Future Generation Computer Systems 184 (2026) 108599

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A modular multi-agent pipeline framework for large-scale code translation ☆

Tibo Bruneel [a,b,*], Sandu Cruceresc u [a], Welf Löwe [a,b], Morgan Ericsson [b], Diego Perez-Palacin [b], Jonas Nordqvist [c]

[a] Softwerk AB, Reveljgränd 5, Växjö, 35236, Sweden
[b] Dept. of Computer Science and Media Technology, Linnaeus University, Universitetsplatsen 1, Växjö, 35252, Sweden
[c] Dept. of Mathematics, Linnaeus University, Universitetsplatsen 1, Växjö, 35252, Sweden

ARTICLE INFO

Keywords:
Code translation
Large Language Models
MLOps
LLMOps
LLM pipeline
Pipeline orchestration

ABSTRACT

This publication presents an established engineering-driven framework for pipelining large language model (LLM) agents for automatic code translation. The framework is applied in a long-term ongoing industrial project with Danfoss Power Solutions aimed at translating five million lines of Delphi to C#. To manage this scale, the source codebase is divided into translatable chunks, which are processed independently through an agentic LLM pipeline, with subsequently the reassembly of the translated code chunks in the target language. Due to the inherent stochasticity of LLMs, the framework incorporates processing and validation functions to ensure consistency and correctness. As an extended version of our short paper (Bruneel et al., 2025), this paper provides a significantly more comprehensive architectural blueprint. Specifically, we present an in-depth background & related work section, formalise the chunking and reassembly strategies, detail the internal mechanics of agent stages, and introduce sequential agent orchestration alongside the dynamic LLM pipeline architecture. Furthermore, we report early empirical observations regarding the pipeline's computational overhead, validation dynamics, and translation quality, noting a practical impact of an initial 10x acceleration over estimated manual translation efforts. Ultimately, we share these expanded insights to inform future development and application in similar large-scale translation tasks.

1. Introduction

Automatic code translation has become an important 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 such as COBOL, Fortran, or Delphi, continue to power finance, governments, and manufacturing but suffer from maintainability issues, a shortage of developers, and incompatibility with modern software ecosystems. While manual rewriting of legacy code is possible, it comes with significant costs and time investment; for larger projects, it even requires maintenance during translation. 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 and involves a large codebase of approximately five 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 is to reimplement the full codebase in C# using .NET. A manual translation of this is estimated to require 3700 person-months of work, given an average monthly pace of 1000 lines of code per developer, as per the company's estimate, highlighting the strong need for an automated solution. In a pre-study [1] of this project, two approaches were evaluated: the first, a Delphi-to-C# transpiler; and the second, LLM translation. The study showed that the LLM track would be faster but would require more manual interaction. Furthermore, the LLM approach would also be better suited for future challenges such as handling other programming languages, test generation, documentation 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. In the context of code translation, specialised agent models can be tuned for specific subtasks (*e.g.*, code

☆ This article is part of a Special issue entitled: 'MLOps' published in Future Generation Computer Systems.

* Corresponding author at: Softwerk AB, Reveljgränd 5, Växjö, 35236, Sweden.
E-mail addresses: tibo.bruneel@softwerk.se (T. Bruneel), sandu.cruceresc u@softwerk.se (S. Cruceresc u), 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).

https://doi.org/10.1016/j.future.2026.108599
Received 30 March 2026; Received in revised form 7 May 2026; Accepted 11 May 2026
Available online 20 May 2026

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. We present the design and implementation of an orchestrated multi-agent pipeline using an engineering-driven agentic framework, developed at the outset of our translation project. To the best of our knowledge, the specific framework notions introduced herein are novel but are strongly grounded in existing literature and engineering-driven concepts, as automatic code translation closely resembles manual code translation. Our goal is to contribute initial insights, design, and practical methods to both academic research and industrial efforts focused on large-scale code modernisation.

Our contributions include:

- A systematic, engineering-driven framework for the design and orchestration of validatable multi-agent pipelines, demonstrated within the context of automatic code translation.
- A scalable, LLM-centric translation strategy that overcomes finite context windows and context dilution by integrating AST-driven source chunking, independent multi-agent LLM processing, and structural target reassembly.
- Two preliminary pipeline architectures (sequential and dynamic) tailored specifically for the Delphi-to-C# translation, currently being integrated into a five MLoC industrial modernisation project.
- Early empirical observations regarding the pipeline's computational overhead, validation dynamics, and translation quality, including insights into its practical impact with an initial 10× acceleration over estimated manual translation efforts.

The remainder of this paper is structured as follows. Section 2 provides the background and related work for this research. Section 3 details the general automatic code translation process, including the chunking and reassembly strategies to handle LLM context windows. Section 4 presents the LLM pipeline design to orchestrate agents using validation to verify LLM outputs. Section 5 presents the conceptual LLM pipelines designed at the outset of the industrial code translation project. Section 6 discusses the current implementation status and preliminary observations from the project including initial metrics and experiments. Finally, Section 7 concludes the paper and discusses future work.

2. Background & related work

To provide context, we review the evolution from single-agent systems to collaborative multi-agent orchestrations, with particular emphasis on their application to automatic code translation. We distinguish between foundational background and directly related work. Section 2.1 and the beginning of Section 2.2 provide background context, illustrating the shift from isolated agents to multi-agent orchestrations. In contrast, the latter half of Section 2.2 and the entirety of Section 2.3 form the direct related work. By examining state-of-the-art multi-agent software engineering frameworks and repository-scale translation approaches, these sections lay the methodological foundations for our execution-aware framework. Finally, in Section 2.4, we describe the extensions made in this publication from preliminary work.

2.1. Agentic LLMs

An autonomous LLM agent represents the evolution of Large Language Models (LLMs) from passive text generators to systems capable of complex problem-solving and goal-directed behaviour. Rather than merely producing a response, the LLM-based agent becomes an entity that receives a task, possibly interacts with its environment, and autonomously executes a sequence of actions until it reaches its objective. The conceptual foundation of this agentic paradigm lies in in-context learning [2] and instruction tuning [3], which enable LLMs to condition their behaviour on natural language task specifications without parameter updates. Building on these capabilities, structured prompting patterns [4] formalise role assignment and behavioural framing through system-level instructions (the system prompt), allowing the model to adopt task-specific personas and operational constraints during execution. Prompting techniques were further developed to influence internal reasoning, with Chain-of-Thought (CoT) [5] and Zero-Shot CoT [6]; these techniques demonstrate the generation of step-wise problem decomposition through reasoning traces, improving a model's ability to solve complex tasks. To overcome the limitations of static pre-trained weights and context, agents also need the ability to act within environments through tool use, as demonstrated in ToolFormer [7] and ToolLLM [8]. The well-established ReAct [9] framework unified these paradigms by interleaving reasoning traces and actions, significantly improving agent capabilities and reducing hallucinations. Achieving further autonomy, frameworks such as Reflexion [10] introduce self-correction mechanisms through verbal reinforcement learning, allowing agents to iteratively evaluate and improve outputs, without expensive weight updates.

Extending the capabilities to software engineering, models such as Codex [11] enabled natural language-to-code synthesis, while CodeBERT [12] advanced code understanding. Large-scale models pre-trained on extensive code corpora demonstrate strong generalisation and code generation capabilities. Moving beyond static generation, self-debugging frameworks [13] enable agents to execute generated code and iteratively repair errors using compiler feedback. Similarly, LDB [14] refines programs through runtime execution information, incorporating dynamic information into the repair process. More generally, Self-Refine [15] improves LLM outputs via iterative self-feedback and refinement, demonstrating that performance gains can emerge from structured feedback loops without additional training. This increasing degree of agent autonomy has enabled repository-level systems such as SWE-agent [16] and RepairAgent [17], which autonomously navigate and patch complex codebases, laying the groundwork for scalable multi-agent orchestrations required for larger and more complex software engineering tasks such as code translation.

2.2. Multi-agent orchestration

Multi-agent orchestration has seen a surge in interest with the emergence of LLMs and their potential for automated task completion, with a large shift from single-model to multi-agent paradigms to overcome inherent limitations and hallucination risks. He et al. [18] highlight how collaborative multi-agent systems significantly improve problem-solving capabilities, cross-validation robustness, and scalability across the complete software development lifecycle. By establishing foundational multi-agent interactions, CAMEL [19] demonstrated autonomous cooperation between agents, introducing a novel "role-playing" framework in which an "AI user" agent and an "AI assistant" agent collaborate to solve complex tasks without human intervention. Addressing more complex problem-solving through human-like group cooperation, frameworks such as AgentVerse [20] orchestrate a collaborative group of expert agents via a four-stage process, featuring a dynamic "Expert Recruitment" mechanism to autonomously adjust the group's composition based on evolving task needs.

