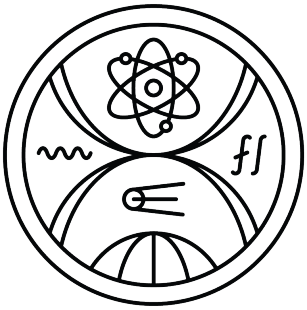
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

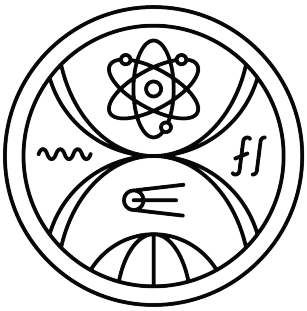


AI-ASSISTED SOFTWARE MODELLING

Master thesis

2025 Bc. Viktor Ovchinnikov

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



AI-ASSISTED SOFTWARE MODELLING

Master thesis

Study program:	Applied informatics
Branch of study:	Applied informatics
Department:	Department of Applied Informatics
Supervisor:	Ing. Lukáš Radoský

Bratislava, 2025 Bc. Viktor Ovchinnikov

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Viktor Ovchinnikov
Študijný program: aplikovaná informatika (konverzný program)
(Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: AI-assisted software modelling
Modelovanie softvéru asistované umelou inteligenciou

Anotácia: Rozsiahle jazykové modely rozumejú okrem prirodzeného jazyka aj formálnym jazykom, vrátane programovacích jazykov. Schopnosti týchto modelov im umožňujú modifikovať, generovať, alebo, naopak, interpretovať zdrojový kód v mnohých jazykoch. Komerčné riešenia pre programovanie s asistentom založeným na rozsiahlych jazykových modeloch, resp. modeloch umelej inteligencie sú dnes bežne dostupné.

Tak ako vo fáze implementácie sú využívané programovacie jazyky, vo fáze návrhu sú využívané modely, resp. diagramy softvéru. Takéto diagramy možno zapísať pomocou deklaratívnych jazykov, napr. PlantUML. Analyzujte možnosti využitia rozsiahlych jazykových modelov pri modelovaní softvéru. Implementujte prototyp pre modelovanie softvéru asistované rozsiahlym jazykovým modelom.

Cieľ: Analýza existujúcich riešení modelovania softvéru asistovaného rozsiahlym jazykovým modelom
Implementácia prototypu pre modelovanie softvéru asistované rozsiahlym jazykovým modelom

Literatúra:
Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, & Lingming Zhang. (2024). Agentless: Demystifying LLM-based Software Engineering Agents.

Radford et al. Better language models and their implications.

A. Fan et al., "Large Language Models for Software Engineering: Survey and Open Problems," 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), Melbourne, Australia, 2023, pp. 31-53, doi: 10.1109/ICSE-FoSE59343.2023.00008.

Kľúčové slová: jazykový model, modelovanie softvéru, virtuálny asistent

Vedúci: Ing. Lukáš Radoský
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátum zadania: 18.11.2024

Dátum schválenia: 21.11.2024 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

I hereby declare that I have written this entire thesis, including all its appendices and images, independently, using the literature listed in the attached list and artificial intelligence tools. I declare that I have used artificial intelligence tools in accordance with applicable laws, academic rights and freedoms, ethical and moral principles, while maintaining academic integrity, and that their use is appropriately indicated in the work.

In preparing this work, Deepl Write AI tools were used for the purpose of correcting English grammar. The author is responsible for the accuracy of the final text.

. . . . . .

Bratislava, 2025 Bc. Viktor Ovchinnikov

Acknowledgement

I would like to thank all my friends and family who supported me while I was writing this paper. I would especially like to thank my parents, who always support me in difficult times with kind words and advice. I would also like to express my enormous gratitude to my supervisor, Ing. Lukáš Radoský, who made this work possible, for his quick and high-quality feedback on my work and his expertise in the field.

Abstract

Large language models understand formal languages, including programming languages, in addition to natural language. The capabilities of these models allow them to modify, generate, or, conversely, interpret source code in many languages. Commercial solutions for assistant programming based on large-scale language models or artificial intelligence models are widely available today. Just as programming languages are used in the implementation phase, software models or diagrams are used in the design phase. Such diagrams can be written using declarative languages such as PlantUML. The present study analyzes the potential applications of large language models in the field of software modeling. The implementation of a prototype for software modeling is achieved through the utilization of a large-scale language model. We initiated the development of a copilot system with the objective of enhancing the functionality of the existing AnimArch project. The primary objective of this system is to provide assistance to the user in the domain of software modeling.

Keywords: LLM, UML, Class Diagrams, Software prototyping

Abstrakt

Veľké jazykové modely rozumejú okrem prirodzeného jazyka aj formálnym jazykom vrátane programovacích jazykov. Schopnosti týchto modelov im umožňujú upravovať, generovať alebo naopak interpretovať zdrojový kód v mnohých jazykoch. Komerčné riešenia na asistenčné programovanie založené na rozsiahlych jazykových modeloch alebo modeloch umelej inteligencie sú dnes široko dostupné. Tak ako sa programovacie jazyky používajú vo fáze implementácie, tak sa softvérové modely alebo diagramy používajú vo fáze návrhu . Takéto diagramy možno napísať pomocou deklaratívnych jazykov, ako je PlantUML. Táto štúdia analyzuje potenciálne aplikácie veľkých jazykových modelov v oblasti modelovania softvéru. Implementácia prototypu pre modelovanie softvéru sa dosahuje prostredníctvom využitia rozsiahleho jazykového modelu. Začal sa vývoj kopilotného systému s cieľom rozšíriť funkčnosť existujúceho projektu AnimArch. Primárnym cieľom tohto systému je poskytovať pomoc používateľovi v oblasti modelovania softvéru.

Kľúčové slová: LLM, UML, Diagramy tried, Prototypovanie softvéru

Contents

List of Figures

List of Tables

Terminology

Terms

- **Few-shot prompting**
  Involves providing a limited number of examples or instructions to the model to perform a specific task. The model learns from these examples and generalizes to similar tasks with minimal training data.
- **Zero-Shot Prompting**
  In zero-shot prompting, the model is expected to perform a task without any explicit training on that task.
- **Embedding**
  Vector representation of text (words, phrases, documents, etc.) obtained by a model such that semantically similar objects have similar vectors in a certain feature space.

Abbreviations

- **LLM** - Large Language Model.
- **RNN** - Recurent Neural Network.
- **BERT** - Bidirectional Encoder Representations from Transformers.
- **RAG** - Retrieval Augmented Generation.
- **DSL** - Domain-specific language.
- **IDE** - Integrated development environment

Motivation

LLM has recently become popular in a variety of fields, including software engineering. The recent development of software solutions has led to the emergence of tools capable of automating the process of code completion within IDE and code editors. The decision was taken to assess the capabilities of LLM in a slightly different branch of software engineering: software modeling. The objective of this study is to examine the capacity of LLM to propose relevant suggestions to the user and accelerate software modeling.

Chapter 1

Large Language Models

In recent years, large language models (LLMs) have become a key technology in the field of modern artificial intelligence. In general terms, an LLM is understood to be an artificial intelligence model that has been trained on a huge corpus of text data and is capable of generating text in a "human-like" manner. Such models are typically founded upon deep neural networks, most frequently the Transformer architecture, and evince the capacity to facilitate the execution of a broad spectrum of linguistic operations. These range from text generation and question answering to text translation into other languages and summarising large texts into smaller ones [4].

