Bachelor's Thesis

Enhancing UML Class Diagram Completion with Fine-Tuned Large Language Models

Kamila Iusupova

January 20, 2026

Advisor:
Alisa Welter Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Isabel Valera Department of Computer Science

Chair of Software Engineering
Saarland Informatics Campus
Saarland University





UNIVERSITÄT
DES
SAARLANDES

Kamila Iusupova: *Enhancing UML Class Diagram Completion with Fine-Tuned Large Language Models,* © January 2026

Erklärung Statement

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich bei der Erstellung der nachfolgenden Arbeit mittels künstlicher Intelligenz betriebene Software (z. B. ChatGPT) ausschließlich zur Textüberarbeitung/-korrektur und zur Code-Vervollständigung und nicht zur Bearbeitung der in der Arbeit aufgeworfenen Fragestellungen zu Hilfe genommen habe. Alle mittels künstlicher Intelligenz betriebenen Software (z. B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit wurden kenntlich gemacht und als Hilfsmittel angegeben. Ich erkläre mich damit einverstanden, dass die Arbeit mittels eines Plagiatsprogrammes auf die Nutzung einer solchen Software überprüft wird. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

Talk to your advisor and check whether this applies to you exactly this way

I hereby declare that I have written this thesis independently and without the involvement of third parties, and that I have used no sources or aids other than those indicated. All passages taken directly or indirectly from publications or other external sources have been identified as such. In particular, I confirm that I have used AI-based software (e.g., ChatGPT) exclusively for the following permitted sub-tasks: text rewriting/revision and code completion , and not to address or formulate the main research questions of the thesis. All parts of the thesis that were generated and/or edited using AI-based software (e.g., ChatGPT) have been disclosed and documented in accordance with the documentation requirements. I agree that the thesis may be checked using plagiarism detection software, including checks for the use of such software. I am aware that any violation of this declaration may result in failing the examination and lead to losing the right to be examined.

Talk to your advisor and check whether this applies to you exactly this way

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Einverständniserklärung (optional) Declaration of Consent (optional)

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____
(Datum Date) (Unterschrift Signature)

Abstract

Unified Modeling Language (UML) class diagrams play a central role in software design and development, but their manual creation is time-consuming and error-prone, especially for large and complex systems. The recent advancements in Large Language Models (LLMs) have prompted interest in their potential application to software engineering tasks, including model completion.

In this work, we investigate fine-tuning LLMs for automated UML class diagram completion. We examine whether incorporating non-modeling data and natural language (Natural Language (NL)) comments during fine-tuning affects the model's ability to reconstruct missing diagram elements. Using a controlled experimental setup, multiple fine-tuned LLaMA-based model variants are compared against a UML-only baseline using structurally oriented evaluation metrics.

The obtained results indicate that augmenting training data with non-modeling information, such as natural language text or source code, has limited and context-dependent impact in the domain of software modeling. Although moderate amounts of domain-adjacent data, such as software engineering text or source code, can lead to small and selective improvements, these effects do not show consistent statistical significance. In contrast, the inclusion of high-level natural language comments does not improve performance under the investigated conditions and may introduce additional ambiguity. These findings highlight both the potential and the limitations of data augmentation strategies for LLM-based UML class diagram completion.

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor, Alisa Welter for her continuous guidance, valuable feedback, insights, and constructive discussions. Her expertise, patience, and encouragement helped me navigate both the technical and conceptual challenges of the research process. I am especially grateful for the trust placed in me to explore this topic independently, while always providing thoughtful direction and feedback when needed.

Furthermore, I would like to thank Professor Sven Apel for providing me the opportunity to conduct this research for fostering an academic environment that encouraged independent thinking and exploration. I am grateful for his valuable feedback and for enabling this work within the broader context of software engineering research.

I would also like to thank my family for their unwavering support throughout my studies. Their constant encouragement, understanding, and belief in me gave me the strength and confidence to pursue my academic goals, even during the most challenging phases of this journey. I am deeply grateful for their love, patience, unconditional support, and for always being there when motivation, reassurance, or simply a kind word was needed.

Finally, I would like to thank my close friends for their support, understanding, countless conversations and moments we shared along the way. Whether through technical discussions, shared moments of stress, or much-needed distractions, they helped me stay grounded, cherished my small accomplishments, and reminded me to enjoy the process as well. Their presence made this journey not only manageable, but truly meaningful and memorable.

Contents

1 Introduction 1
1.1 Goal of this Thesis . . . 3
1.2 Overview . . . 3
2 Background 5
2.1 Unified Modeling Language (UML) and Software Modeling . . . 5
2.1.1 Types of UML diagrams . . . 5
2.1.2 UML in software engineering . . . 7
2.1.3 Model-Driven Software Engineering . . . 7
2.2 The Large Language Models (LLMs) . . . 7
2.2.1 Main concepts in LLMs . . . 8
2.2.2 Transformers and their architecture . . . 9
2.3 Fine-Tuning of LLMs . . . 11
2.3.1 Full Fine-Tuning . . . 11
2.3.2 Parameter-Efficient Fine-Tuning . . . 11
2.3.3 Low-Rank Adaptation . . . 11
2.3.4 Quantized Low-Rank Adaptation . . . 12
2.3.5 Prompt Engineering . . . 13
2.3.6 In-Context Learning . . . 13
2.3.7 Fine-tuning vs. In-Context Learning, Promt Engineering . . . 13
2.4 Definition of Structural and Semantic Correctness . . . 14
2.5 Definition of Model Performance . . . 14
2.6 Definition of Base Model . . . 14
3 Methodology 15
3.1 Research Questions . . . 15
3.2 Research Design . . . 17
3.2.1 Overall Experimental Pipeline . . . 17
3.2.2 Datasets . . . 18
3.2.3 Data Preparation Pipeline . . . 18
3.2.4 LLMs . . . 21
3.2.5 Model Notation . . . 21
3.2.6 Experimental Conditions . . . 22
3.3 Software and Hardware . . . 26
3.3.1 Software . . . 26
3.3.2 Hardware . . . 26
3.4 Metrics . . . 26
3.4.1 Model Completion Accuracy . . . 27
3.4.2 Semantic Similarity Metrics: BLEU, Cosine Similarity . . . 28

3.4.3 Completion and Recovery Rates . . . 28
3.4.4 Relative Performance Improvement . . . 30
4 Evaluation 31
4.1 Experiments . . . 31
4.1.1 Experiment RQ1: External Knowledge Augmentation . . . 31
4.1.2 Experiment RQ2: Integration of Natural Language Comments . . . 34
4.2 Results . . . 36
4.2.1 Results RQ1: External Knowledge Augmentation . . . 36
4.2.2 Results RQ2: Integration of Natural Language Comments . . . 46
4.3 Discussion . . . 51
4.3.1 Discussion RQ1: External Knowledge Augmentation . . . 51
4.3.2 Discussion RQ2: Integration of Natural Language Comments . . . 56
4.4 Threats to Validity . . . 59
4.4.1 Threats to Internal Validity . . . 59
4.4.2 Threats to External Validity . . . 60
5 Limitations and Future Work 63
5.1 Limitations . . . 63
5.2 Future Work . . . 64
6 Related Work 67
6.1 LLMs for Structured Code Understanding and Generation . . . 67
6.2 Augmenting LLMs with External Knowledge and NL Comments . . . 69
6.3 LLMs in Software Modeling and UML Completion . . . 70
7 Concluding Remarks 73
7.1 Conclusion . . . 73
7.2 Code and Reproducibility . . . 73
A Appendix 75
A.1 Types of UML diagrams. . . . 75
A.2 Pilot Study . . . 75

Statement on the Usage of Generative Digital Assistants 79

Bibliography 81

List of Figures

Figure 2.1 An example of a simple class diagram. . . . 6
Figure 2.2 An example of a simple use case diagram. . . . 6
Figure 2.3 Architecture of the Transformer model [42]. . . . 9
Figure 2.4 Graphical illustration of Low-Rank Adaptation (LoRA) [21]. . . . 12
Figure 3.1 General pipeline overview. . . . 17
Figure 3.2 Dataset preparation overview. . . . 19
Figure 4.1 Distribution of per-instance values of all metrics for FTM_{base}. . . . 40
Figure 4.2 Distribution of per-instance values of Completion Rate for each augmented model configuration. . . . 41
Figure 4.3 Distribution of per-instance values of Recovery Rate for each augmented model configuration. . . . 42
Figure 4.4 Distribution of per-instance values of Model Completion Accuracy (MCA) for each augmented model configuration. . . . 43
Figure 4.5 Distribution of per-instance values of BLEU and Cosine Similarity scores for each augmented model configuration. . . . 44
Figure 4.6 Distribution of per-instance values of all metrics for FTM_{NL}. . . . 48
Figure 6.1 Timeline of key works related to LLMs for structured code understanding and generation. . . . 69
Figure 6.2 Timeline of key works related to augmenting LLMs. . . . 70
Figure 6.3 Timeline of key works related to utilizing LLMs in the domain of software modeling. . . . 72

List of Tables

Table 3.1 Dataset splits and number of samples used in the experiments. Validation and test sets remain fixed across all configurations, while the training set varies depending on the augmentation strategy. . . . 20
Table 3.2 Notation for fine-tuned model configurations. . . . 22
Table 3.3 Summary of used variables. . . . 23
Table 3.4 The controlled variables that kept fixed across all experiments. . . . 25

Table 4.1 Mean evaluation metrics for all model configurations. 37
Table 4.2 Relative performance change over the baseline model (FTM_{base}). . . 39
Table 4.3 Average values of alternatively computed Cosine Similarity and BLEU for all model configurations. 45
Table 4.4 Number of malformed (failed-to-be-parsed) predictions per model configuration. 46
Table 4.5 Mean evaluation metrics for FTM_{base} and FTM_{NL} configurations. . . 47
Table 4.6 Relative improvement of FTM_{NL} over the baseline model (FTM_{base}). 47
Table 4.7 Average values of alternatively computed Cosine Similarity and BLEU for all model configurations. 49
Table 4.8 Number of malformed (failed-to-be-parsed) predictions per model configuration. 50
Table A.1 Types of UML diagrams with the corresponding group and description. 77

Listings

3.1 JSON Sample entry. 20
4.1 Example external knowledge entry from the Software Engineering dataset. . 32
4.2 Example external knowledge entry from the Python dataset. 32
4.3 Example prompt used to obtain natural-language comments for a UML class diagram. 35
4.4 Example natural-language comment generated for a UML class diagram. . . 35

Acronyms

BPE Byte Pair Encoding
CPU Central Processing Unit
FFN Feed-Forward Network
GPT Generative Pretrained Transformer
GPU Graphics Processing Unit

HPC High-Performance Computing
ICL In-Context Learning
LLM Large Language Model
LoRA Low-Rank Adaptation
MCA Model Completion Accuracy
MDSE Model-Driven Software Engineering
ML Machine Learning
MM-LLM Multi-modal Large Language Model
NF_4 4-bit NormalFloat
NL Natural Language
NN Neural Network
NLP Natural Language Processing
OO Object-Oriented
PEFT Parameter-Efficient Fine-Tuning
PL Programming Language
QLoRA Quantized Low-Rank Adaptation
UML Unified Modeling Language
RAMC Retrieval-Augmented Model Completion
XMI XML Metadata Interchange

1 Introduction

As software systems continue to grow in size and complexity, the automation of software engineering tasks has become a promising area to improve productivity, reduce human error, and improve the quality of an end product [28]. Tools such as GitHub Copilot[1], powered by LLMs, have demonstrated the significant potential in assisting developers through context-based code suggestions and intelligent code completions, improving their productivity [39]. Following this trend, there are some studies that explore how LLMs and additional Machine Learning (ML) techniques can support and automate complex engineering tasks, involving model-driven software design and, in particular UML model completion.

Existing work in this domain ranges from utilizing ML to extract and generate diagrams from specifications written in natural language [15, 58], to recommending modeling completions based on partial user input and user stories [3, 28]. More recent research has investigated the use few-shot prompt learning to complete domain models without requiring extensive task-specific fine-tuning [6]. Together, these approaches demonstrate that LLMs exhibit great potential for supporting automated software modeling.

A remarkable study done by Chaaben et al. [6] explores a different way of utilizing LLMs. They use the few-shot prompt learning to perform UML model completion without relying on domain-specific datasets and resource-demanding fine-tuning. This approach highlights that LLMs, even in their general form, can provide valuable support for software modeling when carefully prompted, especially in settings where structured modeling data is scarce or fine-tuning is impractical.

The Retrieval-Augmented Model Completion (RAMC) approach introduced by Tinnes et al. [48] is another significant advancement in the domain of software modeling. Their approach demonstrates how historical sequences of software model changes extracted from version-controlled repositories can be used to improve semantic correctness of UML model completions. RAMC utilizes retrieval-augmented generation and fine-tuning, achieving 62% of semantic correctness on industrial datasets, showing that LLMs can learn meaningful modeling patterns when supplied with relevant historical context.

Despite the notable advancements, the existing approaches exhibit significant limitations that constrain their general applicability in practical software engineering environments. Few-shot prompting methods rely heavily on accurate prompts and contextual examples to guide the model's reasoning [6]. Thus, their performance is highly sensitive to prompt structure and the retrieval of relevant contextual information, which limits robustness and reproducibility when applied to complex UML models [6]. Retrieval-augmented approaches such as RAMC depend on the availability of sequential model histories that capture the evolution of software systems [48]. While this assumption holds in version-controlled

1 https://github.com/features/copilot

environments, it excludes many real-life scenarios where models exist as independent, static artifacts rather than as parts of an evolving sequence. As a result, such approaches cannot be directly applied to scenarios involving isolated, fully or partially complete UML models.

Together, these limitations suggest that current LLM-based approaches rely on prompt- or retrieval-based mechanisms which limits their transferability to scenarios involving independent, static UML models. To address this gap, this work investigates whether fine-tuning LLMs on independent, fully complete UML models can improve their structural understanding, completion ability and quality without relying on historical sequence or tailored prompts. Furthermore, we explore whether augmenting LLMs with some NL data such as WIKIPEDIA[2] entries or integrating NL descriptions of the general structure of UML class diagrams can enhance the model's ability to infer and predict the missing UML model fragments.

We propose an approach which involves fine-tuning an LLM on a diverse dataset of standalone UML class diagrams and evaluating its ability to reconstruct missing model elements. In contrast to RAMC approach, which relies on accessing sequential model histories at inference time to exploit software evolution information [48], our method does not assume the availability of historical model versions or external retrieval mechanisms. Instead, we fine-tunes an LLM directly on *independent* UML class diagrams.

Additionally, we explore whether augmenting the dataset with supplementary information such as texts or comments in NL and some code snippets can further strengthen inference. This combination of domain-specific fine-tuning and dataset augmentation aims to provide the model with both the structural regularities of UML models and the conceptual basics needed to interpret modeling semantics.

The key advantage of this approach is that it operates entirely on independent UML artifacts, allowing for applicability in settings where no repository histories or curated prompt libraries exist. Additionally, it offers greater consistency, reproducibility, and stability in generated completions by relying on fine-tuning rather than prompt engineering. The integration of NL and code snippets also has the potential to improve the model's interpretability and reasoning capabilities.

The results indicate that augmenting training data with non-modeling information, such as natural language text or source code, has a highly limited context-dependent impact on the LLM's performance tasked with UML class diagram completion. While moderate amounts of domain-adjacent software engineering data can result in small improvements, general-purpose text, source code and higher augmentation levels often increase failure rates, and degrade the performance of the model. Furthermore, the inclusion of high-level NL comments describing the general intent of a given class diagram does not result in improved completion performance under the investigated conditions and is associated with reduced structural accuracy, likely due to increased ambiguity and insufficient alignment with concrete model elements.

Overall, this thesis contributes to the domain by demonstrating how domain-adapted fine-tuning on independent UML models, combined with dataset augmentation, can serve as a viable alternative to prompt- or history-dependent modeling approaches. It provides insights for designing robust, adaptable, and context-independent tools for automated software modeling using capabilities of modern LLMs.

2 https://www.wikipedia.org

1.1 Goal of this Thesis

This thesis investigates how fine-tuning pretrained LLMs on independent, domain-specific UML data influences their ability to complete software models. Building on this objective, it examines how domain-specific modeling knowledge, external knowledge, and NL comments affect the accuracy and overall quality of model predictions. By exploring how these different forms of knowledge guide the completion process, the thesis aims to propose an approach that may contribute toward the future use of fine-tuned LLMs as practical tools for supporting automated software modeling.

1.2 Overview

This thesis consists of the following four chapters. Chapter 2 presents a comprehensive overview of theoretical principles crucial for a thorough comprehension and a clear understanding of this work. Chapter 3 defines the research questions and introduces the applied metrics and methodologies. Subsequently, Chapter 4 presents the experiments, obtained results and respective discussion. Furthermore, Chapter 5 considers possible limitations and proposes areas for future research. Chapter 6 analyses the relevant related work. Finally, Chapter 7 provides the comprehensive summary of this thesis.

2

Background

In this chapter, we present the theoretical foundations necessary to understand the methodology and contributions of this work. We begin by introducing the relevant types of UML diagrams and outlining their role in software modeling. Next, we provide an overview of LLMs, including their main concepts and the underlying transformer architecture. Finally, we discuss the concept of fine-tuning and outline alternative adaptation techniques relevant to our study.

2.1 Unified Modeling Language (UML) and Software Modeling

Unified Modeling Language (UML) [36] provides a standard way of visualizing the design of a software system or a process, for example, business processes. Providing such visual abstractions, UML models make it possible to conceptualize the architecture of a software system before its implementation, and support further stages of software development, such as testing, debugging, code refactoring, and refinement [13, 26, 44]. Additionally, it addresses the limitations of traditional programming languages that mostly operate at the low level of abstraction and can be insufficient to capture well the high-level structure and behavior of software systems [17]. By providing a various types of diagram notations, UML enables software engineers to create models that describe both the static structure—such as classes, objects, and their relationships—and the dynamic behavior, including interactions, state changes, and workflows within and between system components.

2.1.1 Types of UML diagrams

UML defines 14 different diagram types, which in turn are categorized into 2 groups: structure diagrams and behavior diagrams [36]. Structure diagrams are used to outline a system's static elements, their relationship, and provide the fundamentals for behavior diagrams. One of the prominent and widely used diagrams in Object-Oriented (OO) design and development is a class diagram [4, 17, 26]. It describes different types of objects in the system as classes, their names, structural features as class fields, functions, and relationships that exist between classes [4, 17]. Figure 2.1 represents an example of a simple class diagram. Other examples of structure diagrams include an object diagram, which

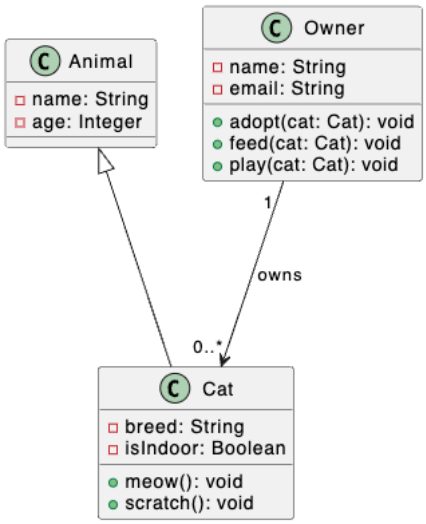


Figure 2.1: An example of a simple class diagram.

represents a snapshot of an object in a system, and a component diagram, which shows the system's components and their interactions [17].

On the other hand, behavior diagrams represent changes in the system state or in other words, the dynamic state of a system, and provide a framework for the execution of behaviors [36]. One of the commonly used in practice behavior diagrams is sequence and use case diagrams [26]. The first one is usually used to model the interaction between the objects in a system and describe the behavior of a system in a concrete scenario, whereas the latter one describes the interaction between the system user also known as an actor and the system itself [17]. Figure 2.2 is an example of a simple use case diagram, where the actor is `Owner` and the diagram itself illustrates the functionality available to the user.

As previously stated, the UML offers numerous diagram types, each with a distinct function in visualizing parts of a software system. The Table A.1 outlines all 14 diagram types, as well as their descriptions and key roles.

Although different UML diagram types serve specific purposes, class diagrams are the most common target for automation and completion tasks, due to their central role in describing the system's structural part [4]. In the context of this thesis, class diagrams are the primary focus.

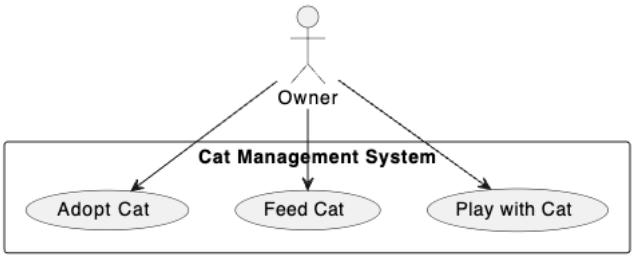


Figure 2.2: An example of a simple use case diagram.

2.1.2 UML in software engineering

UML models serve multiple purposes throughout the software development life cycle, enabling system understanding and design process before implementation, supporting subsequent development stages as testing, debugging, maintenance and documentation Koc et al. [26] and Selic [44]. They also facilitate communication between stakeholders that have different technical knowledge, making it easier to translate natural language requirements into implementable designs [17, 44]. By abstracting low-level details, UML provides a common language that bridges the gap between domain experts and developers, thus enhancing collaboration and reducing misunderstandings.

The widespread adoption of UML is further supported by its compatibility with modern software engineering and development tools. Standardized serialization formats such as the XML Metadata Interchange (XMI)[1] allow UML to be translated into a format that is compatible with various programming languages, enabling their seamless integration with other system components.

2.1.3 Model-Driven Software Engineering

Brambilla et al. [4] defines Model-Driven Software Engineering (MDSE) as "approaches based on modeling for the development of software artifacts" that also increase efficiency and effectiveness in software development. Nowadays, these methods are widely used in fields as software engineering, software design and testing for problem understanding and documentation, and generally, as one of the essential steps in software development [23].

Models are the primary artifacts in MDSE, driving code generation, testing, validation, and transformation. MDSE's primary principle is that raising the level of abstraction through models makes it possible for software engineers to focus more on the software quality, namely system design and problem-solving, rather than low-level implementation details [4, 23].

2.2 The Large Language Models (LLMs)

LLMs are a category of machine learning models with a large number of parameters that are trained on vast corpora of text data. For example, GPT-3 is trained with 175B of parameters and the newest creation of OpenAI[2], GPT-4 claimed to be even larger and to have over a trillion parameters [37]. These types of models are designed to understand, process, and generate NL with high fluency and contextual awareness [34, 42, 45]. The Transformer, proposed by Vaswani et al. [50], serves as the foundational architecture for the most state-of-the-art LLMs, enabling models to handle long-range relationships as well as parallelize training effectively. Utilizing this framework, models like Generative Pretrained Transformer (GPT) [40] and CodeLLaMA [43] have demonstrated outstanding performance across a wide range of tasks involving Natural Language Processing (NLP), from code

1 https://www.omg.org/spec/XMI
2 https://openai.com

generation to question answering in fields like medical and health science [25], multilingual machine translation [59], finance, education, etc [34, 42].

As a result of their versatility and scalability, LLMs have become essential tools for tasks involving natural language understanding, processing, and generation. However, applying these models to specific domains often requires adaptation approaches to enhance task-specific performance while avoiding the computational cost of retraining. This has sparked considerable interest in fine-tuning methods, which adapt pre-trained models to specialized tasks with greater efficiency and lower computational cost than training from scratch [34]. We discuss some of these methods in Section 2.3.

2.2.1 Main concepts in LLMs

This section outlines the important concepts that form a baseline of LLMs. These concepts play a critical role in enabling LLMs to process, learn from, and generate NL effectively.

- **Data Pre-processing**
  Data pre-processing is one of the first and crucial steps in LLM training. It ensures the relevance, quality, and safety of the data. This process focuses on filtering, deduplication, and privacy preservation of the data used. Filtering aims to remove low-quality and irrelevant data, improving performance and model generalization. Deduplication reduces the data redundancy by removing repeating text, ensuring that the model is not overfitting to frequently occurring patterns. Finally, privacy reservation is applied to minimize the risk of exposing personal data or sensitive information like phone numbers, addresses, etc.

- **Tokenization**
  Tokenization is a process of decomposing textual input into units called tokens, which serve as a model input. These tokens can represent words, word-pieces, sub-words, symbols, or characters, depending on the specific tokenization algorithm used. The early formal approaches to tokenization were introduced by Webster and Kit [55], but modern LLMs typically rely on more modern and advanced methods such as WordPiece, UnigramLM, and Byte Pair Encoding (BPE) [27, 42].

