Using LLMs to extract UML class diagrams from Java and Python programs: an empirical study

Hanan Abdulwahab Siala[1], Kevin Lano[1]

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

Abstract

In this paper, we present a comprehensive study of the capabilities of five large language models (LLMs), namely StarCoder2, LLaMA, CodeLlama, Mistral, and DeepSeek, for abstracting UML class diagrams from code, with the aim to provide researchers and developers with insights into the capabilities and limitations of using various LLMs in a model-driven reverse engineering process. We evaluate the LLMs by prompting them to generate UML class diagrams for both Java and Python programs, with the key focus on accuracy, consistency, and F1 score. Our findings reveal that all LLMs have higher accuracy and F1 scores for Python than for Java. DeepSeek and Mistral perform best overall, while LLaMA consistently performs the lowest in all metrics and for both languages.

Keywords

Unified Modeling Language (UML), UML Class Diagram, Model-driven Reverse Engineering (MDRE), Machine Learning, Large Language Models (LLMs), Java programs, Python programs.

1. Introduction

Various organizations have used software systems to carry out their responsibilities. Over time, these systems may become *legacy systems* if they are not adequately maintained and evolved to meet new requirements and needs. *Reverse engineering* aims to understand software systems and facilitate their maintenance and evolution.

Model-Driven Engineering (MDE) [1] can be used together with the reverse engineering process by providing modeling representations and high-level abstractions of software systems. These derived models can then be used with forward engineering to *re-engineer* the software systems or to enable reuse of the abstracted software functions within new MDE developments. It is also possible to use reverse engineering to integrate code and models within a *round-trip engineering* process, whereby developers can work in an agile manner at either modelling or code levels.

The Object Management Group (OMG) has defined open standards for the design and development of software using numerous modelling notations, including the Unified Modeling Language (UML) [2]. UML is the most widely used standard notation for modeling software systems. It includes several diagrams, the most popular of which is the UML class diagram, which is used to represent the classes and relationships in a system.

Large Language Models (LLMs) are a type of machine learning (ML) technology that is initially trained (pre-trained) on massive amounts of textual data, to acquire deep implicit knowledge of the language(s) of the data, including software languages. Once pre-trained, LLMs can be fine-tuned to carry out specific downstream tasks by further training on demonstration examples. Common examples of LLMs include GPT3&4 [3], Bard [4], LLaMA [5], and Mistral [6]. LLMs have had a major impact on various fields, including software engineering, where LLMs have been used in a variety of software engineering tasks [7].

During the last twenty years, a large amount of research has been carried out on abstracting various representations from existing software systems. Modern integrated development environments (IDEs) such as IntelliJ and PyCharm support generating class diagrams through plugins and libraries. These

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025.

hanan.siala@kcl.ac.uk (H. A. Siala); kevin.lano@kcl.ac.uk (K. Lano)
0009-0003-4693-8707 (H. A. Siala); 0000-0002-9706-1410 (K. Lano)

tools often rely on complete, syntactically correct, and fully compilable source code. In contrast, the LLM-based approach offers greater flexibility and automation that can process code and generate structural representations even when the code is incomplete or not compiled.

A recent systematic literature review (SLR) [8] of model-driven reverse engineering (MDRE) approaches over this period identified that using LLMs to abstract different representations from source code is a novel concept. Hence, our research investigates the capabilities and limitations of five open-source LLMs—StarCoder2, LLaMA, CodeLlama, Mistral, and DeepSeek—for abstracting UML class diagrams from both Java and Python programs. The LLMs are evaluated by prompting them to generate UML class diagrams, which are then assessed using accuracy, consistency, and the F1 score metrics. Our goals are:

- To investigate the abstraction of UML class diagrams from Java and Python code using LLMs.
- To compare the generated UML class diagrams with reference models to assess accuracy and completeness.
- To study how LLMs deal with programming languages of different kinds: statically-typed (Java) and dynamically-typed (Python).

The structure of the paper is as follows: section 2 presents related work, while our methodology is explained in section 3. Section 4 illustrates the evaluation of various LLMs. The threats are outlined in section 5, and section 6 provides the conclusions and future work.

2. Related Work

Here we describe related work in the areas of model-driven reverse engineering approaches and the generation of code representations with LLMs.

2.1. Model-driven Reverse Engineering Approaches

Raibulet *et al.* [9] present a comprehensive analysis of MDRE approaches in the literature from 2003 up to 2017. They compared fifteen MDRE approaches and presented their different features, such as the level of automation, extensibility, and genericity.

Siala *et al.* [8] provide an SLR of MDRE over the period 2000–2023, which surveys 55 distinct MDRE approaches. They found that the majority of MDRE research has concentrated on developing code visualisations such as class, sequence, and activity diagrams.