Building on this team-based simulation, several agentic orchestration frameworks simulate entire software development lifecycles and teams: ChatDev [21] utilises a sequential chat chain among specialised roles, MetaGPT [22] enforces structured communication by encoding human-like Standardised Operating Procedures (SOP) into prompt sequences of specialised agents, and AgileCoder [23] integrates agile methodologies with a team of different role agents and dynamic dependency tracking to continuously refine complex generation tasks.

More specialised systems for code generation, translation, and Automatic Program Repair (APR) have also been introduced with the aim

of ensuring both syntactic and functional correctness and equivalence. Specialised systems such as TRANSAGENT [24] and UniTrans [25] introduce collaborative agentic LLMs tailored for translation, debugging, and test-driven development. Explicitly addressing incorrect code generation and self-optimisation, AgentCoder [26] decouples the testing process through a programmer, test designer, and test executor agent, while the Self-Organised Agents (SoA) [27] framework utilises the dynamic production of "Mother" and "Child" agents to scale codebase generation without exceeding context limits. The trend of collaborative agents in code translation is particularly evident in Automatic Program Repair (APR), with a spectrum of approaches, such as FixAgent [28], which uses a team of debugging agents, and LANTERN [29], which uses cross-language multi-agent orchestration. As they move towards industrial applications, open-source programming ecosystems have emerged to facilitate the engineering-driven pipelines required for large-scale tasks. These include frameworks such as LangChain [30], Agno [31], AutoGen [32], and CrewAI [33] for the rapid construction and development of different agentic paradigms, and communication protocols such as Google's Agent2Agent (A2A) protocol [34,35] for structured agent-to-agent communication.

2.3. Automatic code translation

Historically, automatic code translation has relied on transpilers, requiring extensive manual work on creating abstract syntax tree transformations, an approach that becomes exponentially more difficult when moving away from one-to-one translations. A significant paradigm shift is occurring with the surge in LLMs. These models have gained substantial interest due to their capacity for generalisation and complex problem-solving, driven by continuous scaling that enables them to handle increasingly sophisticated tasks. Using LLMs not only eliminates the massive manual effort required to develop transpilers but also unlocks capabilities that static transpilers simply do not offer, such as simultaneous code refactoring, bug fixing, and the generation of accompanying tests and documentation.

The foundation for this shift towards language models was established by approaches such as TransCoder [36], which demonstrated the viability of unsupervised cross-language translation. Subsequent work moves beyond treating the code as plain text by incorporating structural [37] and compiler-level knowledge [38]. Even as LLMs grow more powerful for code synthesis, they still hallucinate; even though they might output syntactically sound code, the code can still be incorrect on a functional level [39]. This limitation has driven the development of execution-aware frameworks that incorporate dynamic analysis [40] and iterative self-correction [41] to ensure functional equivalence and correctness. Furthermore, to perform automatic code translation at an enterprise level, approaches must scale from isolated, smaller-scale functionalities towards complete repository-scale orchestration such as AlphaTrans [42] and K^{3}Trans [43]. Building on the necessity of repository-scale execution and rigorous validation, this work introduces a collaborative multi-agent pipeline that autonomously orchestrates, translates, and verifies complex codebases.

2.4. Extensions from preliminary work

This article is an extended version of our short paper presented at the MLOps workshop [44]. To provide a more comprehensive blueprint, this extended paper introduces several key additions. We begin with a significantly expanded background and related work section, contextualising our approach within agentic orchestration and code translation. This is followed by a more granular formalisation of the AST-driven chunking and reassembly strategies. Next, we present a more in-depth breakdown of individual agent mechanics, including tools, RAG, and delegate agents. Finally, this extended work adds the sequential pipeline architecture alongside the dynamic pipeline, demonstrated through a newly added application within the industrial project.

3. Translation process

The general translation process for a source to target language codebase consists of three main steps, as illustrated in the diagram in Fig. 1:

1. **Code Chunking**: The source codebase is divided into smaller and LLM-processable chunks, according to the chunking strategy. This strategy is applied to each file in the source project.
2. **Agentic LLM Translation Pipeline**: The pipeline of LLM agents translates each code chunk from the source language to the target language.
3. **Code Reassembly**: The translated chunks are reassembled to form the target language codebase, according to the reassembly strategy. This strategy reconstructs the chunks into a file with a structure similar to the original source file.

3.1. Chunking

3.1.1. Context windows

LLMs have finite context windows, meaning they are limited to predefined maximum token sizes per pass. The context windows vary for different models, *e.g.*, GPT-4o supports up to 128K tokens [45], Gemini 1.5 Pro up to 2 million tokens [46], and LLama 4 Scout up to 10 million tokens [47]. Additionally, models may impose further constraints, such as maximum output token limits, as in GPT-4o's 16K limit [45]. To process large amounts of information that exceed these finite context window limits, chunking is applied. Chunking refers to the process of splitting LLM inputs into smaller, manageable pieces that fit within the model's context windows, allowing them to be processed without exceeding the maximum token limit. 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 to ensure it is 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, *e.g.*, GPT-5.2 supports up to 400K tokens [48] compared to GPT-4o [45], it is not always beneficial to maximise the context window. Processing large inputs as a single chunk can lead to context dilution, in which the relevance of specific information diminishes within the broader context. The dilution effect of context prioritisation over long context lengths has been demonstrated in recent studies [49]. This occurs because LLMs distribute attention across the entire input, where important details, even small ones, may be overshadowed by unrelated or less relevant content. This can have a particularly 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.

3.1.2. Chunking strategy

A Delphi language parser was written from scratch at the start of the industrial project. The parser extracts the abstract syntax tree (AST) and code elements from the source. This parsed representation allows the system to differentiate between high-level architectural structures and low-level declarations. The AST, therefore, allows for two distinct chunking strategies:

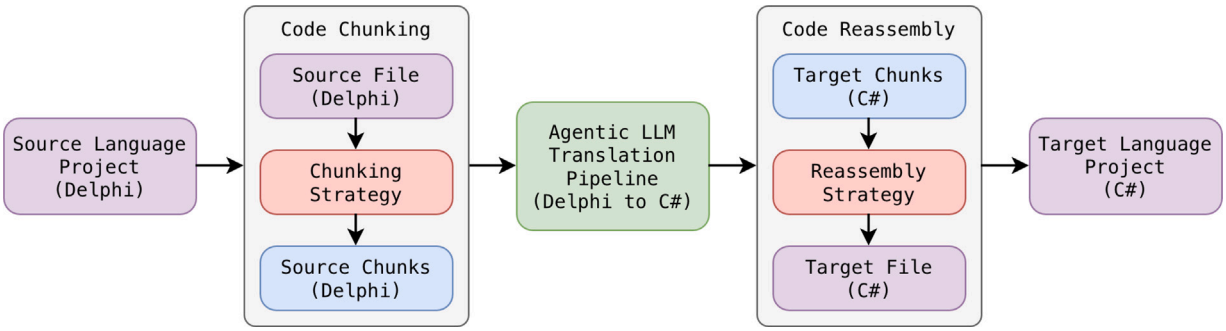

Fig. 1. Diagram of the translation process of a code project from source to target language, consisting of the chunking, code translation, and reassembly strategies.

(a) **Structural Unit Isolation**: For larger and high-level architectural structures such as classes, procedures, and functions, the chunking strategy uses a depth-first traversal of the AST to identify complete syntactic units. By following the tree structure through its leaf nodes and different levels, this approach ensures that nested logic remains encapsulated within its parent structure during the isolation process. Each identified structural unit is extracted as a standalone chunk, which serves as input to the agentic translation pipeline. The isolation is intentional, as it provides the LLM with a clear, focused scope, reducing the risk of overcomplexity and context dilution that might otherwise cause the model to lose track of specific logic amidst a large volume of unrelated code.

(b) **Heuristic Grouping of Atomic Elements**: For smaller code elements such as variable declarations, type definitions, and constants, a heuristic grouping strategy is applied. Rather than placing small atomic elements into separate chunks, which would be computationally inefficient and suboptimal given their low complexity, the approach here aggregates them into composite chunks. This grouping continues until the maximum token threshold is reached.

