Automated Code Translation using an Engineering-Driven Pipeline Approach of Language Models

Tibo Bruneel[1,2,*], Welf Löwe[1,2], Morgan Ericsson[2], Diego Perez-Palacin[2] and Jonas Nordqvist[3]

[1]*Softwerk AB, Reveljgränd 5 - Växjö, Sweden*
[2]*Dept. of Computer Science and Media Technology, Linnaeus University, Universitetsplatsen 1 - Växjö, Sweden*
[3]*Dept. of Mathematics, Linnaeus University, Universitetsplatsen 1 - Växjö, Sweden*

Abstract

This paper presents an engineering-driven approach to pipelining large language models (LLMs) for automatic code translation. The method is applied in an ongoing industrial project with Danfoss Power Solutions aimed at translating five million lines of their code from Delphi to C#. To manage this scale, the codebase is first divided into chunks, each processed independently through the LLM pipeline and subsequently reassembled. Due to the inherent stochasticity of LLM outputs, the pipeline incorporates processing and validation functions to ensure consistency and correctness. While still in its early stages, this approach demonstrates promising results. With this publication, we share our initial LLM agents orchestration with an initial draft pipeline used in the industry case, with insights to inform future development and application in similar large-scale translation tasks.

Keywords

Code Translation, Large Language Models, MLOps, LLMOps, LLM Pipeline, Pipeline Orchestration

1. Introduction

Automatic code translation has become a crucial area of research and development, driven by the continued reliance on legacy programming languages in critical systems. Many of these systems, often written in languages like COBOL, Fortran, or Delphi, continue to power industries such as finance, government, and manufacturing, yet suffer from maintainability issues, a lack of developers, and incompatibility with more modern software ecosystems. While manual rewriting of legacy code is possible, it comes with a significant cost and time investment; for larger projects, it even requires maintenance during the translation itself. In contrast, automated translation offers a more scalable way to modernise codebases, minimising time and costs, and thereby supporting long-term sustainability.

The pipeline approach presented in this study is tied to an industrial case with Danfoss Power Solutions, involving a large codebase of approximately 5 million lines of code (MLOC) written in Delphi. As Delphi becomes increasingly niche and less commonly used, system maintenance becomes more difficult and costly. The objective of the project is to reimplement the full code base in C# using .NET. A manual translation of this is estimated to be an effort of 3700 person-months of work, a massive effort prompting for an automated solution. In a pre-study [1] of this project, two approaches were evaluated, the first being a Delphi-to-C# transpiler, and the second being LLM translation. The study showed that the LLM-track would be faster, but require more manual interaction. But the LLM approach would also be better suited for future challenges such as handling other programming languages, test generation, and code optimisation. Consequently, the LLM-based method was chosen for the automatic translation.

While general-purpose LLMs are capable of performing a wide range of tasks, they often trade depth for breadth and remain too general. In contrast, a collection of specialised expert models, each fine-tuned or prompt-engineered (instructions) for specific task completion, can outperform a single general model. This also applies to code translation, where specialised agent models can be tuned for specific

MLOps25: Workshop on Machine Learning Operations. October 25, 2025, Bologna, Italy
*Corresponding author.
tibo.bruneel@softwerk.se (T. Bruneel); welf.lowe@lnu.se (W. Löwe); morgan.ericsson@lnu.se (M. Ericsson); diego.perez@lnu.se (D. Perez-Palacin); jonas.nordqvist@lnu.se (J. Nordqvist)
0009-0000-4370-5981 (T. Bruneel); 0000-0002-7565-3714 (W. Löwe); 0000-0003-1173-5187 (M. Ericsson); 0000-0002-2736-845X (D. Perez-Palacin); 0000-0002-0510-6782 (J. Nordqvist)

subtasks (*e.g.* code translation, test generation, compiler error correction, and parser warning resolution). However, using a set of expert models instead of a single model requires careful orchestration through pipelining to ensure seamless task coordination and consistency.