2.2. Generating Code Representations with LLMs

Boronat and Mustafa [10] introduce a tool, MDRE-LLM, that integrates LLMs with MDRE to automate and improve domain model recovery from source code. The tool supports diverse use cases, including analyzing undocumented legacy systems, understanding large-scale codebases, validating LLM performance, and generating reproducible datasets. They use retrieval augmented generation (RAG) to improve the accuracy and relevance of LLM responses.

Siala [11] introduces a new approach for using LLMs to abstract UML class diagrams and object constraint language (OCL) from Java and Python programs. In [12], Siala and Lano present the LLM4Models LLM, based on the fine-tuning of the Mistral LLM, which abstracts OCL specifications from Java and Python programs. In [13], the LLM4Models LLM is also used to abstract UML class diagrams from Java programs, while the LLM4Models LLM is used in [14] to abstract UML class diagrams from Python programs. The evaluation results of these papers indicate that the LLM4Models LLM can effectively abstract UML and OCL from Java and Python programs.

3. Methodology

This section details the methodology used in our study, including dataset creation, selection of LLMs, definition of UML abstraction criteria, prompt formulation, and the evaluation metrics employed.

3.1. Dataset Creation

We selected Java and Python cases from established and large-scale program datasets: CoTran [15] and AVATAR [16].

1. Collecting sample programs: We selected 14 Java programs and 16 Python programs from the datasets for our experiment to analyze the reverse-engineering performance of various LLMs on Java and Python. When choosing these programs, we were not concerned with the length of the programs, but rather with representing all the possible elements and all relationships to test the capability of the abstraction process. The selected examples cover all the program elements considered by our evaluation: interfaces, classes, various attributes and methods, visibility, inner classes, and abstract and static attributes and methods. The examples include cases with one to six classes and with various relationships.
2. Automatically generating ground-truth UML class diagrams for these programs: We used the Java2JSON [13] and Python2JSON [14] parsers and rulesets to precisely abstract expected UML models from the selected programs to serve as reference models for comparison with the LLM-generated UML models.

3.2. Model Selection

To investigate the capability of existing pre-trained LLMs to abstract UML class diagrams from Java and Python programs, we experiment using five open-source LLMs, all from Hugging Face [17], which have achieved promising results in various code-related tasks. We selected LLMs with parameters from 7B to 8B to maintain a balance between performance and accessibility. Larger models require significantly more computational resources and inference time, while models in the 7B-8B range can run efficiently on commonly available hardware while still providing sufficient capabilities and high-quality output [18].

1. **StarCoder2**: StarCoder2 [19] is a family of LLMs for code (Code LLMs) trained on 3.3 to 4.3 trillion tokens and evaluated on various Code LLM benchmarks. The smallest model (3B) and the largest model (15B) outperform comparable models [19]. StarCoder2-7B is used in our experiment.
2. **LLaMA**: LLaMA is a collection of LLMs in 8B, 70B, and 405B sizes trained using up to 15 trillion tokens of publicly accessible data from different sources. Our experiment uses Llama-3.1-8B, which was released on July 23, 2024.
3. **CodeLlama**: CodeLlama [20] is a code LLM based on the LLaMA 2 architecture. A large dataset of code and natural language related to code has been used to train CodeLlama to support code generation and infilling tasks in several programming languages, including Java, Python, C#, PHP, and C++. CodeLlama-7b-hf is used in our experiment.
4. **Mistral**: Mistral [6] is a decoder-only transformer. It performs well in natural language understanding and generation when compared to other LLMs, and it may also be utilized for code-related tasks. Mistral-7B-v0.3 is used in our experiment.
5. **DeepSeek**: DeepSeek-R1 [21] is trained through extensive reinforcement learning (RL) to tackle the challenges associated with DeepSeek-R1-Zero. Multi-stage training and cold-start data are integrated before RL. Additionally, DeepSeek is distilled into smaller, dense models based on Qwen and Llama, which deliver outstanding performance on benchmarks. DeepSeek-R1-Distill-Llama-8B is used in our experiment.

A temperature hyperparameter of 0.2 was chosen for the selected LLMs. This limits variation in responses, increases deterministic output, and maintains the ability for diverse responses.

3.3. Identifying Criteria for UML Class Diagram abstraction

The following criteria are used to evaluate the capabilities of various LLMs to abstract UML class diagram representations from Java and Python programs and to identify any potential shortcomings:

C1: Are all classes and interfaces in the reference class diagram correctly abstracted by the LLM?
C2: Are all the attributes and their types in the reference class diagram correctly abstracted?
C3: Are all the methods in the reference class diagram correctly abstracted?
C4: Is each expected inheritance relationship between classes generated in the LLM output?
C5: Is each expected realization relationship between classes/interfaces generated in the LLM output?
C6: Is each correct association relationship between classes generated in the LLM output?
C7: Are correct aggregation/composition relationships between classes included in the generated output?
C8: Are correct abstract classes and abstract methods identified in the generated output?
C9: Are the correct static classes and static methods included in the generated output?
C10: Are correct association multiplicities included in the generated output?
C11: Are correct association role names included in the generated output?
C12: Are correct dependency relationships between classes, identified by examining method parameter types, local variables, or return types, included in the generated output?

