# Investigating the Application of AI Agent Techniques to MDE Code Generation

Qiaomu Xue[1], Kevin Lano[1]

[1]King's College London, London, UK

**Abstract**

Code generation is a key process in model-driven engineering (MDE) approaches, but the construction of code generators can require significant human resources and time. For an agile MDE approach it is essential that code generators can be produced and modified rapidly to meet changing code generation requirements, without undue and excessive effort. Agentic AI approaches appear to be a relevant method to achieve this goal, and in this paper we report on our investigations with an agentic code generator system and the issues which arise with this approach. Agentic AI for code generation is motivated by the need to use large language model (LLM) code knowledge to induce code generation rules, combined with deterministic rule execution.

**Keywords**

Code generation, Model-driven engineering, LLMs, Agentic AI

## 1. Introduction

In previous works, we described and demonstrated the performance of the 'code generation by example' (CGBE) transformation synthesis approach for different code generation or language translation cases [1, 2, 3], and we compared this approach with LLM fine tuning and directly invoking LLM APIs for cases of Kotlin and JavaScript code generators [4]. CGBE is based on symbolic machine learning: a search-based process tries to discover mapping rules that relate the abstract syntax trees (ASTs) of source language examples to the corresponding target language ASTs. Typically this process is used to synthesise code generators mapping from UML/OCL specifications/designs to code in an executable programming language such as Java or Python. The training datasets for CGBE are sets of examples, consisting of pairs `source`, `target` of expressions, types, statements and declarations in the source language and target language. For example:

```
UML/OCL                    Kotlin

attribute x : int          var x = 0
attribute y : double       var y = 0.0
attribute q : String       var q = ''
```

in the case of mapping UML/OCL to Kotlin.

CGBE shows good accuracy of outcomes and low requirements of skills and knowledge for application, whilst LLM code generation approaches exhibit rapid response and flexibility in the code generation task. However, LLM fine tuning for a specific code generation task requires high computation power and large scale datasets, while LLM direct API invoking is less demanding, yet it cannot guarantee accuracy or consistency. CGBE also requires a significant amount of training data, and resources are involved in constructing this data and in iteratively enhancing and extending it in order to attain 100% accuracy for the CGBE-synthesised code generator.

In this paper we investigate AI agents as an alternative AI method for building code generators, which should give high accuracy as well as having low demands on training data and developer expertise. An AI agent uses an LLM in a process to contribute to development productivity, and automates some

*Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026*

qiaomu.1.xue@kcl.ac.uk (Q. Xue); kevin.lano@kcl.ac.uk (K. Lano)

0000-0002-9706-1410 (K. Lano)



© 2026 Copyright (C) for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



published 2026-11-26

CEUR-WS.org/Vol-4338/AgileMDE-paper4.pdf

steps in LLM use, such as dataset refinement, in order to reduce human effort for ML. An AI agent is a complete system for addressing a problem, and the LLM in the AI agent is only one part in the system, used to generate candidate solutions, the other agentic AI parts include tools or a business logic program to carry out actions to complete the given task based on LLM outputs. Compared to a human interacting with an LLM directly, an AI agent can provide a more consistent and automated process for the task. For our goal of code generation production, implemented in an AI agent, the LLM will be used to generate candidate rules for translation from UML/OCL to a target programming language. The LLM will be used once when generating rules, then the other agent steps will apply reasoning and make decisions to build the complete code generator. Although the LLM is pre-trained and could directly answer user instructions to transform or generate code, we want to use the LLM to derive general mapping rules for the code generation task, based on the UML/OCL to target dataset example pairs. The produced rules are deterministic and can be applied without further LLM involvement: this technique addresses the problem that LLM outputs can vary from inference to inference, even when presented with the same inputs. To demonstrate the approach, we construct an OCL to Kotlin code generator through the AI agent method.

## 2. Overall Framework Architecture

The AI agent code generator we will construct is a rule-based AI agent, it will use an LLM to generate rules from the input dataset example pairs, and then it will refine the rules and construct the complete code generator, producing a translator from UML/OCL to the target language.

We use the term 'AI agent' to refer to a system that can iteratively evaluate and update its internal rule set based on evaluation results. We distinguish between: (i) the rule-based code generator (execution/inference layer); (ii) the agent workflow (evaluation and refinement loop); (iii) the overall system (including LLM-based rule induction).