The present state of LLMs has been chiefly influenced by two factors: the increase in the volume of available text data, and the growth of computing power. It is also important to acknowledge the significant contribution to modern language models made by the Transformer architecture with its self-attention mechanism. The seminal paper of Vaswani et al. proposed this architecture [12], marking the advent of the modern era of LLMs and precipitating a surge in research in this domain [4]. It is on this basis that the GPT family of models was developed and trained on terabytes of text and has hundreds of billions of parameters, as is typical of more recent models. Additionally, there are open models, such as LLaMA, which have greater variability in the number of parameters [4].

The existing literature does not provide a consensus on the number of parameters in a language model that can be considered large. The capabilities of a model are contingent not only on the number of parameters, but also on the volume of text data utilised during the training process. Hou et al. observe that the term LLM came into existence to denote "large" pre-trained language models, and underscore the correlation between augmenting the scale of models and the advent of novel capabilities [6].

1.1 LLMs

1.1.1 Architectures and main types of LLM.

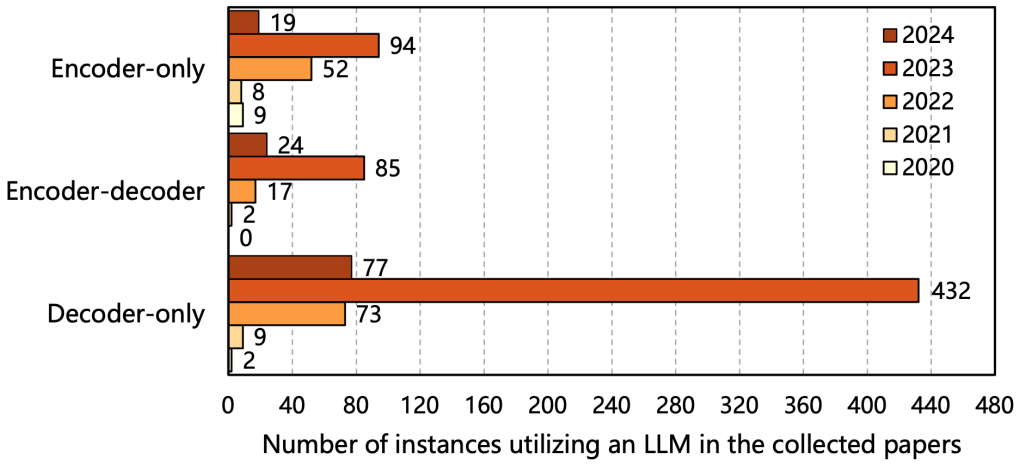


Figure 1.1: Trends in the application of LLMs with different architectures in SE tasks over time [6].

Despite the variety of specific models, the modern literature distinguishes three basic architectural types of LLM based on the Transformer approach: encoder-only, decoder-only, and encoder-decorer. This classification is significant because each type of model is optimal for specific tasks.

Encoder-only models use only the encoder part of the Transformer. The objective of the model is to convert the input sequence into a hidden representation (embedding) that accurately reflects the context and relationships between tokens. Typical examples of this approach include BERT and its derivatives (RoBERTa, DeBERTa). These models have been shown to be particularly effective in tasks related to text understanding, such as classification, entity extraction, similarity assessment, and search, as well as in a number of code analysis tasks where accurate representation of the input is required (e.g., defect search or bug localization) [6].

Decoder-only models, on the other hand, use only a decoder and are trained in autoregressive mode, predicting the subsequent token based on the previously generated sequence and the input context. This class includes the GPT series, LLaMA, Claude, PaLM, as well as a number of specialised models for code (Codex, Code Llama, StarCoder, etc.). The "prompt to continuation" scheme is a particularly useful model in interactive application scenarios, including dialogue systems, text and code autocompletion, question answering and documentation creation [6].

Encoder-decoder models combine an encoder, which builds a hidden representation of the input, and a decoder, which generates the output sequence based on this rep-

resentation. Examples of this include T5 and BART, as well as specialized variants for code, such as CodeT5 and AlphaCode. The application of these models extends to domains in which both understanding and generation are crucial, such as machine translation, summarisation, paraphrasing, and translation between programming languages [6]. The figure 1.1 illustrates trends in the use of different types of LLM architectures. The figure indicates a significant decrease in the number of works in 2024. This is because the system review from which the figure was taken considers works published until January 2024.

Hou et al. present the findings of an extensive analysis of 395 studies, which demonstrate that the distribution of architectures across tasks in applied work is relatively stable [6]. The analysis reveals that encoder-only models predominate in tasks requiring deep interpretation of input (e.g., code understanding, vulnerability localization). Conversely, encoder-decoder models are more frequently employed in "understanding + generation" tasks (e.g., summarisation, translation, programme repair). Finally, decoder-only models dominate in generation tasks (e.g., generation and addition of code, tests, comments). A more detailed distribution of LLMs is shown in Figure 1.2.

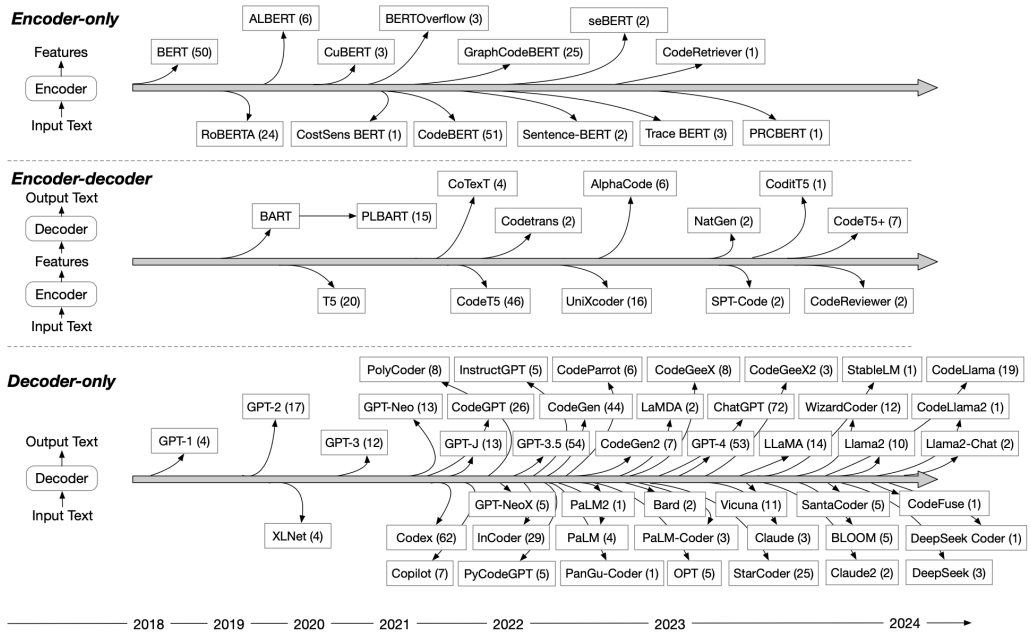


Figure 1.2: Distribution of the LLMs (as well as LLM-based applications) discussed in the collected papers by Hou et al. The numbers in parentheses indicate the count of papers in which each LLM has been utilized [6].

1.1.2 Challenges of LLMs.

Despite the impressive results obtained in both studies, it is emphasized that LLMs possess a number of specific properties that simultaneously create new opportunities

and serious challenges.