Although, in principle, these properties could be automatically checked by comparing the reference and generated UML models, in practice, we found this was not possible due to the variations in the model format produced by the LLMs. The generated output often includes additional text and varies between LLMs and individual LLMs. Thus, we used a manual comparison of the reference and generated models to check the above criteria.

Furthermore, we could not rely only on automatic checking in many circumstances. For example, distinguishing between association, aggregation, and composition relationships is challenging for automated checking. We could not omit an association if it is identified instead of an aggregation relationship, as aggregation is a special form of association and could be overlooked completely by automatic comparison.

The variation in the model format generated by the LLMs can be seen in the processing of Java example 3. Consider the class VitalSigns as an example of this variation in response, where it inherits from the class MedicalData. CodeLama LLM defines the relationship within the class blocks and generates the output shown in Fig. 1a.

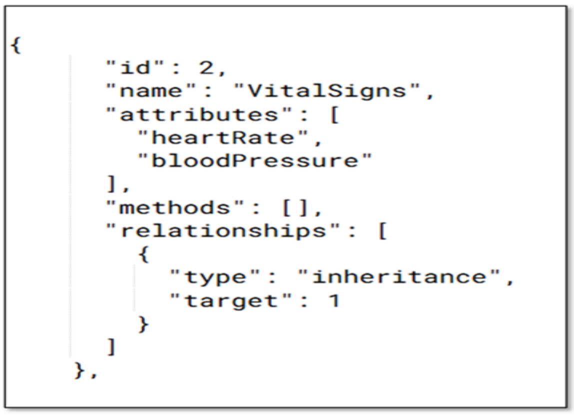
It assigns a unique numerical *id* to each class, allowing references to that class in other classes. For example, the field target refers to class number 1, which corresponds to the MedicalData class. In contrast, the DeepSeek LLM defines all classes, including the VitalSigns class, shown in Fig. 1b, and then it generates all the relationships together, as shown in Fig. 1c.

Likewise, the other LLMs each have their own variations on the result format. For example, Fig. 2a shows a Python example that contains a composition relationship between classes. While both StarCoder2 and Mistral LLMs abstract attributes and methods for the given example, Mistral correctly identifies the composition relationship, as shown in Fig. 2b, whereas StarCoder2 instead abstracts a dependency relationship, as shown in Fig. 2c.

3.4. Prompt Engineering for UML Class Diagram abstraction

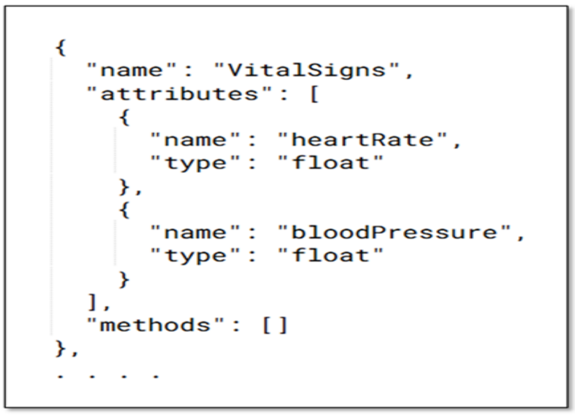
We use an Alapaca-style prompt format. The prompt schema used to abstract the UML class diagram for Java input is shown in Fig. 3. This prompt was engineered to obtain the elements and relationships of UML class diagrams, with the required quality properties of non-redundancy and non-duplication. We found that this version produced the most accurate and consistent results. A corresponding prompt was applied for the Python abstraction.

We requested each LLM to generate output in JSON format, from which the abstracted information could be graphically converted into UML class diagrams using libraries like Graphviz or PlantUML.



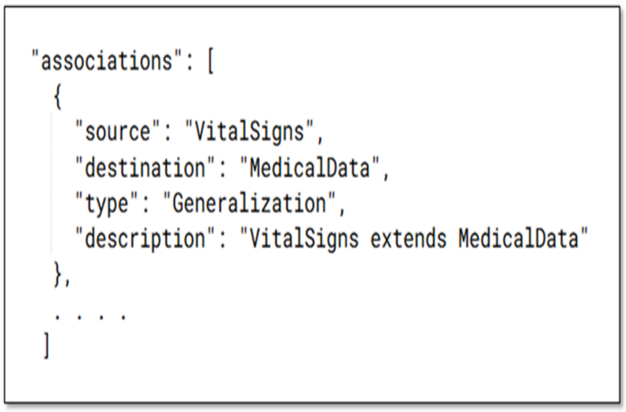
```
{
        "id": 2,
        "name": "VitalSigns",
        "attributes": [
          "heartRate",
          "bloodPressure"
        ],
        "methods": [],
        "relationships": [
          {
            "type": "inheritance",
            "target": 1
          }
        ]
      },
```

