Automated Translation of Real-World Codebases: How Far Are We?

Cristina David[1(✉)], Hanliang Zhang[1,2], and Meng Wang[1]

[1] University of Bristol, Bristol, UK
cristina.david@bristol.ac.uk
[2] Amazon, Seattle, WA, USA

Abstract. Automated translation of legacy software into modern languages is essential for adopting safer programming practices at scale. We review current rule-based, neural and neurosymbolic approaches and show why none fully address the needs of real-world repositories. Rule-based tools scale but mirror the source language, producing unidiomatic target code. LLM-based methods capture idioms but lack correctness guarantees. Hybrid systems partially bridge the gap but remain brittle when faced with complex features such as concurrency or third-party dependencies.

Keywords: Code translation · Large Language Models · Rust

1 Introduction

Software is at the heart of modern life, from banking systems and medical devices to critical infrastructure. Yet ensuring software reliability remains a major challenge, especially as systems grow in size and complexity. A long-standing and pervasive source of reliability issues is memory-safety bugs. Languages such as C and C++, which permit unchecked pointer manipulation, have been responsible for such bugs for decades, with reports from Microsoft and Google revealing that approximately 70% of their security vulnerabilities are rooted in memory errors [7,25].

Traditional approaches—manual code review, testing and formal verification tools—help identify bugs but struggle to keep pace with the size and evolution of modern software systems. As a result, the software engineering community is increasingly embracing modern, safety-focused programming languages that proactively prevent many classes of vulnerabilities from arising in the first place. For instance, languages like Java, Go, C# and Kotlin ensure memory safety through garbage collection, balancing safety with performance. Newer languages such as Swift and Rust take a different approach, using ownership

This work was done prior to joining Amazon.

A. Goharshady and C. Haase (Eds.): SETTA 2025, LNCS 16458, pp. 3–10, 2026.
https://doi.org/10.1007/978-981-95-7826-9_1

models and compile-time checks to achieve memory safety without incurring a runtime overhead.

While adopting modern languages for new development is relatively straightforward, modernising the enormous legacy codebases written in older, unsafe languages remains the major barrier to practical migration. Translation in today's industrial settings is still predominantly manual, requiring deep domain knowledge, considerable engineering resources, and months, if not years, of sustained effort [24,30]. This paper examines whether automated code translation can realistically meet the demands of real-world repositories.

Trade-off Between Scalability, Correctness and Idiomaticity. While automated code translation has received sustained attention across a wide range of language pairs, substantial challenges remain—particularly when the goal is to produce idiomatic, readable and scalable translations suitable for integration into real-world repositories. Traditional rule-based systems offer stability and semantic rigor, but their translations tend to closely mirror the structure and idioms of the source language, yielding code that is syntactically correct yet hard to understand, modify or extend in the target ecosystem. At the other extreme, data-driven or LLM-based approaches can generate high-quality, human-like translations that capture idioms, conventions and design patterns, but lack formal guarantees of semantic preservation and struggle to scale to large, real-world, projects. Consequently, *current techniques are effectively constrained by a three-way trade-off between idiomaticity, scalability and correctness*: improvements along one axis often degrade performance along the others, and different approaches choose different compromises. Breaking this trade-off and achieving translations that are both high quality and applicable to large, evolving software systems remains a core challenge for automated code translation research.

The remainder of this paper examines different translation directions and outlines the unresolved research problems that must be addressed. For concreteness, we focus our discussion on tools that translate other languages into Rust—a language whose growing industrial adoption is driven by its strong guarantees of memory safety and high performance. In recent years, Rust has emerged as a popular target for automated code translation, making it a particularly relevant setting for understanding the strengths and limitations of current approaches.

2 Rule-Based Code Translation

Automated code translation has been a topic of extensive research for decades [6, 20]. In programming languages research, this process is often referred to as "transpilation". Unlike traditional compilers, which convert code from a high-level programming language to a lower-level one, transpilers bridge programming languages that operate at a similar level of abstraction. Throughout this paper, we use the term "translation" to refer to this form of transpilation.