The combined chunking strategy allows each chunk to be processed independently. Furthermore, assuming no syntax-tree-level or branch dependencies, they can be processed concurrently, as different parts of the codebase that have no relation to each other can be translated in parallel.

3.2. Agentic LLM translation

Once the Delphi source code has been partitioned, the resulting chunks are systematically fed into the agentic LLM translation phase. Rather than relying on a single model pass, this phase utilises a flexible design that supports both simple agentic pipelines and the complex, dynamic orchestration of LLM agents tailored to specific subtasks, such as translation, testing, and debugging. As chunks progress, they undergo rigorous validation to ensure the highest possible translation accuracy, verifying both syntactic validity and functional equivalence. The detailed design of this agentic translation phase, including the specific pipelines and agent stages, is presented in Section 4.

3.3. Code reassembly

As the translation process operates on a chunk-by-chunk basis, a robust reassembly mechanism is required to reconstruct all the translated outputs into a cohesive codebase in the target language. While chunks are actively being translated by the agentic pipeline, a structural skeleton of all source files is generated and maintained.

Established using the AST during the chunking phase, this skeleton preserves the codebase's overarching architecture. It retains all high-level elements with their exact signatures but omits internal logic, replacing it with temporary placeholders. As each chunk is translated by the pipeline, its corresponding placeholder in the skeleton signature is replaced with the generated target-language code. In practice, these placeholders act as syntactically valid stand-ins in the target language. For example, a function might be represented in C# as a stub that raises a *NotImplementedException*.

By systematically mapping translated chunks back to their original AST-derived coordinates, the strategy enables seamless reassembly of the translated code that closely and reliably mirrors the original source structure. In the context of the applied industrial project, the primary objective is a direct, one-to-one mapping, making this skeleton-based reassembly an ideal fit. Modernisations that simultaneously require architectural redesign, thereby deviating from a strict one-to-one translation, would require significantly more complex logic than the structural skeleton alone. However, such complexities can be mitigated by adopting a two-phase translation approach. Initially, a strict translation from the source to the target language is performed using the structural skeletons. Subsequently, a second phase dedicated entirely to architectural refactoring can be executed natively within the target language. Decoupling language translation from architectural redesign simplifies the requirements for LLM agents and significantly increases traceability throughout the modernisation lifecycle.

4. LLM pipeline design

In this section, we first introduce the concept of agent inference in Section 4.1, followed by an agent stage in Section 4.2, and finally multi-agent pipelines to communicate between these agent stages in Section 4.3.

4.1. Agent inference

Agent inference involves directly invoking the LLM model within a structured environment, as illustrated in Fig. 2. As the diagram shows, an input can be optionally augmented by delegate agents before reaching the core agent, which is governed by a specific role specification and model configuration. During inference, the agent can also leverage RAG and external tools to produce its final output. 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.

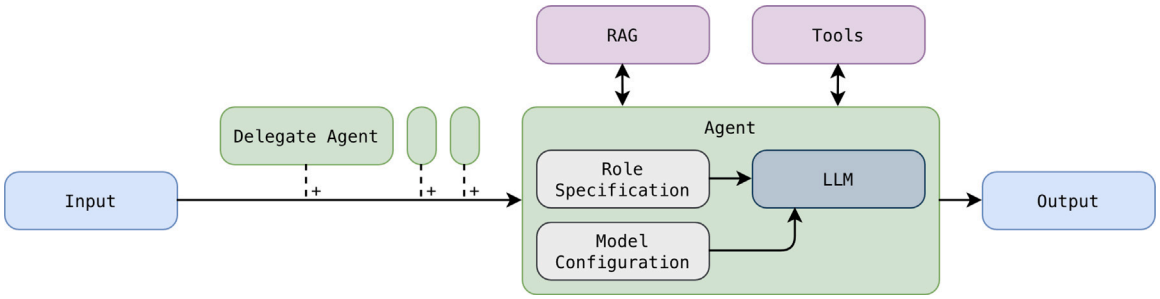

Fig. 2. Diagram illustrating the components and data flow of an agent during inference. It highlights the optional integration of delegate agents for context augmentation prior to the input arriving at the main agent, the internal configuration through the role specification and model configuration of the LLM, and the agent's capacity to utilise external RAG and tool interfaces during inference.

4.1.1. Role specification

The effectiveness of an agent depends strongly on the role assignment of the underlying LLM, which helps define its knowledge, expertise, and goal-directed behaviour. The framework, therefore, employs expert models tuned using persona-based instructions. While model fine-tuning via parameter retraining is a possible approach, the capabilities of modern LLMs often make this unnecessary, as they can generalise effectively and achieve strong performance through prompt-based role specification. The implementation of these roles relies primarily on behavioural framing, in which the agents' roles are established through structured prompting patterns and system-level instructions. This approach allows the models to easily adopt different personas, guiding sub-goal-directed behaviour under clearly defined constraints. Additionally, internal reasoning such as CoT [5,6] can enable the agent to be more effective by encouraging further problem decomposition and more strategic problem-solving before generating model output.

The division of agents and their respective roles is largely determined by the decomposition and complexity of the target problem. Problem decomposition can be defined deterministically by engineers designing the agent system and the tasks it must address. In code generation settings, for example, the overall problem can be separated into specialised subtasks such as translation, test generation, and documentation generation. The level of complexity, on the other hand, may vary widely across models, with some able to handle larger or more complex tasks in a single pass, while others benefit from finer-grained scopes and can leverage more specialised agents. Finally, by assigning specific roles and optimally decomposing the problem, the framework minimises the risk of context dilution.

4.1.2. RAG and tools

To overcome the limitations of static pre-trained weights and context constraints, the framework is designed to equip agents with Retrieval-Augmented Generation (RAG) and tool usage.

While traditional RAG often relies on unstructured semantic search, which can be fragile in large-scale codebases, our framework instead proposes a structure-aware context retrieval approach based on the abstract syntax tree obtained during the chunking phase. By using this parsed representation, the system can identify precise out-of-scope relational definitions that an isolated chunk depends on, such as global variables, external methods, and imported functionality.

Translating chunks without access to the broader context can lead to hallucinations and the introduction of unexpected code. For example, if a required class is defined outside the current chunk, the model might attempt to regenerate the entire class locally to resolve the missing dependency, which would subsequently cause a compilation error. The RAG mechanism is designed to inject the necessary cross-chunk and file-level dependencies directly into the prompt, ensuring the agent has complete relational information within its current context window. Depending on the scale of the dependency, this context may range from the full source code for smaller fragments to strictly essential signatures and type definitions for larger architectural units, thereby mitigating the risk of context overload or dilution. This approach helps manage circular coupling, as the injection of stable signatures provides a consistent reference point regardless of the translation status of the dependent chunk. The granularity of details injected as part of the prompt augmentation can be tuned upon experimental validation as well.

In conjunction, tools represent external functions and interfaces that the agent can autonomously invoke to perform tasks beyond text generation. This capability is a key component of the ReAct [9] framework, interleaving reasoning and acting to solve complex problems. In the context of code translation, tools can provide the model with "execution-aware" capabilities during inference, enabling further validation even during LLM generation. Examples of tools for code translation include compilers, linters, and static analysers.

4.1.3. Delegates

Furthermore, the framework allows delegates to be called upon before agent inference. Delegates are agents themselves that assist in generating a more complete context for the main agent. Within the context of code translation, delegate examples include a planning agent, a software architecture design agent, a pseudocode generation agent, or a solution strategy agent. These delegates generate additional context, which is inserted into the agent's input prompt to improve generation quality by enabling even deeper problem decomposition at the agent level. This modularity allows the primary agent to focus on its specific subtask while benefiting from the strategic output of its delegates. The use of delegates is entirely optional and can be omitted depending on the complexity of the agent requirements.

4.2. 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. In addition to the agent itself, the agent stage encapsulates a sequence of processing and validation functions. It takes text as input and returns text as output, as a generative LLM would. Agent stages are executed sequentially with the given input 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. Each agent stage is uniquely identified by an agent stage card (similar to model cards), which contains metadata (*e.g.*, description and versioning) and a unique ID used for communication between stages during pipeline orchestration. The sequential flow of an agent stage is illustrated in Fig. 3.