Agentic orchestration has seen an uprise with the emergence of LLMs and their potential for automated task completion, with, for example, Google's Agent2Agent (A2A) protocol [2, 3] for agent-to-agent communication. This has also led to a rise in collaborative agent frameworks for code generation and translation. Specialised systems such as TRANSAGENT [4] introduce multiple agents for translation and debugging with automatic repair, while UniTrans [5] and AgentCoder [6] introduce structured, test-driven code translation to ensure functional correctness and equivalence. The trend of collaborative agents within code translation is particularly evident in Automatic Program Repair (APR), with a spectrum of approaches such as FixAgent [7] using a team of debugging agents, LANTERN [8] using cross-language multi-agent orchestration, and single-agent APR with RepairAgent [9]. This agent-based paradigm extends to the simulating of entire software development lifecycles and teams, such as MetaGPT [10] and ChatDev [11]. Existing frameworks demonstrate a wide range of capabilities, from specific applications like code translation and APR to complete agent-based software development lifecycles. Despite this progress, the field is still emerging, and many challenges remain. Applying LLM agents in a collaborative setting effectively to large-scale codebases and complex industrial projects also represents a critical next step for this research area.

We present our design and implementation of an orchestrated multi-agent pipeline using a specific agent stage structure, developed at the outset of our translation project. To the best of our knowledge, the notions introduced herein are novel, but are very much based on engineering-driven concepts as automatic code translation highly resembles the process of general code translation. Our goal is to contribute initial insights and practical methods to both academic research and industrial efforts focused on large-scale code modernisation. In Section 2, code chunks are discussed with chunking and assembly strategies. In Section 3 the pipeline orchestration is introduced with agent stages and their dynamic orchestration. In Section 4 an initial draft of the translation pipeline is provided with discussion, with a final conclusion in Section 5.

2. Code Chunks

Large language models have finite context windows, meaning the models are limited to predefined maximum token sizes for a single pass. The context windows vary for different models, *e.g.* GPT-4o supports up to 128K tokens [12], Gemini 1.5 Pro up to 2 million tokens [13], and LLama 4 Scout up to 10 million tokens [14]. Additionally, closed-source models accessible through APIs may impose further constraints with maximum output token limits, such as GPT-4o's 16K output tokens limit [12].

Chunking refers to the process of splitting LLM inputs into smaller, manageable pieces that fit within model context windows, allowing them to be processed without exceeding maximum tokens. Inputs can be divided into chunks, passed through an LLM pipeline individually, and reassembled afterward. The same holds for LLM code translation, where entire codebases cannot be translated in a single model pass. Code translation requires thoughtful chunking so that the code can be correctly translated from source to target. In this context, chunking involves dividing the codebase into smaller, coherent units that represent semantically or syntactically meaningful blocks, *e.g.*, methods, classes, and objects.

While recent models support increasingly large context windows, it is not always beneficial to maximise the usage of a context window. Processing large inputs as a single chunk can lead to a phenomenon known as context dilution, where the relevance of specific information becomes diminished within the broader context. Liu et al. [15] demonstrated the dilution effect of context prioritisation over long context lengths. This occurs because LLMs distribute attention across the entire input, where important and small details may be overshadowed by unrelated or less relevant content. This can especially have a large effect on automatic code translation, potentially leading to side effects, incorrect behaviour, or missing functionality in the translated code. Moreover, when both context window and output token limits apply, it is critical to account for the token expansion factor, the difference in token

counts between source and target languages. For instance, if translated code typically doubles in token count, chunk sizes must be adjusted accordingly to avoid truncated outputs upon model inference.

The general outline of code translation from a codebase in a source language to a codebase in a target language exists out of the following steps, as also illustrated in the diagram in Figure 1:

1. **Chunking**: For each file in the codebase, chunks are obtained using the chunking strategy. Certain files may be omitted if not relevant in the target language.
2. **Translation**: The pipeline translates each chunk individually, and possibly concurrently.
3. **Assembly**: Translated chunks are reassembled using the assembly strategy.