The AI agent uses the powerful pretrained LLM GPT-4o-mini to induce candidate UML/OCL to target programming language rules from examples. The agent takes as input datasets which are comprised of sets of example OCL/code pairs for training, testing and validating, and the LLM will be applied on the data to induce rules, then the generated rules will be used by the engine part to execute the rules as a prototype code generator. The evaluation part of the AI agent will check the performance of coverage and accuracy of the translation, and identify failure cases. After the evaluation, there is a human-in-the-loop element: we use the performance indicators to manually add to or modify the rules. The process may require several rounds of iteration between the engine, evaluation and manual modification of rules until the performance indicators reach to an acceptable level. The final result is an interactive webpage that contains a transformation button to enable input UML/OCL text to be transformed to the target language.

There are two additional parts which we have incorporated: the auto-patch loop and the No-Output extension, to support code generator maintenance and accurate inference. The auto-patch loop addresses the situation when the AI agent code generator is given a new examples dataset where every example has an OCL input and an expected code output. The auto-patch loop can use this new dataset to automatically update the rule set to improve generator accuracy and increase the generator scope. The No-Output extension addresses the situation when the agent transforms new inputs, producing results with low accuracy. This step combines with the auto-patch loop function to enable the AI agent code generator to learn from the new inputs and generate new rules in the rule set to improve accuracy.

## 3. Workflow of Agent-Based Code Generation

In this section, the detailed agentic code generator construction process will be described, using the case of a UML/OCL to Kotlin code generator. For this case the AI agent code generator is built to apply to UML/OCL source specifications, and the data for inducing rules are example corresponding pairs of UML/OCL and Kotlin texts. These pairs are taken from the existing dataset of approximately 300

examples used by CGBE to synthesise a Kotlin code generator according to the process of [1, 2]. This dataset covers the core OCL datatypes and operators from [5], and the core statements (not including exception handling) of AgileUML [6]. The agentic AI method potentially needs a lower amount of data than the CGBE method. Whilst CGBE needs complete examples covering all input variations of each OCL construct to learn source to target mappings, the agent method should only need one example for each source language construct. This is because the LLM already has background knowledge to recognise the structure and content patterns of the source and target languages. In the Kotlin case the data size for the AI agent is about 100 pairs. Before the rules induction step, the data we prepared also needs a manual data cleaning step to ensure that every example is correctly put in the Json format.

### 3.1. Agent code generator construction

The prepared data is then input to gpt-4o-mini and the agent prompts the LLM to generate the rules. The prompt requires that the rules are written in json format and also requires a particular structure for rules. The requirements of rule structure are based on the needs of the agent engine, which uses regex matching, priority scheduling and Jinja2 rendering to generate code from rules. Every rule includes a name, pattern, priority, template, and post_rules part. When the agent starts to translate an input with the rules, the input will be checked against rule patterns by regex matching and variables are then extracted for later Jinja2 rendering. If several rules are matched by regex matching, then one of these rules will be picked by priority, the highest priority rule is the one which will be used to transform the input source to the target language. After Jinja2 rendering, the target programming code is completed to output.

The initial prompt which we designed was as follows, with an example of input data:

```
You are a strict OCL to Kotlin translator.

Rules:
- Output Kotlin code ONLY.
- Do NOT include explanations.
- Output must be exactly ONE line.
- Follow Kotlin conventions strictly.

General:
- Use Kotlin syntax.
- No extra spaces.
- No comments unless explicitly needed.

Translate each line from OCL to Kotlin.
Return the SAME number of lines.
Each output must be ONE line, no explanation.

The OCL is in the input part of the following data:
{"input": "attribute q : double", "output": }
```

Given a batch of inputs, such as a predefined validation/test set, we can calculate the rates of coverage and accuracy of the translation over these inputs, and also identify failed rules (rules that have caused incorrect results). Coverage rate is calculated from two numbers: the number of input samples for which the agent can find rules to translate the sample, and the number of inputs (the number of samples in the test dataset).

$$Coverage\ rate\ = \frac{Number\ of\ samples\ for\ which\ non-null\ output\ is\ produced}{Total\ number\ of\ samples}$$

Coverage rate measures to what extent adequate rules have been generated for the translation.