(a) Output (elements and relationships) of the class diagram generated by CodeLlama LLM.



```
{
  "name": "VitalSigns",
  "attributes": [
    {
      "name": "heartRate",
      "type": "float"
    },
    {
      "name": "bloodPressure",
      "type": "float"
    }
  ],
  "methods": []
},
. . . .
```

(b) Elements of the class diagram generated by DeepSeek LLM.



```
"associations": [
  {
    "source": "VitalSigns",
    "destination": "MedicalData",
    "type": "Generalization",
    "description": "VitalSigns extends MedicalData"
  },
  . . . .
]
```

(c) Relationships of the class diagram generated by DeepSeek LLM.

Figure 1: Outputs generated by CodeLlama and DeepSeek LLMs.

3.5. Evaluation Metrics

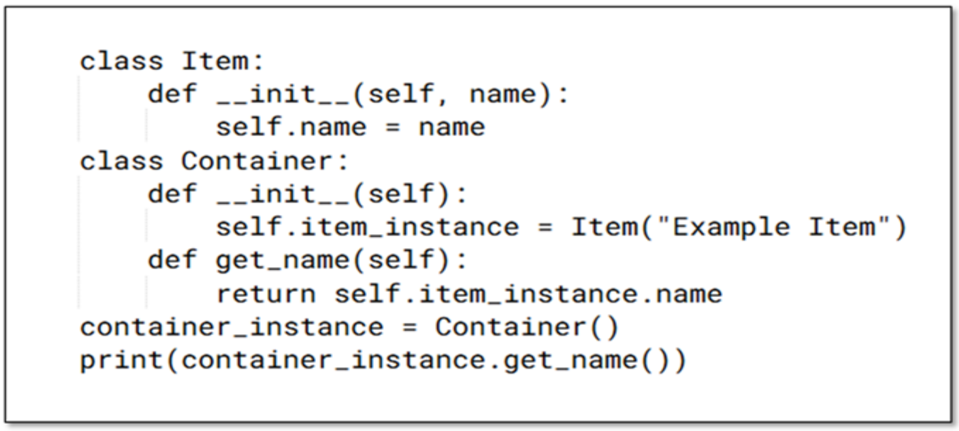
The evaluation was carried out based on three key metrics: accuracy, consistency, and F1 score. We first evaluate the *accuracy* of each LLM, where accuracy is defined as the proportion of source code elements that are accurately represented as UML class diagram elements. Accuracy, also referred to as *recall*, is calculated using Equation (1), where a true positive (TP) is an element accurately translated from code to UML class diagram elements, and a false negative (FN) is an element that is not translated or is translated inaccurately.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{1}$$

Next, we evaluate the *consistency* of each LLM, where consistency is defined as the proportion of elements in the generated UML class diagram that are accurately derived from the source programs. Consistency is important for ensuring traceability and alignment of the derived class diagram element concerning the source code. Consistency, also referred to as *precision*, is calculated using Equation (2), where a false positive (FP) is an element that appears in the UML class diagram that is not correctly derived from a source code element.

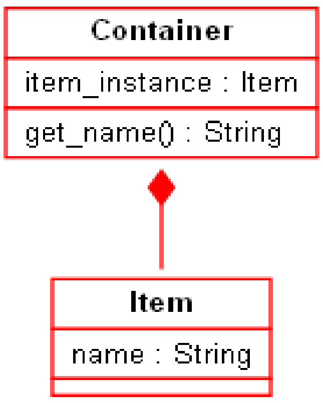
$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

Accuracy evaluates the completeness and correctness of the abstraction process, while consistency evaluates the quality of the generated UML class diagram element in terms of the absence of spurious

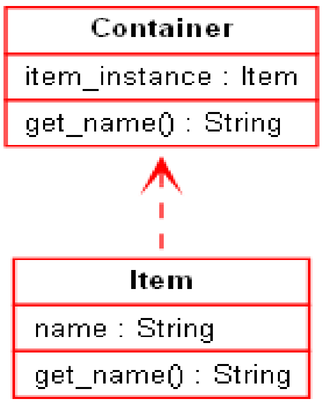


```python
class Item:
    def __init__(self, name):
        self.name = name
class Container:
    def __init__(self):
        self.item_instance = Item("Example Item")
    def get_name(self):
        return self.item_instance.name
container_instance = Container()
print(container_instance.get_name())
```

(a) Python example containing a composition relationship.