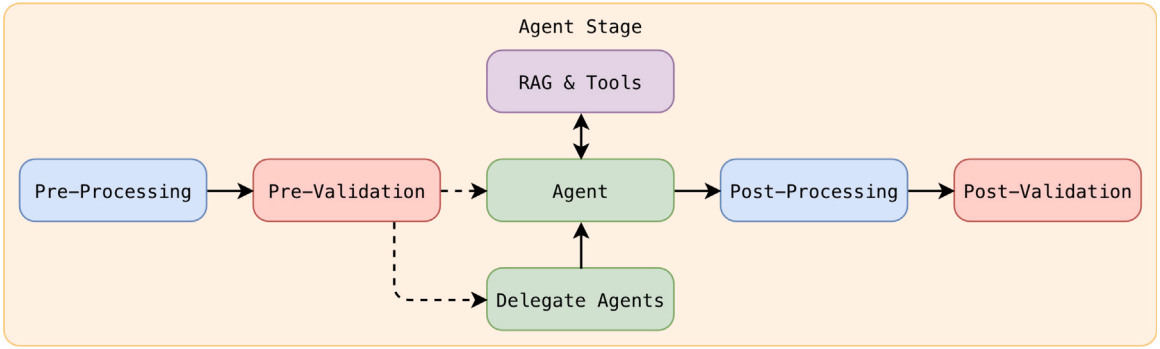

Fig. 3. Diagram of a single agent stage visualising the sequential execution of pre-processing, pre-validation, agent inference, post-processing, and post-validation. Additionally, the core agent may interact with optional delegate agents for context augmentation, alongside the capability for RAG and tools.

4.2.1. Processing and validation functions

To both transform and verify model inputs and outputs, the agent stage incorporates processing and validation functions before and after LLM inference. These functions are entirely optional and can be chained sequentially in any number.

Processing functions handle transformations on model inputs or outputs. The functions receive text as input and return modified text as output. They are applied before or after inference, depending on whether they serve as pre- or post-processing functions. Their primary objective is to prepare an input for a specific model, format a model's output for the subsequent agent stage, or finalise the pipeline's output. Examples of processing functions within code translation include prompt restructuring, code formatting, or deterministic code modifications. Examples of processing functions outside of the code translation context can include data redaction, data normalisation, text restructuring, or context truncation. Validation functions, on the other hand, are responsible for verifying the correctness or suitability of model input and output. These functions take text as input and return a validation result, typically a boolean outcome with accompanying textual feedback. Executed immediately after any processing steps, these functions act as strict guardrails to prevent the propagation of unverified or erroneous information through the pipeline. Examples of validation functions include compilation output, linting, or test runners. In the broader non-code translation context, these validation functions could include checks for dangerous and illegal actions and content.

4.3. Pipelines

With the agent stage established as the fundamental agentic unit of execution, the pipelines orchestrate the structured communication and flow between the individual agent stages. These pipelines can be categorised into two primary architectural patterns: sequential and dynamic.

4.3.1. Sequential pipeline

The sequential pipeline represents a linear orchestration of agent stages, where data flows through the pipeline in a fixed, pre-defined sequence, as illustrated in Fig. 4. The pipeline input is first provided as input to the first agent stage. Following the sequential pattern, the output of each agent stage becomes the input for the subsequent stage until the final agent stage processes the data. The pipeline output is therefore the output produced by this final stage. The data flow in this architecture is deterministic, as the pipeline order is defined during the design phase. This orchestration resembles a waterfall-style process.

If any pre- or post-validation function fails, the pipeline terminates with the validation result (including feedback). Upon all validation functions succeeding in all agent stages, the pipeline successfully completes execution. Throughout the pipeline, the input context is progressively propagated and refined across agent stages, ensuring that the final output represents a composite artifact produced through the sequential contributions of multiple agents.

4.3.2. Dynamic pipeline

In contrast, a dynamic pipeline introduces non-linear workflows in which the execution path is determined at runtime based on the outcomes of previous stages or validation results, *i.e.*, 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:

(a) **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.

(b) **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 continues to the next validation check, the subsequent step within the current stage, or the next stage in the pipeline.

(c) **Iterative Feedback Loops:** The architecture also supports adaptive workflows where an agent stage may loop back to itself or to a previous stage until a post-validation condition is satisfied. To prevent excessive or unstable iteration, particularly due to the stochastic nature of LLM outputs, a predefined iteration limit is enforced.

A general architecture of a dynamic pipeline is illustrated in Fig. 5, 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.

By leveraging these dynamic mechanisms, the framework can better handle LLMs' inherent stochasticity, flagging issues early and adapting the processing path to ensure high-fidelity code translation. Compared to sequential pipeline orchestration, the dynamic pipeline resembles a more agile-style process.

5. Translation pipelines

Having detailed the translation strategy in Section 3 and the LLM pipeline design in Section 4, this section presents two initial pipelines for Delphi-to-C# translation: one sequential and one dynamic. These represent preliminary agent orchestrations that will systematically evolve through modifications and extensions as the project matures. Following MLOps and LLMOps practices, we iteratively refine the

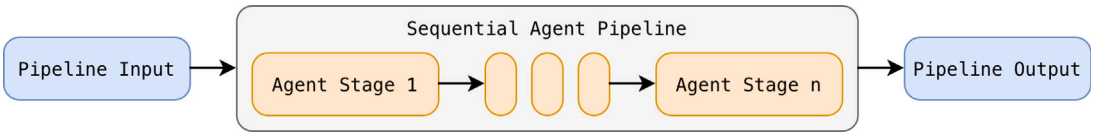

Fig. 4. Diagram of a sequential agent pipeline illustrating the linear and deterministic flow of data from the initial input, through n consecutive agent stages, to the final pipeline output.

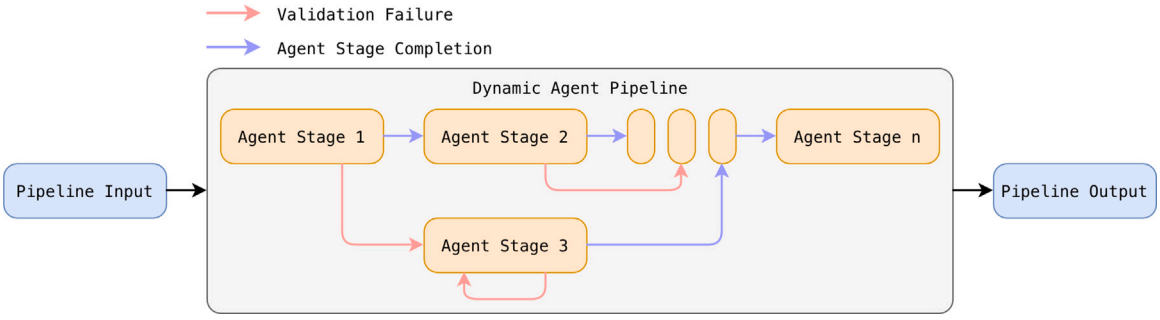

Fig. 5. Diagram of a dynamic agent pipeline illustrating non-linear workflows. The architecture demonstrates conditional re-routing, where an input chunk progresses through stage completion (e.g., Stage 1 to Stage 2) or is redirected upon validation failure (e.g., Stage 1 to Stage 3) to enable adaptive problem-solving.

pipeline by benchmarking with complete translation runs against a large set of metrics (*e.g.*, compilation success rate, number of unique errors/warnings, token usage). This benchmarking is performed on a representative subset of the five MLoC codebase. However, because subtle LLM hallucinations or logical deviations are often not captured by automated metrics, this quantitative benchmarking is paired with qualitative expert review to accurately assess the agent and pipeline's performance. Based on these combined iterative findings, we can introduce specialised agents and fine-tune existing configurations to mitigate the specific translation challenges inherent to this language pair.

5.1. Sequential pipeline

The sequential pipeline is well-suited for straightforward agent orchestrations and problem-solving tasks with a narrower scope. To demonstrate the sequential architecture, we present a pipeline for Delphi-to-C# translation for User Interface (UI) components, where, as part of the pure translation, a modernisation step is applied to the translated code. The early-stage design for this pipeline is illustrated in Fig. 6. The sequential pipeline comprises the following agent stages to be called in linear order:

- **Delphi-to-C# UI Translation Stage**: Agent stage for translating UI Delphi chunks to C# chunks.
- **C# UI Modernisation Stage**: Agent stage for modernising the translated C# UI chunks.

The objective of this pipeline is to translate UI components from Delphi to C#, and on top of this, modernise the UI logic. The translation from Delphi to C# happens in the first agent stage, while the modernisation happens in the subsequent agent stage, which is fine-tuned for modernising the UI logic. Since the UI translation in this case is simpler and modernisation can only happen post-translation, a sequential pipeline is optimal here. The validation functions can be run; however, upon validation failure, the pipelines are terminated for rerun or flagged for manual intervention.