- **Attention Mechanisms**
  Attention mechanisms are one of the core building blocks of the modern LLMs, particularly Transformers. First, attention was introduced by Bahdanau et al. [1] to improve the performance of the basic encoder-decoder architecture and allow a model to automatically search for parts of an input sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly. After that, it was refined in the transformer architecture proposed by Vaswani et al. [50].

  Attention allows the model to weight the input tokens based on their importance and use these weights when generating or interpreting text. There exist several attention strategies, but here we discuss the most relevant ones. In Self-Attention, the representation of each token is computed using all other tokens' values in that sequence, allowing the model to capture contextual and long-range dependencies. Multi-head self-attention mechanism is used in the Transformer architecture and is

built based on multiple self-attention heads running in parallel, allowing the model to simultaneously capture contextual relationships of words in the entire input sequence and produce the attention score for it [35].

- **Activation function**
  Activation function introduces non-linearity into the Neural Network (NN). It plays an important role in the curve-fitting abilities of LLMs architectures [20] and empowers them with the ability to learn complex patterns. Activation functions are applied in the feed-forward sub-layers [50] of Transformers, contributing to the model's depth and expressiveness. There are such activation functions as ReLU, GELU and further GELU [19] variations.

2.2.2 Transformers and their architecture

A variety of modern LLMs like LLaMA [49], BERT [11], and T5 [41] use the transformer architecture as their foundational model. Unlike recurrent and convolutional NN, transformers solely rely on the attention mechanism to determine the dependencies between input and output [50]. This allows the model to process an input sequence of a varying length in parallel and change its focus depending on the length of the input sequence, improving efficiency and scalability.

Figure 2.3 illustrates the architecture of the transformer model, which consists of seven main components that are described further.

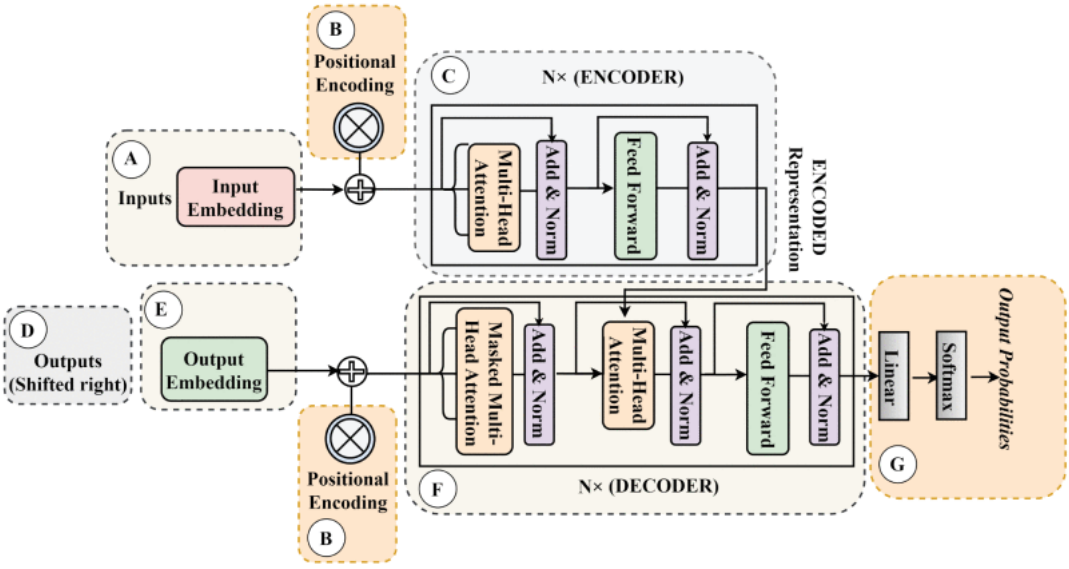


Figure 2.3: Architecture of the Transformer model [42].

A. **Inputs and Input Embeddings**
The input embeddings are the numerical representation of the input. First, the input is tokenized. After the tokenization is done, each token gets assigned a unique number, which connects the linguistic information to the numerical vector - the input

embedding [42]. As a result, the input text has its input embedding, which can be further utilized by the machine learning model. These embeddings help the model understand the meaning of words by arranging them in a mathematical space as vectors [42]. Thus, words or phrases that have a similar meaning are embedded as vectors of equal size and located close to each other.

B. **Positional Encoding**
The positional encoding is used to encode the position of each word in the input sequence as a collection of integers [42, 50]. This is then added to the input embedding to assist the model in understanding the order of words in the input sequence and construct an input that is both grammatically and semantically correct.

C. **Encoder**
The encoder is one of the significant parts of the transformers, which processes the input embeddings. It consists of a stack of $N = 6$ identical layers [50] and each layer consists of the further two sub-layers:

- Multi-Head Self-Attention that was described earlier.
- Feed-Forward Network (FFN): processes the attention score and further transforms it using a fully connected feed-forward network.

D. **Outputs (Shifted right)**
In the training process, the ground truth output is provided to the decoder and it is shifted right. This mechanism ensures that the decoder's predictions for each token are made based on the preceding tokens, and is known as autoregressive training [42].

E. **Output Embedding**
The decoder receives the shifted output sequence as input and embeds it similarly to the input embeddings. Positional encodings are added once again to preserve information about the order of tokens.

F. **Decoder**
The decoder is responsible for generating the output sequence based on the encoded input sequences. Same as the encoder, it consists of N identical layers, with each layer including the following main components:

- Masked Multi-Head Self-Attention: functions similarly to the encoder's attention mechanism. The main difference is that here masking is applied to prevent the decoder from attending to future tokens during the training process [35].
- Multi-Head Encoder-Decoder Attention: the decoder attends to the encoder's output sequence, which allows it to focus on the relevant parts of this sequence when generating each token.
- FFN: works similarly to the encoder [42].

G. **Output Probabilities**
The final part of the Transformer architecture is comparing the predicted probability of each token against the ground truth, which is done using cross-entropy loss [50]. The output sequence produced by the encoder is passed through a linear projection

layer, which maps each token representation to the vocabulary size. After, a softmax function is applied to convert the logits into a probability distribution over all possible output tokens [42]. Finally, at each step of inference, the model selects the token that has the highest probability.

2.3 Fine-Tuning of LLMs

The initial purpose of LLMs is to process text input and generate responses in natural language. To improve the performance of LLMs, add desirable or remove unwanted behavior, one can fine-tune them. Fine-tuning with task-instruction data for downstream tasks improves their generalization ability to unseen tasks, allows domain adaptation, and reduces misaligned behavior [34]. In this section, we talk about different fine-tuning approaches and compare fine-tuning to some of the alternative strategies.

2.3.1 Full Fine-Tuning

LLMs consist of a fixed number of parameters that they have been pretrained on, for example, GPT-3 has approximately 175B parameters [5]. Full fine-tuning involves updating all these parameters to adapt the pretrained model to downstream tasks. While this approach is straightforward, simple, and effective, it demands a lot of computational resources and time for each task due to the large number of parameters involved [7, 34].

2.3.2 Parameter-Efficient Fine-Tuning

Given the resource demands of full fine-tuning, Parameter-Efficient Fine-Tuning (PEFT) methods have been developed to address this problem and reduce computational overhead by adjusting a fraction of trainable parameters while keeping most of the pretrained ones unchanged Chen et al. [7]. These methods minimize additional latency to the model during inference and have demonstrated effectiveness in natural language tasks [33].

Chen et al. [7] has conducted a comparative study between PEFT and full fine-tuning, concluding that PEFT methods tend to be more suitable for low-resource scenarios, perform roughly the same in medium-resource settings, but may underperform when it comes to high-resource tasks.

Common PEFT techniques include Adapter modules, LoRA, Quantized Low-Rank Adaptation (QLoRA), Prompt Tuning, Prefix Tuning, and BitFit. Among these, LoRA and QLoRA are the primary methods utilized and evaluated within the scope of this thesis.

2.3.3 Low-Rank Adaptation

LoRA proposed by Hu et al. [21], is a PEFT method designed to reduce the computational and storage costs of adapting pretrained LLM to downstream tasks. Instead of updating all

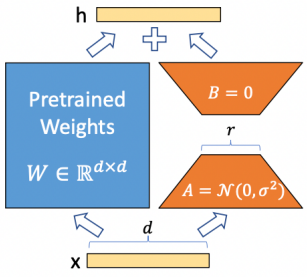


Figure 2.4: Graphical illustration of LoRA [21].

parameters of the pretrained model, LoRA freezes the original weight matrices and learns a low-rank update, expressed as the product of two smaller matrices A and B, which are injected into each Transformer layer's weight matrix. Figure 2.4 illustrates the described adaptation graphically and shows the frozen weight matrix W of the pretrained model which is additively combined with the trainable low-rank update A and B on the input X. This yields the adapted output $h = Wx + BAx$ [21]. This method effectively enables the sharing of the frozen pre-trained weights across multiple tasks while only swapping small task-specific low-rank modules, reduces Graphics Processing Unit (GPU) memory usage, and introduces no additional inference latency to the model [21].

In the context of this work, LoRA's parameter-efficient adaptation is particularly helpful for the model's fine-tuning on software engineering tasks due to limited computational resources and the availability of domain-specific labeled data. Therefore, we aim to utilize the advantages provided by LoRA.

2.3.4 Quantized Low-Rank Adaptation

QLoRA introduced byDettmers et al. [10], extends LoRA by combining low-rank parameter updates with 4-bit quantization of the pretrained model weights. This approach allows fine-tuning of LLMs on a single GPU without performance degradation, while also reducing memory requirements [10].

QLoRA method achieves high-fidelity 4-bit fine-tuning through the integration of the following techniques:

- 4-bit NormalFloat (NF_4): an optimal quantization data type tailored for normally distributed data, which preserves model accuracy better than conventional 4-bit integer or float quantization schemes, allowing high-fidelity fine-tuning even at low precision [10].
- Double Quantization: compresses the quantized weights by applying secondary quantization on them, reducing memory usage and allowing large models to fit into limited GPU memory without degrading their accuracy [10].
- Paged Optimizers use the Nvidia unified memory feature, which automatically manages paging between the Central Processing Unit (CPU) and GPU. This mechanism temporarily offloads parts of the model state and gradients to CPU memory and loads

them back on demand, effectively extending GPU memory capacity and enabling error-free training without manually transferring the data [10].

Within this thesis, QLoRA is a practical alternative to LoRA that allows the training of the models with reduced memory usage and noticeable performance given limited computational resources.

2.3.5 Prompt Engineering

Prompt engineering is a variety of prompt-based methods, such as zero-shot, N-shot, and chain-of-thought techniques, designed to improve the performance of LLMs by guiding their reasoning process through transformed natural language queries [8, 34]. Usually, prompt engineering involves the following steps: firstly, task-specific inputs and examples are transformed using templates that contain placeholders; secondly, during inference, the LLM generates outputs by filling these placeholders; and lastly, the desired outputs are extracted from the generated text using post-processor. Importantly, prompt engineering does not involve any parameter change of the model, they remain fixed [8].

2.3.6 In-Context Learning

In-Context Learning (ICL) also known as few-shot learning is an approach that allows LLMs to learn tasks using a few examples that are provided in the form of demonstration [5, 12, 34]. This approach does not involve any change in models' weights and requires a small amount of task-specific data but rather allows the model to learn from the provided analogies which makes it computationally cheap [5, 12]. ICL is broadly applied in NLP tasks, data engineering, model augmentation, and knowledge updating [12].

2.3.7 Fine-tuning vs. In-Context Learning, Promt Engineering

Together, fine-tuning, prompt engineering, and ICL aim to enhance the abilities of LLMs and tailor them to specific tasks. However, these approaches are fundamentally different. Fine-tuning involves adjusting the model's internal parameters/weights using task-specific datasets, allowing the model to adapt more explicitly and deeply to downstream tasks or specific domains [5, 34]. In contrast, ICL and prompt engineering methods rely on cleverly structured inputs and contextual examples to guide the model's reasoning without altering model parameters [5, 12]. The choice between these approaches depends on factors such as the availability of computational resources, the amount of training data, and the specificity or generality required by the target task.

2.4 Definition of Structural and Semantic Correctness

In the context of automated UML class diagram completion, we distinguish between *structural correctness* and *semantic correctness*.

We define *structural correctness* as the accurate reconstruction of the removed UML elements, such as classes, attributes, relationships, and their properties, without introducing faulty or hallucinated elements. This notion is operationalized through structural metrics, namely MCA and Recovery Rate, which are introduced in Section 3.4. They capture whether the class diagram generated by a fine-tuned model variant is structurally equivalent to the ground truth.

In contrast, *semantical correctness* describes the degree to which the generated output is semantically related to the intended meaning of the removed element. It reflects conceptual or topical similarity between the missing and the predicted elements, such as the use of similar or relevant terminology, even if the exact structural configuration is not correctly reconstructed. Semantic correctness is therefore less strict than structural correctness and does not necessarily imply structural validity.

2.5 Definition of Model Performance

Throughout this thesis, we use the term *model performance* to refer specifically to the quality of UML class diagram completions produced by the fine-tuned LLM. Here, performance does not denote generic text-generation ability, but rather corresponds to how accurately the model reconstructs missing UML elements in XMI format.

To operationalize the concept of *model performance*, we evaluate outputs produced my fine-tuned model using a combination of structural, semantic metrics and graph-based similarity measures that are described further in this section.

Thus, in the context of this work, higher performance means that the model predictions closely align with the ground-truth across the evaluation dimensions.

2.6 Definition of Base Model

The *base model* term that we use in this thesis refers to the fine-tuned LLM trained exclusively on the UML dataset without any form of data augmentation and is denoted as FTM_{base}. This dataset contains only partial XMI-encoded class diagrams generated through the data preparation pipeline described in Section 3.2.3.

The base model serves as the reference point for all subsequent experiments. Its performance establishes the baseline against which all augmented configurations are compared. Any observed performance changes in later experiments are therefore interpreted relative to this model, ensuring that improvements can be attributed specifically to the applied augmentation strategies rather than to differences in training procedure or data preprocessing.

3

Methodology

In this chapter, we present the methodology to evaluate the performance of fined-tuned LLMs. First, we start by introducing the research questions. Then, we move to the details of the procedure, including the setup of the experiments. Finally, we discuss the evaluation metrics that are going to be used to evaluate the performance and address the research questions.

3.1 Research Questions

To address the limitations of existing approaches discussed in Chapter 6 and to evaluate the effectiveness of our method in a systematic and measurable way, we formulate the following research questions.

> **RQ1** What is the impact of integrating non-modeling data on the performance of fine-tuned large language models in UML class diagram completion?

While fine-tuning language models on task-specific data is known to improve performance, and augmenting them with external knowledge has demonstrated benefits for such tasks as code-completion [14, 15, 30, 57], the role of general-purpose external knowledge in the context of UML model completion remains insufficiently explored. In particular, as discussed in Chapter 6, existing approaches predominantly rely on domain-specific or structurally similar datasets.

Therefore, our first research question explores whether augmenting UML training data provided in XMI format with external textual and code sources, such as Wikipedia[1] articles, code repositories, and general software engineering-related documentation, can support model's ability to complete partial UML diagrams. In contrast to existing fine-tuning strategies that train exclusively on structured domain-specific UML representations, we examine whether injecting auxiliary, structurally different knowledge helps the model generate more accurate predictions and infer the missing classes, attributes, relationships and further class diagram elements.

Our experimental setup is inspired by Rozière et al. [43], who showed that sourcing $5-8\%$ of sample data from NL datasets related to code or NL datasets improved Code Llama's [43] overall performance and allowed the model to train its NL understanding skills. We adopt a similar experimental approach. More specifically, we aim to vary the proportion of external knowledge included during model training relative to our main dataset and

1 https://www.wikipedia.org

quantify the effect of adding the auxiliary data on the LLM's completion abilities of class diagrams.

Generally, in this research question, we seek to quantify performance differences between models trained exclusively on UML data and those exposed to different proportions of auxiliary knowledge during fine-tuning.

We hypothesize that adding external knowledge to the training data will improve the model's ability to infer missing UML class diagram elements. External texts provide semantic and contextual information that is typically not explicitly encoded in UML diagrams themselves. Prior studies in code generation and structured prediction suggest that LLM benefit from diverse background knowledge [31, 33]. Therefore, we expect that the presence of external textual knowledge may positively contribute to UML model completion accuracy and overall model's performance.

> **RQ2** What is the impact of including natural language comments on the performance of fine-tuned large language models in UML class diagram completion?

LLMs are initially pretrained on large-scale NL corpora to interpret and generate NL, and prior work in code generation shows that NL inputs can improve model performance, especially in such scenarios when tasks are ambiguous or underspecified [31, 33]. Liu et al. [31] further demonstrated that longer natural language prompts, providing richer contextual or domain background, can enhance accuracy of model's predictions by reducing uncertainty in the its internal reasoning.

Building on these insights, our second research question explores whether integrating natural language comments into the UML fine-tuning dataset can improve an LLM's ability to infer missing class diagram elements. In our work, natural language comments describing the general idea and the domain of the partial UML diagram and providing its high-level overview are first generated my a supplementary LLM and then injected into the partial UML class diagram in a from of XMI comments. By combining these NL descriptions with the corresponding UML representations during fine-tuning, we aim to determine whether such additional context helps the model to infer missing class diagram elements and generate accurate UML diagram completions.

In summary, addressing this research question, we seek to evaluate whether augmenting UML training data with structured natural language comments enhances the model's structural reasoning and improves its performance on UML class diagram completion relative to training on diagram data alone.

We hypothesize that augmenting UML diagrams with natural-language comments provides additional background that helps the LLM to interpret the diagram's structure and improve the ability to infer missing UML class diagram elements. Integrated comments that describe class responsibilities, relationships, and general purpose of the whole diagram can reduce ambiguity in partial diagrams and enable more accurate reconstruction of missing elements. Prior work has shown that LLMs leverage NL context effectively in code-related tasks and diagram reasoning [31]. Therefore, we expect that models trained on comment-integrated diagrams will outperform models trained on UML-only data.

3.2 Research Design

This work is an experimental study that aims to evaluate the performance and generalization capabilities of fine-tuned LLMs on automating software modeling tasks. The main focus of our study is completion of UML class diagrams using XMI-serialized models under varying data-augmentation strategies.

To systematically address the research questions, we define a set of controlled experiments in which we vary selected independent variables (e. g., proportion of external knowledge integrated to the main dataset) while holding the other variables constant. Table 3.3 lists the independent and dependent variables associated with each research question.

To ensure comparability across all experimental settings, we employ a fixed test set of partial UML class diagrams in XMI format, which is derived from the datasets described in Section 3.2.2.This test set is used consistently across all experiments. Additional controlled variables, such as model architecture, fine-tuning hyperparameters, and evaluation metrics, are summarized in Table 3.4 and described in more detail in the subsequent sections.

In line with the research questions, the study is organized into two main groups of experiments. The first group corresponding to RQ1, examines the role of external domain knowledge added to the primary UML dataset. The second group corresponding to RQ2, investigates the effect of embedding natural-language comments within the XMI diagrams. Each experiment follows the same pipeline shown in Figure 3.1, consisting of data preparation, fine-tuning, inference, and evaluation.

3.2.1 Overall Experimental Pipeline

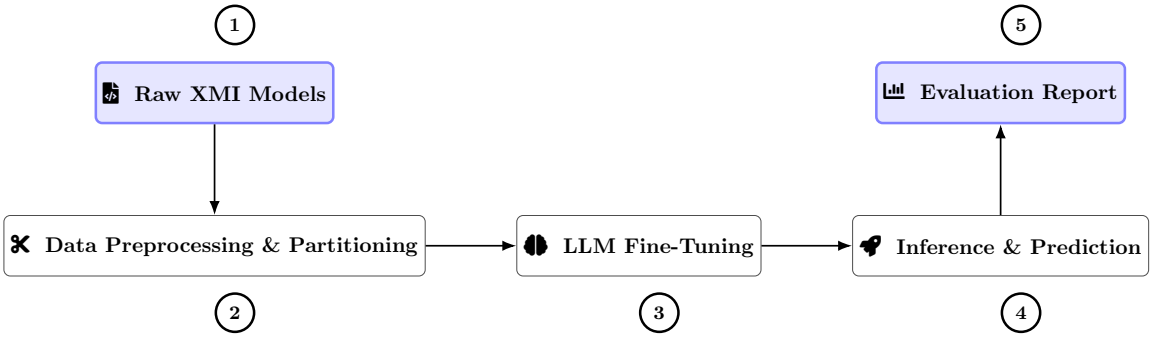


Figure 3.1: General pipeline overview.

Figure 3.1 provides a high-lever overview of workflow employed in this study. The pipeline consists of the five main stages:

1. **Raw XMI Models**: extraction of UML class diagrams from the ModelSet [32] dataset (described in more detail in Section 3.2.2).

2. **Data Preprocessing & Partitioning**: partition of each of the dataset entry into smaller chunks, parsing the partitioned XMI files and randomly removing class diagram

elements, generating partial diagrams, and splitting them into training, validation, and test sets. Here, we also prepare the augmentation datasets in the same manner.

3. LLM **fine-tuning**: training a LLaMA-based model using different configurations of augmented data.
4. **Inference & Prediction**: applying the fine-tuned models to infer the missing class diagram elements and reconstruct the partial UML diagram.
5. **Evaluation report**: assessing the produced outputs using structural and semantic evaluation metrics and summarizing the results.

The pipeline described above defines the common experimental workflow applied throughout this study. While all stages are shared across experiments, the concrete instantiation of the data preparation and fine-tuning procedures differs depending on the respective research question. We describe these research question-specific data preparation and experimental procedures in more detail in Section 4.1.

3.2.2 Datasets

Our work involves 4 different datasets: ModelSet [32], Wikipedia [16], Python [51], and Software Engineering [24]. We describe each one of them in more detail:

- **ModelSet** [32]: contains 10580 models, containing 5460 Ecore and 5120 UML labeled models. The dataset provides model in XMI and JSON formats. We exclusively utilize the UML class diagrams serialized in XMI format, which are 3109 models in total. We refer to this dataset as the *main dataset*. During the fine-tuning process, it is split into a train (80%), test (10%), and validation sets (10%).
- **Wikipedia** [16]: contains Wikipedia articles of all languages and from different time periods. We use the entries written in the English language, from 01.03.2022, as a part of the secondary dataset for RQ1.
- **Python** [51]: is a Python code dataset, consisting of 143327 examples of code, which are verified as working code. Each entry is an instruction-optional input-output triple, where the instruction is written in natural language and some code fragments, and the output is of the form tuples of NL description of the task and working Python code. We integrate the entries from this dataset into our secondary dataset used for augmentation.
- **Software Engineering** [24]: consists of various questions and respective answers in English language from the software engineering domain. This dataset is a part of the secondary dataset that is used for the augmentation.

3.2.3 Data Preparation Pipeline

Since data plays major role during fine-tuning, we describe the whole data preparation process in more detail. The data-processing workflow underlying Steps (1)–(2) of the overall

experimental pipeline (Figure 3.1) is illustrated in more detail in Figure 3.2. It consists of the three main phases designed to transform raw software models into inputs suitable for the fine-tuning process. The main goal of this pipeline is to ensure consistency, experimental reproducibility, and compatibility with token-based training constraints of the LLM.