The accuracy rate is computed from comparison between the translated input and the human defined correct output, for each sample in the validation/test dataset. Before the comparison, the two code texts

will be cleaned by deleting all spaces, which means that it will be a strict comparison between two strings, to test if they are equal to each other.

$$Accuracy\ rate\ =\ \frac{Number\ of\ equal\ comparisons}{Total\ number\ of\ samples}$$

The examples for which equality fails will be printed with the corresponding rules as error cases. In the final step used to optimise the rule set, the error cases will be corrected and added to the rule set to address the LLM limitations for inference. Sometimes the failure case is because of a priority error, sometimes it is due to unstable LLM output, e.g., even if a specific format is required in the prompt, the outcome of rules may include format errors. After modifying the erroneous rules, we check again the coverage, accuracy and error cases, potentially with several rounds of iteration until both coverage and accuracy achieves 100%. The workflow is shown in Figure 1.

The high accuracy rule set will then be adopted and used via the webpage providing interaction. The completed AI agent will use Gradio to launch and run the translation function via the webpage.

### 3.2. Agent code generator refinement and extension

After constructing the basic AI agent code generator, the auto-patch loop step will be used to update the rule set based on new datasets with samples that have inputs and corresponding outputs. In the process, the AI agent code generator will first apply its translation on the source elements of this new dataset, and then evaluate the results against the corresponding target elements to get the failed cases and accuracy. Then it requests the LLM to generate new patch rules based on these failed cases. Because for the failed cases there are also the correct outputs in the new dataset, the LLM knows how to write the patch rules. Then these patch rules will be added in the rule set with high priority to ensure the new rules will always be selected for matching sources. The agent runs an evaluation to check the new rule set performance. If the accuracy compared with the previous accuracy has increased, then the rules will be added in the final rule set and the agent continues the auto-patch loop to generate new rules from any failure cases in this round. The loop will be ended when the updated rule set accuracy does not improve more than a minimum level, and the agent will discard the new rules from this last iteration round, and only keep the rules accepted from previous rounds of the loop. Finally, this process produces an updated rule set for the new dataset. The workflow is shown in Figure 2.

When new inputs transformed by the AI agent translator have low accuracy, the No-Output extension process will run. Because the new inputs do not have their correct outputs already provided, we first build the expected outputs for these new inputs and then run the auto-patch loop to update the rule set to increase the accuracy. The new inputs will be translated via the code generator and the outputs will be scored by heuristics and classified by scores. The results with high scores will be combined with their corresponding inputs as a complete translation example in a new dataset. The corresponding inputs of middle scored and low scored results will be fed to the LLM and these results will be re-generated by the LLM. In this step the LLM tries to generate the expected output of every input. After the LLM generates the expected outputs, the new examples dataset is built, where every example consists of an input and its output generated by the LLM. We check 20% of these samples manually and correct the errors. Then the corrected samples will be merged with the high scored results examples as the new dataset for the AI agent. The auto-patch loop process will then run, because the inputs have expected outputs now. The result is an updated rule set to handle the new inputs.

## 4. Kotlin Agent implementation, evaluation and improvement

As mentioned above, the data for learning UML/OCL to Kotlin translation rules by an LLM is taken from the Kotlin case of CGBE [4]. For the AI agent construction we only keep some examples and delete repeated examples when only one example is needed to fully demonstrate the structure of each element. This should be sufficient because we consider that an LLM has robustness to infer and induce rules based on key examples. The sizes of the training and validation datasets are 91 and 92 examples, respectively.