5.2. Dynamic pipeline

The dynamic pipeline supports larger agent-stage orchestrations and more complex problem decomposition. Here, we present a test-driven code translation pipeline that focuses on translating Delphi code chunks to C# while generating tests in both source and target languages to validate functional correctness. Fig. 7 illustrates an early-stage design for such a pipeline. The pipeline comprises 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.

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. The goal is to generate equivalent tests in both languages, allowing them to be evaluated for the same cases; otherwise, the chunks would end up with deviating test parameters. 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 additional context from other chunks, 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 include tools such as linters, compilers, and static analysis, further validating the correctness of the generated code. The agents communicate dynamically through these post-validation checks, enabling the pipeline to adapt its path based on their outputs. 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, with one stage 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

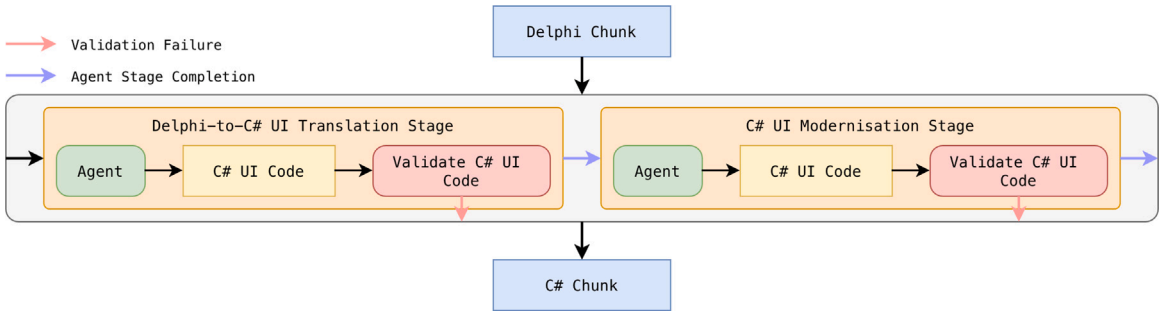

Fig. 6. Diagram of a preliminary pipeline architecture for UI code translation with modernisation, visualising two agent stages communicating using a sequential pipeline.

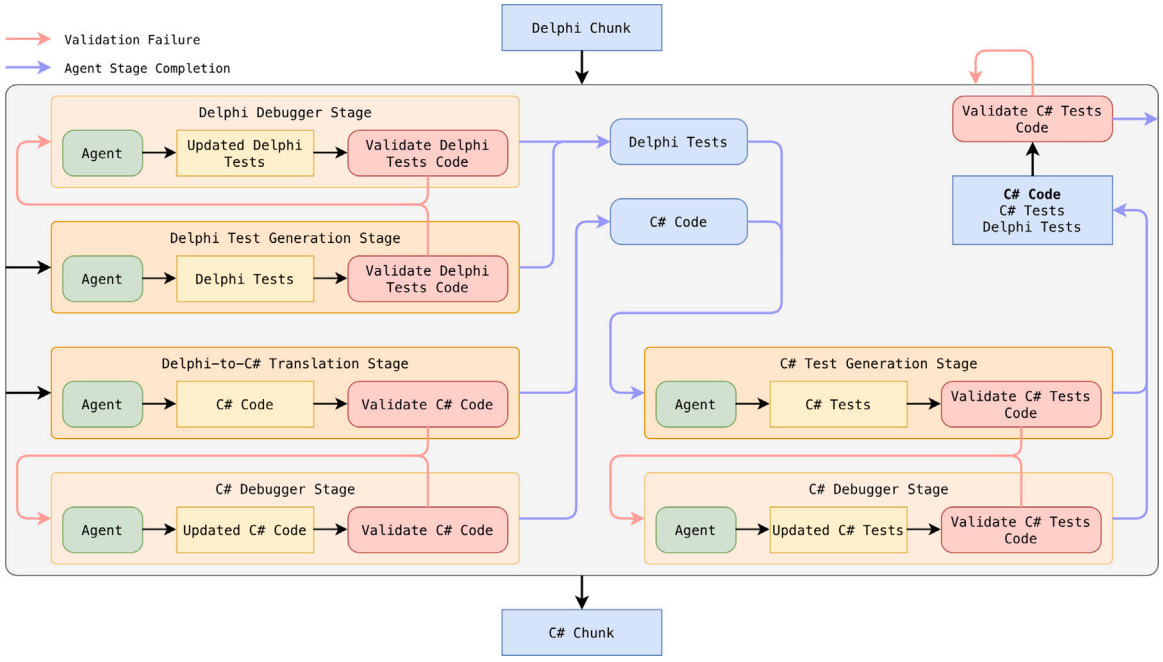

Fig. 7. Diagram of an initial draft pipeline architecture for test-driven code translation, visualising several agent stages communicating dynamically based on validation results on generated code and tests.

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 via simple counters 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 corrected or can only be corrected post-assembly.

In Appendix, we present a step-by-step example walking through the dynamic pipeline.

5.3. Functional correctness and behavioural equivalence

As demonstrated in the architectural design and the preliminary pipelines, our validation strategy relies heavily on syntactic and static analysis, including compilation and linting. However, static analysis cannot guarantee functional correctness or behavioural equivalence between source and target programs. Proving that two Turing-complete programs exhibit identical behaviour on all inputs is undecidable. In particular, program equivalence is undecidable by reduction from the Halting Problem and follows as a consequence of Rice's theorem [50]. This limitation is not specific to language-model-generated code, but reflects a fundamental constraint that also applies to traditional, deterministic compiler systems [51].

Therefore, rather than attempting to prove correctness or equivalence during the code translation process, the framework is designed to systematically maximise the confidence in translation accuracy while mitigating the risks introduced by the stochastic nature of LLM-based systems. To this end, we impose deterministic guardrails around the models to reduce the likelihood of semantic divergence. Concretely, these include (i) static code analysis including linting and compileability checks, (ii) generating and running equivalent test suites in both source and target languages and iterative multi-agent debugging loops in case of test failure, and (iii) AST-driven structural reconstruction guaranteeing structural equivalence onclass level. Following translation, all target-language code additionally (iv) undergoes comprehensive human testing and review. It is important to note that while the other guardrails are fully operational, the dual-language test generation (ii) currently remains under active development, making the human review (iv) an essential step for the time being.

While a 100% guarantee of functional correctness and behavioural equivalence is unattainable in principle, these techniques following engineering practices enable a highly controlled and empirically verifiable level of confidence in the translation process.

Table 1
Lines of Code (LoC) for the benchmarked Delphi units.

Unit	LoC
1	4152
2	2144
3	953
4	5208
5	4287

6. Implementation status and preliminary observations

While the architectural blueprint outlined in this paper establishes a comprehensive theoretical framework, its practical deployment in the 5 MLoC Delphi-to-C# translation project is actively ongoing in a phased manner. The core infrastructure is implemented and operational, including the AST-driven chunking strategy with a Delphi parser, a dynamic multi-agent LLM pipeline, and structural reassembly mechanisms with the skeletons. Within the dynamic pipeline, the translation and debugger agent stages are set up and in use. Conversely, advanced capabilities such as dual-language test generation, delegates, structural RAG, and external toolings integration remain under active development and represent the project's immediate next milestones.

To establish an initial set-up benchmark and explore the behaviour of the agentic pipeline architecture, we conducted and share results of preliminary experiments on five distinct Delphi units from the industrial codebase. These specific units were selected primarily because they were among the initial set of components translated within the active industrial project. Furthermore, establishing a sample of five units provided a reasonable and manageable subset to run multiple independent iterations across different models and pipeline configurations. The varying sizes of these units, which range from 953 to 5208 Lines of Code (LoC), are detailed in Table 1. In this initial benchmark, each unit was translated using a single-agent and a multi-agent pipeline.

To ensure a fair baseline comparison, the single agent uses the exact same prompt as the initial translator agent in the multi-agent pipeline. The multi-agent architecture subsequently employs two distinct debugger agents: one dedicated to addressing compiler errors and another for resolving linter warnings. It is important to note that the prompts for the translator and debugger agents were not iteratively fine-tuned for this evaluation. Additionally, neither retrieval-augmented generation (RAG) nor external tools were employed. These configurations were evaluated using two models: GPT-4o and Claude Opus 4.6. To enhance reliability and reduce stochastic variability, each parameter setting was evaluated in three independent runs, with averaged results reported in Tables 2 and 3.