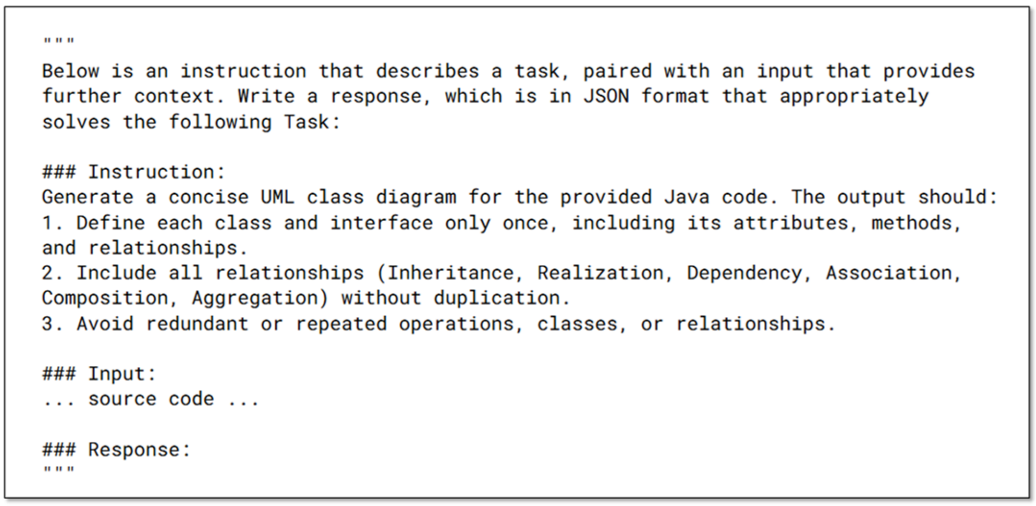


(b) Class diagram generated by Mistral for the given Python example.



(c) Class diagram generated by StarCoder2 for the given Python example.

Figure 2: Python example and UML class diagrams generated by Mistral and StarCoder2 LLMs.



```
"""
Below is an instruction that describes a task, paired with an input that provides
further context. Write a response, which is in JSON format that appropriately
solves the following Task:

### Instruction:
Generate a concise UML class diagram for the provided Java code. The output should:
1. Define each class and interface only once, including its attributes, methods,
and relationships.
2. Include all relationships (Inheritance, Realization, Dependency, Association,
Composition, Aggregation) without duplication.
3. Avoid redundant or repeated operations, classes, or relationships.

### Input:
... source code ...

### Response:
"""
```

Figure 3: Prompt schema to generate UML class diagrams.

elements not derived from the source code.

Note that we do not consider any extra elements found outside the generated UML class diagrams. Any extra explanations or examples provided by the LLM do not affect consistency. Consistency is affected only by extra elements that are part of the generated UML class diagram and are not present in the Java or Python source code.

The F1 score provides a balance between precision and recall, as shown in Equation (3). It considers both false positives and false negatives and can be especially beneficial when the LLM produces conflicting results; for example, high precision but low recall, or vice versa.

$$\text{F1 score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

4. Results and Discussion

In this section, we present a comprehensive analysis of the performance of the LLMs in terms of accuracy, consistency, and F1 score.

4.1. Evaluation of LLMs to Generate UML Class Diagrams from Java Programs

Accuracy Fig. 4a presents the accuracy of various LLMs in abstracting UML class diagrams from 14 Java examples. DeepSeek, Mistral, and StarCoder2 achieve higher peaks, with accuracy scores reaching 1.0 for certain examples. However, StarCoder2 has the lowest accuracy for both examples one and twelve, while Mistral has the lowest accuracy score for example five. CodeLlama follows them in its accuracy scores. Although the accuracy scores of the LLaMA LLM are generally lower compared to other LLMs, these scores are also relatively stable across different examples.

Consistency Fig. 4b shows an overview of the consistency scores for abstracting UML class diagrams from Java programs. Overall, Mistral has higher consistency scores compared to the other LLMs, followed by both StarCoder2 and DeepSeek LLMs. Both LLaMa and CodeLlama show moderate consistency, with consistency scores hovering around 0.6.

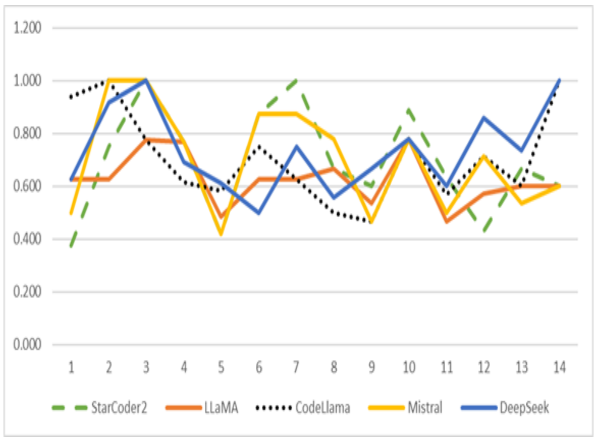
4.2. Evaluation of LLMs to Generate UML Class Diagrams from Python Programs

Accuracy Fig. 4c shows the accuracy of the selected LLMs in abstracting UML class diagrams from 16 Python examples. Both DeepSeek and Mistral achieve higher accuracy scores for abstracting UML class diagrams from Python examples. StarCoder2 follows DeepSeek and Mistral in accuracy scores, while CodeLlama tends to stay stable between 0.8 and 1.0 scores. However, LLaMA shows a significant drop in example 10, where accuracy dips to slightly above 0.2.