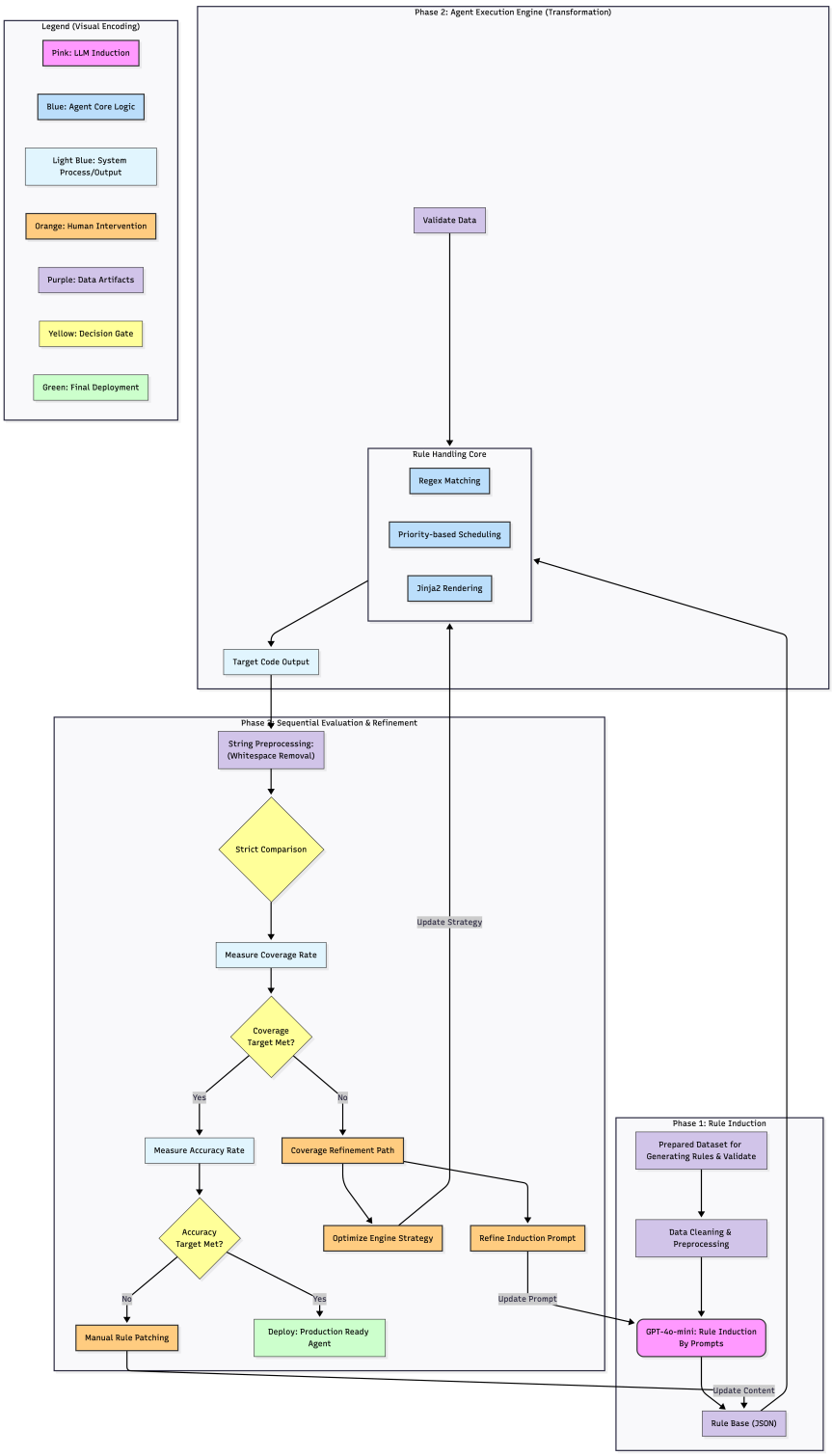



**Figure 1:** Workflow for building AI agent code generator

## 4.1. Kotlin code generator construction

After the first round of rule induction for this translator, there are only 8 rules inferred, and the coverage rate and accuracy are both only 1.2%. The error cases reveal that for most of the inputs there are no matching rules to match the input and produce output, this is caused by the strict matching strategy and the poor LLM performance. The LLM generated rules use an explicit enumeration of cases style to express rules, for example, the attribute declaration rules only cover cases *attribute* $x$. : $T$ whose