6.1. Resource consumption and overhead

Table 2 details the resource consumption and generation efficiency. As anticipated, the multi-agent pipeline incurs a higher computational cost in both tokens and runtime due to its validation and dynamic routing to additional agents. Across both models and units, runtime and absolute token usage increase by a factor of approximately 2 to 2.5 for the three-agent pipeline compared to the single-agent baseline. Specifically, input token volume ranges from 20 to 28 tokens per line of code (LoC) for the single-agent baseline, scaling to 40–56 tokens per LoC for the multi-agent pipelines. Similarly, output tokens per LoC rise from 7–12 for the single-agent approach to 21–39 for the multi-agent pipeline. This empirically confirms that deploying execution-aware debugger agents increases operational computational costs by several factors. However, this can be an acceptable trade-off in an industrial project, where fewer errors from the automatic code translation can lead to a substantial reduction in manual developer effort in post-translation.

6.2. Validation resolution dynamics and agent routing

Table 3 details the number of unique errors and warnings found post-assembly, alongside the rerouting rates to the debugger agents. Across the benchmarked units, the need for a dynamic architecture is clearly reflected in these routing percentages. Specifically, between 84% and 99% of the translated chunks triggered compiler errors, while 60% to 92% produced linter warnings, necessitating immediate redirection to the respective debugger agents. This high rerouting frequency aligns with our expectations, given the strict isolation of chunk-level processing and the complex syntactic differences between Delphi and C#.

Post-assembly, the volume of unique errors and warnings remains comparable between single-agent and multi-agent inference. This plateau in error reduction likely stems from two primary factors. First, many lingering compilation issues are out-of-scope structural dependencies that inherently cannot be resolved at the chunk level, thus requiring a broader architectural scope. Second, the debugger agents currently lack the specialised tuning required to reliably patch certain legacy patterns. Addressing this limitation is the precise motivation behind the AST-driven structural RAG detailed in Section 4.1.2. In future iterations, implementing this mechanism will allow the pipeline to inject stable, cross-file signatures directly into the prompt, equipping agents with necessary relational information without reintroducing context overload or dilution. Additionally, future work will require targeted prompt refinement, as current debugger agents lack the specialised tuning required to reliably patch certain legacy patterns.

6.3. Translation velocity and effort reduction

The ultimate justification for the automatic translation effort and the computational overhead introduced by the multi-agent pipeline is the substantial reduction in overall manual engineering effort. Prior to the project's initiation, the company estimated that a purely manual translation would roughly require up to 160 h per 1000 LoC. Currently, by leveraging the automated pipeline, the engineering team achieves a translation velocity of approximately 15 h per 1000 LoC. Importantly, this operational metric is highly inclusive; it accounts for the entire software development lifecycle, encompassing the automated translation runs, subsequent post-translation manual fixes, testing, and necessary team alignment meetings. This represents an acceleration of more than 10× compared to the manual baseline, empirically validating that the whole framework succeeds in making repository-scale translation significantly more manageable and cost-effective.

Furthermore, even when comprehensively accounting for the time invested in research & development (R&D) and initial tool development (*e.g.*, pipeline creation of LLM agents), the project establishes a solid 4× overall speed-up. Moving forward, the foundational tool development has been completed and transitioned into a low-effort maintenance phase. Concurrently, AI-specific R&D, such as further benchmarking, tool expansion, structural RAG integration, and dual-code test generation, continues to advance in parallel without bottlenecking the active translation effort. Ultimately, these velocity metrics empirically validate that the whole framework succeeds in making repository-scale translation significantly more manageable and cost-effective compared to manual developer effort.

7. Conclusion and future work

In this paper, we present a systematic, modular multi-agent pipeline framework for automating large-scale code translation. To address finite context windows, context dilution, and the inherent stochasticity of LLMs, we introduced an AST-driven chunking strategy, chunk-level translation, and a structural reassembly strategy. By designing the modular framework with initial Delphi-to-C# translation pipelines for an ongoing 5 MLoC industrial modernisation project, we established the

Table 2
Resource consumption and generation efficiency of single-agent versus multi-agent pipelines, using GPT-4o and Opus 4.6. Metrics include average runtime in minutes, absolute token usage, and token volume normalised per Line of Code (LoC). Standard deviations are reported for runtime and output tokens per LoC to indicate system and output stability.

Unit	Model	Pipeline type	Runtime (min)	Tokens (In)	Tokens (Out)	In/LoC	Out/LoC
1	GPT-4o	Single-agent	11.87 ± 0.42	89 819	33 029	21.63	7.95 ± 0.03
		Multi-agent	25.03 ± 1.25	181 430	98 643	43.70	23.76 ± 0.11
	Opus 4.6	Single-agent	20.31 ± 2.89	109 694	47 682	26.42	11.48 ± 0.03
		Multi-agent	47.36 ± 6.73	213 097	140 722	51.32	33.89 ± 0.31
2	GPT-4o	Single-agent	7.31 ± 0.88	49 181	15 587	22.94	7.27 ± 0.16
		Multi-agent	12.54 ± 0.14	94 414	45 248	44.04	21.10 ± 0.13
	Opus 4.6	Single-agent	9.76 ± 0.14	58 861	20 498	27.45	9.56 ± 0.02
		Multi-agent	23.04 ± 0.03	110 653	64 645	51.61	30.15 ± 0.36
3	GPT-4o	Single-agent	2.72 ± 0.05	20 260	6 728	21.26	7.06 ± 1.67
		Multi-agent	5.85 ± 0.74	40 201	21 089	42.18	22.13 ± 0.13
	Opus 4.6	Single-agent	5.42 ± 0.53	24 387	8 865	25.59	9.30 ± 0.13
		Multi-agent	10.29 ± 0.53	49 651	33 189	52.10	34.83 ± 2.25
4	GPT-4o	Single-agent	13.99 ± 0.12	105 587	47 211	20.27	9.07 ± 0.25
		Multi-agent	31.64 ± 5.96	236 360	147 816	45.38	28.38 ± 0.72
	Opus 4.6	Single-agent	20.95 ± 0.07	127 797	56 250	24.54	10.80 ± 0.02
		Multi-agent	57.82 ± 1.56	279 543	198 520	53.68	38.12 ± 2.56
5	GPT-4o	Single-agent	14.01 ± 1.48	91 274	35 738	21.29	8.34 ± 0.19
		Multi-agent	24.29 ± 0.72	194 122	109 347	45.28	25.51 ± 0.33
	Opus 4.6	Single-agent	18.85 ± 0.12	111 214	47 781	25.94	11.15 ± 2.24
		Multi-agent	49.05 ± 1.50	236 969	158 828	55.28	37.05 ± 0.00

Table 3
Error and warning metrics across single-agent and multi-agent architectures. The table reports the average number of unique errors and warnings with standard deviations. For the multi-agent pipeline, the percentage of routing to error and warning debugger agents is also provided.

Unit	Model	Pipeline type	No. unique errors	No. unique warnings	% Error agent	% Warning agent
1	GPT-4o	Single-agent	28.33 ± 1.53	11.67 ± 0.58	–	–
		Multi-agent	28.67 ± 1.53	10.33 ± 0.58	87.94%	89.60%
	Opus 4.6	Single-agent	21.33 ± 1.53	10.67 ± 1.15	–	–
		Multi-agent	27.67 ± 2.08	8.67 ± 1.15	84.87%	67.61%
2	GPT-4o	Single-agent	21.33 ± 4.04	9.00 ± 0.00	–	–
		Multi-agent	25.33 ± 3.06	9.33 ± 1.15	95.98%	77.91%
	Opus 4.6	Single-agent	17.00 ± 1.00	4.67 ± 0.58	–	–
		Multi-agent	25.67 ± 9.29	8.33 ± 0.58	91.57%	61.45%
3	GPT-4o	Single-agent	13.00 ± 1.00	5.33 ± 1.53	–	–
		Multi-agent	14.33 ± 6.11	6.67 ± 1.53	93.75%	83.33%
	Opus 4.6	Single-agent	7.33 ± 0.58	2.67 ± 0.58	–	–
		Multi-agent	13.00 ± 1.73	5.67 ± 0.58	90.63%	60.42%
4	GPT-4o	Single-agent	33.00 ± 3.61	7.67 ± 1.15	–	–
		Multi-agent	22.00 ± 8.66	8.33 ± 2.08	98.16%	91.21%
	Opus 4.6	Single-agent	23.33 ± 1.15	5.00 ± 0.00	–	–
		Multi-agent	21.33 ± 0.58	9.00 ± 1.00	96.11%	73.62%
5	GPT-4o	Single-agent	22.00 ± 1.73	6.67 ± 0.58	–	–
		Multi-agent	21.67 ± 1.15	9.00 ± 1.00	98.87%	91.44%
	Opus 4.6	Single-agent	15.33 ± 0.58	12.33 ± 0.58	–	–
		Multi-agent	18.33 ± 3.21	8.67 ± 1.53	97.30%	78.38%

architectural blueprint for safe partitioning, independent processing, and rigorous validation of complex translations.