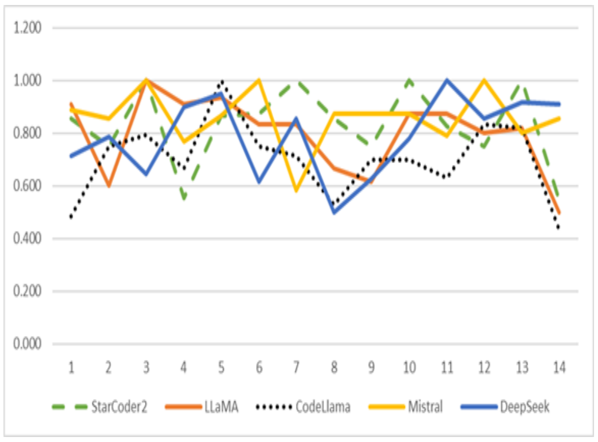
Consistency Fig. 4d presents the consistency scores for the selected LLMs in generating UML class diagrams for Python code. Apart from a drop in performance, for example, in Python case number two, DeepSeek has in general higher consistency scores compared to other LLMs. StarCoder2 and Mistral follow DeepSeek, maintaining generally consistent performance, with a slight decline observed in two examples for each. Meanwhile, LLaMA shows marked inconsistency, with multiple drops, particularly in example 16, whereas CodeLlama performs slightly better than LLama but experiences a significant drop in example 6.

4.3. Overall Results

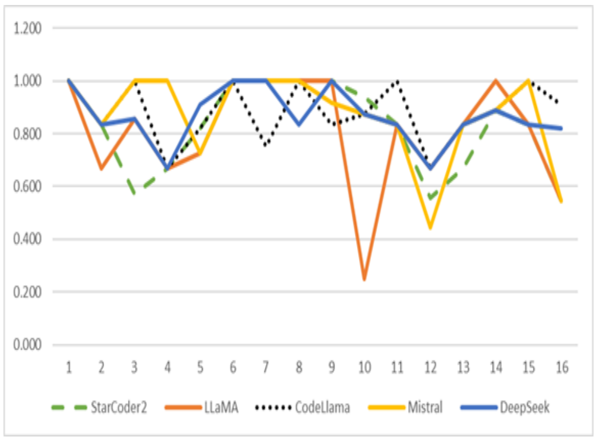
Accuracy Fig. 5a presents the overall accuracy for UML class diagrams generated by the selected LLMs. The results show that all LLMs have more accuracy for Python than for Java. This could seem to be a surprising result because Java programs usually contain more precise typing information about



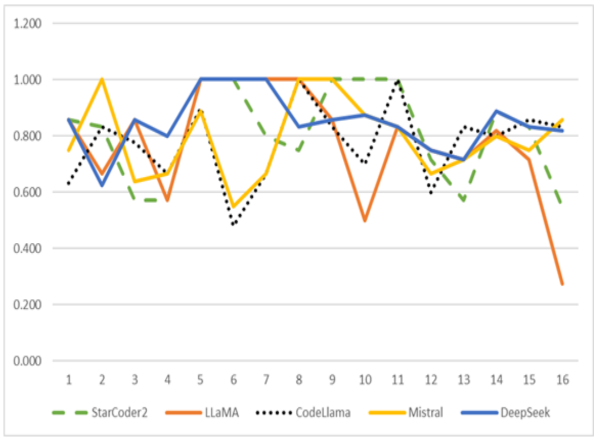
(a) Accuracy for Java examples, numbered 1 to 14.



(b) Consistency for Java examples, numbered 1 to 14.



(c) Accuracy for Python examples, numbered 1 to 16.



(d) Consistency for Python examples, numbered 1 to 16.

Figure 4: Accuracy and consistency results for Java and Python examples.

program features and variables compared to Python programs. On the other hand, Java programs tend to be more structurally complex than Python programs.