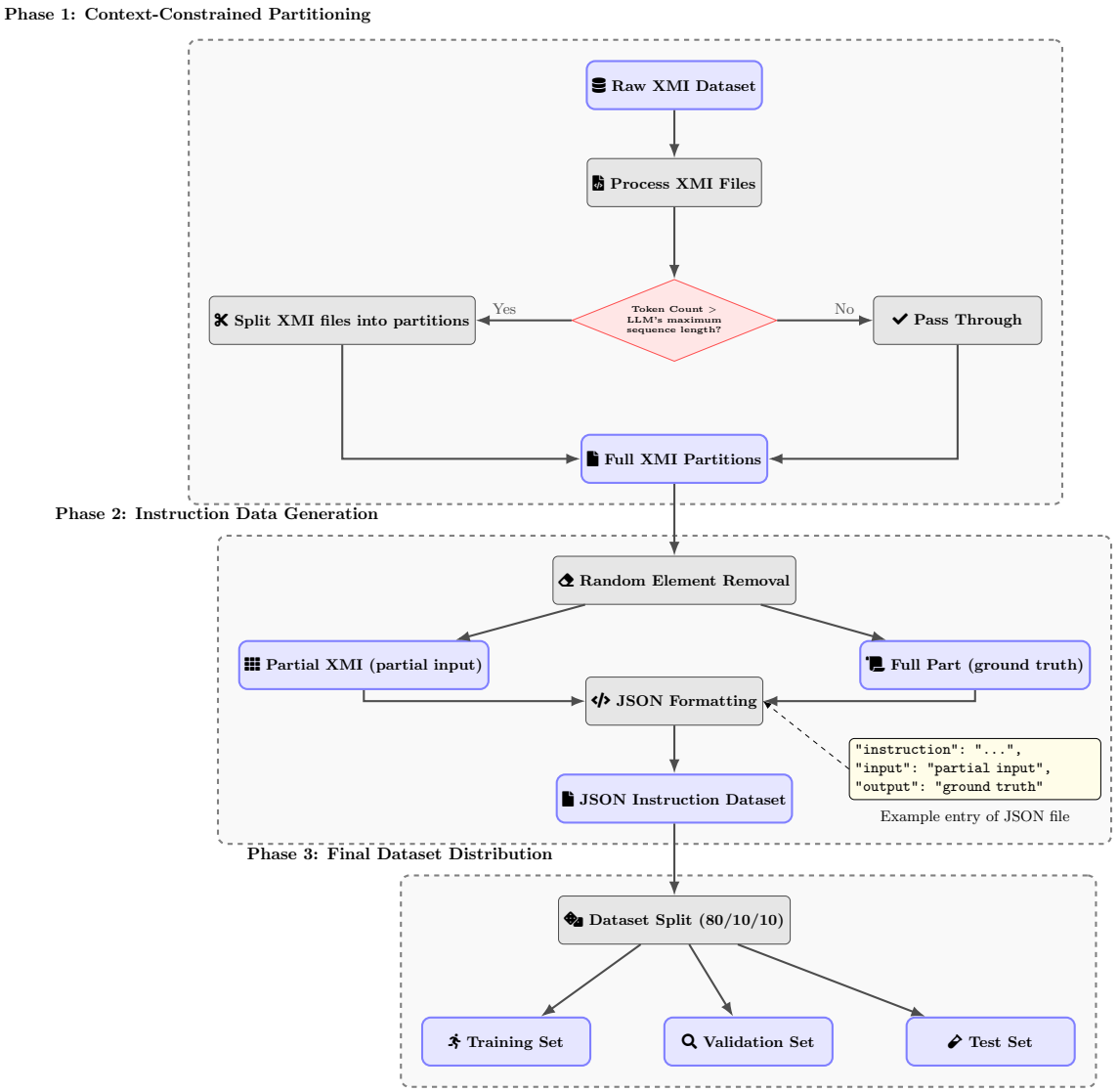


Figure 3.2: Dataset preparation overview.

- **Phase 1: Context-Constrained Partitioning.**
  This phase focuses on segmenting large UML class diagrams in XMI format into smaller, context-preserving partitions when their size exceeds the token constraints enforced by the fine-tuning setup. The partitioning process ensures that each resulting partition remains structurally valid and semantically interpretable, preserving essential relationships between classes. It is important to note that this partitioning step is

applied *exclusively* to the ModelSet dataset [32] described earlier in Section 3.2.2, as it contains complex and highly variable diagrams in terms of size and structural density.

- **Phase 2: Instruction Data Generation.**
  In this phase, the complete UML class diagrams processed in Phase 1 are transformed into training-ready instruction pairs. To simulate the diagram-completion task, we systematically remove model elements, such such as classes, attributes, operations, and associations, from the original diagrams. Each removal generates a *partial* diagram, which serves as the model's input, while the corresponding *full diagram* constitutes the target output. Both, *partial* and *full diagram* are stored in a structured JSON format, together with an instruction describing the completion task. Listing 3.1 provides an example of a resulting JSON entry. For all external datasets, e. g.,textual corpora used in RQ1, the process begins directly at this phase, ensuring that all data conform to the same instruction–input–output structure.

- **Phase 3: Final Dataset Distribution** In the final phase, all generated JSON triples are organized into the three datasets required for experimentation: the *training*, *validation*, and *test* sets. These sets are split in 80/10/10 proportions. The validation and test sets are kept fixed across all experiments to ensure comparability and reproducibility, while the training set is varied to reflect different augmentation configurations. Specifically, we construct multiple training splits corresponding to (i) no external augmentation, (ii) 5% external knowledge augmentation, and (iii) 8% external knowledge augmentation.

  Table 3.1 summarizes the number of instances contained in each split and configuration.

Table 3.1: Dataset splits and number of samples used in the experiments. Validation and test sets remain fixed across all configurations, while the training set varies depending on the augmentation strategy.

Dataset Split	**(i) No Augmentation**	**(ii)** 5% **Augmentation**	**(iii)** 8% **Augmentation**
Training	13056	13708	14100
Validation	1632	1632	1632
Test	1633	1633	1633

All fine-tuned models are evaluated on the same fixed test set and the same fine-tuning model configuration. This approach allows us to ensure that performance differences arise only from changes in the training data composition rather than differences in evaluation samples or model configuration.

Listing 3.1: JSON Sample entry.

```
{
    "instruction": "Given the following incomplete UML model in XMI format, complete the model
         by adding the missing parts" ,
    "input": <partial class diagram>,
    "output": <full class diagram>
```

```
}
```

3.2.4 LLMs

Here, we elaborate on the LLMs used for both fine-tuning and comment generation as well as the augmentation datasets. Additionally, we define the baseline model configuration and the notation for different fine-tuned model configurations.

Primary Model for Fine-Tuning. For all experiments, we fine-tune a publicly available LLaMA-based model with approximately 8 billion parameters. This model is used as the core system whose performance we evaluate under different data augmentation strategies. We choose this model due to its strong general-purpose reasoning capabilities, sufficiently large context window, and open-weight availability, which makes it suitable for fine-tuning on structured data such as UML class diagrams in XMI format.

All experiments use the same base checkpoint, training strategy, and quantization configuration (QLoRA), ensuring full comparability across experimental conditions.

Auxiliary Model for Comment Generation. As it was mentioned earlier, to address the RQ2 we utilize an auxiliary instruction-tuned LLM Mistral with approximately 7 billion parameters to generate comments describing the general purpose of the class diagrams. These comments provide high-level descriptions of the diagram content and are subsequently injected into the XMI files during the dataset preparation phase.

We do *not* evaluate this model as it is solely used to provide additional linguistic context for augmenting the training data. Using a separate model for this task ensures that the generated comments do not introduce distributional biases from the primary model and maintains a clean separation between data generation and model evaluation.

Both, primary and the auxiliary models are accessed via the HuggingFace model hub[2].

3.2.5 Model Notation

To encourage consistent reference to the different fine-tuned model configurations, we introduce a unified notation for all experimental variants. This notation is used consistently throughout the rest of the thesis to ensure clarity and comparability across experimental results. Table 3.2 summarizes the notation.

FTM_{base} refers to the *base model* defined in Section 2.6.

For the experiments addressing the RQ1, models augmented with external knowledge are denoted as FTM^{p}_{EK}, where p indicates the proportion of external data (in percent) added to the main training set. Accordingly, FTM^{5}_{EK} and FTM^{8}_{EK} refer to models trained with 5%, and 8% external knowledge, respectively. The external knowledge set EK is composed of data elements originating from the datasets introduced in Section 3.2.2, formally defined

2 https://huggingface.co/

Table 3.2: Notation for fine-tuned model configurations.

Notation	Description
FTM_{base}	Baseline model fine-tuned exclusively on partial UML class diagrams in XMI format, without any form of data augmentation.
FTM^{p}_{EK}	Model fine-tuned on UML class diagrams augmented with p% external knowledge sampled from the EK set (wiki, se, code), where $p \in \{5, 8\}$.
FTM_{NL}	Model fine-tuned on UML class diagrams enriched with NL comments embedded within the XMI representation.

as EK $\in \{\text{wiki}, \text{se}, \text{code}\}$, where *wiki* corresponds to the WIKIPEDIA entries, *se* software engineering–related textual data, and *code* programming-related instruction data.

Furthermore, for the RQ2, the model trained on UML class diagrams combined with natural language comments embedded within the XMI representation is denoted as FTM_{NL}.

3.2.6 Experimental Conditions

The design of our study follows a controlled experimental framework, where the we manipulate the independent variables while holding the dependent variables constant and measuring them consistently across conditions.

As demonstrated in Table 3.3, two independent variables are manipulated in this study, corresponding directly to RQ1 and RQ2.

RQ1: External Knowledge. The first independent variable that we manipulate is the proportion of external knowledge added to the primary UML dataset. Here, we refer to *external knowledge* as additional natural language text or code data that is not part of the original XMI diagrams. The exact sources, selection criteria, and preprocessing steps for this external data are described in Section 3.2.2 and Section 3.2.3.

Rozière et al. [43] proposed an approach for large-scale code completion and code generation based on fine-tuning LLM on code corpora, complemented by a controlled inclusion of natural language data such as documentation and descriptive text. Their results showed that augmenting the primarily code-focused training data with a small proportion (approximately $5\% - 8\%$) of NL samples improved the model's ability to understand instructions, capture semantic context, and perform code completion tasks more reliably, without degrading its core code modeling capabilities.

Inspired by this training strategy, we adopt a similar experimental approach. More specifically, we vary the proportion of external knowledge included during model training relative to our *main dataset* introduced in Section 3.2.2. Such experimental design allows us to systematically investigate whether limited exposure to complementary information can enhance an LLM's understanding of modeling semantics and improve its ability to infer and complete missing class diagram elements.

Thus, we consider four levels:

- 0% (baseline; UML-only),
- 5%, and
- 8%.

Each percentage refers to the proportion of external knowledge added relative to the size of the main training dataset. All configurations use the same validation and test sets, ensuring that only the training distribution varies across conditions.

Table 3.3: Summary of used variables.

RQ	Variable	Type	Description
	Proportion of external knowledge	Independent	The percentage of non-UML data added to the main dataset during fine-tuning (0%, 5%, 8%).
RQ1	Model performance	Dependent	How well the model completes the partial class diagrams after augmentation.
	Relative performance improvement	Dependent	Difference in performance between fine-tuned model configurations.
	Presence of natural language comments	Independent	Whether natural language comments are integrated in training data (within UML diagrams).
RQ2	Model performance	Dependent	How well the model completes the partial class diagrams when provided with natural language guidance.
	Relative performance improvement	Dependent	Performance change over the baseline configuration.

RQ2: Natural-Language Comments. The second independent independent variable is the presence of natural-language comments embedded into the XMI diagrams. These comments are generated automatically using a large language model and describe the general content or purpose of the respective UML class diagram.

We compare:

- Diagrams without comments (baseline), and
- Diagrams augmented with LLM-generated comments.

Dependent Variables. The primary dependent variable in this study is model performance, defined as the quality of the UML class diagram completions produced by the

fine-tuned LLM. As detailed in Section 3.4, performance is operationalized using a set of complementary evaluation metrics that capture structural correctness and overall reconstruction quality. These include Model Completion Accuracy, BLEU score as a text-based baseline for assessing structural similarity between predicted and ground-truth XMI representations, Cosine Similarity computed over embedded textual representations of the diagrams, Completion Rate reflecting the proportion of missing UML class diagram elements successfully reconstructed, and Recovery Rate measuring the extent to which removed diagram content is recovered relative to the complete ground-truth model.

The second dependent variable that we report is the relative performance improvement of each model variant with respect to the baseline configuration.

Formally, for an evaluation metric M, we compute relative improvement as:

$$\Delta_{\text{rel}}(M) = \frac{M_{\text{exp}} - M_{\text{base}}}{M_{\text{base}}} \times 100\%, \tag{3.1}$$

where:

- M_{exp} denotes the metric value obtained under a given *experimental condition*,
- M_{base} denotes the metric value achieved by the *base model*, and
- $\Delta_{\text{rel}}(M)$ then represents the percentage change relative to the baseline.

This dependent variable provides an intuitive and scale-independent assessment of how each augmentation strategy affects the model's performance. We define the base model used in this comparison and discussed it in detail in Section 3.2.5.

Controlled Variables. To ensure that the the results of each experiment are exclusively based on the manipulations of independent variables, we fix a number of experimental conditions that remain constant across all training runs. These controlled variables are summarized in Table 3.4. Controlling these variables is important for guaranteeing the internal validity and comparability of the results, as variations in model performance could otherwise arise from differences in model architecture, optimization settings, input preprocessing, or evaluation procedure rather than from the experimental manipulation itself. Thus, by keeping these factors constant, we isolate the impact of the independent variables on model performance.

Firstly, we fine-tune the same large language model, namely Llama-3.1 with 8 Billion parameters. Secondly, we maintain a strictly identical LLM configuration across all experiments, including the model architecture, parameter count, fine-tuning strategy, and hardware environment. This prevents confounding effects that can arise from model capacity or hardware-specific variability. Additionally, fine-tuning configuration, including learning rate, number of epochs, batch size, optimizer configuration, and quantization settings, is held constant to ensure that learning dynamics do not differ between experimental conditions. Moreover, all datasets go through the same preprocessing pipeline, including tokenization, handling of XMI syntax, and formatting of JSON instruction entries. This ensures that model inputs are uniform and prevents dataset preprocessing steps from affecting performance.

We further fix the validation and test sets for all experimental conditions. We do this with the intention of varying only the training distribution. This is crucial because even small

Table 3.4: The controlled variables that kept fixed across all experiments.

Controlled Variable	**Description**
Main dataset size	The size of the training set remains constant except for the manipulated portion of the secondary dataset when addressing the RQ1.
Model architecture	Llama-3.1-8B with PEFT.
Hyperparameters	Batch size, maximum sequence length, number of training epochs, learning rate, maximum number of training steps, random seed and further optimizer settings remain unchanged across for all experiments.
Fine-tuning configuration	Rank, target modules to apply QLoRA to, scaling factor, dropout rate, level of bias, and random state remain fixed.
Data splits	The size of the train, validation and evaluation splits remain unchanged in the 80/10/10 proportion.
Training data representation	The instruction template and formatting rules used in the JSON.
Validation and test sets	We keep both test and validation sets fixed among all the experiments to ensure fair evaluation and performance comparisons.
Inference configuration	The decoding parameters used during inference, maximum number of tokens that the fine-tuned model can generate as well as the inference template remain unchanged.

variations in test data composition could result to biased results. Finally, the evaluation metrics and matching procedure used to determine correctness of predicted UML elements remain unchanged for all experiments. By keeping these variables controlled, any observed differences in performance can be interpreted as consequences of the manipulated variables discussed earlier rather than as side effects of unrelated experimental factors.

3.3 Software and Hardware

This section specifies the software environment and hardware resources used to conduct all experiments. Maintaining a consistent software and controlled hardware configurations ensures reproducibility and comparability across the different experimental conditions.

3.3.1 Software

We implement the experiments using Python 3.11, with model training and inference performed using PyTorch 2.2 which provides optimized GPU support and seamless integration with PEFT. To access, load, and preprocess the additional textual datasets, we rely on the HuggingFace `datasets` library [29]. . For accessing, loading, and preprocessing the secondary datasets, we rely on HuggingFace[3] `datasets` library [29]. To preprocess the datasets into the model's supported format, fine-tune it and evaluate the predictions, we implement custom partitioning, parsings, fine-tuning, inferencing and evaluation scripts. The `Unsloth` [9] and `trl` [56] libraries were used for efficient fine-tuning of LLaMA models with LoRA-based parameter-efficient training and optimized support for quantized models.

3.3.2 Hardware

All experiments are executed on the university's High-Performance Computing (HPC) cluster. Each training run utilizes a single NVIDIA L40S GPU with 44.4 GB of VRAM. In addition, a minimum of 8 CPU cores and 100 GB of system memory are allocated per run to support data preprocessing, tokenization, and model loading in quantized form. The cluster nodes are equipped with AMD EPYC-7662 64-core processors.

This controlled hardware setup guarantees consistent runtime conditions across training runs and eliminates hardware variability as a potential confounding factor in the evaluation.

3.4 Metrics

To evaluate the predictions of the fine-tuned variants, we employ a suite of evaluation metrics. These metrics are designed to assess the performance of the fine-tuned LLM on the task of UML model completion from multiple perspectives, including as element-level

3 https://huggingface.co/

prediction accuracy, structural recovery, and semantic similarity between predicted and ground-truth class diagrams.

Specifically, we introduce another accuracy metric that measures the proportion of correctly reconstructed model elements relative to the total number of removed elements and hallucinated predictions. To further capture semantic similarity beyond exact matching, we compute BLEU [38] and Cosine Similarity scores between predicted and reference representations. Moreover, we introduce completion and Recovery Rates to quantify the amount of correctly predicted missing elements. Together, these metrics provide a balanced assessment of both the accuracy and the reliability of the generated class diagrams.

With respect to the research questions, the evaluation for RQ1 focuses on quantifying the impact of incorporating external general knowledge during fine-tuning on the model's completion performance. For RQ2, the same metrics are used to assess whether embedding NL comments within partial UML class diagrams improves the model's reconstruction capabilities. By applying an identical evaluation framework across all experimental conditions, we ensure that performance differences can be attributed directly to the respective data augmentation strategies.

This section describes the suite of evaluation metrics in more detail.

3.4.1 Model Completion Accuracy

We introduce the MCA as the primary evaluation metric to directly quantify the structural quality of the reconstructions produced by the fine-tuned variants. This metric measures the model's ability to correctly infer missing class diagrams elements, while penalizing hallucinated predictions. In the scope of this work, hallucinated predictions refer to UML model elements generated by the fine-tuned model that were not part of the original, complete class diagram prior to data preparation. That is, an element is considered hallucinated if it appears in the model's prediction but does not belong to the set of elements intentionally removed during the construction of the partial diagram.

MCA is defined as:

$$\text{MCA} = \frac{|E_{\text{correct}}|}{|E_{\text{removed}}| + |E_{\text{hallucinated}}|} \tag{3.2}$$

where:

- E_{removed} denotes the set of UML elements removed during data preparation,
- E_{pred} denotes the set of elements generated by the model,
- $E_{\text{correct}} = E_{\text{removed}} \cap E_{\text{pred}}$ the set of correctly reconstructed elements, and
- $E_{\text{hallucinated}} = E_{\text{pred}} \setminus E_{\text{removed}}$ denotes the set of hallucinated elements.

By construction, such accuracy metric rewards accurate recovery while punishing over-generation, making it well suited for structured software modeling tasks where hallucinated elements may affect model validity. An MCA value of 1 indicates perfect reconstruction without any hallucinated elements, whereas lower values reflect either incomplete recovery, or excessive over-generation, or both.

3.4.2 Semantic Similarity Metrics: BLEU, Cosine Similarity

To asses the semantic similarity between the predicted and the ground-truth XMI-encoded class diagrams, we employ BLEU [38] score and Cosine Similarity.

BLEU Score [38]. BLEU measures token-level overlap between serialized UML model representations, capturing surface-level similarity. In the context of UML model completion, it captures the extent to which the predicted XMI representation aligns with the ground-truth model at the syntactic level, including correct reproduction of element names, attributes, relationship labels, and structural keywords present in the XMI serialization. High BLEU values indicate that serialized XMI representations of the ground-truth and predicted models are syntactically similar, while low BLEU values point to surface-level divergence.

Cosine Similarity. Cosine Similarity measures the angular similarity between two vector representations in an embedding space [18]. In this work, both predicted and ground-truth UML representations are embedded into a vector form, and their Cosine Similarity is computed as:

$$\text{Cosine Similarity}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} \tag{3.3}$$

where

- x denotes the vector embedding of predicted UML models and
- y denotes the vector embedding of the ground-truth models

In comparison to the BLEU score, Cosine Similarity does not depend on exact token matches. It instead captures higher-level semantic alignment between the two representations, making it more robust to syntactic variations and reordering of elements. High Cosine Similarity values indicate strong semantic alignment between the predicted and ground-truth models, whereas the low values reflect the opposite.

Both of the metrics provide insight into the semantic similarity of the generated outputs, especially in such cases when predictions are not one-to-one identical but still provide similar information. Nevertheless, due to their limited sensitivity to structural correctness, we do not use them in isolation to judge model performance.

3.4.3 Completion and Recovery Rates

To further asses the behavior of the fine-tuned model variants, we employ Completion and Recovery Rates. These metrics provide complementary insight into how frequently the model attempts to reconstruct missing information and how effectively it is able to infer elements that were explicitly removed during the data preparation phase.

Completion Rate. The Completion Rate measures the extent to which the fine-tuned variant produces outputs relative to the size of the original complete diagram. In comparison

to the accuracy metric, the Completion Rate does not assess correctness but rather reflects the model's tendency to generate UML elements when presented with partial/incomplete input. Formally, we define it as:

$$\text{Completion Rate} = \min\left(\frac{|E_{\text{pred}}|}{|E_{\text{true}}|}, 1\right) \tag{3.4}$$

where:

- E_{true} denotes the set of UML elements present in the original, complete class diagram prior to the element extraction, and
- E_{pred} denotes the set of UML elements predicted by the fine-tuned model.

By construction, the Completion Rate has an upper-bound of 1. A Completion Rate of 1 indicates that the model generates at least as many class diagram elements as present in the original one, whereas lower values indicate poorer predictions. Since this metric does not account for correctness, high Completion Rate may still coincide with hallucinated predictions.

Recovery Rate. The Recovery Rate quantifies the proportion of removed class diagram elements that the fine-tuned model was able to reconstruct correctly. In contrast to the Completion Rate, this metric explicitly focuses on recovering missing information introduced during the data preparation phase.

The Recovery Rate is defined as:

$$\text{Recovery Rate} = \frac{|E_{\text{removed}} \cap E_{\text{pred}}|}{\max(|E_{\text{removed}}|, 1)} \tag{3.5}$$

where:

- E_{removed} denotes the set of UML elements removed from the original diagram,
- E_{pred} denotes the set of elements generated by the model
- $\max(|E_{\text{removed}}|, 1)$ ensures stability in cases where no elements were removed.

This design prevents the Recovery Rate from implicitly rewarding hallucinated predictions. A Recovery Rate of 1 indicates that all removed elements were successfully reconstructed, while lower values indicate partial or unsuccessful recovery. In comparison to the Completion Rate, this metric directly evaluates reconstruction effectiveness.

Put together, these rates provide information on different aspects of model behavior. A model may exhibit a high Completion Rate but a low Recovery Rate, showcasing frequent but inaccurate or hallucinated generation. From the other hand, a lower Completion Rate combined with a higher Recovery Rate suggests more selective yet accurate reconstruction behavior. Therefore, we interpret these metrics in conjunction with the MCA to obtain a comprehensive assessment of model performance.

3.4.4 Relative Performance Improvement

To facilitate direct comparison between different fine-tuned model variants, we report the relative performance improvement (Equation 3.1), with respect to the base model FTM_{base}. This metric quantifies the proportional change in performance for a given evaluation metric M and allows us to assess and compare the effects of different augmentation strategies in an interpretable manner independently of absolute performance.

A positive value of ΔM indicates that the variant model outperforms the base model for the chosen metric, demonstrating an improvement attributable to the applied augmentation strategy. A negative value of ΔM indicates a decrease in performance relative to the base model, suggesting that the augmentation may have had a detrimental effect on this aspect of model behavior. A value of zero indicates no measurable change compared to the baseline.

4

Evaluation

This chapter presents the experiments and the evaluation of the experimental results obtained from the fine-tuned model variants.

We first describe the experiments designed for both of the research questions.

Then, the results for RQ1 and RQ2 obtained during the evaluation phase of these experiments.

Finally, we discuss the implications of obtained results and outline potential threats to validity of the evaluation.

4.1 Experiments

This section integrates the components of the methodology (Chapter 3) and explains how the experiments are organized, executed, and evaluated. The main goal is to investigate the research questions introduced in Section 3.1 by systematically manipulating the independent variables while keeping all controlled variables fixed. The way we structure the experimental design allows to ensure that any differences in model performance arises solely from the data augmentation strategies rather than confounding factors.

We categorize the experiments into two families, corresponding to the two research questions. Both utilize the same experimental pipeline Figure 3.1 and rely on the data preparation workflow described in Figure 3.2. However, they diverge in the construction of their training data.

4.1.1 Experiment RQ1: External Knowledge Augmentation

Following the training data setup described earlier in Section 3.2.3 and the overall experimental pipeline introduced in Section 3.2.1, this section describes the standardized experimental procedure applied for RQ1. While the composition of the training data differs according to the respective experimental condition, the execution of fine-tuning, inference, and evaluation strictly follows the same procedure across all configurations.