Firstly, the authors observe the emergent properties of LLMs – abilities that are unexpected and not explicitly programmed or specifically trained, but arise when a certain scale of model and data is reached. This phenomenon has been observed in a variety of contexts, including positive effects such as the ability to generalize, to reason in multiple steps, and to transfer between tasks, as well as negative manifestations such as hallucination, defined as the generation of plausible but factually incorrect or fictitious content. Within the domain of software engineering, this pertains to the potential for solutions that are syntactically correct yet semantically erroneous (e.g. code with errors), a topic that will be revisited subsequently.

Secondly, Fan et al. emphasize the non-deterministic nature of inference: with an unchanged prompt, the model can produce different answers on different runs, and small changes in the wording of the query can lead to significant differences in the results [4]. To consider the matter from two perspectives, this process has been shown to increase creativity and provide space for finding alternatives. However, on the other hand, it complicates reproducible scientific evaluation and industrial exploitation. It is difficult to say whether a new setting has improved the system if the generation results are statistically noisy.

Thirdly, Hou et al. underline the complexity of evaluating and selecting LLMs for specific tasks [6]. Researchers and practitioners must consider a number of factors, including model architecture, training corpus size and type, fine-tuning strategies (such as fine-tuning and parameter-efficient fine-tuning), prompt engineering techniques, and a wide range of domain-dependent quality metrics.

Fan et al. make an important conclusion in this regard: hybrid approaches combining LLM with traditional verification methods (testing, analysis, static and dynamic checks) are critical for engineering tasks [4]. LLMs are not regarded as a standalone "magical" solution generator but as a powerful yet fallible component whose results need to be verified and filtered.

1.1.3 The range of tasks solved by LLMs.

From the perspective of task classes, LLMs are considered to be a universal instrument for working with any entities that can be represented as a sequence of tokens, including natural languages, formal languages, code, and markup, among others. Fan et al. provide a comprehensive overview of the fundamental language tasks in which LLMs have already demonstrated proficiency: coherent text generation, question answering, machine translation, text summarisation, and sentiment analysis [4].

It is important to note that these capabilities are realized within a single architecture. A model that has been trained on large collections of text is often capable

of solving many tasks without the need for additional specialized layers, using only prompt engineering.

Hou et al. further develop this approach by treating a wide range of software engineering tasks as variations of text analysis and generation, including source code, defect reports, messages in task tracking systems and documentation [6]. This viewpoint enables the application of LLM to a variety of practical tasks, including code summarization, comment generation, automatic program correction and vulnerability detection.

The scale of LLM application requires particular consideration. An analysis of the relevant literature, as conducted by Fan et al., demonstrates an exponential growth in the number of publications related to LLM in general, and LLM in the context of software engineering in particular [4]. By 2023, it is projected that more than 10% of all papers on LLM will be in the field of LLM-based software engineering, thereby illustrating the rapid integration of these models into engineering practice [4].

1.2 Applications of LLMs in SE.

As previously indicated, the preceding discussion focused on large language models in a general capacity, encompassing their architectures and the typical areas of application. The present discussion will narrow its focus to address the role of these models in the field of software engineering.

1.2.1 The range of tasks for LLM in software engineering.

Recent reviews demonstrate that LLMs are already being employed in a wide range of software development activities. Hou et al. provide confirmation of this hypothesis at the level of a systematic review [6]. The classification of 395 works undertaken by the authors reveals that tasks are divided into six types of life cycle activities: requirements, design, development, quality assurance, maintenance, and management. The classification, recommendation and analysis tasks account for a significant share in addition to code generation. In the domain of requirements, this encompasses the management of anaphoric ambiguity, the extraction of traceability, and the generation of use cases. In the realm of design, rapid prototyping and specification synthesis are integral components. In the domain of quality assurance, test generation, vulnerability scanning, and log analysis are crucial. Finally, in the context of maintenance, automatic program repair and clone detection are essential [6].

As Fan et al. observe, the output of LLM is not necessarily limited to code; it may also extend to other development artifacts, including requirements, test scenarios, design diagrams and documentation. The authors emphasize that the linguistic

nature of LLM allows it to generate "any linguistically-defined software engineering artifact" [4]. This observation directly opens up the possibility of considering not only text and program code, but also models, in particular UML diagrams, as another target area of application for LLM, provided that suitable text representation and verification mechanisms are provided for them.

1.2.2 Code generation and code completion based on LLMs.

The systematic review by Hou et al. provides further insight into this area by offering a more detailed classification of tasks and models. The authors analyze a wide range of LLMs used in SE and identify key application areas, including code generation, code completion, summarisation, code search, API interaction, etc. A separate subsection is devoted specifically to code completion: Hou et al. consider code completion to be an assistive feature provided by many modern IDEs and editors designed to automatically display possible code options as the programmer types. The review posits that the function has evolved from n-gram and RNN models to transformers, such as GitHub Copilot and CodeGPT, which have been trained on extensive sets of source code [6].

Hou et al. demonstrate that present-day LLMs with billions of parameters are proficient in the generation of code snippets. Utilizing the context and syntactic structure of the program, they are capable of interpreting the developer's intentions, predicting subsequent snippets, and providing relevant recommendations [6].

Fan et al. adopt a similar approach, yet offer a distinct perspective by exploring the impact of specific models on the development process. In the section on LLM for software engineering, it is noted that in 2021, OpenAI introduced Codex, a descendant of GPT-3 used in GitHub Copilot to provide hints to developers in Visual Studio Code, Visual Studio, Neovim, and JetBrains IDE. According to GitHub's February 2023 report, the average proportion of code written by developers that was written by Copilot was 46%, with this figure rising to 62% in the case of Java. The authors also cite the results of an empirical study, according to which programmers who used Copilot completed a non-trivial task (implementing an HTTP server in JavaScript) 56% faster than the control group. In this context, Fan et al. (2023) highlight that as LLM-based code completion becomes more widespread, a transition in the role of developers is anticipated: a greater proportion of their time will be allocated to reviewing and verifying generated code, as opposed to manual writing [4].

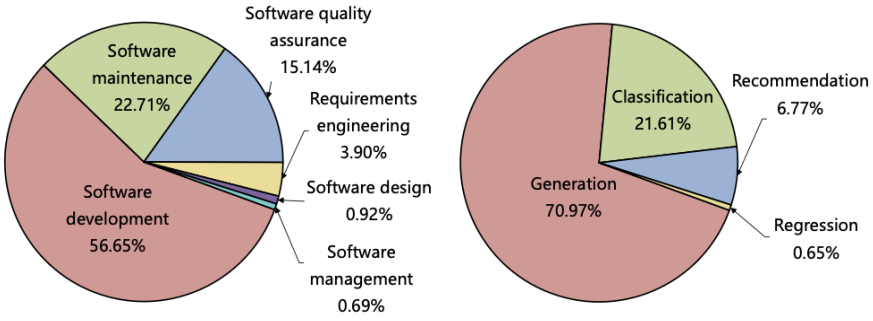
1.3 From code autocompletion to assisted UML model construction.