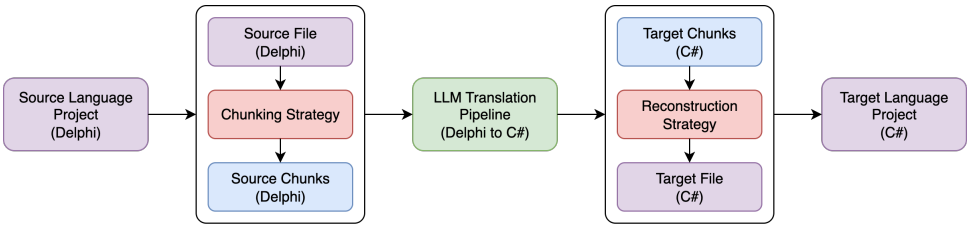

Figure 1: Diagram of a general sequence of translating a code project to a target language, using the chunking strategy, code translation, and reconstruction strategy.

2.1. Chunking Strategy

A parser, in this project for the Delphi language, is used to extract the abstract syntax tree and code elements from the source code. Based on this parsed representation, two distinct chunking strategies are applied. For larger structural elements such as classes and methods, a depth-first traversal is used to isolate each complete syntactic unit, which is then treated as an individual chunk for translation. On the other hand, smaller code elements, such as simpler variable declarations or type definitions, are grouped together in chunks up to a certain maximum token limit.

2.2. Assembly Strategy

During the translation process, a structural skeleton of all source files is maintained. This skeleton preserves all high-level elements with their signatures extracted from the abstract syntax tree in the chunking strategy, replacing their contents with placeholders. For example, a function might be represented in C# as a stub that raises a *NotImplementedException*. As each chunk is translated by the pipeline, its corresponding placeholder in the skeleton signature is replaced with the generated target-language code. This strategy enables the reassembly of the translated code in a way that closely mirrors the original source structure.

3. Pipeline

In this section, we first introduce the concept of an agent stage in Subsection 3.1, followed by dynamic pipelines for communicating between these agent stages in Subsection 3.2.

3.1. Agent Stage

An agent stage can be viewed as a structured wrapper built around a specific agent model. Each agent and its corresponding stage are tailored to a distinct subtask, given specific knowledge and expertise *e.g.* code documentation. In addition to the agent itself, the agent stage encapsulates a sequence of processing and validation functions. It takes in text as input and returns text as output, as a generative LLM itself does. Agent stages are executed with given input sequentially through pre-processing, pre-validation, agent-inference, post-processing and finally post-validation.

Each processing or validation step may include multiple such functions, or may be omitted entirely depending on the stage requirements, especially the processing functions may be omitted. Every agent stage is uniquely identified by an agent stage card (similar to model cards) which contains metadata *e.g.* description, and versioning, and also includes a unique ID used for communication between stages during pipeline orchestration. The sequential flow of an agent stage is illustrated in Figure 2.

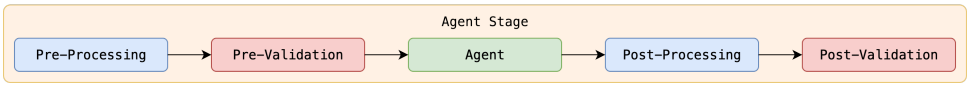

Figure 2: Diagram of a single agent stage visualising the sequence of pre-processing, pre-validation, agent inference, post-processing, and post-validation.

3.1.1. Processing and Validation Functions

Processing functions handle transformations on model input or output. They receive text as input and return modified text as output. These are applied before or after inference, depending on their role as pre- or post-processing functions. Examples include prompt restructuring, code formatting, or deterministic code modifications.

Validation functions on the other hand are responsible for verifying the correctness or suitability of model input or output. These functions take text as input and return a validation result, typically a boolean outcome with accompanying textual feedback. These functions are executed after processing steps. Examples of possible validation functions include compilation output, linting, or test runners.

3.1.2. Agent Inference

Agent inference refers to the direct invocation of the LLM model. This agent can be any model instructed for the specific task, such as code translation, test generation, or other forms of structured output.

While the pipeline is primarily designed for text generation tasks, agent inference is not limited to generative models. Other types of models, such as text classifiers or controller agents, can also be integrated, but would require minor changes and more complex validation logic. For example, a controller agent could dynamically select the next best-suited agent (stage) for the task, leveraging the orchestration mechanism. This flexibility allows the pipeline to support complex, adaptive workflows beyond text or code generation as in this project.