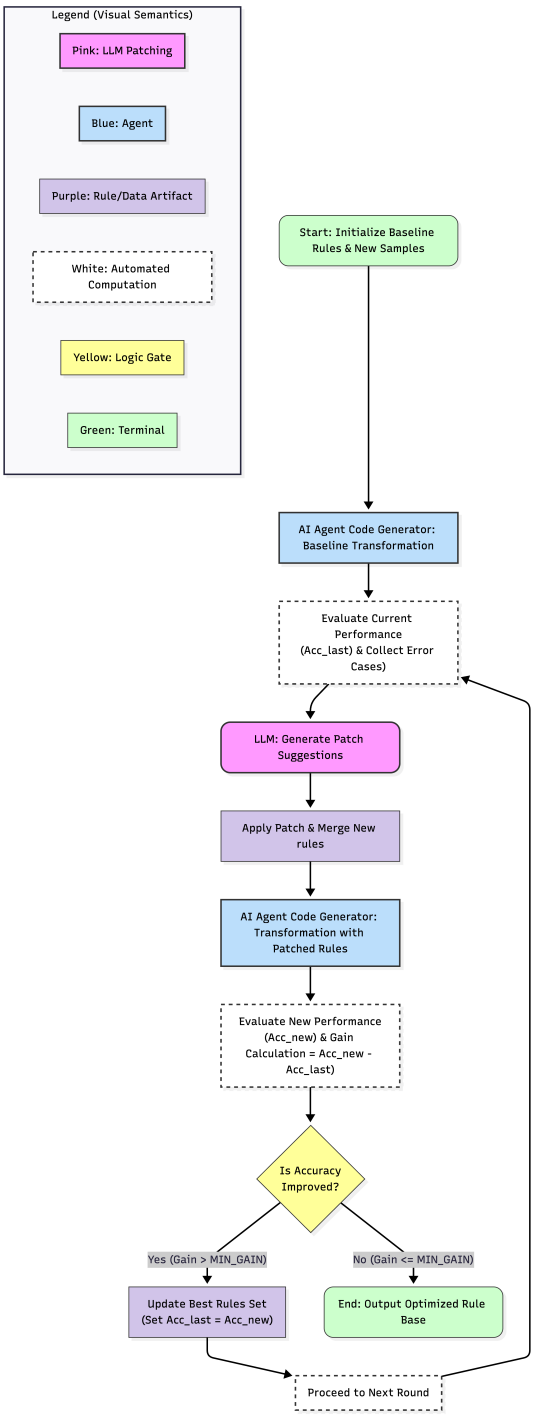



**Figure 2:** Auto-Patch Loop Workflow

type $T$ is included exactly in the rule list, based on the examples, and this cannot cover all situations of possible types. E.g., type *Sequence*(*String*) occurs in the examples, but not other sequence types. The matching of inputs to rules is also restricted to exact regex matching. As for the LLM, producing only 8 rules shows that the LLM needs more detailed prompts to generate useful rules. These three points cause the low coverage rate. To address these problems, we added requirements in the prompts which ask the LLM to output 25 rules at least, and additionally the prompt asks that the LLM must generate rules for all type, declaration, expression and statement constructs included in the training dataset. In the prompts we particularly ask the LLM to follow the rule style which we require, for example, to handle the general case that there are OCL statement blocks ( *statements* ) as the input we tell the LLM

that “OCL blocks in parentheses or braces translate to Kotlin braces { ... }".

After applying these prompt strategies, the coverage rises to 100% and accuracy is improved to over 20%. After applying the auto-patch process for 8 rounds with the validation dataset, the accuracy further rises to 100%, and results in 92 rules in the rules set. Examples of the validation results of the finalised rules are shown in Figure 3.

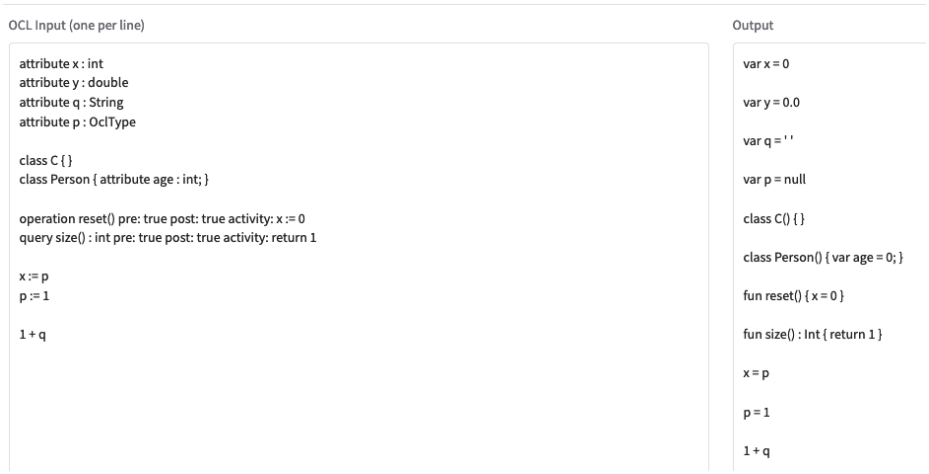



**Figure 3:** Agent validation results

## 4.2. Kotlin code generator refinement

We applied the produced translator to a new test dataset of examples that has 40 pairs of corresponding UML/OCL and Kotlin texts. For this test dataset, the translator produces 17 incorrect answers. The accuracy for the test dataset is therefore only 57.5%.