Training Data Setup for the RQ1 (Pipeline Stages 1-2). For the first research question, we investigate the effect of gradually increasing proportions of external knowledge on the model's ability to complete partial UML class diagrams. After the data processing and partitioning step of the general pipeline, the main dataset contains (16321 partial XMI-encoded class diagrams as mentioned in Table 3.1. These entries originate from complete

diagrams processed in Phase 2 of the data preparation pipeline (Figure 3.2 illustrated in Section 3.2.3, where at least two model elements, e. g.,classes, attributes, operations, or associations are randomly removed to create partial inputs.

To construct the augmented variants of the baseline model, we extend the UML-only dataset with instruction-input-output triples sampled from the external datasets introduced in Section 3.2.2. For each one of these datasets, we randomly select subsets corresponding to 5%, and 8% of the size of the main UML training set. Each selected sample is then transformed into a unified JSON format compatible with the fine-tuning pipeline. The resulting external knowledge entries are then appended to the UML-only training set, forming progressively larger augmented datasets while leaving the validation and test sets unchanged. Listings 4.1 and 4.2 represent examples of such external triples from the Software Engineering [24] and Python [51] datasets.

These external samples are *unrelated* to the main dataset instead, they provide domain-independent linguistic and code context intended to improve generalization abilities.

After constructing the training datasets for the respective experimental condition, each model configuration (Section 3.2.5) is trained, evaluated, and documented using a standardized experimental procedure. This ensures that results across all configurations are directly comparable.

Listing 4.1: Example external knowledge entry from the Software Engineering dataset.

```
{
  "instruction": "How is a use case view represented in UML?",
  "output": "A use case view in UML is represented using use case diagrams. These diagrams
      model actors and their interactions with use cases. They may also include relationships
       such as include, extend, and generalization."
}
```

Listing 4.2: Example external knowledge entry from the Python dataset.

```
{
  "instruction": "Convert the following Java code to Python.",
  "input": "public int[] sumZero(int n) {\n    int[] res = new int[n];\n    for (int i = 1; i
      <= n / 2; i++) {\n        res[i - 1] = i;\n        res[n - i] = -i;\n    }\n    return
      res;\n}",
  "output": "def sum_zero(n):\n    res = []\n    for i in range(1, (n // 2) + 1):\n        res
      .append(i)\n        res.append(-i)\n    if n % 2 != 0:\n        res.append(0)\n
      return res"
}
```

LLM Fine-Tuning (Pipeline Stage 3). For each experimental condition, we load the same base LLaMA-3.1 model with 8 billion parameters and initialize the QLoRA fine-tuning setup. All hyperparameters, including optimizer settings, learning rate, gradient accumulation steps, number of epochs, batch size, quantization configuration, and LoRA parameters, are kept fixed across all runs, as summarized in Table 3.4. To ensure reproducibility, random seeds ($seed = 42$) are fixed for dataset sampling, data shuffling, and model initialization.

The model is then fine-tuned using the training dataset variant. For RQ1, this includes training on UML-only data as well as UML augmented with varying proportions of external

knowledge. To monitor training progress, we use the same fixed validation set. Early stopping is not applied, and all model configurations are trained for a fixed number of epochs ($epochs = 6$) to ensure consistent learning dynamics across conditions.

Inference and Prediction (Pipeline Stage 4). After fine-tuning, each model is used to generate predictions on a fixed test set consisting exclusively of partial UML class diagrams. As described in Table 3.1, the test set contains 1633 instances and remains identical across all experimental conditions. No external knowledge or natural language descriptions are injected at inference time, ensuring that all models are evaluated under the same conditions.

At the inference time, each fine-tuned model variants is prompted to complete the partial class diagram for the test set. The main input to the model is the serialized representation of this partial class diagram, in which specific elements (e.g., classes, attributes, or operations) have been intentionally removed at Pipeline Stages 1-2 (paragraph 4.1.1). We instruct the model provide the complete class diagram by reproducing the provided partial diagram and inserting the missing elements in the appropriate positions. As output, the model generates a completed version of the class diagram in XMI format, which includes both the originally provided elements and the newly inferred ones. This full generated diagram serves as the basis for the subsequent evaluation step.

Evaluation and Report (Pipeline Stage 5). Finally, all generated predictions are evaluated using the suite of metrics introduced in Section 3.4. These include MCA, BLEU, Cosine Similarity, Completion, and Recovery Rates. The evaluation procedure utilizes the same parsing, matching, and comparison logic for all experimental conditions.

Since the Completion Rate measures how many of the required elements the model generated in total, compared to how many should exist in the completed class diagram, the evaluation directly compares all the inferred elements against those elements that are present in the ground-truth class diagram. For MCA, Recovery Rate and semantic similarity metrics (BLEU Score and Cosine Similarity) the main evaluation focuses on the comparison between the elements inferred by the model and the corresponding removed elements from the ground truth, *excluding those elements that were already present in the partial input*. Performing the evaluation process in such way, allows us to better align the assessment with the objective of the class diagram completion task.

For completeness, we report and discuss separately an alternative computation of BLEU and Cosine Similarity scores. This alternative computation is based on comparing the *full generated diagram* against the *complete ground-truth representation*. It is important to note that these alternative results are not used as primary indicators, as full class diagram comparisons tend to underestimate semantic similarity values due to unavoidable deviations in ordering of elements, formatting, and non-critical variations in XMI serialization.

In addition to absolute performance metrics, we compute the relative performance improvement for each configuration, using the baseline model introduced in Section 3.2.5 as reference.

The primary objective of this evaluation procedure is to empirically assess the impact of the proposed data augmentation strategy on the task of UML class diagram completion and to answer the research questions.

4.1.2 Experiment RQ2: Integration of Natural Language Comments

This experiment follows the same overall experimental pipeline and execution procedure as the RQ1 setup described in Section 4.1.1. In particular, model architecture, fine-tuning configuration, preprocessing steps, validation and test splits, inference procedure, and evaluation metrics are kept identical.

The key distinction of the RQ2 experiment lies in the composition of the training data. While RQ1 focuses on augmenting the training set with external knowledge sources, RQ2 investigates the effect of integrating natural language comments that describe the UML model directly into the training instances. The following subsections detail the training data setup and experimental procedure specific to RQ2.

Training Data Setup for the RQ2 (Pipeline Stages 1-2). For the second research question, we examine whether enriching the UML class diagrams with natural language comments improves the model's ability to reconstruct missing elements. Following similar procedure as for the RQ1, we first preprocess and partition the main dataset, resulting in 16321 partial XMI-encoded class diagrams.

Next, we employ an auxiliary instruction-tuned model discussed earlier in Section 3.2.4 to generate descriptive summaries for each complete class diagram. To obtain these, we prompt the auxiliary model with instruction-class diagram tuples, asking it to produce a concise, high-level description of the given XMI-encoded model. The resulting summaries provide high-level contextual information about the structure and intent of the class diagram, potentially supporting the model's reasoning during reconstruction. Listing 4.3 shows an example of such a prompt, while Listing 4.4 demonstrates a corresponding generated comment.

During Phase 2 of the data preparation pipeline (Figure 3.2), the generated comments are injected directly into the XMI representation of the corresponding class diagrams. They are embedded as `<ownedComment>` elements in accordance with the XMI specification, preserving structural validity and ensuring compatibility with the custom parsing tools used throughout this work.

Following the injection step, each commented diagram is processed in the same manner as the UML-only data. First, elements are randomly removed to create partial diagrams. Then, the resulting pairs are transformed into unified instruction-input-output JSON triples. These steps ensure that the UML-only and the UML+NL comment datasets share an identical format, differ solely in the presence of natural language annotations, and pass through the same tokenization and formatting routines.

To construct the training dataset for the comment-augmented condition, all JSON triples generated from commented diagrams are collected and shuffled using fixed random seeds to ensure reproducibility. The validation and test sets remain strictly UML-only and contain no injected comments, ensuring that any observed performance differences arise from the availability of additional natural language context during training. Such design allows us to isolate and quantify the contribution of embedded NL descriptions to the model's ability to infer missing elements of the UML class diagrams.

Listing 4.3: Example prompt used to obtain natural-language comments for a UML class diagram.

```
{
  "instruction": "You are a software engineering expert analyzing UML class diagrams.

    Given the following UML class diagram structure, provide a BRIEF (2-3 sentences) natural
        language description of what this diagram represents. Focus on:
    1. The main purpose/domain of the system
    2. Key classes and their roles
    3. Main relationships between classes

    DO NOT list all classes, their attributes, or relationships between them. Instead, provide
         a high-level summary that captures the essence of the design.

    UML Diagram Structure:",
  "input": <class diagram>
}
```

Listing 4.4: Example natural-language comment generated for a UML class diagram.

```
{
  "comment": "This UML class diagram represents an e-commerce system where customers can
      browse, add, and manage products in their shopping carts. Users can authenticate and
      place orders, which are composed of order details and associated with specific products
      . The system also supports the creation of new customers, who are a specialized type of
       user"
}
```

LLM Fine-Tuning (Pipeline Stage 3). Following the same steps as for the RQ1, we load the same base LLaMA-3.1 model with 8 billion parameters and initialize the QLoRA fine-tuning setup and keep all the hyperparameters fixed. Similarly, we use the fixed validation set is used to monitor training progress.

The key difference at this stage lies in the composition of the training data. More precisely, for RQ2, the training data is augmented with natural language comments.

Inference and Prediction (Pipeline Stage 4). After fine-tuning, the resulting model is applied to the same fixed test set of partial UML class diagrams used across all experiments. This test set contains no natural language comments or external knowledge and remains unchanged for all configurations. During inference, both *base model* and model augmented with NL comments are tasked with predicting and reconstructing the missing UML elements based on the partial diagram input, ensuring a fair and consistent evaluation setting. Similarly to the inference process described for RQ1, the fine-tuned model is prompted to complete the partial class diagram by reconstructing the missing UML elements, and to provide the fully complete class diagram that contains both, the inferred elements and the elements that were already present in the partial input, as an output.

Evaluation and Reporting (Pipeline Stage 5). In the same manner as for the experiment for RQ1, the generated predictions are evaluated using the same set of structural and semantic metrics described in Section 3.4. We again apply a uniform parsing, matching,

and comparison procedure. Additionally, we report absolute performance scores as well as the relative performance improvements with respect to the baseline configuration. For completeness, we also report an alternative computation of BLEU and Cosine Similarity scores for RQ2, where the *full generated class diagram* is compared against the *complete ground-truth*. These results are discussed separately and are not used as primary indicators of model performance.

It is important to notice, that each experimental condition is executed independently, meaning that no model continues training from a previous configuration. This allows us to ensure a clean experimental separation between the different data augmentation strategies.

4.2 Results

This section presents results of our experimental study, structured according to the two research questions introduced earlier in Section 3.1.

We asses all model variants under identical conditions using a fixed test set and the same evaluation strategy described in Section 4.1, ensuring that observed performance differences can be attributed exclusively to the manipulated independent variables. We analyze model performance using the metric suite introduced in Section 3.4, with a particular emphasis on the proposed MCA metric and relative performance improvements.

We first report results for RQ1, which investigates the impact of external knowledge augmentation, followed by RQ2, which examines the integration of natural language comments into partial UML class diagrams.

For each research question, we report absolute performance metrics introduced in Section 3.4, relative performance improvements with respect to the baseline configuration, and analyze their respective distributions using violin plots. In addition, we report and discuss an alternative computation of semantic similarity metrics (Cosine Similarity and BLEU score) to highlight methodological considerations specific to structural modeling tasks. Finally, we analyze common failure cases observed across configurations and summarize the main findings.

Unless stated otherwise, higher metric values indicate better model performance. To facilitate comparability, both absolute performance scores and relative performance improvements with respect to the baseline model FTM_{base} are reported.

All reported results are computed on the fixed held-out test set as discussed in Section 3.2.3.

4.2.1 Results RQ1: External Knowledge Augmentation

To answer the first research question, we assess the impact of integrating external general knowledge into the fine-tuning process on the model's ability to complete partial UML class diagrams.

Specifically, we compare the baseline model FTM_{base}, trained exclusively on UML data, with a series of augmented variants FTM^{p}_{EK}, where increasing proportions of external knowledge are added to the training set.

All models are evaluated on the same fixed test set to ensure comparability of the results. We asses the performance of each of the model configuration using five evaluation metrics: Completion Rate, Recovery Rate, Cosine Similarity, MCA, and BLEU Score. These metrics capture structural completeness, partial correctness, semantic similarity, and exact reconstruction quality, respectively. Additionally, we report the relative performance improvement ΔM (Equation 3.1) with respect to FTM_{base} to quantify the magnitude of observed improvements or degradations.

Unless stated otherwise, the reported BLEU and Cosine Similarity scores in this subsection are computed over the set of *predicted missing elements only*, excluding elements already present in the partial input diagram. We compare this set *predicted missing elements* to the set of the elements removed from the ground truth and compute the semantic similarity metrics. This formulation aligns semantic similarity evaluation with the completion task and avoids penalizing models for differences in unchanged structural components.

An alternative formulation, where semantic similarity is computed by comparing the *complete generated class diagram* against the *full ground-truth representation*, is discussed separately in paragraph 4.2.1. While informative, these alternative results are not treated as primary, as they conflate completion quality with the model's ability to reproduce elements that are already provided as an input at the inference time.

Table 4.1: Mean evaluation metrics for all model configurations.

Model	Completion Rate	Recovery Rate	Cosine Similarity	MCA	BLEU
FTM_{base}	**0.6152**	0.3630	0.7249	0.0983	0.5430
FTM^{5}_{wiki}	0.6001*	0.3508*	**0.7263**	0.0968	**0.5476**
FTM^{8}_{wiki}	0.6090	0.3549	0.7248	0.0972	0.5440
FTM^{5}_{se}	0.6148	**0.3908***	0.7116	0.1048	0.5294
FTM^{8}_{se}	0.6088	0.3504	0.7182	0.0952	0.5354
FTM^{5}_{code}	0.6111	0.3876	0.7121	**0.1056**	0.5338
FTM^{8}_{code}	0.6056	0.3616	0.6025*	0.0978	0.4203*

Numbers in **bold** indicate the highest mean score in a given metric. The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

Absolute Performance Trends. Table 4.1 summarizes the absolute performance of all model configurations (Table 3.2) evaluated under RQ1, including the base model (FTM_{base}) trained exclusively on UML class diagrams and the augmented model variants trained with 5%and 8% of external knowledge samples from datasets introduced in Section 3.2.2.

The baseline (FTM_{base}) achieves a Completion Rate of 0.6152 and a Recovery Rate of 0.3630, Cosine Similarity of 0.37249, MCA of 0.0983 and a BLEU score of 0.5430 serving as the reference point for subsequent comparisons.

To assess whether observed differences are statistically meaningful, we conduct paired Wilcoxon signed-rank test between each augmented configuration and the baseline model using a significance level of $\alpha = 0.05$.

Across augmented configurations, we observe numerical differences for several metrics. However, the statistical analysis indicates that most of these differences are not statistically significant.

In particular, for MCA, none of the augmented configurations yield statistically significant differences compared to the FTM_{base} ($p > \alpha$ for all configurations). Even though FTM^5_{se} and FTM^5_{code} achieve higher results (0.1048 and 0.1056, respectively) compared to the base model (0.0983), they still remain nonsignificant. This suggests that despite small numerical fluctuations, external knowledge augmentation does not lead to significant improvement in structural accuracy as measured by MCA across test instances.

For the Completion rate, we observe a statistically significant decrease for FTM^5_{wiki} configuration ($p = 0.002 < \alpha$). This indicates that inclusion of 5% of general knowledge can negatively affect the LLM's ability to generate at least as many class diagram elements as present in the ground-truth model. Other augmented configurations do not show any statistically significant difference for this me metric when compared to the base model.

Furthermore, statistically significant improvement in Recovery Rate is observed for FTM^5_{se} ($p = 0.049 < \alpha$), which achieves the highest mean Recovery Rate of 0.3908 among all the configurations. This suggests that software engineering-related external knowledge can can support more accurate recovery of missing class diagram elements. On the other hand, FTM^5_{wiki} shows a significant decrease ($p = 0.043 < \alpha$), achieving the lowest mean Recovery Rate of 0.3508 and indicating that general external knowledge at lower augmentation levels can negatively affect the model's ability to infer and recovering missing elements.

For Completion Rate, no statistically significant differences are observed for most configurations, with the exception of FTM^8_{code}, which exhibits a statistically significant decrease ($p \ll \alpha$). This model configuration achieves the lowest average Cosine Similarity score of 0.6025 in comparison to the base model (0.7249), FTM^5_{wiki} with the highest Cosine Similarity value (0.7263) and all other configurations. This indicates that augmenting LLM with higher percentage of code-oriented knowledge may negatively affect semantic similarity at the embedding level.

Similarly, BLEU scores remain largely stable across configurations. Here, we again observe a statistically significant decrease for FTM^8_{code} ($p \ll \alpha$), reinforcing the observation that higher augmentation levels can adversely affect surface-level similarity without necessarily improving structural correctness.

In summary, there results indicate that augmenting UML-specific data with external knowledge during fine-tuning has limited and highly selective effects on LLM performance. Even Though, we have observed numerical differences across several metrics and model configurations, the statistical analysis reveals that most of these differences are not statistically significant at $\alpha = 0.05$.

Relative Performance Trends. In addition to absolute scores, we analyze the relative performance change of each augmented model configuration with respect to the baseline model (FTM_{base}). Relative improvement is computed for each evaluation metric using Equation 3.1 and as described in Section 3.4, allowing us to assess the magnitude of gains attributable solely to the introduced external knowledge.

Table 4.2: Relative performance change over the baseline model (FTM_{base}).

Model	Completion Rate	Recovery Rate	Cosine Similarity	MCA	BLEU
FTM_{base}	-	-	-	-	-
FTM^{5}_{wiki}	▼ 2.45%*	▼ 3.36%*	▲ 0.19%	▼ 1.53%	▲ 0.85%
FTM^{8}_{wiki}	▼ 1.01%	▼ 2.23%	▼ 0.01%	▼ 1.12%	▲ 0.20%
FTM^{5}_{se}	▼ 0.07%	▲ 7.66%*	▼ 1.83%	▲ 6.61%	▼ 2.50%
FTM^{8}_{se}	▼ 1.04%	▼ 3.47%	▼ 0.92%	▼ 3.15%	▼ 1.40%
FTM^{5}_{code}	▼ 0.66%	▲ 6.78%	▼ 1.77%	▲ 7.43%	▼ 1.69%
FTM^{8}_{code}	▼ 1.56%	▼ 0.39%	▼ 16.89%*	▼ 0.51%	▼ 22.60%*

Upward green triangles (▲) indicate a relative improvement over the baseline, while downward red triangles (▼) indicate a relative decrease. Values without symbols denote no measurable change relative to FTM_{base}. The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

Table 4.2 reports the relative performance improvements of each augmented configuration compared to FTM_{base}. We asses the statistical significance of the relative change using a paired Wilcoxon signed-rank tests at a significance level of $\alpha = 0.05$

Overall, relative performance gains remain modest across all metrics.

For the primary metric, MCA, none of the augmented configurations exhibit statistically significant relative improvements over the baseline ($p > \alpha$ for all comparisons). Although, FTM^{5}_{se} and FTM^{5}_{code} result in the relative improvement in MCA of 6.61% and 7.43%, respectively, these gains are small on magnitude and do not reach the level statistical significance. This indicates that external knowledge augmentation does not yield consistent and high enough to be statistically significant MCA gains when evaluated relative to the baseline model.

For Completion Rate, relative improvement trends further highlight potential negative effect of external data augmentation. Specifically, FTM^{5}_{wiki} shows a statistically significant relative decrease ($p = 0.002 < \alpha$) of 2.45%. All other relative Completion Rate differences indicate a decrease relative to the FTM_{base} and remain statistically non-significant.

We observe a different patter in Recovery Rate. The FTM^{5}_{se} configuration demonstrates a statistically significant relative improvement ($p = 0,49 > \alpha$) of 7.66%, which also corresponds to the highest relative improvement observed for Recovery Rate among all models. This suggests that including low percentage of software engineering-related knowledge to the main training set can positively contribute to recovering missing class diagram elements. On the opposite side, FTM^{5}_{wiki} shows a statistically significant relative decrease ($p = 0.043 < \alpha$) in Recovery Rate.

Finally, for semantic similarity metrics, the relative changes remain mostly non-significant. The FTM^{8}_{code} configuration exhibits a statistically significant relative decrease for both of the metrics (Cosine Similarity and BLEU) ($p \ll \alpha$) and achieves the lowest average Cosine Similarity compared to all other models. The same applies for BLEU score, where FTM^{8}_{code} demonstrates the relative decrease by 22.60%.

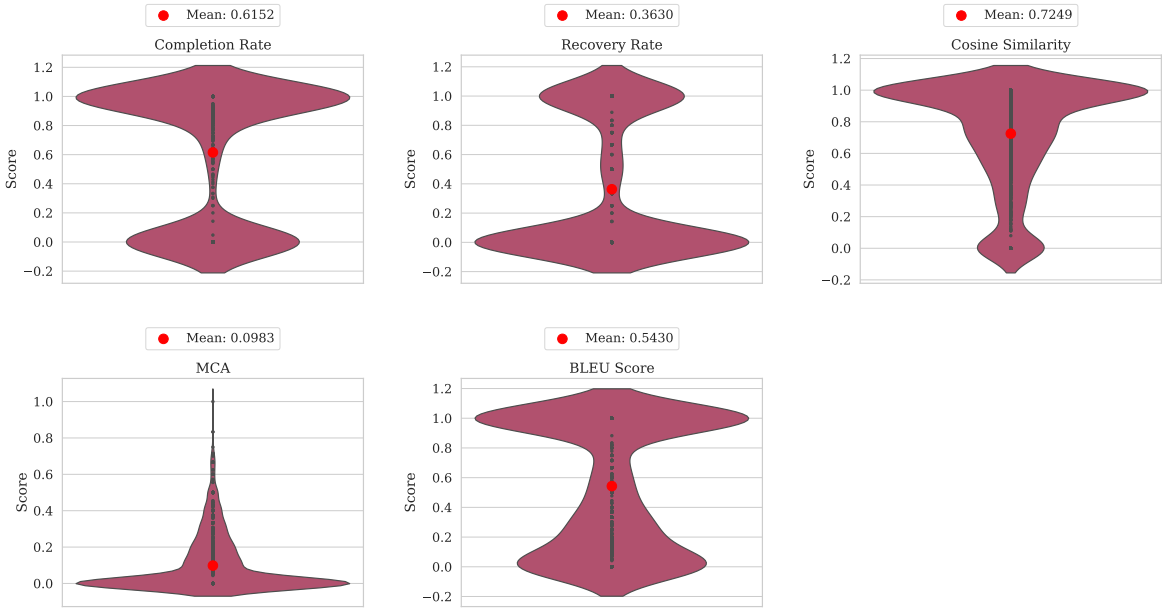


Figure 4.1: Distribution of per-instance values of all metrics for FTM_{base}.

Overall, these results indicate that the effects of external knowledge augmentation are limited, configuration related and often remain statistically non-significant. Even though we have observed isolated improvements, these gains do not generalize across metrics or the levels of augmentation. Importantly, several statistically significant effects correspond to performance degradation rather than improvement.

Distributional Analysis. In this paragraph we analyze the distributions of each metric across all the model configurations and discuss whether the observations align with the numerical results obtained earlier.

Figure 4.1 displays the per-instance metric distributions for the base model (FTM_{base}).

Since Completion Rate by definition (Section 3.4.3) is bounded by 0 and 1, this constraint is directly reflected in its respective distribution. We observe a hight density of values close to 1.0, 1015 out of 1633 instances lying above the mean Completion Rate (0.6152), This indicates that, for a majority of test cases, the base model is able to generate at least as many class diagram elements as present in the ground-truth diagrams. The remaining 618 instances concentrate near lower values.

For the Recovery Rate, only 630 out of 1633 values exceed the mean of 0.3630, revealing a complete opposite performance as for Completion Rate. Although the model frequently generates additional elements that are not present in the ground-truth class diagrams, it is able to recover the correct removed elements in fewer than half of the cases.

We observe a similar pattern in the MCA distribution, where only 566 out of 1633) instances achieve values above the mean (0.0983), suggesting that the baseline model struggles to provide accurate predictions, which can be due to the presence of hallucinated elements, which are heavily penalized by MCA (Section 3.4.1).