3.2. Dynamic Pipeline

A dynamic pipeline architecture enables multiple agent stages to communicate and collaborate in solving complex tasks. Each stage performs a specific role and passes its output to the next stage. Importantly, the next stage can be decided based on the model's output in the current stage, *i.e.*, based on the agent's performance in completing a task. Thereby enabling more modular and adaptive workflows. This pipeline architecture introduces two mechanisms for dynamic communication between stages:

1. **Stage Completion Routing:** Each stage can receive an *onCompletionNextStage* ID. When a stage completes its task, the output is sent as input to the next agent stage identified by this ID. If no identifier is set and execution completes, the pipeline assumes it was the final stage and finishes.
2. **Validation Failure Routing:** Each validation function in both pre- and post-validation steps can optionally receive an *onValidationFailureStage* ID. If a validation function fails and this identifier is set, the pipeline immediately redirects the output to that specific stage instead, potentially for resolving the issue flagged in validation. Afterward, it may either return to this agent stage, or follow an alternative route of stages. If no failure stage identifier is defined and a validation function fails, the pipeline may instead terminate. If the validation passes, the process just continues with the next validation check, the subsequent step within the current stage, or the next stage in the pipeline.

A general architecture of a dynamic pipeline is illustrated in Figure 3, where each input chunk sequentially passes through the stages, with conditional re-routing upon failure, and producing an output chunk in the end, *e.g.* stage 1 is connected to stage 2 through completion, but reroutes to stage 3 on validation failure.

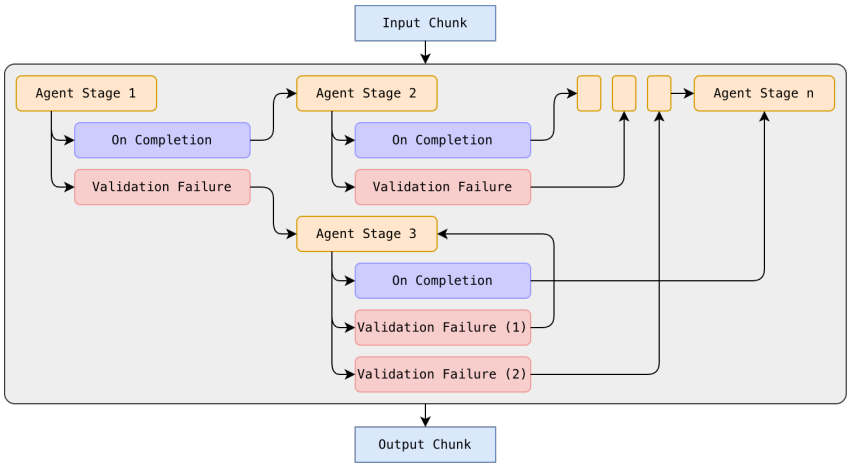

Figure 3: Diagram of a dynamic pipeline illustrating the orchestration of multiple agent stages.

4. Discussion

Having introduced the chunking and assembly strategies, as well as the dynamic pipeline with agent stages, we now provide an initial draft of the translation pipeline for Delphi to C# translation. Figure 4 illustrates an early-stage design for such a pipeline, comprising the following agent stages:

- **Delphi to C# Translation Stage**: Agent stage for translating Delphi chunks to C# chunks.
- **Delphi Test Generation Stage**: Agent stage for generating tests based on the Delphi chunk.
- **Delphi Debugger Stage**: Agent stage for resolving errors/warnings in Delphi code.
- **C# Test Generation Stage**: Agent stage for generating tests based on the translated C# chunk and the generated Delphi test code.
- **C# Debugger Stage**: Agent stage for resolving errors/warnings in C# code.