These results show that the agentic translator has low generalisation ability at this stage. By inspecting the failure cases we identified that the rules produced by the initial generation process overfit for the training and validation examples used to built the translator. In particular the regex based rules have very rigid matching criteria with low generalisation ability: they work only for exact matches or very similar inputs. To address these limitations, we applied the auto-patch loop for the test dataset. After the first round the accuracy increased to 96.2% with 5 wrong answers, and for the second round the accuracy increased to 97% with 4 wrong answers. The third round did not result in any improvements, therefore the loop ended. This loop helps us improve the accuracy to an acceptable level rapidly, however the agentic translator still produces 4 wrong answers.

Although the auto-patch process enables the agentic translator to learn new rules from a new batch of inputs and corresponding target outputs, the problem is that in practical use the agentic translator will always encounter new inputs and there may not be target outputs known for these. Therefore, we created a process, called the No-output extension, to address this situation. The idea of the No-output extension is to use the LLM to generate possible target outputs and to combine the new inputs and these candidate outputs as a new dataset for the auto-patch loop to extend the rules set.

In the first step we input the new UML/OCL examples to the translator, and generate the predicted results for these. Then in the second step, the new samples will be scored by heuristics and separated into three categories: silver, review and fail. The samples in the silver set are considered satisfactory in quality to add into the valid dataset of paired input/output examples for generation of new rules. In the fail set, the samples have obvious errors that cannot be accepted at all and must be addressed. In the review set, the examples are considered not reliable, but not clearly erroneous as the examples in the fail set.

The heuristic score method uses the following rules to score the samples:

1. If there is no output, it reduces the score.

2. If the output has unpaired brackets, reduce the score.
3. If the input is the same as the output, that will add to the score when the input is an atomic identifier, and will reduce the score when the input is a more specific UML/OCL element.
4. If the output retains OCL keywords or special symbols not in the target language (such as $\rightarrow union$), it will reduce the score.
5. If the output has particular errors we recognised previously, reduce the score.
6. If the output has keywords of the target language (e.g., "fun" in Kotlin), add the score.

Score reduction and addition is in units of 1.

The scores are used to classify each sample into their category. Silver cases are those with a positive score. Review cases have zero score and fail cases have negative scores. In the third step, the output of samples in the review and fail sets will be regenerated by the LLM. After generating the new outputs, they will be integrated in the new samples in a draft dataset. In the step four, a human-in-the-loop check verifies that the proposed new rules are acceptable: we check 20% of examples in the merged set of valid draft set and silver set. After checking and removal of invalid examples, the set will be merged with the existing valid dataset for the translator construction. In step 5 we use the entire new valid set to check the translators performance and run the auto-patch rules process if necessary.

The highest final accuracy rate achieved is 98.5%. The remaining flaws can be addressed by manually patching one rule.

## 5. Limitations and Discussion

Based on our expectations before implementation, we considered that the LLM should be able to generate sufficient rules for a complete code generator, however the LLM only directly generates a small proportion of the rules. The LLM increases the flexibility of code generator construction but can produce incorrect results. We improved the initial LLM poor performance by specifying more precise requirements in the LLM prompt.

We made the choice of regex-based rules to implement our rule-based AI agent, these can produce good results, but they are limited to a linear text view of UML/OCL, rather than a grammar/structural view (as in CGBE). For example, a textual scan of the expression

$$x * y + z$$

does not recognise that $+$ is the top-level operator in the expression, i.e., it has the structure $_1 + _2$ in the OCL grammar.

In general the AI agent with our extended construction process can achieve a good accuracy, but usually fails to reach 100% accuracy for new examples. Although for new inputs we defined methods to add rules to maintain the accuracy, this addition of rules affects existing mappings and the overall accuracy was reduced. That may be because with the increased number of rules, some rules cannot be matched because their priority is always lower than another rule (they are shadowed by a higher-priority rule), hence the patched rules only fix part of the results but do not consider the global context in the rule set.

Building an AI agent code generator aims to reduce the human effort and provide a highly automatic process, but in this system construction process, the human interaction was more than we expected. Human effort not only exists in the initial training process, but is also required in the maintenance. For example, there are two checking steps involved for the human to do in the No-Output extension of the process.

### 5.1. Comparison with CGBE and other approaches

With regard to translation accuracy, the CGBE process for the Kotlin case of CGBE can synthesise a code generator with 100% accuracy once the rules have been built for the situations provided in the

examples, so that the transformation is always correct, even when applied to large and complex inputs [1, 2]. But for the Kotlin AI agent, it is not reliable for inference on new inputs. Even after extension steps to add new rules, the accuracy does not reach 100%. Even a 98.5% accuracy is not sufficient for code generation tools in domains (such as critical systems) which have high requirements of accuracy.