In this context, the specific API of such tools is of lesser importance than the general interaction pattern. LLM works as a "pair programming partner", offering useful ideas to improve the existing artifact. Fan et al. underscore that the model output can encompass not only code but also design diagrams, signifying that the principles of contextual autocompletion may be applicable to visual models such as UML class diagrams [4]. This establishes a seamless transition from existing integrated development environment (IDE) assistants to systems where a large language model (LLM) generates suggestions for developing or correcting a partially constructed model. In this context, a specialized infrastructure (in our case, a tool for processing UML diagrams) is responsible for parsing, verifying and visualizing changes. In the ensuing subsection, this observation is utilized as a point of departure for establishing the task of an assistant for constructing UML class diagrams. This task is analogous to tools such as GitHub Copilot, although it operates at the level of project models as opposed to source code.

Chapter 2

Related Work

In recent years, large language models (LLMs) have expanded the toolkit of software engineers. LLMs have the potential to be beneficial in a variety of domains within software engineering. The most prevalent and extensively researched domain of application for LLMs is code generation using software tools such as Github Copilot or Cursor [6]. However, it is important to note that LLMs demonstrate great potential in many other areas. In the context of our research, our focus is on a higher-level abstraction task: software modeling. In contrast, assistant systems in this area receive less attention than those in the field of code generation (0.92% for Software design), as shown in figure 2.1 and table 2.1.



(a) Distribution of LLM usages in SE activities. (b) Problem classification based on collected studies.

Figure 2.1: Distribution of LLM utilization across different SE activities and problem types [6].

2.1 OCL vs PlantUML in the context of class diagram

In work of Camara et al. [2] both advantages and limitations of LLMs are discussed. The report on ChatGPT for building UML class diagrams demonstrates that for minor tasks, the model is consistent with syntax (PlantUML). However, it encounters challenges with semantics, reproducibility, and scalability of extensive solutions comprising

Table 2.1: Distribution of SE tasks over six SE activities [6].

SE Activity	**SE Task**		**Total**
Requirements engineering	Anaphoric ambiguity treatment (4) Requirement analysis and evaluation (2) Coreference detection (1) Specification formalization (1) Use cases generation (1)	Requirements classification (4) Specification generation (2) Requirements elicitation (1) Traceability automation (1)	17
Software design	GUI retrieval (1) Software specification synthesis (1)	Rapid prototyping (1) System design (1)	4
Software development	Code generation (118) Code summarization (21) Code translation (12) API inference (5) API recommendation (5) Code representation (3) Method name generation (2) Agile story point estimation (1) API documentation smells (1) Data analysis (1) Control flow graph generation (1) Instruction generation (1) Others (14)	Code completion (22) Code search (12) Code understanding (8) Program synthesis (6) Code editing (5) Code comment generation (2) Code recommendation (2) API documentation augment (1) API entity and relation extraction (1) Fuzz driver generation (1) Identifier normalization (1) Type inference (1)	247
Software quality assurance	Vulnerability detection (18) Bug localization (5) Testing automation (4) Defect detection (2) Static analysis (2) Compiler fuzzing (1) Invariant prediction (1) Mobile app crash detection (1) Test prediction (1)	Test generation (17) Verification (5) Fault localization (3) GUI testing (2) Binary taint analysis (1) Decompilation (1) Malicious code localization (1) Resource leak detection (1)	66
Software maintenance	Program repair (35) Code review (7) Bug reproduction (3) Duplicate bug report detection (3) Log parsing (3) Sentiment analysis (3) API misuses repair (1) Bug triage (1) Code review explained (1) Crash bug repair (1) Incivility detection (1) Patch detection (1) Rename Refactoring (1) Technical debt payback (1) Web test repair (1) Others (5)	Code clone detection (8) Debugging (4) Review/commit/code classification (3) Logging (3) Code revision (2) Vulnerability repair (2) Bug prediction (1) Code coverage prediction (1) Code-Review defects repair (1) Dockerfile Repair (1) Patch correctness prediction (1) Program merge conflicts repair (1) Tag recommendation (1) Traceability recovery (1) Type error repair (1)	99
Software management	Effort estimation (2)	Software tool configuration (1)	3

more than 8-10 classes. Furthermore, there is variability in responses between sessions, errors in determining the correct relationship between classes, and weaknesses with multiple inheritance. The authors also observe that OCL generation is more reliable than PlantUML [2].

The authors observed a significant dependence of LLM performance on the domain. In domains characterized by extensive code bases, such as banks or financial technologies, ChatGPT has been observed to contribute additional details. However, flaws have been demonstrated to emerge in abstract domains. The authors of the study recommend an iterative approach to dialogue, characterized by a series of step-by-step prompts, as opposed to a single, comprehensive prompt. This recommendation stems from the observation that "one shot prompting" lacks the necessary structure to facilitate effective dialogue. Despite the implementation of iterative prompting, a substantial number of iterations are required for errors to accumulate. When considered as a whole, this restricts the reliability of LLM [2].

2.2 LLMs for sequence diagram modeling.

In the context of class diagrams, it is evident that syntactic stability is maintained despite the presence of semantic vulnerability as the number of entities in the model increases. A parallel can be drawn between sequence diagrams and the previously mentioned case. ChatGPT is capable of generating diagrams that are syntactically acceptable and readable; however, it frequently exhibits deficiencies in terms of completeness and accuracy relative to specifications, particularly in the presence of defects in the requirements [2].

In addition to the development of structural models of systems, there are studies on the generation of behavioral models from requirements expressed in natural language. For instance, in the work of Ferrari et al. [5], ChatGPT is evaluated on the task of obtaining UML sequence diagrams for 28 industrial technical requirements from different domains and requirement formats. The authors observe a diverse array of quality requirements. The degree of understandability, standard compliance, and terminological alignment is at an average or good level. However, the completeness and correctness of the process are susceptible to deficiencies, particularly in circumstances involving ambiguity or inconsistency in the stipulated requirements [5].

The authors have published a catalog of 23 categories of problems, including missing elements or conditions, incorrect interactions or structures, and unnecessary actors. There are discernible parallels with the preceding work that has been examined. For instance, the model's potential vulnerability in a particular domain can result in suboptimal outcomes. The authors also noted the need for an iterative prompting strategy

and richer context. The authors have indicated the presence of considerable variability in LLM responses; however, this variability can also be advantageous, provided that the validity of the response can be verified [5].

2.3 Generating UML class diagrams from source code.

A comparison of structural and behavioral modeling using LLM reveals a consistent limitation in performance. Although LLM demonstrates correct notation and readability in the task under consideration, significant problems are encountered when evaluating the generated model in terms of completeness and correctness. The work of Shenata et al. [10] that examines structural modeling from code reveals the presence of similar behavior. In particular, ChatGPT displays a consistent capacity to restore classes and attributes; however, its performance in restoring associations is notably weaker. On large systems, both semantic and syntactic errors increase [10].

The authors demonstrated that ChatGPT possesses the capacity to reliably generate PlantUML descriptions and restore structure, exhibiting a satisfactory level of quality. The model demonstrated its capacity to restore 90% of classes and attributes from code, while its ability to restore associations was found to be 66%. In the context of large and complex systems, the authors observed a substantial degradation in quality. The authors concur with the conclusions of numerous previously reviewed works in determining that LLM should be regarded as an auxiliary assistant rather than a fully autonomous model generator. The authors also noted the prevalence of errors specifically in relationships (omissions of key connections and additions of unnecessary ones). Syntactic and compilation errors were identified in 5 out of 40 diagrams across the most complex projects [10].