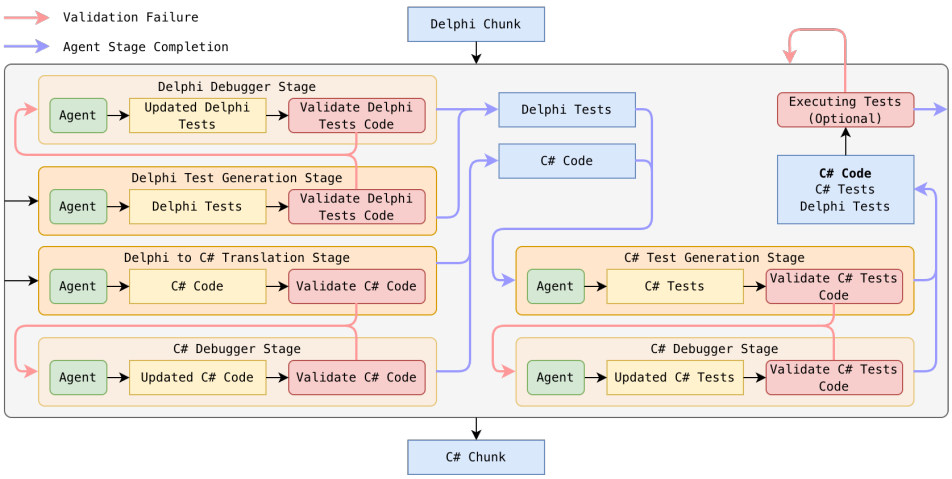

Figure 4: Diagram of an initial draft pipeline architecture for code translation, visualising several agent stages communicating dynamically based on validation results on generated code and tests.

The objective of this pipeline is not only to translate Delphi code into C#, but also to automatically generate test code in both languages. Specifically, Delphi tests are generated for the original code chunks, and C# tests are subsequently derived from both the generated C# code and Delphi tests, with the goal of generating the same tests in both language, allowing them to be tested for the same cases, as otherwise the chunks would end up with different and deviating test cases in both languages. The generated tests can be executed and checked when the chunk contains all required context and does not depend on other chunks being translated, *e.g.* depending on global variables or out-of-scope functions. If the chunk requires other to be translated context, the tests can be run post-assembly to validate code.

Post-validation functions are applied after model inference to ensure correctness. These functions validate both the code and test generations. The validation functions can contain tools such as linters, compilers, and static analysis, further validating the generated code correctness. The agents communicate dynamically through these post-validation checks, allowing the pipeline to adapt its path based on agent output. When no errors are detected, only three of the five stages are invoked. However, if the models struggle to translate more complex code, the debugger stages might be used, and one stage might be executed several times. For example, if the C# code coming from the code translation stage does not compile, the pipeline sends the output to the C# debugger stage to resolve the error. If the validation after this succeeds, it continues to test generation. If not, the error can be handled by the debugger stage again with more feedback. To avoid cyclic dependencies with infinite loops between two or more agent stages in the pipeline, cycles can be detected and either limited to a maximum number of iterations or avoided entirely. It is also important to note that the post-validation focus is on measuring code correctness, but not through immediate execution of code or tests; and that certain errors are automatically solved or can only be solved post-assembly.

The illustrated pipeline represents a preliminary implementation. As the pipeline matures, it will be modified and extended. This is done in an MLOps/LLMOps manner, where benchmarking with full translation runs are performed measuring large sets of metrics, using both quantative and qualitative evaluation to identify weaknesses in the agent performances. Based on these findings, new agents and further tuning can address and mitigate specific issues found in the translation process.

5. Conclusion

In this paper, we presented an engineering-driven pipeline approach for employing LLM agents collaboratively. Although the work is still in its early stages, we briefly discussed all its key components. To illustrate the approach, we presented a preliminary Delphi-to-C# translation pipeline.

Initial results of the concepts are promising, where practical experiences confirm that specialised agents in a collaborative setting, each carefully prompt engineered, consistently outperform single general model passes. Delegation of tasks across distinct specialised stages therefore enhances translation accuracy and translation explainability, providing more control of specific pipeline steps. Practical implementation of the pipeline has revealed that translation accuracy depends on iterative improvements and prompt engineering. In general, test code generation has been found to be more challenging compared to code translation, but it is a crucial part of translation validation, while debugging stages play a vital part in reducing human intervention. However, the tuning and iterative improvements are time-consuming and resource-intensive, as each change needs to be benchmarked; the expectation is that that effort will swiftly diminish as the pipelines mature. Validation of LLM-generated code remains a large step in research and industrial applications due to the inherent stochasticity in models. With our approach, we want to promote and uphold strong generation validation at all steps, flagging issues early on during the translation process and setting more steps towards fully automated translation.