Traditionally, code translation techniques were *rule-based*, which work by applying handcrafted rewrite rules to the source code or some representation of

it such as the Abstract Syntax Tree in order to transform elements of the source language into the target language [5,13,22,29]. One advantage of this approach is its predictability and stability, as identical inputs always yield the same results, as well as the strong correctness guarantees when the rules are rigorously defined and verified. The most significant drawback of rule-based translation is that it generally produces code that is unreadable and unidiomatic, making it challenging to maintain [11,24]. This approach often mirrors the structure of the original code too closely, failing to fully leverage the features and idiomatic patterns of the target language.

While rule-based techniques can effectively scale to entire projects—with notable examples targeting C to Rust [2,9,38], C to Go [1], and Java to C# [3]—developing these rules is a highly labor-intensive process and may fall short of being exhaustive. The sheer number of translation rules required to handle all cases across all involved codebases, including libraries, makes comprehensive coverage a significant challenge [31]. Moreover, rule-based translation systems typically operate with a local view of the code: individual rewrite rules are triggered by syntactic patterns or local AST fragments. However, high-quality translations often require global semantic information—how data flows through the program, how pointers relate, or how abstractions are used across modules. Without such context, these systems tend to replicate low-level constructs from the source language rather than recovering appropriate abstractions in the target language.

A prime example of a rule-based translator is C2Rust [2], an industrial-strength tool that rewrites C programs into Rust syntax while preserving semantics. Because its translation rules operate locally and do not synthesise an ownership model, C pointers are mapped directly to raw Rust pointers. This in turn produces large regions of code that must be marked with unsafe, , signalling that the compiler cannot enforce Rust's safety guarantees and shifting the burden of correctness back onto the programmer. Although the resulting Rust compiles, it preserves C's pointer-centric design rather than Rust's ownership-oriented patterns. The generated code is therefore harder to read, reason about and maintain, and fails to leverage Rust's idiomatic abstractions such as Box, Option, borrowing or pattern matching—ultimately undermining many of the benefits motivating migration to Rust in the first place.

Subsequent work attempts to post-process such translations to recover safer idioms. Laertes [10] explores the space of candidate rewrites, guided by Rust compiler error messages, in order to eliminate unnecessary unsafe blocks. While effective in targeted cases, its search remains brittle and leaves many unsafe constructs unresolved. CROWN [38] addresses this limitation by placing ownership at the core of the translation process. Starting from the C2Rust output, it performs static analyses over Rust MIR to infer a pointer ownership model compatible with Rust's semantics, and then rewrites pointers using safe abstractions (Box<T>, &mut T, etc.). This ownership-guided pipeline enables the translation of real-world C codebases (up to hundreds of thousands of lines of code) into significantly safer and more idiomatic Rust. However, CROWN still struggles when encountering patterns not covered by its predefined rewrite rules, often

producing code that still fails to provide meaningful safety improvements over the original C.

Beyond these approaches that target unsafe pointers, Hong and Ryu examine other specific pain points in the C2Rust translation and redesign them using Rust-appropriate abstractions. They propose automated rewrites for C lock APIs [15], translation of tagged unions [18], and the elimination of output-parameter patterns in favour of algebraic data types [17]. These studies highlight the benefits of domain-specific refactorings, but each targets a narrow class of features, leaving many others untreated. As a result, the overall translation still diverges significantly from what an expert Rust developer would naturally write.

3 Neural-Based Code Translation

Neural-based code translation offers substantial advantages over rule-based methods in terms of idiomaticity. Neural models can produce code that better reflects human-written idioms and, unlike handcrafted rewrite systems, are typically not bound to a single language pair [11,27]. A seminal line of work by Rozière et al. demonstrated that unsupervised neural models can translate functions across C++, Java, and Python without paired training data [30,31], showing that neural representations capture cross-language structure and patterns often invisible to purely syntactic rewriting.