In contrast, semantic similarity metrics show approximately similar trends around their means. The Cosine Similarity distribution is centered at a comparatively high mean of

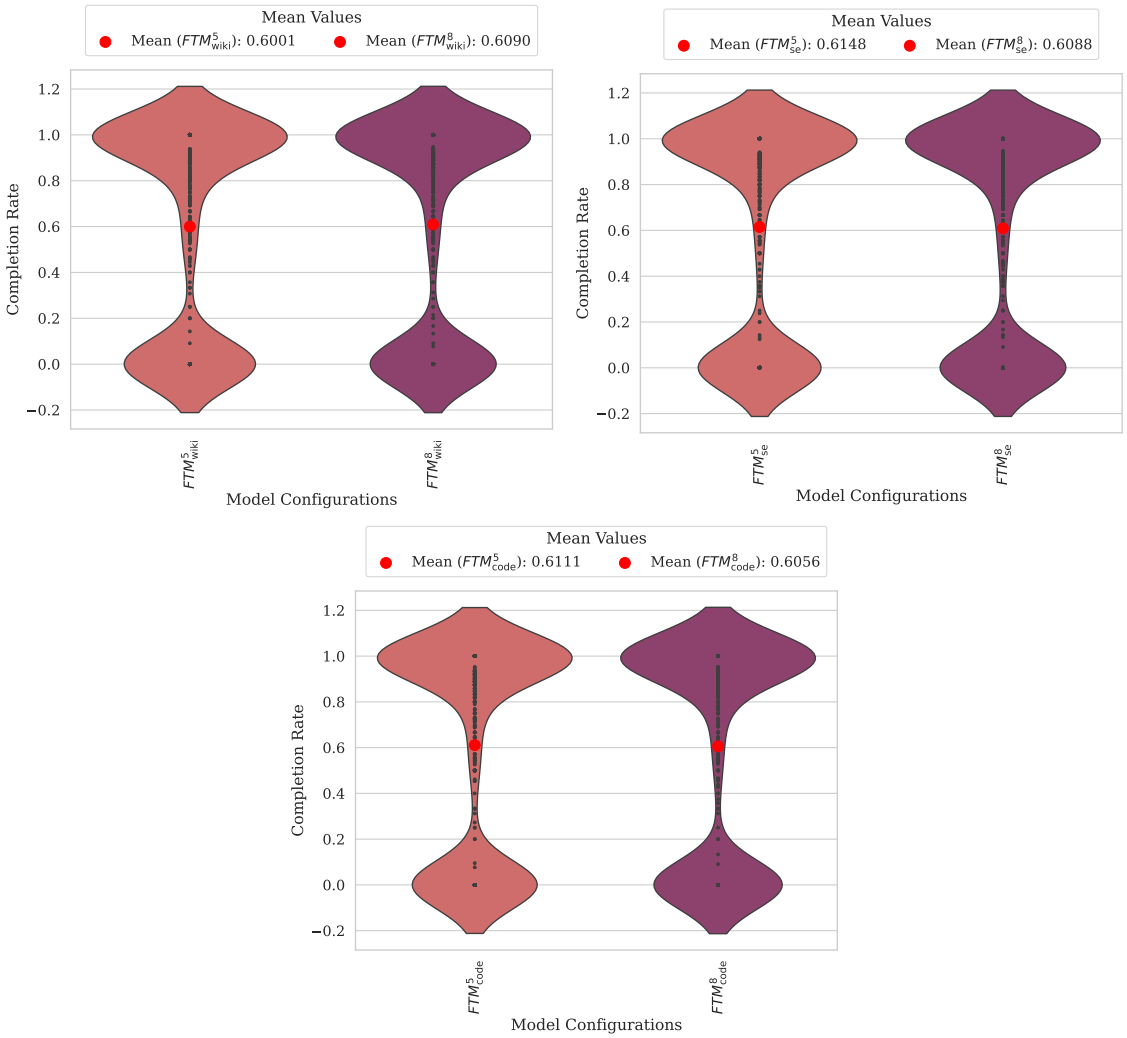


Figure 4.2: Distribution of per-instance values of Completion Rate for each augmented model configuration.

0.7249, with out of 1023 values exceeding this mean. For BLEU Score, 807 instances lie above the mean of 0.5430. These distributions suggest that the generated outputs often remain lexically or semantically aligned with the missing elements that were removed from ground-truth representations.

Now, we discuss the distributions of the augmented models for each of the evaluation metric.

Figure 4.2 illustrates the distribution of per-instance Completion Rate values for the augmented configurations. We observe that for FTM^5_{wiki} and FTM^8_{wiki} configurations, more than 50% of the instances (979 and 997, respectively) lie above the mean of 0.6001. Even though there is a notable concentration of instances above the mean value, the overall distribution for FTM^5_{wiki} is shifted slightly toward lower values compared to the Completion Rate distribution of base model. This observation is consistent with the statistically significant decrease in Completion Rate observed for this configuration $p = 0.002 > \alpha$. For the model configurations augmented with software engineering-related (FTM^5_{se}, FTM^8_{se})

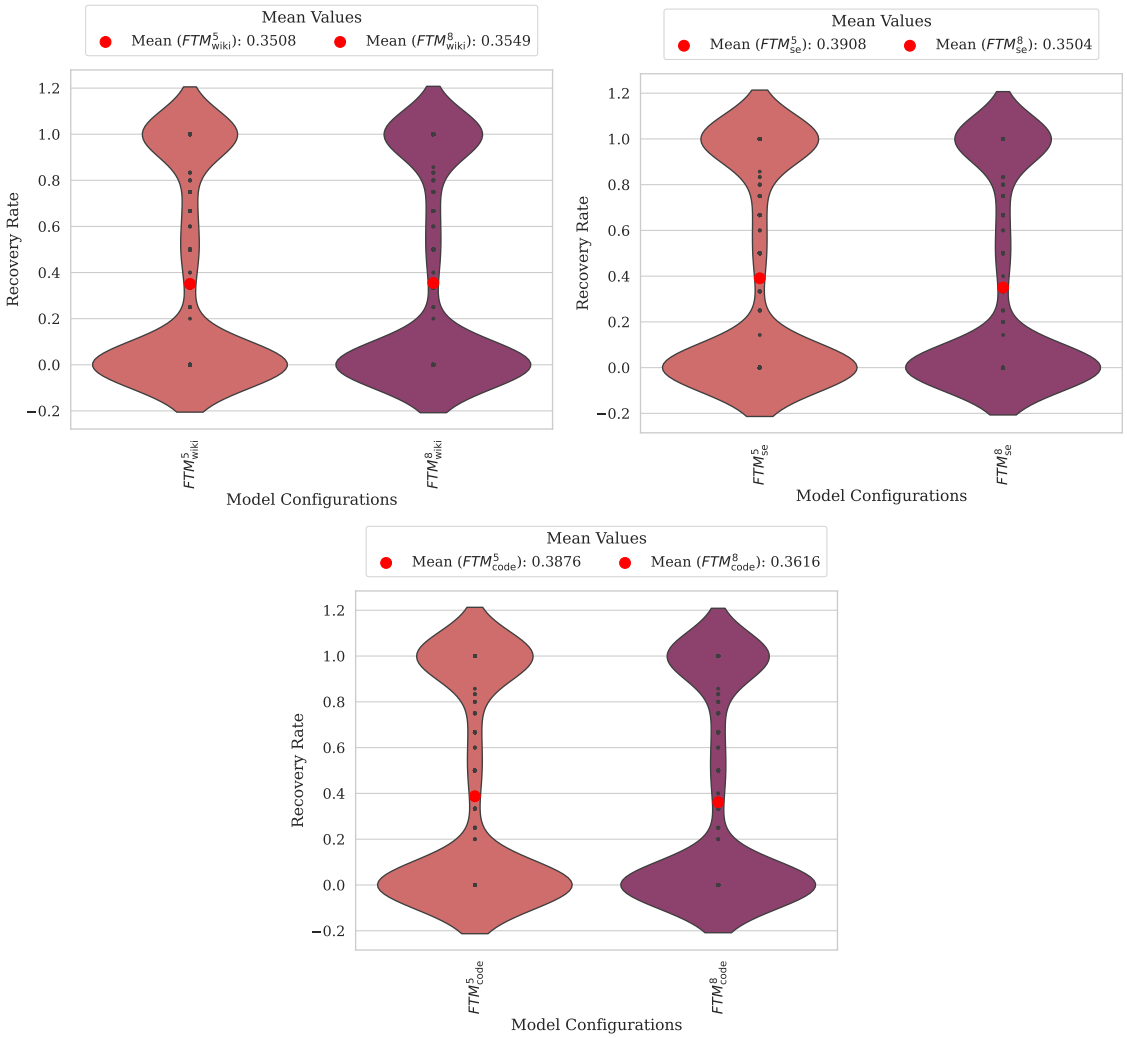


Figure 4.3: Distribution of per-instance values of Recovery Rate for each augmented model configuration.

data and code-related data (FTM^5_{code}, FTM^8_{code}), over 1000 instances lie above the respective mean Completion Rates. However, the shapes of these distributions remain highly similar to that of the base model, having dense regions are the lower and upper boundaries. This visual similarity is reflected in the statistical analysis, where no significant differences in Completion Rate are observed for these configurations.

Figure 4.3 presents the distribution of per-instance Recovery Rate values for each of the augmented configuration. We observe that for all the model configurations most of the instances are clustered below the respective mean values. However, for the FTM^5_{se} and FTM^5_{code} configurations a more dense upper bulb than for the other augmented configurations can be observed, where for FTM^5_{se} 672 instances lie above the mean value of 0.3908 and for FTM^5_{code} 666 out of 1633 instances lie above the mean value of 0.3876. The distributional shift for FTM^5_{se} aligns with the paired Wilcoxon signed-rank test results, where this model variant is the only one that exhibits a statistically significant improvement in Recovery Rate compared to the baseline $p = 0.049 < \alpha$. In contrast, the FTM^5_{wiki} configuration exhibits a

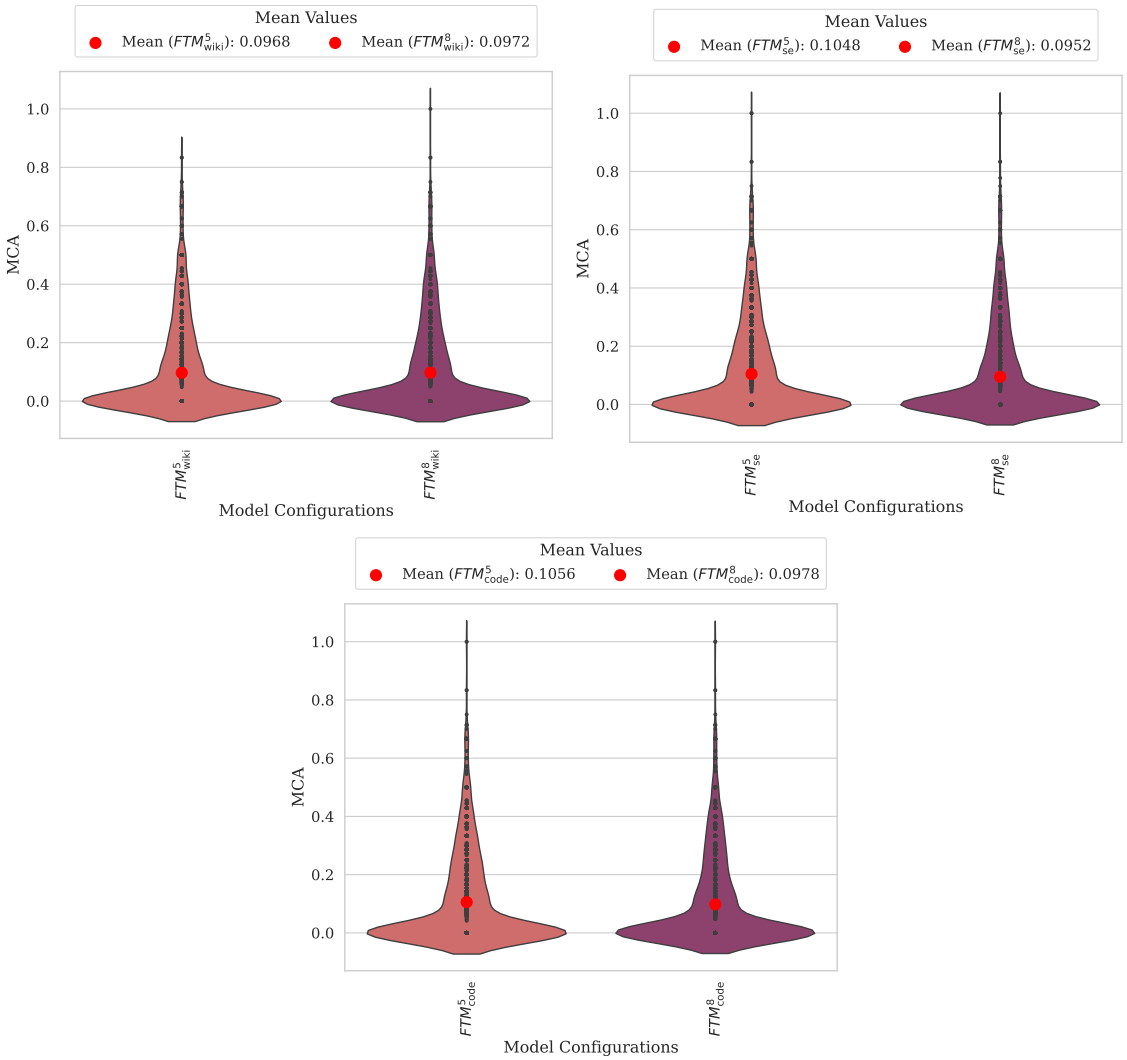


Figure 4.4: Distribution of per-instance values of MCA for each augmented model configuration.

comparatively heavier lower bulb and fewer instances above the mean Recovery Rate (n) of 0.3508. This behavior is consistent with the statistically significant decrease observed for this configuration.

Figure 4.4 shows the distribution of per-instance MCA scores for all augmented model configurations. In contrast to Completion and Recovery Rates, the MCA distributions are more concentrated in the lower score range, where most predictions cluster close to 0.0.Such distributional pattern aligns with the way this metric is defined (Section 3.4.1), penalizing the predictions where the expected class diagram elements are missing or that contain hallucinated elements. When we compare the shape of the distributions to the MCA distribution of the base model, we notice that it generally remains unchanged and exhibits no visible shift towards the higher values. This observation aligns with the statistical analysis, which indicates that none of the augmented models achieve a statistically significant improvement in MCA over the baseline ($p > \alpha$).

The distribution plots for BLEU and Cosine Similarity are provided in Figure 4.5.

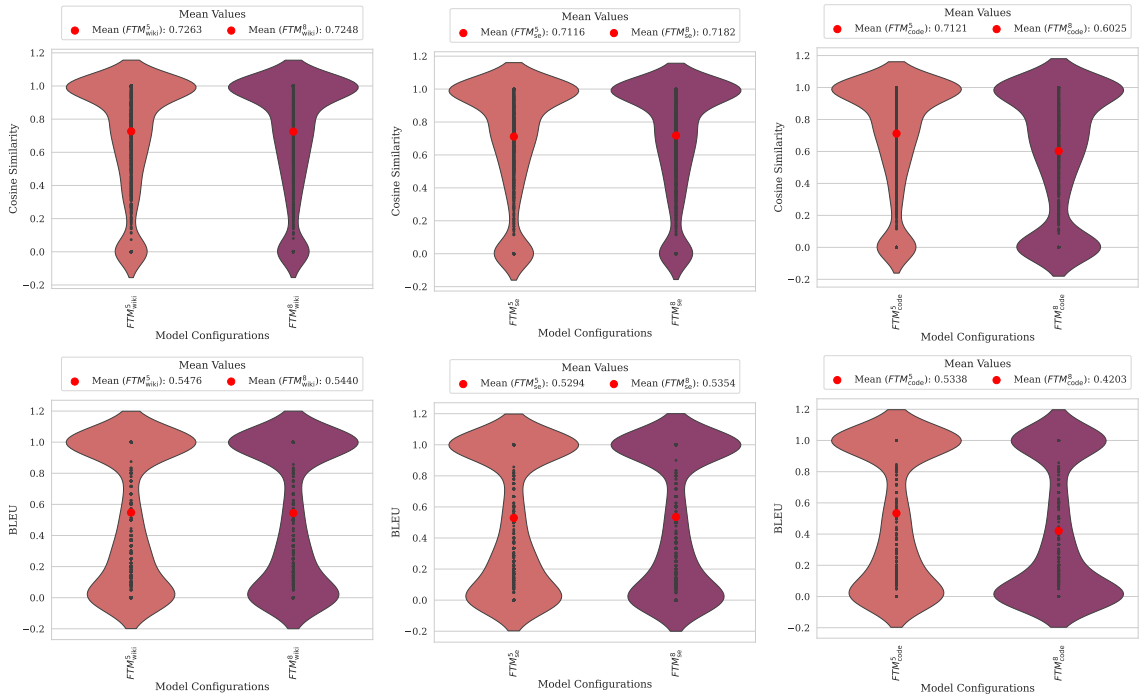


Figure 4.5: Distribution of per-instance values of BLEU and Cosine Similarity scores for each augmented model configuration.

For the Cosine Similarity, we observe that for all of the augmented model configurations, most of the instances lie above the respective mean values, indicating generally stable Semantic Similarity behavior across models. The shape of most of the distributions remain similar to the Cosine Similarity distribution of the base model, with a dense upper bulb near values close to 1.0. An exception to that is the FTM^8_{code} configuration, which has the most dense lower bulb as well as a substantially lower mean Cosine Similarity of 0.6025 compared to other configurations. This observation aligns to the statistical analyses, where we have observed a statistically significant decrease ($p \ll \alpha$) in Cosine Similarity for this configuration.

For BLEU, we observe a differently spaded distribution, where instances concentrated both near low scores and near values close to 1.0. Similarly, most of augmented configurations exhibit distributions comparable to the base model's BLEU distribution. However, the FTM^8_{code} model again stands out, displaying a visibly lower mean BLEU score of 0.4203 and a stronger concentration of low-scoring instances. This behavior aligns with the statistically significant decrease observed for BLEU in this configuration ($p \ll \alpha$).

Alternative Computation of Semantic Similarity Metrics. To assess the sensitivity of semantic similarity metrics to their formulation, we additionally evaluate BLEU and Cosine Similarity using a more traditional setup in which the *complete generated class diagram* is compared against the *full ground-truth representation* without isolating the elements that were removed and reconstructed during inference. The average results obtained under this alternative setup are summarized in Table 4.3.

Table 4.3: Average values of alternatively computed Cosine Similarity and BLEU for all model configurations.

Model	Alternative Cosine Similarity	Alternative BLEU
FTM_{base}	0.2959	0.0235
FTM^{5}_{wiki}	0.2951	0.0237
FTM^{8}_{wiki}	**0.2965**	0.0235
FTM^{5}_{se}	0.2929*	**0.0238**
FTM^{8}_{se}	0.2957	0.0233
FTM^{5}_{code}	0.2900*	0.0237
FTM^{8}_{code}	0.2476*	0.0206*

Numbers in **bold** indicate the highest mean score in a given metric. The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

Overall, the absolute values of both alternative BLEU and Cosine Similarity scores remain low and do not exhibit major fluctuations across model configurations. Such behavior can be explained with the nature of the task itself. Class diagram completion does not require the model to reproduce the entire ground-truth class diagram, but rather to infer missing structural elements provided a partially observed input. Consequently, the model's predictions that contain differences in ordering, formatting, or representation of elements that were already present at inference time can disproportionately affect the similarity metrics, even when the reconstructed elements are structurally correct.

Even though the fluctuations are minor, we still observe statistically significant differences. Both, FTM^{5}_{se} and FTM^{5}_{code} configurations demonstrate statistically significant decrease in alternative Cosine Similarity (both achieving $p < \alpha$), while FTM^{8}_{code} exhibits a statistically significant decrease ($p \ll \alpha$) in both alternative Cosine Similarity and BLEU score achieving the lowest average values (0.2476 and 0.0206, respectively) across all configurations.

We report the alternative semantic similarity results for completeness and do not use them as primary indicators for answering RQ1. Instead, the main analysis relies on a restricted formulation of BLEU and Cosine Similarity that evaluates only the *reconstructed missing elements*, thereby better aligning semantic similarity evaluation with the actual objective of the class diagram completion task. This helps us to ensure that the reported results focus on the quality of inferred content rather than on superficial similarities across full diagram representations.

Failure Cases. Importantly, we observe failure cases in the form of malformed predictions that cannot be parsed as valid XMI representations. These malformed predictions include XMI syntactic errors and incomplete or inconsistent structural definitions, which prevent successful parsing. As a result, such predictions receive 0 scores across all evaluation metrics.

Table 4.4 summarizes the number of successful and failed predictions for each model configuration. A prediction is considered successful if it can be parsed as a valid XMI string and subsequently evaluated using the defined metrics.

Table 4.4: Number of malformed (failed-to-be-parsed) predictions per model configuration.

Model Configuration	Successful Predictions	Failed Predictions	Failure Ratio
FTM_{base}	1633	0	0.00%
FTM^{5}_{wiki}	1633	0	0.00%
FTM^{8}_{wiki}	1633	0	0.00%
FTM^{5}_{se}	1588	45	2.75%*
FTM^{8}_{se}	1625	8	0.50%*
FTM^{5}_{code}	1582	51	3.12%*
FTM^{8}_{code}	1421	212	13.00%*

The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

We observe, that the base model (FTM_{base}) and the models augmented with general knowledge from the Wiki dataset [16] (FTM^{5}_{wiki}, FTM^{8}_{wiki}) produce no malformed outputs, resulting in a failure ratio of 0.00%.

On the other hand, the models augmented augmented with software engineering- and code-related data achieve higher failure ratios. Particularly, FTM^{5}_{se} and FTM^{5}_{code} have failure ratios of 2.74% and 3.12%, respectively, while FTM^{8}_{code} exhibits the highest failure ratio at 13.00%, which corresponds to 212 out of 1633 malformed predictions.

Using paired McNemar tests with the baseline model as reference and a significance level of $\alpha = 0.05$, we find that all non-zero failure ratios marked in Table 4.4 with "*" represent statistically significant increases in malformed predictions. This indicates that the observed rise in number of failed predictions produced by the software engineering– and code-augmented model configurations is not due to random variation, but rather reflects a systematic degradation in the validity of the output.

4.2.2 Results RQ2: Integration of Natural Language Comments

In RQ2, we assess whether integrating natural language comments that describe the general intent and high-level structure of a class diagram directly into the XMI-encoded representation affects the model's ability to complete partial UML class diagrams. For that, we compare the baseline model FTM_{base}, trained exclusively on UML data, with a model trained on the same dataset augmented with injected NL comments (FTM_{NL}).

Both models are evaluated on the same fixed test set to ensure comparability. As in RQ1, performance is assessed using Completion Rate, Recovery Rate, Cosine Similarity, MCA, and BLEU Score. In addition to absolute scores, we report relative performance improvements ΔM (Section 3.4.4) with respect to FTM_{base}.

Unless stated otherwise, the reported BLEU and Cosine Similarity scores in this subsection are computed over the set of *predicted missing elements only*, excluding elements already present in the partial input diagram. We compare this set *predicted missing elements* to the set of the elements removed from the ground truth and compute the semantic similarity

metrics. Similarly to the results of RQ1, we report alternative semantic similarity scores in paragraph 4.2.2 for completeness. Even though the alternative results are informative, these are not treated as primary, as they conflate completion quality with the model's ability to reproduce elements that are already provided as an input at the inference time.

Table 4.5: Mean evaluation metrics for FTM_{base} and FTM_{NL} configurations.

Model	Completion Rate	Recovery Rate	Cosine Similarity	MCA	BLEU
FTM_{base}	0.6152	0.3630	0.7249	0.0983	0.5430
FTM_{NL}	0.5182*	0.2746*	0.6762	0.0786*	0.5243*

The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

Absolute Performance Trend. Table 4.5 summarizes the mean absolute performance of FTM_{base} and FTM_{NL}. As discussed in Section 4.1, both configurations are evaluated on the same fixed test set of partial UML class diagrams and under identical inference conditions.

The baseline model achieves a Completion Rate of 0.6152, a Recovery Rate of 0.3630, Cosine Similarity of 0.7249, MCA of 0.0983, and a BLEU of 0.5430. These values serve as the reference point for assessing the impact of incorporating natural language comments during fine-tuning.