2.4 Software Model Evolution with LLMs.

A review of the literature on UML and OCL reveals significant variability and limitations in the reliability of the results. Additionally, the scalability of diagram generation in a single step is constrained, and the generation quality deteriorates as the number of UML entities increases. Consequently, these limitations have prompted a shift in the research focus from autonomous generation to user assistance scenarios. In light of these findings, the report [11] on the use of RAG (Retrieval Augmented Generation) in the context of software models demonstrate an alternative trajectory. Rather than initiating the process from the beginning, they adopt an approach that utilizes contextual autocompletion, informed by a history of changes made to the document.

Consequently, in contrast to the one-step UML generation from text, recent paper

demonstrates how LLM works as a contextual autocompletion mechanism for evolving models. Tinnes et al. [11] propose an approach that utilizes RAG. The approach employed involves the utilization of a "simple change graph" extracts similar changes from the repository and presents them in the prompt as few-shot examples. In three distinct datasets, the system proposed by the authors achieved semantically correct completions of 62.30% in industrial data and up to 86.19% type-correct completions in a synthetic dataset. At the same time, LLM shows the capability to generate more than 90% of its results in a correctly serialized format. The similarity of examples in retrieval has been demonstrated to have a significant effect on accuracy. Thus, semantic retrieval is statistically significantly higher than random retrieval [11].

A comparison of this approach with the approach of Chaaben et al. [3] (few-shot without history) demonstrates the superiority of the approach proposed by the authors. In the context of recommending new items and connections on a public dataset, this approach exhibits a substantial enhancement in performance when compared to the baseline and random retrieval methods. The authors also investigated how the number of few-shot examples affects the results. Increasing k does not provide any advantage if there are no similar change patterns between the examples [11].

2.5 Methods for assessing the quality and comparing UML diagrams.

In the context of this study, approaches to the comparison and evaluation of UML class diagrams are of significance, as they constitute the foundation for model reuse mechanisms and automatic quality control.

Nikiforova et al. propose a semi-automatic methodology for the comparison of two UML class diagrams [7]. In the first instance, semantically equal elements (classes, interfaces, methods, attributes) are identified in pairs by humans. Then, for each pair, a vector of differences is calculated according to a number of criteria, individually for each type of element. The final semantic distance between the diagrams is the Euclidean distance of the aggregated vector. The higher the value, the more significant the difference. This approach prioritizes structural equivalence, positing that the impact of renaming an element on the resulting semantic distance is less significant in comparison to that of structural changes [7]. This methodological approach permits a relatively thorough examination of structural elements; however, its practical application is constrained by the necessity of human involvement in the automatic comparison of diagram components across disparate projects or domains.

In this context, the advancement of this field was closely associated with efforts to automate the process of structural comparison. Yuan et al. propose a graph model

called UML Class Graph (UCG) for the purpose of automated structural comparison of diagrams [14]. Classes, attributes, operations, and parameters are represented as typed vertices, and relationships are represented as labeled edges; the structure of the diagram is divided into intra-structure (internal class structure) and inter-structure (relationships between classes). In the context of inter-structure, a search for a sequence of maximum common subgraphs is employed. Conversely, for intra-structure, an editorial distance on graphs is utilised. Afterwards, both components are combined to form a unified similarity metric. Experimental findings demonstrate that such a structural metric is more aligned with expert assessments and, in contrast to purely semantic methods, maintains its informativeness when comparing diagrams from different domains.

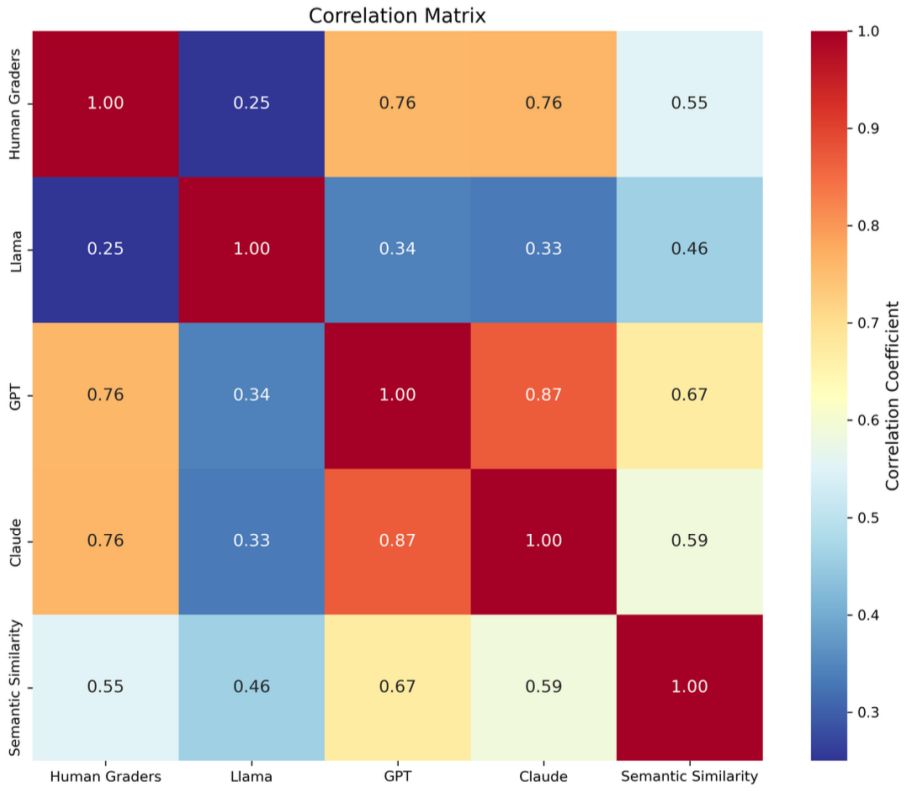


Figure 2.2: Correlation heatmap of all graders [1].

A more recent line of research is related to the automatic evaluation of UML using LLMs. The authors of the paper on automated verification of UML diagrams proceed from the assumption that class diagram tasks allow for multiple correct solutions: students can structure the model in different ways and use synonyms in names while remaining within the framework of correct semantics [1].

In such conditions, strict syntactic comparison with the reference produces a high number of "false errors". In order to account for variability, the following approach is proposed: diagrams are translated into a text description (statements about classes, re-

lationships, multiplicities) in JSON representation. Thereafter, a large language model evaluates the solution according to a predefined criteria and generates text feedback, thereby acting as a "virtual assistant-checker". The paper also considers a variant based on sentence transformers, whereby the text description of the diagram is encoded by Sentence-BERT. The evaluation is reduced to searching for semantically similar sentences between the student's solution and the reference using cosine similarity [1]. The figure 2.2 illustrates a correlation matrix that demonstrates the quality of various approaches (in comparison to human judgement) studied by the author.

The three lines of research demonstrate the evolution of approaches: from manual construction of semantic features (Nikiforova) to graph structural metrics (Yuan) and further to models based on embeddings and LLM, capable of taking into account synonyms, variable structures, and partially correct solutions. This background is of particular significance for the present research, as it demonstrates that LLMs and associated representations can be utilized not only for UML generation but also as a component of user support systems for the comparison, evaluation and editing of class diagrams.

