETTMLOYER, L. 8-bit optimizers via block-wise quantization. *9th International Conference on Learning Representations, ICLR* (2022). <https://doi.org/10.48550/arXiv.2110.02861>.
- [10] ELICIT. Elicit AI. <https://elicit.com>.
- [11] GOTRANSPILE. CxGo Transpiler. <https://github.com/gotranspile/cxgo>.
- [12] IMMUNANT. C2Rust Transpiler. <https://github.com/immunant/c2rust>.
- [13] JELODAR, H., BAI, S., HAMED, P., MOHAMMADIAN, H., RAZAVI-FAR, R., AND GHORBANI, A. Large Language Model (LLM) for Software Security: Code Analysis, Malware Analysis, Reverse Engineering. <https://doi.org/10.48550/arXiv.2504.07137>.
- [14] MACEDO, M., TIAN, Y., NIE, P., COGO, F. R., AND ADAMS, B. INTERTRANS: Leveraging Transitive Intermediate Translations to Enhance LLM-Based Code Translation. In *Proceedings - International Conference on Software Engineering* (2025). <https://doi.org/10.1109/ICSE55347.2025.00236>.
- [15] MICROSOFT. GitHub Copilot. <https://github.com/features/copilot>.
- [16] NAM, D., MACVEAN, A., HELLENDORF, V., VASILESCU, B., AND MYERS, B. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 4 2024), ACM, pp. 1–13. <https://doi.org/10.1145/3597503.3639187>.
- [17] OPEN KNOWLEDGE FOUNDATION PROJECT. Open Data Commons Attribution License (ODC-BY). <https://opendatacommons.org/licenses/by/1.0/>.
- [18] OPENAI. ChatGPT. <https://chatgpt.com>.
- [19] PAN, R., IBRAHIMZADA, A. R., KRISHNA, R., SANKAR, D., WASSI, L. P., MERLER, M., SOBOLEV, B., PAVULURI, R., SINHA, S., AND JABBARVAND, R. Lost in Translation: A Study of Bugs Introduced by Large Language Models While Translating Code. In *Proceedings - International Conference on Software Engineering* (2024). <https://doi.org/10.1145/3597503.3639226>.
- [20] PENEDO, G., LOZHKO, A., KYDLÍČEK, H., ALLAL, L. B., BEECHING, E., LAJARÍN, A. P., GALLOUÉDEC, Q., HABIB, N., TUNSTALL, L., AND VON WERRA, L. Codeforces. <https://huggingface.co/datasets/open-rl/codeforces>, 2025.
- [21] QUILLBOT. Quillbot. <https://quillbot.com>.
- [22] QWEN TEAM. Qwen3.5: Towards native multimodal agents, February 2026. <https://qwen.ai/blog?id=qwen3.5>.
- [23] WANG, J., AND CHEN, Y. A Review on Code Generation with LLMs: Application and Evaluation. In *Proceedings - 2023 1st IEEE International Conference on Medical Artificial Intelligence, MedAI 2023* (2023). <https://doi.org/10.1109/MedAI59581.2023.00044>.
- [24] WEI, J., WANG, X., SCHUURMANS, D., BOSMA, M., ICHTER, B., XIA, F., CHI, E. H., LE, Q. V., AND ZHOU, D. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems* (2022), vol. 35. <https://doi.org/10.48550/arXiv.2201.11903>.
- [25] WOLF, T., DEBUT, L., SANH, V., CHAUMOND, J., DELANGUE, C., MOI, A., CISTAC, P., RAULT, T., LOUF, R., FUNTOWICZ, M., DAVISON, J., SHLEIFER, S., VON PLATEN, P., MA, C., JERNITE, Y., PLU, J., XU, C., SCAO, T. L., GUGGER, S., DRAME, M., LHOEST, Q., AND RUSH, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct. 2020), Association for Computational Linguistics, pp. 38–45. <https://doi.org/10.18653/v1/2020.emnlp-demos.6>.
- [26] YANG, Z., LIU, F., YU, Z., KEUNG, J. W., LI, J., LIU, S., HONG, Y., MA, X., JIN, Z., AND LI, G. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024). <https://doi.org/10.1145/3660778>.