Our initial empirical evaluations demonstrate that while a dynamic multi-agent pipeline introduces computational overhead compared to a single-agent baseline, it provides the necessary operational guardrails to flag syntactic and compiler issues automatically. By reducing the translation and debugging effort from an estimated 160 h to approximately 15 h per 1000 LoC, the framework achieves a 10× acceleration over manual translation. This translates to a solid 4× overall project speed-up when factoring in the initial AI tool development and R&D, effectively shifting the cost burden away from manual human translation toward scalable, automated execution.

It is important to note that our initial benchmarking utilised non-finetuned prompts, which yielded quantitatively similar error rates between single-agent and multi-agent setups at the chunk level. However, our broader practical experience indicates that as the system is iteratively fine-tuned, a collaborative suite of specialised agents offers distinct qualitative advantages. Delegating work across distinct agent stages provides granular control over the pipeline, enhancing explainability and significantly improving code quality to seamlessly align with strict industrial project standards. While iterative prompt engineering and dual-language testing remain resource-intensive challenges, we observe that this effort reliably diminishes over time as the pipelines mature.

Ultimately, this engineering-driven framework establishes a robust foundation for integrating LLMs into enterprise-scale legacy modernisation. Rather than assuming flawless performance from a single model, the approach acknowledges the inherent stochasticity of LLMs by

embedding generative agents within strict validation guardrails and dynamic orchestration mechanisms. As underlying models inevitably evolve and improve, this structured pipeline ensures that their outputs remain verifiable, controllable, and suitable for industrial deployment.

Moving forward, our primary focus will be on the rigorous empirical evaluation of the proposed conceptual architectures. While initial experiments yield promising results, further work is required to assess the scalability, error handling, and adaptability of this multi-agent approach. In the short term, we will concentrate on enhancing agent coordination and establishing comprehensive performance metrics. Ultimately, by formally benchmarking the efficiency gains of this automated pipeline against the estimated manual developer effort, we aim to validate this conceptual blueprint as a sustainable, enterprise-ready solution for legacy code modernisation.

Beyond the immediate scope of the current industrial project, future research will explore the broader generalisation of this framework. We intend to evaluate the pipeline's performance across various foundational LLMs to ensure true model-agnosticism. Additionally, we aim to expand the framework's application to other programming language pairs, encompassing both legacy and modern languages, and to adapt the orchestration for industrial projects that require native architectural refactoring rather than strict one-to-one translations.

CRediT authorship contribution statement

Tibo Bruneel: Writing – original draft, Visualization, Software, Methodology, Investigation, Data curation, Conceptualization. **Sandu Cruceresc u:** Writing – review & editing, Software, Methodology, Conceptualization. **Welf Löwe:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Morgan Ericsson:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Diego Perez-Palacin:** Writing – review & editing, Supervision. **Jonas Nordqvist:** Writing – review & editing, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Tibo Bruneel reports financial support was provided by Softwerk AB. Sandu Cruceresc u reports financial support was provided by Softwerk AB. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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, Sweden. 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 underpins this work, and active support for open research. The authors would also like to thank the engineering team at Softwerk AB for their dedicated efforts on this industrial project, particularly Varun Dalal and Dr. Rüdiger Lincke.

Appendix. Conceptual end-to-end translation example

To illustrate the operational details of the dynamic pipeline, including chunking, routing logic, and reassembly behaviour, we present a conceptual end-to-end trace of a single code chunk using the pipeline architecture presented in Section 5.2 and Fig. 7. We use a standard business logic function, `CalculateFee`, to demonstrate how the fully realised framework will handle the translation from Delphi to C#, including test generation in both languages and a translation failure with the C# debugger agent patching the issue.

The initial source codebase is parsed, and the following Delphi function is isolated as a discrete chunk via the Abstract Syntax Tree (AST):

```
function CalculateFee(Amount: Double;
    IsUrgent: Boolean): Double;
begin
  if IsUrgent then Result := Amount * 0.10
  else Result := Amount * 0.05;
end;
```

While chunks are isolated for processing, a structural skeleton of the full unit is maintained to provide placeholders for each chunk's coordinates during reassembly. The chunk then enters the test-driven dynamic pipeline. Table 4 details the step-by-step routing and validation process, highlighting the recovery from a syntactically invalid C# translation using the routing mechanism.

Following successful validation, the chunk is mapped back to the structural skeleton. The final, reassembled C# target code seamlessly integrates into the broader architecture:

```
public double CalculateFee(double amount,
    bool isUrgent)
{
    if (isUrgent)
    {
        return amount * 0.10;
    }
    return amount * 0.05;
}
```

Alongside the translated functional code, the pipeline successfully generates the corresponding unit tests in both the source and target languages to validate behavioural equivalence across the translation steps.

The generated Delphi baseline test:

```
procedure TestCalculateFee;
begin
  Assert(CalculateFee(100.0, True) = 10.0);
  Assert(CalculateFee(100.0, False) = 5.0);
end;
```

The equivalent C# test derived from the pipeline:

```
[Fact]
public void TestCalculateFee()
{
    Assert.Equal(10.0, CalculateFee(100.0, true));
    Assert.Equal(5.0, CalculateFee(100.0, false));
}
```

Data availability

The data and source code that support the findings of this study are not publicly available due to commercial confidentiality and proprietary restrictions. Specifically, the legacy Delphi codebase and the resulting C# translations are the commercial property of Danfoss Power Solutions.

Table 4
End-to-end operational execution trace for the `CalculateFee` function, demonstrating dynamic routing upon a significant translation failure and cross-language test generation.

Step	Phase/Stage	Input	Operational logic	Output
1	Chunking	Raw Delphi `.pas` source file.	Depth-first traversal of the AST identifies `CalculateFee` as a standalone structural unit.	**(1) Isolated Chunk**: Delphi source code. **(2) Skeleton**: C# target file with `NotImplementedException` placeholders.
2a	Delphi Test Generation	Isolated Delphi code chunk.	Stage `TEST_GEN_01` generates baseline unit tests based on original logic in the Delphi language.	**Delphi Test Code**: `Assert(CalculateFee(100, True) = 10.0);`.
2b	Delphi to C# Translation	Isolated Delphi code chunk.	Stage `TRANS_01` performs inference using persona-based instructions and model configurations.	**Failed C# Code**: `if (isUrgent) result := amount * 0.10;` (Error: Legacy Delphi assignment := and missing `return`).
3	Delphi to C# Translation Post-Validation	Failed C# code snippet.	Deterministic guardrail (compiler) in the `TRANS_01` agent stage post-validation function verifies syntactic validity and flags errors.	**Result:** `False` **Feedback:** "CS1525: Invalid term ':= ' "; "CS0103: 'result' does not exist".
4	Dynamic Routing to C# Debugger Stage	Broken C# code and validation feedback.	`TRANS_01` stage redirects the failed chunk through the `onValidationFailureStageID` to the C# debugger `DEBUG_C#_01` stage, which repairs broken C# code.	**Repaired C# Code**: `public double CalculateFee(...) { if (isUrgent) return amount * 0.10; . . . }`.
5	C# Test Generation	Repaired C# code from step 4 and Delphi tests from step 2a.	Stage `TEST_GEN_02` generates baseline unit tests from both target code and source-language tests to ensure equivalent test parameters.	**C# Test Code**: `Assert.Equal(10.0, CalculateFee(100, true));`.
6	Reassembly	Validated C# chunk and skeleton.	Reassembly strategy replaces skeleton placeholders using original AST-derived coordinates.	**Final Target File**: A cohesive, syntactically valid C# codebase mirroring the source.

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] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D.M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, 2020, arXiv:2005.14165 URL https://arxiv.org/abs/2005.14165.

[3] J. Wei, M. Bosma, V.Y. Zhao, K. Guu, A.W. Yu, B. Lester, N. Du, A.M. Dai, Q.V. Le, Finetuned language models are zero-shot learners, 2021, arXiv:2109.01652 URL https://arxiv.org/abs/2109.01652.