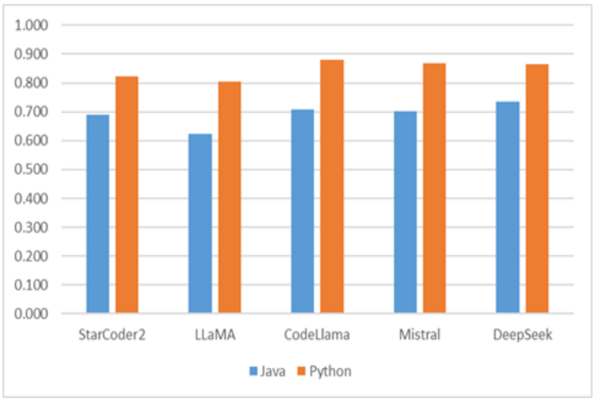
DeepSeek has the highest accuracy score for Java, while both Mistral and DeepSeek share the highest accuracy for Python. By contrast, LLaMA has the lowest accuracy scores in both languages. In addition, the difference in accuracy between the two languages is most noticeable for LLaMA and CodeLlama.

Consistency Fig. 5b presents the overall consistency generated by the selected LLMs. Mistral achieves the highest consistency for Java, while DeepSeek has the highest consistency score. In comparison, CodeLlama and LLaMA share the lowest consistency for Python, whereas CodeLlama has the lowest consistency score for Java. Additionally, the consistency differences between Java and Python are minor compared to the accuracy differences.

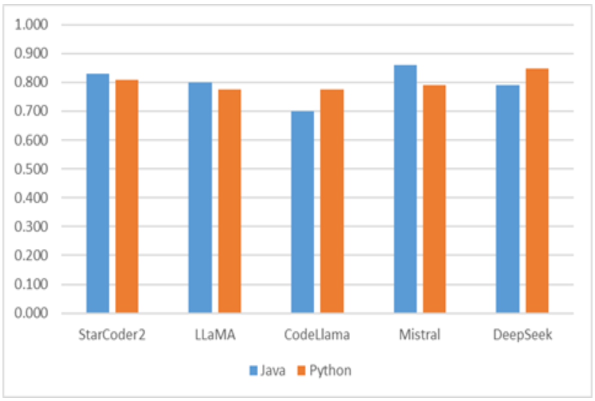
F1 score The analysis of Python outperforms that of Java across all LLMs, according to the F1 score, with DeepSeek achieving the highest F1 score for Python and Mistral achieving the highest value for Java. The F1 scores for Mistral and CodeLlama closely follow those of DeepSeek for Python. In contrast, LLaMA has the lowest F1 score in both languages, as shown in Fig. 5c.

5. Threats to Validity

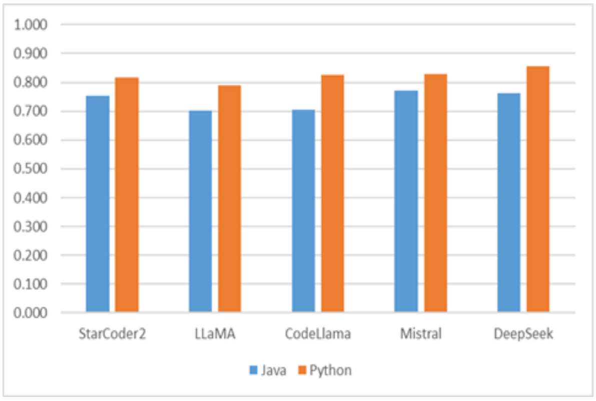
In this section, we outline potential threats that could impact the findings of our experimental study.



(a) Overall accuracy for Java and Python examples.



(b) Overall consistency for Java and Python examples.



(c) Overall F1 score for Java and Python examples.

Figure 5: Overall accuracy, consistency, and F1 score.

5.1. Response of models

There are cases where we receive responses from LLMs unrelated to UML class diagrams. We mitigate this by performing multiple (up to 5 attempts) iterations of the inference process. This approach allowed us to enhance the results and select the most accurate class diagram generated by each LLM.

5.1.1. Evaluation of the responses

Human error and bias can occur when evaluating manually generated UML class diagrams. This manual evaluation process may produce inconsistent or inaccurate results. However, this is mitigated by involving the second author to check cases where the first author is unsure at any stage in the experiment.

As we noted in subsection 3.3 above, due to variability in the LLM result formats, it was not possible to perform an automated comparison of the reference and generated UML models.

5.1.2. Scope of the experiment

Here, we have only used open-source LLMs, and there is a risk that the most powerful LLMs are not considered. However, this risk is partially resolved by including the most powerful open-source LLMs. DeepSeek is the latest powerful open-source LLM included in our experiment.

5.2. Representativeness

We aimed to consider real-world Java and Python programs, typical of those created by practitioners. Thus, we chose cases from well-known and established datasets of such programs, which have been used in other LLM research. We selected cases to cover all the modelling aspects that should be represented in generated class diagrams.