In contrast, the model augmented with NL comments exhibits consistently lower mean absolute performance across all metrics. Specifically, Completion Rate decreases to 0.5182, Recovery Rate to 0.2746, MCA to 0.0786, and a BLEU to 0.5243. All of these observed decreases are statistically significant at $\alpha = 0.05$ with $p \ll \alpha$, as reported in Table 4.5. Cosine Similarity also shows a numerical decrease from 0.7249 to 0.6762, although this decrease is not statistically significant since $p > \alpha$.

These results indicate that augmenting fine-tuning data with natural language comments negatively affects the model's ability to reconstruct missing class diagram elements and maintain structural accuracy. In particular, the statistically significant reductions in MCA, Completion and Recovery Rates suggest that the inclusion of NL comments interferes with the LLM's ability to accurately infer and reproduce missing UML class diagram elements, rather than supporting the completion process.

Table 4.6: Relative improvement of FTM_{NL} over the baseline model (FTM_{base}).

Model	Completion Rate	Recovery Rate	Cosine Similarity	MCA	BLEU
FTM_{base}	-	-	-	-	-
FTM_{NL}	▼ 15.77%*	▼ 24.35%*	▼ 6.72%	▼ 20.04%*	▼ 3.44%*

Downward red triangles (▼) indicate a relative decrease. Values without symbols denote no measurable change relative to FTM_{base}. The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

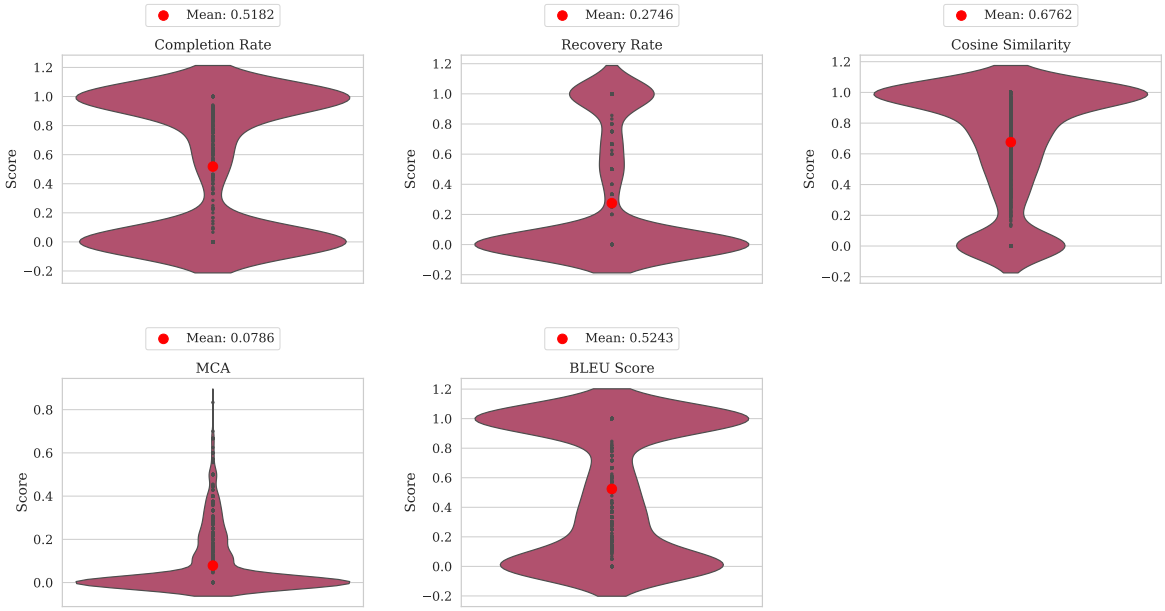


Figure 4.6: Distribution of per-instance values of all metrics for FTM_{NL}.

Relative Performance Improvement. To further quantify the impact of natural language comments, Table 4.6 reports the relative performance of FTM_{NL} with respect to the base model configuration (FTM_{base}). Relative changes are computed per metric, with red downward triangles indicating relative performance degradation.

Across all metrics, FTM_{NL} exhibits a relative decrease compared to the base model performance. Completion Rate shows a statistically significant relative decrease of 15.77%, while Recovery Rate drops by 24.35%, representing the largest relative decrease among all metrics. For MCA, we observe a decreases by 20.04%, and for BLEU by 3.44%, both of which are statistically significant at $\alpha = 0.05$. Furthermore, Cosine Similarity decreases by 6.72%, although this change does not reach statistical significance ($p > \alpha$).

In comparison to FTM_{base}, the FTM_{NL} configuration exhibits a relative decrease of 11.48% in Completion Rate and 16.66% in Recovery Rate. The most notable downfall is observed for MCA, which decreases by 20.00%, highlighting a substantial loss in exact structural reconstruction capability.

In summary, these relative performance trends reinforce the conclusions drawn from the absolute performance analysis. The consistent and statistically significant degradations across metrics show that incorporating NL comments during fine-tuning does not improve, and in fact degrades, the model's performance on the task of UML class diagram completion.

Distributional Analysis. In this paragraph we analyze the distributions of each metric across all both of the model configurations and discuss whether the observations align with the numerical results obtained earlier.

The distributions of the FTM_{base} are illustrated in Figure 4.1 and discussed paragraph 4.2.1, while Figure 4.6 displays the per-instance metric distributions for the model augmented with the NL comments (FTM_{NL}).

For Completion Rate, only 871 out of 1633 instances lie above the mean value of 0.5182, compared to 1015 instances above the baseline mean. Thus,the Completion Rate distribution exhibits more dense lower bulb located near 0.0 than respective distribution for the FTM_{base}, which indicates a higher frequency of incomplete predictions produced by FTM_{NL}. This observation aligns with the statistically significant decrease in Completion Rate observed for FTM_{NL} ($p < \alpha$).

We observe a similar pattern for the Recovery Rate, where only 531 out of 1633 instances exceed the mean value of 0.2746, compared to 630 instances lying above the mean for the base model. The resulting distribution is thus strongly skewed toward lower values, reflecting more frequent failures to reconstruct the original removed elements. This behavior is consistent with the statistically significant decrease in Recovery Rate observed for FTM_{NL} configuration ($p < \alpha$).

For MCA, the distribution of instances is concentrated around 0.0, with only 500 instances lying above the mean value of 0.0786, while the respective distribution for the base model has 566 instances located above the mean. This observation and the absence of a visible upward shift in the distribution align with the results of statistical testing, which indicates a statistically significant reduction in MCA ($p < \alpha$).

In contrast, semantic similarity metrics exhibit more moderate changes. For Cosine Similarity, 1023 out of 1633instances lie above the mean value of 0.6762, mirroring the baseline distribution shape despite a lower mean. This observation is consistent with the absence of a statistically significant difference for the Cosine Similarity metric. However, for BLEU score, 793 instances exceed the mean value of 0.5243, while for the FTM_{base} 807 out of 1633 instances lie above the respective mean (0.5430). Even though the difference is slight, it reflects a statistically significant downward shift compared to the base model configuration. This decrease can be primarily observed by comparing the densities of the lower bulbs of the distributions, where FTM_{NL} model exhibits a more dense one than FTM_{base}.

Table 4.7: Average values of alternatively computed Cosine Similarity and BLEU for all model configurations.

Model	Alternative Cosine Similarity	Alternative BLEU
FTM_{base}	0.2959	0.0235
FTM_{NL}	0.2853*	0.0236

The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

Alternative Computation of Semantic Similarity Metrics. In this paragraph, we report alternative BLEU and Cosine Similarity scores, which are computed over the set of *predicted missing elements only*, excluding elements already present in the partial input diagram. We compare this set *predicted missing elements* to the set of the elements removed from the ground truth and compute the semantic similarity metrics. This formulation aligns semantic similarity evaluation with the completion task and avoids penalizing models for differences in unchanged structural components. The average results obtained under this alternative setup are summarized in Table 4.7.

Under this formulation, the base model (FTM_{base}) achieves an alternative Cosine Similarity of 0.2959 and an alternative BLEU score of 0.0235, while the model augmented with natural language comments (FTM_{NL}) exhibits slightly lower values of 0.2853 and o0.0236, respectively. As reflected in Table 4.7, the difference in alternative BLEU is not statistically significant, whereas the difference in alternative Cosine Similarity for the FTM_{NL} configuration is marked as statistically significant at $\alpha = 0.05$.

We observe that the magnitude of these differences is small. This indicates that when focusing on the *fully* reconstructed class diagrams, NL descriptions do not improve semantic similarity between predicted and ground-truth diagrams.

It is important to note, that we report the alternative semantic similarity results for completeness and do not use them as primary indicators for answering RQ2. Instead, the main analysis relies on a restricted formulation of BLEU and Cosine Similarity that evaluates only the *reconstructed missing elements*, thereby better aligning semantic similarity evaluation with the actual objective of the class diagram completion task. This distinction ensures that the reported results reflect the LLM's ability to correctly infer missing elements rather than surface-level similarity fragments.

Failure Cases. In addition to the observed performance decrease in metrics, we also observe failure cases in the form of malformed predictions that cannot be parsed as valid XMI representations. These malformed outputs include syntactic inconsistencies, incomplete structural definitions, or broken references, which prevent successful parsing and evaluation. As mentioned in RQ1 (paragraph 4.2.1), such predictions receive a score of 0 across all evaluation metrics.

Table 4.8 summarizes the number of successful and failed predictions for both, FTM_{base} and FTM_{NL}. While the base model produces no malformed outputs, FTM_{NL} yields 63 out of 1633 failed predictions, which corresponds to a failure ratio of 3.85%. We run the significance tests and observe that this increase in malformed predictions produced by the augmented model is statistically significant at α-level of 0.05. This observation suggests that the inclusion of NL comments during fine-tuning negatively affects the validity of produced XMI outputs.

Table 4.8: Number of malformed (failed-to-be-parsed) predictions per model configuration.

Model Configuration	Successful Predictions	Failed Predictions	Failure Ratio
FTM_{base}	1633	0	0.00%
FTM_{NL}	1570	63	3.85%*

The "*" symbol indicates a statistically significant result at $\alpha = 0.05$.

4.3 Discussion

In this section, we discuss the results obtained from the controlled experiments conducted in this study. We first analyze and interpret the findings related to RQ1, followed by a discussion of the results obtained for RQ2.

4.3.1 Discussion RQ1: External Knowledge Augmentation

This section discusses the results obtained for RQ1, which investigates the impact of integrating non-modeling data on the performance of fine-tuned large language models in UML class diagram completion. Overall, the findings obtained for RQ1 provide a nuanced view on the role of external knowledge augmentation for UML class diagram completion. While numerical differences across several evaluation metrics can be observed, the statistical analysis shows that these effects are limited, highly selective, and in many cases not statistically significant. Consequently, no definitive conclusions can be drawn regarding performance improvements. Instead, the results suggest that the integration of non-modeling data does not uniformly improve completion performance and under certain conditions may even lead to performance degradation. In the following, we discuss plausible explanations for these observations and relate them to the obtained results.

Impact of External Knowledge. UML class diagram completion is a constrained task that requires precise reconstruction of XMI-encoded structural elements. In comparison to that, several external knowledge sources used for augmentation, especially general-purpose textual data such as Wikipedia entries sampled from [16], are descriptive, loose in structure, and oftentimes do not align with the domain of software engineering. This mismatch between task requirements and the type of the external knowledge is reflected in the statistically significant degradation in Completion and Recovery Rates for the FTM^{5}_{wiki} configuration ($p < 0.05$). Even a modest proportion of general textual knowledge appears to negatively affect the LLM's ability to correctly infer missing elements.

Although increasing the proportion of general textual knowledge to (8%) leads to slightly different numerical outcomes, the overall trend remains consistent and still results in degraded performance across the both of the metrics. This behavior is further supported by the distributional analysis. As shown in Figure 4.2, while a substantial number of instances for both FTM^{5}_{wiki} and FTM^{8}_{wiki} lie above the respective mean Completion Rates, the distributions exhibit an overall shift toward lower mean values compared to the base model. This indicates that not all types of external knowledge are equally beneficial for structured modeling tasks and that descriptive, general-domain text may introduce ambiguity rather than useful guidance in the task of software model completion.

In contrast to general-purpose texts, software engineering–related external knowledge sampled from the Software Engineering dataset [24] (see Section 3.2.2) is observed to have a more favorable alignment with the domain of UML class diagram completion. We observe the statistically significant improvement in Recovery Rate for FTM^{5}_{se} with $p = 0.049$. Although this configuration does not yield statistically significant improvements for other metrics, it remains one of the few models that achieves a relative improvement of6.61% in MCA. Based

on a manual inspection of selected dataset entries, we believe this effect can be attributed to the closer domain alignment of this dataset tot the task of UML class diagram completion. Precisely, both of them often share relevant vocabulary. In addition, some of the entries of the Software Engineering dataset [24] also include information on various types of UML diagrams, and oftentimes include explicit references to software modeling concepts.

A similar but more subtle pattern is observed for code-related external knowledge sampled from the Python code dataset [51]. While code-oriented augmentation at lower proportions (5%) leads to small numerical improvements in some metrics (Recovery Rate and Cosine Similarity), as reported in Table 4.2, these gains do not reach statistical significance, are accompanied by increased failure rates (see Table 4.4) and semantic degradation at higher augmentation levels. In particular, the FTM^{8}_{code} configuration exhibits both the highest failure ratio and statistically significant decreases in semantic similarity metrics (see Table 4.2). This may be because the code exhibits greater syntactic complexity and deeper nesting patterns than the XMI-encoded class diagrams. Even if their overall structural alignment is similar, they originate from distinct task domains. Consequently, augmenting UML data with Python code snippets or code-related content can disrupt the generation of well-formed XMI outputs when such data is overrepresented during fine-tuning.

Finally, the positive impact of software-engineering and code-based augmentation does not persist at higher augmentation ratios. Specifically, we observe that 8% augmentation, the gains observed at lower levels largely vanish, suggesting that enriching an LLM with higher percentage of external knowledge may dilute the structural patterns it learns from the main dataset, which is in our case, UML data. Taken together, these results indicate that the effectiveness of external knowledge augmentation is highly sensitive to both the nature and the proportion of the added data, and that moderate amounts of domain-adjacent information are more likely to support class diagram completion than broad or overly complex external sources, like Wikipedia entries or code-related data.

Task Formulation and Output Level of Detail. Further factor influencing the results is the formulation of the completion task itself. In our work, the model is asked to output a *fully completed* class diagram in a single generation step, *reproducing the given partial diagram* and *inserting the missing elements*. At a first glance such formulation appears intuitive, but it also introduces several challenges. Namely, this leaves room for hallucinations, particularly in situations where only a few class diagram elements are missing or where no elements were removed at all. In addition, we do not apply any explicit stopping condition when prompting a fine-tuned model variant, which enables it to continue generating elements that are structurally plausible but nonetheless incorrect. By structurally plausible but incorrect elements, we refer to elements that could be contextually compatible with the partial diagram but are not the ones that were intentionally removed from it. This, in turn, directly affects strict structural metrics such as MCA.

This helps to explain why MCA values remain consistently low across all configurations and why no statistically significant improvements are observed for this metric, as reported in Table 4.1 and Table 4.2. Even when some missing elements are correctly recovered, the introduction of additional, non-existent hallucinated elements is heavily penalized by the metric, leading to lower scores and degradations for some configurations. This suggests that a formulation in which the model is asked to generate only the missing elements,

rather than the entire diagram, might be better suited for this task. In addition, one could also introduce explicit stopping conditions to further improve the quality of the generated output and avoid hallucinations. Such an approach could reduce hallucinations and allow structural metrics, such as MCA, Completion and Recovery Rates to more directly reflect the quality of inferred content.

Structural Metrics and Metric Interpretation. The difference in behavior of Completion Rate, Recovery Rate and MCA highlights the importance of careful metric interpretation in the context of UML class diagram completion. As we have observed in Table 4.1, Completion Rate tends to achieve higher average values than both Recovery Rate and MCA. This difference is partly explained by the definition of Completion Rate (see Section 3.4.3), which accounts for *all elements* present in the generated diagram, including those that were already provided in the partial input and correctly copied by the model.

In contrast, Recovery Rate and MCA on the *elements that were removed* during data preparation and must be inferred by the model. As a result, these metrics provide a stricter and more explicit assessment of the LLM's inference capabilities. This explains why Recovery Rate and MCA consistently exhibit lower absolute values and distributions that are clustered at lower values, as illustrated in Figure 4.3 and Figure 4.4.

Moreover, the gap between Completion Rate and Recovery Rate indicates that generating a sufficient number of elements does not necessarily imply recovering the *correct* ones. Higher Completion Rate values may therefore reflect not only successful inference, but also the model's general ability to accurately reproduce or copy the provided partial class diagram. MCA further amplifies this distinction by penalizing both missing and hallucinated elements, making it especially sensitive to hallucinations. Although such strictness of both Recovery Rate and MCA results in low scores, it highlights the difficulty of the task and leaves room for further enhancements of LLMs in the domain of software engineering model completion.

Semantic Similarity Metrics and Structural Correctness. The observed behavior of semantic similarity metrics (Cosine Similarity and BLEU) further supports the observations we made for the structural metrics. Across most model configurations, Cosine Similarity remains comparatively high even when structural correctness, as measured by MCA and Recovery Rate, is low (see Table 4.1). This behavior is also reflected by the respective distributions, where Cosine Similarity has a narrow spread centered at relatively high values, while MCA and Recovery Rate exhibit high concentration of instances around lower values (see Figure 4.4 and Figure 4.3). This indicates that even structurally incorrect predictions can remain semantically close to the missing elements at the embedding level. Consequently, Cosine Similarity primarily captures whether the predicted elements are conceptually or semantically related to the removed elements, rather than whether they are structurally correct.

Furthermore, the model configuration augmented with code-related data at higher augmentation level (FTM^{8}_{code}) exhibits the most pronounced degradation in both Cosine Similarity and BLEU by 16.89% and 22.60%, respectively. This effect is visible in the corresponding distributions (Figure 4.5) and supported by statistically significant decreases observed in the Table 4.2. A plausible explanation to such performance decrease is the lexical and syntactic

mismatch between the UML and Python code. While code shares certain structural characteristics with UML, its nesting patterns, token distributions, and syntax differ substantially from XMI-encoded class diagrams. Excessive exposure to such patterns appears to shift the LLM's internal representations away from XMI-specific conventions learned from the main dataset, thereby reducing semantic alignment at both the token and embedding levels.

Finally, When semantic similarity is computed using the alternative formulation (paragraph 4.2.1) that compares the *entire generated class diagram* against the *full ground-truth* representation, both BLEU and Cosine Similarity remain uniformly low across all configurations, as described in Table 4.3. As discussed earlier, the statistical analysis shows that these alternative scores do not yield meaningful performance differences, even in cases where structural metrics indicate partial success. Such behavior can be attributed to the task formulation itself: full-diagram comparisons penalize differences in element ordering, formatting, and XMI serialization, leading to low similarity scores even for structurally correct predictions. Thus, these observations suggest that semantic similarity metrics are sensitive to representation-levels and must therefore be applied and interpreted with caution, particularly when applied to full-diagram comparisons in structured software modeling tasks.

Failure Cases. The analysis of malformed predictions (paragraph 4.2.1) underlines another important dimension of the trade-offs introduced by external knowledge augmentation. While the base model with 0% external knowledge augmentation (FTM_{base}) and configurations augmented with Wikipedia dataset [16] (FTM^{5}_{wiki},FTM^{8}_{wiki}) produce no malformed outputs, software engineering- and code-augmented models (FTM^{5}_{se},FTM^{8}_{se}, FTM^{5}_{code},FTM^{8}_{code}) exhibit statistically significant increases in failure rates, as reported in Table 4.4. The observed effect is the most pronounced for the higher augmentation levels of code-related data. Namely, the FTM^{8}_{code} configuration reaches a failure ratio of 13%, which corresponds to total number of 212 malformed outputs. For the remaining augmented configurations, failure ratios range between $0.50\% - 3.12\%$, yet still remain statistically significant at $\alpha = 0.05$.

In contrast, the configurations augmented with Wikipedia dataset [16] (FTM^{5}_{wiki},FTM^{8}_{wiki}), produce no malformed outputs even at higher augmentation levels. This is likely due to the relatively simple and well-formed linguistic structure of general text, which does not interfere with the formatting requirements of XMI. While such general text may be less useful for improving structural inference, it also appears less likely to destabilize the generation of valid XMI-encoded outputs.

The malformed predictions observed for software engineering- and code-augmented models are likely a consequence of the increased syntactic and structural complexity introduced by these data sources. As discussed earlier, code-related data exposes the model to deeply nested structures, long sequences, and formatting conventions that are different from XMI. As a result, this can interfere with the LLM's ability to consistently generate valid XMI representations, especially when such data is overrepresented during fine-tuning process. Together, these observations align with what we have discussed earlier in this section, indicating that aggressive augmentation can destabilize output generation even when the augmented data is conceptually related to the task domain.

Apart from influencing the output validity, increased failure rates also directly affect the evaluation results. Malformed predictions cannot be parsed and therefore receive value of 0 across all evaluation metrics. As a result, we have lower average performance values and increased variance. Consequently, higher augmentation levels, especially for code-related data, not only introduce increased number of invalid outputs but also negatively impact measured performance. Together, these results reinforce the conclusion that external knowledge augmentation must be carefully balanced with respect to both content and proportion.

Relation to History-Based Approaches. Unlike history-based approaches such as those proposed by Tinnes et al. [48], the model configurations evaluated in our work operate without temporal or contextual grounding. Each class diagram is treated as an independent artifact, providing no evolutionary context to guide structural inference. While the design that we follow increases applicability to real-world scenarios involving isolated models, it also removes an important source of structural guidance. The absence of such context likely contributes to the limited gains observed from external knowledge augmentation alone.

Pilot Study and Implications for Diagram Partitioning. In addition to the main experiments, we conducted a pilot study to explore alternative fine-tuning strategies. We describe the way it was conducted in detail in Appendix, Section A.2. In this pilot study, we trained the model on *entire* XMI-encoded class diagrams directly from the ModelSet dataset (see Section 3.2.2) without partitioning them into smaller chunks. In this setup, full XMI-encoded class diagrams were provided both as input and as the expected output during fine-tuning process. Our intention was to enable end-to-end diagram reconstruction and to assess whether such a formulation could simplify the completion task.

However, the outcomes of this pilot study were consistently unsatisfactory. Models fine-tuned on full diagrams frequently produced malformed outputs that could not be parsed as valid XMI. In many cases, the generated output consisted of copied instructions, empty responses, or incomplete fragments lacking meaningful structural content. Only in rare cases did the model generate a valid class diagram completion. Even then, the number of such successful completions was limited to very short class diagrams that did not exceed the maximum sequence length of the underlying LLM. As a result, most evaluation metrics in this pilot study had either values of 0 or near-zero values across all test instances.

These observations suggest that the limited maximum sequence length of the underlying LLM makes it impractical to process full-scale class diagrams, which often exceed this limit even before completion is attempted. Moreover, requiring the model to reconstruct an entire diagram in a single generation step substantially increases the burden placed on the model, increasing the risk of producing malformed outputs. Put together, full-diagram fine-tuning without partitioning them into smaller chunks is ineffective for this specific task given the same underlying LLM is utilized.

The insights gained from this pilot study directly motivated the partitioning strategy integrated in the data preparation pipeline, described in Section 3.2.3. By decomposing class diagrams into smaller, structurally valid XMI chunks, we ensure that both inputs and outputs remain within supported sequence length boundaries while preserving the valid structure of XMI-encoded class diagrams.

Summary Taken together, the results and discussion of RQ1 demonstrate that integrating non-modeling data into the fine-tuning process has a limited and highly context-dependent impact on the performance of fine-tuned LLM for task of UML class diagram completion. While numerical differences across several metrics and augmented model configurations can be observed, the respective statistical analysis reveals that these differences are often not statistically significant. In some cases, they even lead to performance degradation. This suggests that integrating non-modeling data during fine-tuning does not uniformly improve structural completion performance.

General-purpose textual knowledge, such as Wikipedia entries, appears largely misaligned with the task requirements and can negatively affect both Completion and Recovery Rates. On the other hand, domain-adjacent knowledge (e.g., software engineering texts) can support partial recovery of missing elements at lower augmentation levels (5%). However, these benefits do apply when the proportion is increased to 8%. Code-related augmentation exhibits a more delicate behavior, leading to decreased semantic similarity, increased failure rates, and reduced output validity at higher augmentation levels (8%).

Across all augmented configurations, structural metrics such as MCA and Recovery Rate reveal the difficulty of the task, highlighting the problem of hallucinations Semantic similarity metrics can be useful as complementary indicators, but are not sufficient enough for assessing structural correctness in isolation. Moreover, they highly sensitive to the way the task is formulated and what representation-level is used.