The regex rules have a naturally rigid character, and hence are less flexible and provide less coverage compared to CGBE rules. In future work we will investigate alternative forms of rules, such as the concrete syntax transformation language (CSTL) grammar-based rule format used by CGBE [7], for example:

```
if _2 then _4 else _6 endif  |-->if ( _2 ) _4 else _6
```

defines a grammar-based translation of OCL conditional expressions into Kotlin conditional expressions. However, producing this style of rule requires deeper knowledge in the LLM, concerning the grammar structure of the source language, and the CSTL rule format.

With regard to construction effort, building a code generator with CGBE requires significant effort to construct datasets for training and validating. There may also be iteration of the symbolic ML process if the dataset is insufficient to learn all necessary rules.

Compared to the CGBE code generator construction, the effort involved in AI agent code generation focuses on the patch rules. The training dataset for the LLM needs lower effort to prepare. However, depending on the experiment case, there can be much effort spent on the patching of rules to improve the accuracy. In the initial construction stage, manually patched rules are necessary. And in the later stage, if the accuracy is not good enough for new inputs, the No-Output extension strategy for improving the accuracy also needs human intervention to do the checking tasks. Thus the human effort for building this kind of AI agent could be higher overall than for the CGBE method.

With regard to the skill requirements for construction, the developers need to be familiar with the UML/OCL notations and with the target language, in this paper Kotlin, both in building a CGBE code generator or a rule-based AI agent code generator. For CGBE a deeper knowledge of the language grammars is needed. The rule-based AI agent requires extra skills for writing/understanding regex rules and setting the priority of rules in addition to the ability of writing good prompts. In this respect, the AI agent and CGBE methods are similar in terms of skills needed by developers. Table 1 summarises the results and requirements of the different methods for building a UML/OCL to Kotlin code generator, with the data for CGBE, direct LLM invocation and LLM fine-tuning taken from [4]. Accuracy is measured with respect to the same test data set of 40 examples used in this paper.

**Table 1**
Accuracy and requirements of Kotlin code generator construction approaches

| *Approach* | *Accuracy* | *Effort* | *Skills* |
|---|---|---|---|
| CGBE | 97.5% | 2 person days | UML/OCL and target language grammar knowledge, dataset preparation (medium), install ANTLR parsers |
| Fine-tuned LLM | 53.5% | 4 person days | Dataset preparation (high), fine-tuning process, UML/OCL and target language knowledge, prompt design, hyperparameter choice |
| Direct API LLM | 67% | 0.5 person days | Design prompts, invoke model |
| Agentic AI | 98.5% | 3 person days | Dataset preparation (low), prompt design, UML/OCL and target language knowledge, manual steps in learning. |

## 6. Related work

Agentic systems add actions to the processing of a LLM, in order to define a more intelligent and autonomous process. Normally the agent can carry out planning, reflections and memory steps, integrated into a tool to be used in a software process [8]. It embeds an LLM inside the agentic system

tool, and provides several steps to enhance the LLM advantages in understanding, reasoning and utilising long context windows, as well as compensating for the deficiencies of LLMs in terms of hallucinations and other errors in outputs [8].

There are general architectures for building AI agents such as LangChain [9] and AutoGPT [10], which provide support to define a customised agent. There are also pre-built AI agents for software engineering like SWE-agent [11], and CodeAgent [12]. SWE-agent is a fully-functional software engineering agent, designed to autonomously do real development tasks within large-scale repositories. It provides an operation interface for agents to do tasks, enabling LLMs to complete closed-loop operations in a real software engineering environment. This agent on benchmarks such as SWE-bench achieved a 12.5% bug-fix success rate, more than triple that of non-agentic LLMs, and the indicator on humanevalfix achieves 87.7% [11]. CodeAgent is a repository level code generation and software development agent system [12]. It can operate on software projects to replace human developers to understand, edit, run and test software in the project. It integrates multiple tools and a strategy control system together to provide a complete AI agent [12].

AI agents can address the limitations that: 1. LLMs only operate on single files and lack global context; 2. normally LLMs do not have procedures to validate and execute the results of their processing; 3. LLMs cannot address tasks that need several steps of reasoning. Compared to the CodeX and codeT5+ LLM code tools, the AI agent approach has significant improvements [12].