6. Conclusion and Future Work

This paper conducted an empirical study of using LLMs to abstract UML class diagrams from Java and Python code. We found that DeepSeek LLM and Mistral are the most reliable LLMs in abstracting UML class diagrams for both Java and Python. Generally, LLMs generate better results with Python than with Java, which is contrary to expectations. In future work, we will aim to expand our work to include other programming languages like C++, C#, COBOL, and others. We also aim to fine-tune an open-source LLM (e.g., Mistral or DeepSeek) on a large-scale dataset of these programming languages with their corresponding class diagram representations to improve the performance of LLMs in reverse engineering tasks.

7. Data Availability

The data used in this study - including Java and Python programs and their corresponding UML class diagrams generated by the selected open-source LLMs - are available in our online repository [22].

Acknowledgment

Hanan Siala acknowledges the financial support provided by the Libyan Ministry of Higher Education and Scientific Research. She also acknowledges the use of resources provided by King's Computational Research, Engineering and Technology Environment (CREATE).

Declaration on Generative AI

The authors used MISTRAL to extract UML class diagrams from program code. All outputs were reviewed and validated by the authors, who take full responsibility. No proprietary/third-party confidential data were provided to these tools.

References

[1] S. Kent, Model driven engineering, in: Integrated Formal Methods: Third International Conference, IFM 2002 Turku, Finland, May 15–18, 2002 Proceedings, Springer, 2002, pp. 286–298. doi:`https://doi.org/10.1007/3-540-47884-1_16`.

[2] OMG, OMG Systems Modeling Language (UML), Version 2.5.1, 2017. https://www.omg.org/spec/UML.

[3] W. Zhao, et al., A survey of large language models, 2025. URL: https://arxiv.org/abs/2303.18223.

[4] Google, Bard, 2025. URL: https://bard.google.com/, accessed: [04-03-2025].

[5] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, G. Lample, Llama: Open and efficient foundation language models, 2023. URL: https://arxiv.org/abs/2302.13971. `arXiv:2302.13971`.

[6] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b, arXiv preprint arXiv:2310.06825 (2023). URL: https://arxiv.org/abs/2310.06825. `arXiv:2310.06825`.

[7] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, LLMs for software engineering: a systematic literature review, arXiv 2308.10620 (2023).
[8] H. A. Siala, K. Lano, H. Alfraihi, Model-driven approaches for reverse engineering – a systematic literature review, IEEE Access (2024). doi:10.1109/ACCESS.2024.3394732.
[9] C. Raibulet, F. A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: A systematic literature review, Ieee Access 5 (2017) 14516–14542. doi:10.1109/ACCESS.2017.2733518.
[10] A. Boronat, J. Mustafa, MDRE-LLM: A tool for analyzing and applying LLMs in software reverse engineering, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2025. URL: https://conf.researchr.org/home/saner-2025.
[11] H. A. Siala, Enhancing model-driven reverse engineering using machine learning, in: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE-Companion '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 173–175. URL: https://doi.org/10.1145/3639478.3639797. doi:10.1145/3639478.3639797.
[12] H. A. Siala, K. Lano, Towards using LLMs in the reverse engineering of software systems to object constraint language, in: Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2025. URL: https://conf.researchr.org/home/saner-2025.
[13] H. A. Siala, K. Lano, Using large language models to extract UML class diagrams from Java programs, in: Proceedings of the International Conference on Software and System Engineering (ICoSSE), 2025. URL: http://www.icsse.org/.
[14] H. A. Siala, K. Lano, Leveraging large language models for abstracting UML class diagrams from Python programs, 2025. Under review.
[15] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, V. Ganesh, CoTran: An LLM-based code translator using reinforcement learning with feedback from compiler and symbolic execution, IOS Press, 2024. URL: http://dx.doi.org/10.3233/FAIA240968. doi:10.3233/faia240968.
[16] W. Ahmad, M. Tushar, S. Chakraborty, K.-W. Chang, AVATAR: a parallel corpus for Java-Python program translation, in: Annual Meeting of the Association for Computational Linguistics, 2021. URL: https://api.semanticscholar.org/CorpusID:237304035.
[17] Hugging Face, Hugging face, Online, 2016. Available at: https://huggingface.co/ [Accessed Mar. 2025].
[18] M. Hassid, T. Remez, J. Gehring, R. Schwartz, Y. Adi, The larger the better? improved LLM code-generation via budget reallocation, 2024. URL: https://arxiv.org/abs/2404.00725. arXiv:2404.00725.
[19] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, et al., StarCoder 2 and the stack v2: The next generation, 2024. URL: https://arxiv.org/abs/2402.19173. arXiv:2402.19173.
[20] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al., Code llama: Open foundation models for code, 2024. URL: https://arxiv.org/abs/2308.12950. arXiv:2308.12950.
[21] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, et al., DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. URL: https://arxiv.org/abs/2501.12948. arXiv:2501.12948.
[22] H. A. Siala, K. Lano, Online repository for Java and Python programs with UML diagrams, https://doi.org/10.5281/zenodo.15108621, 2025. Accessed: May 2025.