However, these early approaches were largely evaluated on small, self-contained snippets—typically from GeeksforGeeks [14], competitive programming datasets [23,28], or educational corpora [4,34]. Such settings privilege short, isolated problems and fail to reflect the architectural dependencies, implicit invariants and cross-module interactions characteristic of real software systems. Although subsequent works introduce improved training objectives, prompt strategies or post-generation repair [21,33,36], they largely remain confined to this small-program paradigm.

As a result, recent progress has been driven by large language models (LLMs), which have become the de-facto foundation for neural translation [8,21,33,35,36]. A prominent line of work combines LLMs with static program analysis [12,21,35,36]. In C to Rust pipelines [26,39], scalability is typically achieved by first invoking C2Rust [2] and then using LLMs to "idiomatise" the output. Because these approaches inherit C2Rust's pointer-centric structure, they operate on syntactically translated, largely unsafe Rust. While the LLM is able to correct some of the unidiomatic parts, a large proportion is still present in the result.

For end-to-end LLM approaches without a rule-based front-end, Ibrahimzada et al. [19] propose a compositional pipeline for translating Java to Python, generating code module by module and validating local invariants. Shiraishi and Shinagawa [32] instead target C to Rust, using a context-aware segmentation strategy to partition repositories prior to translation. While both approaches scale to entire projects, their translations largely fail to preserve the behaviour of the original code, highlighting the difficulty of full-project translation with LLMs alone. For translating Go to Rust, Zhang et al. introduce a neurosymbolic

pipeline implemented in Oxidizer [37]. Oxidizer couples LLM-based generation with static mapping of constructs between the source and target languages, type-compatibility checks, and input–output semantic validation, demonstrating that repository-scale translation becomes practical when neural generation is grounded in domain-specific program analysis.

4 Current Limitations

Despite progress in rule-based, neural and neurosymbolic pipelines, current systems still fall short of the requirements for real-world translation. The remainder of this section examines several of the most persistent barriers.

Concurrency. Translating concurrent code remains highly challenging. Programming languages often embody fundamentally different concurrency paradigms—e.g. shared-memory threads, message passing, async/await—which makes rule-based translation unfeasible in the general case. Existing work has so far only tackled narrow subproblems; for instance, Hong and Ryu's Concrat system automatically rewrites C lock APIs to Rust equivalents, but is restricted to lock-based synchronization and requires accurate static summaries of lock behaviour to succeed [16]. LLM-based approaches do not solve this: concurrency invariants are hard to specify and even harder to verify, and neural-based translation may introduce subtle race conditions or deadlocks. Meaningful progress likely depends on advances in scalable program verification for concurrent systems.

Third-Party Dependencies. Automated translation also struggles when source code depends on rich ecosystems of libraries. Rule-based systems can only cover APIs they explicitly know how to rewrite, while neural approaches often hallucinate import paths and APIs, or generate syntactically correct but semantically incompatible replacements. Moreover, target languages may lack direct equivalents for libraries used in the source project. Bridging this gap requires explicit modeling of dependency ecosystems, for example through API mapping databases or retrieval-augmented approaches to API translation.

Lack of Formal Verification in Neural-Based Translation. Most current neural or neurosymbolic translation pipelines ultimately validate the correctness of the generated code using input–output examples. These checks provide only partial assurance: they can detect obvious regressions, but cannot guarantee preservation of behaviour across the (potentially infinite) input space or through complex execution paths. This limitation is particularly acute for repository-level translation, where small errors in data structures or pointer manipulation can remain latent until deployment. Formal verification offers a more principled alternative, but so far has only seen early adoption. One notable example is VERT [35], which combines LLM-generated Rust with an oracle Rust program compiled from WebAssembly, and then performs property-based testing and bounded model checking to verify semantic equivalence. While VERT demonstrates that formal

guarantees are feasible, it remains constrained: verification must be performed function-by-function, requires bounds on loops, and faces scalability limits for real-world codebases. Meaningful progress will likely require advances in scalable software verification, including automated invariant inference and robust verification techniques that operate across language boundaries.