While initial experiments and results are promising, further work is needed to evaluate the scalability, error handling, and adaptability of the approach. We will focus on enhancing agent coordination, evaluation metrics, and benchmarking the efficiency gains this approach offers compared to manual developer effort, as the dynamic pipeline matures over iterative improvements.

References

[1] S. Crucerescu, I. Falk, Evaluating accuracy and development effort of code translated with transpilers and LLMs: Focusing on Delphi to C#, Bachelor's Thesis, Linnaeus University, 2024.

[2] Google, Agent2Agent (A2A) Protocol, https://github.com/google/A2A, 2025. Accessed: 2025-05-12.

[3] I. Habler, K. Huang, V. S. Narajala, P. Kulkarni, Building A Secure Agentic AI Application Leveraging A2A Protocol, 2025. URL: https://arxiv.org/abs/2504.16902. arXiv:2504.16902.

[4] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, Y. Lou, TRANSAGENT: An LLM-Based Multi-Agent System for Code Translation, 2024. URL: https://arxiv.org/abs/2409.19894. arXiv:2409.19894.

[5] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, G. Li, Exploring and Unleashing the Power of Large Language Models in Automated Code Translation, 2024. URL: https://arxiv.org/abs/2404.14646. arXiv:2404.14646.

[6] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, H. Cui, AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation, 2024. URL: https://arxiv.org/abs/2312.13010. arXiv:2312.13010.

[7] C. Lee, C. S. Xia, L. Yang, J. tse Huang, Z. Zhu, L. Zhang, M. R. Lyu, A Unified Debugging Approach via LLM-Based Multi-Agent Synergy, 2024. URL: https://arxiv.org/abs/2404.17153. arXiv:2404.17153.

[8] W. Luo, J. W. Keung, B. Yang, J. Klein, T. F. Bissyande, H. Tian, B. Le, Unlocking LLM Repair Capabilities in Low-Resource Programming Languages Through Cross-Language Translation and Multi-Agent Refinement, 2025. URL: https://arxiv.org/abs/2503.22512. arXiv:2503.22512.

[9] I. Bouzenia, P. Devanbu, M. Pradel, RepairAgent: An Autonomous, LLM-Based Agent for Program Repair, 2024. URL: https://arxiv.org/abs/2403.17134. arXiv:2403.17134.

[10] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, J. Schmidhuber, MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework, 2024. URL: https://arxiv.org/abs/2308.00352. arXiv:2308.00352.

[11] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, M. Sun, ChatDev: Communicative Agents for Software Development, 2024. URL: https://arxiv.org/abs/2307.07924. arXiv:2307.07924.

[12] OpenAI, GPT-4o Platform Docs, https://platform.openai.com/docs/models/gpt-4o, 2025. Accessed: 2025-05-09.

[13] Google-DeepMind, New Features for the Gemini API and Google AI Studio, https://developers.googleblog.com/en/new-features-for-the-gemini-api-and-google-ai-studio/, 2025. Accessed: 2025-05-09.

[14] MetaAI, LLaMA 4 and the Scout Model, https://www.llama.com/models/llama-4/, 2025. Accessed: 2025-05-09.

[15] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang, Lost in the Middle: How Language Models Use Long Contexts, 2023. URL: https://arxiv.org/abs/2307.03172. arXiv:2307.03172.

Acknowledgments

Tibo Bruneel's work was funded by the Industry Graduate School on "Data Intensive Applications (DIA)" at Linnaeus University, which is partially funded by the Knowledge Foundation. Additional funding for the research was provided by Softwerk AB and Danfoss Power Solutions. The authors would like to thank Danfoss Power Solutions for their valuable collaboration, continued commitment to the industrial project that forms the basis of this work, and their active support for open research.

Declaration on Generative AI

The authors have not employed any Generative AI tools.