2.6 Prompt engineering in the context of software engineering

As demonstrated in the above UML diagram, a single generation step in the process of UML diagram generation is characterized by limited reliability. At the same time, RAG with contextual autocompletion has been shown to offer an observable advantage. In the following section, the focus will shift to the examination of work on prompt patterns, which provide a formalized framework for the construction of such processes. Thus, research on the application of LLM in software engineering is gradually shifting from random approaches to prompt construction to more systematic ones. The utilization of prompt patterns assumes a pivotal role in this context, as they function as reusable constructs that systematize goals, context, and structure. This development has led to the establishment of a novel independent discipline, which has been termed "prompt engineering". The utilization of such patterns is essential for the reduction of variability in LLM responses, the enhancement of quality, and the augmentation of control in the response generation process [9].

In the present study, White et al. offer a catalog of 13 prompt design templates, which are grouped into categories according to their life cycle stage: Requirements Elicitation, System Design and Simulation, Code Quality and Refactoring in table 2.2. The catalog is further supplemented by auxiliary methods intended to overcome limitations of LLMs in context, including a Few-shot Example Generator and Domain-

specific language (DSL) Creation. The templates are uniformly described, simplifying the combination of templates into a set for specific tasks [13].

Table 2.2: Classifying Prompt Patterns for Automating Software Engineering [13].

Requirements Elicitation	Requirements Simulator Specification Disambiguation Change Request Simulation
System Design and Simulation	API Generator API Simulator Few-shot Example Generator Domain-Specific Language (DSL) Creation Architectural Possibilities
Code Quality	Code Clustering Intermediate Abstraction Principled Code Hidden Assumptions
Refactoring	Pseudo-code Refactoring Data-guided Refactoring

The practical value of the catalog is evident primarily in the initial stages of the project, specifically in the domains of requirements analysis and system design. The Requirements Simulator transforms the LLM into an interactive "system mock-up" that identifies incompleteness and contradictions in requirements through step-by-step scenarios. The Specification Disambiguation process is a systematic procedure that aims to identify and resolve ambiguities in specifications before they are incorporated into code and interfaces. The utilization of these templates, when employed in tandem with visualization and role "personification" (e.g. "act as a terminal/API"), has been demonstrated to enhance manageability and observability of the process. These formalized approaches are detailed in the catalog [13].

Templates such as the API generator and the API simulator enable the maintenance of specifications over time. The acceleration of early formalization and interface testing enables a rapid evaluation of the convenience of the proposed solution without the necessity of a full implementation. The Few-shot Example Generator and the DSL Creation module generate dense representations of the system (examples, mini-languages) that are convenient to retrieve back into the prompt, bypassing the limits of the context window. At the architectural level, Architectural Possibilities systematizes the generation of alternatives and experimental selection. When considered as a whole, these templates do not "magically improve" the quality of output; rather, they reduce the variability of responses through explicit output rules and expected formats [13].

In addition, Schmidt et al. emphasise that prompt engineering constitutes a distinct engineering discipline, within which the characteristics of quality software are equally applicable, such as maintainability, compatibility, the controlled development

of prompt sets, and the reproducibility of results when updating LLM. The authors emphasise that by transitioning from fragmented techniques to template catalogs, engineers will be able to transfer knowledge between domains and teams, reduce dependence on individual experience, and establish teamwork processes on top of standardised LLM interaction schemes [9].

Within the paradigm of software modelling employing UML diagrams, these outcomes offer a valuable opportunity to reduce the variability of LLM suggestions. Requirements templates facilitate the definition of the structure for the prompt being formed for tasks such as context extraction or few-shot training of an assistant. Furthermore, the exploitation of few-shot examples and the creation of DSLs are well-suited to RAG approaches. Few-shot examples and mini-languages have been shown to become building blocks of the entire context, which increases the stability of responses for the autocomplete system.

2.7 AnimArch.

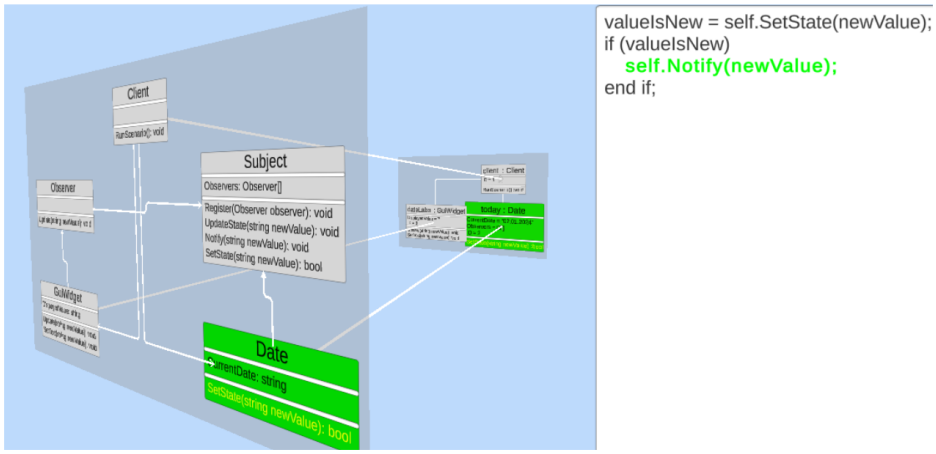


Figure 2.3: Overview of graphical user interface of AnimArch [8].

The LLM work for UML demonstrates a satisfactory level of performance in formalizing diagrams. However, it also exposes certain disadvantages, including issues with the semantics and completeness of the proposed model, as well as the necessity for external validation and iterative interaction. AnimArch offers a multi-layered environment capable of visualizing and animating software, with explicit links between the UML class diagram, the objects it generates, and the code [8].

Radosky and Polasek propose an "Executable Multi-Layered Software" approach, in which the static structure (UML class diagram) is combined with the dynamics of execution (OAL/xUML source code) and visualized in the form of animations. This approach serves to reduce the discrepancy between software design and implementation [8].

A system of this nature is great for testing whether an LLM can suggest useful edits to UML class diagrams. On this basis we are creating an assistant for class diagram construction: the LLM generates candidates for change, and AnimArch parses, validates and visualizes them.

Chapter 3

System Design Overview

The system under development is an interactive tool for modelling UML class diagrams, offering intelligent suggestions based on an analysis of the current model state and recent user actions. The client part is implemented as a Unity application that provides a graphical diagram editor and control over the suggestions mode. The editor supports the in-memory representation of UML models (classes, attributes, methods and relationships), as well as logging recent changes for the current session. This data is used as the main input for the recommendation generation subsystem.