5 Conclusions

Automated translation of real-world codebases remains challenging. Rule-based approaches scale and preserve semantics, but typically produce unidiomatic code that is hard to read and maintain. Neural approaches generate idiomatic code, yet remain brittle and opaque. Neurosymbolic pipelines offer a pragmatic middle ground, combining analysis-driven guidance with validation, but still struggle with external dependencies, concurrency abstractions and the absence of formal guarantees. Addressing these challenges will require hybrid methods that integrate language-specific reasoning, richer specification mechanisms and verification that goes beyond input–output testing to provide trustworthy translations at repository scale.

References

1. C to Go translator. https://github.com/gotranspile/cxgo. Accessed 16 Dec 2024
2. C2Rust transpiler. https://c2rust.com/. Accessed 16 Dec 2024
3. Sharpen - automated Java-C# coversion. https://github.com/mono/sharpen. Accessed 16 Dec 2024
4. Ahmad, W.U., Tushar, M.G.R., Chakraborty, S., Chang, K.: AVATAR: a parallel corpus for java-python program translation. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (eds.) Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9–14, 2023, pp. 2268–2281. Association for Computational Linguistics (2023). https://doi.org/10.18653/V1/2023.FINDINGS-ACL.143
5. Babel is a javascript compiler. https://babeljs.io/. Accessed 09 Dec 2024
6. Bastidas Fuertes, A., Pérez, M., Meza Hormaza, J.: Transpilers: a systematic mapping review of their usage in research and industry. Appl. Sci. **13**(6) (2023). https://doi.org/10.3390/app13063667, https://www.mdpi.com/2076-3417/13/6/3667
7. Chromium: Chromium security - memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed 20 Nov 2024
8. Di, P., et al.: CodeFuse-13B: a pretrained multi-lingual code large language model. In: International Conference on Software Engineering: Software Engineering in Practice, p. 418–429. ICSE-SEIP '24, ACM (2024). https://doi.org/10.1145/3639477.3639719
9. Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating C to safer rust. Proc. ACM Program. Lang. **5**(OOPSLA), 1–29 (2021)
10. Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating c to safer rust. Proc. ACM Program. Lang. 5(OOPSLA) (2021). https://doi.org/10.1145/3485498
11. Eniser, H.F., et al.: Towards translating real-world code with LLMs: a study of translating to Rust (2024). https://arxiv.org/abs/2405.11514
12. Farrukh, M., Shah, S., Coskun, B., Polychronakis, M.: Safetrans: LLM-assisted transpilation from C to rust (2025). https://arxiv.org/abs/2505.10708

13. Foundation, T.P.S.: 2 to 3 – automated python 2 to 3 code translation. https://docs.python.org/3.12/library/2to3.html. Accessed 09 Dec 2024
14. GeeksforGeeks: a computer science portal for geeks. https://www.geeksforgeeks.org/. Accessed 10 Dec 2024
15. Hong, J., Ryu, S.: Concrat: An automatic C-to-rust lock API translator for concurrent programs. In: Proceedings of the 45th International Conference on Software Engineering (ICSE), pp. 716–728. IEEE (2023). https://doi.org/10.1109/ICSE48619.2023.00069
16. Hong, J., Ryu, S.: Concrat: an automatic C-to-rust lock API translator for concurrent programs. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, pp. 716–728. IEEE (2023). https://doi.org/10.1109/ICSE48619.2023.00069
17. Hong, J., Ryu, S.: Don't write, but return: replacing output parameters with algebraic data types in C-to-rust translation. Proc. ACM Program. Lang. **8**(PLDI) (2024). https://doi.org/10.1145/3656406
18. Hong, J., Ryu, S.: To tag, or not to tag: translating C's unions to rust's tagged unions. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 40–52. ACM (2024). https://doi.org/10.1145/3691620.3694985
19. Ibrahimzada, A.R., et al.: Repository-level compositional code translation and validation (2024). https://arxiv.org/abs/2410.24117
20. Intel: MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users. Technical report, Intel (1978). Accessed 11 Dec 2024
21. Jana, P., Jha, P., Ju, H., Kishore, G., Mahajan, A., Ganesh, V.: Attention, compilation, and solver-based symbolic analysis are all you need (2023). arXiv preprint arXiv:2306.06755
22. Kimura, K., Sekiguchi, A., Choudhary, S., Uehara, T.: A javascript transpiler for escaping from complicated usage of cloud services and APIs. In: 25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, pp. 69–78. IEEE (2018). https://doi.org/10.1109/APSEC.2018.00021
23. Lu, S., et al.: CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. In: Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks. vol. 1 (2021). https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c16a5320fa475530d9583c34fd356ef5-Paper-round1.pdf
24. Malyala, A., Zhou, K., Ray, B., Chakraborty, S.: On ML-based program translation: perils and promises. In: 45th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE, Melbourne, Australia, pp. 60–65. IEEE (2023). https://doi.org/10.1109/ICSE-NIER58687.2023.00017
25. Microsoft: a proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (2019). Accessed 20 Nov 2024
26. Nitin, V., Krishna, R., do Valle, L.L., Ray, B.: C2saferrust: transforming C projects into safer rust with neurosymbolic techniques (2025). https://arxiv.org/abs/2501.14257
27. Pan, R., et al.: Lost in translation: a study of bugs introduced by large language models while translating code. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024, pp. 82:1–82:13. ACM (2024). https://doi.org/10.1145/3597503.3639226