[4] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D.C. Schmidt, A prompt pattern catalog to enhance prompt engineering with ChatGPT, 2023, arXiv:2302.11382 URL https://arxiv.org/abs/2302.11382.

[5] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-Thought prompting elicits reasoning in large language models, 2022, arXiv:2201.11903 URL https://arxiv.org/abs/2201.11903.

[6] T. Kojima, S.S. Gu, M. Reid, Y. Matsuo, Y. Iwasawa, Large language models are zero-shot reasoners, 2022, arXiv:2205.11916 URL https://arxiv.org/abs/2205.11916.

[7] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, T. Scialom, Toolformer: Language models can teach themselves to use tools, in: A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Advances in Neural Information Processing Systems, Vol. 36, Curran Associates, Inc., 2023, pp. 68539–68551, URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

[8] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, M. Sun, ToolLLM: Facilitating large language models to master 16000+ Real-world APIs, 2023, arXiv:2307.16789 URL https://arxiv.org/abs/2307.16789.

[9] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, 2023, arXiv:2210.03629 URL https://arxiv.org/abs/2210.03629.

[10] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, S. Yao, Reflexion: Language agents with verbal reinforcement learning, 2023, arXiv:2303.11366 URL https://arxiv.org/abs/2303.11366.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.d. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F.P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W.H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A.N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, W. Zaremba, Evaluating large language models trained on code, 2021, arXiv:2107.03374 URL https://arxiv.org/abs/2107.03374.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained model for programming and natural languages, 2020, arXiv:2002.08155 URL https://arxiv.org/abs/2002.08155.

[13] X. Chen, M. Lin, N. Schärli, D. Zhou, Teaching large language models to Self-Debug, 2023, arXiv:2304.05128 URL https://arxiv.org/abs/2304.05128.

[14] L. Zhong, Z. Wang, J. Shang, Debug like a human: A large language model debugger via verifying runtime execution Step-by-step, 2024, arXiv:2402.16906 URL https://arxiv.org/abs/2402.16906.

[15] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B.P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, P. Clark, Self-Refine: Iterative refinement with self-feedback, 2023, arXiv:2303.17651 URL https://arxiv.org/abs/2303.17651.

[16] J. Yang, C.E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, O. Press, SWE-agent: Agent-Computer interfaces enable automated software engineering, 2024, arXiv:2405.15793 URL https://arxiv.org/abs/2405.15793.

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

[18] J. He, C. Treude, D. Lo, LLM-Based multi-agent systems for software engineering: Literature review, vision, and the road ahead, ACM Trans. Softw. Eng. Methodol. 34 (5) (2025) http://dx.doi.org/10.1145/3712003.

[19] G. Li, H. Hammoud, H. Itani, D. Khizbullin, B. Ghanem, CAMEL: Communicative agents for "Mind" exploration of large language model society, in: A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, S. Levine (Eds.), Advances in Neural Information Processing Systems, Vol. 36, Curran Associates, Inc., 2023, pp. 51991–52008, URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a3621ee907def47c1b952ade25c67698-Paper-Conference.pdf.

[20] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C.-M. Chan, H. Yu, Y. Lu, Y.-H. Hung, C. Qian, Y. Qin, X. Cong, R. Xie, Z. Liu, M. Sun, J. Zhou, AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors, 2023, arXiv:2308.10848 URL https://arxiv.org/abs/2308.10848.

[21] 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, arXiv:2307.07924 URL https://arxiv.org/abs/2307.07924.

[22] 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, arXiv:2308.00352 URL https://arxiv.org/abs/2308.00352.

[23] M.H. Nguyen, T. Phan Chau, P.X. Nguyen, N.D.Q. Bui, AgileCoder: Dynamic collaborative agents for software development based on agile methodology, in: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), 2025, pp. 156–167, http://dx.doi.org/10.1109/Forge66646.2025.00026.

[24] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, Y. Lou, TRANSAGENT: An LLM-based multi-agent system for code translation, 2024, arXiv:2409.19894 URL https://arxiv.org/abs/2409.19894.

[25] 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, arXiv:2404.14646 URL https://arxiv.org/abs/2404.14646.

[26] 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, arXiv:2312.13010 URL https://arxiv.org/abs/2312.13010.

[27] Y. Ishibashi, Y. Nishimura, Self-Organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization, 2024, arXiv:2404.02183 URL https://arxiv.org/abs/2404.02183.

[28] 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, arXiv:2404.17153 URL https://arxiv.org/abs/2404.17153.

[29] 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, arXiv:2503.22512 URL https://arxiv.org/abs/2503.22512.

[30] H. Chase, LangChain: The agent engineering platform, 2022, https://github.com/langchain-ai/langchain.

[31] Agno, Agno: Framework for building multi-modal agents and workflows, 2024, https://github.com/agno-agi/agno.

[32] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, et al., AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023, arXiv:2308.08155.

[33] J. Moura, CrewAI: Framework for orchestrating role-playing, autonomous AI agents, 2023, https://github.com/crewAIInc/crewAI.

[34] Google, Agent2Agent (A2A) Protocol, 2025, https://github.com/google/A2A. (Accessed 12 May 2025).

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

[36] B. Roziere, M.-A. Lachaux, L. Chanussot, G. Lample, Unsupervised translation of programming languages, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, Vol. 33, Curran Associates, Inc., 2020, pp. 20601–20611, URL https://proceedings.neurips.cc/paper_files/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf.

[37] S. Tipirneni, M. Zhu, C.K. Reddy, StructCoder: Structure-Aware transformer for code generation, 2022, arXiv:2206.05239 URL https://arxiv.org/abs/2206.05239.

[38] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, G. Synnaeve, Code translation with compiler representations, 2023, arXiv:2207.03578 URL https://arxiv.org/abs/2207.03578.

[39] R. Pan, A.R. Ibrahimzada, R. Krishna, D. Sankar, L.P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, 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, ACM, 2024, pp. 1–13, http://dx.doi.org/10.1145/3597503.3639226.

[40] M. Shetty, N. Jain, A. Godbole, S.A. Seshia, K. Sen, Syzygy: Dual code-test C to (safe) rust translation using LLMs and dynamic analysis, 2024, arXiv:2412.14234 URL https://arxiv.org/abs/2412.14234.

[41] X. Yin, C. Ni, T.N. Nguyen, S. Wang, X. Yang, Rectifier: Code translation with corrector via LLMs, 2024, arXiv:2407.07472 URL https://arxiv.org/abs/2407.07472.

[42] A.R. Ibrahimzada, K. Ke, M. Pawagi, M.S. Abid, R. Pan, S. Sinha, R. Jabbarvand, AlphaTrans: A Neuro-Symbolic compositional approach for repository-level code translation and validation, Proc. ACM Softw. Eng. 2 (FSE) (2025) 2454–2476, http://dx.doi.org/10.1145/3729379.

[43] G. Ou, M. Liu, Y. Chen, X. Du, S. Wang, Z. Zhang, X. Peng, Z. Zheng, Evolving triple knowledge-augmented LLMs for code translation in repository context, 2025, arXiv:2503.18305 URL https://arxiv.org/abs/2503.18305.

[44] T. Bruneel, W. Löwe, M. Ericsson, D. Perez-Palacin, J. Nordqvist, Automated code translation using an engineering-driven pipeline approach of language models, in: Workshop on Machine Learning Operations, (MLOps), Vol. 4109, CEUR-WS.org, 2025, pp. 48–54.

[45] OpenAI, GPT-4o Platform Docs, 2025, https://developers.openai.com/api/docs/models/gpt-4o. (Accessed 09 May 2025).

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

[47] MetaAI, LLaMA 4 and the Scout Model, 2025, https://www.llama.com/models/llama-4/. (Accessed 09 May 2025).

[48] OpenAI, GPT-5.2 Platform Docs, 2025, https://developers.openai.com/api/docs/models/gpt-5.2. (Accessed 20 February 2026).

[49] 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, arXiv:2307.03172 URL https://arxiv.org/abs/2307.03172.

[50] H.G. Rice, Classes of recursively enumerable sets and their decision problems, Trans. Amer. Math. Soc. 74 (2) (1953) 358–366, http://dx.doi.org/10.1090/S0002-9947-1953-0053041-6.

[51] G. Goos, W. Zimmermann, Verification of compilers, in: Correct System Design, Recent Insight and Advances, (To Hans Langmaack on the Occasion of His Retirement from His Professorship At the University of Kiel), Springer-Verlag, Berlin, Heidelberg, 1999, pp. 201–230.