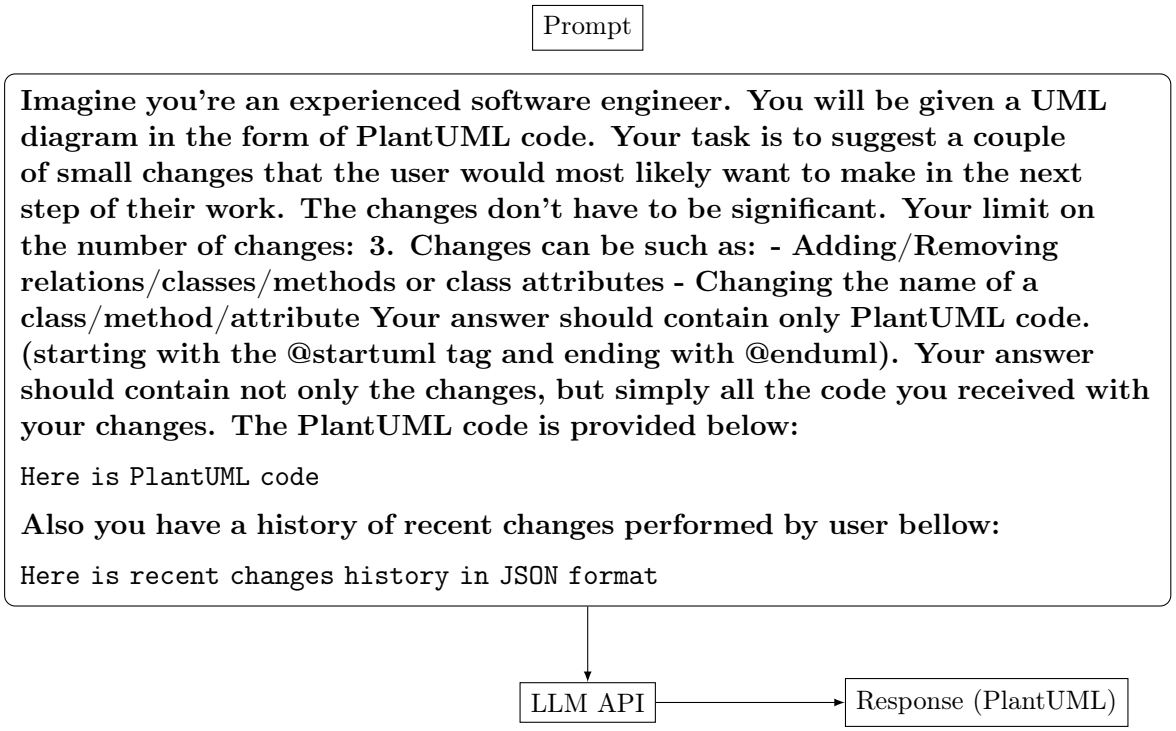


Figure 3.1: Detailed prompt used in the system.

A key element of the architecture is the interaction loop with the LLM. When suggestion mode is activated, the client application converts the current in-memory diagram representation into PlantUML code and generates a prompt (Figure 3.1) con-

taining: (1) a text instruction for the LLM; (2) a PlantUML representation of the current model; and (3) a history of recent user changes. The prompt specifies a strict response format, which is a complete diagram in PlantUML with no more than three local changes made by the LLM, such as adding or deleting entities and relationships, or renaming elements. At this stage, zero-shot prompting (i.e., prompts without additional examples related to the task) was used. This approach is used as a baseline for comparing it with other prompting methods to see how the quality of the response is affected. The LLM response is validated syntactically as PlantUML; if this is unsuccessful, the same request is resubmitted in order to obtain a correct version by exploiting the model's non-determinism. The figure 3.2 shows a state diagram detailing the system architecture.

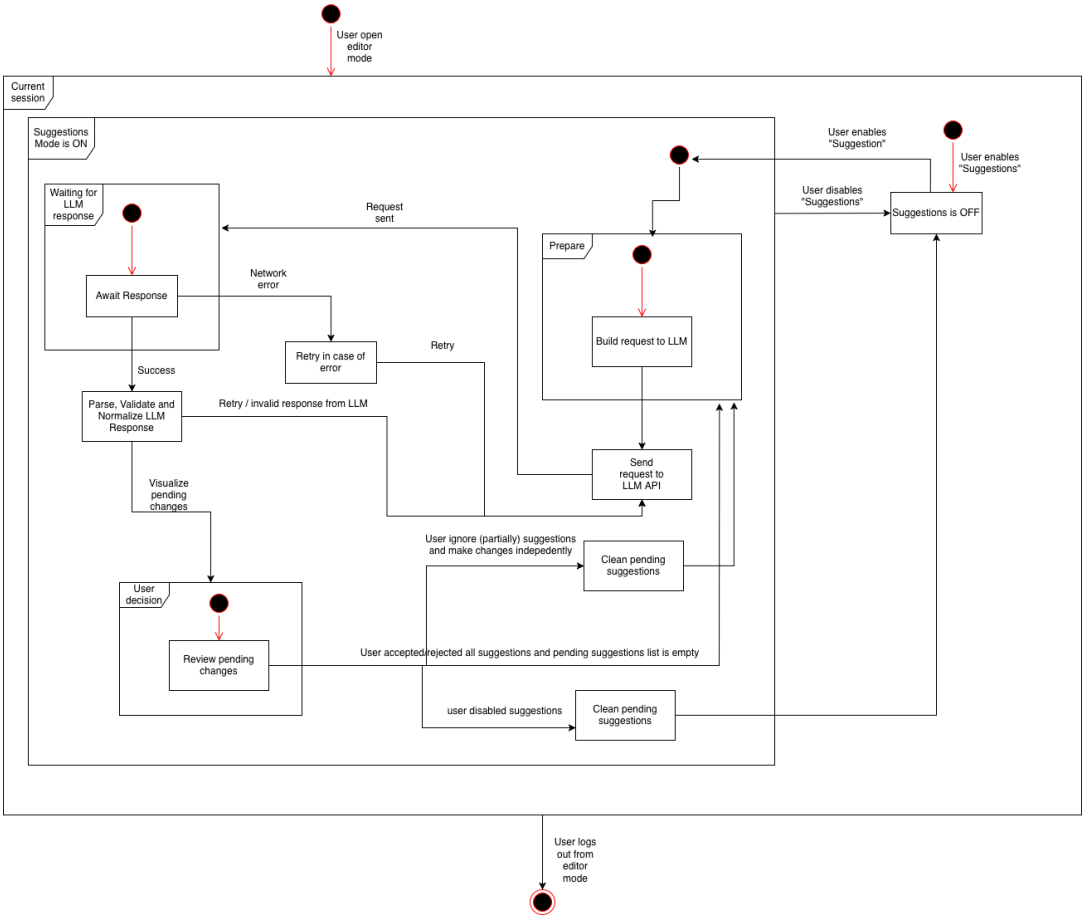


Figure 3.2: The state diagram shows the entire cycle of the suggestion mode in the editor.

Once a valid response has been received, the system calculates the difference between the current state of the diagram and the suggestion made by the LLM. Based on a comparison of the two models, the system recognises elementary operations such as adding, deleting or modifying classes, attributes, methods and relationships. These changes are translated into a set of human-readable suggestions and visualised in the

editor interface. The user can then accept or reject each suggestion individually, or apply or reject them all at once.

User feedback closes the suggestions cycle. Once changes have been accepted, they are entered into the in-memory diagram model and the log of recent changes. The next time the LLM is requested, the system will already be relying on the updated state of the model and the extended session history. This creates an adaptive, iterative loop between the user, the LLM and the editor, in which the model carefully continues the trajectory set by the user, suggesting locally plausible next steps in diagram construction, rather than attempting to "rewrite" the project.