For software development or code generation tasks, an AI agent employs LLMs to give the ability to debug, test, and iterate to achieve the expected results. However, because the agent is closely based on the LLM's performance, it does not always present high reliability, and it is also costly in computation. Although there has been much recent work on the use of LLMs in MDE [13, 14, 15, 16, 17], and some initial work in the use of agentic AI to assist MDE processes [18, 19], we did not find any previous work which specifically considered AI agents as a means of constructing MDE code generators.

## 7. Summary

In this paper we investigated the use of an AI agent method for constructing a UML/OCL to Kotlin code generator. We showed that the approach can be a valid code generation method for this type of task. It can achieve up to 100% accuracy when the examples are small, and it has an accuracy improvement mechanism which can automatically patch rules to extend and improve the code generator. But there remain some issues which should be addressed in the future. The LLM-constructed rules are restricted in form, in particular regex rules are a design trade-off, they lack structural awareness and limit generalisation, but enable rapid rule induction and transparent execution. This work positions regex as a first-step implementation. The AI agent method also requires significant human interaction in some steps. But it has low skills requirements and provides automation of most steps. In the future, we will investigate improved forms of rules and a higher degree of automation to provide a code generator construction method with reduced effort and higher accuracy.

## Declaration on Generative AI

Generative AI (GPT-4o) was used in the research but not in writing the paper.

## References


[1] K. Lano, Q. Xue, Code generation by example using symbolic machine learning, Springer Nature Computer Science (2023).

[2] K. Lano, Q. Xue, H. Haughton, A concrete syntax transformation approach for software language processing, Springer Nature Computer Science (2024).

[3] Q. Xue, K. Lano, Software language translation by example, in: AgileMDE 2024, STAF 2024, 2024.

[4] Q. Xue, K. Lano, Comparing LLM-based and MDE-based code generation for agile MDE, in: AgileMDE 2025, STAF 2025, 2025.

[5] Object Management Group, Object Constraint Language (OCL) 2.4 specification, 2014.
[6] K. Lano, Z. Rahman, L. Alwakeel, Model-based software sustainability analysis and improvement, Software and Systems Modeling (2026).
[7] K. Lano, Q. Xue, Agile specification of code generators for model-driven engineering, in: 2020 15th International Conference on Software Engineering Advances (ICSEA), 2020, pp. 9–15.
[8] Y. Cheng, C. Zhang, Z. Zhang, X. Meng, S. Hong, W. Li, Z. Wang, Z. Wang, F. Yin, J. Zhao, X. He, Exploring large language model based intelligent agents: Definitions, methods, and prospects, 2024. URL: https://arxiv.org/abs/2401.03428. arXiv:2401.03428.
[9] H. Chase, Langchain, https://github.com/langchain-ai/langchain, 2022. Accessed: 2026-01-28.
[10] Significant-Gravitas, Auto-gpt, https://github.com/Significant-Gravitas/AutoGPT, 2023. Accessed: 2026-01-28.
[11] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, N. Jha, SWE-agent: Agent-computer interfaces enable software engineering language models, arXiv preprint arXiv:2405.15793 (2024). URL: https://arxiv.org/abs/2405.15793.
[12] K. Zhang, J. Li, G. Li, X. Shi, Z. Jin, Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges, 2024. URL: https://arxiv.org/abs/2401.07339. arXiv:2401.07339.
[13] A. Boronat, J. Mustafa, MDRE-LLM: A tool for analysing and applying LLMs in software reverse engineering, in: SANER '25, IEEE, 2025.
[14] H. Siala, K. Lano, Towards using LLMs in the reverse engineering of software systems to OCL, in: SANER 2025, 2025.
[15] K. Lano, S. Rahimi, Q. Xue, Sustainability implications of LLM use in MDE, in: SusMod 2025, MODELS, 2025.
[16] A. Sadik, S. Brulin, M. Olhofer, A. Ceravola, F. Joublin, LLM as a code generator in agile model-driven development, arXiv (2024).
[17] H. Siala, K. Lano, A comparison of LLMs and MDRE for reverse engineering, Frontiers in Computer Science (2025).
[18] S. Das, Model and agentic AI-driven middleware for distributed systems design and validation, in: Middleware '25, ACM, 2025.
[19] Z. U. Rizqi, et al., Model-based ai-driven systems engineering (MBSE-AI), in: IEEE International Symposium on Systems Engineering, 2025.