28. Puri, R., et al.: CodeNet: a large-scale AI for code dataset for learning a diversity of coding tasks (2021). arXiv preprint arXiv:2105.12655
29. Radoi, C., Fink, S.J., Rabbah, R.M., Sridharan, M.: Translating imperative code to mapreduce. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages nd Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20–24, 2014, pp. 909–927. ACM (2014). https://doi.org/10.1145/2660193.2660228
30. Rozière, B., Lachaux, M., Chanussot, L., Lample, G.: Unsupervised translation of programming languages. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual (2020). https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html
31. Rozière, B., Zhang, J., Charton, F., Harman, M., Synnaeve, G., Lample, G.: Leveraging automated unit tests for unsupervised code translation. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25–29, 2022. OpenReview.net (2022). https://openreview.net/forum?id=cmt-6KtR4c4
32. Shiraishi, M., Shinagawa, T.: Context-aware code segmentation for C-to-rust translation using large language models (2024). https://arxiv.org/abs/2409.10506
33. Tang, Z., et al.: Explain-then-translate: an analysis on improving program translation with self-generated explanations. In: Findings of the Association for Computational Linguistics: EMNLP 2023. pp. 1741–1788. Association for Computational Linguistics (2023). https://doi.org/10.18653/v1/2023.findings-emnlp.119
34. Yan, W., Tian, Y., Li, Y., Chen, Q., Wang, W.: CodeTransOcean: a comprehensive multilingual benchmark for code translation (2023). arXiv preprint arXiv:2310.04951
35. Yang, A.Z.H., Takashima, Y., Paulsen, B., Dodds, J., Kroening, D.: VERT: verified equivalent Rust transpilation with large language models as few-shot learners (2024). https://arxiv.org/abs/2404.18852
36. Yin, X., Ni, C., Nguyen, T.N., Wang, S., Yang, X.: Rectifier: code translation with corrector via LLMs. CoRR abs/2407.07472 (2024). https://doi.org/10.48550/ARXIV.2407.07472
37. Zhang, H., David, C., Wang, M., Paulsen, B., Kroening, D.: Scalable, validated code translation of entire projects using large language models (under submission) (2024). https://arxiv.org/abs/2412.08035
38. Zhang, H., David, C., Yu, Y., Wang, M.: Ownership guided C to rust translation. In: Enea, C., Lal, A. (eds) Computer Aided Verification (CAV). LNCS, vol. 13966, pp. 459–482. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_22
39. Zhou, T., Lin, H., Jha, S., Christodorescu, M., Levchenko, K., Chandrasekaran, V.: LLM-driven multi-step translation from C to rust using static analysis (2025). https://arxiv.org/abs/2503.12511