Chapter 4

Implementation of a prototype

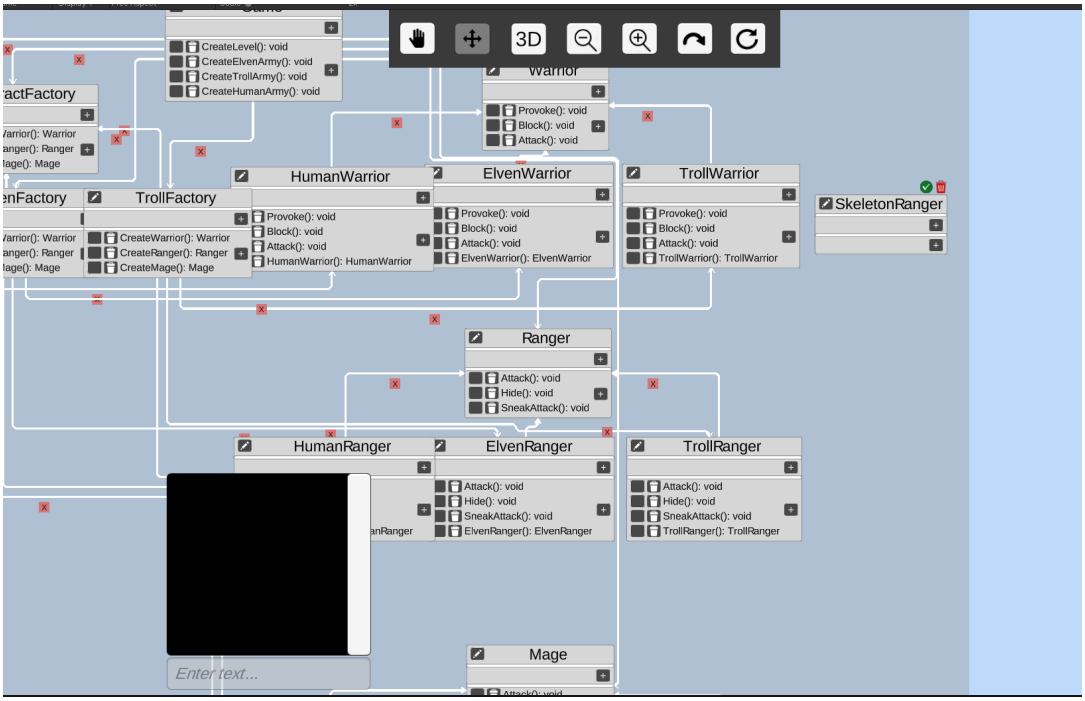


Figure 4.1: A new class called SkeletonRanger has been added. Model before enabling suggestions mode.

In Figure 4.1, as an experiment, an empty class called "SkeletonRanger" was added to the existing model and then the suggestions system was enabled. Several seconds later, Figure 4.2 shows a visualization of the system at work. Methods similar to those already in place in similar classes were automatically generated and added. This example demonstrates how such a system can save users time on routine tasks.

Users can then decide how to proceed with the suggested changes. They can either accept or reject each change individually, or accept or reject all of the suggested changes at once by clicking on the sidebar.

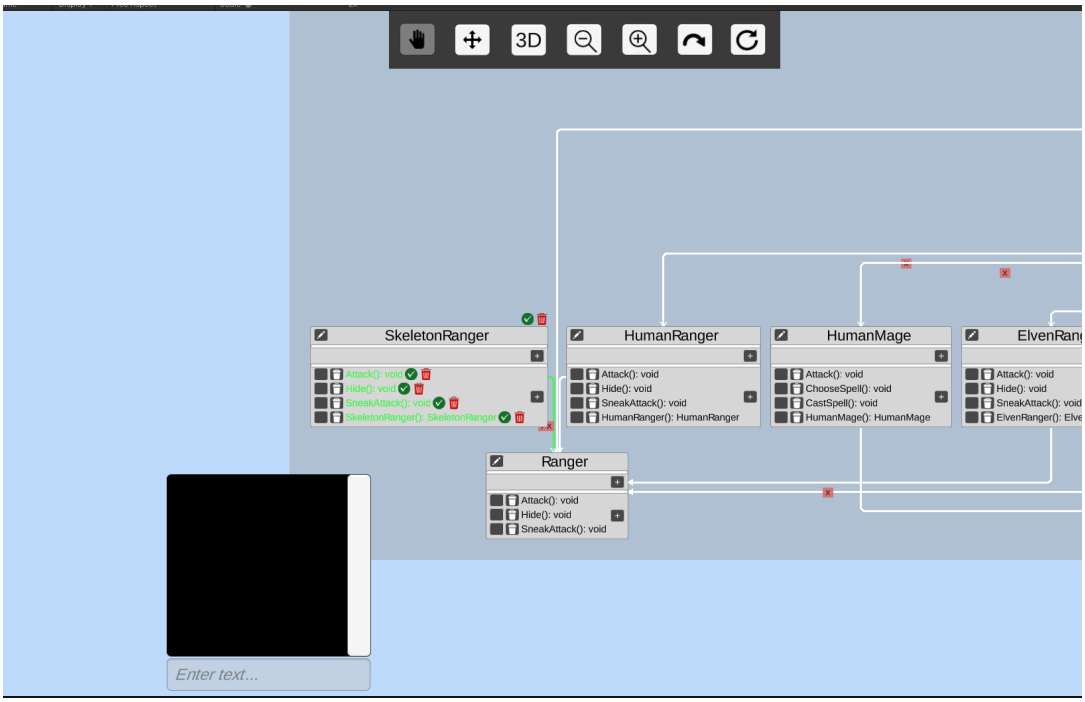


Figure 4.2: Suggestions were obtained and visualized here. Model after enabling suggestions mode.

Chapter 5

Evaluation

Chapter 6

Results

Conclusion

Bibliography

[1] Nacir Bouali, Marcus Gerhold, Tosif Ul Rehman, and Faizan Ahmed. Toward automated uml diagram assessment: Comparing llm-generated scores with teaching assistants. In *17th International Conference on Computer Supported Education, CSEDU 2025*, pages 158–169. Science and Technology Publications, Lda, 2025.

[2] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Software and Systems Modeling*, 22(3):781–793, 2023.

[3] Meriem Ben Chaaben, Lola Burgueño, and Houari Sahraoui. Towards using few-shot prompt learning for automating model completion. In *Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '23, page 7–12. IEEE Press, 2023.

[4] 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)*, pages 31–53. IEEE, 2023.

[5] Alessio Ferrari, Sallam Abualhaijal, and Chetan Arora. Model generation with llms: From requirements to uml sequence diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, pages 291–300. IEEE, 2024.

[6] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8), December 2024.

[7] Oksana Nikiforova, Konstantins Gusarovs, Ludmila Kozacenko, Dace Ahilcenoka, and Dainis Ungurs. An approach to compare uml class diagrams based on semantical features of their elements. In *The tenth international conference on software engineering advances*, pages 147–152, 2015.

[8] Lukas Radosky and Ivan Polasek. Executable multi-layered software models. In *Proceedings of the 1st International Workshop on Designing Software*, Designing '24, page 46–51, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Douglas C Schmidt, Jesse Spencer-Smith, Quchen Fu, and Jules White. Towards a catalog of prompt patterns to enhance the discipline of prompt engineering. *ACM SIGAda Ada Letters*, 43(2):43–51, 2024.

[10] Mina Shehata, Blaire Lepore, Hailey Cummings, and Esteban Parra. Creating uml class diagrams with general-purpose llms. In *2024 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 157–158, 2024.

[11] Christof Tinnes, Alisa Welter, and Sven Apel. Software model evolution with large language models: Experiments on simulated, public, and industrial datasets. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 649–649. IEEE Computer Society, 2025.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[13] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*, pages 71–108. Springer, 2024.

[14] Zhongchen Yuan, Li Yan, and Zongmin Ma. Structural similarity measure between uml class diagrams based on ucg. *Requirements Engineering*, 25(2):213–229, 2020.