> Integrating non-modeling data has a limited and highly context-dependent impact on the performance of fine-tuned LLMs for UML class diagram completion. While moderate amounts of domain-adjacent software engineering data can result in small, selective improvements, general-purpose text, code-related data and higher augmentation levels often increase failure rates, and degrade the performance of the model.

4.3.2 Discussion RQ2: Integration of Natural Language Comments

In this section we discuss the results obtained for RQ2, which investigates the impact of including natural language comments on the performance of fine-tuned large language models in UML class diagram completion. In contrast to RQ1, where external knowledge was provided as separate auxiliary training data, RQ2 embeds natural language descriptions directly within the model input alongside the partial class diagrams.

The findings obtained for RQ2 indicate a predominantly negative or neutral impact of integrating NL comments across all evaluated metrics. In contrast to RQ1, several of the observed effects are statistically significant, which allows for more definitive conclusions. In the following, we discuss these results in detail and provide plausible explanations for the observed performance trends.

Impact of Natural Language Comments on Structural Completion. The absolute and relative performance results show that the model augmented with NL comments (FTM_{NL}) underperforms the baseline model across all structural metrics, including Completion

Rate, Recovery Rate, and MCA. As reported in Table 4.6, these degradations are statistically significant at $\alpha = 0.05$. Moreover, the distributional analysis further support this observation, as the distributions of FTM_{NL} for respective metrics (see Figure 4.6) shift toward lower mean values and are concentrated around lower values, having less dense upper bulbs compared to the base model (FTM_{base}).

We can attribute this behavior to the integration of NL comments into the XMI-encoded class diagrams, as this is the only factor distinguishing the two model configurations under comparison. The NL comments used in this experiment provide a high-level summaries that capture the essence of class diagram rather than explicitly specifying all concrete structural elements such as class names, attributes, or relationships. While such descriptions are meaningful for human interpretation, they provide limited guidance for an LLM tasked with reconstructing missing elements. As a result, the additional NL information does not support the inference of missing class diagram elements and instead introduces ambiguity that leads to decreased performance.

Semantic Similarity Metrics. The behavior of semantic similarity metrics in RQ2 closely mirrors the patterns observed in RQ1. Cosine Similarity remains relatively high despite significantly low values of MCA, Completion and Recovery Rates, indicating that predictions remain conceptually related to the missing elements even when structural reconstruction fails. In contrast, BLEU results in a significant decrease of 3.44%, although the absolute magnitude of this chnage is relatively low.

These observations suggest that integrating NL comments negatively affects not only the structural correctness of the generated class diagrams, but also their semantic alignment with the ground truth. This can be due to the nature of the injected comments. The comments provide abstract, high-level descriptions that are not explicitly grounded in concrete structural elements, such as specific classes or relationships. As a result, they may introduce additional linguistic variability or ambiguity, which affects both the LLM's ability to reconstruct precise elements and its ability to maintain semantic alignment with the missing elements.

The alternative computation of BLEU and Cosine Similarity, based on *full-diagram* comparisons, produces low values across both model configurations. In this setting, we observe a statistically significant difference in Cosine Similarity. This effect is likely driven by representation-level differences rather than inference quality. When *entire* diagrams are compared to each other, variations in element ordering, formatting, and XMI serialization are heavily penalized, even when the inferred elements are correct. Consequently, these alternative semantic similarity scores reflect the sensitivity of similarity metrics to structural and representational variations, rather than meaningful differences in diagram completion quality.

Failure Cases. For the FTM_{NL} configuration, we observed a total of 63 out of 1633 malformed predictions, which corresponds to failure ratio of 3.85%. While this number remains relatively small compared to the overall test set size, it nevertheless remains statistically significant, as reported in Table 4.8.

This behavior can be attributed to the integration of NL comments into the XMI-encoded class diagrams. Although the comments are syntactically well-formed, they introduce

additional tokens and linguistic structures that are not differ from syntax and formatting requirements of XMI strings. In some cases, this added linguistic content appears to interfere with the LLM's ability to maintain valid output, leading to incomplete XMI, broken references, or invalid XMI syntax. In contrast to code-related augmentation, which mainly increases syntactic complexity, combining NL comments with XMI structures can introduce additional ambiguities or divert the LLM's attention away from learning the XMI class diagram encoding toward the NL comments instead.

Importantly, malformed predictions cannot be parsed and therefore receive a score of 0 across all evaluation metrics. As a result, even a relatively small number of such failures contributes to lower average metric values and increased variance in the reported results. This effect also underlines the observed performance degradation for the FTM_{NL} configuration and reinforces the fact that integrating NL comments can negatively impact both output validity and overall model performance.

Task Formulation and Nature of the Comments. Additionally, the negative impact of NL comments should also be interpreted in light of the task formulation. In this setting, the LLM is required to reconstruct a *full* class diagram while simultaneously processing textual descriptions that are only loosely connected to concrete structural elements. Since there is no explicit alignment between the integrated comments and specific diagram components, for example a direct reference, annotation or constraint, the model is left to resolve this mapping implicitly. This implicit reasoning may introduce further ambiguities, which interfere with the LLM's ability to focus on learning and reproducing the XMI syntax and structural patterns that are necessary for accurate UML completion. As a result, the additional NL information is more likely to distract the model from the task rather than provide useful guidance, contributing to the observed performance degradation.

In addition to the task formulation, the *nature* of the integrated NL comments likely plays a critical role in the observed performance degradation. The comments used in our experiments are intentionally high-level and abstract, such that the model does not get explicit information about the missing elements, but rather tries to infer them from general summary describing functionality of the class diagram. While such descriptions are well-suited for human comprehension, they may be insufficient for guiding a model toward precise inference. In particular, abstract comments may encourage the model to reason at a conceptual level, potentially prioritizing semantic correctness over adherence to the formal UML structure required by the task.

Summary. Put together, the results of RQ2 show that integrating natural language comments into XMI-encoded class diagrams does not improve, and in many cases degrades, the performance of fine-tuned LLMs for UML class diagram completion. Across all evaluated metrics, the augmented model consistently underperforms the base model, with several of the observed differences being statistically significant. These findings suggest that the inclusion of NL comments does not provide useful structural guidance.

While NL descriptions are beneficial for human understanding, they do not align well with the nature of the task, which requires precise structural inference. Furthermore, the additional linguistic content increases input complexity, potentially distracting the model from learning the XMI representation.

The analysis of semantic similarity metrics and failure cases further reinforces these findings. Although predictions remain conceptually related to the missing elements, structural correctness is consistently reduced, and malformed outputs contribute to lower average metric values.

While such comments may enhance human interpretability, they appear to offer limited benefit for fine-tuned LLMs tasked with recovering formal and syntactically constrained model structures. Consequently, the effectiveness of natural language augmentation in this context appears to be highly sensitive to the granularity, informativeness, and alignment of the comments with the underlying structural representation.

> The inclusion of natural language comments has a negative or neutral impact on the performance of fine-tuned LLMs for UML class diagram completion.

4.4 Threats to Validity

This section discusses potential threats to validity of our experimental setup and results. We differentiate between internal and external validity threats and elaborate on the measures we took to mitigate them. We do so by first discussing factors that may affect whether the observed performance differences can be attributed to the manipulated independent variables rather than to unintended experimental confounders. Then, we outline limitations regarding the generalizability of the obtained results beyond the experimental setting.

4.4.1 Threats to Internal Validity

A first threat concerns the alignment between the evaluation metrics and the underlying task of UML class diagram completion. While semantic similarity metrics such as BLEU and Cosine Similarity provide a general indication of textual overlap, they do not explicitly capture the structural correctness of the generated predictions. To mitigate this, the results analysis places primary emphasis on MCA, Completion Rate, and Recovery Rate, which directly evaluate the correctness of reconstructed class diagram elements. Nevertheless, these metrics are sensitive to hallucinated elements, which may lead to low absolute scores even when predictions appear accurate at a semantical level.

Next internal threat arises from malformed predictions that cannot be reconstructed and parsed as valid XMI representations. Such predictions receive a score of 0 across all evaluation metrics, which directly affects average performance values and variance. To mitigate this threat, all generated predictions are processed using the same fixed parsing and evaluation pipeline (see Section 3.2.1), and malformed predictions are explicitly identified, reported, and analyzed separately for RQ1 and RQ2 as *failure* cases in paragraph 4.2.1 and paragraph 4.2.2, respectively.

Finally, effects introduced during the LLM fine-tuning process, such as random initialization and data shuffling, may influence individual training runs. We mitigate this threat by fixing random seeds ($seed = 42$), maintaining identical fine-tuning and inference configu-

rations as it was mentioned earlier in Section 3.2, and ensuring that no model continues training from a previous condition.

4.4.2 Threats to External Validity

A primary external validity threat concerns the representativeness of the *main dataset*. All experiments are conducted on UML class diagrams derived from the ModelSet dataset introduced in Section 3.2.2. While this dataset is diverse, it may not fully capture the complexity, scale, or domain-specific conventions of real-world modeling practices. Moreover, this work focuses exclusively on the UML class diagrams, which limits the applicability of the results to the other types of the UML diagrams, such as sequence, activity or state diagrams. Consequently, the observed effects may not directly generalize to all real-world software modeling scenarios.

Another external validity threat arises from the choice of model architecture and training setup. The obtained results are specific to the chosen model architecture and training setup. All experiments employ a single base LLM, LLaMA-3.1 with 8 billion parameters, fine-tuned using QLoRA. Different model sizes, architectures, or fine-tuning strategies may exhibit different sensitivities to external knowledge or natural language augmentation.

The task formulation itself introduces an additional threat to external validity. In our work, model configurations are required to reconstruct an *entire* UML class diagram in a single generation step, rather than producing only the missing elements. While such task formulation enables end-to-end comparison with the ground-truth and evaluation, it may not reflect alternative completion settings where models operate in a more constrained or interactive manner. Different task formulations, for example, focusing only on element-wise completion or incremental reconstruction of missing elements, may result in different performance, failure rates and sensitivity to data augmentation strategies.

Closely related to task formulation is the partitioning strategy applied to the input data, described in Section 3.2.3. To address sequence length constraints, we decompose class diagrams into smaller, structurally valid XMI fragments. Although we made sure that the XMI fragments are valid and such approach is necessary to enable fine-tuning and inference with current LLMs, it may alter the global structural context of the diagrams. In real world settings, LLMs may be required to reason over *complete* diagrams or the ones that are even larger than the class diagrams presented in ModelSet [32]. Therefore, the observed results may not directly generalize to scenarios where no partitioning is applied or where alternative segmentation strategies, for example, batch processing, are used.

The scope of the augmentation strategies further constrains generalizability. For RQ1, we consider a limited set of external knowledge sources and augmentation ratios are considered. Other types of domain-adjacent data or alternative proportions may lead to different outcomes. Similarly, for RQ2, the injected NL comments consist of high-level, abstract summaries of class diagram intent rather than explicit, element-level specifications. While this design choice reflects realistic documentation practices, it does not capture the full range of possible natural language augmentations. More structured annotations, explicit constraints, or element-specific descriptions could potentially influence model behavior differently.

Finally, the generation of natural language comments relies on an auxiliary LLM, as mentioned in Section 3.2.4. Any biases, inaccuracies, or stylistic characteristics of this auxiliary model are propagated into the training data through the injected comments. As a result, the observed effects in RQ2 cannot be attributed only to NL augmentation in general, but rather to the nature of the generated comments, their abstraction and quality level. Alternative comment generation strategies or manually curated annotations may therefore yield different results.

5 Limitations and Future Work

This chapter discusses the main limitations of our study and outlines the directions for the future research.

5.1 Limitations

A primary limitation of this study concerns hardware constraints. All fine-tuning experiments were conducted under restricted computational resources, which imposed limitations on the choice of model size, batch size, and the number of repeated training runs. In particular, the use of a single base model, LLaMA-3.1 with 8 billion parameters, fine-tuned via QLoRA, was driven by GPU memory constraints. While PEFT allows for experimentation with LLMs under limited hardware, it restricts the exploration of larger architectures, alternative model families, or full fine-tuning strategies that may exhibit different learning dynamics and completion capabilities.

Consequently, the number of fine-tuning runs per model configuration was limited. Although random seeds were fixed to ensure reproducibility, this restricts the ability to quantify variance across multiple independent training runs. Thus, the absolute and relative performance values reported in Section 4.2 should be interpreted as indicative rather than exhaustive, even though relative comparisons between configurations remain valid due to the controlled experimental setup.

Another important limitation arises from the way the task is formulated. The models are trained and evaluated to reconstruct an *entire* UML class diagram in a single generation step. Such task formulation increases the difficulty of the task and introduces room for hallucinations, especially in cases where only a small number of elements are missing or no elements were removed. As discussed in the results (Section 4.2), model oftentimes generate additional classes, attributes or relationships even when the input diagram is missing no elements. This behavior negatively affects structural metrics like MCA and Recovery Rates, which account for hallucinated elements ad thus result in low absolute values. Although this limitation affects all configurations uniformly and therefore does not bias relative comparisons, it constraints the achievable absolute performance.

The inference strategy further contributes to this limitation. During the evaluation process, each of the fine-tuned configurations are allowed to generate up to a fixed maximum of 4096 new tokens, and no explicit stopping condition is enforced beyond the model's internal end-of-sequence prediction. Similarly, this increases the likelihood of unnecessary generation and hallucinations.

Finally, the handling of large XMI-encoded class diagrams introduces additional constraints. We partitioned long diagrams into smaller fragments to ensure compatibility with the LLM's maximum context length. Although this enables training and inference on large models and enlarges the size of the initial dataset, such partitioning may disrupt global structural context and introduce boundary effects that impact the quality of the completions.

5.2 Future Work

From the limitations we identified above and the findings discussed in Section 4.3, there are several directions for future work.

First, making the inference process more controlled is an important are for future research. Introducing explicit stopping criteria, such as structural completeness checks or schema-aware constraints, could help to avoid the over-generation behavior in cases where no elements are missing. Similarly, dynamically adapting the maximum number of generated tokens based on the size or completeness of the input diagram may help prevent unnecessary hallucinations and improve structural accuracy of the generated output.

Second, future work could explore alternative text generation settings during inference by varying further parameters such as temperature and nucleus sampling (top_p). These parameters directly influence output determinism and randomness. Thus, systematically studying their effects may help balance the precision and diversity of generated output in the task of class diagram completion.

Moreover, alternative strategies for handling long XMI-encoded class diagrams can be explored. Instead of fixed partitioning, batching or window-based approaches could be applied to dynamically segment long diagrams while preserving more global structure and context. Such strategies may reduce information loss introduced by rigid partition boundaries and ensure that complete diagrams consistently fit within the LLM's context window.

Another promising direction concerns reformulating the learning objective. Rather than training the model to reconstruct the *entire* class diagram, future approaches could restrict the output to predicting only the *removed or missing* structural elements. Under this formulation, the model would be explicitly optimized to infer classes, attributes, and relationships that were intentionally removed during data preparation process. A corresponding inference setup could prompt the model to generate only the missing elements instead of a full diagram reconstruction. This targeted prediction strategy may reduce the hallucination rate, improve structural accuracy, and yield higher MCA scores by aligning the generation task more closely with the evaluation metrics and lead to higher structural accuracy.

Furthermore, introducing additional evaluation metrics that explicitly capture model hallucinations is another promising direction for a future work. While metrics such as MCA or Recovery Rate indirectly penalize hallucinations, they do not explicitly quantify the extent to which hallucinated elements are introduced. Defining a dedicated Hallucination Rate, which measures the proportion of generated classes, attributes, or relationships that are not present in the ground truth, could provide more insight into model behavior.

Closely related to this is the extension of the evaluation strategy itself. Rather than reporting aggregate performance scores alone, future work could analyze performance

at the level of individual UML element types. For example, separately evaluating classes, attributes, operations, associations, and inheritance relationships. Such evaluation design would make it possible to identify whether fine-tuned models systematically struggle with a specific type of structural components and whether certain augmentation strategies disproportionately benefit or harm particular element types. Such an analysis could guide more targeted model improvements.

Another important direction concerns the nature and integration of natural language comments. Future work could explore alternative forms of NL comments that vary in structure, specificity, and their placement. For example, comments could be integrated alongside the UML model rather than embedded directly within its respective XMI representation, or designed to explicitly reference concrete diagram elements instead of providing abstract high-level summaries. Comparing inline comments, external descriptions, and element-level annotations could help clarify under which conditions natural language information can support the inference process.

In addition, further experimentation with augmentation ratios and data sources represents a natural extension of this thesis. Rather than examining fixed proportions, future studies could investigate gradual augmentation strategies, where external knowledge is introduced incrementally during fine-tuning. Exploring additional domain-adjacent datasets or combining multiple augmentation sources may also lead to valuable results.

While this work focuses on XMI-encoded class diagrams, future research could explore JSON-based encoding, which may offer a more intuitive and less verbose structure for LLMs. Mixing different serialization formats or jointly training on both XMI and JSON UML diagram representations could potentially improve robustness and generalization by exposing the model to multiple structural encodings of the same underlying information.

Finally, extending the approach to further UML diagram types, such as sequence or activity diagrams, and evaluating performance in interactive scenarios would improve external validity and practical relevance of this approach.

6 Related Work

Recent research on applying LLMs to structured prediction tasks highlights their potential in software modeling. Studies utilizing LLMs for code-focused tasks demonstrate that these models can effectively learn and reproduce hierarchical structures and semantic dependencies through domain-specific fine-tuning [43], general-purpose pretraining [30], and bimodal code–text pretraining [14]. These studies show that LLMs are capable of handling structurally rich data, identifying long-range dependencies, and generating coherent structural outputs. Since UML class diagrams exhibit similar properties such as structural hierarchy, techniques introduced for code understanding and completion can be effectively transferred to UML model completion tasks due to the shared need for structural reasoning and context-aware reconstruction.

Further research explores techniques for augmenting LLMs with structured external knowledge and NL, showing that such augmentation can substantially improve generalization, semantic accuracy, and structured prediction capabilities. Prior work demonstrates the benefits of incorporating code–text bimodal pretraining [14, 53], external structured knowledge sources [57], and NL annotations that provide missing context for comprehensive code understanding [47]. Furthermore, additional studies confirm that LLMs can support software modeling activities when provided with suitable contextual information [2, 6, 15, 48]. The most relevant of these approaches are discussed in more detail in the following sections.

6.1 LLMs for Structured Code Understanding and Generation

In this section, we explore some of the recent advancements in code-related applications of LLMs. While these models are designed for the code tasks, their architecture and training methods offer insights for UML diagram completion. Similarly to code, UML diagrams encode structural, hierarchal information, and completing partial diagrams has similar challenges in contextual inference and semantic accuracy. Such structural similarity suggests that methods developed for code completion, instruction tuning, and prompt engineering may offer useful insights for software model completion tasks as well.

Rozière et al. [43] introduced Code LLaMA, a family of LLMs fine-tuned on code using infilling, instruction fine-tuning, and a code-heavy dataset for cascade training. The results of their work demonstrate that domain-specialized fine-tuning (e. g.,Python code-specific dataset) can significantly improve structured code generation, while instruction-tuned models align better with human instructions. These findings highlight the value of task-

relevant data and natural language guidance for structured prediction tasks. Moreover, the infilling objective conceptually aligns with UML model completion, where the model tries to reconstruct missing elements given partial input. Complementing this approach, Li et al. [30] introduced StarCoder and StarCoderBase, open-access Code LLMs trained on a multilingual source code from over 80 programming languages. StarCoder demonstrates robust generalization capabilities without task-specific fine-tuning. This supports the hypothesis that training LLMs on structurally rich but also diverse datasets can lead to improved performance on structured generation tasks.

CodeBERT is a bimodal pretrained model for NL and Programming Language (PL) introduced by Feng et al. [14]. This model is trained with a combination of code and NL documentation, demonstrating strong generalization across NL-PL tasks like natural language code search and code documentation generation and achieving state-of-the-art performance after fine-tuning[14]. Such joint pretraining showcases that combining structural and textual data improves model understanding and generalization abilities, which in the scope of this thesis is a key premise behind the integration of NL comments in partial UML diagrams.

Wang et al. [52] examined prompt tuning as an alternative to fine-tuning for code intelligence tasks, utilizing two state-of-the-art models for source code, CodeBERT [14] and CodeT5 [54]. Their results demonstrate that prompt tuning can perform fine-tuning under full data, data scarcity, and cross-domain settings, and it is a promising alternative to fine-tuning. Similarly, Shin et al. [46] have compared prompt engineering and fine-tuning, together with conducting a user study. The authors performed a quantitative and qualitative study utilizing GPT-4 and compared its effectiveness with 17 fine-tuned LLMs in the context of code-related tasks, like code summarization, generation, and translation. Their results show mixed outcomes, GPT-4 with prompt engineering does not consistently outperform the fine-tuned models in all tasks. However, when using task-specific prompting, including conversational prompts and human feedback, GPT-4 shows a notable performance in code summarization and holds potential for automated code-related tasks [46]. While the examined alternatives [46, 52] have shown impressive results in different NL-based tasks, their effectiveness in highly structured domains like UML modeling remains underexplored. Given the fixed semantics of UML class diagrams, our work prioritizes fine-tuning to tailor the LLM to better capture the hierarchical structure and syntax of modeling artifacts.

LLaMA-Reviewer, an innovative framework leveraging LLMs for automating the software code review process was presented by Lu et al. [33]. This work demonstrates that by employing PEFT methods, even the model with a small number of parameters, in this case, LLaMA (6.7B parameters), can achieve performance comparable to existing code-review-foscused-models, while considerably reducing computational resources. The framework addresses key aspects of code review automation, including review necessity prediction, comment generation, and code refinement, evaluated across multiple public datasets. The study highlights the importance of input alignment with pretraining data and initial instruction tuning to improve performance. Moreover, it was concluded that among PEFT methods, LoRA is more suitable for automation code review tasks [33]. Inspired by this, we incorporate LoRA in the fine-tuning process to balance the model's performance with its efficiency.

Put together, these studies establish a foundation for using LLMs in the generation tasks of structural data. It has been demonstrated that LLMs are capable of learning structural patterns through domain-specific fine-tuning [43], general purpose training [30] and bi-modal pretraining [14]. Building on the insights of the studies, we adapt used approaches to the domain of software models, assessing whether similar gains can be achieved by incorporating external knowledge and embedding natural language comments.

To illustrate the historical development of these approaches,Figure 6.1 presents a timeline of the key works on code-focused LLMs that form the foundation for our analysis.

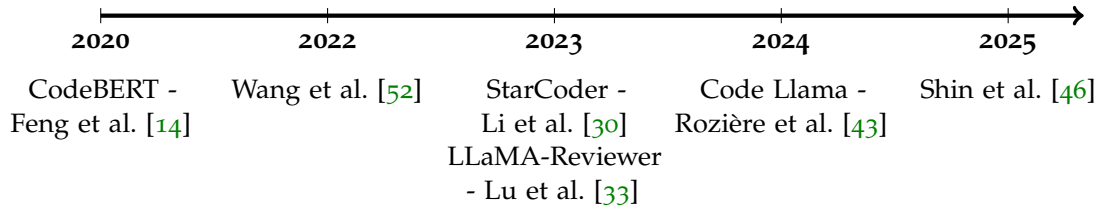


Figure 6.1: Timeline of key works related to LLMs for structured code understanding and generation.

6.2 Augmenting LLMs with External Knowledge and NL Comments

Augmenting LLMs with structured or domain-specific external knowledge and NL comments can improve their performance on complex structured tasks, like code or model completion, and their ability to generalize. Out of various existing augmentation approaches, we explore the ones that are the most relevant to our work.

Xu et al. [57] enhanced NL-to-code generation models by integrating external knowledge from sources like API documentation and Stack Overflow. Their strategy was to pre-train the model on the NL-code pairs from external sources and further fine-tune it on a manually curated corpus. This work demonstrates that integrating external knowledge, no matter if it is noisy or not necessarily task-specific, can significantly boost the model's performance and its contextual understanding. Wang et al. [53] introduced CodeT5+, a family of encoder-decoder LLMs designed for code generation and understanding. These models achieve enhanced adaptability, robustness, and impressive performance across various code- and non-code-related tasks due to their flexible architecture, initialization from strong base models, instruction tuning, and integration of diverse pretraining objectives [53]. Methods such as contrastive learning, span denoising, and text-code matching expand the model's comprehension of code and textual data, thereby improving generalization and bridging the gap between pretraining and downstream fine-tuning. Moreover, CodeT5+ utilizes both unimodal and bimodal multilingual data that allows for boosting its versatility and robustness through exposure to diverse syntax, semantics, and contexts. Similarly, StarCoder [30] and CodeBERT [14] highlight the benefits of incorporating diverse, large-scale external training data and joint pre-training on natural language and code. Together, these models demonstrate that integrating structured data with textual information can significantly

improve semantic comprehension and output quality, reinforcing the hypothesis underlying our approach.

Further studies explored the impact of NL comments on structured tasks. In the development of CodeBERT, Feng et al. [14] highlighted that pretraining models on data consisting of NL-code pairs lead to improved performance on tasks like documentation generation. The comment integration enabled the model to better understand and capture the semantics. Ferrari et al. [15] further investigated the role of NL in software modeling by exploring the abilities of Chat-GPT to generate a specific type of model, UML sequence diagrams, from the NL requirements. This study evaluates LLM-generated models for completeness and correctness, focusing on challenges caused by textual ambiguities and inconsistencies. The results show that while the model highly depends on the granularity, clarity, and specificity of NL requirements, it is still capable of following modeling standards and producing understandable outputs. Recently, Song et al. [47] proposed a novel data augmentation approach that generates comments for existing code and integrates them into code dataset by utilizing LLMs. Their findings showcase that increasing comment density during pre-training leads to significant improvements in the model's performance on structured downstream tasks. Moreover, the models trained on augmented, comment-rich data showed greater performance than those trained without such augmentation, highlighting the fact that the proposed approach improves the models' understanding and generation capabilities by encouraging better code-NL alignment. The findings that these studies share directly align with our hypothesis that embedding clear and structured natural language annotations within UML training data during the fine-tuning process could similarly improve the model's semantic understanding, output quality, and predictive performance.

Collectively, these studies [14, 15, 30, 47, 53, 57] confirm that augmenting LLMs with external knowledge and structured natural language annotations can significantly boost model performance on structured prediction tasks. Figure 6.2 demonstrates the timeline of these approaches. In our research, we uniquely integrate them in the context of UML class diagram completion, aiming to leverage external textual and code resources combined with natural language comments to achieve greater semantic accuracy, robust inference capabilities, and cross-diagram generalization.

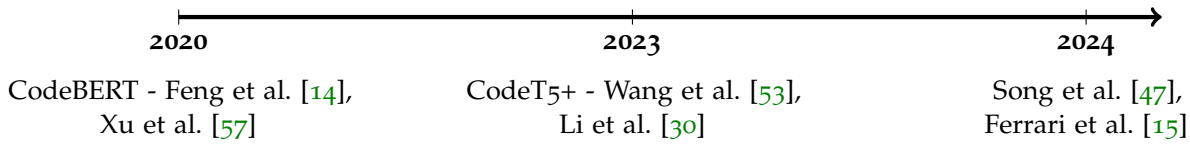


Figure 6.2: Timeline of key works related to augmenting LLMs.

6.3 LLMs in Software Modeling and UML Completion

The integration of LLMs in software modeling has gained attention for automating tasks such as software model completion [6, 48], UML diagram generation [15] and their translation into executable code [2]. To contextualize the evolution of these contributions, Figure 6.3

provides a timeline of the key works on LLM-based software modeling and UML-related tasks. These tasks oftentimes require domain-specific knowledge and human input. However, LLMs have demonstrated significant potential to automate model completion, offer design suggestions, and generate representations of software models that are accurate enough.

Chaaben et al. [6] explored the use few-shot prompt learning approach to improve and automate software modeling activities utilizing GPT-3. Their method fully relies on the LLM's general reasoning capabilities, and, notably, avoids the need for domain-specific fine-tuning or large datasets. This is appealing because collecting structured modeling data and fine-tuning LLMs are resource-intensive and often infeasible in practice [22, 48]. However, despite its efficiency, the robustness and applicability of few-shot prompting for UML-related tasks remains uncertain. Such approaches provide limited control over the model's internal reasoning processes, heavily rely on the precisely tailored prompt and contextual examples, leading to non-deterministic and difficult-to-reproduce outputs. This is problematic for software modeling, where diagram elements must follow strict structural rules and remain globally consistent across classes, relationships, and types. Because few-shot prompting lacks explicit structural constraints and domain adaptation, even minor prompt variations can produce inconsistent or structurally-inaccurate UML diagrams making this approach unsuitable for real-world modeling scenarios that require stable, precise, and reproducible results.

Tinnes et al. [48] introduced and alterative approach to software model completion based on RAMC and compared it with fine-tuning. Their method leverages model change graphs that reflect actual software evolution scenarios to guide the completion process, and their evaluation on simulated, open-source, and industrial datasets yielded substantial improvements in semantic accuracy. Nevertheless, the applicability of RAMC is fundamentally constrained by its reliance on detailed model evolution histories. In many practical settings, especially when dealing with independently developed, static UML diagrams, such historical information is unavailable, making the approach unsuitable outside controlled experimental conditions. This limitation leaves a significant gap for scenarios where evolving model histories are not available. This gap is specifically addressed in our research.

In another direction, Bates et al. [2] focused on automating the conversion of UML diagrams into executable UML code by leveraging Multi-modal Large Language Models (MM-LLMs). The study introduces a new technique that integrates the visual representation of UML diagrams and textual data through MM-LLMs, optimizing the model's performance for UML code generation tasks by using standard fine-tuning and LoRA methods. For training and testing purposes, authors created synthetic data consisting of UML activity and sequence diagrams and assessed the accuracy of the model-generated code using such commonly-known metrics as BLEU [38]. Although their approach demonstrates notable results in generating accurate and executable code, it primarily focuses on translating UML diagrams into executable code rather than enhancing diagram completeness or investigating cross-diagram generalization.

Chen et al. [8] and Ferrari et al. [15] further contribute to this field by exploring applications of LLMs to automate the generation of software models from NL descriptions. As previously discussed, [15] assessed Chat-GPT's capability to produce UML sequence diagrams, establishing that while outputs were generally understandable and followed the UML standards, their completeness and correctness were significantly affected by ambiguous

natural language requirements. Correspondingly, Chen et al. [8] highlighted the potential of generating initial software models from natural language descriptions and the understanding capabilities of GPT models. However, the primary focus of their study was initial diagram generation rather than addressing fully automated model completion, evolution, or cross-diagram generalization.

Altogether, these works demonstrate significant advances in employing LLMs for various UML modeling tasks, but there are still gaps remaining. Few studies explicitly address the challenge of transferring knowledge learned from one UML diagram type to another, nor do they extensively investigate the effectiveness of incorporating external general-purpose knowledge and structured natural language annotations during fine-tuning/training in independently developed, static modeling scenarios. In our research, we directly address the existing limitations by systematically exploring cross-diagram generalization, the integration of external and contextual knowledge, and structured natural language annotations within partial UML class diagrams.

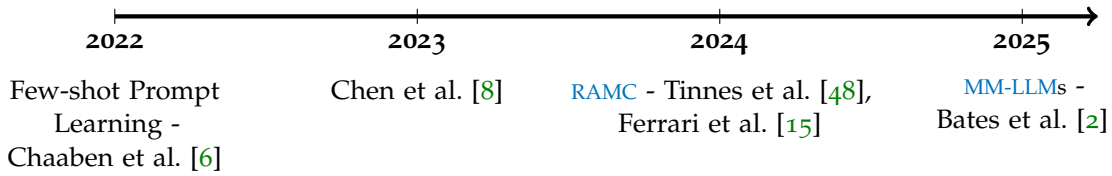


Figure 6.3: Timeline of key works related to utilizing LLMs in the domain of software modeling.

7 Concluding Remarks

7.1 Conclusion

This thesis investigated the impact of fine-tuning large language models for the task of UML class diagram completion, with a particular focus on the role of external knowledge augmentation and the integration of natural language comments. Motivated by the limitations of prompt- and retrieval-based approaches [6, 48], this work explored whether fine-tuning on independent, standalone software models can improve structural understanding and reconstruction quality without relying on historical model sequences or tailored prompts.

To address the first research question, we examined the impact of integrating non-modeling data, including natural language and source code samples, into the fine-tuning process. The obtained results demonstrate that external knowledge augmentation has a limited and selective effect on completion performance. While moderate amounts of domain-adjacent software engineering data can result in small, selective improvements, general-purpose text, code-related data and higher augmentation levels often increase failure rates, and degrade the performance of the model.

The second research question assessed whether integrating NL comments directly into XMI-encoded class diagrams improves model performance. Under the investigated experimental conditions, the results show that high-level NL comments summarizing general diagram intent do not improve completion performance and instead lead to statistically significant degradations. The findings suggest that inclusion of natural language comments has a negative or neutral impact on the performance of fine-tuned LLMs for UML class diagram completion.

Overall, this work contributes empirical evidence on the strengths and limitations of fine-tuned LLMs for software model completion tasks. By systematically evaluating different data augmentation strategies under controlled conditions and emphasizing structural evaluation metrics, the thesis advances understanding of how LLMs can be adapted in the domain of software modeling and provides directions for further research.

7.2 Code and Reproducibility

To ensure transparency and reproducibility of the presented results, all code developed for this thesis is made available. This includes the data preprocessing pipeline, XMI parsing and partitioning scripts, fine-tuning configurations, inference and evaluation scripts used to compute all reported metrics.

The repository can be found under: https://github.com/ykamila/fine-tuning-thesis.git.

Appendix

This appendix provides supplementary material that supports the main body of the thesis. It includes background information on UML diagram types as well as the detailed description of the pilot study.

A.1 Types of UML diagrams.

Table A.1 outlines all 14 diagram types, as well as their descriptions and key roles.

A.2 Pilot Study

This section presents the pilot study along with the insights derived from it and discusses its implications for the data preparation pipeline, in particular for data partitioning strategy introduced in Section 3.2.3, as well as for the fine-tuning setup.

In addition to the main experiments, we conducted a preliminary pilot study to explore alternative fine-tuning strategies. Specifically, we trained the LLM directly on *entire* UML class diagrams without partitioning them into smaller chunks. In this setup, complete XMI-encoded class diagrams were provided both as input and as the expected output during fine-tuning process. The main intention behind this fine-tuning strategy was to enable end-to-end diagram reconstruction and explore whether such strategy brings valuable insights.

However, the results of this pilot study were consistently poor. Model configurations fine-tuned on full class diagrams mostly produced heavily malformed outputs that could not be parsed as valid XMI representations. In many cases, the generated output consisted of copied instructions or random symbols, empty responses, or incomplete fragments without meaningful structural content. Only in rare cases the model produced a valid completion, and these cases were limited to very short XMI representations that did not exceed the LLM's maximum sequence length constraint.

These observations suggest that two related factors critically constrained model performance in this setting. First, the limited maximum sequence length of the underlying LLM makes it impractical to process full-scale class diagrams, which often exceed this limit even before completion is attempted. Second, representing entire diagrams as a single generation target substantially increases the burden placed on the model, amplifying the risk of truncation, loss of structural coherence, and output degeneration. As a consequence,

evaluation metrics in this pilot setup were either uniformly 0 or exhibited extremely low values across all metrics.

These findings directly motivated the partitioning strategy adopted in the main experimental pipeline introduced in Section 3.2.3, where class diagrams are decomposed into smaller, structurally valid XMI fragments. This design choice ensures that both input and output sequences remain within the boundary of maximum sequence length supported by the underlying model while preserving syntactic correctness and structural semantics. By adopting this approach, we were able to fine-tune all model configurations without encountering errors or problems related to maximum sequence length and obtain meaningful and valuable results that we present and discuss in Chapter 4.

From a broader perspective, the pilot study highlights that performance limitations observed in RQ1 are not only due to the choice of external knowledge, but are also strongly influenced by representational constraints and task formulation. Without addressing sequence length and structural granularity, even domain-adapted fine-tuning strategies fail to produce useful and interpretable outputs. This reinforces the conclusion that effective UML model completion with LLMs requires not only appropriate training data, but also careful consideration of how software models are serialized, segmented, and presented to the model.

Table A.1: Types of UML diagrams with the corresponding group and description.

Diagram Type	Group	Description
1. Class Diagram	Structure	Represents the static structure of a system, showing classes and their attributes, interfaces, operations, and relationships [17, 36].
2. Component Diagram	Structure	Displays the system's physical components and their relation [17, 36].
3. Composite Structure Diagram	Structure	Represents the internal structure of the class by hierarchically decomposing it [17].
4. Deployment Diagram	Structure	Shows the physical layout of a system, describing which software parts run on what pieces of hardware [17].
5. Object Diagram	Structure	Similar to class diagrams, displays the snapshots of the objects in a system at a point in time, together with their attributes and relationships between multiple objects [17].
6. Package Diagram	Structure	Organizes model elements into packages and shows dependencies between these packages [17].
7. Profile Diagram	Structure	Provides an extension mechanism to custom UML diagrams for specific domains [17].
8. Activity Diagram	Behavior	Models control workflows in a system, describing the sequential steps and activities [17, 36].
9. Communication Diagram	Behavior	Describes the interaction between participants and the data flow between them [17].
10. Interaction Overview Diagram	Behavior	Combines together sequence and activity diagrams to present the control flow [17].
11. Sequence Diagram	Behavior	Captures the behavior of a scenario, showing what objects are involved and how they interact with one another [17].
12. State Machine Diagram	Behavior	Models the behavior of an object based on the state the object is currently in [17, 36].
13. Timing Diagram	Behavior	Focuses on timing constraints for one or multiple objects, describing the behavior change over time [17].
14. Use Case Diagram	Behavior	Captures functional requirements of a system from the user's perspective, describing the interaction between the user and the system [17].

Statement on the Usage of Generative Digital Assistants

For this thesis, the following generative digital assistants have been used: I have used ChatGPT[1] was used primarily to obtain inspiration for the organization of selected sections and to assist with text revision. It was mainly used to improve the structure of the sentences, the grammar and spell-checking of my writing. This tool was not used to generate core scientific content, experimental results, or conclusions.

Additionally, I have used Copilot[2] for code completion and plots visualization. Its application focused on automating repetitive coding patterns, rather than on designing algorithms or defining experimental logic.

I am aware of the potential dangers of using these tools and have used them sensibly with caution and with critical thinking. Outputs generated by these tools were never adopted without careful review. All suggestions were manually evaluated and only incorporated when they aligned with my own understanding and intent. The conceptual foundations, methodological design, implementation, and interpretation of results were developed independently.

1 https://chatgpt.com
2 https://github.com/features/copilot

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473.

[2] Averi Bates, Ryan Vavricka, Shane Carleton, Ruosi Shao, and Chongle Pan. *Unified Modeling Language Code Generation from Diagram Images Using Multimodal Large Language Models*. 2025. arXiv: 2503.12293 [cs.SE]. URL: https://arxiv.org/abs/2503.12293.

[3] Maxim Bragilosvki, Fabiano Dalpiaz, and Arnon Sturm. "From user stories to domain models: Recommending relationships between entities." In: *Proceedings of the Workshop on Natural Language Processing in Requirements Engineering (NLP4RE'23)*. CEUR WS. 2023, pp. 1–11.

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Vol. 1. Sept. 2012. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[5] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[6] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. *Towards using Few-Shot Prompt Learning for Automating Model Completion*. 2022. arXiv: 2212.03404 [cs.SE]. URL: https://arxiv.org/abs/2212.03404.

[7] Guanzheng Chen, Fangyu Liu, Zaiqiao Meng, and Shangsong Liang. *Revisiting Parameter-Efficient Tuning: Are We Really There Yet?* 2022. arXiv: 2202.07962 [cs.CL]. URL: https://arxiv.org/abs/2202.07962.

[8] Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. "Automated Domain Modeling with Large Language Models: A Comparative Study." In: *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2023, pp. 162–172. DOI: 10.1109/MODELS58315.2023.00037.

[9] Michael Han Daniel Han and Unsloth team. *Unsloth*. 2023. URL: http://github.com/unslothai/unsloth.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG]. URL: https://arxiv.org/abs/2305.14314.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[12] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. *A Survey on In-context Learning*. 2024. arXiv: 2301.00234 [cs.CL]. URL: https://arxiv.org/abs/2301.00234.

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. " Large Language Models for Software Engineering: Survey and Open Problems." In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008. URL: https://doi.ieeecomputersociety.org/10.1109/ICSE-FoSE59343.2023.00008.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[15] Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. *Model Generation with LLMs: From Requirements to UML Sequence Diagrams*. 2024. arXiv: 2404.06371 [cs.SE]. URL: https://arxiv.org/abs/2404.06371.

[16] Wikimedia Foundation. *Wikimedia Downloads*. URL: https://dumps.wikimedia.org.

[17] Martin Fowler and Kendall Scott. *UML Distilled: A brief Guide to the Standard Object Modeling Language*. Jan. 2003, pp. 2–17, 35–60, 99–104. ISBN: 0-321-19368-7.

[18] Jiawei Han, Jian Pei, and Hanghang Tong. "Chapter 2 - Data, measurements, and data preprocessing." In: *Data Mining (Fourth Edition)*. Ed. by Jiawei Han, Jian Pei, and Hanghang Tong. Fourth Edition. Morgan Kaufmann, 2023, pp. 23–84. ISBN: 978-0-12-811760-6. DOI: https://doi.org/10.1016/B978-0-12-811760-6.00012-6. URL: https://www.sciencedirect.com/science/article/pii/B9780128117606000126.

[19] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs)*. 2023. arXiv: 1606.08415 [cs.LG]. URL: https://arxiv.org/abs/1606.08415.

[20] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[22] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. "Large language models for code completion: A systematic literature review." In: *Computer Standards & Interfaces* (2024), p. 103917.

[23] John Hutchinson, Jon Whittle, and Mark Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure." In: *Science of Computer Programming* 89 (2014). Special issue on Success Stories in Model Driven Engineering, pp. 144–161. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0167642313000786.

[24] JuanjoLopez19. *JuanjoLopez19/Software-Engineering-Dataset*$_9 0_1 0_E N$. URL: https://huggingface.co/datasets/JuanjoLopez19/Software-Engineering-Dataset_90_10_EN.

[25] Yash Khare, Viraj Bagal, Minesh Mathew, Adithi Devi, U Deva Priyakumar, and CV Jawahar. "MMBERT: Multimodal BERT Pretraining for Improved Medical VQA." In: *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*. 2021, pp. 1033–1036. DOI: 10.1109/ISBI48211.2021.9434063.

[26] Hatice Koc, Ali Mert Erdoğan, Yousef Barjakly, and Serhat Peker. "UML Diagrams in Software Engineering Research: A Systematic Literature Review." In: *Proceedings* 74 (Mar. 2021), p. 13. DOI: 10.3390/proceedings2021074013.

[27] Taku Kudo. *Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates*. 2018. arXiv: 1804.10959 [cs.CL]. URL: https://arxiv.org/abs/1804.10959.

[28] Tanja Kuschke, Patrick Mäder, and Patrick Rempel. "Recommending Auto-completions for Software Modeling Activities." In: *Model-Driven Engineering Languages and Systems (MODELS 2013)*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke. Vol. 8107. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2013, pp. 199–214. DOI: 10.1007/978-3-642-41533-3_11.

[29] Quentin Lhoest et al. "Datasets: A Community Library for Natural Language Processing." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 175–184. arXiv: 2109.02846 [cs.CL]. URL: https://aclanthology.org/2021.emnlp-demo.21.

[30] Raymond Li et al. *StarCoder: may the source be with you!* 2023. arXiv: 2305.06161 [cs.CL]. URL: https://arxiv.org/abs/2305.06161.

[31] Qibang Liu, Wenzhe Wang, and Jeffrey Willard. *Effects of Prompt Length on Domain-specific Tasks for Large Language Models*. 2025. arXiv: 2502.14255 [cs.CL]. URL: https://arxiv.org/abs/2502.14255.

[32] José Antonio Hernández López, Javier Luis Cánovas Izquierdo, and Jesús Sánchez Cuadrado. "ModelSet: a dataset for machine learning in model-driven engineering." In: *Softw. Syst. Model.* 21.3 (2022), pp. 967–986. URL: https://doi.org/10.1007/s10270-021-00929-3.

[33] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. "LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning." In: *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 2023, pp. 647–658. DOI: 10.1109/ISSRE59848.2023.00026.

[34] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[35] Aaron Nicolson and Kuldip K Paliwal. "Masked multi-head self-attention for causal speech enhancement." In: *Speech Communication* 125 (2020), pp. 80–96.

[36] Object Management Group. *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. https://www.omg.org/spec/UML/2.5.1/About-UML. 2017.

[37] OpenAI. "GPT-4 Technical Report." In: *arXiv preprint arXiv:2303.08774* (2023). URL: https://arxiv.org/abs/2303.08774.

[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. "BLEU: a Method for Automatic Evaluation of Machine Translation." In: (Oct. 2002). DOI: 10.3115/1073083.1073135.

[39] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. "The impact of ai on developer productivity: Evidence from github copilot." In: *arXiv preprint arXiv:2302.06590* (2023).

[40] Alec Radford and Karthik Narasimhan. "Improving Language Understanding by Generative Pre-Training." In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG]. URL: https://arxiv.org/abs/1910.10683.

[42] Mohaimenul Azam Khan Raiaan, Md. Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. "A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges." In: *IEEE Access* 12 (2024), pp. 26839–26874. DOI: 10.1109/ACCESS.2024.3365742.

[43] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: https://arxiv.org/abs/2308.12950.

[44] Bran Selic. "Models, Software Models and UML." In: *UML for Real: Design of Embedded Real-Time Systems*. Ed. by Luciano Lavagno, Grant Martin, and Bran Selic. Boston, MA: Springer US, 2003, pp. 1–16. ISBN: 978-0-306-48738-5. DOI: 10.1007/0-306-48738-1_1. URL: https://doi.org/10.1007/0-306-48738-1_1.

[45] Yiqiu Shen, Laura Heacock, Jonathan Elias, Keith D. Hentel, Beatriu Reig, George Shih, and Linda Moy. "ChatGPT and Other Large Language Models Are Double-edged Swords." In: *Radiology* 307.2 (2023). PMID: 36700838, e230163. DOI: 10.1148/radiol.230163. eprint: https://doi.org/10.1148/radiol.230163. URL: https://doi.org/10.1148/radiol.230163.

[46] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. *Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code*. 2025. arXiv: 2310.10508 [cs.SE]. URL: https://arxiv.org/abs/2310.10508.

[47] Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, and Dahua Lin. *Code Needs Comments: Enhancing Code LLMs with Comment Augmentation*. 2024. arXiv: 2402.13013 [cs.CL]. URL: https://arxiv.org/abs/2402.13013.

[48] Christof Tinnes, Alisa Welter, and Sven Apel. *Software Model Evolution with Large Language Models: Experiments on Simulated, Public, and Industrial Datasets*. 2024. arXiv: 2406.17651 [cs.SE]. URL: https://arxiv.org/abs/2406.17651.

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[51] Vezora. *Vezora/Tested-143k-Python-Alpaca*. 2024. URL: https://huggingface.co/datasets/Vezora/Tested-143k-Python-Alpaca.

[52] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, 382–394. ISBN: 9781450394130. DOI: 10.1145/3540250.3549113. URL: https://doi.org/10.1145/3540250.3549113.

[53] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. "Codet5+: Open code large language models for code understanding and generation." In: *arXiv preprint arXiv:2305.07922* (2023).

[54] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: https://arxiv.org/abs/2109.00859.

[55] Jonathan J. Webster and Chunyu Kit. "Tokenization as the initial phase in NLP." In: COLING '92. Nantes, France: Association for Computational Linguistics, 1992, 1106–1110. DOI: 10.3115/992424.992434. URL: https://doi.org/10.3115/992424.992434.

[56] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. *TRL: Transformer Reinforcement Learning*. https://github.com/huggingface/trl. 2020.

[57] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. "Incorporating external knowledge through pre-training for natural language to code generation." In: *arXiv preprint arXiv:2004.09015* (2020).

[58] Song Yang and Houari Sahraoui. "Towards automatically extracting UML class diagrams from natural language specifications." In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, 396–403. ISBN: 9781450394673. DOI: 10.1145/3550356.3561592. URL: https://doi.org/10.1145/3550356.3561592.

[59] Wenhao Zhu, Hongyi Liu, Qingxiu Dong, Jingjing Xu, Shujian Huang, Lingpeng Kong, Jiajun Chen, and Lei Li. *Multilingual Machine Translation with Large Language Models: Empirical Results and Analysis*. 2024. arXiv: 2304.04675 [cs.CL]. URL: https://arxiv.org/abs/2304.04675.